Predavanji 8-9

Objektno-usmerjeni programski jeziki

Iztok Savnik, FAMNIT

April, 2024.

Literatura

- Učbeniki:
  - John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapters 10-13)
  - Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapter 9)
- Primeri iz knjige:
  - Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, O'REILLY & Associates, 2000 (Chapter 15)

Pregled

1. Uvod
2. Razredi in objekti
3. Dedovanje
4. Dinamično povezovanje
5. Podtipi in zamenljivost
6. Abstraktni razredi
7. Generativnost
8. Popularni OO programski jeziki

Podatkovna abstakcija

- Z razvojem kompleksnih aplikacij je postala podatkovna abstrakcija ključna za programski inženiring.
  - Zmanjšuje konceptualno obremenitev z minimizacijo količine podrobnosti o katerih mora programer razmišljati v danem trenutku.
  - Omejitev napak z onemogočanjem programerju uporabo komponent na nepravilen način.
  - Nudi nivo neodvisnosti med komponentami programa, ki so ločeni po funkcionalnosti.

Podatkovna abstakcija

- V prog. jezikih mehanizmi abstrakcije poudarjajo splošne lastnosti nekega segmenta kode in skrijejo podrobnosti.
  - Funkcije, procedure, tipi, razredi, moduli
- Podatkovna abstrakcija uveljavi čisto razločitev med abstraktnimi lastnostmi in konkretnimi podrobnostmi podatkovnega tipa
- Oblike podatkovnih abstrakcij
  - Skrivanje informacij, inkapsulacija,
  - Vmesnik / Implementacija
  - Neodvisnost od predstavitve / implementacije

Podatkovna abstrakcija in inkapsulacija

- Abstrakciji: razred in objekt
  - Kreirajo poenostavljen pogled na neko stvar.
  - Fokusiranje pozornosti na najbolj pomembne značilnosti.
  - Odmislimo podrobnosti o predstavitvi in implementaciji.
  - Podatki so skriti v objektu.
  - Zunanji opazovalec razreda vidi samo abstrakten pogled na razred in primerke razreda;
    - To omogoča vmesnik razeda
  - Vmesnik je sestavljen iz dobro definiranih metod s katerimi delamo s primerki razreda.
- Razredi in objekti inkapsulirajo interno strukturo in obnašanje objektov.

Objektno-usmerjen model

- Objekt torej sestavljata množica operacij in skriti podatki
  - Vsa interakcija z objektom poteka preko sporočil oz. klicev operacij (metod)
- Objekti so grupirani v razrede, ki služijo kot prototipi objektov
  - Razredi so množice primerkov (denotacijski pogled)
  - Kot prototipi, razredi predstavljajo tipe sestavljene iz signatur metod in podatkovnih članov
  - Razredi podedujejo lastnosti njihovih nad-razredov (super-razredov).

Primer: int_stack

- Sklad celih števil
  - Implementacija v Ocaml
  - Sklad je predstavljen s seznamom
- Inkapsulacija
  - Predstavitev in implementacija sklada se lahko spremeni n
  - Vmesnik (abstrakcija) ostane isti

```
# class int_stack =
    object
      val mutable l = ([] : int list)
      method push x = l <- x::l
      method pop = match l with
                              [] -> raise Empty |
                              a::l' -> l <- l'; a
      method clear = l <- []
      method length = List.length l
    end;;
# let is = new int_stack;;
val is : int_stack = <obj>
# is#push 1; is#push 2;
   is#push 3; is#push 4;;
- : unit = ()
# is#length;;
- : int = 4
# is#pop;;
- : int = 4
```

Zgodovina

- Simula, 1960, Norveški računski center
  - Prvi objektno-usmerjen jezik
  - Vsebuje vse kar vsebujejo novejši OO jeziki
  - Objekti/razredi, dedovanje, podrazrede, virtualne procedure
- Smalltalk, 1970, Xerox PARC
  - Objektno-usmerjen jezik z dinamičnim preverjanjem tipov
  - Vse je objekt; še vedno zanimiva zasnova (Scala)
  - Objekt/razred, sporočila, podrazredi, meta-razredi
  - Strukturna in računska refleksija: objekt lahko opazuje in vpliva na lastno strukturo in obnašanje

Osnovni OO programski jeziki

- C++, 1983, Bjarne Stroustrup, Bell Labs
  - Široko uporabljan OO jezik s statičnim preverjanjem tipov
  - Razširitev programskega jezika C; standardiziran 2014
  - Razredi/objekti, dedovanje (večkratno), polimorfizem, kalupi, virtualne funkcije in razredi, izjeme
  - Kompatibilnost s C
- Java, 1995, James Gosling, Sun Microsystems
  - Vzporeden, osnovan na razredih, objektno-usmerjen
  - Prevede se v zlogovno kodo; izvaja se na kateremkoli Javinem virtualnem stroju
  - Enkratno dedovanje, abstraktni razredi, vmesniki, generičnost, introspekcija, moduli osnovani na dat.sistemu
  - Prenosljivost, reliability, varnost, dinamično nalaganje, niti, enostavnost, učinkovitost

Pregled

1. Uvod
2. Razredi in objekti
3. Dedovanje
4. Dinamično povezovanje
5. Podtipi in zamenljivost
6. Abstraktni razredi
7. Generativnost
8. Popularni OO programski jeziki

Razredi in objekti

- Model objektov je blizu simulacijskem pogledu k reševanju problemov: naredi simulacijski model in ga poženi
  - Objekti predstavljajo komponente modela
  - Objekti so lahko med seboj povezani v kompleksno statično strukturo
  - Object lahko sporoči drugemu objektu naj požene opravilo tako, da mu pošlje sporočilo
  - Skupno obnašanje objektov je definirano s komunikacijo med povezanimi objekti
  - Uporabljene abstrakcije so veliko bližje človeški predstavitvi problema

Razredi in objekti

- Razred lahko vidimo kot prototipni objekt iz katerega kreiramo primerke (člane razreda)
  - Razred vsebuje definicijo statične strukture in obnašanja
  - Obnašanje primerkov razreda je implementirano z metodami
  - Metode uporabljajo interno logiko za komunikacijo z ostalimi objekti pri reševanju podproblemov
  - Nove razrede lahko konstruiramo iz obstoječih razredov
    - Specializacija in generalizacija
    - Kompozicija in dekompozicija

Definicija razreda v Ocaml

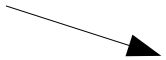
- Podatkovni člani
  - Imajo poljubni tip
  - Vrednosti so spremenljive
- Metode
  - Metode imajo parametre p1,...,pn
- Razredi imajo parametere

```
val name = expr
val mutable name = expr
name <- expression
```

```
method name p1 . . . pn = expr
```

```
class name p1 ... pn =
  object
  ...
  instance variables
  ...
  methods
  ...
end
```

Tip!



```
#class point x_init =
  object
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
  end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end
```

Kreiranje objekta

```
# new point;;
- : int -> point = <fun>
# let p = new point 7;;
val p : point = <obj>
```

- Večina jezikov uporablja operator `new( )`
  - Ocaml nima konstruktorjev podobno kot C++, Java
  - Razred funkcionira kot generator: razred je funkcija, ki kreira objekte
  - Parametri razredov podajo vrednosti uporabljene za inicializacijo podatkovnih članov objekta
- Inicializacija
  - Kodo lahko dodamo pred in po kreaciji objekta
  - Pred: `let` stavek lahko vrinemo pred `object`
  - Po: Ocaml uporablja posebno funkcijo `initializer`
- Kako je inicializacija narejena v Java?

