

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

- **Motivacija:** simptomi urušavanja programa
- **Primjer:** problem i rješenje
- **Tehnike:** pregled dijagrama i programskih koncepata
- Načela **logičkog** oblikovanja
- Načela **fizičkog** oblikovanja
- **Zaključak**

MOTIVACIJA: PROBLEMI

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

Zašto je teško program napisati kako treba “isprve”?

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

Programiranje je teško: kako prebroditi teškoće?

- konstruktivnije tražiti rješenje nego tražiti krivicu!
- rješenje: organizaciju postupno usklađivati sa saznanjima o domeni!

MOTIVACIJA: SIMPTOMI

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

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

Specifični primjeri patologije (anti-obrasci):

- **pretjerana međuovisnost**: “spaghetti code”
- **pretjerana složenost**: “Swiss-Army Knife”
- **ponavljanje** umjesto ponovnog korištenja: “reinvent the wheel”

MOTIVACIJA: 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
 - protiv krutosti se obično borimo **enkapsulacijom** i **apstrakcijom**
- sitna “poludnevna” intervencija pretvara se u višednevni maraton istitravanja promjene kroz sustav
- **strah** od ispravljanja problema koji nisu kritični
- pozitivna povratna veza:
krutost → izbjegavanje promjene → još veća krutost

MOTIVACIJA: KRHKOST

Kada je programski sustav **krhak**?

- tendencija programa da “puca” nakon promjena:
 - uzrok: **implicitna** međuovisnost uslijed **ponavljanja**
 - **jedna** konceptualna izmjena mora se unijeti na **više** mjesta
 - propusti rezultiraju suptilnim greškama koje je teško pronaći
- uzroci ponavljanja:
 - “neizbježno” (heterogenost, komentari, dokumentacija, jezik)
 - nepažljivost ili nestrpljivost (jačaju s **viskoznošću**)
 - neusklađenost razvojnog tima (komunikacija!)
- krhkost jača krutost (Y2K fijasko, 3e11 USD)
- ponovo, pozitivna povratna veza evoluciju čini teškom
krhkost, krutost → izbjegavanje promjene → veća krhkost, krutost

MOTIVACIJA: KRHKOST, NPR

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

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

Komponenta Line je krhka, jer tko god promijeni x1 itd. mora se sjetiti promijeniti i len.

Moguća (vjerojatna) posljedica: cjelodnevni bubolov.

Krhke komponente olakšavaju unošenje suptilnih bugova koje statička analiza ne može pronaći.

MOTIVACIJA: NEPOKRETNOST

Nepokretnost programskog sustava:

- otežano višestruko korištenje prethodno razvijenih modula
- čest uzrok: pretjerana međuovisnost zbog neadekvatnih sučelja
- nesuđeni novi korisnik otkriva da postojeći modul ima previše “prtljage” koju nije lako eliminirati
- moduli se pišu iznova, umjesto evolucije kroz ponovno korištenje
- nepokretnost potiče **ponavljanje**, odnosno krhkost i krutost

MOTIVACIJA: NEPOKRETNOST, NPR

```
//Object.hpp
class Object{
    ...
}

//Vector.hpp
class Vector{
    ...
    Vector(int i);
    Object* operator [] (int i);
    ...
};

//Triangle.hpp
class Triangle: public Object{
    // ...
};

// client code:
Vector v1(3);
v1[0]=new Triangle; // OK!
v1[0]=new std::string("burek");
// the vector class does not work for classes
// which do not derive from Object!
```

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

Razred Image koji koristi libAcmeTiff (vidi uvodno predavanje) također je nepokretan.

MOTIVACIJA: VISKOZNOST

Programski sustav je **viskoz** 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
 - potreban znatan napor za održavanje integriteta programa
- **viskoznost razvojnog procesa:** spora, neefikasna razvojna okolina
 - komplicirani sustav za verziranje implicira rjeđe sinkronizacije kôda te kasnije otkrivanje problema u vezi s integracijom
 - sporo prevođenje pospješuje unošenje “**zakrpa**” umjesto primjerenog održavanja organizacije

MOTIVACIJA: VISKOZNOST ORGANIZACIJE, NPR

Tipičan kod 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;
    }
}
```

MOTIVACIJA: VISKOZNOST ORGANIZACIJE, NPR

Dojava grešaka preko povratne vrijednosti pospješuje viskoznost:

- od silnih provjeravanja se ne vidi što program radi
- iznimke su puno bolja opcija!

S druge strane, vraćanje golih pokazivača pospješuje krhkost:

- nakon `GetInputProps` moramo se sjetiti pozvati `Release`
- `new/delete`, `malloc/free` considered harmful in client code

MOTIVACIJA: UZROCI

Zajednički nazivnik **patologije**:

- zbog neadekvatnog oblikovanja ili izmijenjenih zahtjeva dolazi do neplaniranih izmjena
- izmjene uzrokuju degradiranje organizacije:
 - neželjena **međuviznost** komponenata ili funkcionalnosti
 - **ponavljanje** u implementaciji ili organizaciji
- degradacija organizacije uzrokuje otežanu evoluciju programa
- loša organizacija → slaba evolucija → loša organizacija → ...
- rješenje: uspostavljanje integriteta organizacije programa!

MOTIVACIJA: ŽIVO BLATO



PRIMJER

Pretpostavimo da pišemo korisnički program koji obavlja funkciju računalnog vida

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 img;

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

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

        // ... 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 img;  
        switch(myVS){  
            case VSFile:  
                file.getFrame(img);  
                break;  
            case VSGrab:  
                grabber.getFrame(img);  
                break;  
        }  
        myAlgorithm.process(img);  
        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 img;
        switch(myVS){
            ...
        }
        myAlgorithm.process(img);
        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 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!

(osnovni zadatak oblikovanja je zauzdavanje kombinatorne eksplozije)

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!

Što ćemo sad?

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

PRIMJER: V5

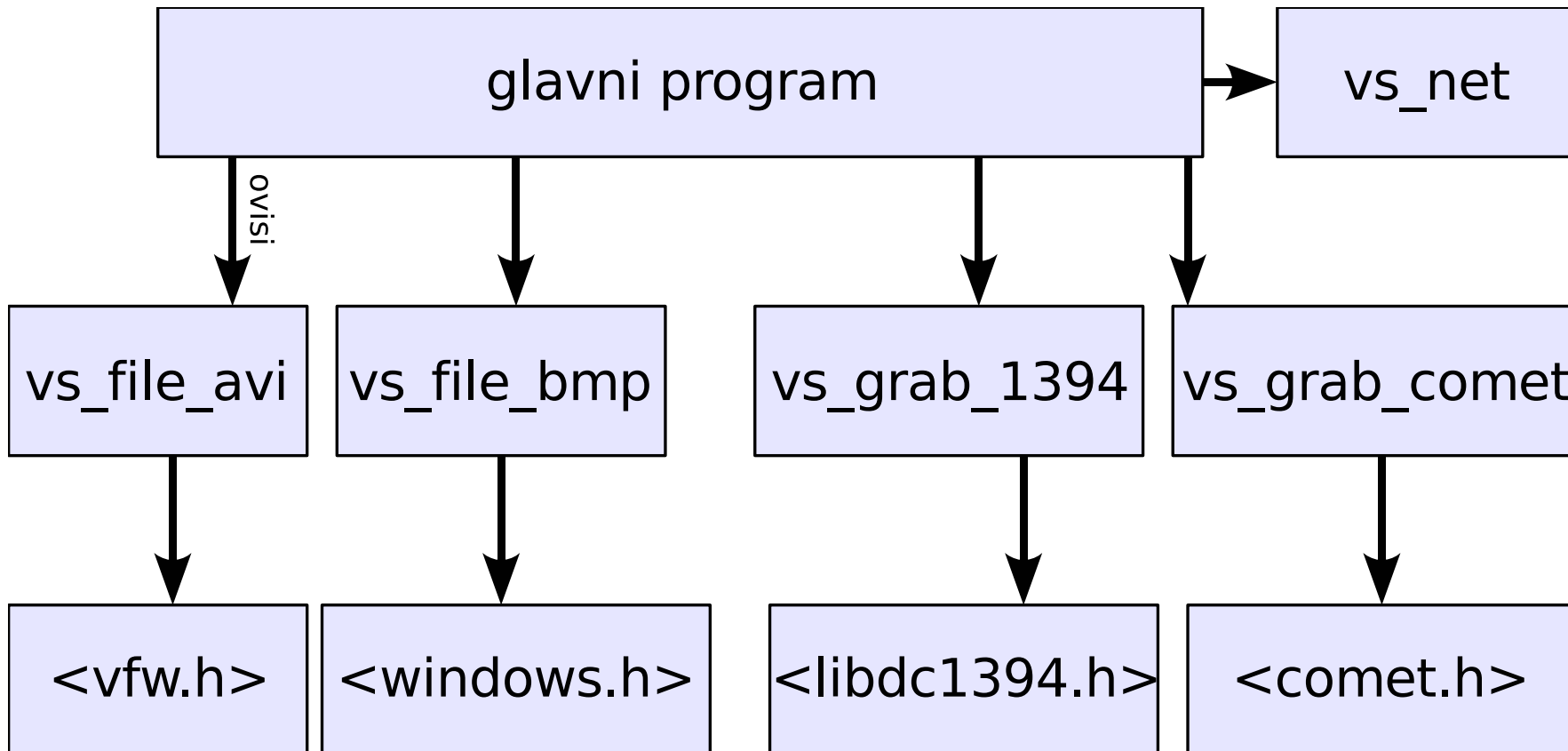
Nova petlja koristi (dinamički) **polimorfizam**, i u potpunosti je neovisna o konkretnom pribavljanju, obradi i spremanju slika:

```
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(algorithm.nDst());
    while(!vs.eof()){
        img_wrap 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 organizacija: **puno** komponenata o kojima mnogo toga ovisi



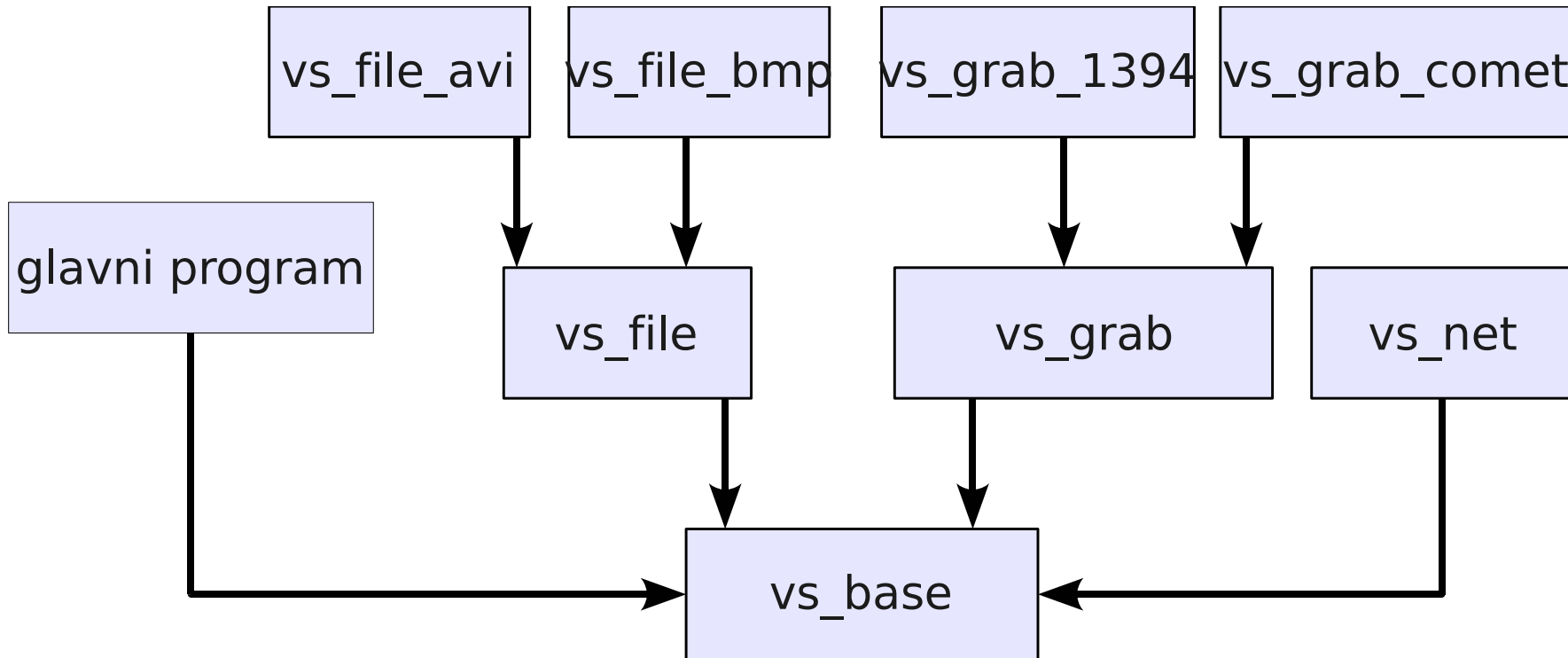
Održavanje “tehnoloških” komponenti **utječe** na glavni program

Npr: glavna komponenta ne može se **prevesti** bez <comet.h>!

(vendor lock-in anti pattern)

PRIMJER: DIJAGRAM V5

Krajnja organizacija: puno više **neovisnih** komponenata



Širimo funkcionalnost **bez prevođenja** glavne komponente

Ovisnost usmjerena od složenog prema apstraktnom

Smanjen pritisak ovisnosti na stožer aplikacije

TEHNIKE

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

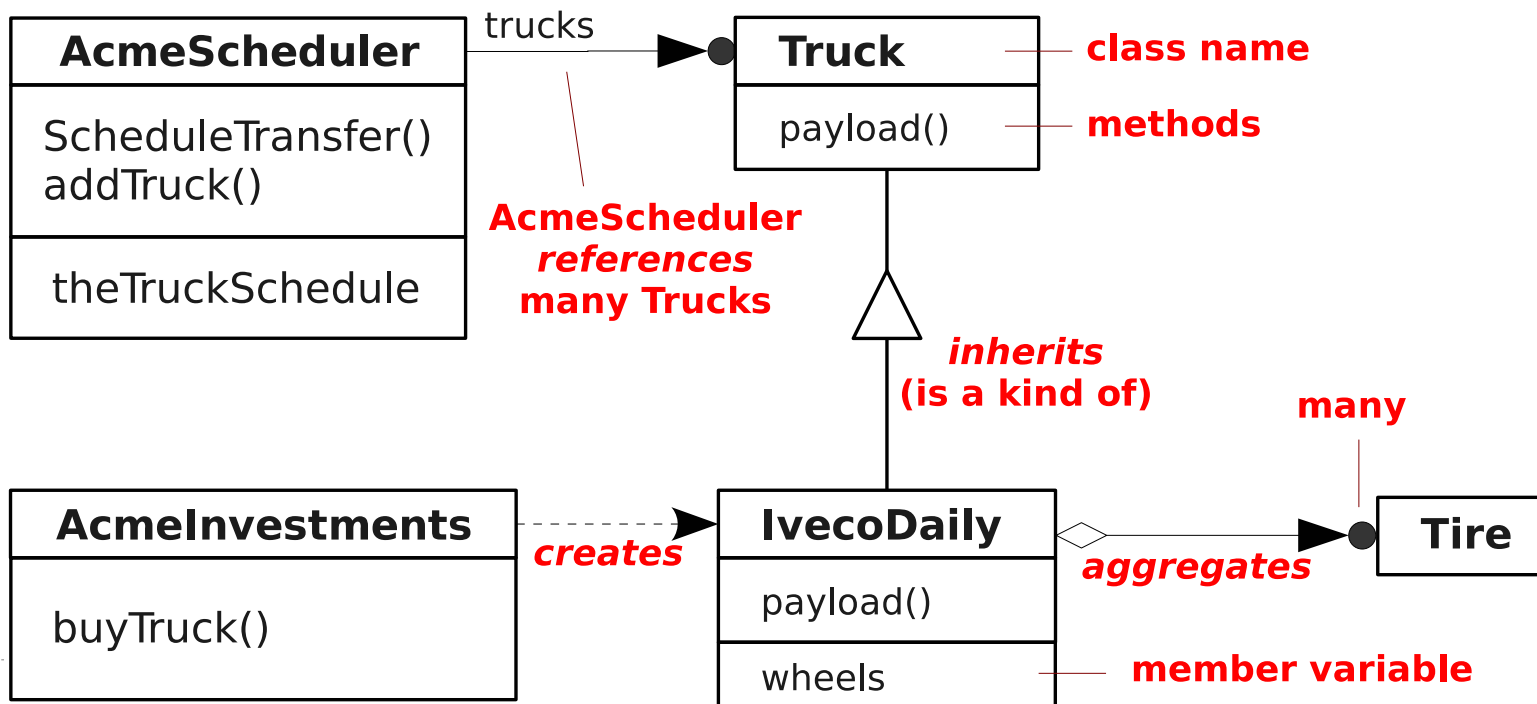
Malo terminologije:

- **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!**
- dobra organizacija poštuje i logička i fizička načela

TEHNIKE: GoF OMT

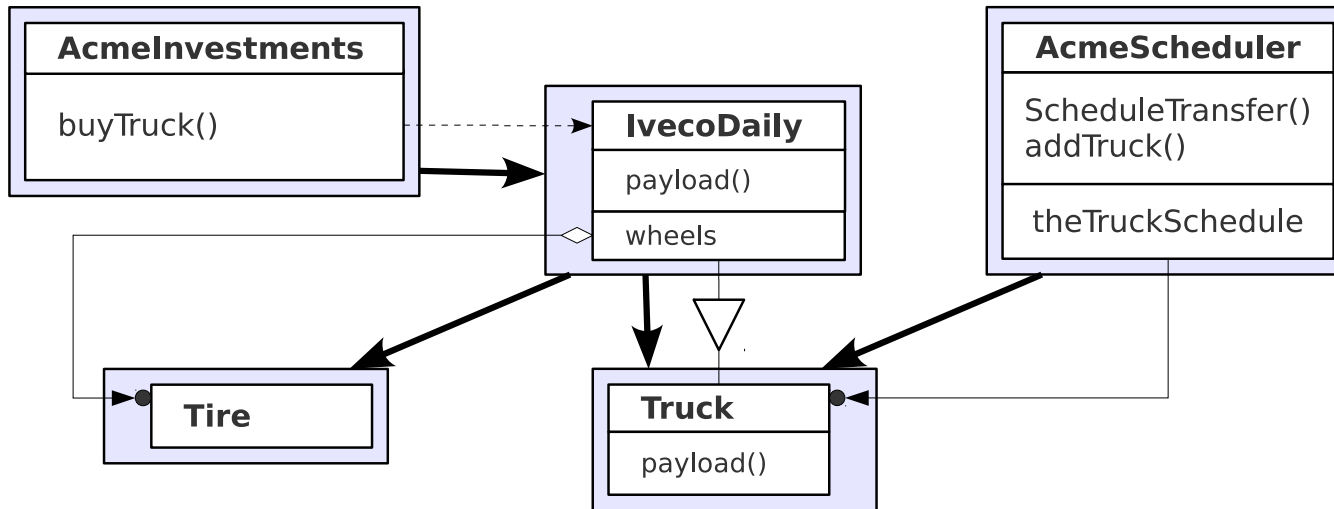
OMT (Object Modelling Technique, 1991): jezik (!) za modeliranje programske podrške (jednostavniji prethodnik UML-a)

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

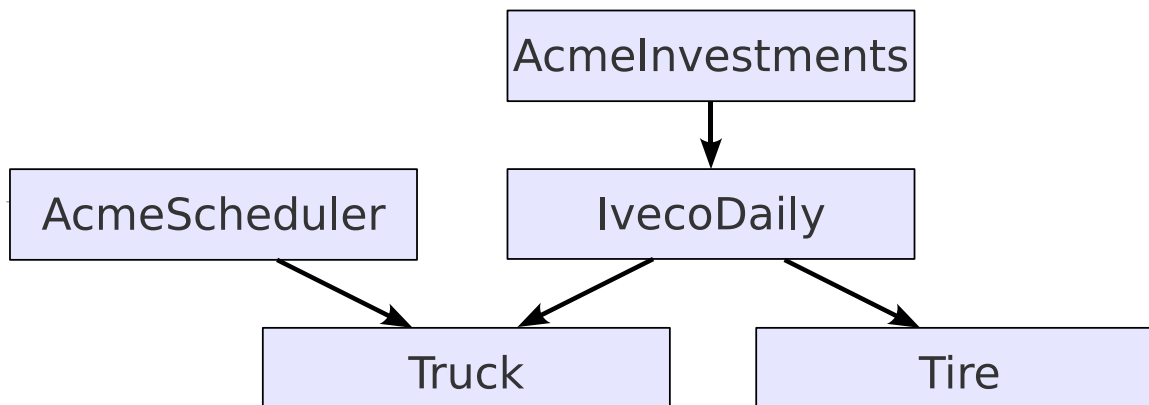


TEHNIKE: LAKOS96

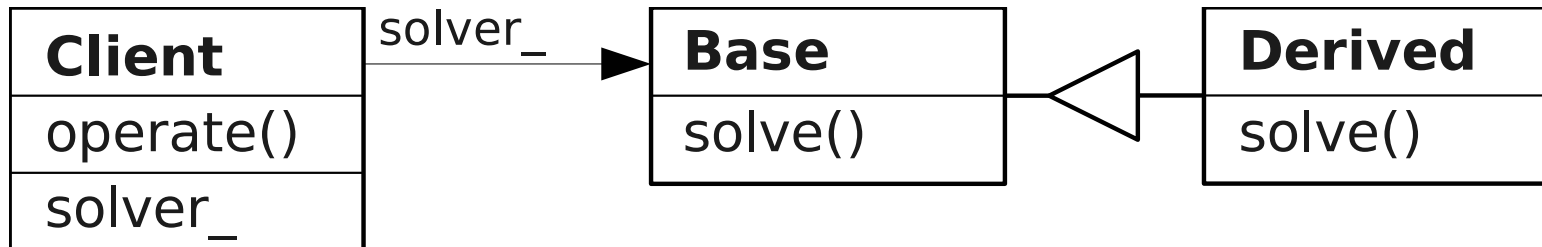
OMT ne može prikazati odnose u fizičkom oblikovanju pa uvodimo hibridnu notaciju iz knjige [Lakos96]:



Hibridnom notacijom možemo opisati i **fizičke** odnose: **ovisnost** komponenata pri prevođenju odnosno testiranju



TEHNIKE: DINAMIČKI POLIMORFIZAM U C++-U



```
//==== 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_;
};
```

```
//==== base.hpp
// don't forget include guards!
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;}
};
```

TEHNIKE: C++ VS C

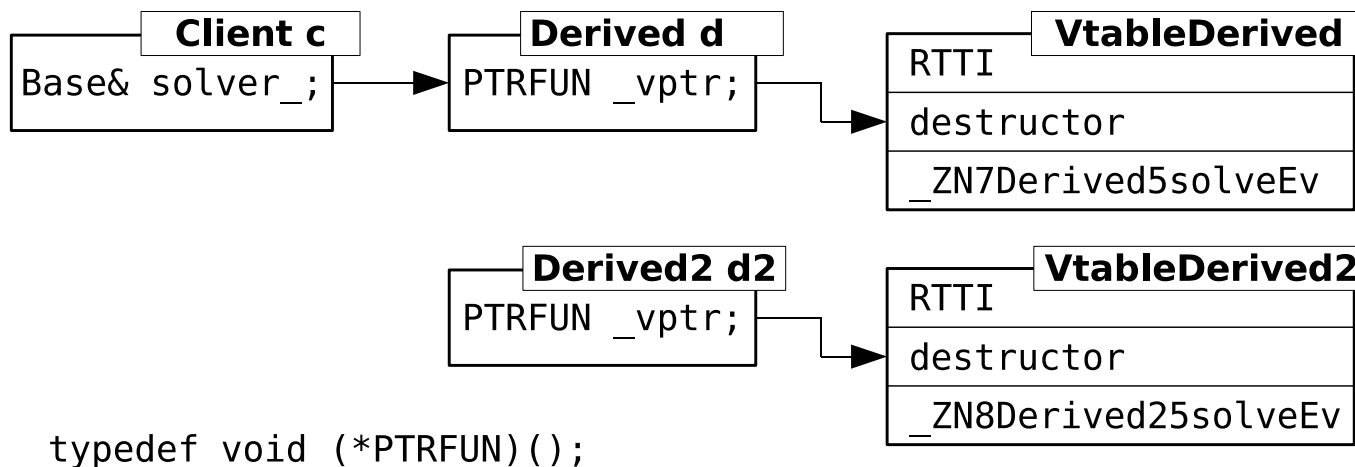
Demistificirani objektni model C++-a [Lippman96]:

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

□ `_ZN6Client7operateEv(&c);`

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

□ `(solver_.vptr[1])(&solver_);`



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:
    def solve(self):
        return -1

class Derived(Base):
    def solve(self):
        return 42

class NonDerived:
    def solve(self):
        return 0

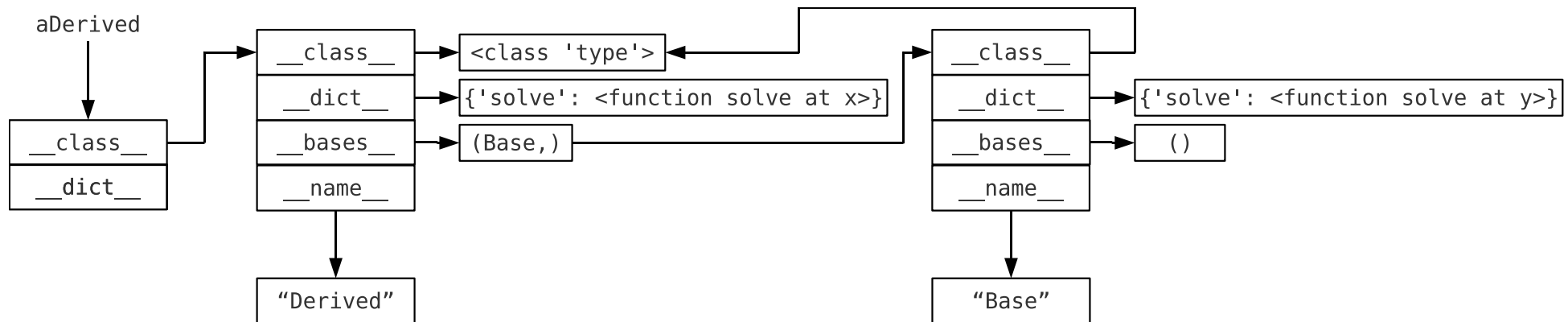
class Client:
    def __init__(self, solver):
        self.solver_ = solver
    def operate(self):
        print(self.solver_.solve())
# this works as expected:
aDerived = Derived()
c = Client(aDerived);
c.operate()
# this also works (duck typing!):
c2 = Client(NonDerived()); c2.operate()
```

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

- asocijativni pristup mapi funkcija razreda `Derived`
- u slučaju neuspjeha, asocijativni pristup mapi funkcija roditeljskog razreda (`Base`)
- poziv metode može rezultirati prozivanjem `n` asocijativnih mapa!

TEHNIKE: PYTHON (2)

Objektni model Pythona: prikaz objekta aDerived razreda Derived:



Razlike u odnosu na C++ i Javu:

- puno fleksibilnija izvedba dinamičkog polimorfizma (duck typing)
- mogućnost dodavanja metoda objektu tijekom izvođenja (!!)
- znatno slabija performansa, ali:
 - brzina poziva se poboljšava cacheiranjem
 - Python JIT?

TEHNIKE: 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())`), striktno tipiziranje, jednaka učinkovitost, veća izražajnost

TEHNIKE: GENERICI (2)

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

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

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

TEHNIKE: STL

Ortogonalnost (nema međuovisnosti) 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";
}
```

TEHNIKE: GENERICI VS. VIRTUALNE FUNKCIJE

Odnos generičkog i virtualnog poziva:

- generički izvršni kôd obično ima povoljniju složenost:
 - manja vremenska složenost zbog razrješavanja polimorfnog poziva tijekom prevođenja
(moguće je čak izravno umetanje odgovarajuće izvedbe)
 - manja podatkovna prostorna složenost
(nema potrebe za pokazivačima na tablice funkcija)
 - veća prostorna složenost generiranog kôda
- nakon prevođenja fleksibilnost generičkog koda **nestaje**
- predložci prikladni za manje, često korištene programske jedinice

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 ("Što sve treba implementirati?") [shalloway05]

OOP razmatra **dinamiku** (evoluciju) sustava (npr, "Kako postići da kôd kojeg pišemo danas ispravno radi s kôdom kojeg ćemo pisati dogodine?")

Pri OO oblikovanju često se pitamo "Što će se vjerojatno mijenjati u budućnosti?"; na temelju te procjene pokušavamo se zaštititi od **promjena**

Točnost prognoze određuje hoće li će se obećanja OOP ispuniti ili ne (programiranje intrinzično teško: no silver bullet!)

TEHNIKE: DBC

Oblikovanje temeljeno na ugovoru: sintagma B. Meyera, kreatora Eiffela

- komponente surađuju ispunjavanjem obaveza definiranih **ugovorom**
- ugovor: eksplicitan formalni opis zadatka komponente
- ambicija: laka ili čak automatska detekcija kršenja ugovora

Tri osnovna elementa ugovora komponente:

- **preuvjeti** (preconditions) za primjenljivost komponente
- **invarijante**, ili interni pokazatelji integriteta komponente
- **postuvjeti** (postconditions), ili garancija ispravnosti rezultata

U domeni oblikovanja prvenstveno nas interesiraju preuvjeti i postuvjeti

TEHNIKE: DBC, NPR

Gotovo svi jezici omogućavaju provjeravanje elemenata ugovora

Ključni konstrukt je najčešće `assert`
(alternativno, može se baciti iznimka, **throw**)

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
    assert(val>=0);

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

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

    return result;
}
```

LOGIČKA NAČELA

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

- načelo **nadogradnje bez promjene**
dodavanje funkcionalnosti bez utjecaja na klijente
- 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**

(načelo zatvorenosti, open-closed principle)

- cilj: omogućiti proširenje funkcionalnosti komponente **bez mijenjanja** njene implementacije
(**fleksibilnost**: nadogradnja ne utječe na klijente)
- dobivamo komponente koje su **otvorene** za nadogradnju, ali **zatvorene** za promjene (open-closed principle)
- **važna ideja**: stari kôd “radi” s novim kôdom!
- vezano uz ideje **skrivanja informacije** [Parnas72] i **apstrakcije podataka** [Liskov74]
- motivacija za mnoge moderne (i manje moderne) koncepte (OOP, generici, aspekti, introspekcija, refleksija, kontekstne funkcije...)

LOGIČKA NAČELA: NBP I NASLJEĐIVANJE

Primjenom nasljeđivanja, nadogradnju bez promjene možemo ostvariti:

1. nasljeđivanjem implementacije [meyer88]

- novi razredi pozivaju temeljnu implementaciju nasljeđivanjem starog razreda (nema polimorfnih poziva!)
- postojeći klijenti ne mogu doći do nove funkcionalnosti
- ideja nas ne uzbuđuje pretjerano: **novi kôd poziva stari kôd**

2. nasljeđivanjem sučelja, naglasak na polimorfizmu [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: NBP I NASLJEĐIVANJE (2)

NBP nasljeđivanjem implementacije:

```
// OCP by inheriting implementation
class Old{
public:
    void method();
};
// 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(){
        // do something
        member.method();
        // do something else
    }
};
```

NBP nasljeđivanjem sučelja:

```
// OCP by inheriting interface
//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, changed in 2008
int main(){
    Old o; client(&o);
    //no need to change (or even
    // recompile) client.cpp
    //old code calls new code!
    New n; client(&n);
}
```

LOGIČKA NAČELA: NBP I PROCEDURALNI STIL?

Proceduralni stil uzrokuje **kruti** i **krhki** kôd

(jednu konceptualnu izmjenu potrebno unijeti na **više** mjesta)

```
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*);

void drawShapes(Shape** list, int n){
    for (int i=0; i<n; ++i){
        struct Shape* s = list[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 ipak i **kruto** i **krhko**:

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

LOGIČKA NAČELA: NBP I OOP

OOP omogućava rješavanje problema s prethodne stranice

- ključni mehanizam: polimorfni poziv novog kôda preko starog sučelja
- C,C++,Java: implementacija pokazivačima iz virtualne tablice

Enkapsulacija i apstrakcija poboljšavaju dinamička svojstva programa:

```
class Shape{
public:
    virtual void draw()=0;
};
class Circle :public Shape
{
    double radius_;
    Point center_;
public:
    virtual void draw();
};
class Polyline :public Shape
{
    std::vector<Point> points_;
public:
    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: NBP I GP

NBP se može postići i **generičkim** programiranjem (*statičkim* polimorfizmom)

- **statički** polimorfizam u C++-u temeljen na **predlošcima**
- nema ograničenja na razred objekta nad kojim se primjenjuje polimorfni poziv, posebno pogodno za **biblioteke**
- čarolija funkcionira samo tijekom prevođenja (nadograđenu komponentu potrebno 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: NBP I GP, PRIMJER

```
template<typename T> class Array {
public:
    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)]; }
    //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_;
}; // example: Array<int> X(20);
```

Array je NBP jer radi sa svim tipovima koji omogućavaju:

- podrazumijevanu konstrukciju, preslikavanje, pridjeljivanje te destrukciju

U odnosu na **polje** u C-u:

- jednaka učinkovitost optimiranog kôda
- provjera pristupa u neoptimiranom kôdu
- mogućnost transparentnog rasta (`std::vector::push_back()`)

LOGIČKA NAČELA: NBP I GP, PRIMJER STD::MAP

Program za određivanje histograma riječi u ulaznom toku:

```
#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;
    }
}
```

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

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

LOGIČKA NAČELA: NBP U PRAKSI

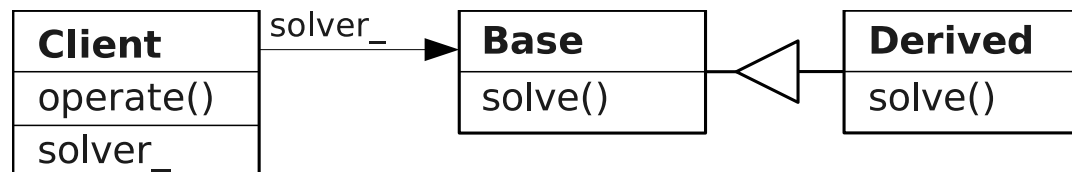
Koncepti koji pospješuju nadograđivanje bez promjene:

- **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 NBP)
- **apstraktni razredi** (čiste apstrakcije):
 - nemaju podatkovnih elemenata \Rightarrow enkapsulirani
 - imaju virtualne funkcije
- **generički programi:** injekcija novog kôda u stari pri prevođenju

LOGIČKA NAČELA: LNS

Načelo **nadomjivosti** osnovnih razreda

- AKA Liskovino načelo supstitucije [Liskov93]
(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**
- na slici dolje, svi razredi koji nasljeđuju Base moraju moći raditi u kombinaciji s Clientom



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
 - primjer na sljedećoj stranici
- 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 razredom koji krši LNS
- **patologija** je kršenje NBP-a: klijent se mora mijenjati kad god se u program unese izvedeni razred prijestupnik

LOGIČKA NAČELA: LNS – GEOMETRIJSKI PRIMJER

```
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);
}
```

Circle ne zadovoljava postavjet roditeljskog razreda: krši se LNS, NBP

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

client prestaje biti NBP čim naslijedimo Ellipse kršeći LNS.

LOGIČKA NAČELA: LNS – PTIČJI PRIMJER

```
class Bird{
public:
    Bird(){}
    virtual ~Bird(){}
public:
    double 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()>0.0);
}
```

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

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

Penguin ne zadovoljava postavljene roditeljske klase, krši se LNS, NBP

LOGIČKA NAČELA: LNS – PTIČJI PRIMJER (2)

```
class Bird{
public:
    Bird(){}
    virtual ~Bird(){}
public:
    double 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()>0.0);
}
```

```
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 korištenje iznimaka, u nadi da će integritet ponovo uspostaviti komponenta na višoj razini apstrakcije

LOGIČKA NAČELA: LNS – POMORSKI PRIMJER

```
class Boat{
    // ...
public:
    virtual void leavePort();
    virtual void navigate(Point x);

    virtual bool contactBase();
    //PRECONDITION: none
public:
    bool hasOrders(){
        return o_.valid();
    }
private:
    Orders o_;
};

void doSecretMission(Boat& b){
    b.leavePort();
    b.navigate(goal);
    while (!b.hasOrders){
        b.contactBase();
    }
    // ...
}
```

```
class Submarine:
    public Boat
{
    // ...
public:
    virtual void leavePort(){
        Boat::leavePort(x);
        submerge();
    }

    virtual bool contactBase();
    //PRECONDITION: depth()==0
    //(can't communicate while submerged)
public:
    virtual void submerge();
    virtual double depth();
    // ...
}

int main(){
    Submarine horrible;
    doSecretMission(horrible);
    //...
}
```

Submarine pooštrava preduvjete roditeljske klase: krši se LNS, NBP

LOGIČKA NAČELA: LNS – IMPLIKACIJE

Nasljeđivanje modelira relaciju **je_** vrsta:

- izvedeni razredi **moraju poštivati** ugovore roditelja
- javno nasljeđivanje **rijetko** 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 LNS-a može se popraviti na 3 načina:

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

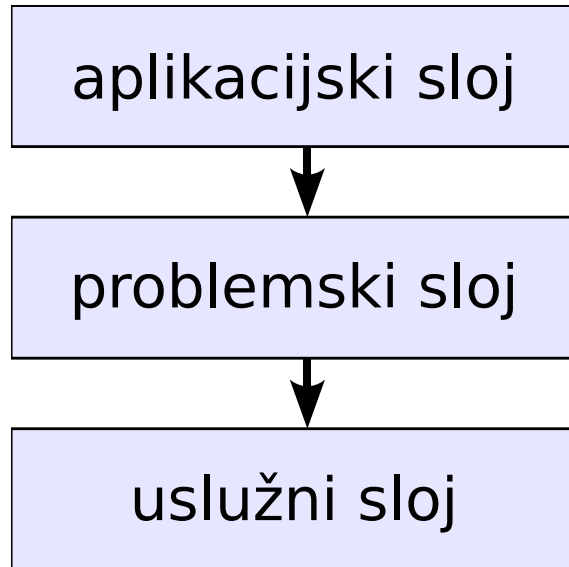
LOGIČKA NAČELA: NIO

Načelo inverzije ovisnosti (dependency inversion principle)

- vidjeli smo da je nadomjestivost (LNS) nužni uvjet nadogradnje bez promjene
- sada: implikacije nadogradivosti i nadomjestivosti na strukturu ovisnosti komponenata vode k načelu **inverzije ovisnosti**
- pokazat će se da se nadogradivost može ostvariti usmjeravanjem ovisnosti prema **apstraktnim** sučeljima

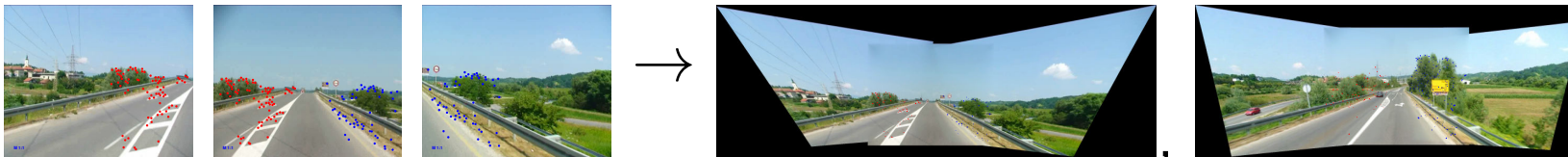
LOGIČKA NAČELA: NIO: MOTIVACIJA

proceduralni stil dovodi do piramidalne strukture međuovisnosti:



Primjer iz stvarnog života:

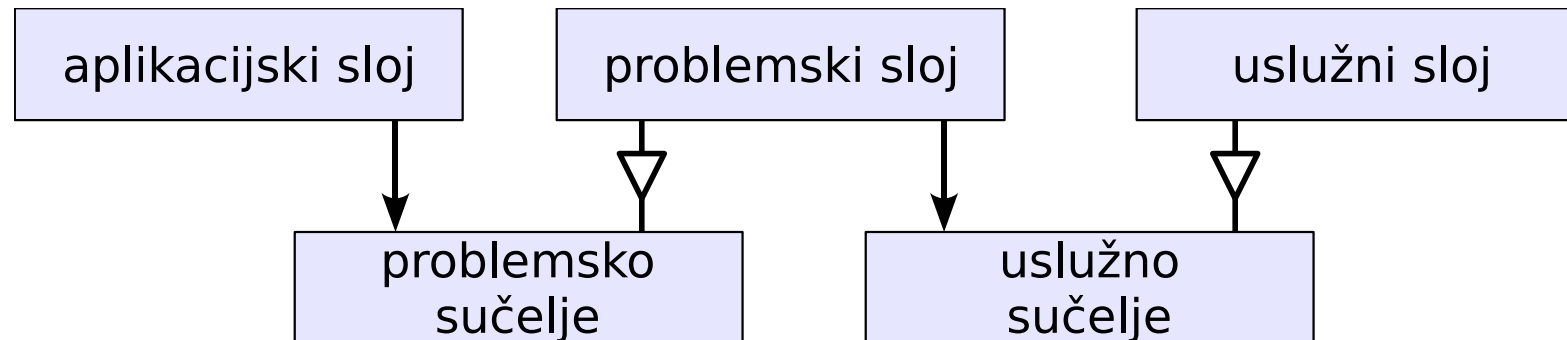
- Aplikacijski sloj:
program za računalni vid
- Problemski sloj:
postupak za spajanje slika
- Uslužni sloj:
pronalaženje korespondencija



