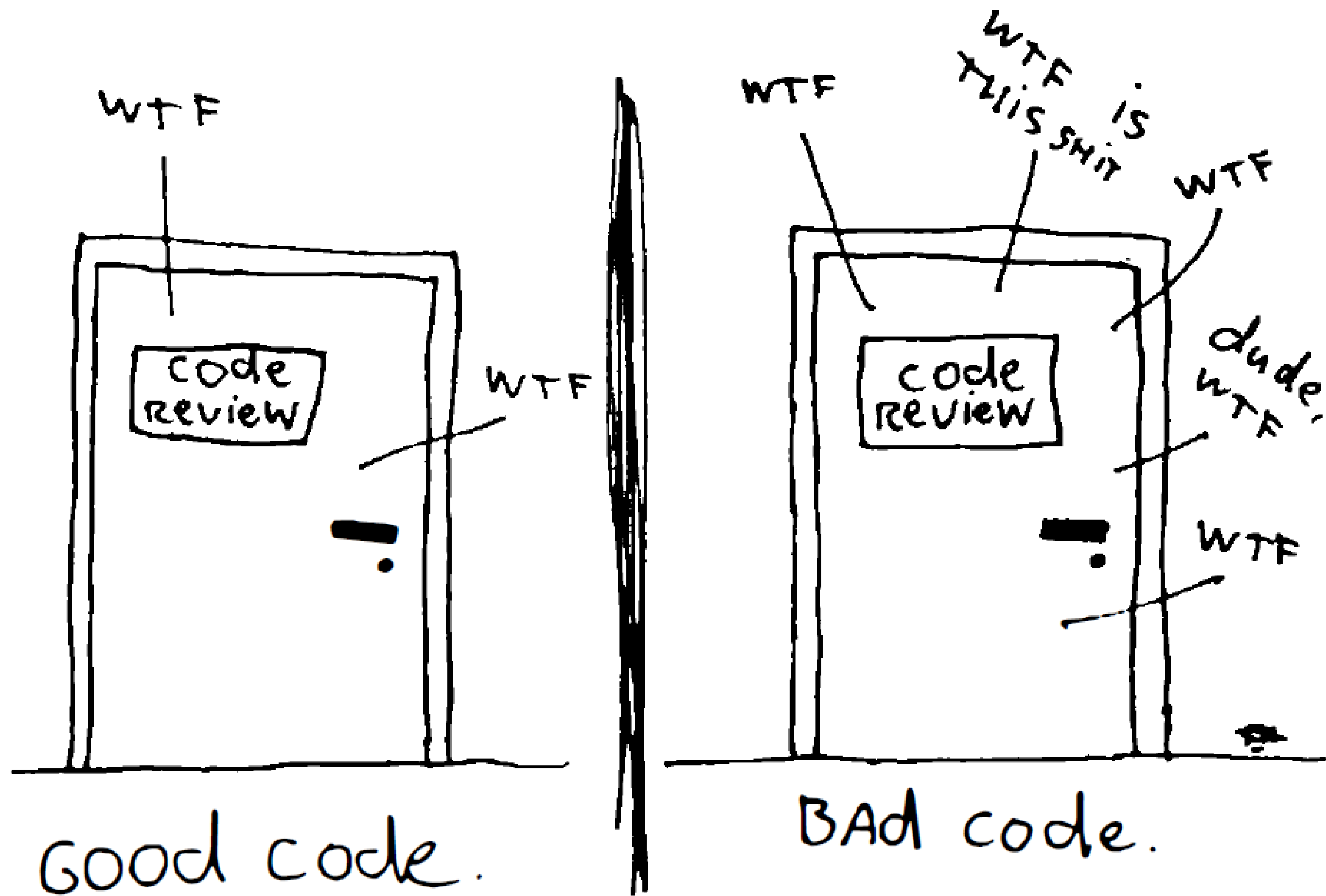
David Bortňák
Milan Kodejš

CODE REFACTORING

Jediná skutečná míra kvality kódu: Počet průšvihů za minutu



Principy refaktorování

Co je refaktorování?

- Refaktorování je disciplinovaný proces provádění změn v softwarovém systému takovým způsobem, že nemají vliv na vnější chování kódu, ale vylepšují jeho vnitřní strukturu s minimálním rizikem vnášení chyb.

- Je opak běžného chátrání software
- Při refaktorování provádíme jednoduché až primitivní kroky
- Kumulativní efekt těchto drobných změn však může podstatně vylepšit návrh softwaru

Proč refaktorovat?

- Vylepšuje návrh softwaru
- Kód se stává snadněji pochopitelným
- Pomáhá při hledání chyb
- Umožňuje programovat rychleji a efektivněji
- Zlepšuje celkovou kvalitu softwaru
- Návrh programu zůstává dobrý po celou dobu vývoje
- Spěje k rovnováze provádění změn mezi vylepšováním návrhu a přidávání nových funkcí

Kdy refaktorovat?

- Na refaktorování by se neměl vyhrazovat čas ve vývojovém cyklu, refaktorovat by se mělo průběžně

- při třetím opakování
- při přidávání funkce
- při opravování chyby
- při revizi kódu

„The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.“

Don Roberts

Proč refaktorování funguje?