Primer:

```
# class printable_point x_init =
   let origin = (x_init / 10) * 10 in
   object (self)
     val mutable x = origin
     method get_x = x
     method move d = x <- x + d
     method print = print_int self#get_x
     initializer print_string "new point at ";
                 self#print; print_newline()
   end;;
class printable_point :
  int ->
  object
    val mutable x : int
    method get_x : int
    method move : int -> unit
    method print : unit
  end
# let p = new printable_point 17;;
new point at 10 val p : printable_point = <obj>
```

Metode

- Pošiljanje sporočila
  - o#message( ) (Ocaml)
  - Klic funkcije v C++, Java, Ocaml, ...
  - Dejanska sporočila v Smalltalk, Erlang, ...
  - Primerna metoda mora biti implementirana v razredu
  - Tipi dejanskih parametrov se morajo ujemati s formalnimi param.

```
# class point (x_init,y_init) =
    object
      val mutable x = x_init
      val mutable y = y_init
      method get_x = x
      method get_y = y
      method moveto (a,b) = x <- a ; y <- b
      method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
      method to_string () =
          "("^ (string_of_int x) ^ ", "^ (string_of_int y) ^")"
      method distance () = sqrt (float(x*x + y*y))
    end ;;
```

```
# let p1 = new point (0,0);;
val p1 : point = <obj>
# let p2 = new point (3,4);;
val p2 : point = <obj>
# p1#get_x;;
- : int = 0
# p2#to_string () ;;
- : string = "(3, 4)"
# if (p1#distance () ) = (p2#distance () )
   then print_string ("That's just chance\n")
   else print_string ("We could bet on it\n");;
We could bet on it
- : unit = ()
```

Privatne metode

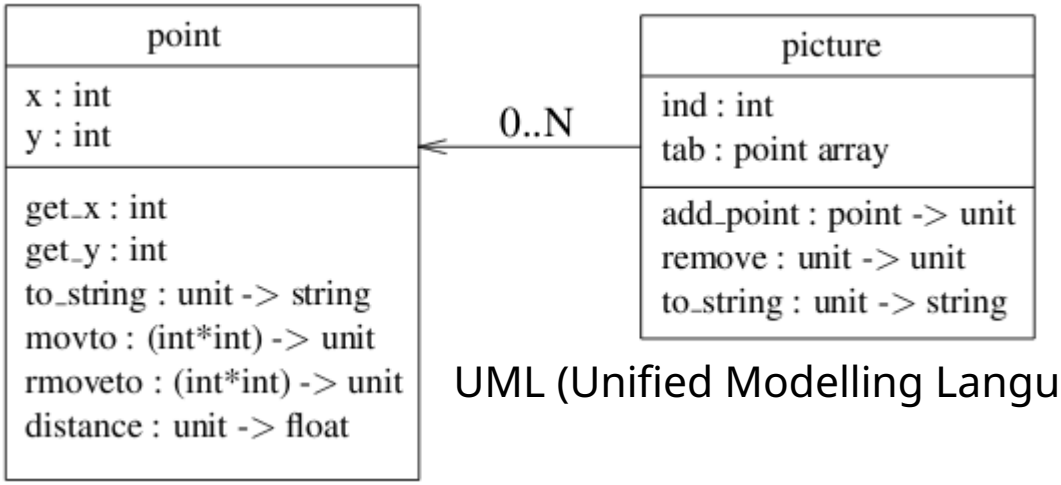
- Privatne in javne metode
  - Privatne metode so dostopne samo lokalno iz objektov
- Primeri privatnih metod
  - Točka si zapomni prejšnjo pozicijo
  - mem_pos()

```
method private name = expr
```

```
# class point (x0,y0) =
    object(self)
      val mutable x = x0
      val mutable y = y0
      val mutable old_x = x0
      val mutable old_y = y0
      method get_x = x
      method get_y = y
      method private mem_pos () = old_x <- x ; old_y <- y
      method undo () = x <- old_x; y <- old_y
      method moveto (x1, y1) = self#mem_pos (); x <- x1; y <- y1
      method rmoveto (dx, dy) = self#mem_pos (); x <- x+dx; y <- y+dy
      method to_string () =
            "("^ (string_of_int x) ^ ","^ (string_of_int y) ^")"
      method distance () = sqrt (float(x*x + y*y))
   end ;;
```

Strukture objektov in agregacija

- Objekt je sestavljen iz drugih objektov
  - Razredi definirajo druge razrede kot komponente
  - Objekti definirajo zveze med dvemi ali več objekti
- Agregacija je ena izmed dveh pomembnih abstrakcij uporabljenih v OO jezikih
  - Agregacija (kompozicija): relacija “ima” (has-a)
  - Dedovanje (specializacija): relacija “je” (is-a)
- Primer v Ocaml
  - Razred `Picture` vsebuje polje primerkov razreda `Point`
  - Metoda `to_string()` pokliče metodo `to_string()` razreda `Point`



UML (Unified Modelling Language)

Primer

Tip

```
# class picture n =
    object
      val mutable ind = 0
      val tab = Array.create n (new point(0,0))
      method add p = tab.(ind)<-p ; ind <- ind + 1
      method remove () = if (ind > 0) then ind <-ind-1
      method to_string () =
        let s = ref "["
        in for i=0 to ind-1 do
              s:= !s ^ " "^ tab.(i)#to_string () done ;
           (!s) ^ "]"
    end ;;
```

```
class picture :
  int ->
  object
    val mutable ind : int
    val tab : point array
    method add : point -> unit
    method remove : unit -> unit
    method to_string : unit -> string
  end
```

Pregled

1. Uvod
2. Razredi in objekti
3. Dedovanje
4. Dinamično povezovanje
5. Podtipi in zamenljivost
6. Abstraktni razredi
7. Generativnost
8. Popularni OO programski jeziki

Dedovanje

- Dedovanje je implementacija abstrakcije specializacija / generalizacija
  - Aristotel: genus + differentia specifica.
  - Podrazred je specializacija nadrazreda.
  - Podrazred podeduje vse lastnosti nadrazreda.
- Ocaml sintaksa za definicijo dedovanja

```
inherit name1 p1 . . . pn [ as name2 ]
```

  - Parametri nadrazreda so p1, ..., pn.
  - Del objekta predstavljen z nadrazredom lahko referenciramo znotraj razreda z uporabo spremenljivke name2.

