Dr Milunka Damnjanović, red.prof,
OBJEKTNO ORIJENTISANE TEHNIKE PROJEKTOVANJA SISTEMA

6 Usavršavanje modela zahteva

Usavršavanje modela zahteva:

Posle identifikacije zahteva i formiranja dijagrama analize na osnovu dijagrama klasa korisničkih slučajeva, sledeći korak je *usavršavanje* (*prečišćavanje)* ovog modela.

Ovaj postupak se izvodi da bi se kreirali uslovi za razvijanje komponenata koje mogu biti ponovo korišćene u tekućem ili u budućim projektima.

Razvoj zasnovan na komponentama:

Nepotrebno i nepraktično je projektovati svaku komponentu iz nacrta.

Arhitekta podrazumeva postojanje cigle, prozora, vrata, crepa,..., a projektuje kuću zasnovanu na tim komponentama.

Softverske komponente – problemi:

Startovanje sa pozicije kao da se ništa ne zna može imati značajne nepogodnosti – može se desiti da se otkrivaju već otkrivene stvari.

Dobri profesionalci uvek nastoje da nauče što više iz svog iskustva i iskustva kolega.

Na primer, nepotrebno je projektovati neke industrijski standardne komponente kao .DLL fajlove

Sindrom NOR:

Pored postojanja mnogo bibliotečkih izvora, sindrom Nije Ovde Razvijeno (NOR) prevlada kod nekih projektanata.

Jedno rešenje je objektna orijentacija jer ona veoma olakšava korišćenje bibliotečkih komponenata.

5

Organizacija modela:

Teško je kreirati model koji će biti korišćen na različitim stepenima istog projekta, a još je teže projektovati strukturirani model koji će biti korišćen u budućim projektima.

Objektna orijentacija može ovde imati veliki doprinos u ponovnoj upotrebi kod analize zahteva.

6

Kako objektna orijentacija pomaže ponovnu upotrebu #1:

Razvoj standardnih komponenata.

Inkapsulirani unutrašnji detalji komponente tako da druge komponente kojima je potreban njen servis ne moraju da znaju kako će biti izveden servis.

Različiti delovi softvera mogu biti efektivno izolovani i redukuju probleme pri uzimanju raznih podsistema (razvijenih u različito vreme ili na različitim jezicima) koji treba da ostvare interakciju. Podsistemi razvijeni na ovaj način zovu se međusobno rasparenim.

7

Kako objektna orijentacija pomaže ponovnu upotrebu #2:

Efekat se može proširiti na kompleksne podsisteme primenom istog principa inkapsulacije u veće grupe ili objekte. Zahtevi:

- **Komponenta mora da ima jasno definisane opšte osobine**
- **Komponenta mora imati jednostavan i dobro definisan spoljašnji interfejs.**

Otuda je objektna orijentacija koja podrazumeva inkapsulaciju pogodna za ponovnu upotrebu – potrebno je da objekat koji zahteva servis samo zna protokol za poruke i koji objekat može da obezbedi servis.

8

Ponovna upotreba zahteva:

Ponovna upotreba zahteva je, u stvari, ponovna upotreba modela zahteva.

Obično se ponovo mogu koristiti samo delovi modela.

9

Generalizacija:

Primer apstrakcije generalizacijom: projekat cigle je takav da se može koristiti mnogo puta.

Projektanti OO projekata imaju prednost pristupa – nasleđivanje. Tako se projektuju i prave veći delovi nove softverske komponente unapred ostavljajući da se samo posebni detalji dorade kasnije. Ovo je omogućeno osobinom hijerarhije klasa da su zajedničke osobine potklasa sadržane u superklasi i trenutno su dostupne potklasama.

10

Kompozicija:

Kompozicija je tip apstrakcije koji inkapsulira grupe klasa koje zajedno mogu da budu ponovo korišćene.

Primer: upotreba prozora kao celine (iako ima više delova) u kući koja se gradi.