- 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 | OOP

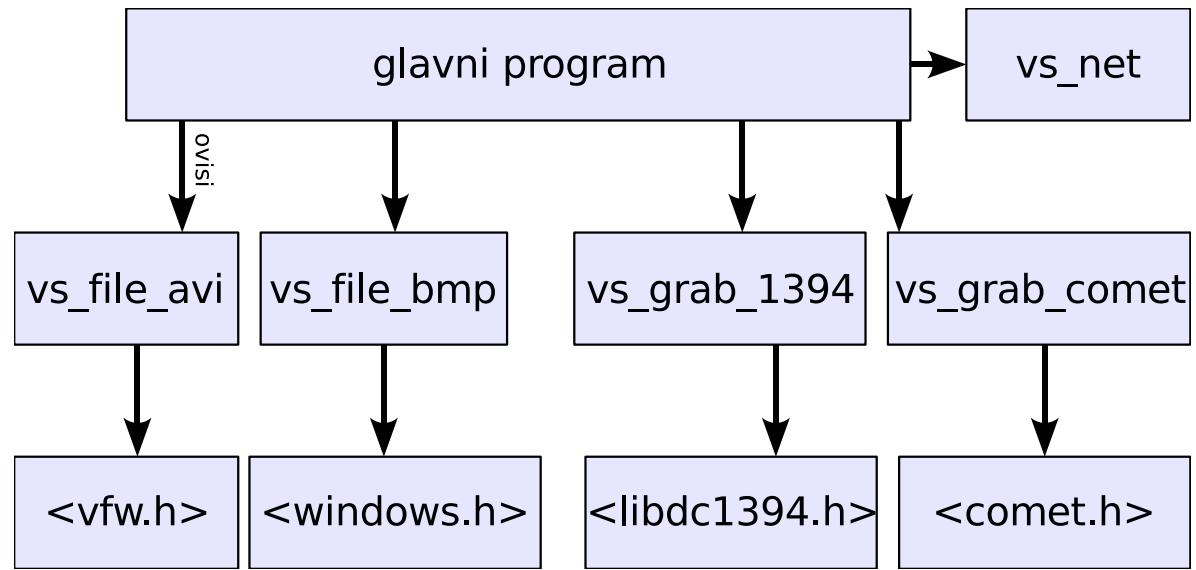
- nedostaci se mogu riješiti promjenom strukture ovisnosti:



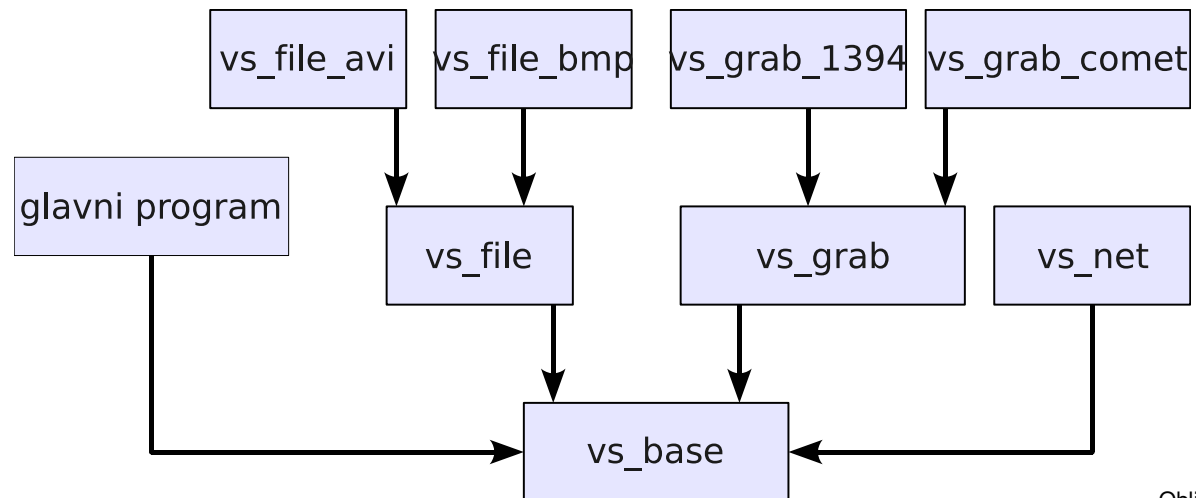
- u novoj organizaciji, ovisnosti idu prema apstrakcijama (koje imaju malo razloga za promjenu jer nemaju implementaciju)
- ne ovisimo više o nepostojanim modulima s detaljnim implementacijama: kažemo da je ta ovisnost **invertirana**
- koristi inverzije ovisnosti:
 - piramidalna struktura preokrenuta: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

LOGIČKA NAČELA: NIO – PRIMJER

PRIJE:



POSLIJE:



LOGIČKA NAČELA: NIO U PRAKSI

- ovisnosti u projektu **u načelu** trebaju ići prema apstrakcijama (**NE** od modula visoke razine prema modulima niske razine)
 - komponenta bez implementacije se rjeđe mijenja
 - apstrakcije omogućavaju širenje bez promjene
- ovisnost o postojećim 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**

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();
        // ...
    }
};
```

- ovisnost uslijed stvaranja izvlači se u zasebnu komponentu!
- maksimiziramo inverziju ovisnosti, odnosno lokaliziramo ovisan kôd

LOGIČKA NAČELA: NIO, INJEKCIJA OVISNOSTI, PRIMJER

```
//==== client1.hpp
class Client1 {
    ConcreteDatabase myDatabase;
public:
    void transaction() {
        myDatabase.getData();
        // ....
    }
};

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

```
//==== testClient2.cpp
#include "client2.hpp"
// ...

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

    // construct test object (`DI`!)
    Client2 client(*pdb);

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

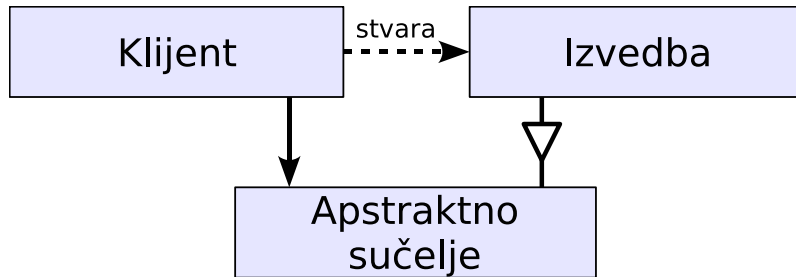
    // test behaviour #2
    // ...

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

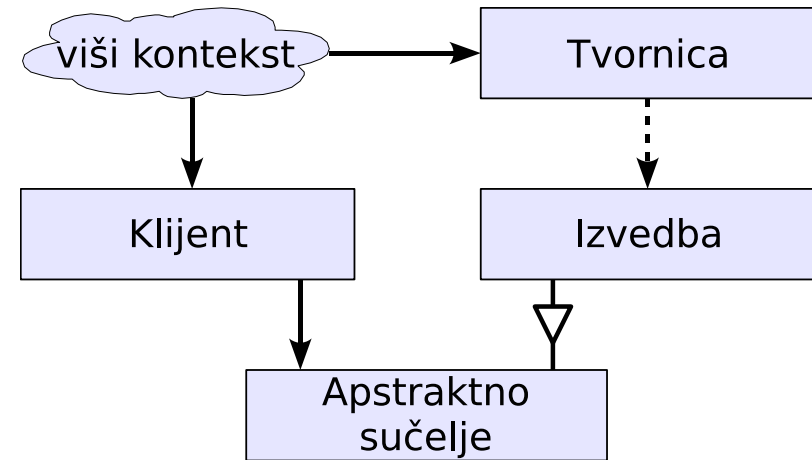
Client2 koristi injekciju ovisnosti, za razliku od Client1

Mogućnost **neovisnog ispitivanja** s obzirom na ConcreteDatabase

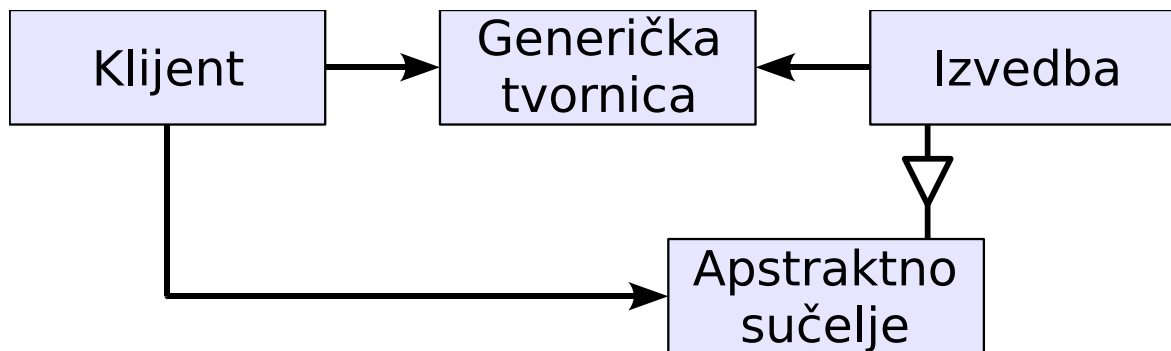
LOGIČKA NAČELA: NIO, INJEKCIJA OVISNOSTI, EFEKTI



1. bez injekcije ovisnosti



2. injekcija ovisnosti + tvornica



3. primjena generičke tvornice

LOGIČKA NAČELA: PLAN PREDAVANJA (PODSJETNIK)

- Uvodno predavanje
 - motivacija i ciljevi programskog oblikovanja, razvojne metodologije, opis predmeta
- Načela oblikovanja (design principles)
 - motivacija: zašto se programi urušavaju?
 - primjer: kako se problemi javljaju, te kako ih se riješiti
 - pregled tehnika: dijagrami razreda i komponenata, statički i dinamički polimorfizam, OO analiza, ugovorno oblikovanje
 - Načela logičkog oblikovanja:
 - ◇ NBP, LNS, NIO
 - ◇ Načelo jedinstvene odgovornosti, načelo izdvajanja sučelja
 - Načela fizičkog oblikovanja

LOGIČKA NAČELA: NJO

Načelo **jedinstvene odgovornosti** (engl. single responsibility principle)

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

Kako jedinstvena odgovornost modula pospješuje 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 (**krhkost**, **nepokretnost**)

NJO: oblikovati **ortogonalan** sustav u kojem raspodjela poslova odgovara intrinzičnoj strukturi problema

(svaka komponenta modelira svoju **os promjene**)

LOGIČKA NAČELA: NJO – PRIMJER

Razmotrimo organizaciju programa za vektorsku grafiku, gdje razred `Rectangle` obuhvaća više od jedne odgovornosti:

```
class Shape{
public:
    virtual ~Shape();
    virtual void rotate(double phi)=0;
    virtual void draw(Window& wnd)=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& wnd)
{
    Drawing::iterator it=d.begin();
    while (it!=d.end()){
        it->draw(wnd);
    }
}
//package Geom
void Geom::intersect(const Shape& s1,
                    const Shape& s2, Shape& sout)
{
    //...
}
```

Loše: paket `Geom` ovisi o paketu `GUI`, bez konceptualnog opravdanja (Geom moramo prevesti kad god se promijeni sučelje metode `draw`)

Bolje 1: premjestiti metodu `draw(Window&)` u razred `GUI::Shape` (razred `GUI::Rectangle` bi izvodio `GUI::Shape` i sadržavao `geom::Rectangle`)

Bolje 2: uvesti `Shape::toSpline` te crtanje izvesti u zasebnoj komponenti

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 – ORTOGONALNOST

Jedinstvena odgovornost usko povezana s ortogonalnošću [Hunt00]:

- ortogonalnost \Rightarrow promjena jednog modula ne utječe na ostale
- jedinstvene odgovornosti \Leftrightarrow ortogonalni sustavi

Ortogonalnost implicira neovisnost među nepovezanim stvarima
(skalarni produkt ortogonalnih vektora je 0)

Ortogonalnost je usko povezana s izbjegavanjem ponavljanja
(ako se ista funkcionalnost nalazi na više mjesta, sustav nije ortogonalan)

Neortogonalni sustavi se teško kontroliraju, i zato ih valja izbjegavati:

```
class Car{
public:
    virtual void brakePedal(double x);
    virtual void clutchPedal(double x);
    virtual void gasPedal(double x);
    virtual void switchGears(int pos);
};
```

```
class OrthogonalCar: private Car{
public:
    enum DriveStyle{DSEco, DSSport, ...};
    virtual void accelerate(double which);
    virtual void setStyle(DriveStyle s);
};
```

LOGIČKA NAČELA: NJO – ZAKLJUČAK

Načelo jedinstvene odgovornosti je u temelju 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, rotiranje oblika i crtanje)
- 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 programiranja
- metode: primjeri korištenja, probni baloni, prototipi, reverzno inženjerstvo

LOGIČKA NAČELA: NIS

Načelo **izdvajanja sučelja** (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)
 - unosimo **suvišne ovisnosti** o nekorištenim elementima sučelja
- načelo sugerira da nekoherentnim konceptima (ako ih baš moramo imati), klijenti trebaju pristupati preko **izdvojenih** sučelja:
nije dobro primoravati klijente na ovisnost o sučelju kojeg ne koriste

LOGIČKA NAČELA: NIS - PREOPSEŽNO SUČELJE

```
//==== door_v1.hpp
class Door{
public:
    virtual void lock() = 0;
    virtual void unlock() = 0;
    virtual bool isLocked() = 0;
};
// new requirement: a timed door!
//==== timer.hpp
class Timer{
public:
    void register(int timeout,
                  TimerClient* client);
};
class TimerClient{
public:
    virtual void timeOut() = 0;
};
//==== door_v2.hpp
#include "timer.hpp"
class Door_V2: public TimerClient
{
public:
    virtual void lock();
    virtual void unlock();
    virtual bool isLocked();
    virtual void timeOut();
};
```

loše: svi klijenti vrata rekompajliraju
kad god se promijeni `TimerClient`

```
// new requirement: multiple events!
// (*all* door clients need to recompile)
//==== timer_v3.hpp
class Timer_v3{
public:
    void register(int timeout,
                  int id, TimerClient* client);
};
class TimerClient_v3{
public:
    virtual void timeOut(int id) = 0;
};
// solution: interface segregation
//==== door_v3.hpp (same as door_v1.hpp)
//==== timedDoor.hpp
#include "door_v3.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);
};
```

puno bolje: samo specijalna vrata ovise o
`Timeru`!

LOGIČKA NAČELA: NIS – SAŽETAK

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

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

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

```
class Shape{
public:
    virtual void rotate(double phi)=0;
};

class Graphic{
public:
    virtual void draw(Window&) =0;
};

class GraphicShape:
    public Shape, public Graphic
{};

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

typedef std::list<Graphic*> Drawing;

void GUI::draw(
    Drawing& d, Window& wnd)
{ //...
}
```

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

LOGIČKA NAČELA: LOGIČKA NAČELA – SAŽETAK

NBP, LNS, NIO: recept za nasljeđivanje

- klijenti mogu izbjeći ovisnost o konkretnim izvedbama (NBP)
- nasljeđivanje treba modelirati relaciju `je_vrsta` (LNS)
- organizacijski je povoljno ovisnost usmjeriti prema apstraktnim komponentama (NIO)

NJO: recept za grupiranje funkcionalnosti po komponentama

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

NIS: recept za višestruko nasljeđivanje

- kako napraviti kompromis s NJO i ostati živ

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)
(you ain't gonna need it - YAGNI)
- do not repeat yourself (DRY) [Hunt99]
(the refactor rule of three)

LOGIČKA NAČELA: DRUGE SISTEMATIZACIJE

Npr, načela iz HFDP [Freeman04]:

Encapsulate what varies. (NBP)

Program to interfaces, not implementations. (NBP)

Favor composition over inheritance. (LNS)

Strive for loosely coupled designs between objects that interact.
(ortogonalnost, NJO)

Classes should be open for extension but closed for modification. (NBP)

Depend on abstractions. Do not depend on concrete classes. (NIO)

Principle of Least Knowledge - talk only to your immediate friends
(NBP)

The Hollywood Principle - don't call us, we'll call you (obrazac MVC)

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 koda: 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

Obično težimo testirati n komponenti **zasebno**, a ne sve moguće interakcije (2^n): **inkrementalno** vs **sveobuhvatno** (big bang) testiranje

Kombinatorna eksplozija ispitnih vektora obično se javlja kad škrto sučelje enkapsulira sofisticiranu funkcionalnost:

```
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 ;
};
```

Sklopovski primjer: kako ispitati čip s $1e6$ tranzistora i 30 pinova?
(nužno oblikovanje **ispitne funkcionalnosti** u IC industriji!)

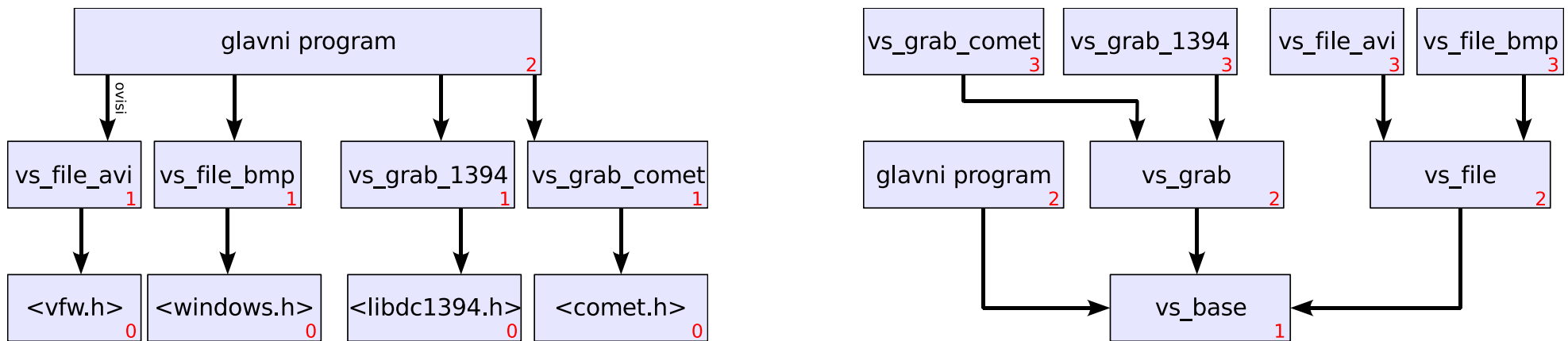
Posebna pogodnost kod programskog oblikovanja: testiranje **u izolaciji**

Važna ideja: rano automatsko testiranje (regresijsko, inkrementalno)

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!



Sada možemo izraziti načelo fizičkog oblikovanja komponenata:

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

FIZIČKA NAČELA: CIKLUSI

Ciklusi ometaju **ispitivanje**, **višestruko korištenje**, i **razumijevanje**

Šteta je tim veća što je ciklus veći (funkcionalnost, broj komponenata)

ekstrem: komponenta na dnu hijerarhije ovisi o vršnoj komponenti

Iako rješenje za dokidanje ciklusa 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!

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,

FIZIČKA NAČELA: CIKLUSI, PRIMJER

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

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

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

```
// viewbase.hpp
class ViewBase{
public:
    virtual void update()=0;
};
// document.hpp (includes viewbase.hpp)
class Document{
public:
    void setState(){
        // ...
        pview_ ->update();
    }
    void getState();
    void attach(ViewBase* pv);
private:
    ViewBase *pview_;
};
// view.hpp (includes document.hpp)
class View: public ViewBase{
    Document *pdoc_;
public:
    View(Document *doc): pdoc_(doc){
        pdoc_ ->attach(this);
    }
    void update();
}
```

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** 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**
 - ⇒ grupiramo komponente koje se **zajedno** izdaju i **održavaju**
 - ⇒ 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 → 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

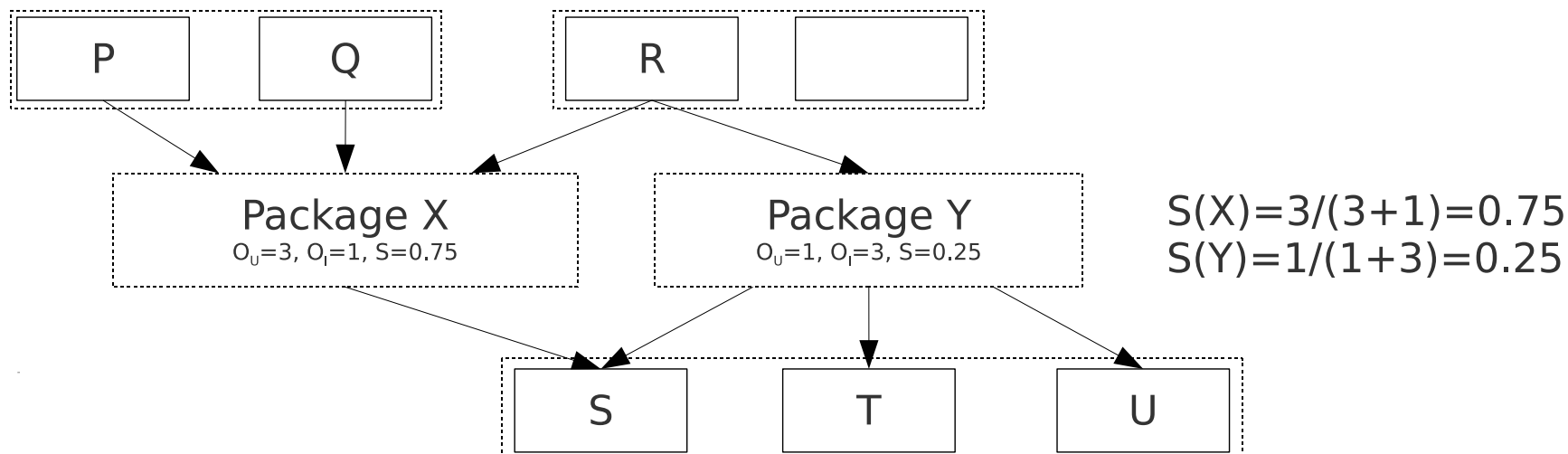
OBLIKOVANJE PAKETA: METRIKA STABILNOSTI

Stabilnost \equiv otpornost na promjene:

- pozitivne i negativne konotacije: **pouzdanost** vs. **inertnost**!
- inertnost je određena **međuvisnostima** (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?

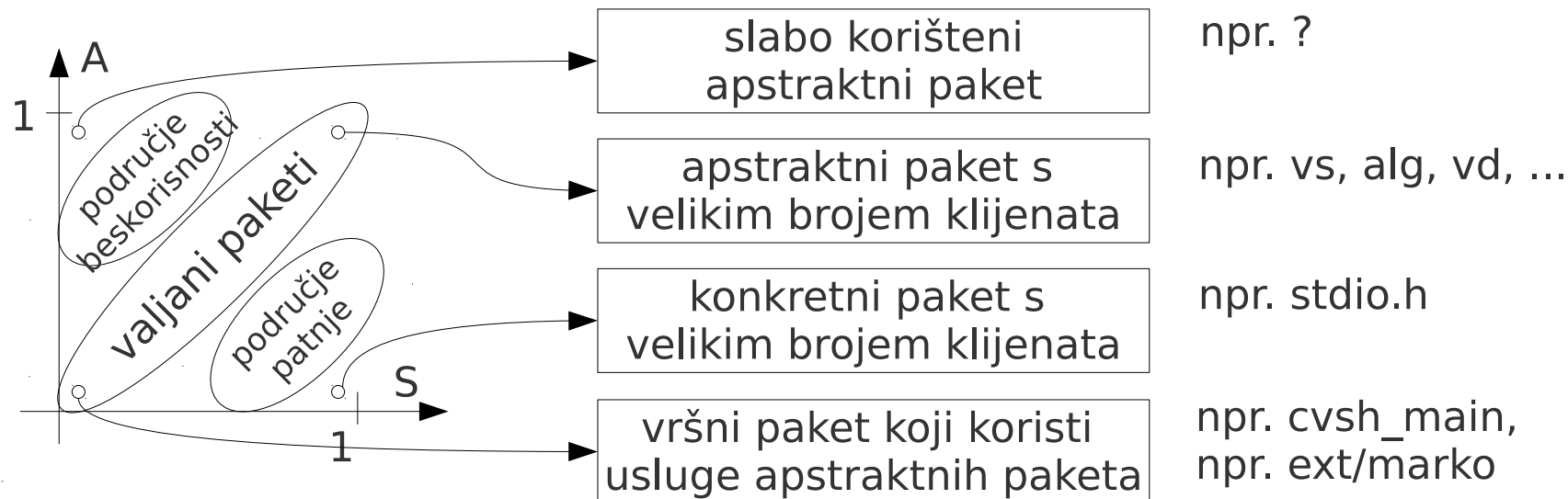
Načelo primjerene apstraktnosti:

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

OBLIKOVANJE PAKETA: PRIMJERENA APSTRAKTNOST

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

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

- zdrav razum (KISS, YAGNI, DRY)
- tehnike** (strukturiranje, enkapsulacija, polimorfizam)
- načela**
- 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 suptilno: jako je važan osjećaj za **mjeru**

Važan element uspjeha je **iskustvo!**