Oblikovni obrasci u programiranju

Načela programskog oblikovanja

Siniša Šegvić

Sveučilište u Zagrebu

Fakultet elektrotehnike i računarstva

Zavod za elektroniku, mikroelektroniku
računalne i inteligentne sustave

Sadržaj

Načela programskog oblikovanja

- **Simptomi**: urušavanje kvalitete programa
- **Primjer**: problem i rješenje
- **Tehnike**: pregled dijagrama i tehnika programiranja
- Načela **logičkog** oblikovanja
- Načela **fizičkog** oblikovanja
- **Zaključak**

Problem: uzroci

Vidjeli smo da organizacija određuje **dinamička svojstva** programa, odnosno sposobnost projekta za **održivi razvoj**

Zašto je teško program organizirati kako treba ``isprve"?

- početni zahtjevi često su nepotpuni i pogrešni
  (slabo početno znanje o domeni)
- zahtjevi se mijenjaju, posebno ako je program popularan
  (živimo u dinamičnom svijetu)

Rješenje: organizaciju postupno usklađivati sa saznanjima o domeni

- konstruktivnije raditi na rješenju nego tražiti krivicu

Programsko oblikovanje svodi se na traženje brzih i odgovarajućih odgovora na promijenjene okolnosti

- dugoročno planiranje najčešće ne pali u stvarnom svijetu

Problem: simptomi

Koji su **simptomi** programa koji propada, urušava se, ili je naprosto neprikladno organiziran [Martin04]?

- krutost (engl. rigidity): teško nadograđivanje, promjene rezultiraju domino efektom
- krhkost (engl. fragility): lako unošenje suptilnih grešaka
- nepokretnost (engl. immobility): teško višestruko korištenje
- viskoznost (ili trenje, engl. viscosity): **sklonost** k slabljenju integriteta programa
  (uslijed pretjerane složenosti, ponavljanja, ...)

Problem: krutost

Kada je programski sustav **krut**?

- program je teško promijeniti čak i na jednostavne načine, jer svaka promjena zahtijeva nove promjene (domino-efekt)
  - najčešće uslijed dugog lanca **eksplicitne** ovisnosti:
    - A ovisi o B, B ovisi o C, ..., Y ovisi o Z
    - promjena u Z se može propagirati sve do A
  - sitna ``poludnevna'' intervencija može se pretvoriti u višednevni maraton istitravanja promjene kroz sustav
- Posljedica: **strah** od ispravljanja problema koji nisu kritični
  - može doći do pozitivne povratne veze:
    krutost $\rightarrow$ izbjegavanje promjene $\rightarrow$ još veća krutost
- Protiv krutosti se obično borimo kraćenjem lanca ovisnosti primjenom **apstrakcije** i **enkapsulacije**

Problem: krhkost

Kada je programski sustav **krhak**?

- tendencija programa da ``puca'' nakon promjena:
  - uzrok: **implicitna** međuovisnost uslijed ponavljanja (zalihosti)

```
//line.hpp
struct LineSegment{
  double x1;
  double y1;
  double x2,
  double y2;
  double len;
};
```

```
// client code:
LineSegment l;
l.x1=35;
l.y1=25;
l.x2=45;
l.y2=50;
// ouch, forgot to set len...
```

- Komponenta `LineSegment` je krhka, jer tko god promijeni `x1` itd. mora se sjetiti promijeniti i `len` (len je mogao nastati i naknadno!)
  - **jedna** konceptualna izmjena mora se unijeti na **više** mjesta
  - propusti rezultiraju suptilnim greškama koje je teško pronaći

Moguća (vjerojatna) posljedica: cjelodnevni bubolov.

Problem: krhkost (2)

Vidjeli smo da krhke komponente olakšavaju unošenje suptilnih bugova koje statička analiza ne može pronaći.

Krhkost se obično javlja uslijed ponavljanja: u prethodnom primjeru duljina dužine sadržana implicitno u `x1`,`y1`, `x2`,`y2` te eksplicitno u `len`.

Uzroci ponavljanja u izvornom kôdu:

- "neizbježno" (jezik, heterogenost, komentari, dokumentacija)
- nepažljivost ili nestrpljivost (jačaju s ostalim simptomima)
- neusklađenost razvojnog tima (komunikacija!)

Krhkost jača krutost (Y2K fijasko, 3e11 USD)

Pozitivna povratna veza ponovo evoluciju čini teškom
krhkost, krutost $\rightarrow$ izbjegavanje promjene $\rightarrow$ veća krhkost, krutost

Problem: nepokretnost

Nepokretnost programskog sustava:

- otežano višekratno korištenje razvijene funkcionalnosti
  - lakše napisati novi kôd nego koristiti postojeći
- komponente se pišu iznova, umjesto evolucije kroz ponovno korištenje
- čest uzrok: pretjerana međuovisnost zbog neadekvatnih sučelja i neadekvatne razdiobe funkcionalnosti po komponentama
  - nesuđeni novi korisnik otkriva da postojeći modul ima previše ``prtljage'' koju nije lako eliminirati
  - takav primjer predstavlja i razred `Image` (vidi uvodno predavanje) zbog ovisnosti o `libAcmeTiff`
- nepokretnost potiče ponavljanje, odnosno krhkost i krutost

Problem: nepokretnost, npr

```
// Shape.hpp
class Shape{
  ...
}

// MyVector.hpp
class MyVector{
public:
  MyVector(int i);
  Shape* operator[](int i);
  ...
private:
  Shape** data;
};
```

```
//Triangle.hpp
class Triangle:
  public Shape
{
  // ...
};

// client code:
// ...
MyVector X(3);
X[0]=new Triangle; // OK
X[1]=new std::string("burek"); //error!
// MyVector does not work for classes
// which do not derive from Shape
```

Polimorfni vektori u C++-u nisu najsretnije rješenje.

- C++ taj problem rješava predlošcima
- Java: zajednički osnovni tip svih objekata
- Python: implicitno tipiziranje (engl. strong dynamic typing)

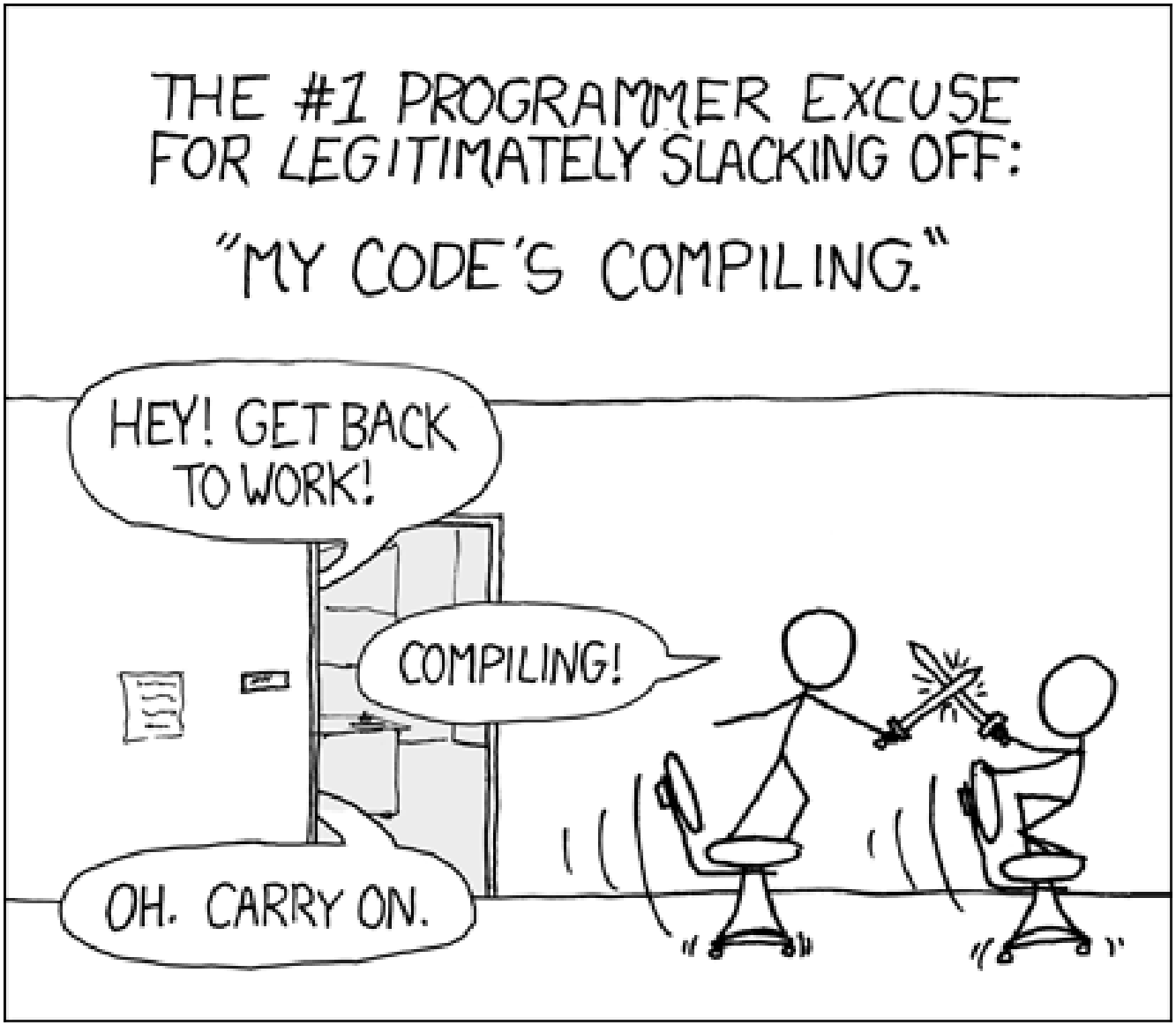
Problem: viskoznost

Programski sustav je **viskozan** kad ga je teško nadograđivati uz očuvanje konceptualnog integriteta programa.

Javljaju se dvije vrste viskoznosti (trenja):

- **viskoznost programske organizacije**: nadogradnje koje čuvaju integritet zahtijevaju puno manualnog rada ili nisu očite
  - promjene je teško unijeti u skladu s originalnom zamisli
  - novu funkcionalnost je lakše dodati na dugoročno loš način
- **viskoznost razvojnog procesa**: spora, neefikasna razvojna okolina
  - npr. komplicirani sustav za verziranje implicira rjeđe sinkronizacije kôda te kasnije otkrivanje problema u vezi s integracijom
  - npr. sporo prevođenje pospješuje unošenje ``zakrpa'' umjesto primjerenog održavanja organizacije

Problem: viskoznost procesa



http://xkcd.com/303/

Problem: viskoznost organizacije, npr

Tipičan kôd koji koristi Windows API:

```
for (DWORD i=0; i<dwInputCount; i++){
  if (FAILED(m_pWMWriter->GetInputProps(i,&pInputProps))){
    SetErrorMessage("Unable to GetInput Properties");
    goto TerminateConstructor;
  }
  if (FAILED(pInputProps->GetType(&guidInputType))){
    SetErrorMessage("Unable to Get Input Property Type");
    goto TerminateConstructor;
  }
  if (guidInputType==WMMEDIATYPE_Video){
    m_pVideoProps=pInputProps;
    m_dwVideoInput=i;
    break;
  }
  else{
    pInputProps->Release();
    pInputProps=NULL;
  }
}
```

Problem: viskoznost organizacije, npr

Dojava grešaka preko povratne vrijednosti pospješuje viskoznost:

- od silnih provjeravanja ne vidimo što program radi!
- ekvivalentni kôd s iznimkama:

```
for i in range(input_count):
  input_props = writer.get_input_props(i)
  if input_props.get_type() == "video":
    self.video_props=input_props
    self.video_index=i
    break
```

Vraćanje stvorenih objekata preko golih pokazivača (u jezicima bez automatskog rukovanja memorijom) pospješuje viskoznost i krhkost:

- nakon `GetInputProps` moramo se sjetiti pozvati `Release`
- new/delete, malloc/free, considered harmful in client code
  - RAII (C++), rukovatelji konteksta (Python)

Problem: sažetak

Zajednički nazivnik **patologije**:

- zbog neadekvatnog oblikovanja ili izmijenjenih zahtjeva dolazi do degradiranja organizacije:
  - neželjene međuovisnosti komponenata ili funkcionalnosti
  - ponavljanje u implementaciji ili organizaciji
  - otežano prenošenje komponente u druge projekte
  - otežano očuvanje integriteta programa
- degradacija organizacije uzrokuje otežanu evoluciju programa
- loša organizacija $\rightarrow$ slaba evolucija $\rightarrow$ loša organizacija $\rightarrow$ ...

Specifični primjeri patologije (anti-obrasci):

- potpuna međuovisnost: ``spaghetti code''
- proizvoljna odgovornost: ``Swiss-Army Knife''
- potpuna nepokretnost: ``reinvent the wheel''

Problem: živo blato

Spirala smrti programskog projekta:
loša organizacija $\rightarrow$ slaba evolucija $\rightarrow$ loša organizacija $\rightarrow$ ...

Takvi programski projekti proživljavaju iskustvo koje se može usporediti s živim blatom [brooks95tmmm]

Recept za spas: redovno prilagođavati organizaciju znanju o domeni u skladu s **načelima oblikovanja**

- vidjet ćemo da se to u mnogome svodi na ograničavanje međuovisnosti komponenata.

Primjer

Pretpostavimo da pišemo korisnički program koji obrađuje video...

U prvoj fazi, testiramo tehnike za pretprocesiranje slike

Treba nam petlja u kojoj ćemo:

- pribaviti sliku iz digitalizatora
- obraditi sliku u prikladnim algoritmom
- iscrtati obrađenu sliku u stvarnom vremenu

Primjer: v1

Zamišljena petlja obrade slike:

```
void mainLoop(){
  ...
  while(1){
    img_wrap image;

    // pribavljamo sliku iz u-i medusklopa...
    grabber.getFrame(image);

    // obradujemo je...
    myAlgorithm.process(image);

    // ... i iscrtavamo rezultate
    img_wrap& imgDst=myAlgorithm.imgDst()
    window.putFrame(imgDst);
  }
}
```

Ali, kako to već biva, poslije je ispalo da bi bilo korisno da slike možemo učitavati i s diska...

Primjer: v2

Nakon omogućavanja čitanja slika s diska:

```
enum EVSource{VSFile,VSGrab}
...
void mainLoop(EVSource myVS){
  while(1){
    img_wrap image;
    switch(myVS){
    case VSFile:
      file.getFrame(image);
      break;
    case VSGrab:
      grabber.getFrame(image);
      break;
    }
    myAlgorithm.process(image);
    window.putFrame(myAlgorithm.imgDst());
  }
```

Međutim, sad vidimo da bi bilo dobro moći spremiti i obrađene slike...

Primjer: v3

Nakon omogućavanja upisa rezultata na disk:

```
enum EVSource{VSFile,VSGrab}
enum EVDest{VDFile,VDWindow}
...
void mainLoop(EVSource myVS, EVDest myVD){
  while(1){
    img_wrap image;
    switch(myVS){
      ...
    }
    myAlgorithm.process(image);
    switch(myVD){
    case VDFile:
      file2.putFrame(myAlgorithm.imgDst());
      break;
    case VDWindow:
      window.putFrame(myAlgorithm.imgDst());
      break;
    }
  }
```

Cool :-)

Primjer: v4?

Međutim, sad bismo htjeli još i različite postupke obrade...

...i različite digitalizatore...

...i različite formate ulaznih slika...

...i prikaz međuslika u postupku obrade...

...moramo stvarati sve objekte za pribavljanje slike
(iako koristimo samo jednog) ...

```
// v4?????
enum EVSource{VSFileAVI,VSFileBMP,... , VSNet,
              VSGrabComet,VSGrab1394,VSGrabMCI}
enum EVDest{VDFileAvi,..., VDWindow,VDNet}
enum EAlgorithm{AlgFColour,AlgFMotion,AlgFSkin,...}
void mainLoop(EVSource myVS, EVDest myVD,
  EAlgorithm alg){
  ...
}
```

Primjer: v4??

Vrlo brzo smo došli do situacije u kojoj program nije moguće lako nadograđivati

Verzija v4 je kruta, viskozna i nepokretna

Ali kako se to dogodilo da od elegantne v1 dođemo do nespretne i glomazne v4?

Promijenili su se zahtjevi!

- veza između programske organizacije i termodinamike: nered uvijek raste

Što ćemo sad?

Naravno, prekrojiti organizaciju (uskladiti je sa znanjem o domeni).

