Objektno orjentirano programiranje

Predavanje 7

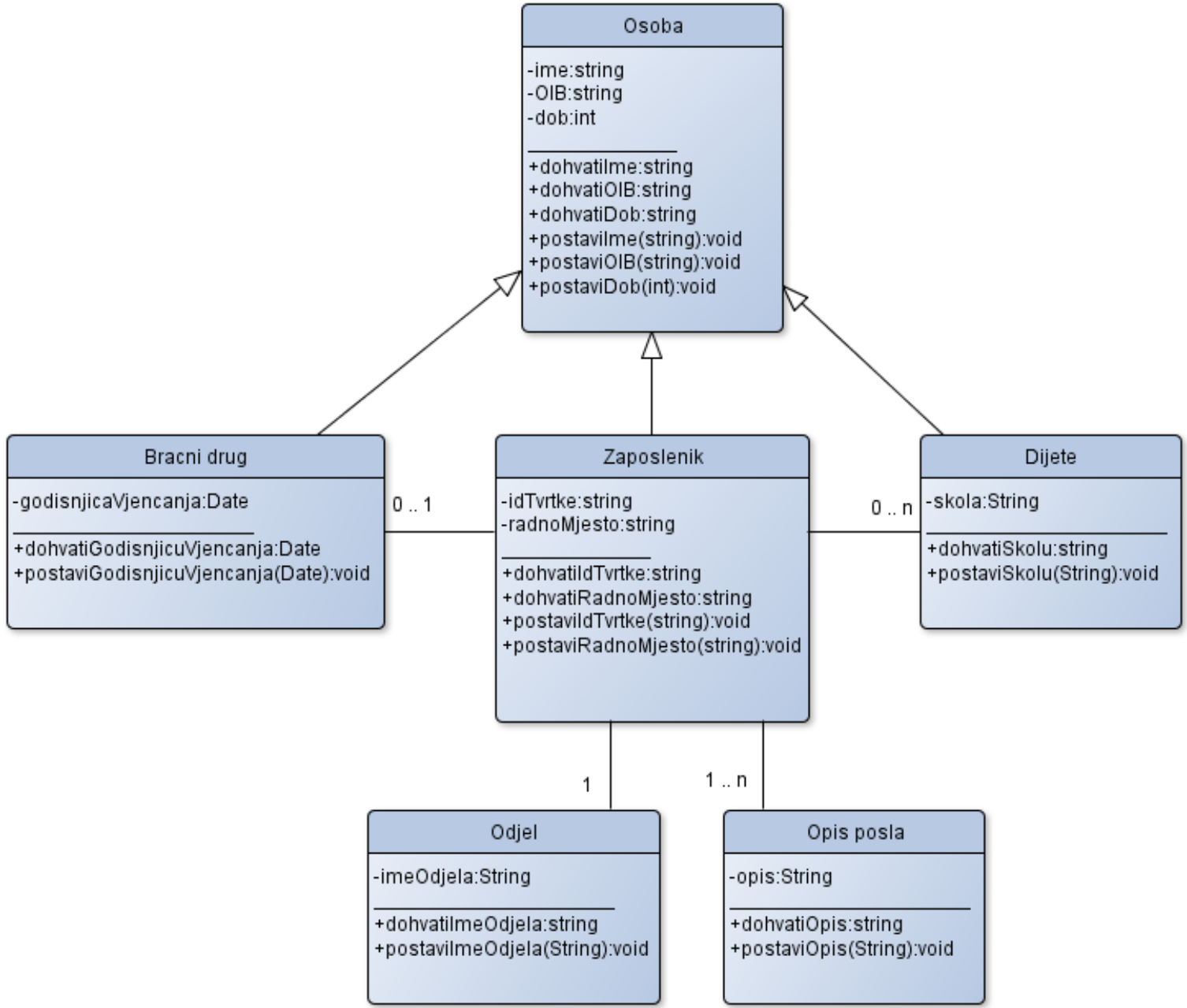
Klase – nastavak, portabilni podaci, notacija