- Hodnota programu – Dnes X Zítra
- Přidávání funkcionality
  - Často však nevíme jakou funkcionalitu budeme přidávat.
  - Refaktorováním měníme strukturu programu tak, aby šla nová funkcionalita jednoduše přidat.
  - Zítra totiž můžeme zjistit, že včerejší rozhodnutí bylo naivní.

- Programy, které se špatně čtou se špatně upravují
- Programy, které mají opakující se logiku se špatně upravují
- Programy, které mají složitou podmíněnou logiku se špatně upravují

Problémy s refaktorováním (1/3)

- Žádný koncept není univerzální a to ani refaktorování

- Okruhy problémů
  - Databáze (především objektové databáze)
  - Návrhové změny centrálních komponent
  - Návrhové chyby
  - Refaktorování **mění rozhraní**

Problémy s refaktorováním (2/3)

- Refaktorování centrálních komponent může být velice složité
- Je potřeba si rozmyslet jak složité by bylo refaktorovat jednoduchý návrh na složitější
- Pokud je to jednoduché můžeme zvolit ten jednodušší
- Centrální návrh by měl být dostatečně univerzální k zamýšlenému záměru

Problémy s refaktorováním (3/3)

- Refaktorování mění rozhraní (inteface)
- Pokud je rozhraní publikované (published) nemůžeme dosáhnout ke všem částem kódu které ho využívají.
  - je potřeba zachovat staré rozhraní, dokud ho uživatelé našeho kódu nepřestanou používat
  - publikovat pouze nezbytná rozhraní
  - Staré rozhraní může volat nové rozhraní
- Je potřeba označit staré rozhraní jako zastaralé (deprecated) pomocí nástrojů programovacího jazyka
- Pozor na výjimky z více rozhraní → je potřeba definovat rodiče výjimek pro celý balíček(modul)

```
procedure MyProcedure; deprecated;

function MyFunction(AParam: integer): boolean; deprecated;


TRGBTriple = record
  Red: word;
  Green: word;
  Blue: word;
  Alpha: word;
end deprecated;
```

```
__declspec(deprecated) void func1(int) {}

__declspec(deprecated("** this is a deprecated function **")) void func2(int) {}


struct __declspec(deprecated) X {
   void f(){}
};

struct __declspec(deprecated("** X2 is deprecated **")) X2 {
   void f(){}
};
```

Kdy nerefaktorovat

- Pokud je časově méně náročné napsat celý kód znovu
- Pokud obsahuje kód tolik chyb, že se nedaří ho dostat do stabilního stavu (při refaktorování by měl kód již fungovat téměř korektně)
- Velký projekt je možné nejdříve refaktorovat do několika zapouzdřených komponent a poté se rozhodnout pro každou zvlášť
- Před uzávěrkou

Refaktorování a návrh

- Refaktorování může zastat roli úvodního návrhu, ale není to nejefektivnější přístup
- Dobrý úvodní návrh může značně snížit dobu potřebnou na refaktorování
- Čím později provedeme změnu návrhu tím je dražší
- Čas strávený na vymyšlení „nejlepšího“ návrhu může být také velmi neefektivní a později můžeme zjistit že to stejně není nelepší řešení a následná změna může být nákladná
- S refaktorováním se zaměřujeme na nalezení dostačujícího a jednoduchého řešení, protože změny nejsou nákladné

	Jednoduchý návrh	Flexibilní návrh
Složitost	Nízká	Vysoká
Údržba	Levná	Drahá
Flexibilita	Získáme pouze potřebnou flexibilitu	Obsahuje mnoho nevyužité flexibility
Změny návrhu	Refaktorování	Většinou nejsou potřeba
Cena	Nízká úvodní cena	Vysoká úvodní cena
Čas dokončení projektu	Není přesně definováno	Poměrně dobře definován

Refaktorování a rychlost

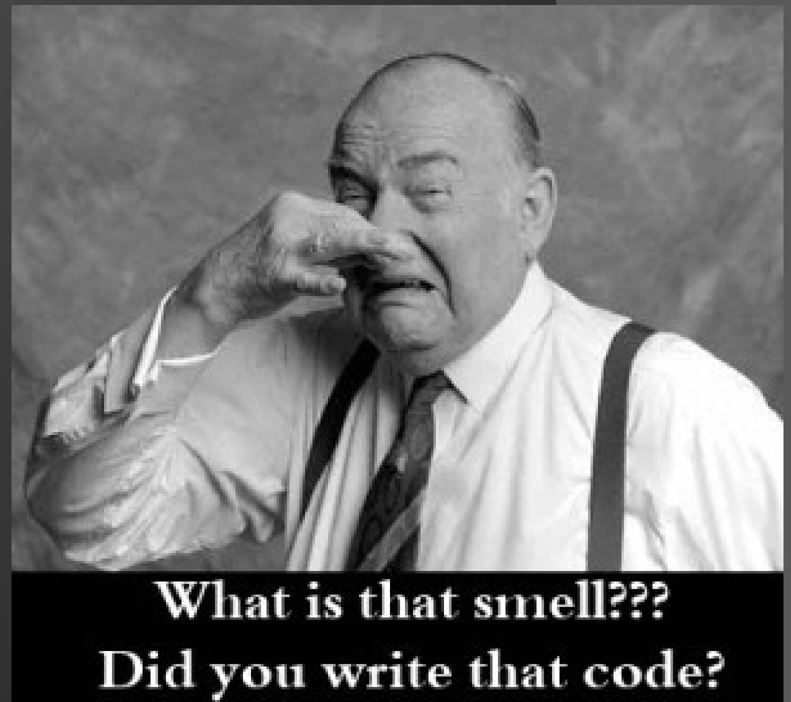
- Vylepšení srozumitelnosti programu často vede ke snížení jeho výkonu
- 90% prostředků je však využita velmi malou částí programu (méně než 10%)
  - při optimalizaci veškerého kódu je tedy více než 90% času vynaloženo neefektivně
  - V dobře strukturovaném programu jdou tyto části snadno detekovat a poté cíleně optimalizovat
- Program spustíme v profileru, který zaznamená části kódu, které využívají nejvíce času a paměti
- Lze přesněji určit místa která je potřeba optimalizovat a také jich bude méně

