DIZAJN I IMPLEMENTACIJA SOFTVERA

Ian Sommerville - SOFTWARE ENGINEERING – Chapter 7

Uvod

Dizajn softvera podrazumeva proces prepoznavanja raznih komponenti sistema i odnosa između njih, koji su bazirani na informacijama koje smo dobili od klijenata.

Implementacija podrazumeva realizaciju tog dizajna u vidu programa.

Cilj poglavlja:

1. Da se pokaže kako se procesi modeliranja sistema i dizajna arhitekture sprovode u praksi i razvoju OO dizajna softvera

2. Da nas uvede u neke implementacione probleme kao sto su:
   - poslovna iskorišćenost programa (software reuse)
   - upravljanje konfiguracijom (configuration menagment)
   - host – target development

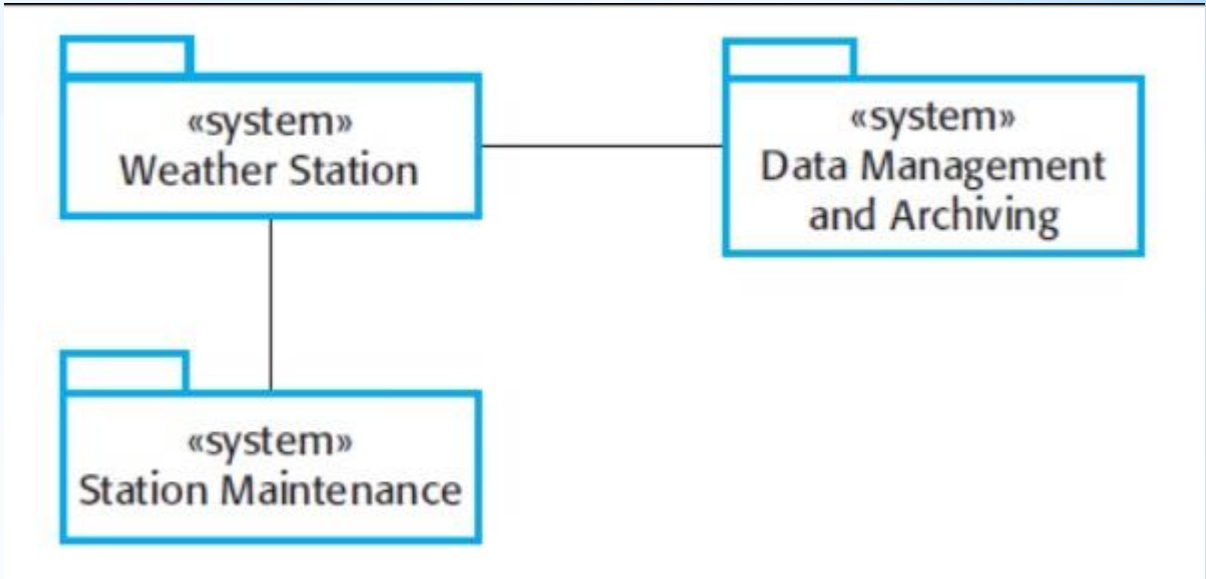
3. Osnova o open source programiranju

OO DIZAJN KORIŠĆENJEM UML-A

Proces OO dizajna podrazumeva dizajniranje klasa i veza medju tim klasama. U procesu razvijanja, od konceptualnog pa do detaljnog, objektno orjentisanog dizajna, postoji nekoliko stvari koje treba da uradimo. To su:

1. Razumevanje i definisanje konteksta našeg projekta i njegove spoljašnje interakcije sa sistemom.
2. Dizajniranje arhitekture sistema.
3. Identifikovanje glavnih objekata u sistemu
4. Modeliranje dizajna
5. Interfejsi

PRIMER – METEOROLOŠKE STANICE



Ove stanice su deo većeg sistema (slika). Ove komponente imaju sledeće značenje:

• The weather station system – služi da prikuplja podatke, radi neku inicijalnu obradu tih podataka i prosleđuje ih za „Data managment and archiving system“ .

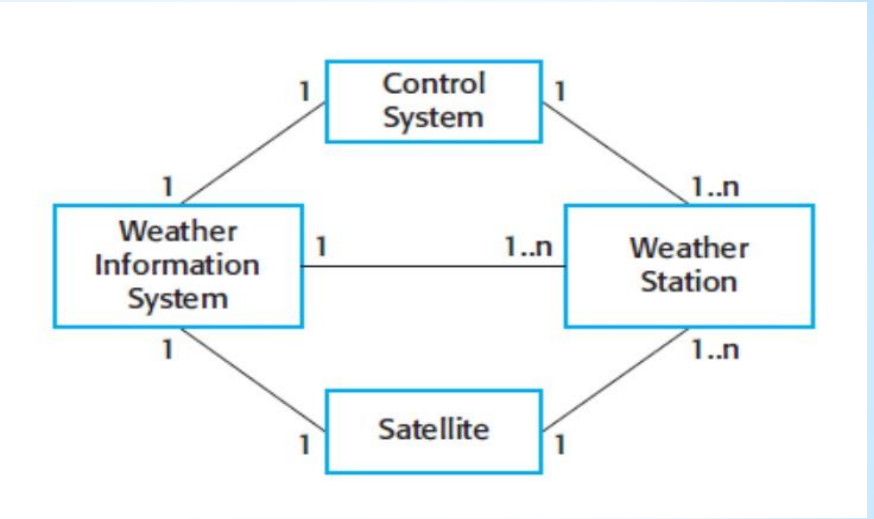
• The data managment and archiving system – prikuplja podatke iz pojedinačnih stanica, obrađuje ih i vrši potrebne analize, skladišti te podatke u formi koja može biti zatražena od strane drugih sistema, kao što su sistemi za vremensku prognozu.