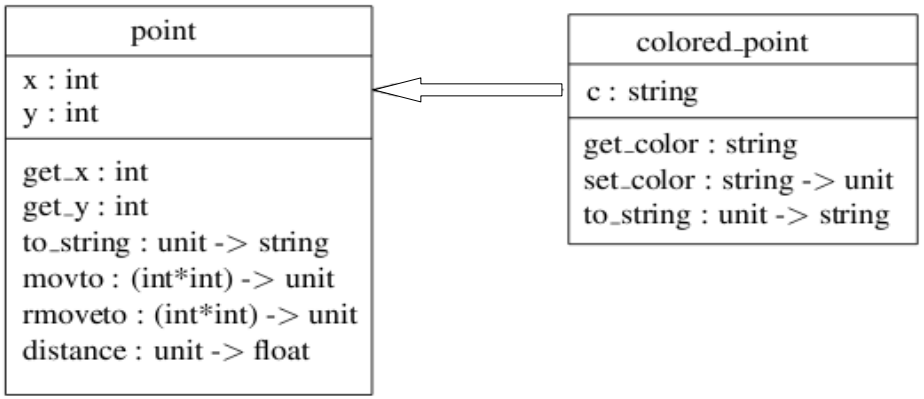
Primer podrazreda

```
# class colored_point (x,y) c =
    object
      inherit point (x,y)
      val mutable c = c
      method get_color = c
      method set_color nc = c <- nc
      method to_string () = "("^ (string_of_int x) ^
                            ","^ (string_of_int y) ^ ")"^
                            "["^ c ^ "]"
    end ;;
```

```
# let pc = new colored_point (2,3) "white";;
val pc : colored_point = <obj>
# pc#get_color;;
- : string = "white"
# pc#get_x;;
- : int = 2
# pc#to_string () ;;
- : string = "(2,3)[white] "
# pc#distance;;
- : unit -> float = <fun>
```

Tip:

```
class colored_point :
  int * int ->
  string ->
  object
    val mutable c : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_color : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method set_color : string -> unit
    method to_string : unit -> string
  end
```



Prekrivanje metode

- Metoda to_string() iz razreda colored_point() prekrije metodo to_string() iz razreda point()
  - Ob klicu se požene najbolj specifična metoda (če ni večkratnega dedovanja)
    - Bolj natančen opis relacije med tipi metod: kovarianca in kontravarianca.
    - Razmerja med tipi metod bo predstavljena bolj podrobno.
- Posledice prekrivanja metod zaradi dedovanja
  - Dinamično povezovanje tipa metode s korektno kodo v času izvajanja (dinam. zaradi dedovanja in zamenjlivosti)
    - Sicer se koda določi v času prevajanja (statično)
  - Večkratno dedovanje
    - Razred podeduje isto metodo od dveh nadrazredov. Rešitve tega problema bomo videli kasneje.

Referenci na self in super

- Ko definiramo metodo morami imeti dostop do
  - samega objekta in
  - dela objekta, ki je predstavljen z nadrazredom.
- Referenca na sam objekt
  - Ključna beseda “self” v Smalltalk. Java in C++ ?
  - Ocaml dovoljuje uporabniku definirati ime na sam objekt
- Referenca na nadobjekt
  - Ocaml dovoljuje definicijo posebnega imena za nadobjekt
  - Omogoča dostop do spemenljivke v primeru, da je prekrita z neko drugo spremenljivko
  - Klicanje metod samega objekta

```
super.remove();        // Java
base.remove();        // C#
super remove.          // Smalltalk
[super remove]         // Objective C
```

Referenci na self in super

- Metoda to_string()
  - Uporaba to_string() iz nadrazreda
  - Uporaba self za klicanje get_color()

```
# class colored_point (x,y) c =
    object (self)
      inherit point (x,y) as super
      val mutable c = c
      method get_color = c
      method set_color nc = c <- nc
      method to_string () = super#to_string () ^
                            "["^ self#get_color ^ "]"
    end ;;
```

Inicializacija objekta

- Instanca razreda C se inicializira z
  - Kodo inicializatorjev vseh nadrazredov C
  - Kodo inicializatorja razreda C
- Vrstni red evaluacije je od najbolj splošnega proti bolj specifičnim inicializatorjem
- Kako je objekt inicializiran v Javi? C++?

Inicializacija objekta

- Instanca razreda C se inicializira z
  - Kodo inicializatorjev vseh nadrazredov C
  - Kodo inicializatorja razreda C
- Vrstni red evaluacije je od najbolj splošnega proti bolj specifičnim inicializatorjem
- Kako je objekt inicializiran v Javi? C++?
  - V Javi napišemo v prvo vrstico konstruktorja klic konstruktorja nadrazreda. Sicer se pokliče privzet konstruktor nadrazreda.
  - V C++ so uporabljena podobna pravila kot v Javi.

Primer inicializacije objekta

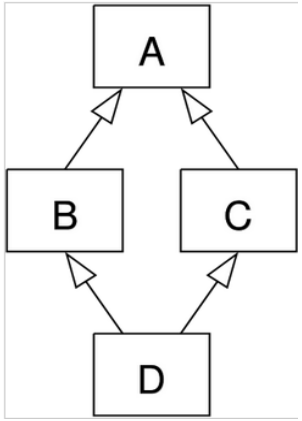
```
# class point (x_init,y_init) =
    object (self)
      val mutable x = x_init
      val mutable y = y_init
      method get_x = x
      method get_y = y
      method moveto (a,b) = x <- a ; y <- b
      method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
      method to_string () =
        "(" ^ (string_of_int x) ^ "," ^ (string_of_int y) ^ ")"
      method distance () = sqrt (float(x*x + y*y))
      initializer
        Printf.printf ">> Creation of point: %s\n"
                      (self#to_string ());
    end ;;
```

```
# class verbose_point p =
    object (self)
      inherit point p as super
      method to_string () = "point=" ^ (super#to_string ()) ^
              ",distance=" ^ string_of_float (self#distance ())
      initializer
        Printf.printf ">> Creation of verbose point: %s\n"
                      (self#to_string ())
    end ;;
# new verbose_point (1,1);;
>> Creation of point: (1,1)
>> Creation of verbose point: point=(1,1), distance=1.414213
- : verbose_point = <obj>
```

Večkratno dedovanje

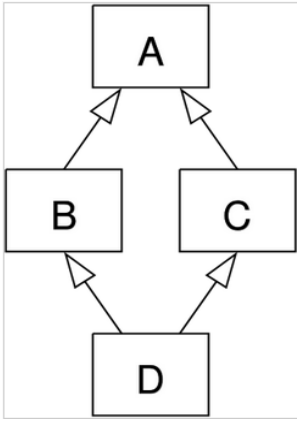
- Koncept, ki ga modeliramo z razredom je lahko specializacija več kot enega koncepta
  - Razred ima lahko več kot enega nadrazreda
  - Nekateri jeziki dovolijo samo enkratno dedovanje
    - Java, C#, Ruby, Scala, ...
  - Drugi jeziki uporabljajo večkratno dedovanje
    - C++, Ocaml, Perl, ...
- Pogosta konceptualna napaka
  - Avto lahko definiramo kot razred, ki podeduje od razredov kolesa, motor, itd.
  - Specializacijo ne smemo uporabljati za modeliranje kompozicije, čeprav je to tehnično mogoče