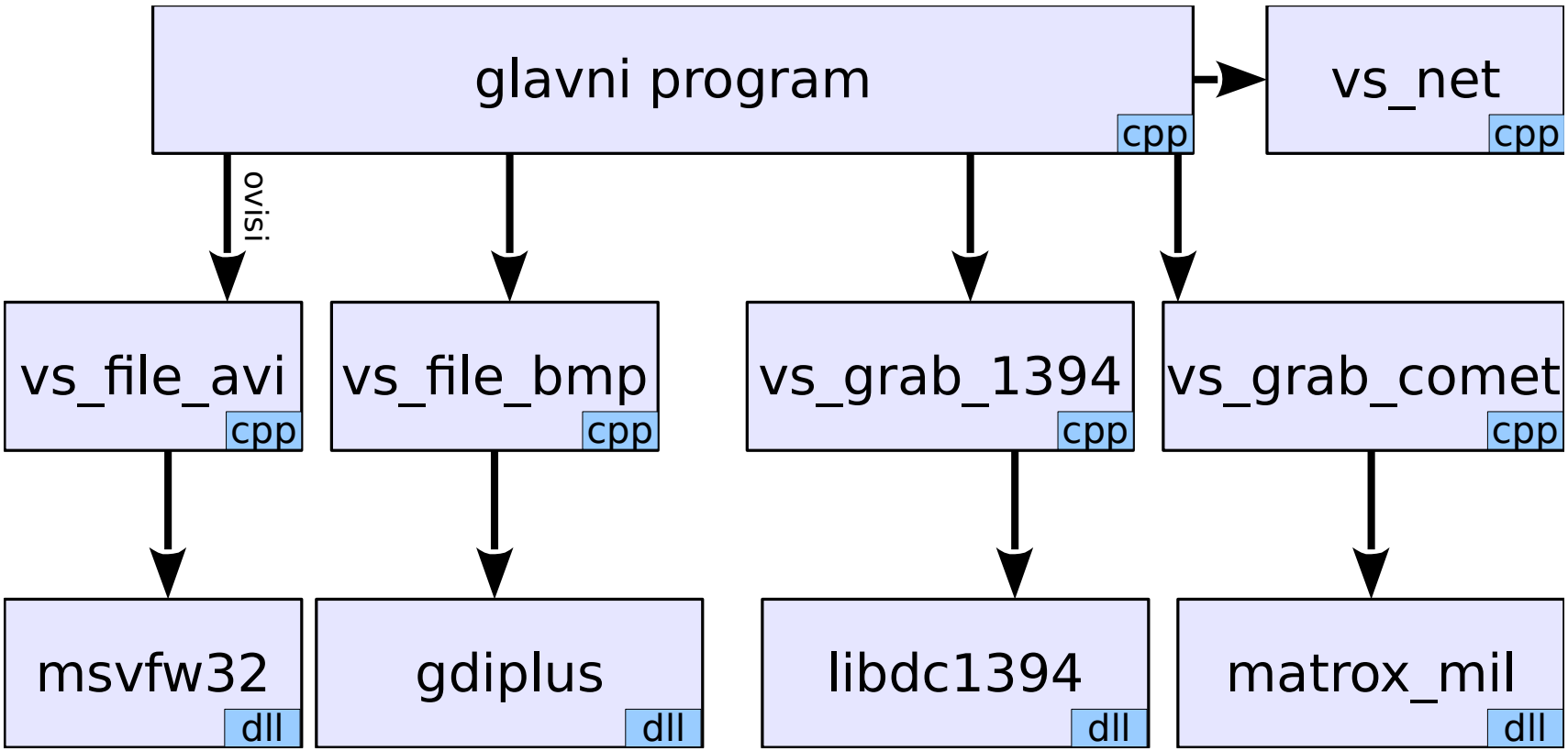
Primjer: v5

Nova petlja koristi (dinamički) **polimorfizam**, i u potpunosti je neovisna o konkretnom pribavljanju, obradi i spremanju slika:

```
#include "vs_base.hpp"  // video source (getFrame)
#include "vd_base.hpp"  // video destination (putFrame)
#include "alg_base.hpp" // image processing algorithm (process)
...
void mainLoop(vs_base& vs, alg_base& algorithm, vd_base& vd){
  std::vector<vd_win> vdWins(algorithm.nDst());
  while(!vs.eof()){
    img_wrap image;
    vs.getFrame(image);
    algorithm.process(image);
    for (int i=0; i<algorithm.nDst(); ++i){
      vdWins[i].putFrame(algorithm.imgDst(i));
    }
    vd.putFrame(algorithm.imgDst(0));
  }
}
```

Primjer: dijagram v3

Početna organizacija: puno komponenata o kojima mnogo toga ovisi
(A ovisi o B ako se A ne može testirati bez B)

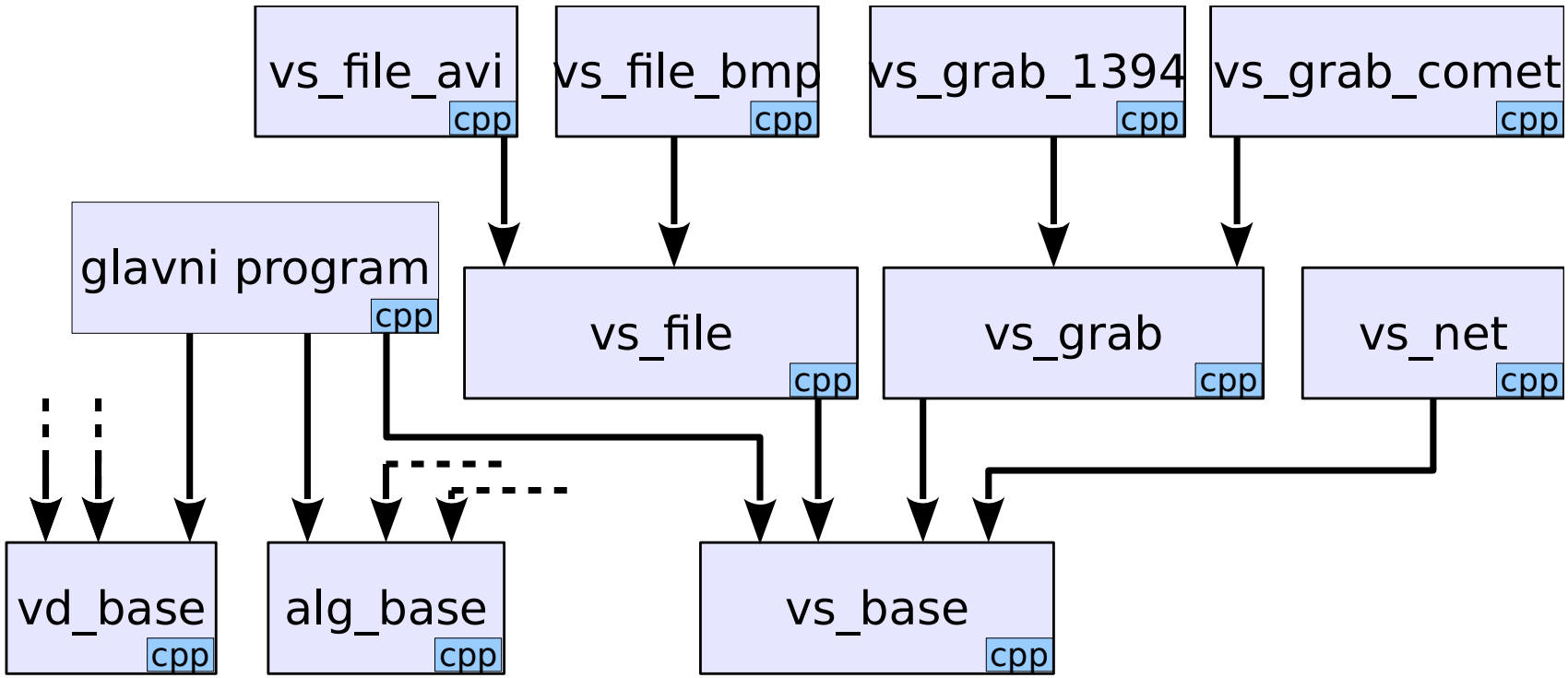


Odabir "tehnoloških" komponenti utječe na glavni program

Glavna komponenta ne može se ni testirati bez <comet.h>!
(nepokretnost, vendor lock-in anti pattern)

Primjer: dijagram v5

Poboljšana organizacija: puno više **neovisnih** komponenata



Sada možemo širiti funkcionalnost **bez prevođenja** glavne komponente

Ovisnost usmjerena od složenog prema apstraktnom (**važna ideja!**)

Smanjen pritisak ovisnosti na glavni dio programa

Tehnike

Cilj nastavne cjeline: kratki pregled tehnika i metoda za razvoj većih programskih sustava

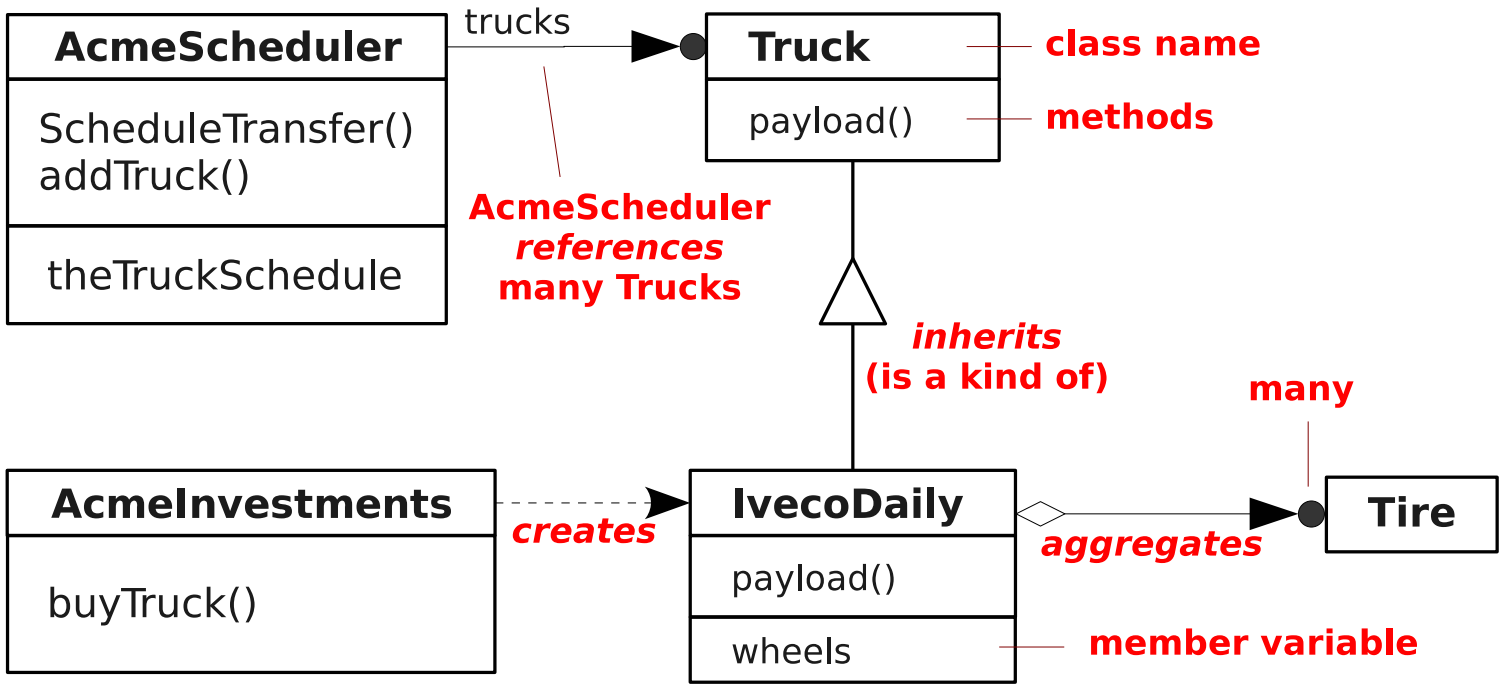
Malo terminologije:

- logičko vs. fizičko oblikovanje:
  - logičko oblikovanje - elementi programskog jezika (moduli: **razredi** i **funkcije**)
  - fizičko oblikovanje - raspodjela funkcionalnosti po datotekama
    - **komponenta** je temeljna jedinica: sastoji se od sučelja (`.hpp`) i implementacije (`.cpp`, `.lib`, `.a`, `.dll`, `.so`)
    - komponenta sadrži jednu ili više logičkih jedinica
    - komponenta: temeljna jedinica pri **verziranju** i **testiranju**!
  - do dobre organizacije dolazimo pažljivim logičkim i fizičkim oblikovanjem

Tehnike: GoF OMT

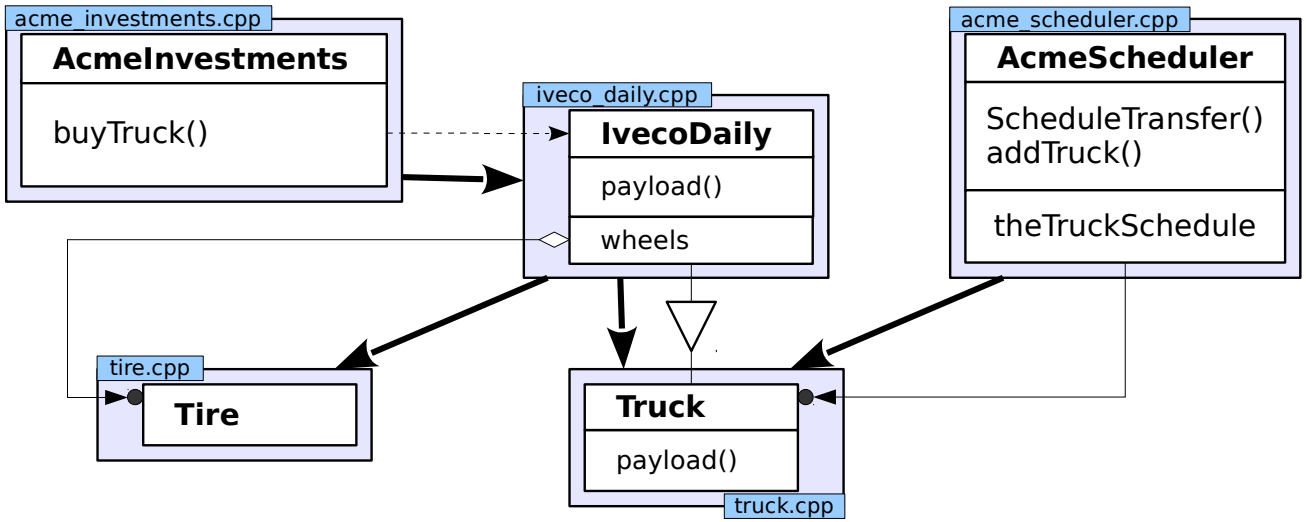
OMT (Object Modelling Technique, 1991): jezik (!) za modeliranje programske podrške (jednostavniji prethodnik UML-a)

Koristimo pojednostavljene dijagrame razreda za opis **logičkih** odnosa: izvodi (nasljeđuje), referencira, sadrži, stvara, ...



Tehnike: Lakos96

OMT ne može prikazati odnose u fizičkom oblikovanju pa uvodimo hibridnu notaciju iz knjige [Lakos96]:



Komponente fizičkog oblikovanja (datoteke izvornog kôda, `.cpp`, `.c`, itd.) prikazujemo sivim pravokutnicima.

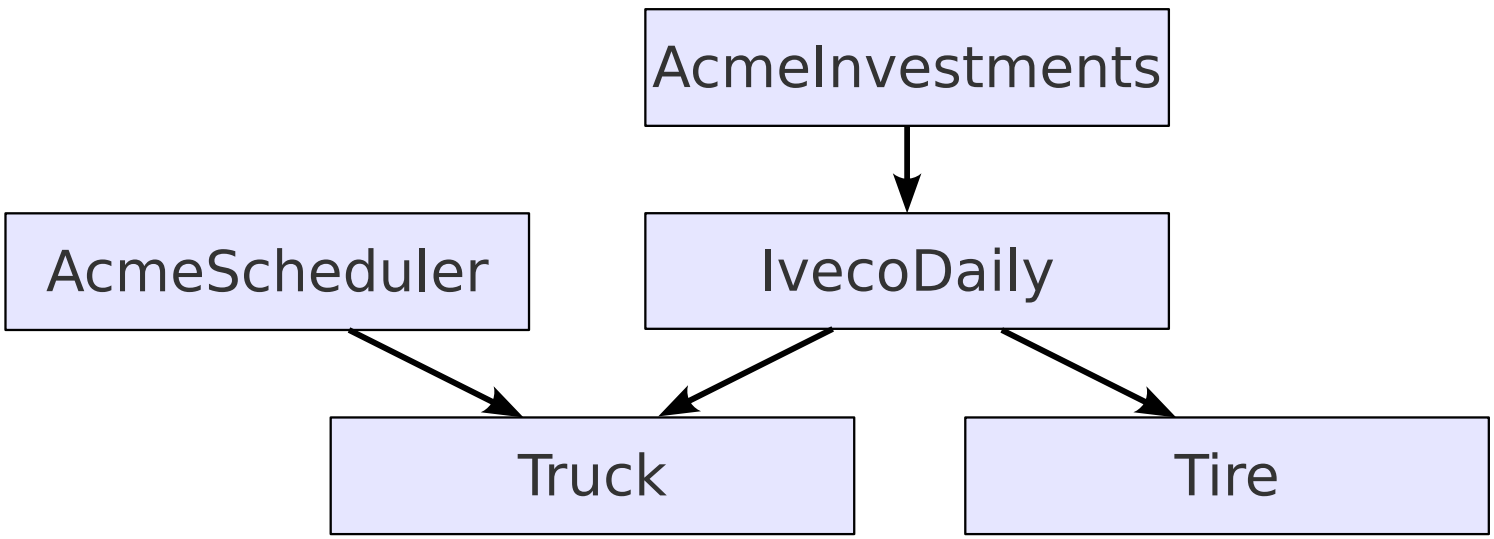
Veze između sivih pravokutnika opisuju ovisnost komponenata

- A ovisi o B ako se A ne može prevesti i ispitati bez B
- ako A koristi lokalnu varijablu tipa B, taj odnos neće utjecati na logički dijagram nego samo na fizički.

Tehnike: Lakos96 (2)

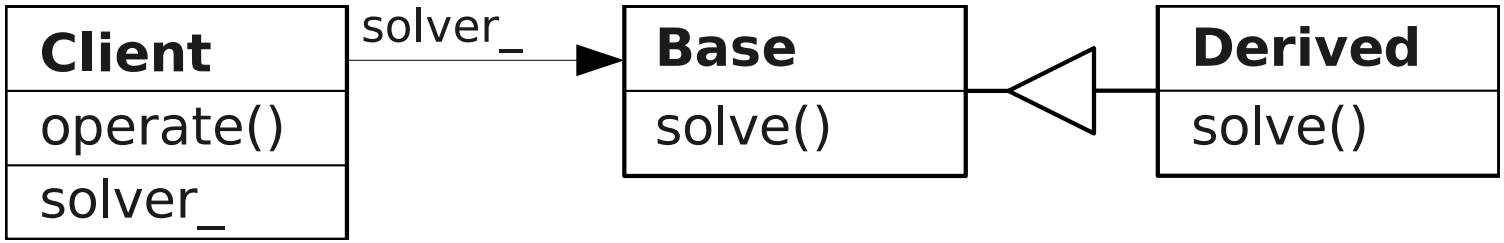
Fokusiranje na **fizičko** oblikovanje pruža vrlo jasan uvid u međusobnu ovisnost komponenata.