• The station maintenance system – Ovaj sistem može da komunicira putem satelita sa svim meteorološkim stanicama koje smo postavili u gore opisanim područjima i da na taj način vrši nadzor, da proverava da li sve stanice rade i da prikuplja izveštaje o greškama ukoliko ih ima. Takodje može da vrši update zastarelog softvera koji se nalazi na tim stanicama. U slučaju problema, ovaj sistem ima mogućnost da putem remote pristupa popravi grešku na stanici.

1. Razumevanje i definisanje konteksta našeg projekta i njegove spoljašnje interakcije sa sistemom.

- Dijagram konteksta sistema i dijagram interakcije predstavljaju komplementarne poglede na veze izmedju sistema i njegovog okruženja. Dijagram konteksta sistema je strukturni dijagram i prikazuje nam druge sisteme iz pogleda sistema koji razvijamo.

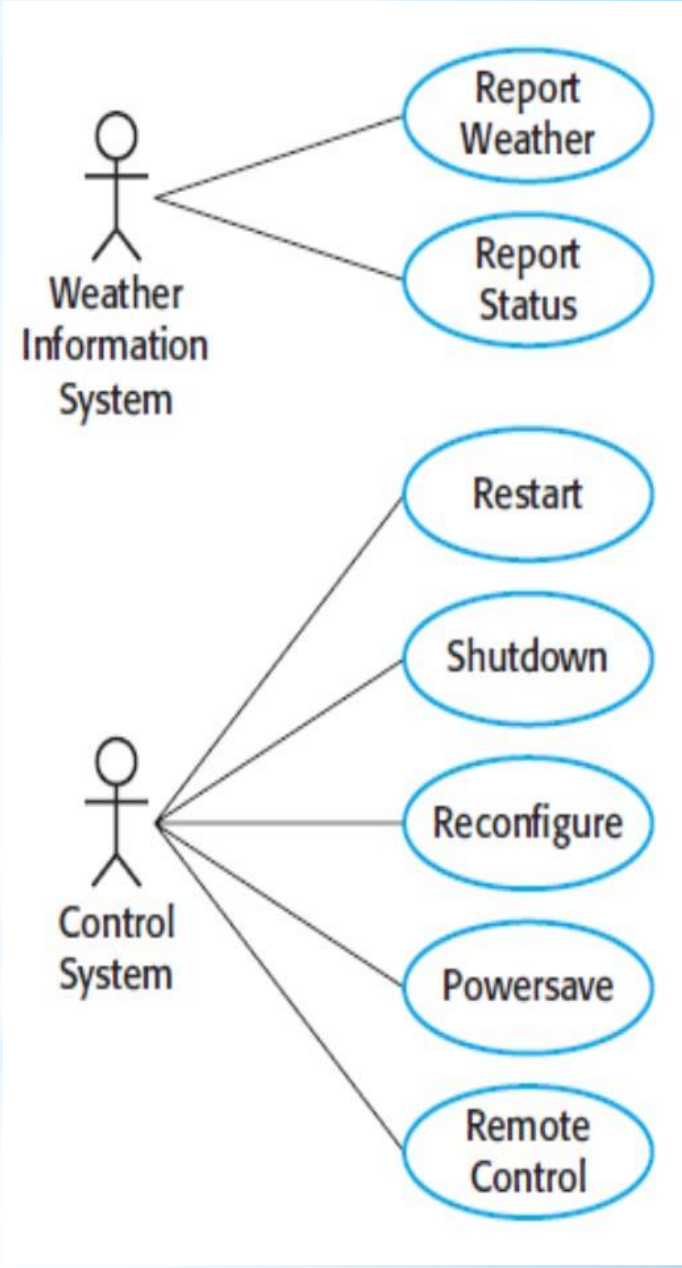
- Dijagram konteksta sadrži asocijacije.One jednostavno pokazuju veze izmedju entiteta.



Jedan od načina da se prikaže kako sistem iteraguje sa svojim okruženjem jeste dijagram slučajeva upotrebe.

