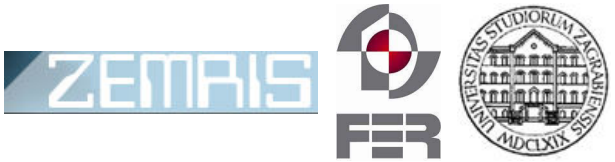


Oblikovni obrasci u programiranju

Načela programskog oblikovanja

Siniša Šegvić

Zavod za elektroniku, mikroelektroniku,
računalne i inteligentne sustave
Fakultet elektrotehnike i računarstva
Sveučilište u Zagrebu

Sadržaj

Načela programskog oblikovanja

- **Simptomi** urušavanja programa
- **Primjer** (problem,rješenje)
- **Metode** (dijagrami komponenti i razreda,C++,C)
- Načela **logičkog** oblikovanja
- Načela **fizičkog** oblikovanja
- **Zaključak**

Simptomi

Vidjeli smo da arhitektura određuje **dinamička svojstva** projekta

Zašto je teško program napisati kako treba "isprve"?

- slabo početno znanje o domeni
  (nepotpuni, pogrešni zahtjevi)
- nesavršeni arhitekti (to smo mi!)
- živimo u dinamičnom svijetu
  (zahtjevi se mijenjaju)

Programiranje je teško: kako preživjeti?

- konstruktivnije tražiti rješenje nego tražiti krivicu!
- (jedno) rješenje: arhitekturu postupno usklađivati sa saznanjima o domeni!

SIMPTOMI: POPIS

Koji su **simptomi** (zadasi) arhitekture koja propada, urušava se, ili je naprosto neprikladna?

- krutost: teško mijenjanje, ¬podatnost
- krhkost: lako unošenje grešaka
- nepokretnost: teško višestruko korištenje
- trenje (viskoznost): teško provođenje arhitektonskih načela

Specifični primjeri patologije (anti-obrasci):

- pretjerana međuovisnost: "spaghetti code" AP
- pretjerana složenost: "Swiss-Army Knife" AP
- ponavljanje umjesto ponovnog korištenja: "reinvent the wheel" AP

Simptomi: krutost

Kada je programski sustav **krut**?

- program je teško promijeniti, čak i na jednostavne načine
- svaka promjena domino-efektom zahtijeva nove promjene u povezanim modulima
  (najčešće zbog **eksplicitne** međuovisnosti komponenata)
- sitna "poludnevna" intervencija pretvara se u višednevni maraton istitravanja promjene kroz sustav
- **strah** od ispravljanja problema koji nisu kritični
- pozitivna povratna veza:
  krutost $\longrightarrow$ izbjegavanje promjene $\longrightarrow$ još veća krutost

SIMPTOMI: KRHKOST

Kada je programski sustav **krhak**?

- tendencija programa da "puca po šavu" nakon promjene
  - uzrok: **implicitna** međuovisnost uslijed ponavljanja (**DRY**!)
    - "neizbježno" (heterogenost, komentari, dokumentacija, jezik)
    - nepažljivost na razini komponente
    - nestrpljivost u održavanju (jača s **trenjem**)
    - neusklađenost razvojnog tima (komunikacija!)
  - **jedna** konceptualna izmjena mora se unijeti na **više** mjesta
  - propusti rezultiraju suptilnim greškama koje je teško pronaći
- krhkost jača krutost (Y2K fijasko)
- ponovo, posljedična nelagoda i gubitak samopouzdanja uzrokuju isforsirane i neplanske reakcije
- pozitivna povratna veza evoluciju čini teškom

Simptomi: nepokretnost

Nepokretnost programskog sustava:

- otežano višestruko korištenje prethodno razvijenih modula
  - pretjerana međuovisnost zbog neadekvatnih sučelja modula
  - tendencija povećane krhkosti ili krutosti
  - ključna mikroarhitektura komponenti i podsustava
- nesuđeni novi korisnik otkriva da postojeći modul ima previše "prtljage" koju nije lako eliminirati
- moduli se pišu iznova, umjesto evolucije kroz ponovno korištenje

Simptomi: viskoznost

Kada je programski sustav **viskozan**?

Kakve vrste **trenja** se mogu pojaviti?

- **viskoznost arhitekture** otežava donošenje dugoročnih rješenja (zakrpe traže puno manji kratkoročni angažman)
  - zaobilazi se održavanje konzistencije arhitekture
- **trenje razvojnog procesa** vezano je uz sporu i neefikasnu razvojnu okolinu
  - spori sustav za verziranje povlači rjeđe sinkronizacije kôda, kasnije otkrivanje problema
  - ako je prevođenje sporo, inženjeri unose "zakrpe" umjesto da prekroje arhitekturu (**ISP**!)

Simptomi: uzroci

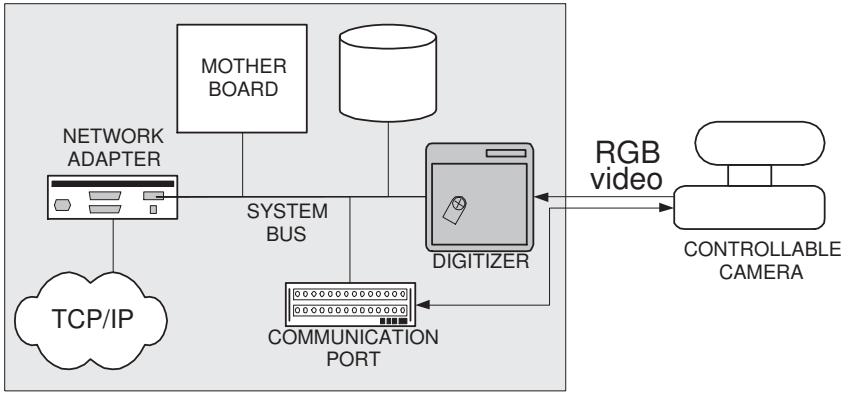
Zajednički nazivnik **patologije**:

- zbog neadekvatnog oblikovanja ili izmijenjenih zahtjeva dolazi do neplaniranih izmjena
- izmjene uzrokuju degradiranje arhitekture:
  - neželjena međuovisnost komponenata ili funkcionalnosti
  - ponavljanje u implementaciji i arhitekturi
- degradacija arhitekture uzrokuje otežanu evoluciju
- kako je evolucija arhitekture glavni mehanizam obrane, efekt je analogan AIDS-u
- rješenje: očuvanje integriteta arhitekture!
- izbjegavanje i eksplicitnih i implicitnih međuovisnosti može zahtijevati kompromise!

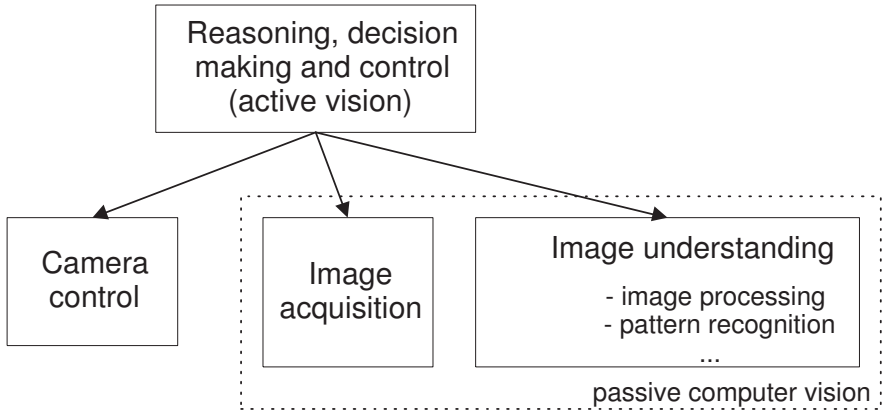
SIMPTOMI: ŽIVO BLATO

PRIMJER

Sustav aktivnog računarskog vida:



(a)



(b)

a) sklopovlje u sustavu aktivnog računalnog vida

b) četiri glavna programska podsustava

PRIMJER: V1

Zamišljena petlja obrade (samo pasivni vid):

```
void mainLoop(){
  ...
  while(1){
    img_data img;
    grabber.getFrame(img);
    algorithm.process(img);
    window.putFrame(
      algorithm.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_data img;
    switch(myVS){
    case VSFile:
      file.getFrame(img);
      break;
    case VSGrab:
      grabber.getFrame(img);
      break;
    }
    algorithm.process(img);
    window.putFrame(algorithm.imgDst());
  }
```

Međutim, sad vidimo da bi bilo dobro i obrađene slike spremati na disk...

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_data img;
    switch(myVS){
      ...
    }
    algorithm.process(img);
    switch(myVD){
    case VDFile:
      file2.putFrame(algorithm.imgDst());
      break;
    case VDWindow:
      window.putFrame(algorithm.imgDst());
      break;
    }
  }
```

Cool :-)

PRIMJER: V4?

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

...i različite digitalizatore...

...i različite formate ulaznih slika...

...i udaljenu obradu, uz prijenos slike preko mreže...

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

```
// 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 se našli u nebranom grožđu!
(programiranje je stresan posao)

verzija v4 je kruta i viskozna!

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

promijenili se zahtjevi!

što ćemo sad?

naravno, prekrojiti arhitekturu (uskladiti je sa znanjem o domeni).