Trki imen



- Metoda m je definirana v A, B in C
  - Metodo m pošljemo objektu razreda D
  - Katero implementacijo metode izbrati?
- Implicitna razrešitev trka imen
  - Jeziki razrešijo dostop do konfliktnih imen z množico pravil
  - Pravila uporabljajo sintaktično urejenost napovedi nadrazredov
    - Prva definicija v hierarhiji nadrazredov z rekurzivnim iskanjem (Python, CLOS, Perl, itd.)
    - Zadnja definicija v hierarhiji nadrazredov z rekurzivnim iskanjem (Ocaml)
  - Dylan, Python, Perl: C3 vrstni red iskanja metode [OPSLA,96]

Trki imen



- Eksplicitna razrešitev trka imen
  - Program eksplicitno razreši konflikt imen z direktnim dostopom do izbranega imena (C++, Ocaml)
  - Metoda je identificirana eksplicitno (nedvoumno)
  - Ocaml: Uporaba imen nadrejenih razredov
  - C++: Uporaba poti do ciljnega razreda B::m() ali C::m()
- Trki imen niso dovoljeni
  - Jezik ne dovoli definicijo trkov imen
    - Java, C#, Swift, Go, Scala, … vendar ima Java8 problem z vmesniki vmesnikih

C3 linearizacija super-razredov

- Definira vrstni red v katerem iščemo metodo v hierarhiji super-razredov z večkratnim dedovanjem.
- C3 urejenost za iskanje metod (MRO)
  - Rekurziven algoritem (podoben topološkem sortiranju)
    - Začne z razredom, ki je koren hierarhije; Napreduje nivo za nivojem; Uporaba linearizacije staršev za izračun linearizacije otrok; Lep algoritem za zlivanje linearizacij staršev.
  - Uporabljeno: Dylan, Python, Perl, ...
- C3 linearizacija super-razredov ima naslednje tri (C3) pomembne lastnosti:
  - Konsistenten razširjen prednostni graf
  - Ohranitev lokalnih prednostnih urejenosti in
  - Zagotavljanje monotonosti

Primer večkratnega dedovanja

```
# class colored_point (x,y) c =
    object (self)
      inherit point (x,y) as super
      val mutable c = c
      method get_color = c
      method set_color nc = c <- nc
      method to_string () = super#to_string () ^
                            "["^ self#get_color ^ "]"
    end ;;
```

```
# class reference n d =
      object
        val mutable name = n
        val mutable descr = d
        method to_string () = "{"^ name ^ "}"
      end ;;
```

```
# class reference_point (x,y) c n d =
    object (self)
      inherit colored_point (x,y) c as cp
      inherit reference n d as ref
    end ;;
# let rp1 = new reference_point (1,1) "red" "r1"
"reference point";;
val rp1 : reference_point = <obj>
# rp1#get_x;;
- : int = 1
# rp1#to_string ();;
- : string = "{r1}"
```

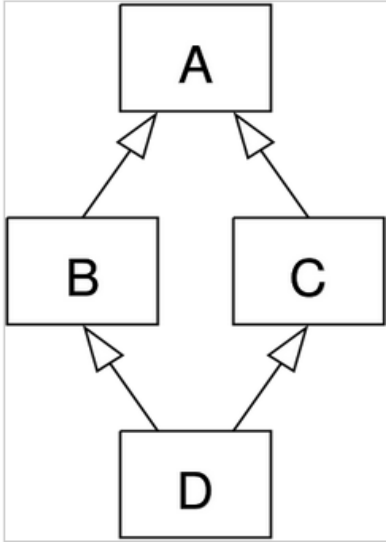
Zakaj?

Večkratno dedovanje v Ocaml

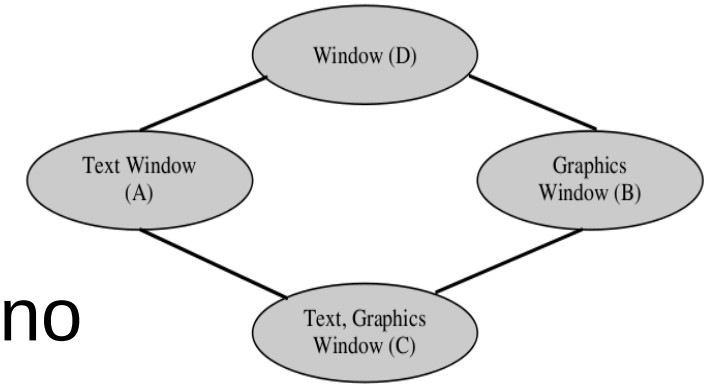
```
# class reference_point (x,y) c n d =
    object (self)
      inherit colored_point (x,y) c as cp
      inherit reference n d as ref
      method to_string () = cp#to_string () ^ ref#to_string ()
    end ;;
# let rp1 = new reference_point (1,1) "red" "rp1" "reference point";;
val rp1 : reference_point = <obj>
# rp1#to_string ();;
- : string = "(1, 1)[red]{rp1}"
```

Problem diamanta

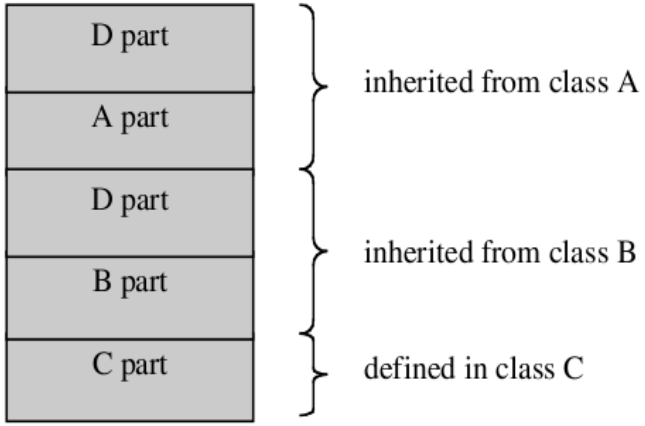
- Hierarhija razredov v obliki diamanta: primerki D imajo dve kopiji objekta A!
  - Razred D deduje od razreda B in C.
  - Oba razreda B in C dedujeta podatkovne člane in metode od rezreda A.
  - Primerki D vsebujejo dve kopiji podatkovnih članov A.
  - Rešitve?



Problem diamanta



- Primer: Tekstovno | grafično okno
  - C vsebuje dve instanci objekta D
  - Ker A in B dedujeta od D
    oba vsebujeta primerek D
- Ni čiste rešitve!
  - Dve kopiji D je lahko v redu
    - Pojavi se problem imen (trk)
    - C++ in Ocaml lahko sledita poti dedovanja
  - Daj eno kopijo D v instanco C
    - C++ virtualni (osnovni) razredi
    - Instanci A in B referencirata isto instanco D
    - To je lahko problem, če A in B obravnavata D del različno



Pregled

1. Uvod
2. Razredi in objekti
3. Dedovanje
4. Dinamično povezovanje
5. Podtipi in zamenljivost
6. Abstraktni razredi
7. Generativnost
8. Popularni OO programski jeziki

Dinamično povezovanje

- Posledice dedovanja in hierarhije razredov
  - Izpeljan razred D ima vse člane—podatkovne in metode—osnovnega razreda C
  - Vse kar bi hoteli narediti na objektu razreda C lahko naredimo tudi na objektu razreda D
- Objekt sprejme sporočilo; kako se določi koda?
  - Koda metode se določi na osnovi implementacije objekta
  - Objekt “izbere” odgovor na sporočilo.
  - Različni objekti izvedejo isto sporočilo na drugačen način.
- Zelo uporaben gradnik OO PJ
  - Pošiljanje istega sporočila objektom iz kolekcije, ki lahko vsebuje objekte različnih tipov.