Ovisnost komponenata je ključna relacija pri testiranju i ponovnom korištenju komponenata



Tehnike: dinamički polimorfizam

Promotrimo osnovni idiom OO oblikovanja (povjeravanje, delegiranje):



- `Client` delegira dio svog posla metodi `solve` apstaktnog razreda `Base`
- `Client` pozna `Base`, ali ne pozna `Derived`
- objekt tipa `Client` poziva metode razreda o kojem ne ovisi!

Za poziv `solver_.solve()` kažemo da je polimorfan jer odredište u trenutku pisanja programa **nije poznato**

- preciznije, radi se o dinamičkom polimorfizmu jer odredište postaje poznato tek u trenutku izvođenja
- polimorfizam je ključni koncept OO oblikovanja

Tehnike: dinamički polimorfizam u C++-u

Izvedimo minimalni OO idiom u C++-u

- preferiramo C++ zbog sljedećih didaktičkih prednosti
  - omogućava usporedbu virtualnih i ne-virtualnih metoda
  - možemo ga usporediti s C-om i tako vidjeti prednosti OO dizajna
  - prevodi se u strojni kod koji možemo analizirati
- koristit ćemo minimalni podskup C++-a
  - razredi, obične metode, virtualne metode, statičke metode
  - kasnije ćemo pokazati i minimalni podskup predložaka

Tehnike: dinamički polimorfizam u C++-u (2)

Pokažimo za početak sučelje i implementaciju komponente `Client`

- može se odvojeno prevesti s `g++ -c client.cpp`

```
//==== client.hpp
class Base;

class Client{
  Base& solver_;
public:
  Client(Base& b);
  void operate();
};

.
```

```
//==== client.cpp
#include "base.hpp"
#include "client.hpp"
#include <iostream>

Client::Client(Base& b):
  solver_(b) {}

void Client::operate(){
  std::cout <<solver_.solve() <<"\n";
}
```

Tehnike: dinamički polimorfizam u C++-u (3)

Razred `Base` definira čistu virtualnu metodu `solve` i prazni destruktor

Razred `Derived` nasljeđuje razred `Base` i definira virtualnu metodu `solve`.

```
//==== base.hpp
class Base{
public:
  virtual ~Base();
  virtual int solve()=0;
};
```

```
//==== base.cpp
#include "base.hpp"

Base::~Base(){}

.
```

```
//==== derived.hpp
#include "base.hpp"

class Derived:
  public Base
{
public:
  virtual int solve();
};
```

```
//==== derived.cpp
#include "derived.hpp"

int Derived::solve(){
  return 42;
}
```

Tehnike: dinamički polimorfizam u C++-u (4)

Komponenta `test.cpp` sadrži glavni program koji:

- instancira objekt tipa `Derived` i naziva ga `d`
- instancira objekt tipa `Client`, naziva ga `c` te mu u konstruktor šalje `d`
- poziva metodu `operate` nad objektom `c`

```
//==== test.cpp
#include "client.hpp"
#include "derived.hpp"

int main(){
  Derived d;
  Client c(d);
  c.operate();
}
```

Tehnike: dinamički polimorfizam u C++-u (5)

Prevođenje provodimo datoteku po datoteku:

```
g++ -c test.cpp
g++ -c client.cpp
g++ -c derived.cpp
g++ -c base.cpp
g++ test.o client.o derived.o base.o
```

Svaka komponenta zasebno se prevodi kao da je sama u svemiru.

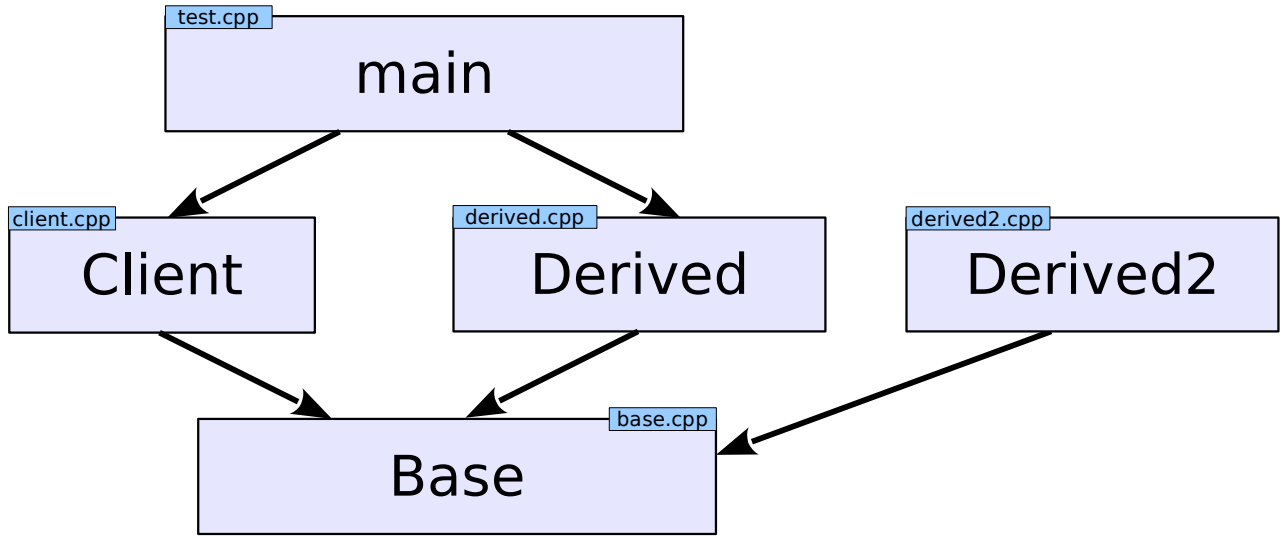
Izvršni kod možemo pokrenuti s `./a.out`

Prevođenje možemo zadati i jednom naredbom (ona iza zavjese također prevodi datoteke jednu po jednu):

```
g++ test.cpp client.cpp derived.cpp base.cpp
```

Tehnike: dinamički polimorfizam - fizički pogled

Fizička organizacija prethodnog primjera:



Ako vam se ovo čini prebanalno, razmotrite supstituciju:

- Client → FirefoxRenderer
- Base → ExtensionInterface
- Derived → ExtensionJava
- Derived2 → ExtensionH264

Tehnike: pozivi metoda: C++ -> C

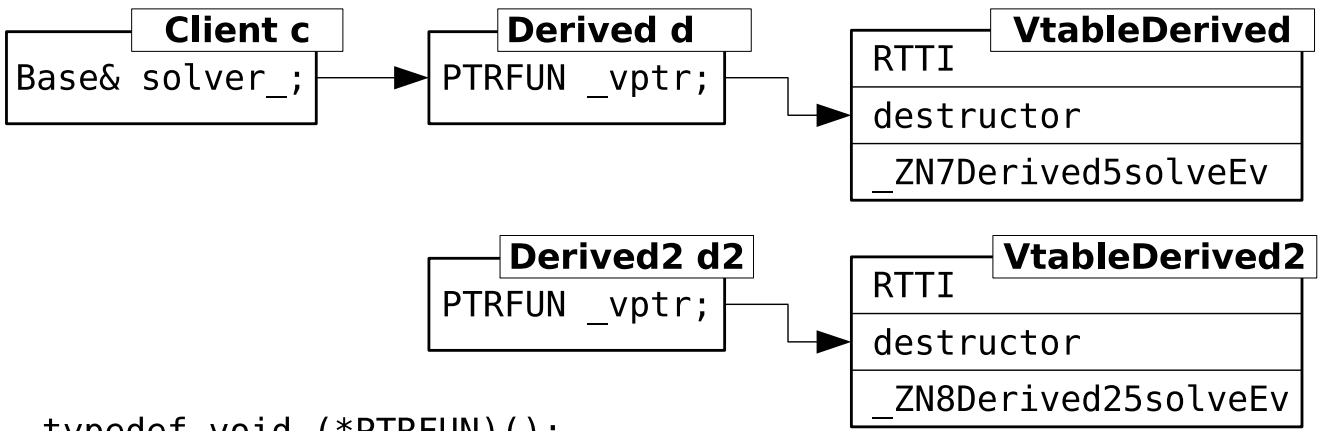
Demistificirani **objektni model** C++-a [Lippman96]:

U kakav pseudo-C kod se pretvara poziv `c.operate()`?

- `_ZN6Client7operateEv(&c);`

U kakav pseudo-C kod se pretvara poziv `solver_.solve()`?

- `(solver_.vptr[1])(&solver_);`




```
typedef void (*PTRFUN)();
```

Stan Lippman: Inside the C++ object model
https://itanium-cxx-abi.github.io/cxx-abi/cxx-vtable-ex.html

Tehnike: pozivi metoda: C++ -> strojni kod

Kako dobiti strojni kod g++-om?

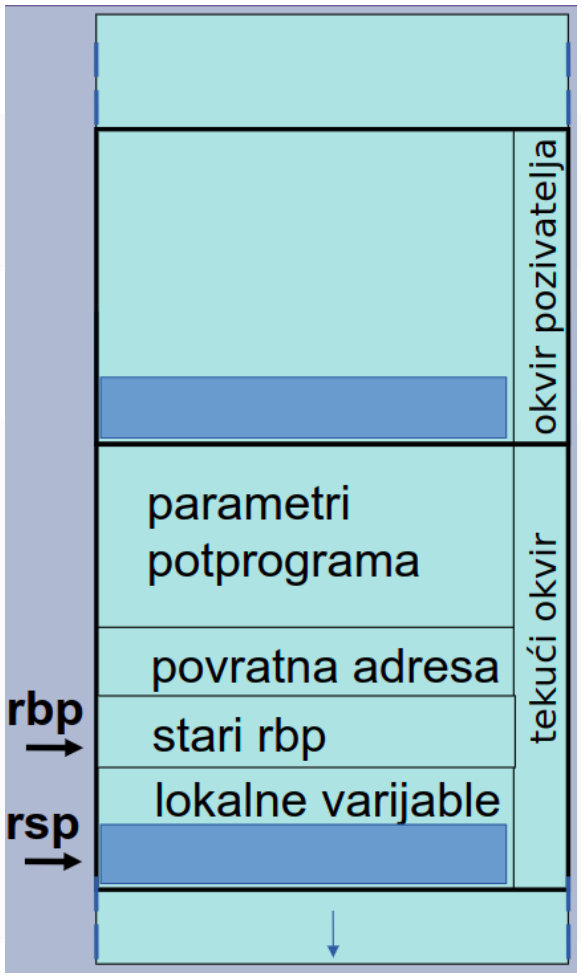
```
$ g++ -S -masm=intel test.cpp
$ g++ -S -masm=intel client.cpp
```

Kakav strojni kod daje poziv `c.operate()`?

```
    lea     rax, -32[rbp]
    mov     rdi, rax
    call    _ZN6Client7operateEv@PLT
```

Kakav strojni kod daje `Client::operate`?

```
    mov     QWORD PTR -8[rbp], rdi
    mov     rax, QWORD PTR -8[rbp]
    mov     rax, QWORD PTR [rax]
    mov     rax, QWORD PTR [rax]
    add     rax, 16
    mov     rdx, QWORD PTR [rax]
    mov     rax, QWORD PTR -8[rbp]
    mov     rax, QWORD PTR [rax]
    mov     rdi, rax
    call    rdx
```



Tehnike: DP, C++

Zadan je razred A u C++-u s dvije virtualne funkcije i jednim podatkovnim članom tipa `int`. Koliko će mjesta na stogu 32-bitne arhitekture zauzeti lokalno polje od 100 objekata tipa A?

Dinamički polimorfizam u C-u može se ostvariti:

- dinamički polimorfizam u C-u nije moguće ostvariti
- korištenjem tablice pokazivača na funkcije
- korištenjem pretprocesorskih makroa
- pozivanjem regularnih funkcija C-a
- korištenjem vanjskih biblioteka

Tehnike: Python

And now for something completely different:

```
class Base:                 class Client:
  def solve(self):            def __init__(self, solver):
    return -1                   self.solver_=solver
                                def operate(self):
class Derived(Base):              print(self.solver_.solve())
  def solve(self):          # this works as expected:
    return 42               d = Derived()
                            c = Client(d)
class NonDerived:           c.operate()
  def solve(self):          # this also works (duck typing!):
    return 0                c2 = Client(NonDerived()); c2.operate()
```

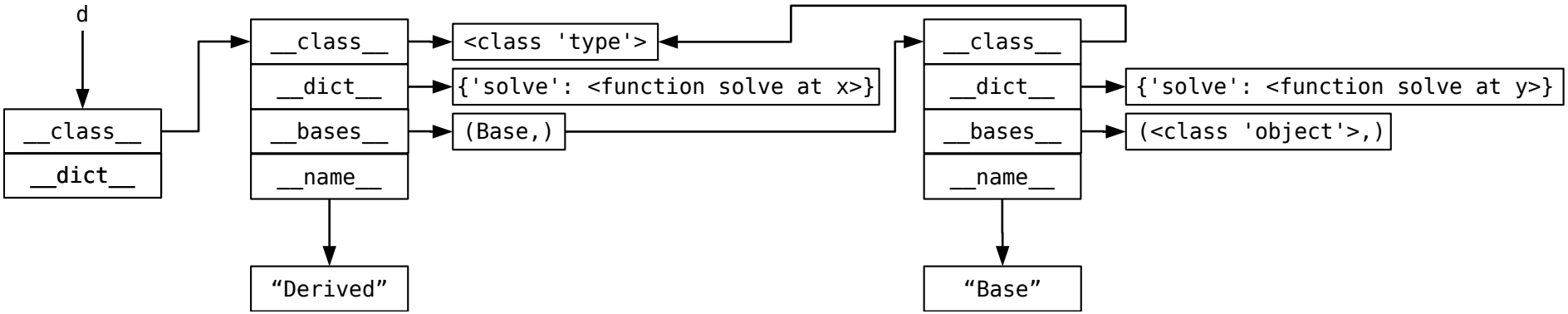
Što se događa kod prvog poziva `solve()`?

- asocijativni pristup rječniku objekta `d`
- u slučaju neuspjeha, asocijativni pristup rječniku razreda `Derived`
- u slučaju neuspjeha, asocijativni pristup rječniku roditelja (`Base`)

Vidimo da poziv metode može rezultirati prozivanjem n rječnika!

Tehnike: Python (2)

Objektni model Pythona: prikaz objekta `d` razreda `Derived`:




```
>>> type(c)
<class 'Client'>
>>> c.__class__.__name__
'Client'
>>> d.__class__.__name__
'Derived'
>>> d.__class__.__bases__
(<class 'Base'>,)
>>> d.__class__.__class__
<class 'type'>
```

```
>>> d.__class__.__class__.__class__
<class 'type'>
>>> d.__class__.__bases__[0].__bases__
(<class 'object'>,)
>>> d.__class__.__bases__[0].__name__
'Base'
>>> d.__class__.__dict__['solve'](d)
42
>>> d.__class__.__bases__[0].__dict__['solve'](d)
-1
```

Tehnike: Python vs C++, Java

Prednosti u odnosu na prozivanje virtualnih tablica:

- fleksibilnija primjena (duck typing)
- mogućnost dodavanja metoda tijekom izvođenja (!)

```
class Cat:
  def greet(self):
    print("mijau!")
  def hit(self):
    print("cap!")

def action(cat):
  cat.greet()
  cat.hit()

ofelija = Cat()
action(ofelija)
```

```
# dodavanje metode razredu
Cat.action=action
ofelija.action()
# mijau! cap!

# dodavanje metode objektu
from types import MethodType
m = MethodType(action, ofelija)
m() # mijau! cap!

ofelija.act = m
ofelija.act() # mijau! cap!
```

Tehnike: Python vs C++, Java (2)

Nedostatci u odnosu na prozivanje virtualnih tablica:

- sporije izvođenje (zbog pretraživanja stabla nasljeđivanja), ali:
  - brzina poziva se poboljšava cacheiranjem i JIT-om
- virtualne metode također nisu besplatne:
  - povećavaju memorijski otisak polimorfnih objekata
  - onemogućuju ugrađivanje (eng. inline) funkcijskih poziva!

Postoji li polimorfizam koji je jednako brz kao i ožičeni poziv?

- ako odustanemo od polimorfnosti izvršnog koda, možemo spojiti Pythonsku fleksibilnost i brzinu C-a
- statički polimorfizam, predlošci u C++-u

Tehnike: predlošci

Statički polimorfizam: odluku o odredištu poziva povjeriti prevoditelju

Primjer: funkcijski predložak `action` parametriziramo razredom `Animal`:

```
class Cat{
public:
  void greet(){
    std::cout <<"mijau\n";
  }

  void hit(){
    std::cout <<"cap!\n";
  }
};
```

```
template <typename Animal>
void action(Animal& a){
  a.greet();
  a.hit();
}

int main(){
  Cat branka;
  action<Cat>(branka);
}
```

Predložak možemo pozvati s objektom (skoro) **proizvoljnog** razreda.

Jedini zahtjev na parametar predloška je da ima metode `greet` i `hit`!

Nedostatak: magija funkcionira samo unutar jedinice prevođenja

- izmjena parametra predloška zahtijeva novo prevođenje

Tehnike: predlošci (2)

Uvodni primjer izveden statičkim polimorfizmom:

```
//==== client.hpp
template <typename Solver>
class Client{
public:
   Client(Solver &s);
   void operate();
   Solver& solver_;
};

template <typename Solver>
Client<Solver>::Client(Solver &s):
  solver_(s){}

template <typename Solver>
void Client<Solver>::operate(){
  std::cout <<solver_.solve()
   <<"\n";
}
```

```
//==== mysolver.hpp
class MySolver{
public :
  int solve();
};

//==== mysolver.cpp
int MySolver::solve(){
  return 42;
}

//==== test.cpp
int main (){
  MySolver s;
  Client<MySolver> c(s);
  c.operate();
}
```

Jedini zahtjev na parametar predloška je da ima metodu `solve`!

Tehnike: predlošci (3)

Predlošcima možemo apstrahirati i funkcionalnost i podatke:

```
template <typename T>
inline T mymax(T x, T y) {
  if (x < y) return y;
  else return x;
}

int main(){
  std::cout <<mymax(3, 7) <<","
    <<mymax(std::string("alpha"), std::string("beta")) <<","
    <<mymax(3.1, 7.1) <<"\n";
}
```

Prikazani kôd generira **tri** asemblerske izvedbe predloška.