Vytváření testů

- Většinu času při vytváření kódu nenáleží samotnému kódování, ale debugování
- Po změně v kódu je důležité otestovat zda byla zachována korektnost dané funkcionality
- Při refaktorování provádíme mnoho změn a proto je potřeba mít k dispozici kvalitní sadu automaticky vyhodnocujících se testů.
- Nejdříve napíšeme test poté funkcionalitu, tím se při návrhu funkcionality soustředíme na rozhraní nikoliv na implementaci
- Kvalitní sada testů značně urychluje vývoj
- Více o testech v následující přednášce

Babiččino pravidlo: Když to páchne, vyměň to

Páchnoucí kód



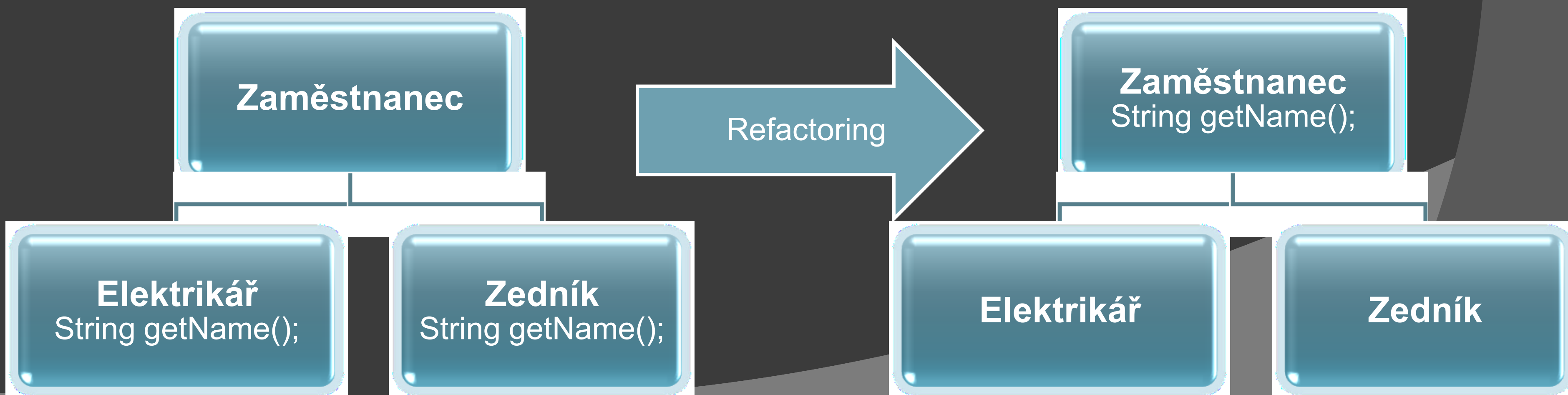
Co to je páchnoucí kód?

- symptom, který obvykle značí hlubší problém
- není to obvykle chybný úsek zdrojového kódu
- slabina v implementaci
- způsobuje zpomalení až zastavení evoluce programu
- Patří sem:

  Duplicitní kód, příliš dlouhé metody nebo struktury, dlouhý seznam předávaných parametrů, switche u oop, zbytečné komentáře

Duplicitní kód: využití dědičnosti

- u metod sourozeneckých tříd
- V takovém případě je ho možné odstranit přepsáním metody do třídy, ze které potomci dědí



Duplicitní kód: vyjmutí metody

- pokud se vyskytuje stejná část kódu ve více metodách dané třídy

```
void printOwing(){
   printBanner();
   System.out.println ("name:" + _name);
   System.out.println ("amount " + getOutstanding());
}
```

```
void printOwing() {
   printBanner();
   printDetails(getOutstanding());
}

void printDetails (double outstanding) {
   System.out.println ("name:" + _name);
   System.out.println ("amount " + getOutstanding());
}
```

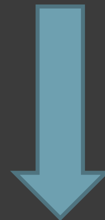
Dlouhá metoda nebo struktura

- častokrát kvůli duplicitnímu kódu
- kdysi: snaha o co nejmenší počet volání funkcí
  - nepřehlednost
  - těžké testování
- dnes: snaha o rozdělení kódu na více malých úseků
  - zpřehlednění kódu
  - lepší údržba kódu
- ve většině případů stačí vyjmout metodu

Dlouhá metoda: odstranění nepotřebných proměnných

- Použijeme takzvanou anonymní instanci nebo proměnnou

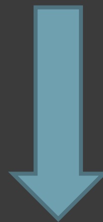
```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```