Dinamično in statično povezovanje

- Kdaj in kako se metoda poveže s kodo?
- Dinamično in statično povezovanje
  - Dinamično povezovanje se zgodi v času izvajanja
    - Smalltalk, Objective-C, Ocaml, Python, Ruby (samo d.p.)
    - Java, Eiffel (privzeto); `final` classes can be optimized
  - Statično povezovanje se zgodi v času prevajanja
    - Simula, C++, C#, Ada95 (default); ključna beseda `virtual` omogoči dinamično povezovanje
  - Nekateri PJ uporabljajo oboje.
- Dinamično povezovanje je dražje kot statično.
  - Primerna metoda se išče v Vpogledni Tabeli Metod (okr. VTM)
    - Smalltalk išče po tabeli v času izvajanja
    - C++, Ocaml, ... izračuna indeks v času prevajanja; pri vpogledu metode dostopa do VTM.

Dinamično povezovanje

- Klasičen primer dinamičnega povezovanja
  - Imamo kolekcijo objektov razreda C
  - Kolekcija lahko vsebuje tudi instance podrazredov razreda C
  - Isto sporočilo, na primer to_string(), se pošlje vsem objektom v kolekciji .
  - Vsak objekt izbere implementacijo sporočila dinamično.
- Primer v Ocaml
  - Definirali smo razrede point, colored_point in verbose_point
  - Vsi razredi imajo definirano metodo to_string()

Primer 1

```
# class picture n =
    object
      val mutable ind = 0
      val tab = Array.create n (new point(0,0))
      method add p = tab.(ind)<-p ; ind <- ind + 1
      method remove () = if (ind > 0) then ind <-ind-1
      method to_string () =
        let s = ref "["
        in for i=0 to ind-1 do s:= !s ^ " "^ tab.(i)#to_string () done ;
           (!s) ^ "]"
    end ;;
```

```
# let pic = new picture 3;
>> Creation of point: (0,0)
val pic : picture = <obj>
# pic#add (new point (1,1));
  pic#add ((new colored_point (2,2) "red") :> point);
  pic#add ((new verbose_point (3,3)) :> point);;
- : unit = ()
# pic#to_string () ;;
- : string = "[ (1,1) (2,2)[red] point=(3,3),distance=4.24264068712]"
```

Primer 2

```
# class colored_point (x,y) c =
    object (self)
      inherit point (x,y) as super
      val mutable c = c
      method get_color = c
      method set_color nc = c <- nc
      method to_string () = super#to_string () ^
                                "["^ self#get_color ^ "]"
    end ;;
```

```
# let p1 = new colored_point (1,1) "Blue";;
val p1 : colored_point = <obj>
# p1#to_string () ;;
- : string = "(1,1) [Blue] "
# let p2 = new colored_point_1 (1,1) "Blue";;
val p2 : colored_point_1 = <obj>
# p2#to_string () ;;
- : string = "(1,1) [UNKNOWN] "
```

```
# class colored_point_1 coord c =
  object
    inherit colored_point coord c
    val true_colors = ["white"; "black"; "red"; "green"; "blue"; "yellow"]
    method get_color = if List.mem c true_colors then c else "UNKNOWN"
  end ;;
```

Implementacija objektov v Ocaml

- Objekt je predstavljen z dvemi deli
- Spremenljivi del
  - Vsebuje podatkovne člane objekta kot v primeru zapisov
  - Uporablja se model referenc ali vrednosti (nekateri PJ uporabljajo oboje)
- Fiksni del
  - Vpogledna tabela metod VTM (angl. method lookup table) shrani metode do katerih dostopamo dinamično
  - VTM shrani podatke o metodah razreda vključno z metodami nadrazredov.
  - Fiksni del je isti za vse instance nekega razreda
  - Obstajajo različne implementacije VTM v različnih PJ

Dinamičen vpogled metode (implementacija)

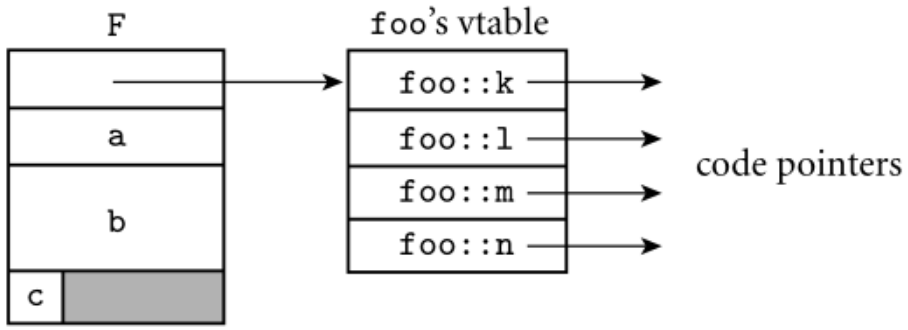
- Poišči kodo metode za klic metode `o.m(v1,v2,...)`
- Statičen vpogled metode
  - Prevajalnik določi metodo na osnovi tipa metode.
- Dinamični vpogled metode
  - Koda, ki jo generira prevajalnik mora najti pravo metodo
  - Objekt je predstavljen z zapisom, ki vsebuje kazalec na vpogledno tabelo metod
    - Vsak razred ima svojo VTM; reference na nove metode so dodane metodam super-razredov.
    - Prekrivanje ⇒ ref na staro metodo se zamenja z novo metodo
    - Ocaml: Isti indeks služi za isto metodo v celotni družini razredov
  - Vpogledna tabela metod lahko vsebuje tudi kazalec na opis tipa (deskriptor) za izvedbo preverjanja tipov v času izvajanja

Implementacija objektov v C++

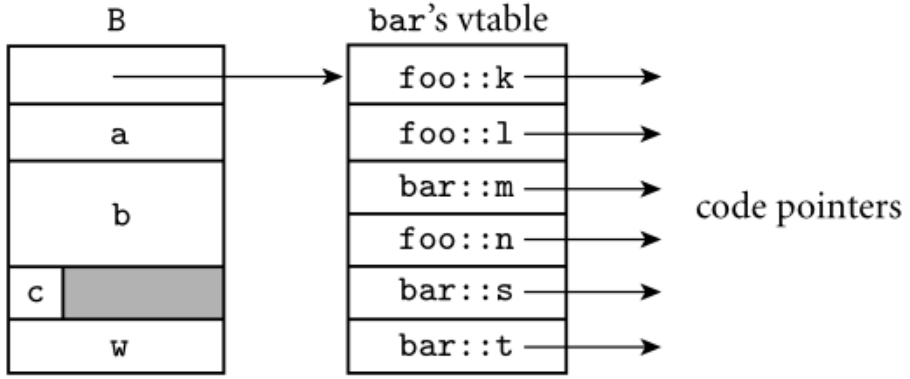
- Statiči in dinamični del objekta
- C++ preveri tipe statično
- Dinamično povezovanje deluje samo na virtualnih metodah