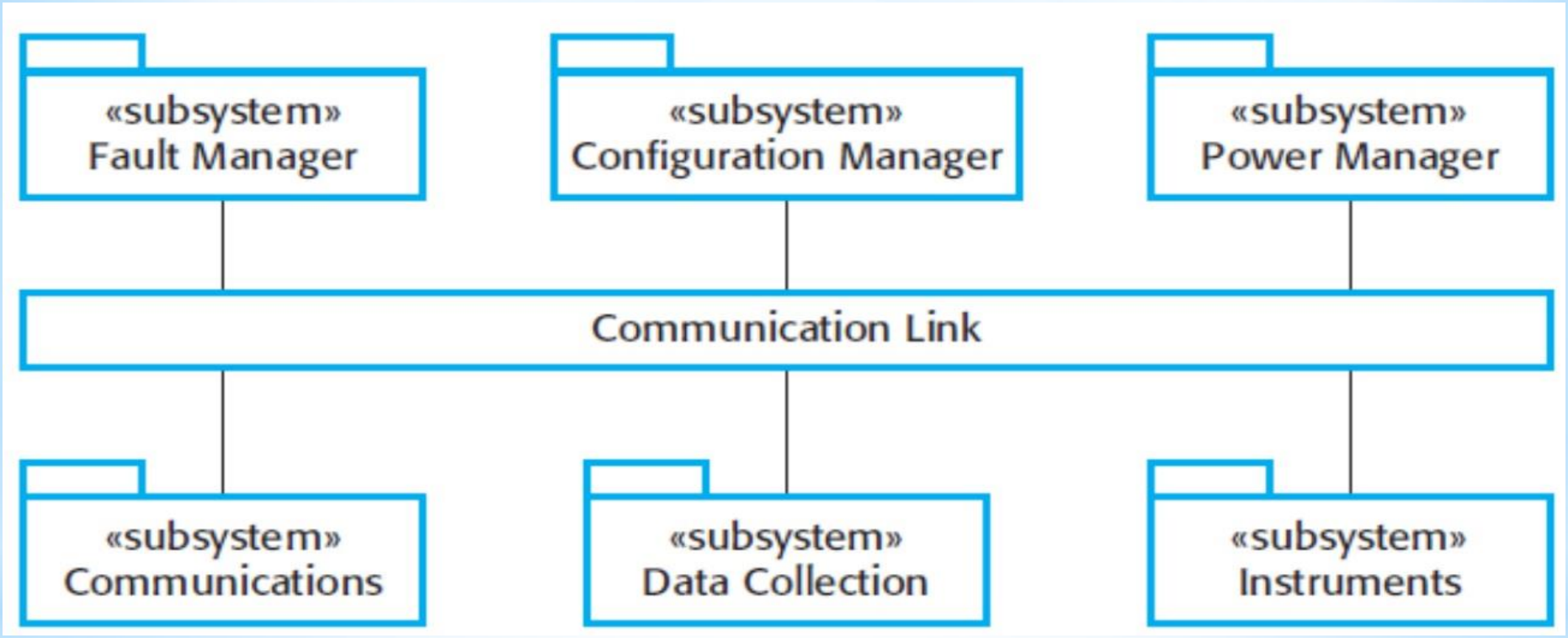
Slučaj upotrebe (Use case) je specifikacija skupa akcija koje vrši sistem, koje proizvode vidljiv rezultat koji je, po pravilu, od vrednosti za jednog ili više učesnika u
Sistemu

Slučajevi upotrebe(našeg primera):
Izveštaj o vremenu – stanica šalje podatke o vremenu sistemu za informacije o vremenu
Izveštaj o statusu – stanica šalje informacije o svom statusu sistemu za informacije o vremenu
Restartovanje – ako se stanica ugasi, kontrolni sistem može da je restartuje
Gašenje – kontrolni sistem može da ugasi stanicu
Konfigurisanje – kontrolni sistem može da vrši konfigurisanje proizvoljne stanice
Powersave – kontrolni sistem može da stavi stanicu u „štedljivi“ mod
Pristup sa udaljenog računara (remote control) – kontrolni sistem može da pristupi stanici i da vrši bilo kakve komande na bilo kom njenom podsistemu



2. Dizajniranje arhitekture sistema.

Za naš primer, arhitektura bi mogla da izgleda ovako:



3. Identifikovanje glavnih objekata u sistemu

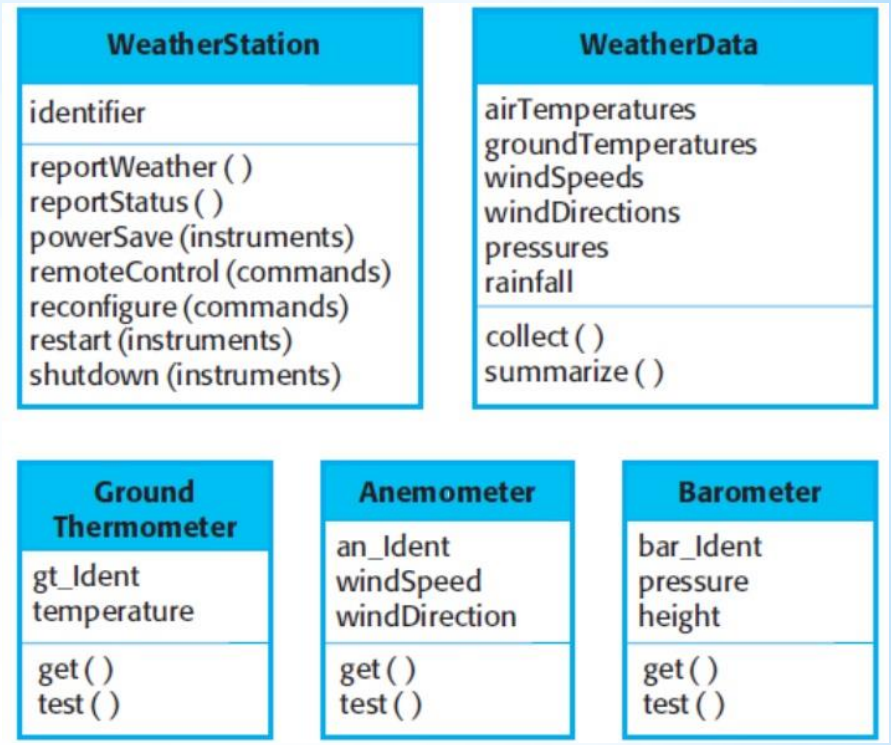
Do sada smo mogli da steknemo utisak o nekim glavnim objektima koji će nam se pojaviti u sistemu koji dizajniramo. Definicije slučajeva upotrebe mogu dosta da nam pomognu pri definisanju objekta i operacija koje oni vrše.