11

Kompozicija u UML-u:

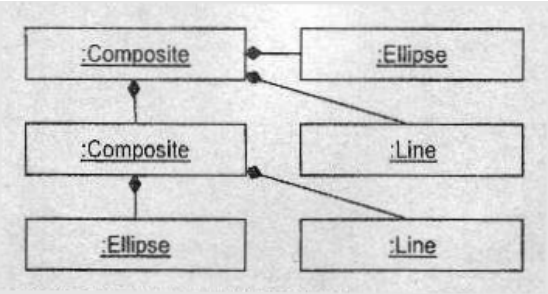
Kompozicija ili kompozitna agregacija zasnovana je na agregaciji kod mnogo OO jezika. Relacija između agregacije i kompozicije:

Kompozitna agregacija je jaka forma agregacije koja zahteva da uzorak bude uključen u najviše jedan kompozit istovremeno i da kompozitni objekat ima samoodgovornost za dispoziciju njegovih delova. Ovo znači da je kompozitni objekat odgovoran za kreiranje destrukcije njegovih delova.

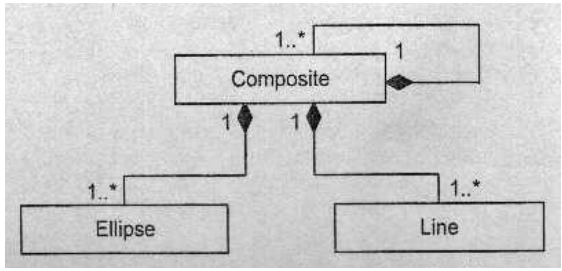
Primer: paket za crtanje na računaru.

12

Programski paket za crtanje:



Kompozicija korišćena u dijagramu klasa za predstavljanje kompozitnih objekata:



Notacija:

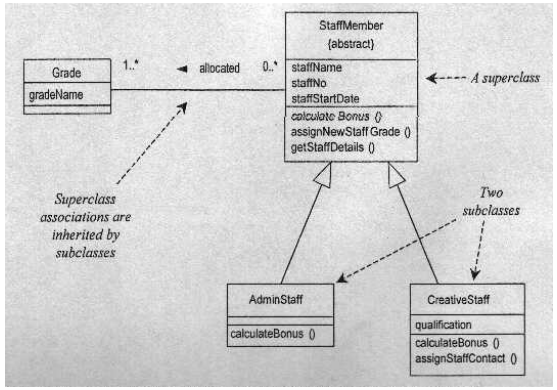
Notacija je slična prostim asocijacijama, ali sa rombom na kraju celine.

Romb je pun kad indicira kompoziciju, a nepopunjen kad označava agregaciju.

Modelovanje generalizacije:

Sa aspekta potklase, nasleđena svojstva su osobine koje pripadaju potklasama ali su pomerene na viši nivo apstrakcije. Ovo su zajedničke osobine koje određuju hijerarhiju generalizacije. Generalizacija pomaže analitičaru da ne mora ove osobine da prikazuje u svim potklasama u kojima se pojavljuju. Nekoliko potklasa mogu da imaju zajedničke atribute, operacije i asocijacije koje su prikazane samo jednom. Svaka potklasa mora imati bar jednu osobinu koja je razlikuje od drugih klasa u hijerarhiji.

Primer hijerarhije generalizacije:



U UML-u ova predstava se zove stil odvojenog cilja.

Apstraktne i konkretne klase:

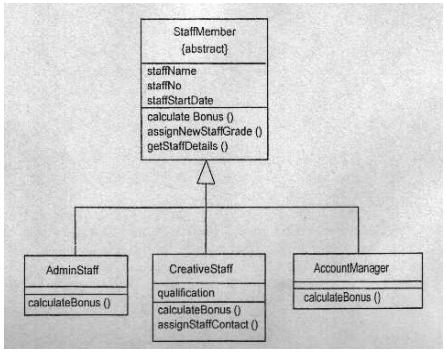
Svojstvo *apstraktosti* mogu da dobiju samo superklase u hijerarhiji generalizacije. Sve ostale klase imaju najmanje jedan uzorak i za njih se kaže da su *konkretne* ili *uzorkovane*.

Razlog za postojanje superklasa je što postoje na višem nivou od njihovih potklasa. Ova generalizacija im dopušta prilagođavanje drugim sistemima.