OOP – kontrola pristupa

- **private** – samo ova klasa ima pristup poljima i metodama
- **protected** – ova klasa i njeni nasljednici imaju pristup poljima i metodama
- **public** – i ostale klase imaju pristup poljima i metodama

Kardinalnost

- Kardinalnost je broj objekata koji sudjeluju u nekom aspektu kompozicije
- Sudjelovanje može biti opcionalno ili obavezno
- Da bi se ustanovila kardinalnost potrebno je postaviti sljedeća pitanja:
  - Koji objekti surađuju s drugim objektima?
  - Koliko objekata učestvuje u svakoj suradnji?
  - Da li je suradnja opcionalna ili obavezna?

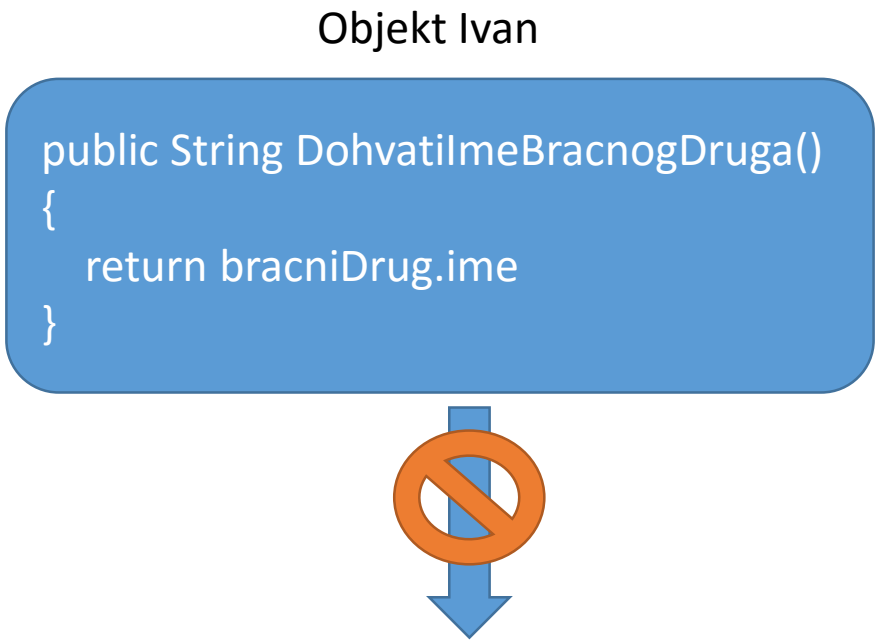

Osoba
-ime:string
-OIB:string
-dob:int
+dohvatiIme:string
+dohvatiOIB:string
+dohvatiDob:string
+postaviIme(string):void
+postaviOIB(string):void
+postaviDob(int):void
Bracni drug
-godisnjicaVjencanja:Date
+dohvatiGodisnjicuVjencanja:Date
+postaviGodisnjicuVjencanja(Date):void
0 .. 1
Zaposlenik
-idTvrtke:string
-radnoMjesto:string
+dohvatiIdTvrtke:string
+dohvatiRadnoMjesto:string
+postaviIdTvrtke(string):void
+postaviRadnoMjesto(string):void
0 .. n
Dijete
-skola:String
+dohvatiSkolu:string
+postaviSkolu(String):void
1
1 .. n
Odjel
-imeOdjela:String
+dohvatiImeOdjela:string
+postaviImeOdjela(String):void
Opis posla
-opis:String
+dohvatiOpis:string
+postaviOpis(String):void

Asocijacija 0 prema n

```
//Java
public class Zaposlenik extends Osoba{
    private String idTvrtke;
    private String radnoMjesto;

    private BracniDrug bracniDrug;
    private Dijete[] dijete;
    private Odjel odjel;
    private OpisPosla[] opisPosla;

    public String dohvatiIdTvrtke() {return idTvrtke;}
    public String dohvatiRadnoMjesto() {return radnoMjesto;}
    public void PostaviIdTvrtke(string Id){idTvrtke =Id;}
    public void PostaviRadnoMjesto(string rMjesto){radnoMjesto =rMjesto;}

}
```