Na primer, iz „Report weather“ slučaja upotrebe, mi možemo da zaključimo da ćemo sigurno imati objekte koji predstavlja instrumente koji vrše merenja, kao i objekat koji će služiti da skladišti podatke o tim merenjima.

Takodje, obično postoji i jedan objekat koji na neki način predstavlja sistem koji dizajniramo i koji u sebi ima definisane sve one operacije koje su opisane u slučajevima upotrebe.

Sa ovim početnim zapažanjima, možemo da počnemo identifikovanje osnovnih klasa našeg sistema.

- „WaetherStation“ klasa omogućava osnovni interfejs meteorološke stanice prema njenom okruženju. Operacije odgovaraju vezama našeg sistema sa ostalim delovima sistema koje smo opisali u dijagramima slučajeva upotrebe.
- „WeatherData“ klasa je odgovorna za izveštaje o vremenu. Ona šalje sumirane
podatke od svake meteorološke stanice pa do sistema za informacije o vremenu.
- „Ground Thermometer“, „Anemometer“ i „Barometer“ su klase koje su direktno vezane za instrumente kojima se vrši merenje. Odgovaraju hardverskim entitetima sistema i operacije koje imaju su direktno povezane sa kontrolisanjem tog hardvera. Objekti ovih klasa automatski prikupljaju podatke i na zahtev ih šalju objektu klase „WeatherData“.