Uvek dobijaju opis *apstrakna klasa*.

Korist od generalizacije:

Lako je dodati novu potklasu:



U UML-u ova predstava se zove sil zajedničkog cilja.

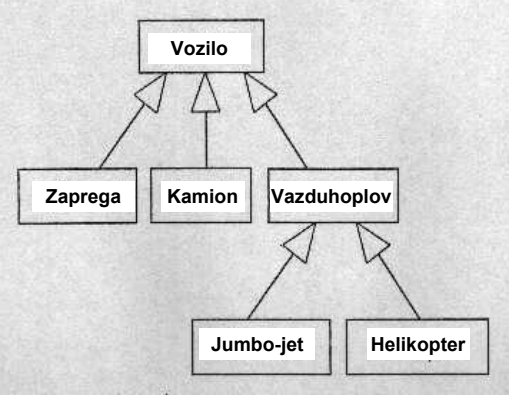
Identifikovanje generalizacije (pristup odozgo-na dole):

Lako je naći generalizaciju ako su klase već definisane. Ako se neka asocijacija može opisati sa 'je vrsta od', onda se obično ona može modelovati generalizacijom.

Mogu se pronaći višestruki nivoi generalizacije – Superklasa u jednoj relaciji može biti potklasa u nekoj drugoj.

U praksi, više od četiri ili pet nivoa generalizacije je previše.

Primer: hijerarhija odozgo na dole:



21

Dodavanje generalizacije (pristup odozdo-nagore:

Traženje sličnosti među klasama u modelu i razmatranje da li model može da 'naraste', tj. da se formira superklasa koja će da apstrahuje sličnosti.

Cilj ovog postupka je da se podigne nivo apstrakcije modela.

Mora se poštovati princip: Svaka generalizacija mora da prođe sve testove prethodno opisane pa i frazu '*je vrsta od*'.

22

Kada ne treba koristiti generalizaciju:

• Ako su razlike između potencijalnih potklasa suviše velike, generisanje nove superklase može da izazove konfuziju.

• Prosuđivanje na osnovu iskustva!

• Generalizacija se modeluje da bi se omogućilo uvođenje novih potklasa u budućnosti.

23

Višestruko nasleđivanje:

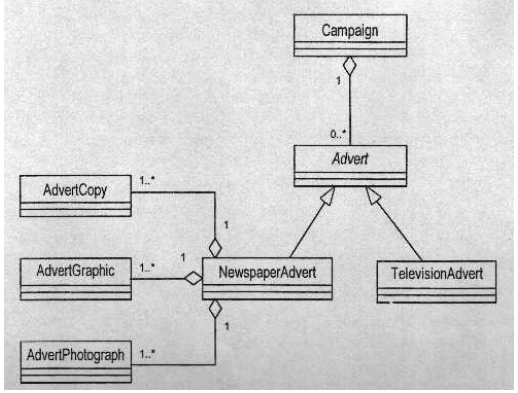
Moguće je, a često i pogodno, da klasa bude istovremeno potklasa za više od jedne superklase.

Primer: Šolja je posuda iz koje se pije, lep ukras, potencijalni izvor mikroba, ...

Kod modelovanja, posebno u toku projektovanja, može biti korisno definisati klase koje nasleđuju svojstva od više od jedne superklase. U tom slučaju, nasleđuju se sva svojstva od svake superklase.

24

Kombinovanje generalizacije sa kompozicijom ili agregacijom:



25

Analitički paketi:

Potrebna je i veština i prosuđivanje od strane analitičara da bi kreirao model koji će biti otporan na promene zahteva.

UML paket je alat za upravljanje kompleksnošću modela, kao i koristan način identifikovanja podsistema koji postoje kao komponente.

Paketi su sredstva pomoću kojih projektant može da izdvoji klase ili strukture koje imaju potencijalnu upotrebu u širem kontekstu nego što je sam projekat.

Vrlo je važno da se čuva trag zavisnosti između različitih klasa i paketa.