PRIMJER: V5

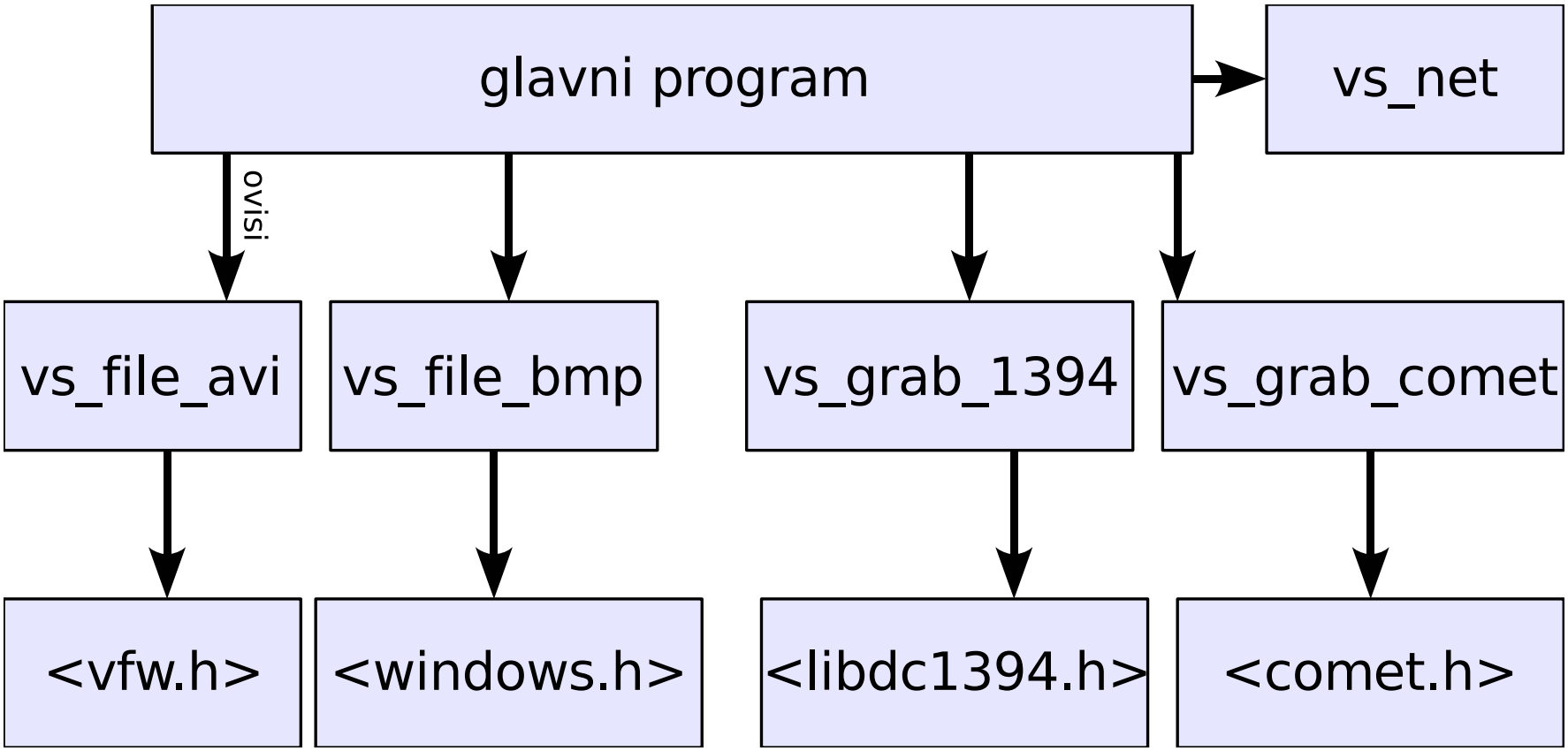
Nova petlja koristi (dinamički) **polimorfizam**, i u potpunosti je neovisna o konkretnom pribavljanju, obradi i spremanju slika!

```
class vs_base; // video source (getFrame)
class vd_base; // video destination (putFrame)
class alg_base;// image processing algorithm (process)
...
void mainLoop(vs_base& vs, alg_base& algorithm, vd_base& vd){
  std::vector<vd_win*> pvdWins(4);
  for (int i=0; i<algorithm.nDst(); ++i){
    pvdWins[i]=new vd_win;
  }
  while(!vs.eof()){
    img_data img;
    vs.getFrame(img);
    algorithm.process(img);
    for (int i=0; i<algorithm.nDst(); ++i){
      pvdWins[i]->putFrame(algorithm.imgDst(i));
    }
    vd.putFrame(algorithm.imgDst(0));
  }
}
```

Juhu :-)

PRIMJER: DIJAGRAM V3

Početna arhitektura programa:

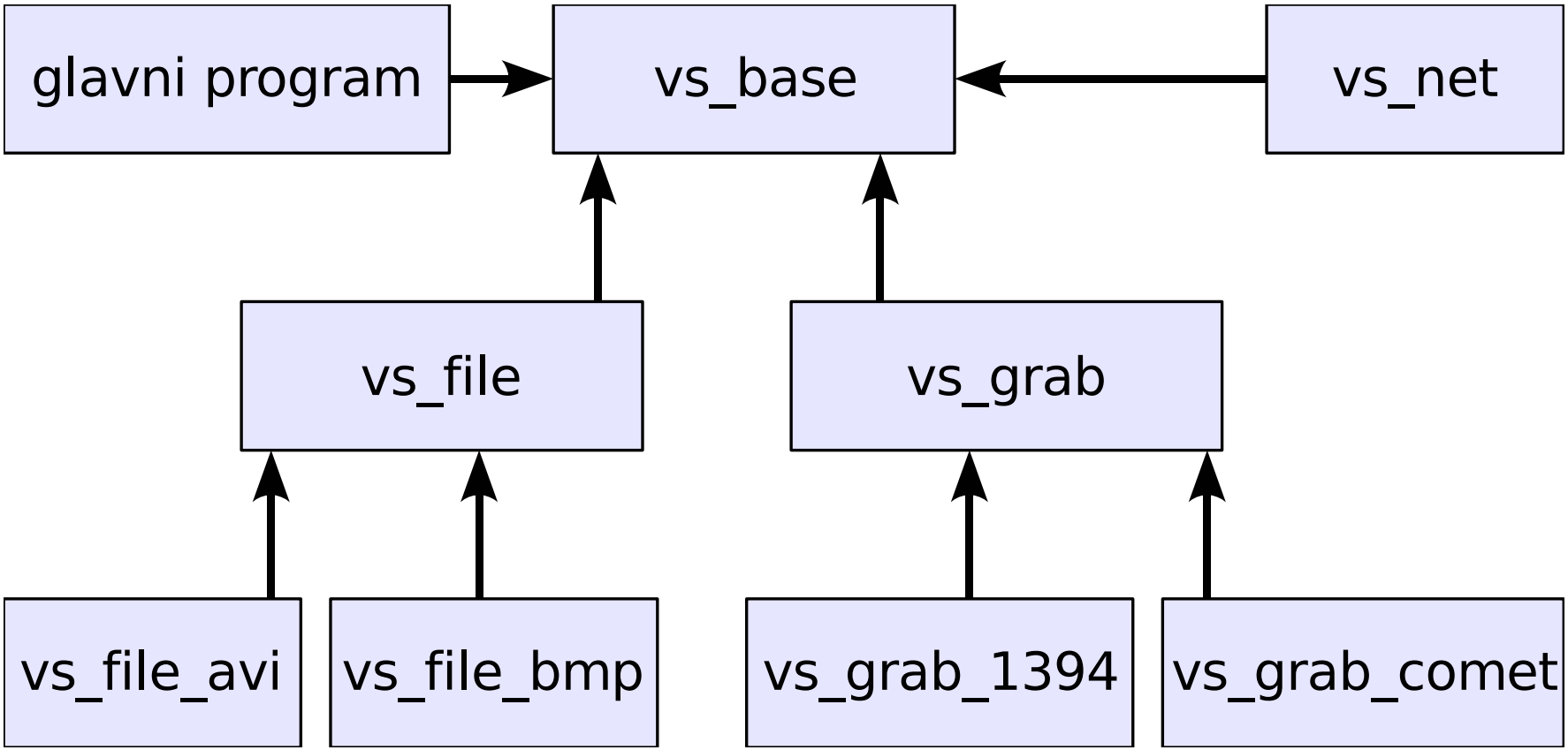


Glavna komponenta ne može se prevesti bez <comet.h>! (VLAP)

Održavanje "tehnoloških" komponenti utječe na glavni program

PRIMJER: DIJAGRAM V5

Krajnja arhitektura programa:



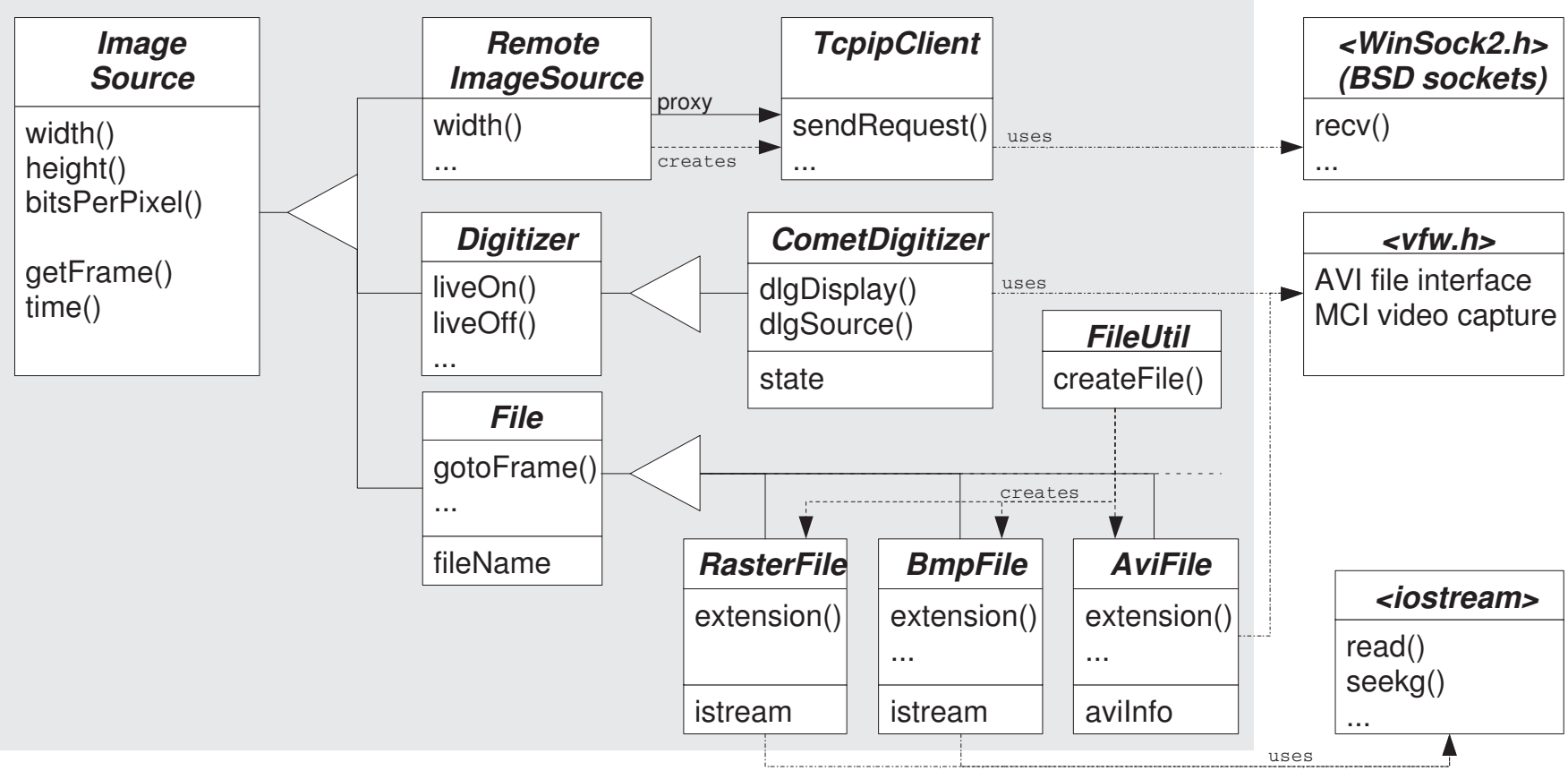
Širimo funkcionalnost **bez prevođenja** glavne komponente!

Ovisnost usmjerena od složenog prema apstraktnom!

Smanjen pritisak ovisnosti na stožer aplikacije!

PRIMJER: STABILNOST

Povijesni dijagram: paket za pribavljanje slike u 2000 g. (5KLoC):



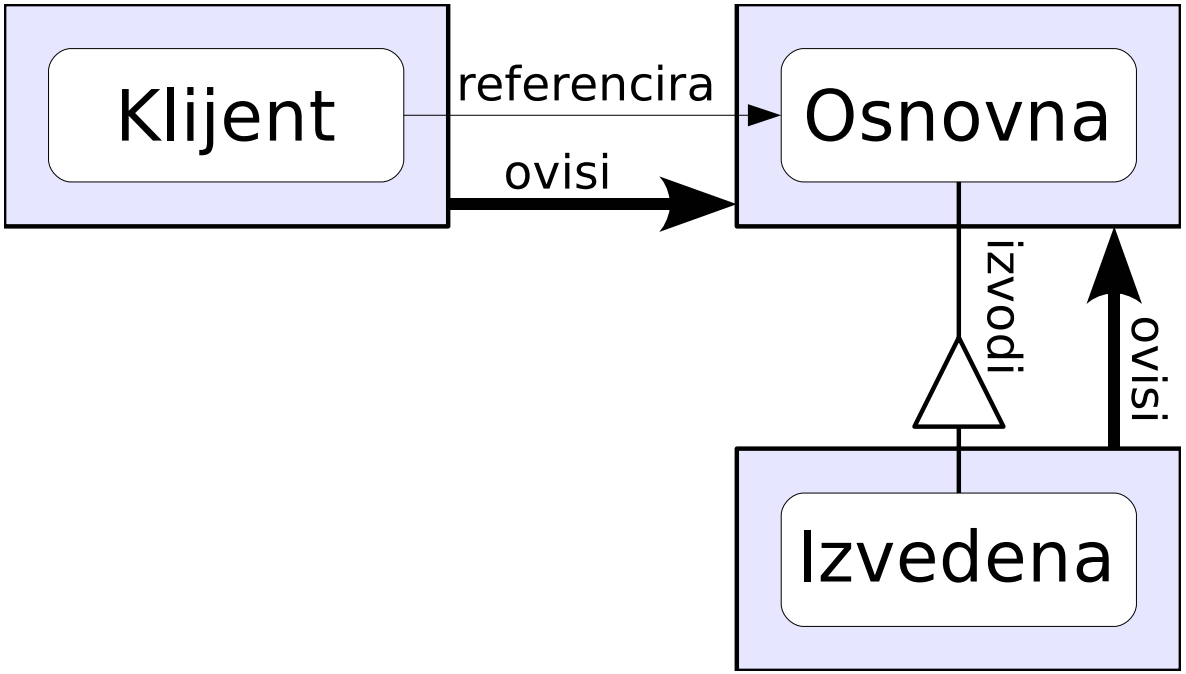
Metode

Terminologija:

- načelo vs. obrazac: načelo vrijedi uvijek, obrazac zadovoljava načela kod stanovite klase problema
- 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`)
    - komponenta sadrži jednu ili više logičkih jedinica
    - komponenta: temeljna jedinica pri **verziranju** i **testiranju**!
- dobra arhitektura poštuje i logička i fizička načela

METODE: LAKOS96

Mješoviti dijagram razreda i komponenata:



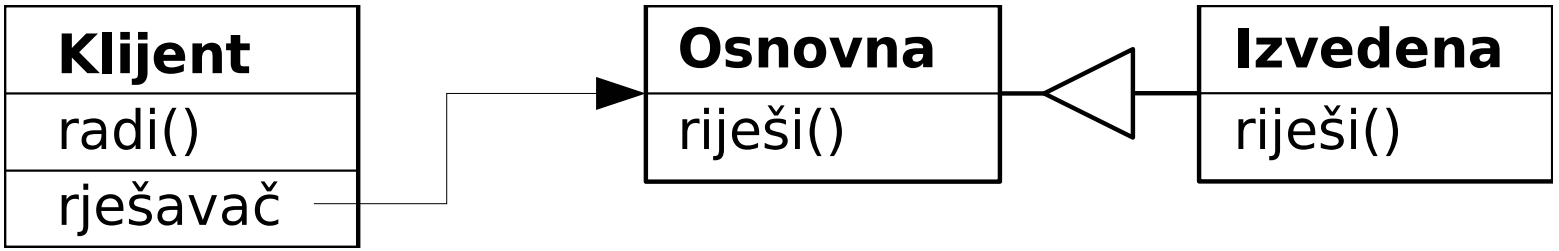
Logički odnosi među razredima:

izvodi (nasljeđuje), referencira, sadrži, stvara, ...