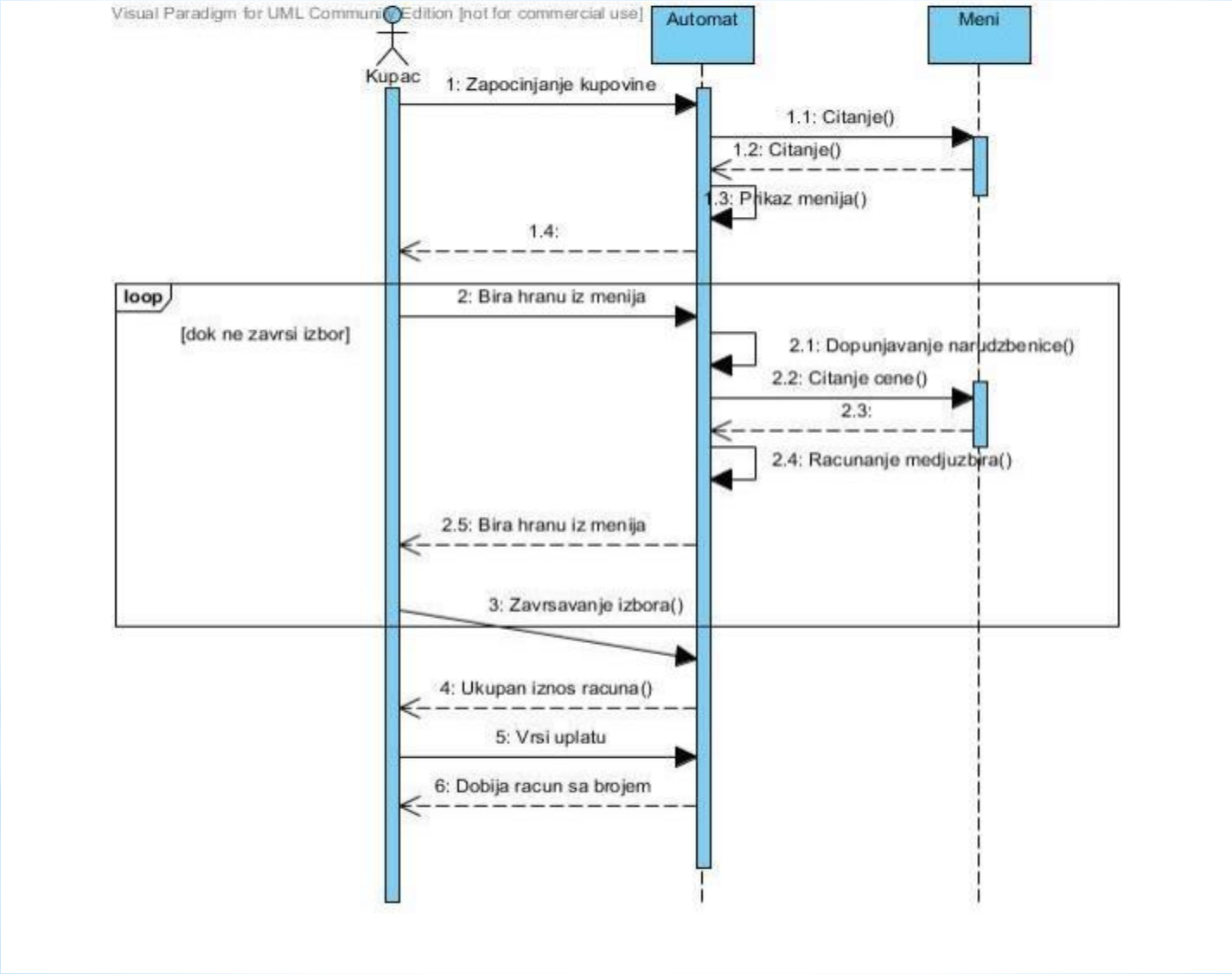


4. Modeliranje dizajna (UML)

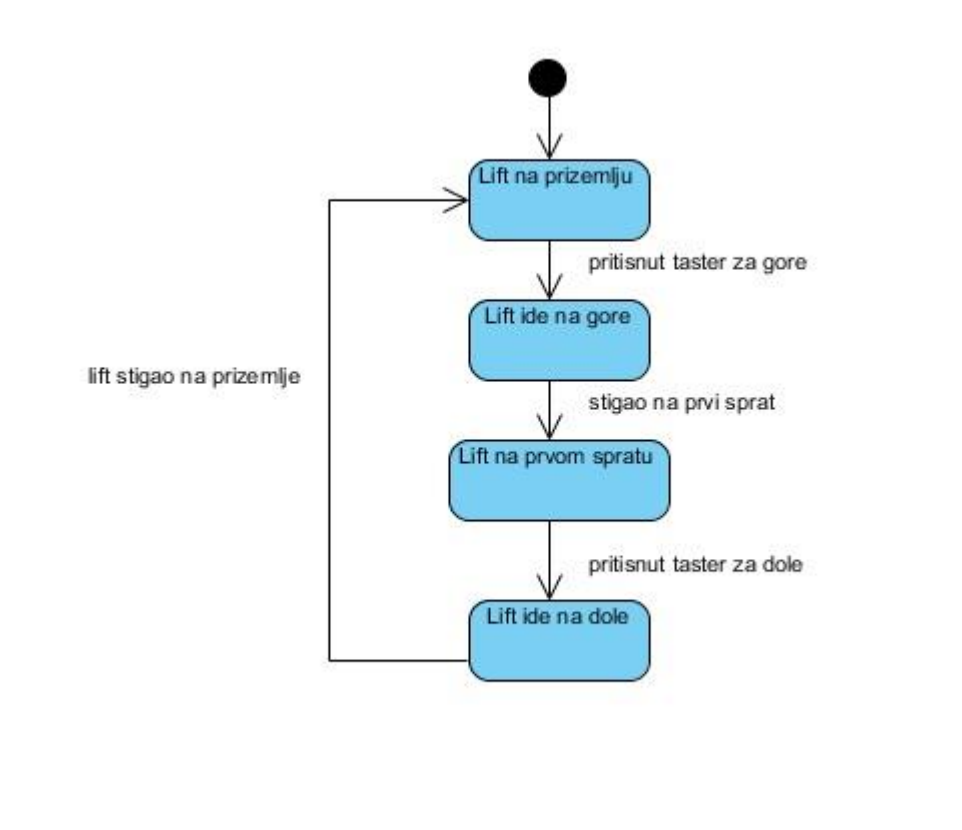
UML (Unified Modeling Language) je grafički jezik (ne programski jezik) za vizueliziranje, specificiranje, konstruisanje i dokumentovanje sistema programske podrške.

• UML podržava 13 različitih vrsta dijagrama, ali mi skoro nikad nećemo koristiti sve. Obično kada pravimo te modele, odnosno dijagrame, treba da napravimo dve vrste dijagrama. Jedan strukturni, koji će nam govoriti o samoj strukturi našeg sistema. I drugi „dinamički“, koji će nam opisivati kako se to naš sistem ponaša.