```
return (anOrder.basePrice() > 1000)
```

Dlouhá metoda: dekompozice složitých podmiňovacích výrazů

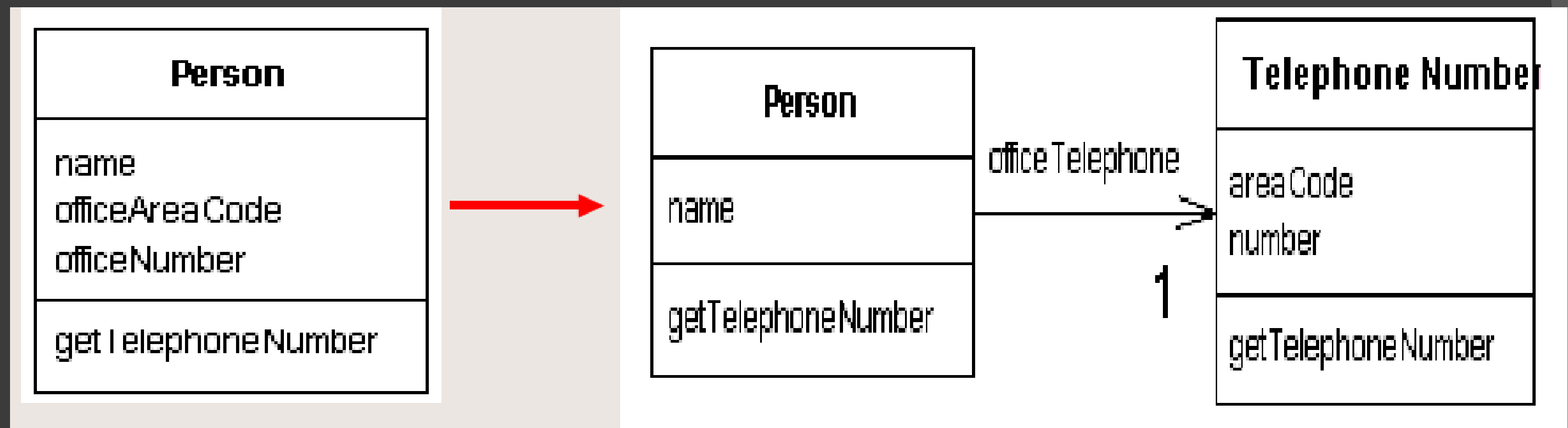
```
if (date.before (SUMMER_START) || date.after(SUMMER_END)){
    charge = quantity * _winterRate + _winterServiceCharge;
}
else {
    charge = quantity * _summerRate;
}
```



```
if (notSummer(date)) {
    charge = winterCharge(quantity);
}
else {
    charge = summerCharge (quantity);
}
```

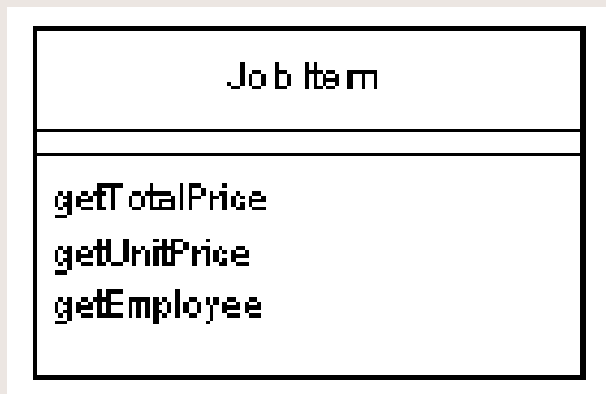
Dlouhá struktura: vyjmutí struktury

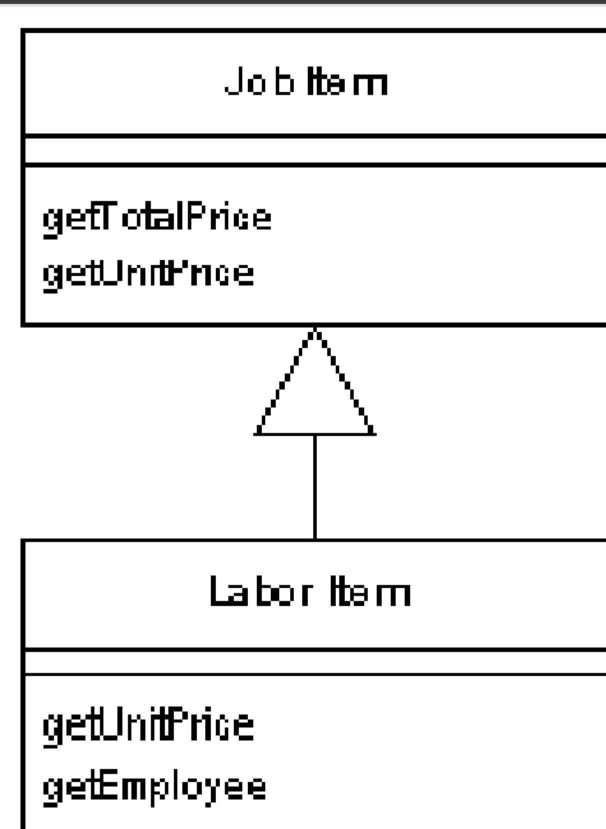
- z jedné třídy uděláme vyjmutím více tříd



Dlouhá struktura: vyjmutí struktury jako potomka

- vhodné zejména tam, kde některé metody třídy vyžívají jenom některé instance dané třídy





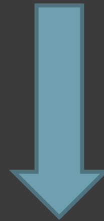
Dlouhý seznam parametrů

- Parametry nahradili používaní globálních proměnných
- Proto vznikají problémy s počtem předávaných parametrů

Dlouhý seznam parametrů: nahrazení metodou

- pokud má metoda přístup k datům, můžeme parametr odstranit