Predložak `mymax` možemo primijeniti gdje god je definiran poredak

- ovo ponašanje je slično dinamičkim jezicima npr. Pythonu.

U odnosu na makro C-a: veća sigurnost (cf. `mymax(++i, fun())`), striktno tipiziranje, jednaka učinkovitost, veća izražajnost

Tehnike: predlošci (4)

Proširene gramatike za **parametriziranje** funkcija i razreda:

- CLU (1974), Ada (1977), C++ (1994), Haskell (2001).

Prevođenje se **odgađa** do trenutka kad parametri postaju poznati (nakon instanciranja predloška koristi se **osnovna** gramatika).

Mogućnost **metaprogramiranja** (sličan cilj kao `constexpr`, ali moćnije):

```
template <int n>
int factorial() {
  return n * factorial<n-1>();
}

template <>
int factorial<0>() {return 1;} }
```

```
#include <iostream>
int main(){
  // the line below compiles to:
  // mov DWORD PTR [esp+4], 3628800
  int x=factorial<10>();
  std::cout <<"10! =" <<x <<"\n";
```

Posebno prikladno za biblioteke u statički tipiziranim jezicima

- npr. STL (Stepanov 1981-1994): i učinkovitost i prilagodljivost

Tehnike: STL

Ortogonalnost (nema međuovisnosti) algoritama i spremnika:

- algoritam `reverse` možemo zvati nad vektorom, poljem i listom

```
// ...
int main(){
  std::vector<int> v(3);
  v[0] = 7; v[1] = 3; v[2] = 5;
  std::reverse(v.begin(), v.end()); //vektor
  for (int i = 0; i < v.size(); ++i)
    std::cout << "v[" << i << "] = " << v[i] <<"\n";

  double A[] = { 1.2, 1.3, 1.4, 1.5, 1.6};
  int n = sizeof(A)/sizeof(*A);
  std::reverse(A, A + n); //polje

  std::list<double> L(A, A + n);
  std::reverse(L.begin(), L.end()); //lista
  std::list<double>::iterator it = L.begin();
  while (it!=L.end())
    std::cout <<"  " <<*it++ <<"\n";
}
```

Tehnike: STL (2)

Evo i implementacije iz /usr/include/c++/9.3.0/bits/stl_algo.h:

```
/**
 *  This is an uglified reverse(_BidirectionalIterator,
 *                              _BidirectionalIterator)
 *  overloaded for random access iterators.
*/
template<typename _RandomAccessIterator>
  void
  __reverse(_RandomAccessIterator __first,
            _RandomAccessIterator __last,
            random_access_iterator_tag)
  {
    if (__first == __last)
      return;
    --__last;
    while (__first < __last)
      {
        std::iter_swap(__first, __last);
        ++__first;
        --__last;
      }
  }
```

Tehnike: STL (3)

Treći argument predloška `reverse` daje nam predložak `iterator_traits`

- taj argument bira odgovarajuću izvedbu (random ili bidirectional)

```
// stl_algo.h
template<typename _BidirectionalIterator>
  constexpr inline void
  reverse(_BidirectionalIterator __first,
          _BidirectionalIterator __last)
  {
    // ...
    std::__reverse(__first, __last,
        std::__iterator_category(__first));
  }

// stl_iterator_base_types.h
template<typename _Iter>
  inline constexpr
  typename iterator_traits<_Iter>::iterator_category
  __iterator_category(const _Iter&)
  {
    return typename iterator_traits<_Iter>::iterator_category();
  }
```

Tehnike: STL: iterator_category

Kako `iterator_traits<std::vector<int>::iterator>::iterator_category()` vrati vrijednost tipa `random_access_iterator_tag`?

- ustanovimo prvo da `iterator_traits<T>::iterator_category` vraća svojstva koja su definirana u iteratorima
- zgodno: svaki iterator može reći što može
  - random, bidirectional, forward, input, output

```
// stl_iterator_base_types.h
template<typename _Iterator>
  struct iterator_traits
  {
    typedef typename _Iterator::iterator_category iterator_category;
    typedef typename _Iterator::value_type        value_type;
    typedef typename _Iterator::difference_type   difference_type;
    typedef typename _Iterator::pointer           pointer;
    typedef typename _Iterator::reference         reference;
  };
```

Tehnike: STL: iterator_category (2)

Primijetimo da je `std::vector<int>::iterator` izveden kao `normal_iterator<pointer,vector>`:

- ima smisla: taj isti predložak koristit ćemo i u npr. `std::deque<T>`
- pogledajmo kako izgleda `normal_iterator<pointer,vector>`!

```
// stl_vector.h
template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
  class vector : protected _Vector_base<_Tp, _Alloc>
  {
    typedef _Vector_base<_Tp, _Alloc> _Base;
    typedef typename _Base::pointer pointer;
    typedef __normal_iterator<pointer, vector> iterator;
    // ...
  }
```

Tehnike: STL: iterator_category (3)

Primijetimo da `normal_iterator<pointer,vector>` ne definira svoj `iterator_category`:

- umjesto toga, preuzimaju se svojstva iz `std::iterator_traits<pointer>`
- ima smisla: ionako ćemo trebati `std::iterator_traits<T*>` jer želimo da se pokazivači mogu koristiti kao iteratori!

```
// stl_iterator.h
template<typename _Iterator, typename _Container>
  class __normal_iterator
  {
  protected:
    _Iterator _M_current;
    typedef std::iterator_traits<_Iterator>
        __traits_type;
    typedef typename __traits_type::iterator_category
        iterator_category;
  }
```

Tehnike: STL: iterator_category (4)

Konačno, kategoriju iteratora razreda vektor određuje specijalizacija predloška `std::iterator_traits` za pokazivače:

- u istoj datoteci pronalazimo i definiciju tipa `random_access_iterator_tag`
- hijerarhija omogućava primjenu naprednih iteratora u jednostavnijim algoritmima

```
// stl_iterator_base_types.h
template<typename _Tp>
  struct iterator_traits<_Tp*>
  {
    using iterator_category = random_access_iterator_tag;
  };

struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag :
  public input_iterator_tag { };
struct bidirectional_iterator_tag :
  public forward_iterator_tag { };
struct random_access_iterator_tag :
  public bidirectional_iterator_tag { };
```

Tehnike: statički vs dinamički polimorfizam

Odnos složenosti poziva predloška i virtualnog poziva:

- poziv predloška tipično ima povoljniju složenost:
  - **manja vremenska složenost** jer se poziv predloška razrješava tijekom prevođenja
    - nema potreba za prozivanjem virtualne tablice
    - moguće je i ugrađivanje funkcijskog poziva u kôd klijenta
  - **manja prostorna složenost** (isti razlog kao gore)
    - argument poziva ne mora imati pokazivač na virtualnu tablicu
    - argument poziva može biti elementarni podatkovni tip (npr. int ili char*)
- predlošci mogu dovesti do većeg izvršnog kôda (ali mjesta na disku ovih dana obično ima)
- nakon prevođenja fleksibilnost predložaka nestaje
- predlošci prikladni za manje, često korištene programske jedinice

Tehnike: statički vs dinamički polimorfizam (2)

Zadan je klijentski predložak koji delegira apstraktnom pružatelju `op`:

- poziv `op.process` virtualan $\rightarrow$ dinamički polimorfizam
- poziv `op.process` izravan $\rightarrow$ statički polimorfizam
- omjer vremena pokazuje relativnu cijenu virtualnog poziva.

```
// benchmark.cxx
template <typename T>
double process_data(T& op,
    const std::vector<double>& data,
    double& result)
{
  auto start = std::chrono::steady_clock::now();

  for (size_t i = 0; i < data.size(); ++i) {
    result += op.process(data[i]);
  }

  auto end = std::chrono::steady_clock::now();
  return (end - start).count();
}
```

Tehnike: statički vs dinamički polimorfizam (3)

Sučelja virtualnih pružatelja definiraju metodu `process`:

```
//alg.hxx
class AlgBase{
public:
  virtual double process(double value) = 0;
  virtual ~AlgBase() = default;
};

class AlgAdd: public AlgBase{
  double val_;
public:
  AlgAdd(double value);
  double process(double value) override;
};

class AlgMul: public AlgBase {
  double val_;
public:
  AlgMul(double value);
  double process(double value) override;
};
```

Tehnike: statički vs dinamički polimorfizam (4)

Minimalne izvedbe virtualnih pružatelja:

```
//alg.cxx
#include "alg.hxx"

AlgAdd::AlgAdd(double value):
  val_(value)
{}
double AlgAdd::process(double value){
  return value + val_;
}

AlgMul::AlgMul(double value):
  val_(value)
{}

double AlgMul::process(double value){
  return value * val_;
}
```

Tehnike: statički vs dinamički polimorfizam (5)

Jednostavni pružatelji definiraju iste metode ali bez ključne riječi **virtual**:

```
//benchmark.cxx
class AddSimple {
private:
  double val_;
public:
  AddSimple(double value): val_(value) {}
  double process(double value) {
      return value + val_;
  }
};

class MulSimple{
private:
  double val_;
public:
  MulSimple(double value): val_(value) {}
  double process(double value){
    return value * val_;
  }
};
```

Tehnike: statički vs dinamički polimorfizam (6)

Mjerna funkcija poziva klijentski predložak `process_data` s dva virtualna i dva jednostavna pružatelja:

```
//benchmark.cxx
int benchmark(
    const std::vector<double>& data,
    AlgBase& ab1,
    AlgBase& ab2)
{
  double result = 0;

  // benchmark virtual calls
  double timeVirtualAdd = process_data(ab1, data, result);
  double timeVirtualMul = process_data(ab2, data, result);

  // benchmark template calls
  AddSimple add(5.2);
  MulSimple mul(3.1);
  double timeTemplateAdd = process_data(add, data, result);
  double timeTemplateMul = process_data(mul, data, result);
  //...
}
```

Tehnike: statički vs dinamički polimorfizam (7)

Glavni program stvara argumente i poziva mjernu funkciju:

```
//test.cxx
int main() {
  const size_t long DATA_SIZE = 1000*1000*1000;
  std::vector<double> data(DATA_SIZE);

  AlgAdd add_virt(5.2);
  AlgMul mul_virt(3.1);

  // Generate random data
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_real_distribution<> dis(0.0, 1.0);
  for (size_t i = 0; i < data.size(); ++i) {
      data[i] = dis(gen);
  }

  benchmark(data, add_virt, mul_virt);

  return 0;
}
```

Tehnike: statički vs dinamički polimorfizam (8)

Rezultati mjerenja pokazuju da je virtualni poziv 4x sporiji:

- statički polimorfizam je brži zbog **ugrađivanja** poziva `op.process`;
- dinamičke pozive nije moguće ugraditi jer su definirani u drugoj komponenti
- statički polimorfizam ne može pozivati konkretne implementacije iz drugih komponenata.

```
Data size: 1000000000 elements
--------------------------------
Virtual function (Add)      : 2.40119 ns
Template function (Add)     : 0.627777 ns
Speed improvement (Add)     : 3.82491x
--------------------------------
Virtual function (Multiply): 2.40917 ns
Template function (Multiply): 0.599501 ns
Speed improvement (Multiply): 4.01862x
```

Tehnike: iznimke

Programi susreću iznimne okolnosti pri kontaktu sa stvarnim svijetom:

- interakcija s korisnicima: neispravni ulaz, nepostojeća datoteka
- usluge OS-a: nedozvoljeni pristup, puni disk, iscrpljena memorija
- povezivanje: nedostupni poslužitelj, nekompatibilni protokol

```
void process(const std::string& fn1, const std::string& fn2){
  Image img;
  read_png(fn1, img);
  rotate(img);
  write_png(fn2, img);
};

void read_png(const std::string& fname, Image& img){
  // ...
  if (parseHeaderFailed) {
    throw std::runtime_error(
      "read_png: " + fname + " improper header!");
  }
  // ...
};
```

Tehnike: iznimke - gdje hvatati

Specijalni slučajevi zahtijevaju kompromise:

- vrlo važno ih je detektirati i obraditi
- ne želimo umanjiti jasnoću glavnog toka
- iznimke su jezični mehanizam izbora za tu potrebu!

Osnovno pravilo iznimki je: **bacaj nisko, hvataj visoko**

- problemi se tipično detektiraju u kôdu na niskoj razini apstrakcije
- primjeren oporavak provediv samo na visokoj razini apstrakcije:
  - taj kôd zna procijeniti ozbiljnost problema i komunicirati s korisnikom
  - jesmo li na robotu, pregledniku, poslužitelju, mobitelu, ...?

```
int main(int argc, char *argv[]) {
  // check whether argc==3...
  try {
    process(std::string(argv[1]), std::string(argv[2]));
  } catch (const std::exception& e) {
    std::cerr << "Main: " << e.what() << std::endl;
  }
}
```

Tehnike: iznimke - biblioteke

Ponekad biblioteke trebamo ohrabriti da koriste iznimke:

- puno čišće nego testirati uspjeh i baciti našu iznimku
- takav pristup omogućava jasan kôd i prirodnu podjelu odgovornosti

```
std::string getFirstLine(const std::string& filename) {
  std::ifstream file;
  file.exceptions(
    std::ifstream::failbit |
    std::ifstream::badbit);
  file.open(filename); // can throw!

  std::string line;
  return std::getline(file, line); // can throw!
}
```

Tehnike: iznimke - strukturirano hvatanje

Iznimke možemo sortirati pri hvatanju:

- ima smisla: različite iznimke traže različite procedure oporavka
- takav pristup umanjuje potrebu za redundantnim testiranjem
- preporuka: bacati primjerke **specifičnih tipova** koji nasljeđuju `std::exception`

```
int main() {
  try {
    getFirstLine("example.txt");
  } catch (const std::ios_base::failure& e) {
      std::cerr << "File I/O error: " << e.what() << std::endl;
      return 1;
  } catch (const std::exception& e) {
      std::cerr << "Caught exception: " << e.what() << std::endl;
      return 2;
  } catch (...) {
      std::cerr << "Unknown error occurred." << std::endl;
      return 3;
  }
  return 0;
}
```

Tehnike: iznimke - EAFP

Iznimke omogućuju izravnije izražavanje:

- umjesto testiranja platforme, izravno izraziti namjeru
  - u primjeru: uskladiti program s hrvatskim govornim područjem
  - testovi koji nisu izravno povezani s namjerom odvraćaju pažnju
- Grace Hopper: Easier to Ask for Forgiveness than get Permission

```
import locale
def mySetHrLocale():
  try:
    # Linux, FreeBSD, itd.
    locale.setlocale(locale.LC_COLLATE,
        ('hr', locale.getpreferredencoding()))
  except locale.Error:
    try:
      # MS Windows
      locale.setlocale(locale.LC_COLLATE, "hrv_HRV.1250")
    except locale.Error:
      raise
```

Tehnike: iznimke - detalji

Iznimke koristimo za obradu specijalnih slučajeva:

- = događaji koji se *mogu* pojaviti (viša sila, zapunio se disk)
- za događaje koji se *ne smiju* pojaviti preferiramo `assert`

Eksplicitno zauzeti resursi mogu iscuriti...

- `malloc`, **`new`**, `pthread_mutex_init`, `fopen`, `mmap`, `socket`, **`...`**
- međutim, C++ garantira pozivanje destruktora lokalnih objekata

```
void tricky_function() {
  // will leak in case of an exception
  int* p2 = malloc(1000000);

  // will not leak in case of an exception
  std::unique_ptr<BigObject> data(new BigObject);
  std::unique_ptr<int, void(*)(void*)>
      p1(malloc(1000000), free);

  // code that may throw
}
```

Tehnike: DbC

Oblikovanje temeljeno na ugovoru (B. Meyera, kreator Eiffela):

- komponente surađuju ispunjavanjem obaveza definiranih eksplicitnim **ugovorima**
- ugovor: uređuje odnos među komponentama
- ambicija: laka ili čak automatska detekcija grešaka

Terminologija: ako komponenta A **ovisi** (poziva, referencira, stvara, ...) o komponenti B, onda je A --- klijent, a B --- pružatelj

Dva osnovna elementa ugovora između klijenta i pružatelja (supplier):

- preduvjeti (preconditions) garantiraju primjenljivost komponente (reguliraju obaveze klijenta prema pružatelju)
- postuvjeti (postconditions) garantiraju ispravnost rezultata (reguliraju obaveze pružatelja prema klijentu)

Postoje još i invarijante (interni pokazatelji integriteta) i popratni efekti (funkcionalnost komponente), ali oni ne utječu na oblikovanje interakcije

Tehnike: DbC - provedba

Pokazat ćemo kako provesti **automatsko** testiranje ugovora u praksi

U većini jezika, ključni konstrukti su iznimke i naredba `assert`.

U C-u (i C++-u), `assert` je makro koji se evaluira u ništa pri optimiziranom prevođenju (`NDEBUG`, `man assert`)

```
double mysqrt(double val){
  //precondition (alternativno: assert(val>=0))
  if (val<0){
    throw std::invalid_argument(
        "mysqrt received a negative argument.");
  }

  double result;
  // here we calculate the result ...

  //postcondition
  assert(fabs(result*result-val)<1e-7);

  return result;
}
```

Tehnike: RAII

Zauzimanje inicijalizacijom (engl. Resource Acquisition Is Initialization)

Tehnika za garantirano korektno kontroliranje vlasništva nad resursom:

- resurs (datoteku, memoriju, ...) zauzimamo inicijalizacijom objekta
- otpuštanje se zbiva automatskim pozivom destruktora
- iznimke ne uzrokuju curenje resursa
- greške ne moramo provjeravati kad nam to ne odgovara

C++, Python (`with`), C# (`using`), Java 7 (try-with-resources)

Primjer: privremeno preusmjeravanje standardnog izlaza u datoteku

```
with open('help.txt', 'w') as f:
  with redirect_stdout(f):
    help(pow)

# automagic exception-safe cleanup:
#   redirection reverted, file closed
# https://docs.python.org/3/library/contextlib.html
```

Tehnike: RAII - primjer (C++)

Zauzimanje inicijalizacijom u C++-u:

```
static std::mutex my_mutex;

void append_msg(const std::string file, const std::string msg){
  // acquire mutex (must release it when done)
  std::lock_guard<std::mutex> lock(my_mutex);

  // open file (must close it when done,
  //            maybe invalid path, insufficient permission, ..
  std::ofstream myfile(file, std::ofstream::app);
  myfile.exceptions (std::ifstream::failbit |
                     std::ifstream::badbit );

  // write message (maybe full disk, disk failure, ...)
  myfile <<msg <<std::endl;

  // automagic exception-safe cleanup: i) file ii) mutex

  // all errors handled by clients (throw low, catch high)
}
```

Tehnike: OOP

Povijest: Smalltalk (Xerox PARC), CLU (MIT), 1970-1980;
Turingova nagrada za 2008. uručena Barbari Liskov (CLU)

U čemu je prednost OOP nad alternativama (neovisno o jeziku)?