Implementacija objektov v C++

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```




```
class bar : public foo {
    int w;
public:
    void m() override;
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```



Pregled

1. Uvod
2. Razredi in objekti
3. Dedovanje
4. Dinamično povezovanje
5. Podtipi in zamenljivost
6. Abstraktni razredi
7. Generativnost
8. Popularni OO programski jeziki

Podtipi

- Relacija podtip je definirana nad tipi; omogoča da se vrednosti enega tipa uporabljajo namesto vrednosti drugega tipa.
  - Predpostavljamo, da opazujemo jezik s tipi
- Ocaml: tip razreda C je zapis, ki vsebuje tipe vseh članov (podat.&fun.) razreda C
  - Primer: tip razreda Point
- Formalno tip razreda Point vsebuje samo javne funkcije

```
< distance : unit → float; get_x : int; get_y : int;
  moveto : int * int → unit; rmoveto : int * int → unit;
  to_string : unit → string >
```

```
class point :
  int * int ->
 object
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
end
```

Podtipi

V Ocaml je razred funkcija,
ki konstruira objekt.

- Tip je različen koncept od razreda
  - Razred je tovarna, oz. prototip, ki generira objekte
  - Razred predstavlja množico objektov danega tipa
  - Tip je konceptualno entiteta, ki je en nivo višje od instanc
- Relacija podtip je definirana nad tipi razredov
  - Tip t’ je podtip t, formalno $t' \leq t$, čče $\sigma_i \leq \tau_i$ za $i \in \{1, \ldots, n\}$.

    $t =< m_1 : \tau_1 ; \ldots m_n : \tau_n >$ in

    $t' =< m_1 : \sigma_1 ; \ldots ; m_n : \sigma_n ; m_{n+1} : \sigma_{n+1} ; \ldots >$
  - Tip t' ima lahko več komponent kot t in tipi skupnih komponent morajo biti v relaciji podtip.

Podtipi funkcij

- Funkcijski klic
  - Če velja f : t → s, a : t' in t' ≤ t potem je tip (f a) dobro definiran in ima vrednost s
  - Intuitivno: funkcija f pričakuje argument tipa t in lahko varno sprejme argument podtipa t' tipa t
  - Zakaj? Primerki t' imajo lahko kvečjemu dodatne komponente ali bolj specifične komponente
- Tip funkcije
  - Tip t' → s' je podtip tipa t → s, kar zapišemo t' → s' ≤ t → s, čče s' ≤ s in t ≤ t'
  - Relacija s' ≤ s se imenuje kovarianca, in relacijo t ≤ t' imenujemo kontravarianca.
  - Čeprav presenetljivo, se relacijo da enostavno upravičiti

Kovarianca/kontravarianca

- Obrazložitev kovariance/kontravariance
  - Naj bo $f : t \rightarrow s$ in $f' : t' \rightarrow s'$.
  - Kdaj lahko klic f zamenjamo s klicom f' ?
  - Argument f se lahko uporabi kot argument f' če velja $t \leq t'$. To je kontravarianca.
  - Rezultat f' je sprejemljiv v kontekstu rezultata f če velja $s' \leq s$. To je kovarianca.
  - Velja torej $t' \rightarrow s' \leq t \rightarrow s$, čče $t \leq t'$ in $s' \leq s$.

Podtipi metod

- Predpostavimo, da imata razreda $c1$ in $c2$ oba definirano metodo m
  - Metoda m ima tip $t_1 \rightarrow s_1$ v $c1$, in tip $t_2 \rightarrow s_2$ v $c2$
  - $m_{(1)}$ je metoda m iz $c1$ in $m_{(2)}$ je metoda $c2$
  - $c2 \leq c1$ in $t_2 \rightarrow s_2 \leq t_1 \rightarrow s_1$
- g : $s_1 \rightarrow \alpha$, in g(o#m(x)), kjer velja $o{:}c1$ in $x{:}t_1$
  - g predstavlja kontekst---tip s_1
  - Drugi scenarij: $a = o\#m(x)$ kjer je $a{:}s1$ in $x{:}t1$
- Kovarianca
  - g pričakuje objekt tipa $c1$ kot argument
  - $c2 \leq c1 \Rightarrow$ legalno je uporabljati objekt tipa $c2$
  - $o\#m(x)$ je $m_{(2)}$, kjer je tip vrnjene vrednosti s_2
  - g pričakuje argument tipa $s_1 \Rightarrow s_2 \leq s_1$ je OK

Primer: covariance / contravariance

- Kontravarianca
  - Metoda m pričakoje, da je x tipa t_1
  - Uporabimo o tipa c2 ⇒ uporabi se metoda $m_{(2)}$
  - Ta pričakuje argument tipa t_2, torej je $t_1 \leq t_2$ OK

Ko/Konra/In-varianca v modernih PJ

- Nekateri jeziki zahtevajo natančno isti tip
- Java, C++, C# uporabljajo kovarianten tip rezultata
- Kontravarianca je uporabljena v Python (mypy) in Sather
  - Scala uporablja kontravar. na tipih kolekcij (množica, polje, ...)
- Kontravarianca se zdi nepotrebna
  - Primer: Socialna omrežja
    - $\mathrm{employee\#follow(employee)} \leq \mathrm{person\#follow(person)}$
  - Okolje metod je drugačno od okolja funkcij
- Kovarianca nad parametri se zdi razumna rešitev
  - Eiffel in Dart uporabljata ta pristop
  - Pristop ne zagotavlja varnih tipov

Zamenljivost (ang. substitutivity)

- Princip, ki je tesno povezan z relacijo podtip je zamenljivost.
  - Relacijo A je-podtip B zapišemo $\mathrm{A} \leq \mathrm{B}$
  - Če $\mathrm{A} \leq \mathrm{B}$, potem se lahko pojavi instanca A v vseh okoljih kjer se pričakuje instanca B
    - Funkcija $\mathrm{f} : \mathrm{B} \rightarrow \mathrm{C}$ se lahko aplicira na vseh objektih tipa A, kjer je $\mathrm{A} \leq \mathrm{B}$
    - Instanca tipa $\mathrm{A} \leq \mathrm{B}$ se lahko priredi spremenljivki s tipom B
    - Kolekcija instanc tipa B lahko vsebuje tudi instance tipa A ≤ B

Pretvorba tipa v Ocaml

- Večina objektno-usmerjenih programskih jezikov podpira zameljivost (angl. substitutivity)
  - Ocaml uporablja eksplicitno pretvorbo tipa :>
  - Angl. coerction

```
(name : sub_type :> super_type)
(name :> super_type)
```

- Pretvorba tipa navzgor :>
  - Angl. upcast
  - Objekt tipa colored_point se obravnava kot instanca nadrazreda point
  - Po pretvorbi tipa objekt še vedno pozna originalen tip!
    - … lahko uporablja svoje metode!