26

Software development patterns #1:

Svaki pattern opisuje problem koji se ponavlja i onda opisuje srž rešenja tog problema na takav način da se to rešenje može koristiti još milion puta, a da se nikad ne radi na isti način dvaput.

Pattern predstavlja rešenje koje se može primeniti na različite načine kod različitih problema.

27

Software development patterns #2:

Gabriel (1996):

Svaki pattern je trodelno pravilo koje izražava relaciju između nekog konteksta, nekog sistema sila koje se ponavljaju u kontekstu, i neke konfiguracije softvera koja dopušta ovim silama da se razreše.

Ova definicija se fokusira na tri elementa:

- ***kontekst*** **(skup okolnosti ili preduslova),**
- ***sile*** **(pojmovi koje treba dodeliti), i**
- ***konfiguracija softvera*** **koja razrešava sile.**

28

Software development patterns #3:

Strategija – plan akcije da se postigne cilj.

Pattern – šablon.

29

Software development patterns #4:

Coplien (1996):

Kritični aspekti pattern-a su:

- **On rešava problem.**
- **To je dokazani koncept.**
- **Rešenje nije očigledno.**
- **On opisuje odnos.**
- **Pattern ima značajnu ljudsku komponentu**

30

Kategorije pattern-a #1:

***Arhitekturni* pattern opisuje strukturu i odnose najvećih komponenata u softverskom sistemu. Arhitekturni pattern identifikuje podsisteme, njihove odgovornosti i njihove međuodnose.**

***Projektni* pattern identifikuje međuodnose grupa softverskih komponenata opisujući njihove odgovornosti, saradnje i strukturne odnose.**

31

Kategorije pattern-a #2:

***Idiom* opisuje kako implementirati pojedinačne aspekte softverskog sistema u datom programskom jeziku.**

***Analitički* pattern definiše se radi opisa grupa koncepata koji predstavljaju opšte konstrukcije u domenu modelovanja. Mogu se primeniti na jedan ili više domena.**

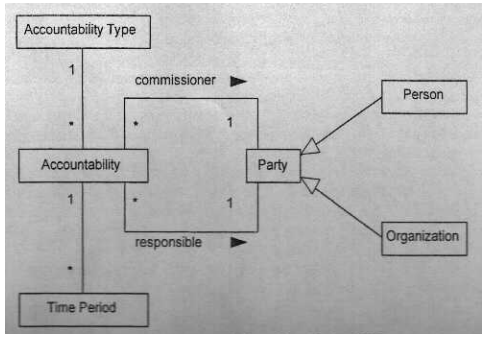
32

Analitički pattern:

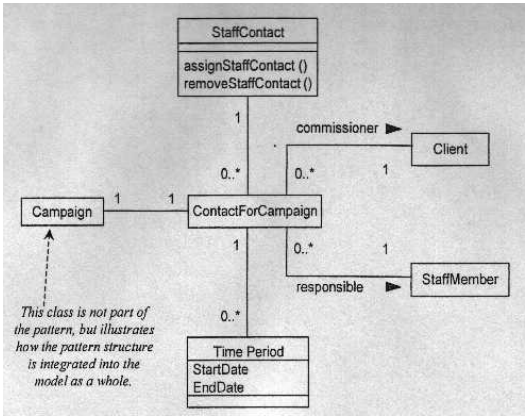
To je struktura klasa i asocijacija koje se ponavljaju u različitim situacijama koje treba modelovati. Svaki pattern može se upotrebiti za opšte razumevanje modelovanja nekog skupa zahteva i model ne treba izmišljati svaki put od početka kad se pojavi specifična situacija.

S obzirom da pattern može da sadrži celu strukturu klasa, apstrakcija se dešava na višem nivou nego što je normalno moguće koristeći samu generalizaciju.

Analitički pattern :



Primenjen prethodni analitički pattern:



Kratak pregled:

Osnovni cilj modela prečišćavanja zahteva je maksimiziranje ponovne upotrebe.

Nove reupotrebljive komponente mogu se razviti i za upotrebu u drugim projektima.

Mehanizmi apstrakcije OO projektovanja pogodni za projektovanje reupotrebljivih komponenata.