- fokus strukturiranog programiranja na ostvarivanju zadanih (statičkih) svojstava [shalloway05]
- OOP razmatra **evoluciju** sustava: kako postići da kôd koji pišemo danas ispravno radi s kôdom koji ćemo pisati dogodine?"

Pri OO oblikovanju pitamo se: "Što će se mijenjati u budućnosti?";

- na temelju te procjene pokušavamo se zaštiti od promjena
- točnost prognoze određuje hoće li OOP pomoći ili ne
- programiranje intrinsično teško (no silver bullet [brooks 1986])

Logička načela

Načela oblikovanja elemenata logičke organizacije
(razredi i funkcije, **ne** datoteke)

- načelo nadogradnje bez promjene
  dodavanje funkcionalnosti bez utjecaja na postojeći kôd
- načelo nadomjestivosti osnovnih razreda
  ako A izvodi iz B, onda A možemo koristiti i kao B
  npr, tko god zna voziti auto, zna voziti i Fiat Punto
- načelo inverzije ovisnosti
  usmjeravanje ovisnosti prema apstraktnim sučeljima
- načelo jedinstvene odgovornosti
  komponente modeliraju koncepte koji imaju jasnu odgovornost
- načelo izdvajanja sučelja
  ne tjerati klijente da ovise o onom što ne koriste

Logička načela: nadogradnja bez promjene

Načelo nadogradnje bez promjene (NNBP) nas uči kako organizirati program tako da može primiti novu funkcionalnost bez izmjene postojećeg izvornog kôda

NBP (eng. open-closed principle): funkcionalnost komponente možemo proširiti **bez mijenjanja** njene implementacije

- **fleksibilnost**: nadogradnja ne utječe na klijente

Cilj: komponente **otvorene** za nadogradnju, ali **zatvorene** za promjene

- **važna ideja**: stari kôd poziva novi kôd!
- vezano uz ideje **skrivanja informacije** [Parnas72] i **apstrakcije podataka** [Liskov74]
- motivacija za mnoge programske tehnike i koncepte: predlošci, aspekti, introspekcija, refleksija, gniježdene funkcije...

Logička načela: NNBP + nasljeđivanje

Literatura predlaže dva pristupa za ostvariti NBP

U oba pristupa ključan je mehanizam **nasljeđivanja**:

1. nasljeđivanje implementacije [meyer88]
   - novi razredi pozivaju temeljnu implementaciju nasljeđivanjem starog razreda (nema polimorfnih poziva!)
   - postojeći klijenti ne mogu doći do nove funkcionalnosti
     - osim ako novi razred nadjača virtualne metode starog razreda
     - ovo proširenje stare ideje odgovara obrascu okvirna metoda
   - ideja nas ne uzbuđuje pretjerano: novi kôd poziva stari kôd
2. nasljeđivanje sučelja, oslanjanje na polimorfizam [martin96]
   - klijenti transparentno pristupaju novoj implementaciji polimorfnim pozivom preko starog sučelja
   - to je već zanimljivije: **stari kôd poziva novi kôd**!

Vrijeme je pokazalo da polimorfni pristup nudi veće mogućnosti

- za prvi kontekst danas preferiramo **kompoziciju**! (aggregation)

Logička načela: NNBP + nasljeđivanje (2)

Pretpostavimo da je zadan razred `Old`:

```
class Old{
public:
  void method();
};
```

Evo kako bismo taj razred nadogradili nasljeđivanjem implementacije (lijevo) i kako istu funkcionalnost postići agregacijom (desno):

```
// original Meyer's idea
class New: public Old{
public:
  //new code calls old code
  void newmethod(){
    // new functionality
    method();
    // more new code
  }
};
```

```
// modern variant:
class New{
  Old member;
public:
  void newmethod(){
    // new functionality
    member.method();
    // more new code
  }
};
```

Logička načela: NNBP + nasljeđivanje (3)

Evo i kako nadograditi razred `Old` nasljeđivanjem sučelja:

```
//old.hpp, written in 2007
class Old{
public:
  virtual void method();
};


//client.cpp, written in 2007
void client(Old* p){
  p->method();
};
```

```
//new.hpp, written in 2008
class New: public Old{
public:
  virtual void method();
};

//main.cpp
int main(){
  Old o; client(&o);
  New n; client(&n);
}
```

Vidimo da klijent iz 2007. uspješno radi s razredom iz 2008.

- nema potrebe za mijenjanjem (ni prevođenjem) komponente `client.cpp`
- stari kôd zove novi kôd (**vrlo korisno!**)

Logička načela: NNBP + proceduralni stil?

Proceduralni stil tipično dovodi do krutog i krhkog kôda

- **jednu** konceptualnu izmjenu potrebno je unijeti na **više** mjesta

To ćemo ilustrirati na elementima programa za vektorsku grafiku:

```
struct Point{/*...*/};

enum EShapeType {ESCircle, ESPoly};
struct Shape{EShapeType type_;};
struct Circle{
  EShapeType type_;
  double radius_;
  Point center_;
};
struct Polyline{
  EShapeType type_;
  int nPts_;
  Point* pPts_;
};

void drawPolyline(struct Polyline*);
void drawCircle(struct Circle*);
```

Logička načela: NNBP + proceduralni stil (2)

Razmotrimo sada izvedbu komponente za crtanje crteža:

```
void drawShapes(Shape** drawing, int n){
  for (int i=0; i<n; ++i){
    struct Shape* s = drawing[i];
    switch (s->type_){
    case ESPoly:
      drawPolyline((struct Polyline*)s);
      break;
    case ESCircle:
      drawCircle((struct Circle*)s);
      break;
    default:
      assert(0); exit(0);
    }
} }
```

Rješenje ima **integritet** (strukturirano je i jasno), ali je kruto i krhko:

- ne možemo ispitati `drawShapes()` u izolaciji (`drawPolyline()`, ...)
- ponavljanje **case** konstrukcije u `drawShapes()` i `moveShapes()`
- mukotrpno dodavanje novih objekata

Logička načela: NNBP + OOP

OOP **omogućava** rješavanje problema s prethodne stranice

- potreba: omogućiti `drawShapes` da apstrahira konkretne objekte!
- rješenje: polimorfni poziv preko zajedničkog sučelja

```
class Shape{
public:
  virtual void draw()=0;
};
class Circle :public Shape{
  virtual void draw();
  // ...
};
class Polyline :public Shape{
  virtual void draw();
  // ...
};

void drawShapes(const std::list<Shape*>& fig){
  std::list<Shape*>::const_iterator it;
  for (it=fig.begin(); it!=fig.end(); ++it){
    (*it)->draw();
} }
```

Logička načela: NNBP + SP

NNBP se može postići i *statičkim* polimorfizmom

- predlošci u C++-u, parametarski polimorfizam u ML-u, Scali i Haskellu
- nema ograničenja na razred objekta nad kojim se primjenjuje polimorfni poziv, posebno pogodno za **biblioteke**
- čarolija funkcionira samo tijekom prevođenja (nadograđenu komponentu potebno ponovo prevesti)
- statički vs dinamički polimorfizam (predložak vs. virtualna funkcija):
  - komplementarna primjenljivost
  - bolja učinkovitost statičkog polimorfizma
  - veća elegancija i lakši razvoj dinamičkog polimorfizma

Logička načela: NNBP + SP, primjer

Prednosti predložaka kolekcija u odnosu na polja C-a:

- mogućnost provjere pristupa uz jednaku učinkovitost optimiranog kôda
- mogućnost transparentnog rasta (`std::vector::push_back()`)
- mogućnost automatskog otpuštanja dinamički alociranog buffera
- mogućnost finog upravljanja operacijama nad konstantnim objektima

```
template<typename T> class Array {
public:
  Array(int sz): size_(sz), data_(new T[sz]) {}
  ~Array(){ delete[] data_; }
  int size() const { return size_;}
  const T& operator[](int i) const { return data_[check(i)]; }
        T& operator[](int i) { return data_[check(i)]; }
  //TODO: copy construction, assignment, resizing, ...
private:
  inline int check(int i) const {
    assert (i>=0 && i<size_); // or: throw "bound check error";
    return i;
  }
  int size_;
  T*  data_;
};
```

Logička načela: NNBP + SP, primjer (2)

Predložak `Array` je NBP jer radi sa svim tipovima koji omogućavaju:

- podrazumijevanu konstrukciju, preslikavanje, dodjeljivanje te destrukciju

```
int main(){
  Array<int> X(20);
  X[0]=5;

  Array<std::string> B(20);
  B[0]="bla";
}
```

Predložak `Array` je const-korektan:

```
template<typename T>
int process(const Array<T>& a){
  std::cout <<a[0];
  // a[0]=T(); does not compile!
}
```

Logička načela: NNBP + SP, primjer std::map

Program za određivanje histograma riječi u ulaznom toku:

```
#include <iostream>
#include <map>

int main(){
  std::map <std::string,int> wordcounts;
  std::string s;
  while (std::cin >> s){
    ++wordcounts[s];
  }

  for (auto keyval : wordcounts){
    std::cout <<keyval.first <<' ' <<keyval.second <<"\n";
  }
}
```

Kolekcija `std::map` je NBP jer radi sa svim tipovima koji podržavaju:

- podrazumijevanu konstrukciju, preslikavanje, pridijeljivanje te destrukciju
- nad ključevima treba dodatno biti definiran uređaj f(x,y):=x<y

Logička načela: NNBP u praksi

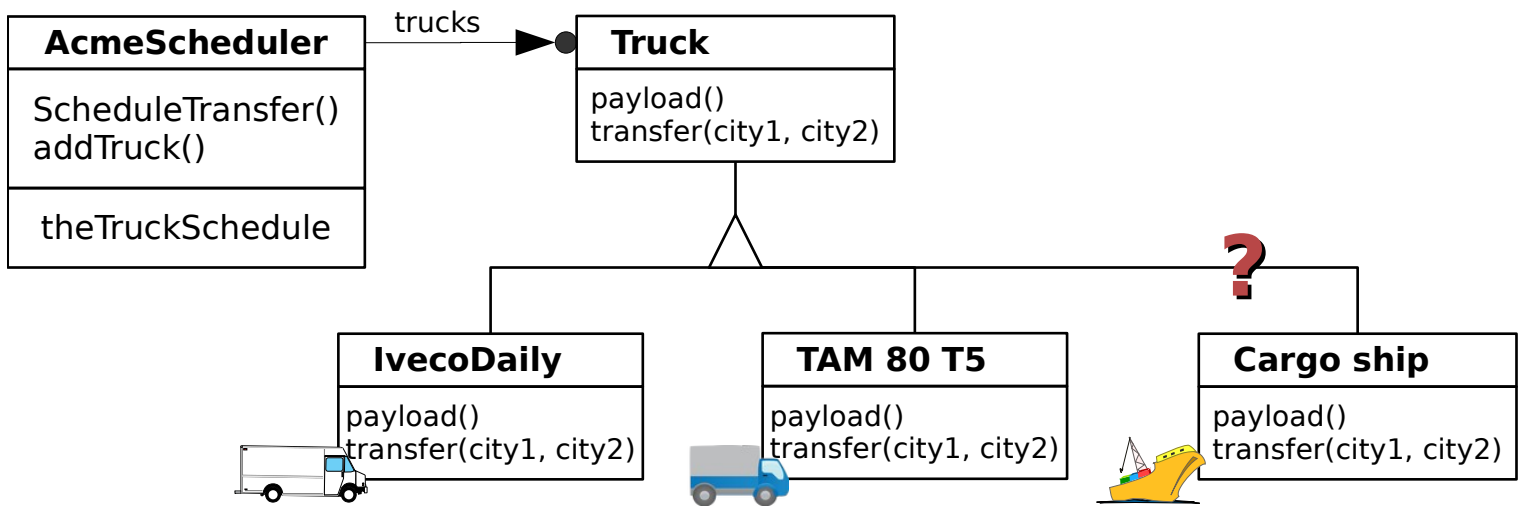
Koncepti koji pospješuju nadograđivanje bez promjene:

- **enkapsulacija**:
  - klijenti razreda ne smiju izravno referencirati podatkovne članove
  - inače, razred se ne može održavati bez utjecanja na klijente
  - $\Rightarrow$ svi podatkovni članovi razreda privatni
- **virtualne funkcije**:
  - moguće pozivanje modula napisanih godinama nakon klijenta: stari kôd zove novi kôd (suština NNBP)
- **apstraktni razredi** (apstraktna sučelja):
  - nemaju podatkovnih elemenata $\Rightarrow$ enkapsulirani
  - imaju virtualne funkcije
- **statički polimorfizam**: injekcija novog kôda u stari pri prevođenju

Logička načela: LNS

Načelo nadomjestivosti osnovnih razreda

- AKA Liskovino načelo supstitucije [Liskov93] (eng. Liskov substitution principle)
  - Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.
- osnovni razredi moraju se moći nadomjestiti izvedenim klasama
- npr: tko god zna voziti **auto**, zna voziti i **Fiat Punto**
- npr: svi potomci razreda `Truck` moraju moći surađivati s `AcmeScheduler`om



Logička načela: LNS --- je_vrsta

Načelo upućuje na pravilnu upotrebu nasljeđivanja:

- nasljeđivanje modelira relaciju **je_vrsta** (IS_A_KIND_OF)
- izvedeni razredi trebaju poštivati ugovore osnovnog razreda:
  - preduvjeti izvedenih metoda (na parametre ili na stanje objekta) moraju biti jednaki onima u osnovnom razredu ili oslabljeni
  - slično, postuvjeti izvedenih metoda moraju biti isti ili postroženi
- izvedeni razred krši LNS ako neki klijent koji korektno radi s osnovnim razredom ne može raditi s izvedenim razredom
- simptom patologije: klijenti moraju propitivati imaju li posla s izvedenim pružateljem koji krši LNS
- patologija je kršenje NNBP-a: klijent se mora mijenjati kad god se u program unese neispravno izvedeni pružatelj

Logička načela: LNS - geometrijski primjer

Razred `Circle` ne zadovoljava roditeljski postuvjet: krši se LNS, NNBP

```
class Ellipse{
public:
  virtual void setSize(
      double cx, double cy);
  //POSTCONDITION: width()==cx
  //POSTCONDITION: height()==cy
public:
  virtual void width() const;
  virtual void height() const;
protected:
  double width_, height_;
};

void client(Ellipse& e){
  e.setSize(10,20);
  assert(e.width()==10 &&
         e.height()==20);
}
```

```
class Circle:
  public Ellipse
{
public:
  virtual void setSize(
      double cx, double cy);
  //POSTCONDITION: width()==cx
  //POSTCONDITION: height()==cx
};
void Circle::setSize(
    double cx, double){
  width_=height_=cx;
}

int main(){
  Circle c;
  client(c);
}
```

Funkcija `client` prestaje biti NBP čim naslijedimo `Ellipse` kršeći LNS.

- krug nije vrsta elipse nego njen specijalni slučaj
- krug se može tretirati poput elipse samo ako je nepromjenljiv

Logička načela: LNS - ptičji primjer

```
class Bird{
public:
  Bird(){}
  virtual ~Bird(){}
public:
  double altitude() const{
    return altitude_;
  }
  virtual void fly();
  //POSTCONDITION: altitude()>0
  //...
protected:
  double altitude_;
};

void client(Bird& b){
  b.fly();
  assert(b.altitude()>0.0);
}
```

```
class Penguin:
  public Bird
{
//INVARIANT: altitude_=0.0
public:
  virtual void fly(){
    return; // do nothing
  }
};
//...

int main(){
  Penguin bird;;
  client(bird);
}
```

Problem: pingvin ne zadovoljava postuvjete roditelja, krši se LNS

Klijenti ptica prestaju biti NBP, jer moraju provjeriti rade li s pingvinima

Logička načela: LNS - ptičji primjer (2)

```
class Bird{
public:
  Bird(){}
  virtual ~Bird(){}
public:
  double altitude() const{
    return altitude_;
  }
  virtual void fly();
  //POSTCONDITION: altitude()>0
  //...
protected:
  double altitude_;
};

void client(Bird& b){
  b.fly();
  assert(b.altitude()>0.0);
}
```

```
class ExceptionCannotFly:
  public std::runtime_error
{};

class Penguin:
  public Bird
{
//INVARIANT: altitude_=0.0
public:
  virtual void fly(){
    throw ExceptionCannotFly;
  }
};
//...

int main(){
  Penguin bird;;
  client(bird);
}
```

Jedan pristup problemu je bacanje iznimke

- integritet uspostavlja neka komponenta na višoj razini apstrakcije

Još bolje: prekrojiti kôd, uvesti razrede `WalkingBird` i `FlyingBird`.