Primer dijagrama sekvenci

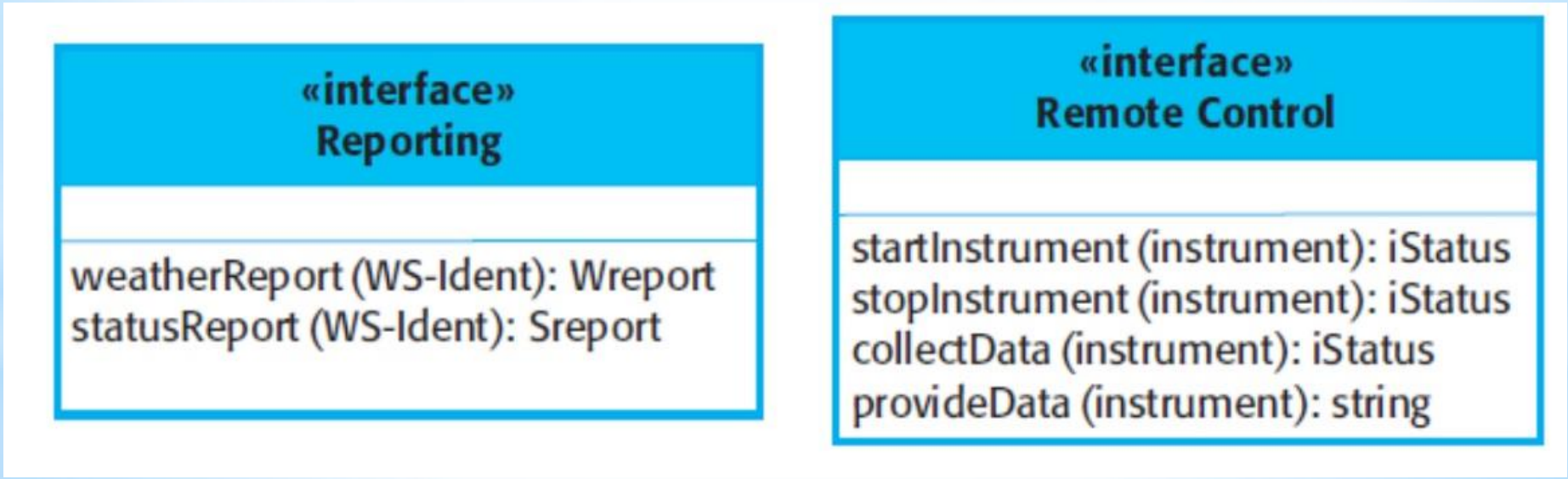


Primer dijagrama stanja



5. Interfejsi

Bitan korak koji je deo procesa dizajniranja jeste odredjivanje interfejsa koji su vezani za komponente koje smo dizajnirali. Interfejsi se mogu predstaviti dijagramom klasa UML-a, samo što nemamo deo sa atributima, već samo operacije koje služe da se pristupi podacima, odnosno da se ti podaci menjaju.



* Uzorci za projektovanje(design patterns)

Primenu istog koncepta rešavanja problema na različite probleme nazivamo UZORKOM ZA PROJEKTOVANJE.

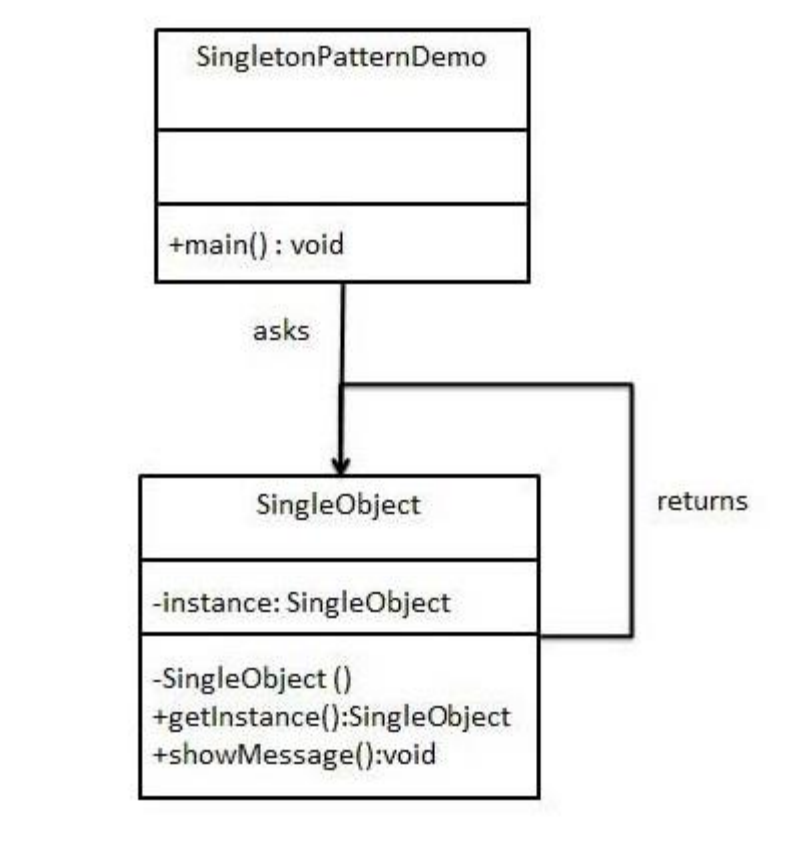
„Svaki uzorak opisuje problem koji se stalno ponavlja u našem okruženju i zatim opisuje suštinu rešenja problema tako da se to rešenje može upotrebiti milion puta, a da se dva puta ne ponovi na isti način.“

Christipher Alexander

Kada se razvijamo neki sistem mi nikako ne možemo unapred da znamo koji uzorak za projektovanje će nam biti potreban, niti da li će nam uopšte biti potreban. Prvo se vrši neko početno dizajniranje, pa se dobro razmotri celokupan problem, pa tek onda prepoznamo kakav uzorak za projektovanje treba da iskoristimo.

- Uzorci za projektovanje su mnogo dobra ideja i vrlo su korisni, ali da bi se pravilno koristili, potrebno je dosta iskustva. Cilj je prepoznati situacije gde ćemo te uzorke pravilno iskoristiti.
- Neki poznati uzorci: Unikat, Posetilac, Apstraktna fabrika, Dekorater, Posmatrač, …