Opcionalne asocijacije

- Potrebno je obaviti provjeru prije korištenja opcionalnih asocijacija
- U kodu treba provjeriti da li je asocijacija *null*



Orthodox Canonic Class Form (OCCF)

- Naputak za ispravni dizajn klase (C++)
- OCCF specificira 4 funkcionalnosti za implementiranje kod definiranja korisničkih tipova podatka
- Osigurava ispravno ponašanje u 4 osnovna konteksta korištenja
- *default konstruktor*
- *kopirni konstruktor* (eng. *copy constructor*)
- *kopirni operator pridruživanja vrijednosti* (eng. *copy assignment operator*)
- *destruktor*

Kopirni konstruktor

- Služi za stvaranje objekata iz već postojećih objekata
- Ako ne deklariramo kopirni konstruktor kompajler ga sam dodijeli
- U tom slučaju se radi o plitkom kopiranju
- Ako koristimo pointere i dinamičku alokaciju memorije u objektima dobro je definirati svoj kopirni konstruktor

```
class Tocka{
  int x;
  int y;
  public Tocka(Tocka& t)
  {
    x = t.x;
    y = t.y;
  }
};

int main()
{
  Tocka t1(0, 0);
  Tocka t2(t1);
}
```

Kopirni operator pridruživanja vrijednosti

- Inicijaliziranje stanja objekta korištenjem već postojećeg inicijaliziranog objekta
- Također bitno ako želimo duboko kopiranje

```
class Tocka{
  int x;
  int y;
  public operator=(const drugi& t)
  {
    x = t.x;
    y = t.y;
  }
};

int main()
{
  Tocka t1(5, 5),t2;
  t2=t1;
}
```

friend funkcije i klase u C++

- Funkcija koja je deklarirana kao *friend* ima pristup privatnim članovima

```
#include<iostream>
using namespace std;
class Tocka{
  int x;
  int y;
  public:
  Tocka(int x_, int y_)
  {
    x = x_;
    y = y_;
  }
  friend int ZbrojKoordinata(Tocka t);
};
int ZbrojKoordinata(Tocka t)
{
  return t.x + t.y;
}
int main()
{
  Tocka t1(5, 5);
  cout << ZbrojKoordinata(t1);
}
```

friend klase

- Prijateljstvo nije simetrično
- Prijateljstvo nije tranzitivno
- Prijateljstvo nije nasljedno

```
#include<iostream>
using namespace std;
class A{
  friend class Tocka;
  int x;
};
class Tocka{
  int x;
  int y;
  public:
  Tocka(A a, int y_)
  {
    x = a.x;
    y = y_;
  }
};
```

Vrste nasljeđivanja u C++

```
class A
{
public:
   int x;
protected:
   int y;
private:
   int z;
};

class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};
```

```
class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

Namespaces

- Zajednički prostor imenovanja omogućava da odvajamo jedan skup imena od drugog
- Koristeći namespace svi simboli u tom namespace-u postanu vidljivi bez dodavanja prefiksa
- Može nam uštedjeti na pisanju koda, ali i izazvati konflikte u imenovanju

```
#include <iostream>

int main()
{
   std::cout << "Hello World";
  return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World";
   return 0;
}
```

Namespaces

- Namespaces možemo promatrati kao pakete
- Ako je klasa definirana u nekom namepace-u onda joj preko tog namespace-a i pristupamo