```
# let cp = new colored_point (1,1) "red";;
val cp : colored_point = <obj>
# let p = (cp :> point);;
val p : point = <obj>
# p#get_color ();;
Error: This expression has type point
It has no method get_color
# p#to_string ();;
- : string = "(1,1)[red]"
```

is

Pretvorba tipa navzgor/navzdol v Javi

- Java in C++ uporabljata pretvorbe (angl. cast)
  - Implicitna/eksplicitna pretvorba navzgor (upcast)
    - Prireditveni stavek ali eksplicitna pretvorba navzgor
    - Objekt lahko dostopa do specifičnih metod

```
Apple apple = new Apple();
Fruit castedApple = apple;
Fruit castedApple = (Fruit)apple;
```

  - Eksplicitna pretvorba tipa navzdol (downcast)
    - Objekt mora biti pretvorjen navzgor da bi ga lahko pretvorili navzdol!

Vodi do izjeme

```
Fruit fruit = new Fruit();
Apple notApple = (Apple) Fruit;
```

Pravilno

```
Fruit fruit = new Apple();
Apple castedApple = (Apple) fruit;
```

Polimorfizem v izvajalnem času

- Dinamično povezovanje in podtipi so uporabljeni pri izražanju polimorfizma v času izvajanja
- Naj bo C osnovni razred iz katerega so definirani podrazredi S_i
  - Razred C je koren družine razredov S_i
  - Podtipi in zamenljivost omogočajo obravnavo instanc S_i kot instance razreda C
  - Dinamično povezovanje zagotavlja, da bo za dano ime metode in instanco razreda S_i, metoda povezana z razredom S_i
- Polimorfizem v izvajalnem času
  - Izbor metode m poklicane na instanci o razreda S_i je odvisen od implementacije m v S_i
  - Isto sporočilo pošljemo objektom različnih "oblik" (tipov) kar da različno obnašanje

Pregled

1. Uvod
2. Razredi in objekti
3. Dedovanje
4. Dinamično povezovanje
5. Podtipi in zamenljivost
6. Abstraktni razredi
7. Generativnost
8. Popularni OO programski jeziki

Abstraktni razredi

- Pogosto je v OO programih koristno definirati splošne koncepte kot korene družine razredov
  - Splošni koncepti (razredi) so pogosto implementirani kot abstraktni osnovni razredi
    - Za te razrede ne obstaja implementacija
  - Edini namen abstraktnega razreda je služiti kot osnova za druge, konkretne razrede
    - So generalizacija konkretnih razredov neke družine razredov
    - Primeri: kontejner, račun, geo.oblika, vozilo, itd.
- Abstraktni razredi so uporabljeni za organizacijo hierarhije razredov
  - Nimajo primerkov razreda
  - Definirajo skupen vmesnik družine razredov

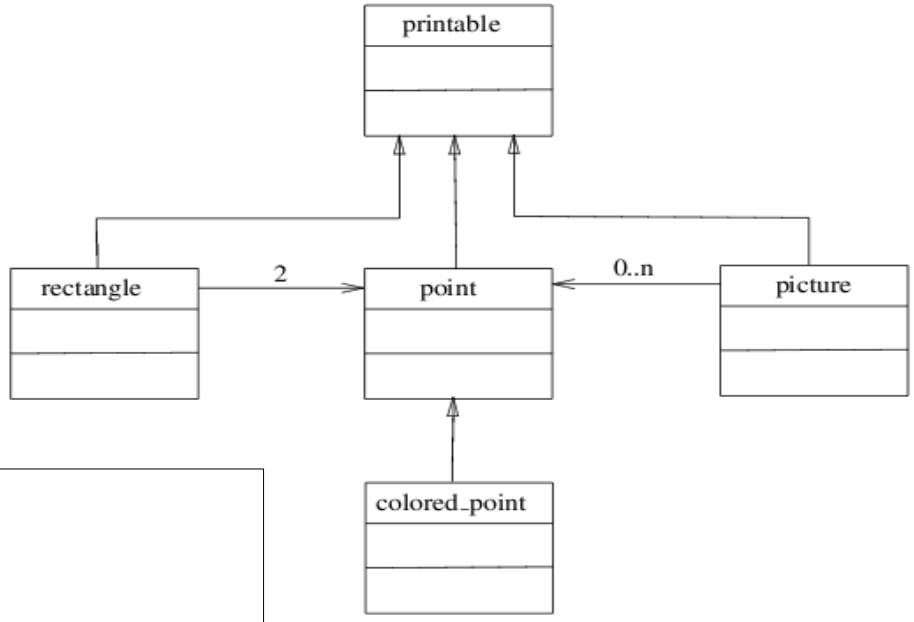
Abstraktni razredi

- Abstraktni razredi vsebujejo virtualne metode
  - Virtualne metode so definirane s tipom metod
  - Razred je abstrakten, če vsebuje vsaj eno virtualno metodo
  - Podrazred abstraktnega razreda implementira virtualne metode in postane konkreten
    - Sicer mora biti podrazred definiran kot abstrakten
  - Abstraktni razredi v Ocaml

```
class virtual name = object . . . end
method virtual name : type
```

- Večkratno dedovanje abstraktnih razredov
  - Če imamo definirano več podatkovnih članov z enakim tipom potem je vidna zadnja definicija

Primer

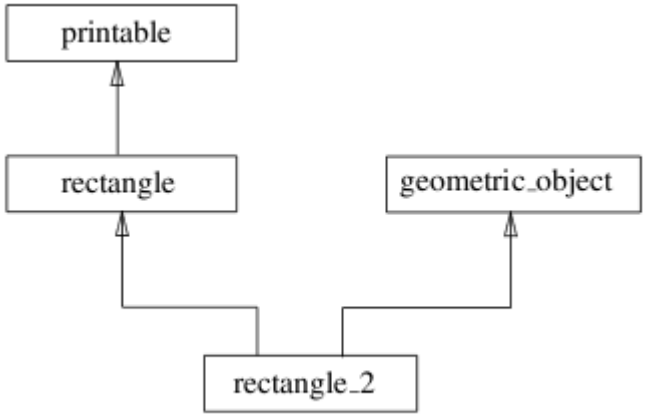



```
# class virtual printable () =
       object(self)
         method virtual to_string : unit -> string
         method print () = print_string (self#to_string () )
       end ;;
# class rectangle (p1,p2) =
      object
        inherit printable ()
        val llc = (p1 : point)
        val urc = (p2 : point)
        method to_string () = "["^llc#t_string ()^","^urc#to_string ()^"]"
      End ;;
# let r = new rectangle (new point (2,3), new point (4,5));;
val r : rectangle = <obj>
# r#print () ;;
[(2,3),(4,5)]- : unit = ()
```

Abstraktna metoda v osnovnem razredu nudi “kavelj” za dinamično povezovaje metode

print () je implementirana z metodo, ki še ni implementirana

Primer 2




```
# class virtual geometric_object () =
    object
      method virtual compute_area : unit -> float
      method virtual compute_peri : unit -> float
    end;;
# class rectangle_2 (p2 :  point * point) =
      object
      inherit rectangle p2
      inherit geometric_object ()
      method compute_area () =
        float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
       method compute_peri () =
        float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
   end;;
```

Večkratno dedovanje od abstraktnih razredov

Pregled

1. Uvod
2. Razredi in objekti
3. Dedovanje
4. Dinamično povezovanje
5. Podtipi in zamenljivost
6. Abstraktni razredi
7. Generativnost
8. Popularni OO programski jeziki

Generativnost

- Generično programiranje
  - Programi na osnovi tipov, ki so določeni kasneje
- Oblike generativnosti
  - Polimorfične funkcije, polimorfični tipi
  - Parametrizirani razredi, funktorji (param. moduli)
- Generativnost razvita v okviru teorije tipov in polimorfičnega lambda računa
  - Pristop se je začel z ML (1972)
    - Definicija polimorfičnih tipov in abstraktnih podatkovnih tipov (APT)
  - Sledi definicija polimorfičnih razredov kjer so tipi parametri.

Parametrizirani razredi

- Parametrični polimorfizem je razširjen na razrede
  - Sintaksa je blizu definiciji parametriziranih tipov
    - Spremenljivke tipov so uporabljene za definicijo razredov.
    - Definicja razreda je v bistvu definicija tipa razreda.