Logička načela: LNS - implikacije

Nasljeđivanje modelira relaciju **je_vrsta**:

- izvedeni razredi **moraju poštivati ugovore roditelja**
- javno nasljeđivanje **rijetko** koristimo za ponovno korištenje
- kršenje načela obično je posljedica slabog znanja o domeni
  - krug teško može biti vrsta elipse (niti elipsa vrsta kruga)
  - intuicija ponekad vara, a sve ptice ne lete

Kršenje LNS-a može se popraviti na 3 načina:

1. smanjiti odgovornosti osnovnog razreda
   (pojačati preduvjete, oslabiti postuvjete, reducirati sučelje)
2. povećati odgovornost izvedenog razreda
   (oslabiti preduvjete, pojačati postuvjete)
3. odustati od izravnog roditeljskog odnosa dvaju razreda

Logička načela: LNS i implicitno tipiziranje

Mnogi moderni jezici (Python, Ruby, JavaScript) imaju implicitno tipiziranje (engl. duck typing)

```
import numpy as np
def lincomb(a,b):
  return 3*a + 2*b

>>> lincomb(3,4)
17
```

```
>>> lincomb('a','b')
'aaabb'
>>> lincomb([1],[2,3])
[1, 1, 1, 2, 3, 2, 3]
>>> lincomb(np.array([1,2]), np.array([2,2]))
array([ 7, 10])
```

Značajke jezika s implicitnim tipiziranjem:

- tip argumenata funkcija i atributa razreda nije statički određen
- nadomjestive tipove možemo graditi i bez nasljeđivanja
- nasljeđivanje koristimo manje nego u statički tipiziranim jezicima

LNS u takvim jezicima izražavamo pomoću pojma nadomjestivosti

- tip S može *nadomjestiti* tip T ako za svako svojstvo q vrijedi $\forall x \in T, \forall y \in S : q(x) \Rightarrow q(y)$
- poopćeno načelo: klijentima ne valja slati nenadomjestive inačice pružatelja

Logička načela: LNS - zaključak

U statički tipiziranim jezicima (C++, Java, C#):

- LNS predstavlja recept za korištenje **nasljeđivanja**
- **nasljeđivanje** koristimo za modeliranje **nadomjestivih** tipova
  - klijenti koriste izvedene razrede preko osnovnog sučelja
- za preuzimanje (engl. reuse) funkcionalnosti preferiramo kompoziciju

U implicitno tipiziranim jezicima (Python, Ruby, JavaScript):

- poopćeni LNS formuliramo neovisno o nasljeđivanju, uz pomoć pojma **nadomjestivosti** tipova
- ako klijent treba raditi s različitim pružateljima, pružatelji moraju biti međusobno nadomjestivi (u suprotnom klijent nije NBP)

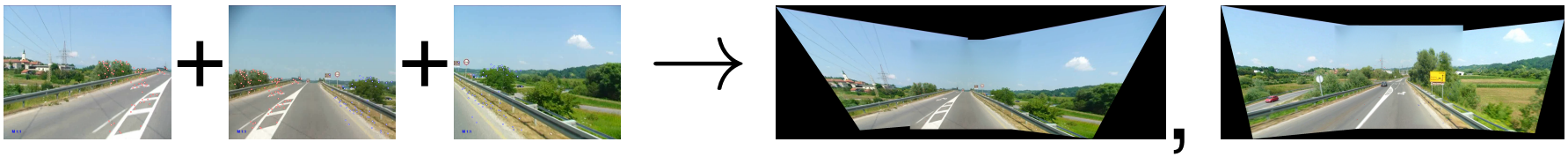
Logička načela: NIO

Načelo inverzije ovisnosti (eng. dependency inversion principle)

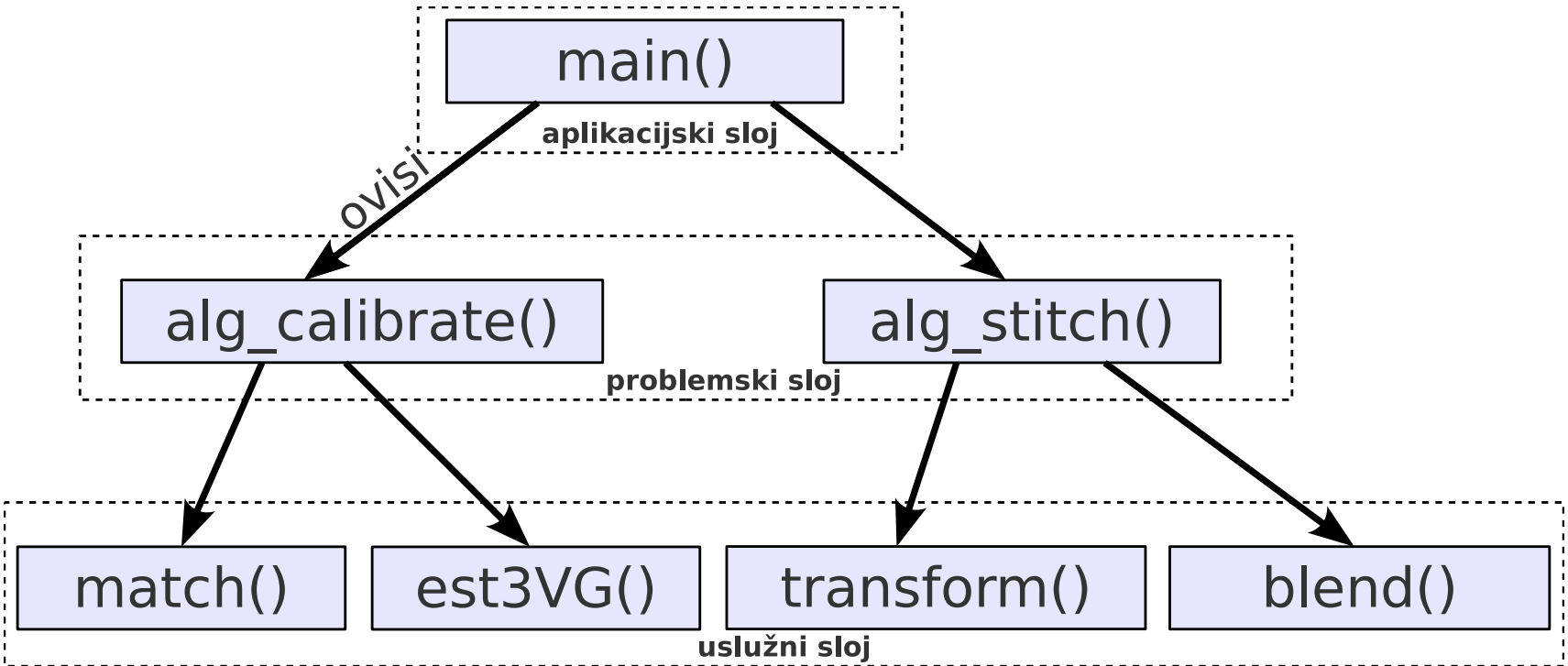
- vidjeli smo da je nadomjestivost (LNS) nužni uvjet nadogradnje bez promjene
- sada: implikacije nadogradivosti i nadomjestivosti na strukturu ovisnosti komponenata vode na načelo inverzije ovisnosti
- pokazat ćemo da se poželjna struktura ovisnosti (izbjegavanje krutosti i neprenosivosti) postiže usmjeravanjem ovisnosti prema **apstraktnim** sučeljima

Logička načela: NIO: motivacija

Primjer iz stvarnog života: program za spajanje slika u mozaik

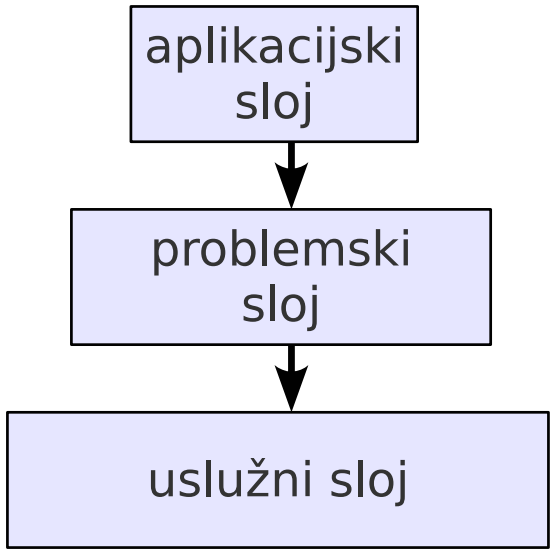


Ako koristimo **proceduralni stil**, program bismo organizirali kao na sljedećoj slici:



Logička načela: NIO: motivacija (2)

Nažalost, **proceduralni stil** dovodi do piramidalne strukture međuovisnosti:

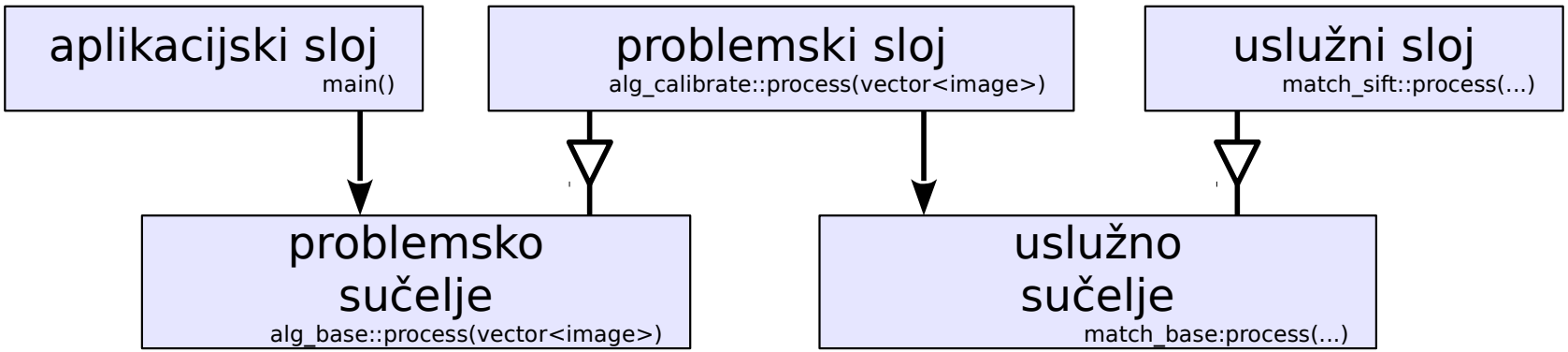


- Aplikacijski sloj:
  program za računalni vid
- Problemski sloj:
  kalibriranje i spajanje slika
- Uslužni sloj:
  korespondencije, geometrija,
  transformiranje i miješanje piksela

- loše: moduli visoke razine ovise o izvedbenim detaljima
  (ne mogu se ni prevesti ni ispitati ako niži moduli nisu dovršeni)
- loše: pri mijenjanju modula niže razine često se javlja domino-efekt
  (promjene se propagiraju prema višim razinama)

Logička načela: NIO i OOP

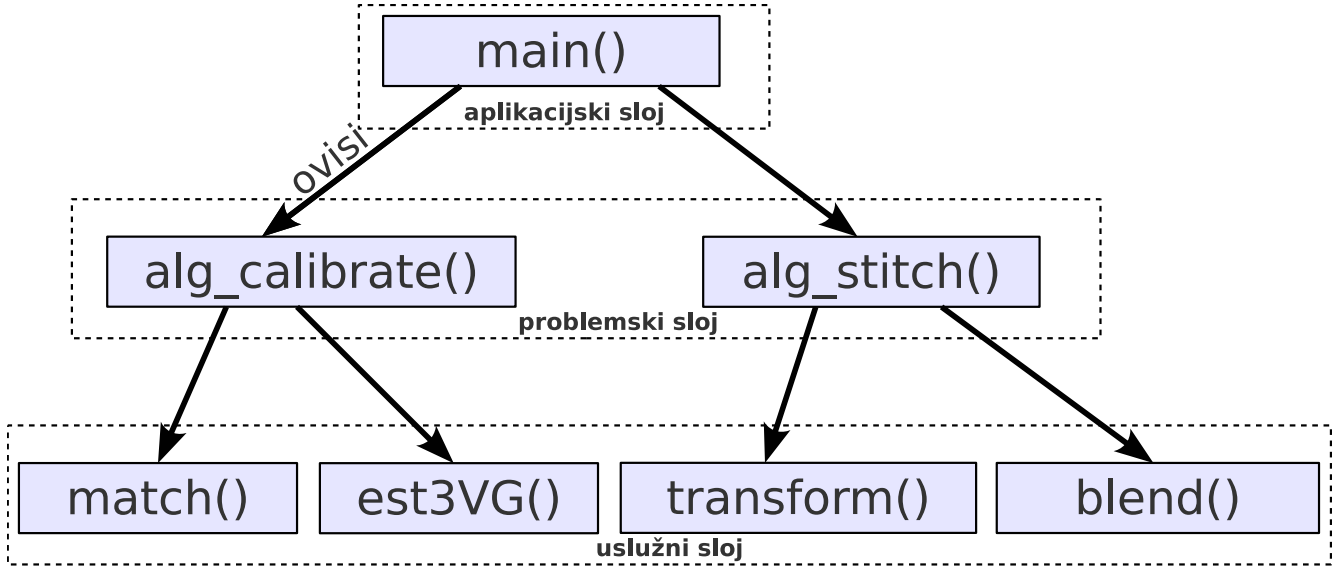
Nedostatci se mogu riješiti promjenom strukture ovisnosti:



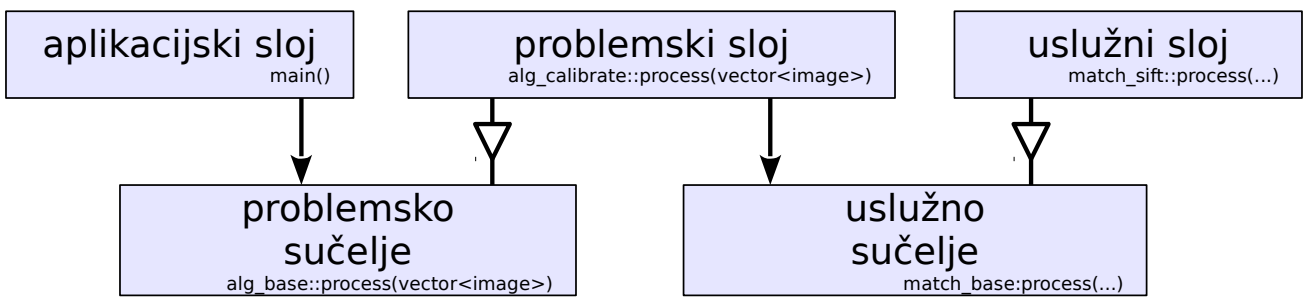
- u novoj organizaciji, ovisnosti idu prema apstrakcijama (koje imaju malo razloga za promjenu jer nemaju implementaciju)
  - isti glavni program koristimo i za kalibraciju i za spajanje
- ne ovisimo više o nepostojanim modulima s detaljnim implementacijama: kažemo da je ta ovisnost **invertirana**
  - glavni program ne zna koji problem rješava
- rezultati primjene inverzije ovisnosti:
  - piramidalna struktura preokrenuta: **ovisimo o apstrakcijama**
  - promjer grafa je smanjen (putevi ovisnosti su **kraći**)

Logička načela: NIO - primjer

PRIJE: △



POSLIJE: ▽



Logička načela: NIO u praksi

Ovisnosti **u načelu** trebaju ići prema apstrakcijama
(**NE** od modula visoke razine prema modulima niske razine)

- komponenta bez implementacije se rjeđe mijenja
- apstrakcije omogućavaju nadogradnju bez promjene

Ovisnost o **postojanim** konkretnim modulima je OK
(nećemo apstrahirati elemente standardne biblioteke!)

Važan problem: stvaranje objekata konkretnih razreda

- stvaranje implicira ovisnost: kako izbjeći ovisnost glavnog programa o modulima niske razine?
  - primjenom injekcije ovisnosti i obrasca tvornice
- injekcija ovisnosti je strukturni idiom koji omogućava lokaliziranje ovisnosti o konkretnim komponentama
  - obradit ćemo ga sada!
- tvornice ćemo danas *samo najaviti*
  - konkretizacija na kasnijim predavanjima i laboratorijskoj vježbi 3

Logička načela: NIO, injekcija ovisnosti

Injekcija ovisnosti: umjesto hardkodiranog kompliciranog konkretnog člana, **uvodi se** **konfiguriranje** preko reference na osnovni razred

```
// without dependency injection
class Client1 {
  ConcreteDatabase myDatabase;
public:
  Client1():
    myDatabase() {}
public:
  void transaction() {
    myDatabase.getData();
    // ....
  }
};
```

```
// with dependency injection
class Client2 {
  AbstractDatabase& myDatabase;
public:
  Client2(AbstractDatabase& db):
    myDatabase(db) {}
public:
  void transaction() {
    myDatabase.getData();
    // ...
  }
};
```

`Client2` koristi injekciju ovisnosti, za razliku od `Client1`:

- ovisnost uslijed stvaranja izvukli smo u zasebnu komponentu
- maksimizirali smo inverziju ovisnosti, odnosno lokalizirali ovisan kôd

Logička načela: NIO, injekcija ovisnosti (2)

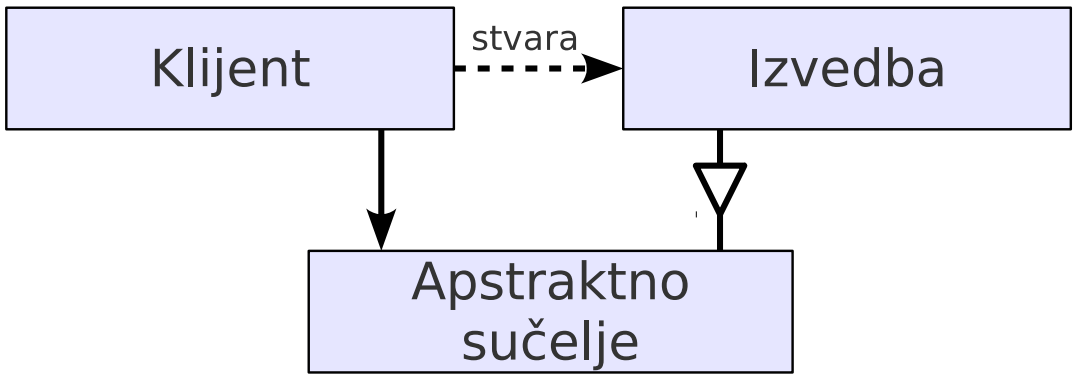
Mogućnost **neovisnog ispitivanja** s obzirom na `ConcreteDatabase`:

```
//==== client2.hpp
class Client2 {
  AbstractDatabase&
    myDatabase;
public:
  Client2(
    AbstractDatabase& db)
  :
    myDatabase(db)
  {}
public:
  void transaction() {
    myDatabase.getData();
    // ...
  }
  // ...
};
```