```
namespace MyNamespace
{
  class MyClass
  {
  };
}
```

Objekti i portabilni podaci

- Java je postigla veći uspjeh upravo zbog svoj portabilnosti – izvodi se na različitim platformama pomoću JVM
- .NET osigurava portabilnost među programskim jezicima
- Jezici su jedna polovina jednadžbe, druga polovina su podaci
- Sustavi se temelje na podacima
- **XML** je standardni format za definiciju i transport podatka između potencijalno heterogenih sustava
- **JSON** se također koristi u slične svrhe

Portabilni podaci

- Povijesno gledano, veliki problemi dolaze zbog različitosti formata pohrane – posebice u velikim tvrtkama i sustavima
- XML se koristi prilikom prebacivanja podataka između heterogenih sustava
- Različite grane industrija su razvile svoju vrstu/standard markup jezika – vokabular
- Low level podaci nisu portabilni, cilj je stvoriti high-level podatke na informacijskoj razini

XML

- XML – eXtensible Markup Language
- HTML – također markup jezik
- Markup jezik je računalni jezik koji koristi *tagove* da bi definirao elemente unutar dokumenta
- Human-readable i može sadržavati standardne riječi umjesto računalne sintakse
- Mogu se spremati kao *plain text*
- I HTML i XML su nasljednici SGML – Standard Generalized Markup Language koji se pojavio 70-ih i standardiziran u 80-ima

XML

- XML nije propriatery (vlasnički)
- World Wide Web Consortium (W3C) predlaže preporuke standarda
- www.w3schools.com
- HTML je za prezentaciju podataka
- XML opisuje podatke
- HTML tagovi su predefinirani
- XML tagove sami određujemo (ili ih definira standard industrije)
- Podatke definiramo sa DTD file-om

DTD

- DTD- Document Type Definition
- Definiraju se tagovi i struktura
- XML file-ovi se provjeravaju da li udovoljavaju DTD-u
- Nije nužno koristiti DTD za XML ali je dobra praksa
- Validiranje dokumenata čini XML robusnijim i sprječava slučajne pogreške
- Ako dokument nije ispravno formatiran generira se greška

XML i objektno orijentirani jezici

- Komunikacija između heterogenih sustava korištenjem OOP i XML
- Racimo da želimo omogućiti komunikaciju između dva sustava – jedan podatke pohranjuje u ORACLE bazi a drugi u SQLServer
- Sustavi su fizički udaljenim lokacijama
- Komunikacija treba biti prilagodljiva namjeni

Dijeljenje podataka između dva sustava

- Definicija vrste podataka koji se trebaju razmjenjivati
- Pisanje DTD-a na osnovu prethodne analize
- Generiranje XML-a sa integriranim DTD-om
- Razmjena podataka i testiranje komunikacije
- Testirati slučajeve nepotpune komunikacije i nepotpunih dokumenata
- Svaki sustav na svojoj platformi ima svoj XML parser i generator ako je potrebna dvosmjerna komunikacija

JSON

- JSON – JavaScript Object Notation
- XML je više strukturiran, posebice korištenjem DTD-a, JSON spada u kategoriju fleksibilnijih formata
- JSON koristi JavaScript sintaksu za opisivanje objekata, ali je ipak neovisan o platformi i jeziku
- JSON parseri i biblioteke postoje za različite programske jezike
- JSON je *lightweight* format za razmjenu tekstualnih podataka
- JSON je neovisan o jeziku
- JSON je *self-describing* i jednostavan za razumjeti

JSON