```
class [’a,’b,...] name = object . . . end
```

- Parametrizirani razredi uporabljeni v programskih jezikih
  - Kalupi (template) v C++
  - Generics v Javi, C#
  - Generativnost v Delphi, Haskell, Scala, ...

Primer

Prvi pristop:
ignoriraj tipe.
Slaba ideja?

```
# class pair x0 y0 =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ;;
Characters 6-106:
Some type variables are unbound in this type:
  class pair :
    'a ->
    'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
The method fst has type 'a where 'a is unbound
```

Pravilna definicija

```
# class ['a,'b] pair (x0:'a) (y0:'b) =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ;;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

Parametriziran razred

```
# let p = new pair 2 'X';;
val p : (int, char) pair = <obj>
# p#fst;;
- : int = 2
# let q = new pair 3.12 true;;
val q : (float, bool) pair = <obj>
# q#snd;;
- : bool = true
```

Dedovanje parametriziranih razredov

Definicija razreda z dedovanjem od param. razreda

```
# class point_pair (p1,p2) =
      object
        inherit [point,point] pair p1 p2
      end;;
class point_pair :
   point * point ->
   object
     val x : point
     val y : point
     method fst : point
     method snd : point
  end
```

```
# class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
      object
        inherit ['a,'b] pair x0 y0
        method get1 z = if x = z then y else raise Not_found
        method get2 z = if y = z then x else raise Not_found
      end;;
class ['a, 'b] acc_pair :
   'a ->
   'b ->
   object
     val x : 'a
     val y : 'b
     method fst : 'a
     method get1 : 'a -> 'b
     method get2 : 'b -> 'a
     method snd : 'b
   end
# let p = new acc_pair 3 true;;
val p : (int, bool) acc_pair = <obj>
# p#get1 3;;
- : bool = true
```

Parametriziran razred lahko deduje od param. razreda

Primer: sklad v Ocaml

```
# class ['a] stack =
    object
      val mutable l = ([] : 'a list)
      method push x = l <- x::l
      method pop = match l with
                              [] -> raise Empty |
                              a::l' -> l <- l'; a
      method clear = l <- []
      method length = List.length l
    end;;
  class ['a] stack :
  object
    val mutable l : 'a list
    method clear : unit
    method length : int
    method pop : 'a
    method push : 'a -> unit
  end
```

```
# let s = new stack;;
val s : '_a stack = <obj>
# s#push 1;;
- : unit = ()
# s#push 2;;
- : unit = ()
# s#pop;;
- : int = 2
# s#pop;;
- : int = 1
```

Primer: sklad v Scali

```
class Stack[A] {
  private var elements: List[A] = Nil
  def push(x: A): Unit =
    elements = x :: elements
  def peek: A = elements.head
  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}
```

Pregled

1. Uvod
2. Razredi in objekti
3. Dedovanje
4. Dinamično povezovanje
5. Podtipi in zamenljivost
6. Abstraktni razredi
7. Generativnost
8. Popularni OO programski jeziki

Scala

- Martin Odersky, EPFL, Lausanne, 2003
- Več paradigm
  - Vzporednost, funkcijski, imperativni in objektno-usmerjen jezik
- Izvori
  - Lisp, Eiffel, Erlang, F#, Scheme, Haskell, Java, Ocaml, SML
- Značilnosti
  - Nespremenljivost, Curry oblika, funkcije višjega reda, lena evaluacija, nadaljevanja, polimorfizem, ujemanje vzorcev, algebrajski podatkovni tipi, Curry tipi, kovarianca/kontravarianca, tipi višjega reda, čisti OO jezik, vsaka vrednost je objekt.
- Vzporednost in porazdeljenost
  - Vzporednost: model Actor (iz Erlang), asinhrono programiranje
  - Porazdeljenost: Apache Spark

Google Go

- Statično prever. tipov, prevajalnik visoko-nivojskega PJ
  - Robert Griesemer, Rob Pike, and Ken Thompson, 2007
- Glavne značilnosti
  - Zasnovan za: večjedrnost, stroje v omrežju, in za velike baze programja
  - C sintaksa, varno delo s spominom, čiščenje smeti, strukturno preverjanje tipov, abstraktne podatkovne strukture
- Koncepti Golang
  - Spremenljivke, Konstante, For, If/Else, Switch, Polja, Rezine, Slovarji, Funkcije, Zaprtja, Rekurzija, Kazalci, Strukture, Metode, Vmesniki, Generativnost.
  - Goroutine, Kanali, Asinhrona sporočila, Iztečni čas, Budilke, Števci, Semaforji, Procesi, Signali.
  - Podoben prog. jeziku Erlang (Ericsson); Yahoo dela z Erlang.

C#

- Anders Hejlsberg, Microsoft, 2000
- Implementacija ogrodja .NET
- Več paradigm:
  - Imperativni, deklarativni, funkcijski, generični, objektno-usmerjeni in komponentno usmerjeni programski jezik
- Značilnosti:
  - Strogi tipi, izpeljava tipov.
  - Podatkovne strukture: polja, kolekcije, množice, slovarji, seznami, vrsta, sklad, vreče.
- Razvoj
  - Java, funkcijski gradniki, generičnost, delni tipi, iteratorji, statični razredi, dinam.povezovanje, imenovani argumenti, asinhronost, prevajalnik kot servis, izjeme, izhodne spremenljivke, ujemanje vzorcev, izrazi poizvedb, λ izrazi, itd.

F#

- Don Syme (BDFL), Microsoft Research, 2005
  - ML družina, osnovan na Ocaml
  - Vpliv: C#, Python, Haskell, Scala, in Erlang.
  - Več paradigm jezika
    - Funkcijsko, imperativno, and objektno-usmerjeno programiranje
  - Nekatere značilnosti
    - Strogi tipi, izpeljava tipov, takojšnja evaluacija, zaprtja, lambda izrazi, funkcije višjega reda, ujemanje vzorcov
  - Programski stili
    - Asinshrono, vzporedno, porazdeljeno, meta programiranje
- Implementacija
  - .NET okolje implementira jedro Ocaml-a
  - Uporabljen Common Lang. Infrastructure (CLI), JavaScript in GPU