Singleton Pattern



```java
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

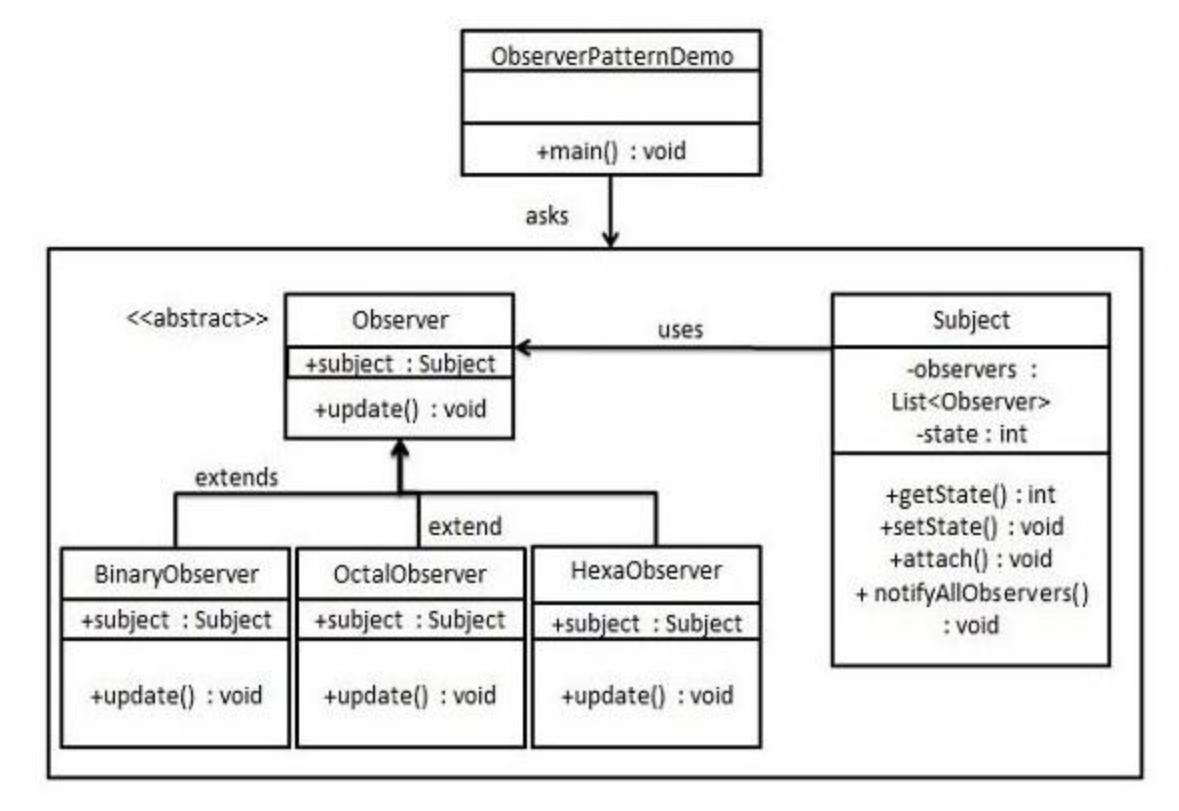
```java
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Observer Pattern



* Implementacioni problemi

Razvoj softvera uključuje sve aktivnosti, počev od inicijalnih zahteva sistema, pa sve do održavanja i upravljanja razvijenog sistema. Bitan deo ovog procesa jeste implementacija sistema, gde mi pravimo izvršnu verziju programa.

E sad, mi ovde pretpostavljamo da znamo da kodiramo, tako da nećemo pričati o samom kodiranju u nekom specifičnom programskom jeziku, nego o nekim aspektima same impementacije koji su jako važni za razvoj softvera.
Oni su:
1. „Reuse“ – ponovna upotrebljivost
2. „Configuration managment“ – vođenje računa o konfiguracijama.
3. „Host-target development“ – razvijeni softver se obično ne izvršava na istom računaru gde se razvijao. Mnogo češće se dešava da mi razvijamo na jedno računaru (host) a da ga izvršavamo na nekom sasvim drugom mestu (target system). Nekada se dešava da su host i target jedna ista stvar, ali baš retko.

SOFTWARE REUSE

Od 1960-ih do 1990-ih većina softvera se razvijala od samog početka, pisajući kod u programskom jeziku visokog nivoa i jedini značajni „reuse“ jeste bio ponovna upotrebljivost funkcija i objekata iz biblioteka tih jezika.
Sve u svemu, danas je ovakav pristup gotovo nemoguć zbog cene takvog pristupa. Reuse se koristi za razne biznis sisteme, naučni softver, embedded programiranje,…

Ponovnom upotrebljivošću softvera, omogućen nam je brži razvoj, sa manje rizika i sa manjom cenom. Prednost ovako napravljenog softvera je u tome što je već testiran na nekim drugim aplikacijama, pa je samim tim potencijalno pouzdaniji od novog softvera. Ali ipak, i ovde moramo da platimo neku cenu:
1**. Vreme potrošeno** na pretragu da nađemo softver koji ćemo da upotrebimo i na proveravanje da li je to zaista ono što nam treba. Moramo da testiramo taj softver da vidimo da li lepo radi u našem okruženju, pogotovo ako je naše okruženje drugačije od onog okruženja u kom je softver razvijan.
2. Ponekad se vrši **kupovina softvera** koji se može ponovno upotrebiti, što takođe ima svoju cenu. Za velike sisteme ova cena može biti jako velika.
3. **Cena adaptacije i konfiguracije softvera** koji planiramo da iskoristimo na našem sistemu
4.Cena integracije više elemenata, koje planiramo da iskoristimo, u jedan (ako koristimo softver sa više različitih izvora) sa našim kodom. U slučaju kada koristimo kod sa više različitih mesta, to može da bude prilično teško i skupo jer može da se desi konfliktna situacija da nešto iz jednog sistema isključuje nešto drugo iz drugog sistema…
Kako da ponovno upotrebimo nešto što već postoji treba da bude prva stvar o kojoj ćemo promisliti pre nego što počnemo da razvijamo naš sistem.

UPRAVLJANJE KONFIGURACIJAMA

Upravljanje konfiguracijama je ime koje podrazumeva generalno upravljanje projektom i izmenama na projektu. Uloga upravljanja konfiguracijama je da podrži proces integracije sistema, tako da svi programeri mogu da pristupe kodu i dokumentaciji na pravi način, da mogu da vide kakve promene su pravljene, da kompajliraju sistem itd.