Fizički odnosi među komponentama:

ovisnost (pri prevođenju, pri povezivanju)

METODE: OMT VS C++




```cpp
//==== test.cpp
#include "client.hpp"
#include "derived.hpp"
int main(){
  Derived d;
  Client c(d);
  c.operate();
}
//==== client.hpp
#include <iostream>
#include "base.hpp"
class Client{
public:
  Client(Base& b): solver_(b){}
  void operate(){
    std::cout <<solver_.solve() <<"\n";
  }
private:
  Base& solver_;
};
```

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

//==== derived.hpp
#include "base.hpp"
class Derived: public Base{
public:
  virtual int solve(){return 42;}
};

//==== derived2.hpp (not in diagram!)
#include "base.hpp"
class Derived2: public Base{
public:
  virtual int solve(){return 0;}
};
```

METODE: C++ VS C

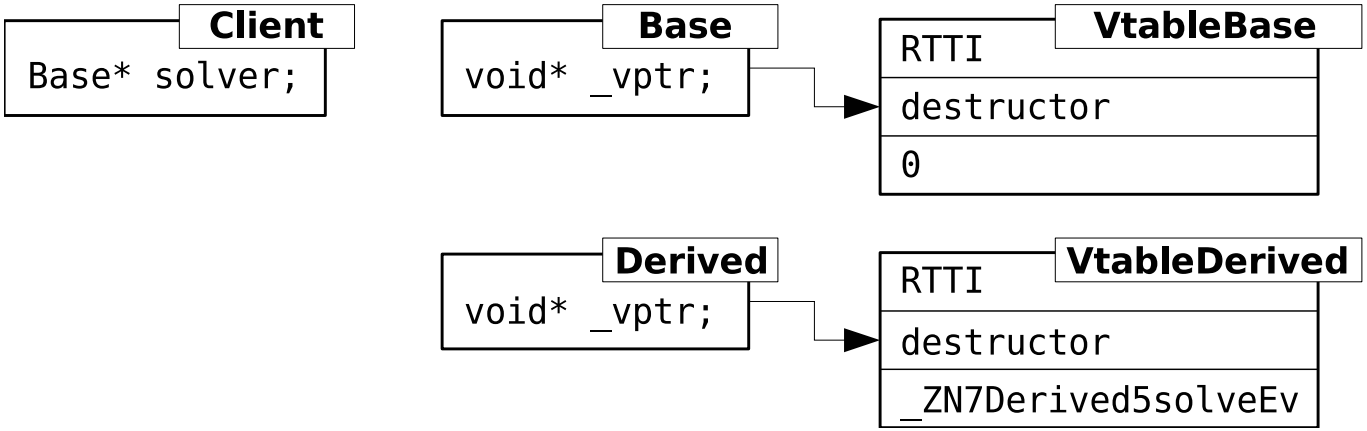
Demistificirani **objektni model** C++-a:

Što se događa kod poziva `c.operate()`?

- `_ZN6Client7operateEv(&c);`

Što se događa kod poziva `solver_.solve()`?

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




```
\url{http://www.codesourcery.com/cxx-abi/cxx-vtable-ex.html}
\url{http://theory.uwinnipeg.ca/gnu/gcc/gxxint_15.html}
```

METODE: GENERICI

Generičko programiranje: **proširena** gramatika omogućava **parametrizirane** konstrukte

```
#include <iostream>

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

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

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

METODE: GENERICI (2)

Prevođenje se **odgađa** do trenutka kad parametri postaju poznati (instanciranje konstrukta, koristi se **osnovna** gramatika)

Ada (1977), C++ (1994), Haskell (2001), Java (2004), C# (2005);

Posebno prikladno za biblioteke u statički tipiziranim jezicima

- npr. STL (Stepanov 1981-1994): i učinkovitost i prilagodljivost
- dinamički tipizirani jezici (Smalltalk, Python)?

Sofisticirane mogućnosti:

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

```
#include <iostream>
int main(){
  std::cout <<"10!="
            <<factorial<10>() <<"\n";
}
//prints: 10!=3628800
```

METODE: STL

Ortogonalnost algoritama i spremnika:

```
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>

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

  double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
  std::reverse(A, A + 6);
  std::list<double> L(A, A+6);
  std::reverse(L.begin(), L.end());
  std::list<double>::iterator it = L.begin();
  while (it!=L.end())
    std::cout <<"L " <<*it++ <<"\n";
}
```

Parametrizirani funkcijski objekti:

```
#include <list>
#include <algorithm>
#include <functional>
#include <iostream>

int main(){
  int x[]={3,4,7,5,4,6,8,3};
  std::list<int> L(x, x + sizeof(x)/sizeof(*x));
  int nv = std::count_if (L.begin(), L.end(),
    std::bind2nd (std::greater<int>(), 5));
  std::cout <<"Result: " <<nv <<"\n";
}
```

METODE: GENERICI VS OOP

Odnos predložaka i virtualnih poziva:

- manja složenost izvršnog kôda (prostorna i vremenska)
- fleksibilnost predložaka dostupna samo do trenutka prevođenja
- predlošci prikladni za manje, često korištene programske jedinice

```
struct ImageOOP{
  virtual Pixel&
  getPixel(int x, int y) =0;
};
struct ImageInt: public ImageOOP{
  virtual PixelInt&
  getPixel(int x, int y){
    return buf_.at(y*height()+x);
  }
private:
  //sizeof(PixelInt)?
  std::vector<PixelInt> buf_;
};
void clearImageOOP(ImageOOP& img){
  for (int i=0; i<img.width(); ++i)
    for (int j=0; j<img.height(); ++j)
      img.getPixel(i,j).setScalar(0);
}
```

```
template <typename Pixel>
struct ImageGP{
  inline Pixel& getPixel(int x, int y){
    return buf_.at(y*height()+x);
  }
private:
  std::vector<Pixel> buf_;
};
template <typename T>
void clearImageGP(ImageGP<T>& img){
  for (int i=0; i<img.width(); ++i)
    for (int j=0; j<img.height(); ++j)
      img.getPixel(i,j).setScalar(0);
}
// * smaller footprint (same as C)
// * faster pixel access (same as C)
// * need to improvise in order not to
//   force clients to use templates
```

Logička načela

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

- otvorenost za proširenje, zatvorenost za promjene (OCP)
  dodavanje funkcionalnosti bez ponovnog prevođenja klijenata
- nadomjestivost osnovnih razreda (LSP)
  učenik 7. razreda mora znati sve što i učenici 4. razreda!
- inverzija i injekcija ovisnosti (DIP, DI)
  poticanje ovisnosti o stabilnim apstraktnim komponentama
- jedinstvena odgovornost (SRP)
  razredi modeliraju jedinstvene koncepte
- razdvajanje sučelja (ISP)
  klijenti ne ovise o onom što ne koriste

Logička načela: OCP

Načelo zatvorenosti (uz zadržavanje otvorenosti za promjene)

- promjena ponašanja moguća **bez mijenjanja** postojećeg kôda
- proširenja ne utječu na klijente, uključujući ispitne komponente
- **ideja**: stari kôd "radi" s novim kôdom!
- dva načina primjene principa:
  1. nasljeđivanje implementacije [meyer88]
     novi kôd koristi staru implementaciju preko izvedenog sučelja
     (NB: danas se za to preferira **kompozicija**!)
  2. nasljeđivanje sučelja polimorfizmom [martin96]
     novi kôd nudi novu implementaciju iza starog sučelja
- vrijeme je pokazalo da kasniji pristup nudi veće mogućnosti
- vezano uz ideju **skrivanja informacije** [Parnas72]

LOGIČKA NAČELA: PROCEDURALNI OCP?

Proceduralni stil uzrokuje kruti kôd:

```
struct Shape{
  enum EType {circle, square};
  EType type_;
};
struct Circle{
   Shape::EType type_;
   double radius_;
   Point center_;
};
struct Square{
   Shape::EType type_;
   double radius_;
   Point center_;
};
void drawSquare(struct Square*);
void drawCircle(struct Circle*);
```

```
void drawShapes(Shape** list, int n){
  for (int i=0; i<n; ++i){
    struct Shape* s = list[i];
    switch (s->type_){
    case Shape::square:
      drawSquare((struct Square*)s);
      break;
    case Shape::circle:
      drawCircle((struct Circle*)s);
      break;
    default:
      assert(0);
      exit(0);
    }
  }
}
```

Rješenje je strukturirano i jasno, ali ipak kruto:

- mukotrpno dodavanje novih objekata
- teško ispitivanje funkcije `drawShapes()`
- ponavljanje **case** konstrukcije u `drawShapes()` i `moveShapes()`

LOGIČKA NAČELA: OCP – OOP

Polimorfni OCP u objektno orijentiranom programiranju

- prijelaz sa starog sučelja na novi kôd dinamičkim polimorfizmom
- C,C++: prijelaz putem pokazivača iz tablice virtualnih funkcija

Sa stanovišta arhitekture, puno bolje rješenje:

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

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

.