```
var employees = [
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter","lastName": "Jones"}
];
```

Notacija

- Za notaciju koristimo UML dijagrame
- Dobro definirana i ekspresivna notacija je vrlo važna za razvoj softvera
- Jednako važni kao nacrti u arhitekturi
- Dio razvoja koji se bavi razradom konceptualne strukture
- 1990. – prva verzija UML-a : Booch, RumBaugh i Jacobson, Rational Software Corporation
- Komunicirali sa drugim tvrtkama i ljudima koji se bave metodologijama da bi predložili standard jezika za modeliranje grupi OMG (Object Management Group) konzorciju koji stvara i održava standarde u računalnoj industriji
- 1997. UML je prihvaćen kao standard

UML

- Trenutna verzija – 2.5
- Kao i kod drugih disciplina (kemija, arhitektura, glazba) koji imaju svoj notaciju za predstavljanje „objekata” UML se koristi za modeliranje i predstavljanje sustava koji se izrađuje
- UML model koji napravimo će predstavljati stvarni sustav koji će biti izrađen
- UML ima različite vrste dijagrama koji predstavljaju različite pogleda na naš sustav
- Analogija sa sportskim događajem i snimanje iz više kutova da bi se potpuno razumjela aktivnost

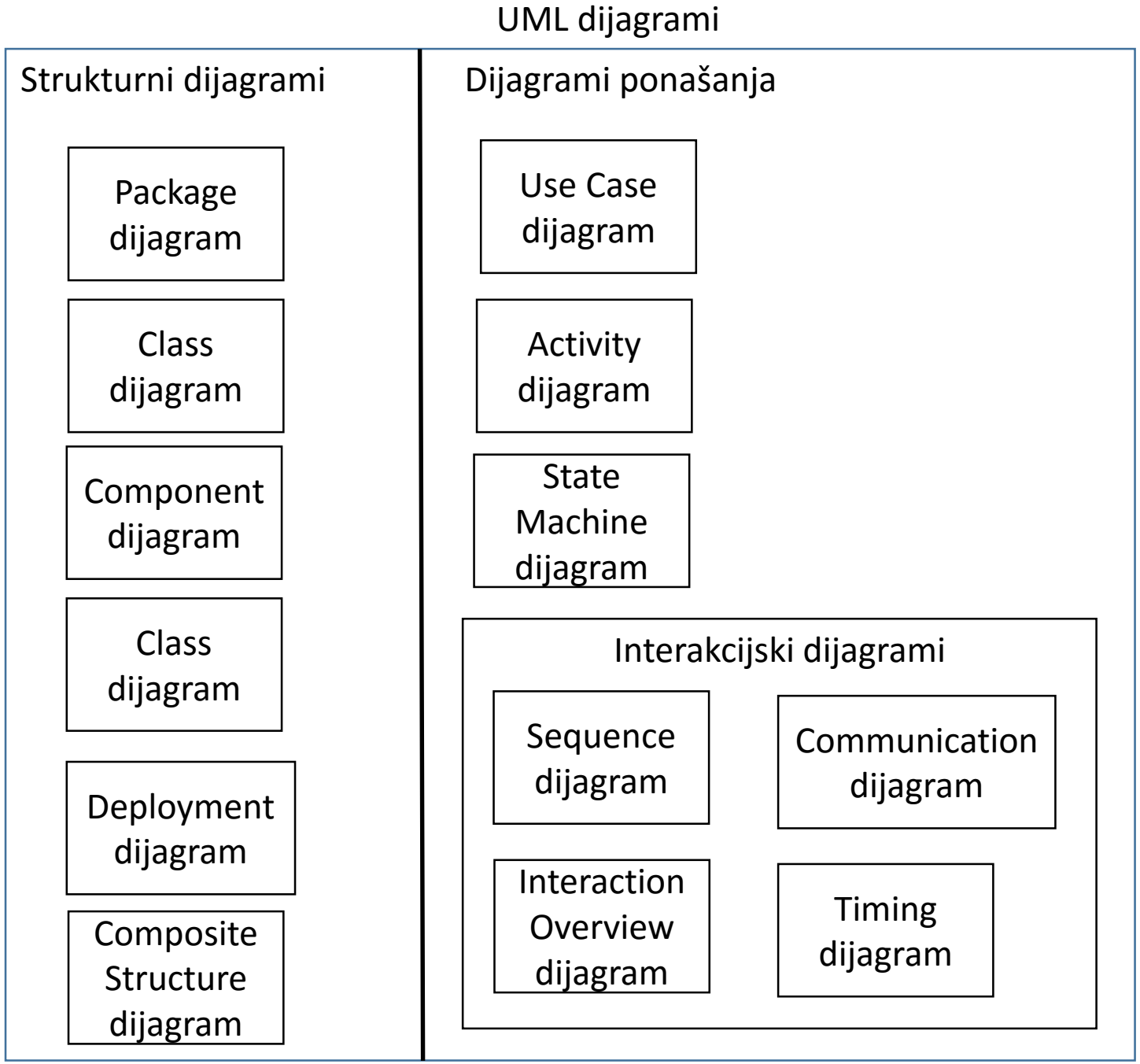
UML

- Razmotrimo aplikaciju koja se sastoji od 100 klasa
- Nije moguće jednim dijagramom predočiti sve klase i njihove odnose
- Koristili bi više class dijagrama od kojih bi svaki predstavio jedan pogled na model
- Npr. jedan dijagram bi prikazivao hijerarhiju nasljeđivanja, drugi koje sve klase komuniciraju s nekom određenom klasom
- U svim vrstama dijagrama iste klase moraju imati ista imena

Taksonomija dijagrama

- UML dijagrami se mogu klasificirati u dvije grupe:
  - **Strukturni dijagrami (eng. Structure diagrams)**
  - **Dijagrami ponašanja (eng. Behavior diagrams)**
- Kompleksnost sustava ovisi i o broju i uređenju elemenata (struktura), kao i o načinu na koji ti elementi surađuju (ponašanje)

Taksonomija dijagrama



Strukturni dijagrami

- Ovi dijagrami se koriste da bi se prikazala statička struktura elemenata sustava
- Mogu prikazivati različite aspekte poput arhitekture sustava, fizičke elemente sustava, elementi vezani za domenu
  - Package diagram
  - Class diagram
  - Component diagram
  - Deployment diagram
  - Object diagram
  - Composite structure diagram