```
int basePrice = quantity * itemPrice;
double discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice, discountLevel);
```

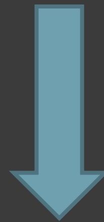


```
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

Dlouhý seznam parametrů: předávání objektem

- místo předávání jednotlivých dat struktury metodě, předáme metodě celou instanci struktury

```
int min = dennaTeplota.getMin();
int max = dennaTeplota.getMax();
mojPlan = plan.rozmedzie(min, max);
```



```
mojPlan = plan.rozmedzie(&dennaTeplota);
```

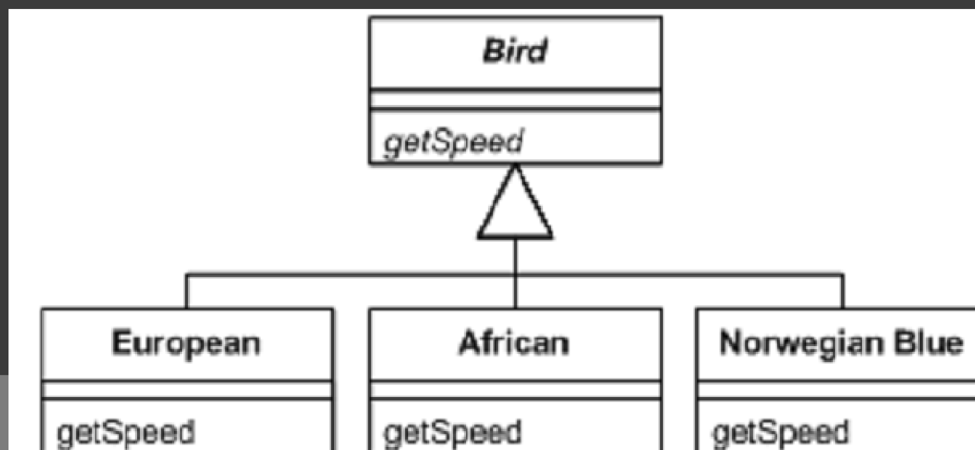
Složité switche

- Jedním z nejzřetelnějších znaků objektového programování je vzácný výskyt příkazů switch. Důvodem je zejména duplicita příkazů ve více větvích.
- Nejlepším přístupem k řešení takové duplicity je použití principu polymorfizmu. To znamená, že jestli objekty mění své chování v závislosti na svých typech není třeba zahrnovat takovou podmínku explicitně.

Složité switche: náhrada polymorfizmem

- pokud testujeme objekt podle nějakého typu

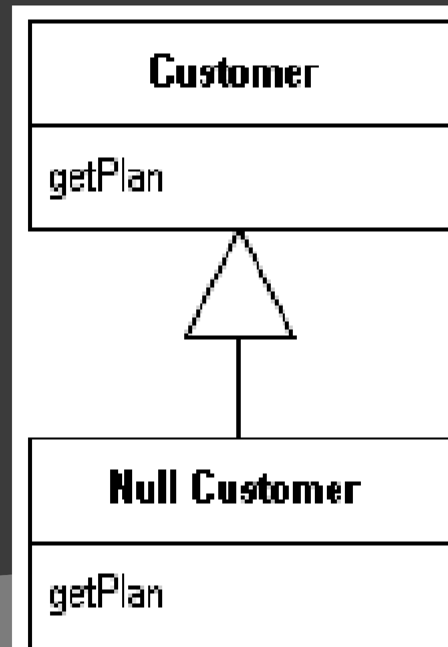
```
double getSpeed() {
   switch (_type) {
     case EUROPEAN:
         return getBaseSpeed();
     case AFRICAN:
         return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
     case NORWEGIAN_BLUE:
         return (_isNailed) ? 0 : getBaseSpeed(_voltage);
   }
   throw new RuntimeException ("Should be unreachable");
 }
```



Složité switche: zavedení objektu null

- pokud testujeme objekt na null, je lepší vytvořit instanci poděděné třídy null

```
if (customer == null)
        plan = BillingPlan.basic();
else
        plan = customer.getPlan();
```



Nesprávně umístněné komentáře

- Vznikají často jako důsledek špatného pojmenování proměnných a funkcí.
- Z toho důvodu je potřeba daný úsek kódu okomentovat, protože není jasné „co to dělá“.
- Komentáře by mněli vypadat následovně:
  - Na úrovni knihoven, programů a funkcí popisují „co to dělá“
  - Uvnitř knihoven, programů a funkcí by mněli popisovat „jak to dělá“
  - Na úrovni jednotlivých řádků kódu by mněli popisovat „proč to dělá“

Inicializace místo přiřazení

- hrozí použití nenastavené hodnoty
- C++ je pomalejší

```
void foo() {
    int i;
    ....
    i = 7;
    ....
}
```