LOGIČKA NAČELA: OCP – GP

Polimorfni OCP u **generičkom** programiranju (**statički polimorfizam**)

- proširenje tijekom prevođenja, ekspanzijom **predložaka**
- GP vs DP: teži razvoj, bolja učinkovitost, komplementarnost

```
template<typename T>
struct Array {
  Array(int sz=10): 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)]; }
  Array(const Array<T>&);                  //TODO!!
  Array<T>& operator= (const Array<T>&);//TODO!!
private:
  int size_;
  T*  data_;
  inline int check(int i) const {
    assert (i>=0 && i<size_); // or: throw "bound check error";
    return i;
  }
};// example: Array<int> X(20);
```

- U odnosu na C-polje: jednaka učinkovitost$^{(R)}$, provjera pristupa$^{(D)}$, automatsko otpuštanje memorije, mogućnosti proširenja

LOGIČKA NAČELA: OCP – STL MAP

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

int main(){
  typedef std::map <std::string,int> MyMap;
  MyMap wordcounts;
  std::string s;

  while (std::cin >> s){
    wordcounts[s]++;
  }

  MyMap::iterator it=wordcounts.begin();
  while (it != wordcounts.end()){
    std::cout <<it->first <<' '
              <<it->second <<"\n";
    ++it;
  }
}
```

LOGIČKA NAČELA: OCP – STL SET

```
#include <set>
#include <iostream>
#include <iterator>
using namespace std;

struct ltstr{
  bool operator()(const char* s1, const char* s2) const{
    return strcmp(s1, s2) < 0;
  }
};

int main(){
  const char* a[] = {"mirko", "burek", "slavko", "foobar"};
  const char* b[] = {"burek", "cevap", "foobar"};
  set<const char*, ltstr> A(a, a + sizeof(a)/sizeof(*a));
  set<const char*, ltstr> B(b, b + sizeof(b)/sizeof(*b));

  cout << "Union: ";
  set_union(A.begin(), A.end(), B.begin(), B.end(),
            ostream_iterator<const char*>(cout, " "), ltstr());
  cout << endl;

  set<const char*, ltstr> C;
  set_difference(A.begin(), A.end(), B.begin(), B.end(),
                 inserter(C, C.begin()), ltstr());
  cout << "Set C (difference of A and B): ";
  copy(C.begin(), C.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;
}
```

LOGIČKA NAČELA: OCP – IMPLIKACIJE

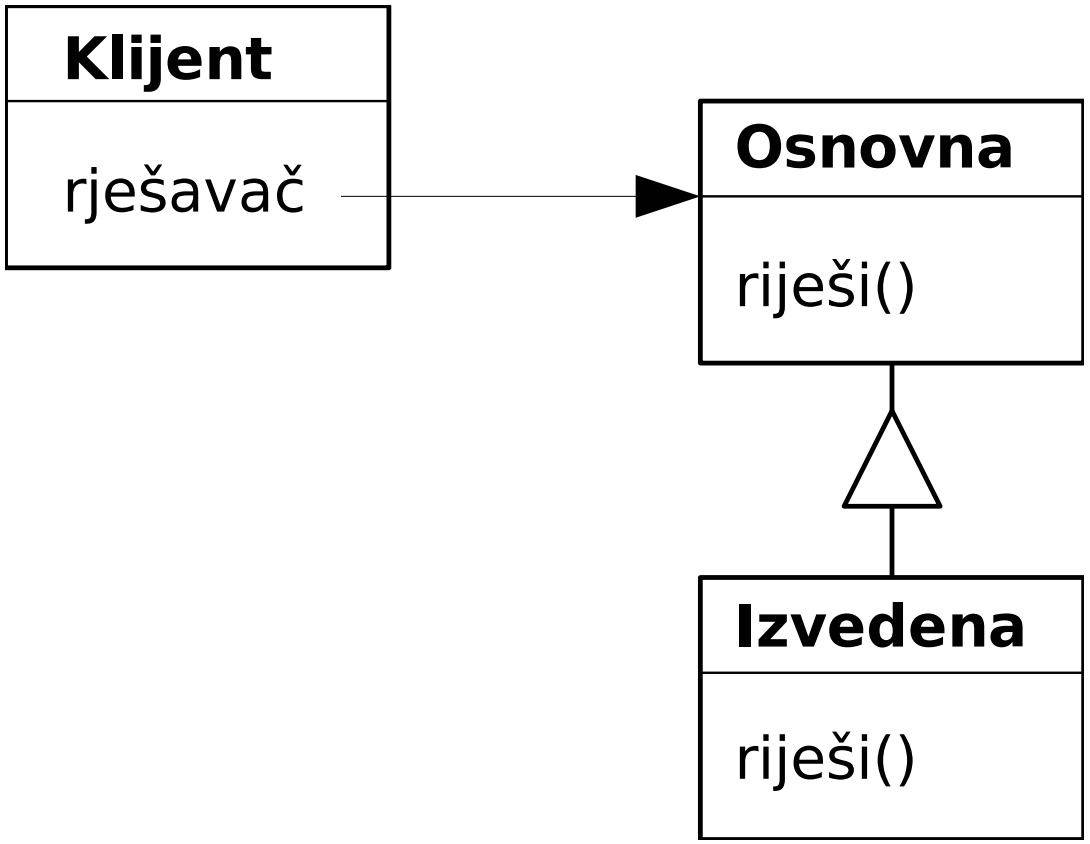
Koncepti koji pospješuju OCP:

- **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 OCP)!
- **apstraktni razredi** (čiste apstrakcije):
  - nemaju podatkovnih elemenata $\Rightarrow$ enkapsulirani
  - imaju virtualne funkcije
- **generički programi**: injekcija novog kôda prilikom instanciranja

LOGIČKA NAČELA: LSP

Načelo nadomjestivosti osnovnih razreda

- AKA Liskovino načelo supstitucije [Liskov93]
- osnovni razredi moraju se moći nadomjestiti izvedenim klasama
- student 3. godine mora biti upoznat s gradivom 1. godine



LOGIČKA NAČELA: LSP – ISA

Načelo upućuje na pravilnu upotrebu nasljeđivanja

- Nasljeđivanje modelira relaciju **je_vrsta_od** (IS_A_KIND_OF)
- pravnim jezikom, to znači da u odnosu na metode bazne klase:
  - preduvjeti u izvedenim klasama ne mogu biti pojačani
  - postuvjeti u izvedenim klasama ne mogu biti oslabljeni
- simptom patologije: klijenti moraju propitivati imaju li posla s izvedenim razredom prijestupnikom
- patologija: kršenje OCP-a, jer klijent se mora mijenjati kad unesemo izvedeni razred

LOGIČKA NAČELA: LSP – PRIMJER 1

```
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);
}
```

LOGIČKA NAČELA: LSP – PRIMJER 2

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

void client(Bird& b){
  b.fly();
  assert(b.altitude()>0);
}
```

LOGIČKA NAČELA: LSP – PRIMJER 3

```
class vs_base{
protected:
  vs_base();
public:
  virtual ~vs_base();

public:
  virtual int count() =0;
  virtual int tell() =0;
  virtual void seek(int) =0;
// ...
};

class vs_file:
  public vs_base
{
// ...
public:
  virtual int count();
  virtual int tell();
  virtual void seek(int);
// ...
};
```

```
class vs_grab:
  public vs_base
{
private:
  int dummyFrame_;
// ...
public:
  virtual int count(){
    return INT_MAX;
  }
  virtual int tell(){
    return dummyFrame_;
  }
  virtual void seek(int where){
    dummyFrame_=where;
  }
// ...
};
```

.

LOGIČKA NAČELA: LSP – IMPLIKACIJE

Nasljeđivanje modelira relaciju **je_vrsta_od**:

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

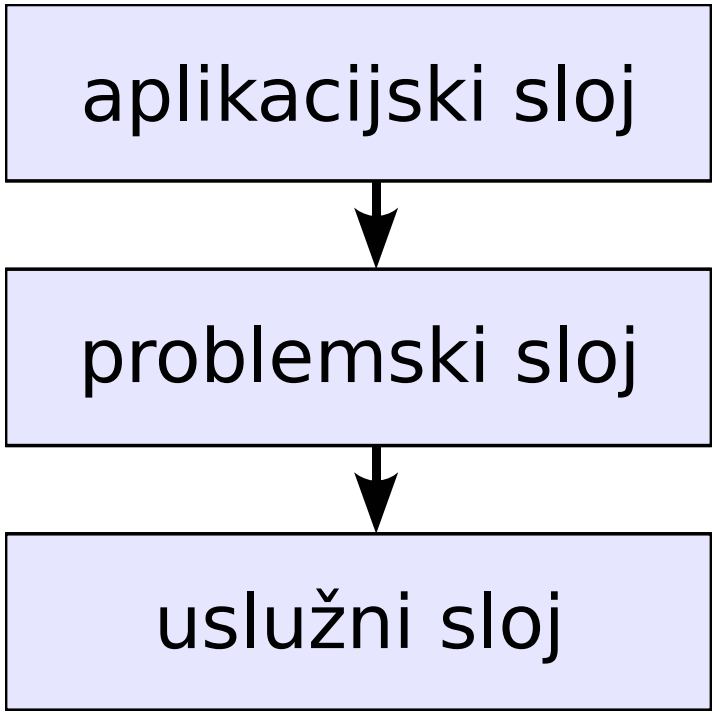
Kršenje LSP-a može se popraviti na 3 načina:

1. pojednostavniti funkciju osnovnog razreda
   (pojačati preduvjete, oslabiti postuvjete)
2. povećati složenost izvedenog razreda
   (oslabiti preduvjete, pojačati postuvjete)
3. odustati od rodbinskog odnosa