Postoje 3 fundamentalne aktivnosti samog procesa upravljanja konfiguracijama. To su:
1. **Upravljanje verzijama** – podržano čuvanje različitih verzija komponenti sistema. Sistem za upravljanje verzijama uključuje alate za koordinaciju programera tako da više njih istovremeno može da radi na jednoj komponenti. Ne dozvoljavaju jednom programeru da prebriše nešto što je radio neki drugi programer.
2. **Integracija sistema** – Služi da pomogne programerima da definišu koja verzija kojih komponenata treba da se koristi za koju verziju sistema. Ovaj opis se onda koristi pri automatskom bildovanju sistema kompajliranjem i linkovanjem potrebnih komponenti.
3. **Praćenje problema** – Omogućava korisnicima da prijave bug-ove i druge probleme. Omogućava programerima da vide ko od njih radi na kom problemu i kada je koji problem popravljen.

HOST TARGET DEVELOPING

• Većina softvera se bazira na host-target modelu. Softver se razvija na jednom računaru (the host), a pokreće na drugoj mašini (target). Generalno, kada pričamo o ovome, pričamo o platformi na kojoj se razvija i o platformi na kojoj se pokreće. Platforma je više od samog hardvera. Ona uključuje operativni sistem, kao i drugi softver, kao što je sistem za upravljanje bazama podataka ili razvojno okruženje i slično.
• Ponekad, ove dve platforme su iste, što bi omogućilo i razvoj i testiranje na istoj mašini. Mnogo češće su različite, pa je potrebno vršiti testiranje na mašini na kojoj će se program izvršavati. Simulatori se često koriste kod razvoja embedded sistema. Mi možemo da simuliramo hardverske uređaje kao što su senzori i slično. Simulatori značano
ubrzavaju razvojni proces za embedded sisteme tako što svaki programer može da ima svoju platformu za izvršavanje, bez potrebe za skidanjem nekog softvera za hardver. Ali sa druge strane, simulatori su skupi.
• Platforma za razvoj softvera treba da obezbedi neki skup alata, kako bi podržala proces razvoja softvera. Ovo može da uključuje:
1. Integrisani kompajler i editor koji omogućava pisanje, menjanje i kompajliranje koda,
2. Sistem za debagovanje,
3. Alate za grafičko editovanje, npr za UML dijagrame,
4. Alate za testiranje, kao što je Junit koji mogu automatski da pokrenu skup testova…
• Pored ovih standardnih alata, tu mogu da se nadju i neki specifični alati kao što su razni analizatori i slično, ali o
tome će biti više reči u poglavlju 15. Uobičajeno je da razvojno okruženje uključuje i šerovan server na kom se
nalazi upravljač konfiguracijama.

* Open source razvoj

LICENCIRANJE

• Iako je osnovni princip open source razvoja taj da izvorni kod treba da bude dostupan svima, to ne znači da svako sa njim može da radi šta želi. U tom smislu onaj ko je originalno razvijao kod predstavlja vlasnika tog koda (u pravnom smislu). Vlasnik koda može postavljati različite uslove vezane za taj softver kroz neke obavezujuće klauze u open source licenci.
• Najčešće licence su izvedene iz jednog od sledećih modela (a jako često se i koriste sami modeli):
1. **GPL(General Public Licence)** – Ako koristimo softver pod ovom licencom, onda i naš softver mora da bude open source i to pod ovom licencom
2. **LGPL (Lesser General Public Licence)** – Ovo je varijanta GPL licence gde možete pisati softverske komponente vezane za open source izvorni kod, bez toga da moramo objaviti izvorni kod ovih komponenti. Ipak, ako promenimo open source kod koji smo koristili, onda moramo objaviti sve kao open source.
3. **BSD (Berkly Standard Distribution) Licence** – Izvorni kod pod ovom licencom može da se koristi proizvoljno, ali mora da se navede originalni kreator izvornog koda (Za kod koji nije naš moramo navesti čiji je.)

OPEN SOURCE

• Bayersdorfer (2007) je predložio kompanijama koje imaju projekte koji koriste open source komponente, sledeće:

1. Uspostavljanje sistema za čuvanje informacija o open source komponentama koje su korišćene. Potrebno je čuvati kopiju licence za svaku komponentu kojom je ta komponenta bila licencirana u vreme korišćenja. Licence se mogu promeniti, tako da moramo znati koje uslove smo prihvatili.
2. Treba biti svestan različitih tipova licenci i kako je određena komponenta licencirana pre nego što je iskoristimo. Možete odlučiti da neku komponentu koristite u jednom projektu, a u drugom da je ne koristite
3. Treba biti svestan evolucije open source komponenti koje koristimo da bi znali kako se one mogu razviti odnosno promeniti u budućnosti
4. Treba obrazovati ljude o open source-u. Nije dovoljno imati spremne procedure koje će obezbediti saglasnost sa licencama. Potrebno je i obrazovati razvojni tim o open source-u i open source licenciranju.
5. Potrebno je imati revizorske sisteme. Programeri usled kratkih rokova mogu prekršiti neke od licenci. Ako je moguće treba imati softver koji ovo može da primeti i zaustavi.
6. Učestvujte u open source zajednici. Ako se oslanjate na open source proizvode, trebali biste da učestvujete u zajednici i da podržavate njihov razvoj.

KRAJ!

HVALA NA PAŽNJI!