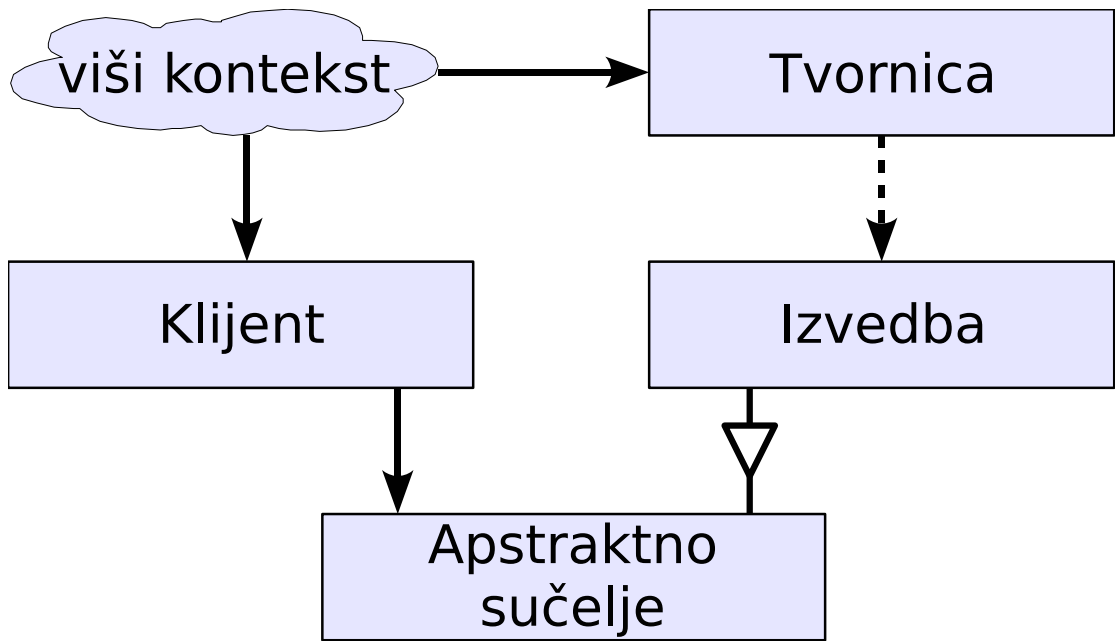
```
//==== testClient2.cpp
int main(){
  // construct database
  MockDatabase* pdb =
    new MockDatabase();

  // construct test object
  // (dependency injection)
  Client2 client(*pdb);

  // test behaviour #1
  client.transaction();
  assertGetDataWasCalled(*pdb);

  // test behaviour #2
  // ...
}
```

Logička načela: NIO, efekti

Organizacija bez injekcije ovisnosti:

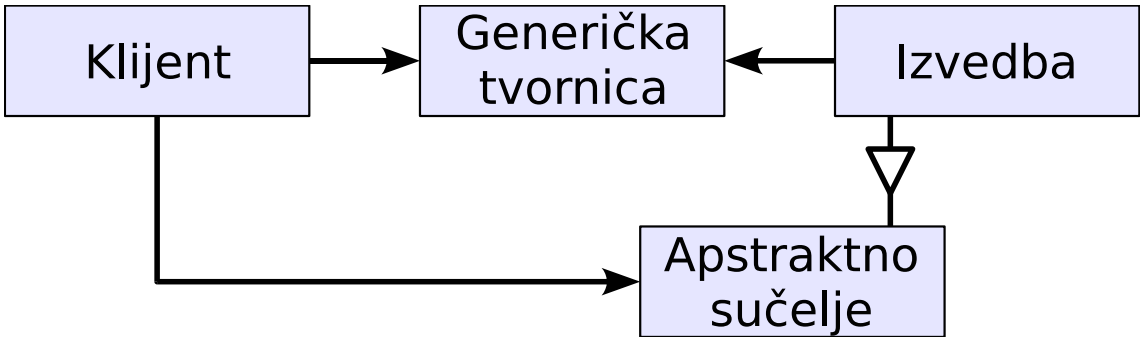


Organizacija s injekcijom ovisnosti i tvornicom:



Logička načela: NIO, generička tvornica

Idealno: tvornica ne ovisi o konkretnom tipu



Evo kako izgleda izvorni kod (3. laboratorijska vježba):

```
// client.cpp
...
Animal *pb = (Animal*)
  factory.create("parrot");
```

```
// parrot.cpp
#include "animal.hpp"
#include "factory.hpp"

class Parrot{ /* ... */  };
static void* creator(){
  return new Parrot();
}

// may require flags (C++20 6.9.3.3)
int hreg=factory.
  register("parrot", creator);
```

Logička načela: NIO, C

U C-u, NIO možemo izvesti prema receptu iz prve laboratorijske vježbe:

```c
#include "dog.h"
int main(){
  struct Animal* pDog=createDog("Hamlet");
  pDog->vptr[0](pDog); // vau!
  // ...
}
```

Gore, klijentski kod mora poznavati `createDog`, dakle uključiti `"dog.h"`, (kao i u C++-u). To možemo izbjeći ovako:

```c
#include "animal.h"
int main(){
  animal_t brundo; // klijent ne pozna animal_t!
  animal_create(&brundo, "dog", "Hamlet");
  animal_greet(brundo);
}
```

C omogućava definirati `animal_t` bez otkrivanja interne strukture tog tipa

- to bismo napravili u `"animal.h"`, sljedeća stranica otkriva kako

Logička načela: NIO, C (2)

Mehanizam s prethodne stranice nalazimo u brojnim bibliotekama poput `<pthread.h>`!

Jednostavnim pretraživanjem zaglavlja pronalazimo sljedeće definicije:

```
typedef unsigned long int pthread_t; // Linux

typedef struct _opaque_pthread_t *pthread_t; // MacOS

typedef struct pthread *pthread_t; // FreeBSD
```

Evo kako to možemo istražiti uz pomoć C++-a:

```
int main() {
  std::cout << "pthread_t je: "
    << typeid(pthread_t).name() << "\n";
}
```

Logička načela: NJO

Načelo jedinstvene odgovornosti (eng. single responsibility principle)

- programski moduli moraju imati *samo jednu* odgovornost!
- srodno načelu kohezije [DeMarco79]

Kako razdioba odgovornosti po modulima može pospješiti organizaciju programa?

- odgovornosti modula odgovaraju razlozima za promjenu (veza 1:1)
- ukoliko svi moduli imaju jedinstvenu odgovornost, sve promjene rezultiraju promjenom samo jednog modula!
- suprotno, ako modul ima više odgovornosti, među njima se javljaju neprirodne međuovisnosti (krutost, nepokretnost)

NJO: oblikovati ortogonalan sustav u kojem razdioba poslova odgovara intrinsičnoj strukturi problema
(svaka porodica komponenata modelira svoju **os promjene**)

Logička načela: NJO - primjer

Razmotrimo organizaciju programa za vektorsku grafiku, gdje obitelj Shape obuhvaća više od jedne odgovornosti:

```
class Shape{
public:
  virtual ~Shape();
  virtual void rotate(double)=0;
  virtual void draw(Window&)=0;
};
class Rectangle: public Shape{
  // ...
public:
  void rotate(double phi);
  void draw(Window& wnd);
private:
  Point pt;
  int width, height;
}
```

```
typedef std::list<Shape*> Drawing;
//package GUI
void GUI::draw(Drawing& d, Window& w)
{
  Drawing::iterator it=d.begin();
  while (it!=d.end()){
    it->draw(w); ++it;
  }
}
//package Geom
void Geom::intersect(const Shape& sin1,
    const Shape& sin2, Shape& sout)
{
  // as above...
}
```

- Loše: paket Geom ovisi o paketu GUI, bez opravdanja (Geom::intersect ovisi o Shape koji zna za Window)
- Loše: kako omogućiti da se crtež iscrta u sliku?
- **Bolje**: crtanje izvesti u zasebnoj komponenti (Posjetitelj ili Most)

Logička načela: NJO - odgovornost

Što je **odgovornost** modula?

- odgovornost je kvant funkcionalnosti iz domene aplikacije (svaka odgovornost je ujedno i razlog za promjenu modula)
- svaki modul bi trebao imati jednu, samo jednu odgovornost (Ali koju? To *mi* trebamo otkriti!)
- često teško pogoditi isprve: ono što se ispočetka čini kao jedinstvena odgovornost, kasnije se može pokazati kao više srodnih odgovornosti (vidi prethodni primjer)
- **analiza domene** (ono što smo rekli da je teško!) u mnogome se svodi na određivanje odgovornosti (odnosno osi promjene)
- kad smo sigurni da moduli nemaju jedinstvenu odgovornost, podsustav treba prekrojiti (*refactor*) (što ranije to bolje!)

Logička načela: NJO - zaključak

Načelo jedinstvene odgovornosti je temelj programskog inženjerstva

- smanjivanje nepotrebne međuovisnosti te poticanje **održive evolucije**

Kontekst za ispravnu raspodjelu odgovornosti **izranja** postupno, kako naše razumijevanje domene postaje bolje

- ispočetka težimo neopravdanom gomilanju odgovornosti razreda npr. crtanje po prozoru, snimanje u nekom specifičnom formatu
- organizacijski nedostatci postaju vidljivi tek kad projekt uznapreduje
- stoga tada funkcionalnost selimo iz metoda u zasebne komponente

Ortogonalna konceptualizacija: sveti gral programske organizacije

- pronalaženje i dekoreliranje odgovornosti suštinski zadatak oblikovanja programske podrške
- metode: primjeri korištenja, probni baloni, prototipi, reverzno inženjerstvo

Logička načela: NIS

Načelo izdvajanja sučelja (eng. interface segregation principle):

- složeni koncepti čije korištenje ovisi o klijentu ipak se javljaju: ponekad zgodno napraviti kompromis s jedinstvenom odgovornošću
- izradom monolitnog sučelja za takve koncepte:
  - nepotrebno zbunjujemo autore klijenata (swiss army knife anti pattern)
  - omogućavamo suvišne ovisnosti o nekorištenim elementima sučelja
- načelo sugerira da nekoherentnim konceptima (ako ih baš moramo imati) klijenti trebaju pristupati preko **izdvojenih** sučelja:
  - ako baš moramo imati objekte koji rade više od jedne stvari, idemo barem od te komplikacije sačuvati klijente
  - ne želimo klijente primoravati na ovisnost o sučelju kojeg ne koriste

Logička načela: NIS - primjer

Razmatramo sučelje `Door` koje enkapsulira operacije nad vratima:

```
//====  door_v1.hpp
class Door{
public:
  virtual void lock()   = 0;
  virtual void unlock() = 0;
  virtual bool isLocked() = 0;
};
```

Pretpostavimo da se javila potreba za automatskim zaključavanjem!

Imamo novi zahtjev: naša vrata od sada se moraju moći pozivati i preko sučelja `TimerClient`.

```
//====  timer.hpp
class Timer{
public:
  void subscribe(int timeout, TimerClient* client);
};
class TimerClient{
public:
  virtual void timeOut() = 0;
};
```

Logička načela: NIS - preopsežno sučelje

Zahtjev zadovoljavamo oblikovanjem razreda `Door_V2`.

```
//==== door_v2.hpp
#include "timer.hpp"
class Door_V2:
  public TimerClient
{
public:
  virtual void lock();
  virtual void unlock();
  virtual bool isLocked();
  virtual void timeOut();
};
```

Razred `Door` sada postaje `Door_V2` pa njegovi klijenti postaju ovisni o komponenti `TimerClient`.

Ako se bilo što dogodi komponenti `TimerClient`, klijenti razreda `Door_V2` ne mogu se ni testirati ni ponovno koristiti

Klijenti vrata postaju ranjivi na promjene koje mogu zadesiti `TimerClient`.

Logička načela: NIS - preopsežno sučelje

Pretpostavimo sada da je sučelje `Timer` izmijenjeno na način da podržava višestruke događaje

```
class Timer_v3{
public:
  void subscribe(int timeout,
      int id,
      TimerClient* client);
};
class TimerClient_v3{
public:
  virtual void timeOut(int)=0;
};
```

Loše: ako bismo sada išli mijenjati `Door_V2`, promjena bi se odrazila na sve klijente (i na one koji ne trebaju automatska vrata)

Zaključak: sučelje `Door_V2` je preopsežno, treba ga prekrojiti

Logička načela: NIS - preopsežno sučelje

Rješenje: razdvojiti sučelja korisnika običnih i automatskih vrata.

```
// solution:
//  interface segregation
//====  timedDoor.hpp
#include "door_v1.hpp"
#include "timer_v3.hpp"
class TimedDoor:
  public Door,
  public TimerClient_v3
{
public:
  virtual void lock()  ;
  virtual void unlock();
  virtual bool isLocked();
  virtual void timeOut(int id);
};
```

Sučelje `TimedDoor` oblikovano je u skladu s načelom izdvajanja sučelja: sada su specijalna vrata u skladu s naknadnim zahtjevima, dok obična vrata **ne ovise** o `Timer`u!

Logička načela: NIS - sažetak

Izdvojena sučelja primjenjujemo kod složenih koncepata koji se koriste na više ortogonalnih načina (recept za **višestruko nasljeđivanje**)

U žargonu, izdvojena sučelja nazivaju se: **mixin** (C++) i **interface** (java)

Najbolje je ipak takve koncepte izbjeći ako je moguće (NJO).

Logička načela: Logička načela - sažetak

NNBP: identificirati odgovornosti u kojima očekujemo promjene te ih delegirati vanjskom pružatelju

- delegiranje provodimo preko reference (pokazivača) koja skriva konkretnu vrstu pružatelja
- u statičkim jezicima referenciramo osnovni razred pružatelja
- kod statičkog polimorfizma, vanjski pružatelj određen je parametrom predloška

LNS: alternativni pružatelji trebaju biti nadomjestivi

- u statičkim jezicima, nadomjestivost formuliramo nasljeđivanjem (koje modelira relaciju `je_vrsta`)
- u dinamičkim jezicima, nadomjestivost tipova proizlazi iz kompatibilnosti sučelja

NIO: duge lance ovisnosti možemo skratiti usmjeravanjem ovisnosti prema apstraktnim sučeljima

- konkretne pružatelje injektira vanjski kontekst

Logička načela: Logička načela - sažetak (2)

NJO: recept za grupiranje funkcionalnosti po komponentama

- grupiramo funkcionalnost koja (prema trenutnim saznanjima) ima jedan zajednički razlog za promjenu

NIS: recept za višestruko nasljeđivanje

- kako napraviti kompromis s NJO kod objekata koje različiti klijenti koriste na različite načine

Prikazani sustav logičkih načela: sažete smjernice za oblikovanje funkcija i razreda koje se mogu dobro nositi s promjenom

Logička načela: druge sistematizacije

Načela dobrog logičkog oblikovanja mogu se izraziti na različite načine

Prednost izložene sistematizacije [martin04]: optimalni odnos općenitosti i sažetosti

Naravno, postoje i druge sistematizacije načela oblikovanja

Elementarna načela (zdrav razum?):

- keep it simple, stupid (KISS)
  - uvijek dajemo prednost jednostavnim rješenjima koja se brzo razvijaju i lako održavaju
- you ain't gonna need it - YAGNI
  - ne tipkamo funkcionalnosti koje nisu prioritetne
- do not repeat yourself (DRY) [Hunt99]
  - ponavljanje je zlo jer dovodi do krhkosti
- the refactor rule of three
  - kad isti dio koda napišeš treći put, implementacije valja sjediniti u zasebnu komponentu

Logička načela: druge sistematizacije

Npr, načela iz knjige Head-First Design Patterns [Freeman04]:

- Encapsulate what varies (enkapsulacija, NNBP)
- Program to interfaces, not implementations (enkapsulacija, NNBP)
- Favor composition over inheritance (LNS)
- Strive for loosely coupled designs between objects that interact (sva načela logičkog oblikovanja)
- Classes should be open for extension but closed for modification (NNBP)
- Depend on abstractions. Do not depend on concrete classes (NIO)
- Principle of Least Knowledge - talk only to your immediate friends (enkapsulacija, NNBP)
- The Hollywood Principle - don't call us, we'll call you (callback funkcije, oblikovni obrazac Naredba)
- A class should have only one reason to change (NJO)

Fizička načela

Vidjeli smo kako se **nadogradivost**, **razumljivost** i **fleksibilnost** pospješuju logičkim načelima

Od traženih dobrih osobina ostalo je **lako ispitivanje** (testiranje) (lako ispitivanje usko vezano s lakim razumijevanjem!)

Ispitivanje najzgodnije provoditi nad **datotekama** izvornog kôda: analiziramo fizičku organizaciju

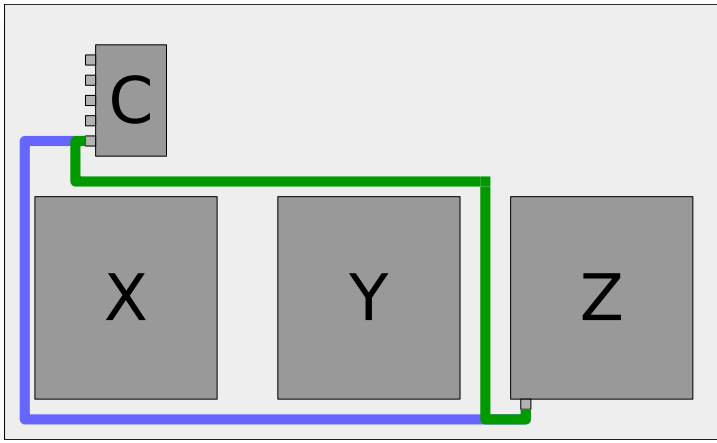
Razmatrat ćemo prikladnost odnosa među komponentama oblikovanja u smislu olakšavanja **inkrementalnog testiranja** programskog projekta

Na kraju ćemo dati smjernice za organizaciju **paketa** u veće programske sustave (100kLoC)

Fizička načela: o ispitivanju, uvod

Kako sveobuhvatno testirati prospajanje elektroničkog sklopa?

```
class P2P_Router{
public:
  P2P_Router(const Polygon& p);
  void addObstruction(const Polygon& p);
  void findPath(
    const Point& start, const Point& end,
    int width, Polygon& rv) const ;
};
```



- zadatak je težak jer naša komponenta može dobro raditi samo za neke rasporede prepreka

Još jedan primjer: kako ispitati čip s 1e6 tranzistora i 30 pinova?