- Strukturni dijagrami se koriste zajedno sa dijagramima ponašanja da bi se opisao određeni aspekt sustava

Dijagrami ponašanja

- Događaji (eng. events) se događaju dinamički u svakom softverskom sustavu: objekti se stvaraju i uništavaju, objekti međusobno šalju poruke na uređeni način, u nekim sustavima postoje vanjski okidači (eng. triggers) koji utječu na unutrašnje stanje sustava

- Use case diagram
- Activity diagram
- State machine diagram
- Interaction diagrams:
- Sequence diagram
- Communication diagram
- Interaction overview diagram
- Timing diagram

Korištenje dijagrama u praksi

- Iako UML može predstavljati vrlo detaljnu specifikaciju to ne znači da sve njegove elemente uvijek nužno moramo koristiti
- Ispravan podskup ovih notacija je dovoljan da se izrazi velika većina elemenata analize i dizajna sustava
- Notacija je samo sredstvo opisivanja i razumijevanja načina rada sustava, a nije cilj dizajna
- Koristimo samo onu notaciju koja nam je nužna za predočiti smisao
- Moramo odlučiti koliku razinu detalja predočiti u kojoj fazi dizajna

Konceptualni, logički i fizički model

- Konceptualni model predstavlja sustav u okviru entiteta domene i njihove povezanosti s drugim entitetima sustava
- Konceptualni model bi se trebao izraziti samo u terminima domene, bez tehničkih detalja
- Logički model iz konceptualnog izvlači ključne apstrakcije i mehanizme koji će tvoriti arhitekturu sustava i cjelokupni dizajn
- Fizički model opisuje konkretnu softversku i hardversku strukturu sustava i njegovu implementaciju
- Tijekom života projekta model će evoluirati iz konceptualnog, kroz logički, do fizičkog stupnja
- Različiti dijagrami se koriste u različitim fazama života projekta

Produkti OO developmenta

- Use case i Activity dijagrami (izražavaju ponašanje sustava kroz različite scenarije)
- Class dijagrami (izražavaju uloge i odgovornosti agenata koji definiraju ponašanje sustava)
- Interaction i State machine dijagrami (izražavaju ponašanje temeljeno na događajima)
- Arhitektura sustava se izražava sa: package diagrams, class digrams, object diagrams, component diagrams, deployment diagrams

Skalabilnost

- UML dijagrami se koriste i kod malih sustava koji imaju nekoliko klasa i kod velikih sustava koji imaju tisuće klasa
- Dijagrami se mogu mijenjati u različitima iteracijama ako se za to stvori potreba
- Ovakva notacija je neovisna o programskom jeziku