LOGIČKA NAČELA: DIP

Načela inverzije i injekcije ovisnosti

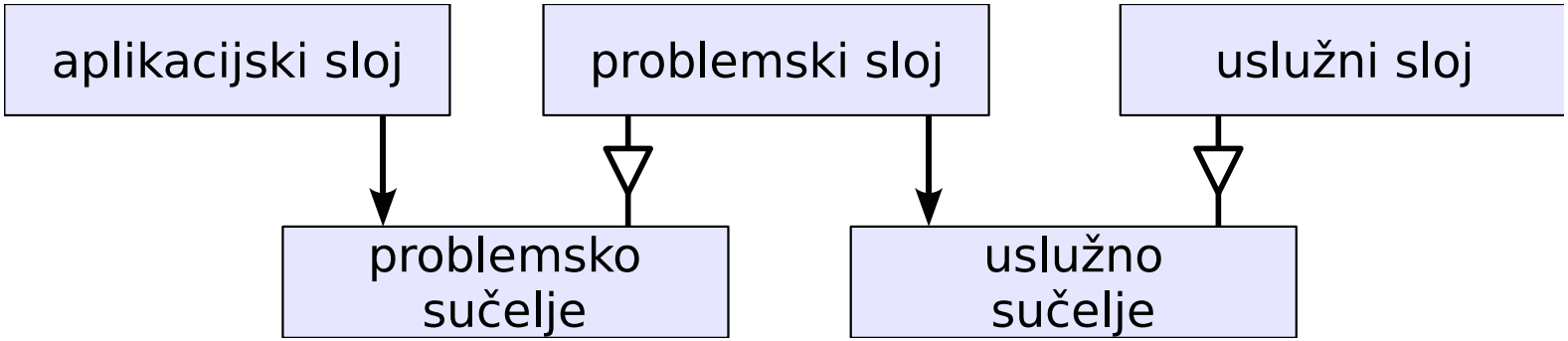
- vidjeli smo da je nadomjestivost nužni uvjet zatvorenosti
- sada: strukturne implikacije zatvorenosti i nadomjestivosti vode k načelu inverzije ovisnosti
- nedorečenosti inverzije ovisnosti rješavaju se pomoćnim načelom injekcije ovisnosti
- **proceduralni stil** teži piramidalnoj strukturi međuovisnosti



- loše: moduli visoke razine ovise o izvedbenim detaljima
- loše: program se ne može mijenjati bez domino-efekta

LOGIČKA NAČELA: DIP – OOP

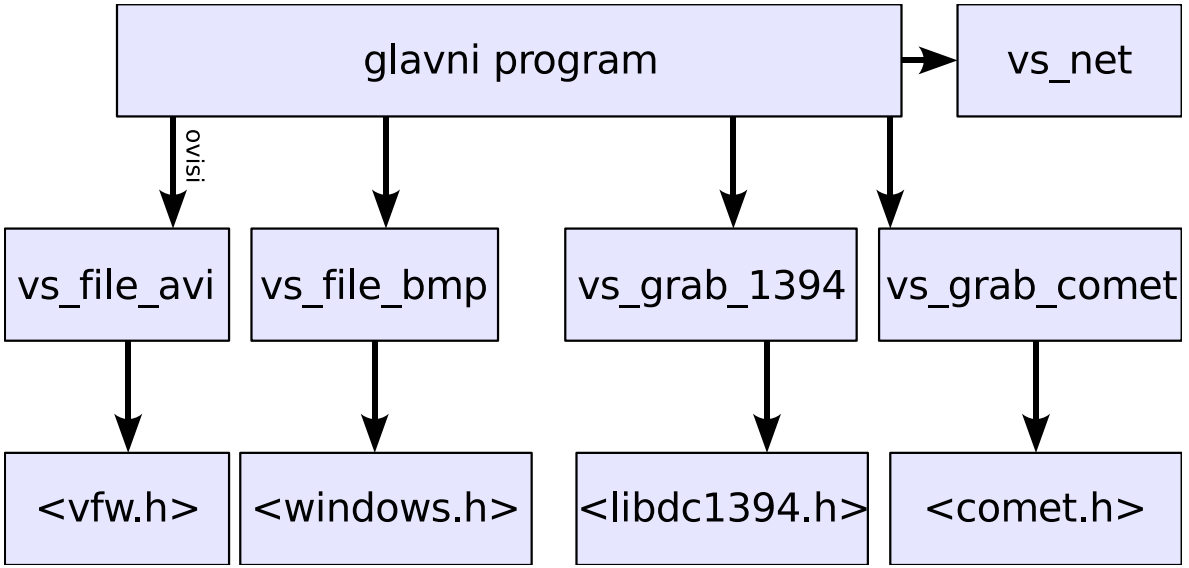
- nedostatci se mogu riješiti promjenom strukture ovisnosti:



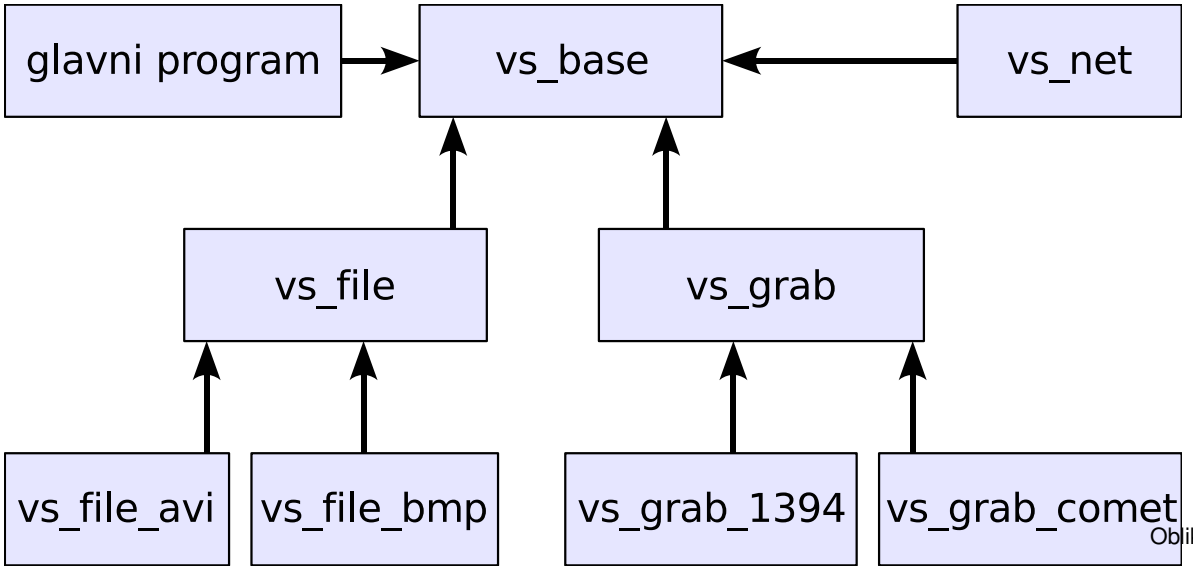
- u novoj organizaciji, ovisnosti idu prema apstrakcijama (koje su stabilne jer nemaju implementaciju)
- ne ovisimo više o nepostojanim modulima s detaljnim implementacijama: kažemo da je ta ovisnost **invertirana**
- koristi inverzije ovisnosti:
  - piramidalna struktura preokrenuta: **ovisimo o apstrakcijama**
  - promjer grafa je smanjen (putevi ovisnosti su **kraći**)

LOGIČKA NAČELA: DIP – PRIMJER

PRIJE:



POSLIJE:



LOGIČKA NAČELA: DIP – IMPLIKACIJE

- ovisnosti u projektu **u načelu** trebaju ići prema apstrakcijama (**NE** od modula visoke razine prema modulima niske razine)
  - komponenta bez implementacije se rijeđe mijenja
  - apstrakcije omogućavaju širenje 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 načela injekcije ovisnosti i obrasca tvornice:
    - **r1**: već i lokalizirana, ograničena ovisnosti je korisna
    - **r2**: neovisnost se postiže sofisticiranim oblikovnim obrascima

Logička načela: DI

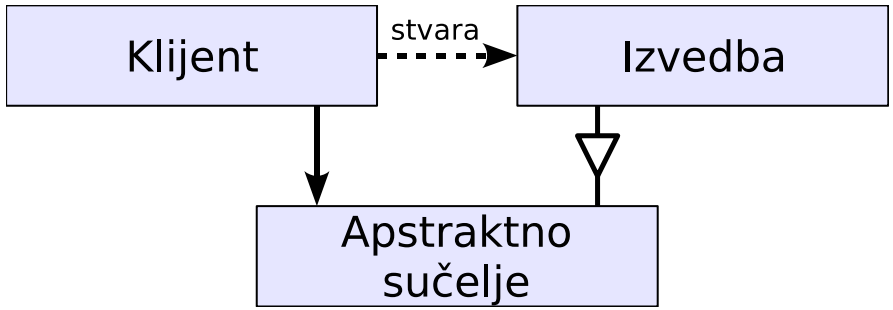
Načelo injekcije ovisnosti: eskalacija stvaranja konkretnih objekata

- ovisnost zbog stvaranja izvlači se u zasebnu komponentu!
- postiže se inverzija ovisnosti, te lokalizacija ovisnog koda