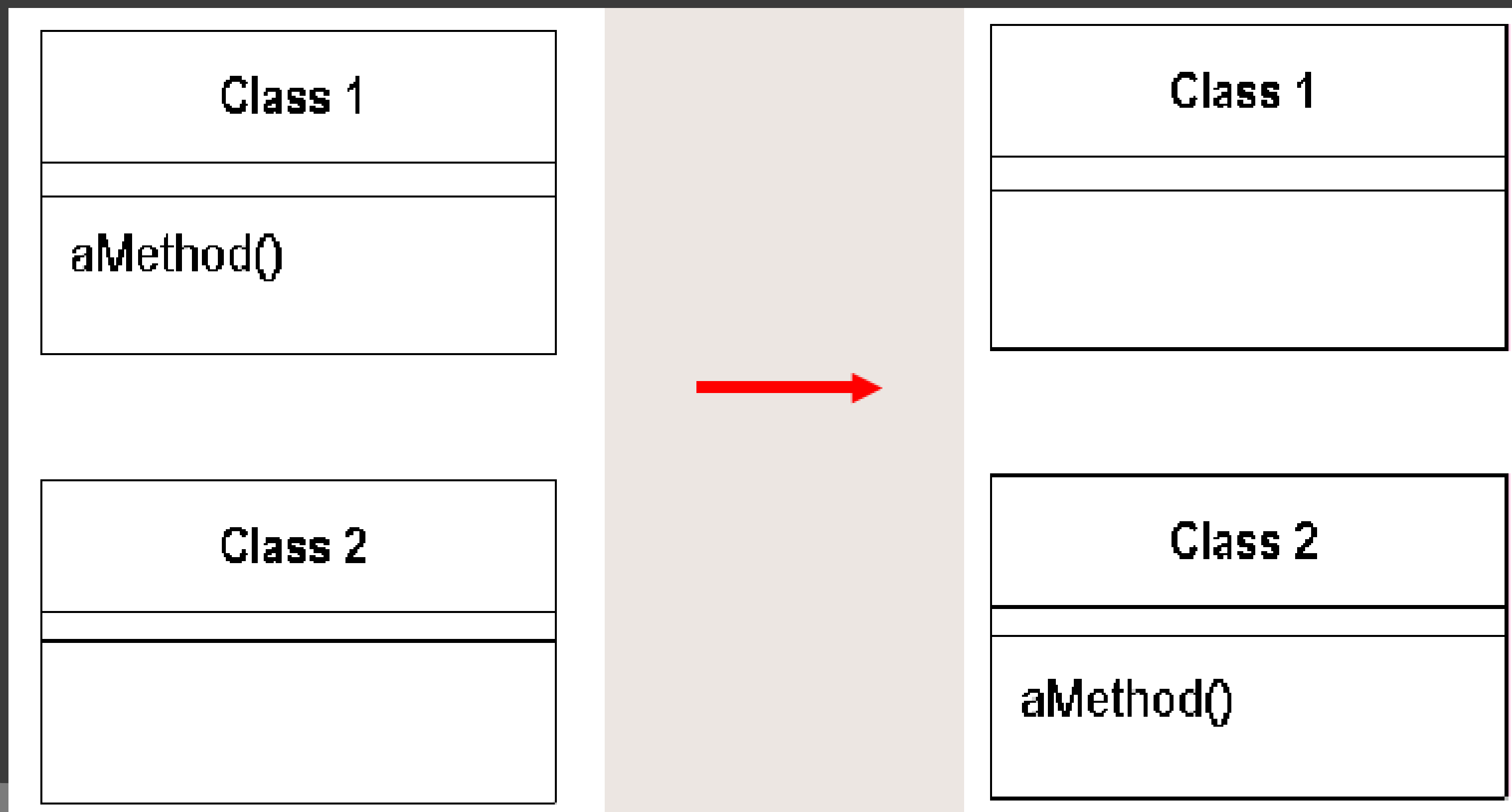
```
void foo() {
    ...
    int i = 7;
}
```

Rozptýlené úpravy

- Často při úpravě jedné metody musíme provést řadu jiných úprav, protože jedna malá změna nám ovlivní celý kód.
- Obvykle je těžké takové změny realizovat, proto je výhodné přesunout všechny změny do jedné třídy a pokud taková třída neexistuje, vytvoříme ji.