- složeni elektronički sustavi rješavaju taj problem **ispitnim modulima**!

Fizička načela: o ispitivanju, pristup

Pretpostavimo da trebamo testirati n komponenata

- je li bolje komponente testirati **zasebno** ili zajedno?

Na stolu su dvije opcije:

- inkrementalno testiranje komponenata (engl. unit testing) vs
- sveobuhvatno testiranje sustava (engl. big bang integration testing)

Ako komponente testiramo zajedno, moramo uzeti u obzir i interakcije (u najgorem slučaju, 2^n)

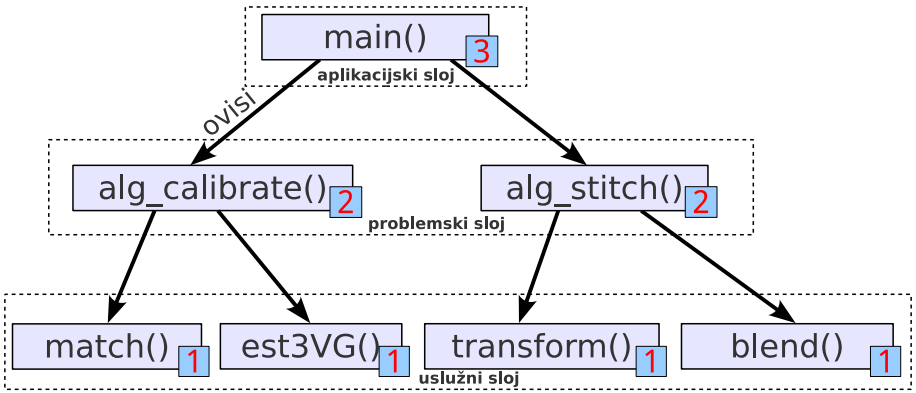
Inkrementalno testiranje je uvijek bolji odabir!

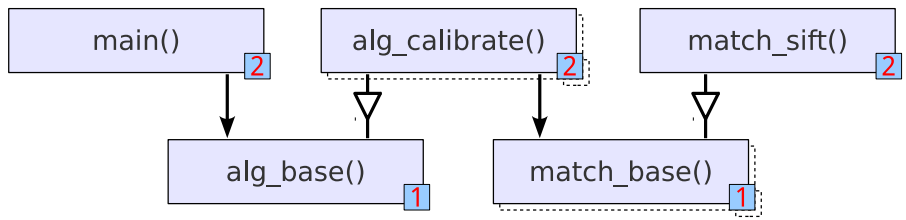
- kod programskog oblikovanja možemo (i trebamo!) testirati **inkrementalno**, **rano**, **automatski**, **regresijski**

Fizička načela: razine

U kontekstu testiranja i fizičke organizacije, ključna relacija je ovisnost

Ako je ovisnost **aciklička** $\Rightarrow$ komponentama možemo dodijeliti razine; razine definiraju redoslijed **inkrementalnog** testiranja komponenata!





Prvo testiramo komponente na razini 1

- to možemo provoditi u potpunoj izolaciji

Zatim, inkrementalno testiramo komponente na razini 2

- odvojeno testiramo svaku komponentu na razini 2 s minimalnim podskupom (već testirane) razine 1

Svaka komponenta ima svoj makefile i svoj test driver s funkcijom main

Fizička načela: razine (2)

Sada možemo izraziti načela fizičkog oblikovanja komponenata:

- Povoljno: **plitka** i **nepovezana** struktura ovisnosti
- Nepovoljno: duboka ili monolitna struktura ovisnosti, te ciklusi

Vidimo da su ciljevi fizičkog i logičkog oblikovanja kompatibilni:

- logička načela implicitno potiču smanjivanje međuovisnosti
- fizička načela rade to isto, samo eksplicitno

Struktura **međuovisnosti**: vrlo važan pokazatelj organizacijske kvalitete

Fizička načela: ciklusi

Ciklička struktura ovisnosti ometa ispitivanje, ponovno korištenje, i razumijevanje

Šteta je tim veća što je ciklus veći (funkcionalnost, broj komponenata) ekstrem: komponenta na dnu hijerarhije ovisi o vršnoj komponenti

Empirijski rezultat na 78 velikih programa u Javi:

- 45% programa ima ciklus od barem 100 razreda (!)
- 10% programa ima ciklus od barem 1000 razreda (!)
- Melton and Tempero: An Empirical Study of Cycles among Classes in Java, OOPSLA'06

Iako rješenje gotovo uvijek postoji (iznimka: iteratori i kolekcije), općenitog recepta za prekidanje ciklusa nema

Motivacija velikog broja **oblikovnih obrazaca**: uklanjanje cikličkih ovisnosti u posebnim slučajevima od širokog značaja!

Fizička načela: ciklusi, primjer

Čest problem u grafičkim aplikacijama: cirkularna ovisnost dokumenta i pogleda

```
// document.hpp
#include "view.hpp"
class Document{
public:
  void setState(){
    // ...
    pview_->update();
  }
  void getState();
private:
  View *pview_;
  // or: list<View*> pviews_;
};
```

```
// view.hpp
#include "document.hpp"

class View{
  Document *pdoc_;
public:
  void update(){
    // use pdoc_->getState()
  }
};

// Circular dependency Doc <-> View
// can't test neither of Doc, View!
```

Kako omogućiti nezavisno ispitivanje dvaju razreda?

Fizička načela: ciklusi, rješenje

Cikličku ovisnost često možemo raspetljati **inverzijom ovisnosti**:

```
// document.hpp
#include "viewbase.hpp"
class Document{
public:
  void setState(){
    // ...
    pview_->update();
  }
  void getState();
  void attach(ViewBase* pv);
private:
  ViewBase *pview_;
};
// Document depends on ViewBase
// Document does not know View
```

```
// viewbase.hpp
class ViewBase{
public:
  virtual void update()=0;
};

// view.hpp (includes document.hpp)
class View: public ViewBase{
  Document *pdoc_;
public:
  View(Document *doc): pdoc_(doc){
    pdoc_->attach(this);
  }
  void update();
};
```

Kako izgleda konačni graf ovisnosti?

Oblikovanje paketa

Kako program raste i usložnjava se, **komponente** postaju presitne (kad projektiramo avion, ne razmišljamo o vijcima)

Potreba za **većim** oblikovnim jedinicama koje se **zajedno** razvijaju i koriste: takve jedinice nazivamo paketima (npr, biblioteke su paketi!)

Ne može se postići da ovisnosti komponenata ne prelaze granice paketa

Pitanja na koje trebamo odgovoriti:

1. kako **grupirati** komponente u pakete (načela **koherentnosti**) ?
2. kako valja urediti **odnose** među paketima (načela **stabilnosti**) ?

Oblikovanje paketa: koherencija

Načela koherencije bave se **raspodjelom** komponenata po paketima:

1. grupiranje prema korelaciji korištenja:
   - izdavanje paketa iziskuje napor s obje strane (autor, klijent)
   - komponente koje se ne koriste zajedno **ne pripadaju** istom paketu
2. grupiranje prema zajedničkom izdavanju:
   - promjene elemenata paketa moraju biti međusobno **usklađene**
   - $\Rightarrow$ grupiramo komponente koje se **zajedno** izdaju i **održavaju**
   - $\Rightarrow$ interakcija između oblikovnih (korelacija korištenja) i poslovnih kriterija (izdavanje)
3. grupiranje prema odgovornosti (osjetljivosti na promjene):
   - komponente osjetljive na promjene iz **istog** skupa
   - **samo jedan** razlog za novo izdanje paketa

Oblikovanje paketa: koherencija(2)

Koherencija paketa srodna jedinstvenoj odgovornosti modula, ali...

Nije dovoljno da komponente modeliraju jedinstvenu os promjene složenog sustava (tj, da imaju jedinstvenu odgovornost)

Moramo razmatrati i međusobno suprotstavljene zahtjeve:

- grupiranja prema **izdavanju** (razvijanju i održavanju)
- grupiranja prema **korištenju**

Dinamička ravnoteža među gornjim zahtjevima i potrebama aplikacije

- česte **promjene** raspodjele komponenti po paketima tijekom napredovanja projekta
- prioritet s lakog razvijanja postupno prelazi na lako korištenje (grupiranje prema izdavanju $\rightarrow$ grupiranje prema korištenju)

Oblikovanje paketa: stabilnost

Načela stabilnosti bave se **uređajem** odnosa među paketima

Razdioba razreda po paketima mora prigušivati promjene
inače, udio integracije u velikom projektu ne može se ograničiti

1. načelo **acikličke ovisnosti**: nužni element prigušenja promjena
(ništa novo, jednako kao i za komponente)

2. načelo **stabilne ovisnosti**:
usmjeravanje ovisnosti prema **inertnijim** paketima
(analogno načelu inverzije ovisnosti za razrede)

3. načelo **primjerene apstrakcije**:
paket treba biti tim apstraktniji što je više stabilan

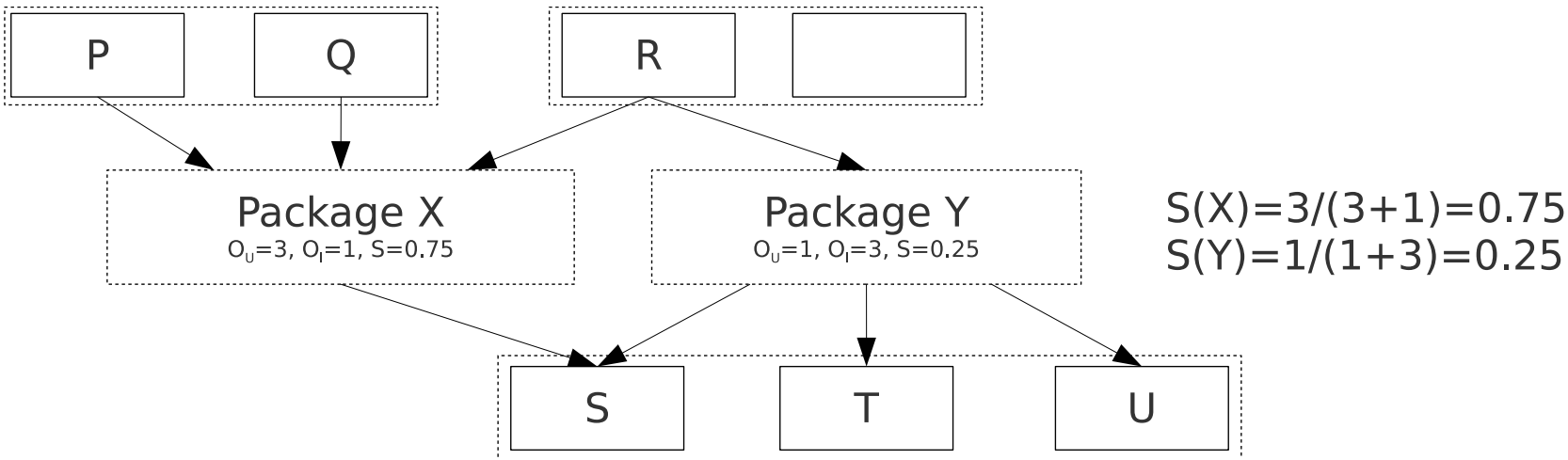
Oblikovanje paketa: metrika stabilnosti

Stabilnost $\equiv$ otpornost na promjene:

- pozitivne i negativne konotacije: **pouzdanost** vs. inertnost!
- inertnost je određena međuovisnostima (odgovornost, ovisnost)!

Metrika stabilnosti paketa $S = \frac{O_U}{O_U + O_I} \in \langle 0, 1 \rangle$:

- O_U ... odgovornost, broj vanjskih komponenti koje ovise o paketu
- O_I ... nepostojanost, broj vanjskih komponenti o kojima paket ovisi



Oblikovanje paketa: NIO za pakete

Načelo stabilne ovisnosti:

- ovisnost usmjeriti prema stabilnijim (inertnijim) paketima
- metrika S treba **rasti** uzduž puteva u grafu ovisnosti!

Stabilnost je u kontradikciji s izlaznima ovisnostima: stabilni paketi mogu biti **ili** nadogradivi bez promjene **ili** relativno jednostavni

Kako popraviti kršenje načela?

- tj, što napraviti kad stabilni paket X ovisi o fleksibilnom paketu Y?
- rješenje je paket Y prekrojiti u dva paketa Y_S i Y_I (NIO za pakete):
  - Y_S sadrži sučelja početnog paketa
  - Y_I sadrži implementacije početnog paketa
- X sada ovisi samo o stabilnom Y_S, pa načelo vrijedi!

Oblikovanje paketa: metrika apstrakcije

Uvedimo metriku apstraktnosti paketa $A = N_a/N_c \in \langle 0, 1 \rangle$:

- N_a ... broj apstraktnih razreda paketa
- N_c ... ukupni broj razreda paketa
- ``računaju'' se samo razredi o kojima ovise vanjski paketi!

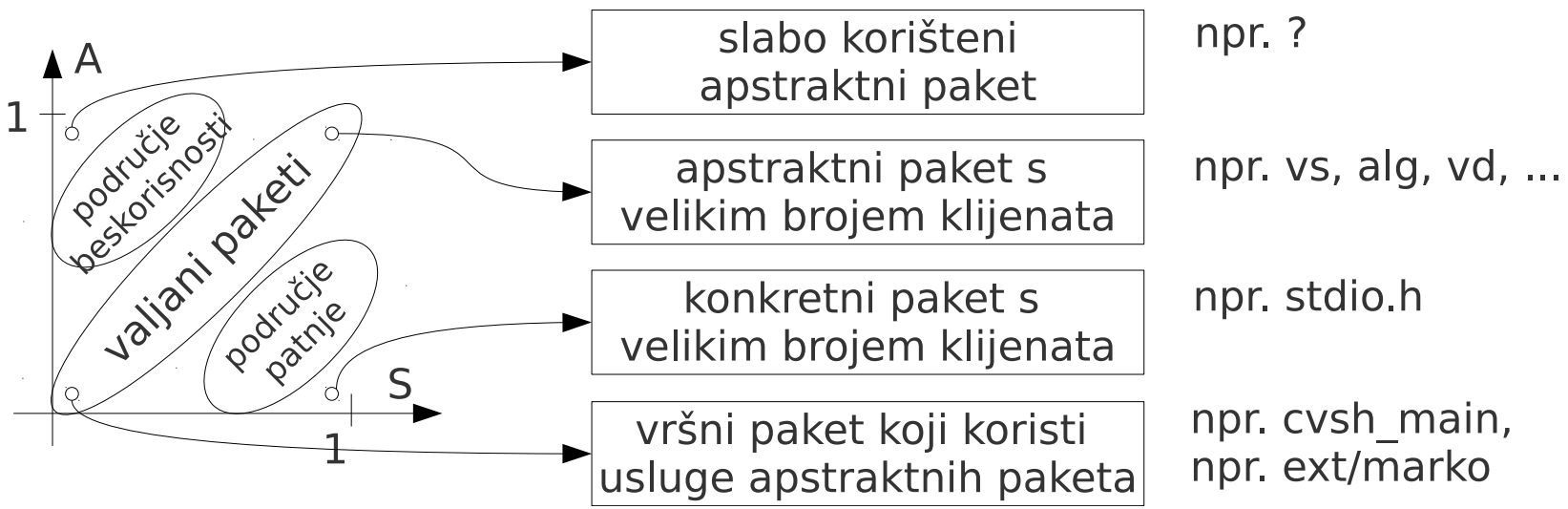
Pitanje: kolika je primjerena apstraktnost paketa?

Načelo primjerene apstraktnosti:
paket treba biti toliko apstraktan (A) koliko je i stabilan (S)!

Oblikovanje paketa: primjerena apstraktnost

Razmotrimo svojstva paketa u ovisnosti o koordinatama (S, A)

- $S >> A$ ... područje patnje (može biti OK, ako je paket zreo)
- $A >> S$ ... područje beskorisnosti (simptom pretjeranog oblikovanja)
- $A = S$ ... dobro oblikovani paketi (idealno A=S=0 ili A=S=1)
- $D' = |A - S| > x_{\text{th}} \Rightarrow$ paket je kandidat za prekrajanje



Zaključak

Vidjeli smo da su logički (nadogradivost, podatnost, razumljivost) i fizički (lako ispitivanje) ciljevi oblikovanja **kompatibilni**

Dobra struktura ovisnosti: **plitka**, **nepovezana** i **bez ciklusa**!

Kako je postići?

Pristupi, prema razini apstrakcije i redoslijedu korištenja:

- zdrav razum (KISS, YAGNI, DRY, SPOT, PoLA)
- **tehnike** (strukturiranje, enkapsulacija, apstrakcija i polimorfizam)
- **načela**: NNBP, LNS, NIO, NJO, NIS, ograničavanje ovisnosti, izbjegavanje cikličkih ovisnosti
- oblikovni obrasci (design patterns)
- arhitektonski obrasci (architectural patterns)

Riječ prije kraja: overdesign

Nikad ne zaboraviti: zadatak programskog inženjera je borba protiv složenosti!

Linus Torvalds:

> *Nobody should start to undertake a large project. You should start with a small trivial project, and you should never expect it to get large. If you do, you'll just overdesign and generally think it is more important than it likely is at that stage. Or, worse, you might be scared away by details. Don't think about some big picture fancy design. If it doesn't solve some fairly immediate need, it's almost certainly overdesigned.*

Oblikovanje je važno ispravno **dozirati**: glavni kriterij je jednostavnost izvedbe.

Važan element uspjeha je **iskustvo**!