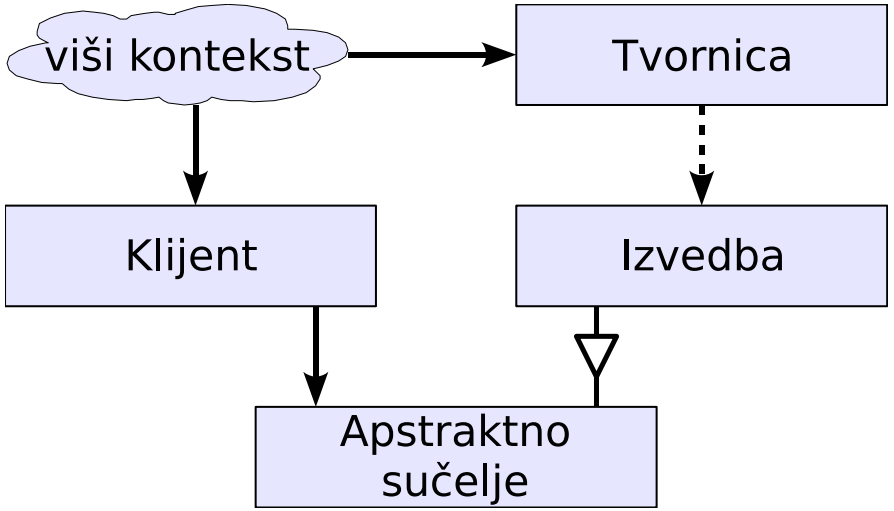
```
//====  example.hpp
class Example {
  ConcreteDatabase myDatabase;
public:
  void DoStuff() {
    myDatabase.GetData();
    // ....
  }
};
//==== example2.hpp
class Example2 {
  AbstractDatabase& myDatabase;
public:
  Example2(AbstractDatabase& db):
    myDatabase(db) {}
public:
  void DoStuff() {
    myDatabase.GetData();
    // ...
  }
};
```

```
//==== testExample2.cpp
#include "example2.hpp"
// ...

int main(){
  // construct database
  MockDatabase* pdb = new MockDatabase();

  // construct test object (`DI`!)
  Example2 example(*pdb);

  // test behaviour #1
  example.DoStuff();
  assertGetDataWasCalled(*pdb);

  // test behaviour #2
  // ...

  // test behaviour #3
  // ...
}
```

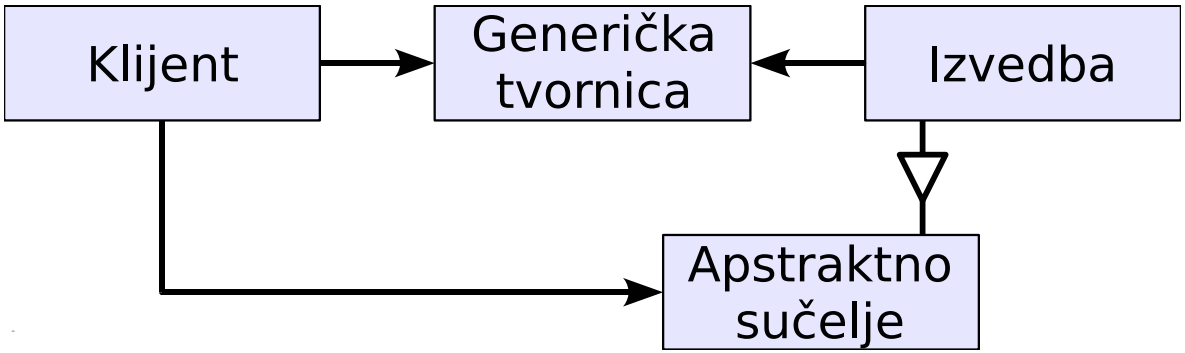
LOGIČKA NAČELA: DI – EFEKTI



1. bez injekcije ovisnosti



2. primjena obrasca tvornice



3. primjena generičke tvornice

Logička načela: SRP

Načelo jedinstvene odgovornosti

- Vidjeli smo kako se **nadogradivost** pospješuje zatvorenošću (OCP), nadomjestivošću (LSP) i inverzijom ovisnosti (DIP)
- sada razmatramo postizanje **podatnosti** suzbijanjem međuovisnosti
- načelo jedinstvene odgovornosti ili kohezije [DeMarco79]:

  ne smije postojati više od jedne odgovornosti modula!
- ideja: svaka odgovornost odgovara stabilnoj **osi promjene**:
  - **koherencija** modula pospješuje prirodnu raspodjelu poslova pri promjeni zahtjeva
  - dobivamo ortogonalan sustav (**podatnost** i **razumljivost**)
- s druge strane, ako modul ima više odgovornosti, među njih unosimo neprirodnu **međuovisnost** (krutost, krhkost, nepokretnost)

LOGIČKA NAČELA: SRP – PRIMJER

Razmotrimo još jednu arhitekturu programa za crtanje, gdje razred `Rectangle` obuhvaća više od jedne odgovornosti:

```
struct Shape{
  virtual ~Shape();
  virtual void rotate(double phi)=0;
  virtual void draw(Window& wnd)=0;
};

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

```
typedef MyList std::list<Shape*>;

void GUI::draw(
  MyList& shapes,
  Window& wnd)
{
  MyList::iterator it=shapes.begin();
  while (it!=shapes.end()){
    it->draw(wnd);
  }
}

void Geom::intersect(const Shape& sin1,
    const Shape& sin2, Shape& sout)
{
  //...
}
```

Loše: `Geom` ovisi o `GUI`, bez konceptualnog opravdanja

(`Geom` moramo prevesti kad god se promijeni sučelje `draw`)

Bolje: pokušati izvesti unificiranu `GUI::draw(Shape&, Window&)`

Logička načela: SRP – odgovornost

Što je **odgovornost** modula?

- odgovornost je kvant funkcionalnosti iz domene aplikacije:
  $\exists$ bijekcija prema **razlozima promjene** programskog sustava
- svaki modul bi trebao imati jednu, samo jednu odgovornost
  (Ali koju? To *mi* trebamo otkriti!)
- nekad teško pogoditi isprve: ono što se ispočetka čini kao jedna odgovornost, često se ispostavi kao srodnih odgovornosti (primjer)
- **analiza domene** (ono što smo rekli da je teško!) u mnogome se svodi na određivanje odgovornosti
- kad smo sigurni da moduli nemaju jedinstvenu odgovornost, arhitekturu treba prekrojiti (*refactor*) (što ranije to bolje!)

Logička načela: SRP – ortogonalnost

Jedinstvena odgovornost usko povezana s ortogonalnošću [Hunt00]:

- ortogonalnost $\Rightarrow$ promjena jedne odgovornosti ne utječe na ostale
- jedinstvene odgovornosti $\Leftrightarrow$ ortogonalni sustavi

Ortogonalnost implementacije vs ortogonalnost sučelja:

```
struct Car{
  virtual void brake(int howMuch);
  virtual void clutch(int howMuch);
  virtual void gas(int howMuch);
  virtual void switchGears(int pos);
};
```

```
struct OrthogonalCar:
  public Car
{
  virtual void accelerate(int howMuch);
  virtual void setDriveStyle(int which);
};
```

Obično preferiramo **ortogonalnost sučelja**!

Neortogonalne programske sustave valja izbjegavati!

Logička načela: SRP – zaključak

Načelo jedinstvene odgovornosti u temelju programskog inženjerstva

Ortogonalna konceptualizacija problema smanjuje parazitne međuovisnosti te pospješuje **održivu evoluciju**

Bez dovoljnog znanja o domeni, težimo neopravdanom grupiranju odgovornosti (npr, prikaz oblika i crtanje).

Kontekst za ispravnu raspodjelu odgovornosti **izranja** postepeno, kako naše razumijevanje problema postaje bolje

Metode: analiza domene, eksperimentiranje, pisanje prototipa, proučavanje srodnih programa

Pronalaženje i dekoreliranje odgovornosti dobar dio "onoga" o čemu se u programiranju radi

LOGIČKA NAČELA: ISP

Načelo razdvajanja sučelja:

- složeni koncepti čije korištenje ovisi o klijentu ipak se javljaju: ponekad zgodno napraviti kompromis sa SRP-om
- izradom **jedinstvenog** sučelja za takve koncepte
  - nepotrebno zbunjujemo pisce klijenata (SAK AP)
  - unosimo suvišne ovisnosti o nekorištenim elementima sučelja
- načelo sugerira da nekoherentnim konceptima klijenti **ne bi trebali** pristupati preko jedinstvenog sučelja:

  nije dobro primoravati klijente na ovisnost o sučelju koje ne koriste
- razmotrit ćemo **slabosti** nekoherentnih i "preopsežnih" ("pretilih") sučelja, te uvjete za njihovo javljanje
- pokazat ćemo kako **ublažiti** slabosti izdvajanjem sučelja

LOGIČKA NAČELA: ISP - PREOPSEŽNO SUČELJE

```
//====  door_v1.hpp
struct Door{
    virtual void Lock()   = 0;
    virtual void Unlock() = 0;
    virtual bool IsDoorOpen() = 0;
};

//====  timer_v1.hpp
struct Timer{
  void Register(int timeout,
      TimerClient* client);
};
struct TimerClient{
    virtual void TimeOut() = 0;
};

//==== door_v2.hpp
#include "timer.hpp"
struct Door:
  public TimerClient
{
  virtual void Lock();
  virtual void Unlock();
  virtual bool IsDoorOpen();
  virtual void TimeOut();
};
```

loše: sva vrata moraju se mijenjati kad god se promijeni mjerioc vremena!