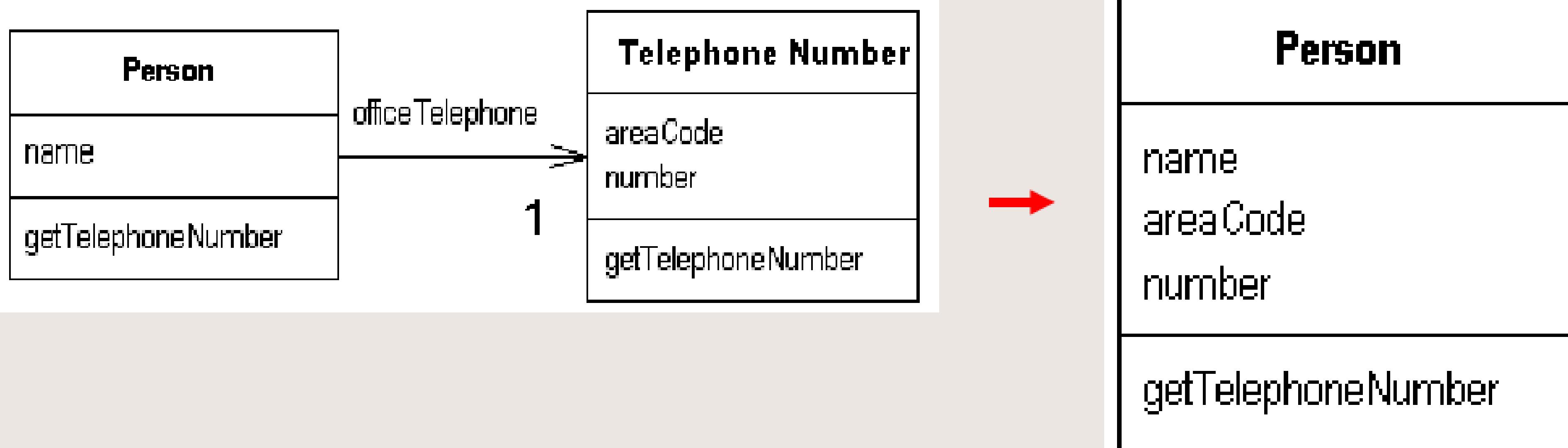
Rozptýlené úpravy: přesunutí metody

- pokud metoda více pracuje s daty jiné třídy
- v původní třídě pouhé volání metody



Rozptýlené úpravy: odstranění třídy

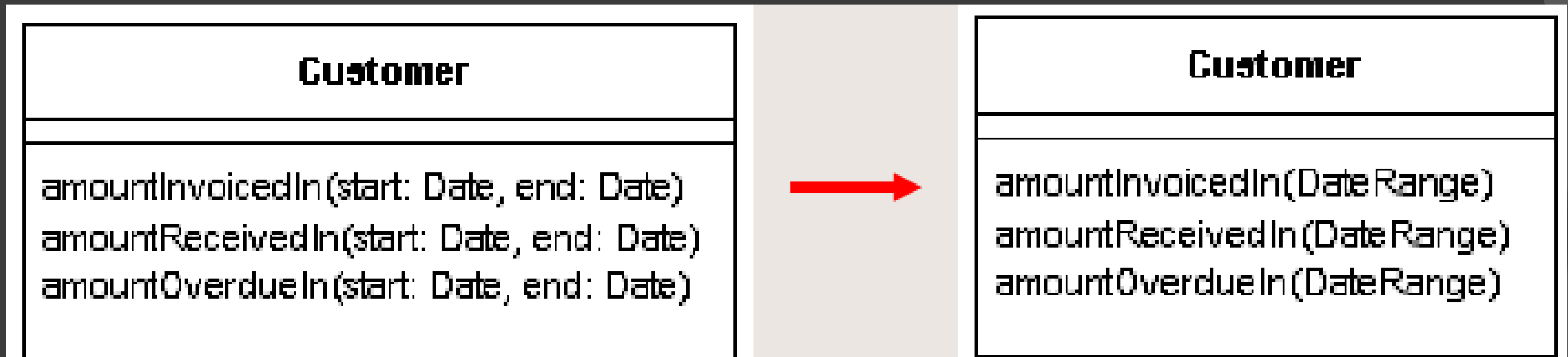
- pokud neobsahuje téměř žádné metody
- nesplňuje to, na co byla vytvořená
- zpravidla důsledek jiného refaktorování



Datové shluky

- Mnohdy je nacházíme spolu na nějakém místě v programu
- Odstraníme je jednoduše tak, že je sdružíme do objektu.

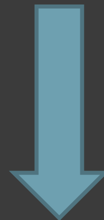
Datové shluky: zavedení objektu pro parametre



Datové shluky: nahrazení pole objektem

- pokud máme pole s prvky různého významu

```
String[] data = new String[3];
data[0] = "Jozef";
data[1] = "34";
```



```
Osoba student = new Osoba();
student.setMeno("Jozef");
student.setVek("34");
```

Další způsoby zlepšení kódu

- odstranění dvojité negace
- testování v podmínce na TRUE nikoliv na FALSE
- rozdělení cyklu – může kód překvapivě zrychlit
- nahrazení „magického“ čísla konstantou