```
//====  timer_v1.hpp
struct Timer{
  void Register(int timeout,
      int id, TimerClient* client);
};
struct TimerClient{
    virtual void TimeOut(int id) = 0;
};

//====  door_v3.hpp
struct Door{
    virtual void Lock()   = 0;
    virtual void Unlock() = 0;
    virtual bool IsDoorOpen() = 0;
};

//====  timedDoor.hpp
struct TimedDoor:
  public Door,
  public TimerClient
{
  virtual void TimeOut(int id);
};
```

puno bolje: samo specijalna vrata ovise o mjeriocu vremena!

LOGIČKA NAČELA: ISP – SAŽETAK

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

U žargonu, ortogonalna sučelja **mixin** (C++) i **interface** (java, C#)

Razdvajanjem sučelja dobivamo prihvatljivu organizaciju programa za crtanje, i uz nekoherentno sučelje `GraphicShape`

```
struct Shape{
  virtual void rotate(double phi)=0;
};

struct Graphic{
  virtual void draw(Window&) =0;
};

struct GraphicShape:
  public Shape, public Graphic
{};
```

```
void Geom::intersect(const Shape& sin1,
    const Shape& sin2, Shape& sout)
{ //...
}

typedef MyList std::list<GraphicShape*>;

void GUI::draw(
  MyList& shapes, Window& wnd)
{ //...
}
```

Najbolje je ipak takve koncepte izbjeći ako je moguće (SRP)!

Fizička načela

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

Od traženih dobrih osobina ostalo je **lako ispitivanje**

Ispitivanje najzgodnije provoditi nad **datotekama** izvornog koda: analiziramo fizičku arhitekturu

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

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

Fizička načela: o ispitivanju

Problem testiranja — sofisticirana funkcionalnost iza škrtog sučelja:

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

Lakše testirati n komponenti **zasebno** nego sve moguće interakcije (2^n)
inkrementalno vs sveobuhvatno (big-bang) (kombinatorna eksplozija)

Sklopovski primjer: integrirani sklop s 1e6 tranzistora i 30 pinova
(nužno oblikovanje **ispitne funkcionalnosti** u IC industriji!)

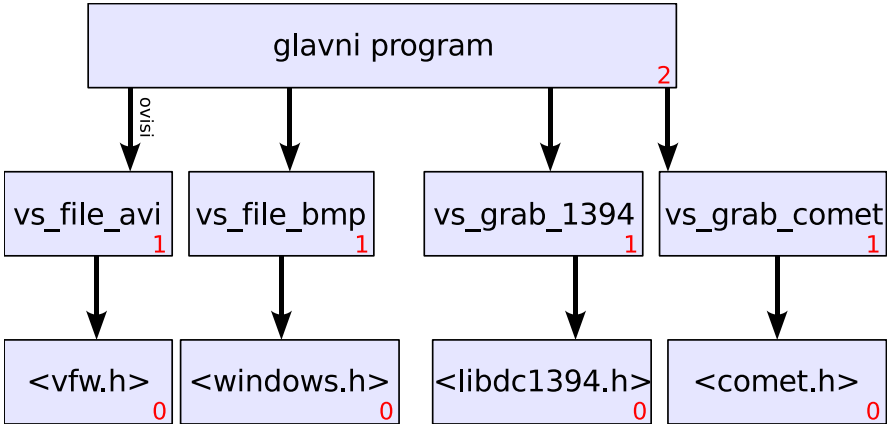
Razmatranje **testiranja** nužno i kod **programskog** oblikovanja;
ali prednost: moguće testiranje **u izolaciji**

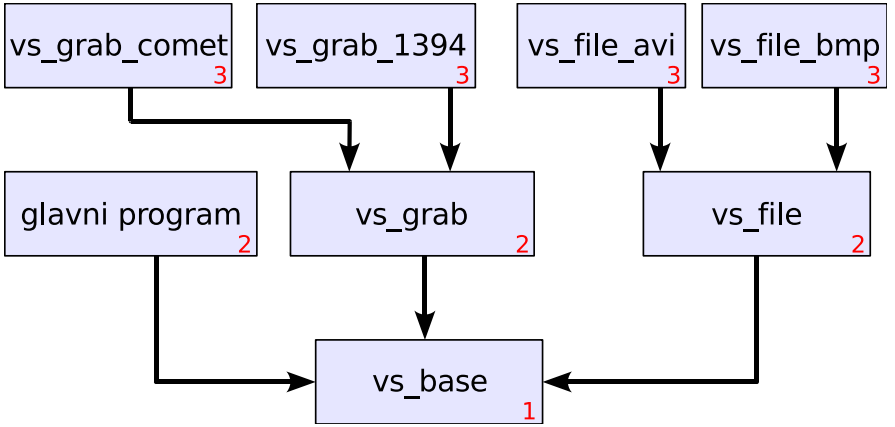
Važno: rano automatsko testiranje (regresijsko, inkrementalno)

Fizička načela: razine

U kontekstu testiranja i fizičke arhitekture, ključna relacija je ovisnost

Ovisnost inducira **aciklički graf** $\Rightarrow$ komponentama dodjeljujemo razine





Testiranje (kao i razumijevanje te ponovna upotreba) zahtijeva razmatranje komponenata na nižim razinama

Povoljno: **plitka** i **nepovezana** hijerarhija

Nepovoljno: duboka ili monolitna hijerarhija, te ciklusi

Fizička načela: ciklusi

Pored ispitivanja u izolaciji, ciklusi ometaju i višestruko korištenje te razumijevanje modula

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

Iako rješenje uvijek postoji, općenitog recepta nema:
različita rješenja prikladna u različitim situacijama

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

U temeljima oblikovnih obrazaca su ponovo načela oblikovanja (DIP)

Fizička načela: ciklusi, primjer

```
struct View;
struct Document{
  void setState(){
    // ...
    pview_->update();
  }
  void getState();
private:
  View *pview_;
  // ili:
  // std::list<View> views_;
};

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

//  Cyclic dependency: Doc  <-> View
//  may be bad!
```

```
struct ViewBase{
  virtual void update()=0;
};

struct Document{
  void setState(){
    // ...
    pview_->update();
  }
  void getState();
  void attach(ViewBase* pv);
private:
  ViewBase *pview_;
};

struct View: public ViewBase{
  View(Document *doc): pdoc_(doc){
    pdoc_->attach(this);
  }
  void update();
private:
  Document *pdoc_;
}
```

Gordijski čvor smo raspetljali **inverzijom ovisnosti**

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** razrede u pakete (načela **koherentnosti**) ?
2. kako valja urediti **odnose** među paketima (načela **stabilnosti**) ?
3. što oblikovati prije:
   - pakete (s vrha prema dnu)?
   - razrede (odozdo prema gore)?

Oblikovanje paketa: koherencija

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

1. zajedničko ponovno korištenje:
   - izdavanje paketa iziskuje napor s obje strane (autor, klijent)
   - razredi koji se ne koriste zajedno **ne pripadaju** istom paketu
2. ekvivalencija korištenja i izdavanja:
   - promjene elemenata paketa moraju biti međusobno **usklađene**
   - $\Rightarrow$ paket grupira razrede koji se **zajedno** izdaju i **održavaju**
   - $\Rightarrow$ interakcija između oblikovnih i poslovnih kriterija
3. jedinstvena odgovornost (osjetljivost na promjene):
   - razredi osjetljivi 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 razredi paketa modeliraju jedinstvenu os promjene složenog sustava (tj, da imaju jedinstvenu odgovornost)

Moramo razmatrati i međusobno usprotivljene zahtjeve:

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

Dinamička ravnoteža među navedenim silama i potrebama aplikacije

- česte **promjene** raspodjele komponenti po paketima tijekom napredovanja projekta
- prioritet s lakog razvijanja postupno prelazi na lako korištenje

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 **stabilnijim** paketima

3. načelo **stabilnih apstrakcija**:
apstrahiranje paketa s puno ulaznih ovisnosti

Oblikovanje paketa: stabilna ovisnost

Stabilnost $\equiv$ otpornost na promjene:

- pozitivne i negativne konotacije: **pouzdanost** vs. inertnost!
- inertnost u mnogome određena međuovisnostima (odgovornost)!

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

- O_U ... broj vanjskih komponenti koje ovise o paketu
- O_I ... broj ovisnosti o vanjskim paketima

načelo: ovisnost treba ići prema stabiln(ij)im paketima

S – metrika treba rasti uzduž puteva u grafu ovisnosti!

stabilnost implicira odgovornost, a ona zahtijeva **pouzdanost**:

stabilni paketi moraju **ili** zadovoljavati OCP **ili** biti vrlo jednostavni

Oblikovanje paketa: stabilne apstrakcije

Povoljno: stabilni paketi otvoreni za **proširenja**, zatvoreni za promjene

Rješenje: **apstrahirati** stabilne pakete (odnosno njihova sučelja)!

Metrika 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!

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

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

- $S \gg A$ ... zona patnje (može biti OK, ako je paket zreo)
- $A \gg S$ ... zona beskorisnosti (simptom pretjeranog oblikovanja)
- $D' = |A - S| > x_{\text{th}} \Rightarrow$ paket je kandidat za prekrajanje

Oblikovanje paketa: 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 suptilno: jako je važan osjećaj za **mjeru**

Nezaobilazan element uspjeha je **iskustvo**!

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?

Alati, prema redosljedu korištenja:

- zdrav razum (KISS, YAGNI, DRY)
- metodologije (strukturiranje, enkapsulacija, polimorfizam, idiomi)
- **načela**
- oblikovni obrasci (design patterns)
- arhitektonski obrasci (architectural patterns)