…

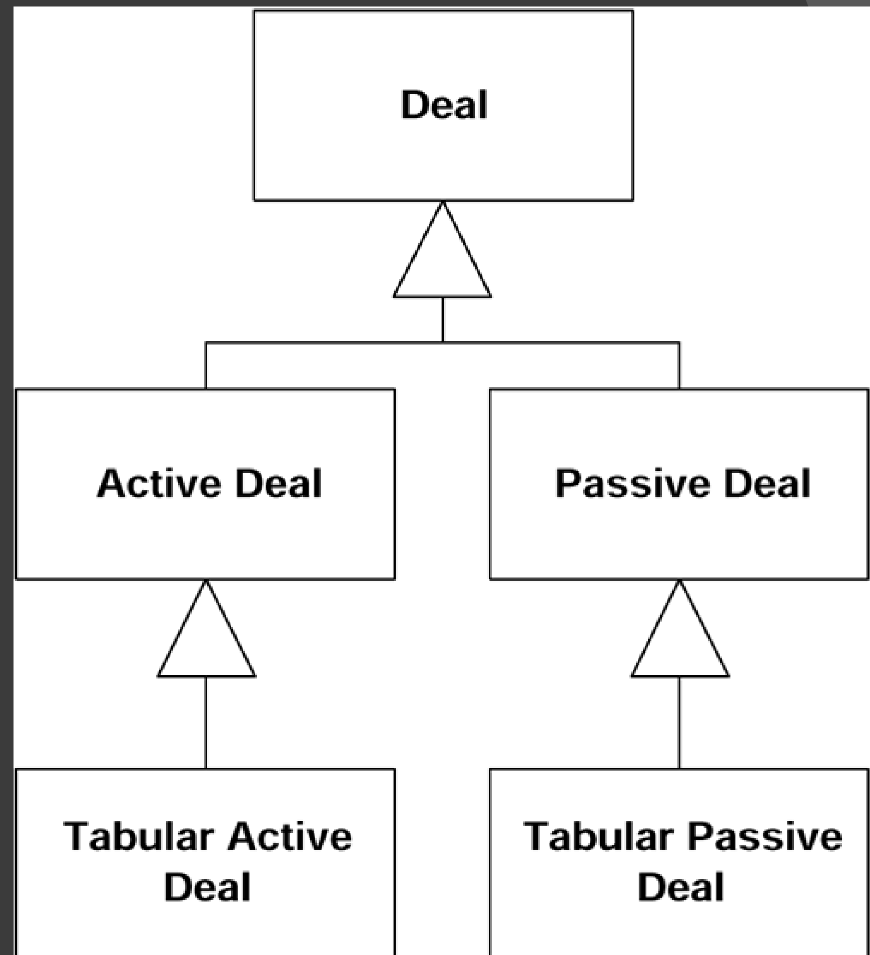
„Velká“ refaktorizace

„Velké“ refaktorizace

- Roztrhnutí dědičnosti
- Oddělení datového modelu od prezentace (MVC)
  - Již zaznělo v jiné prezentaci
- Převést procedurální návrh na objektový

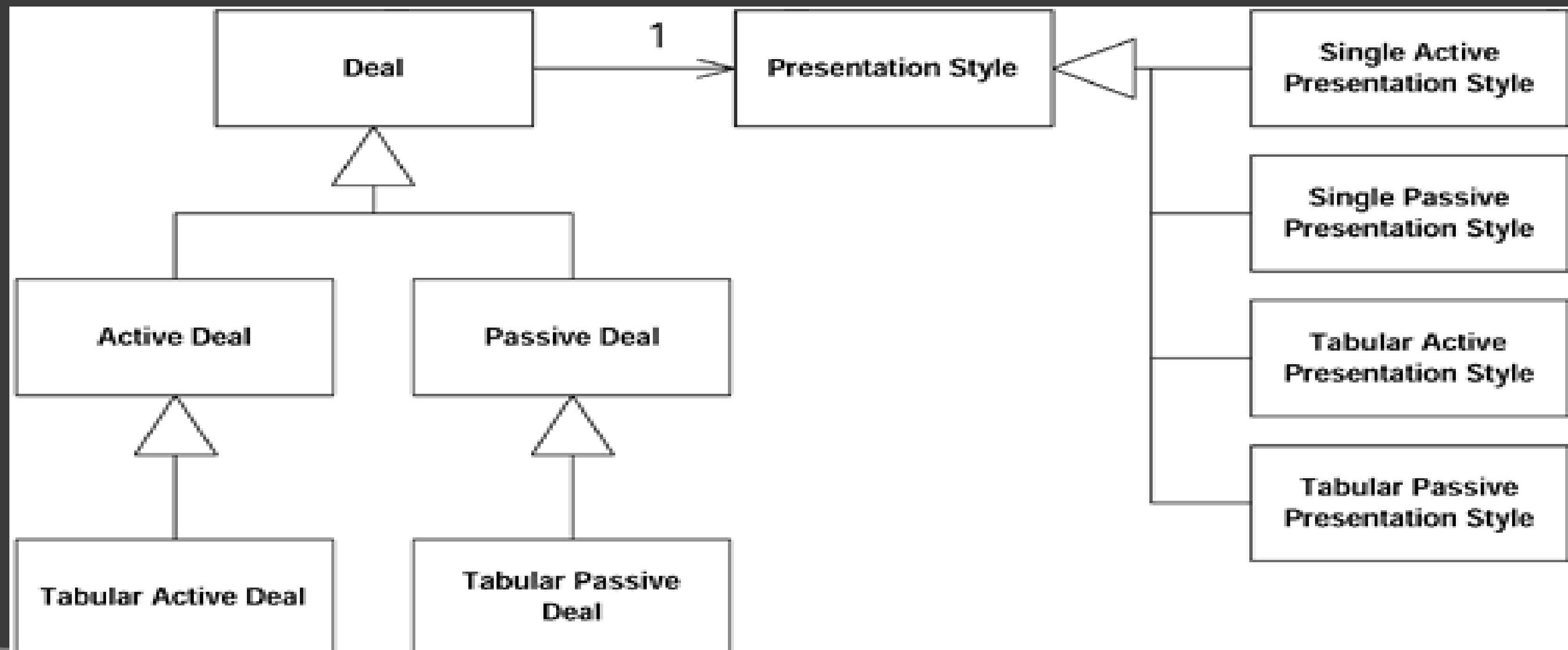
Roztrhnutí dědičnosti (1/3)

- Máme hierarchii dědičnosti, která vykonává 2 rozdílné úlohy
- Přidání dalšího typu jedné funkcionality je komplikované



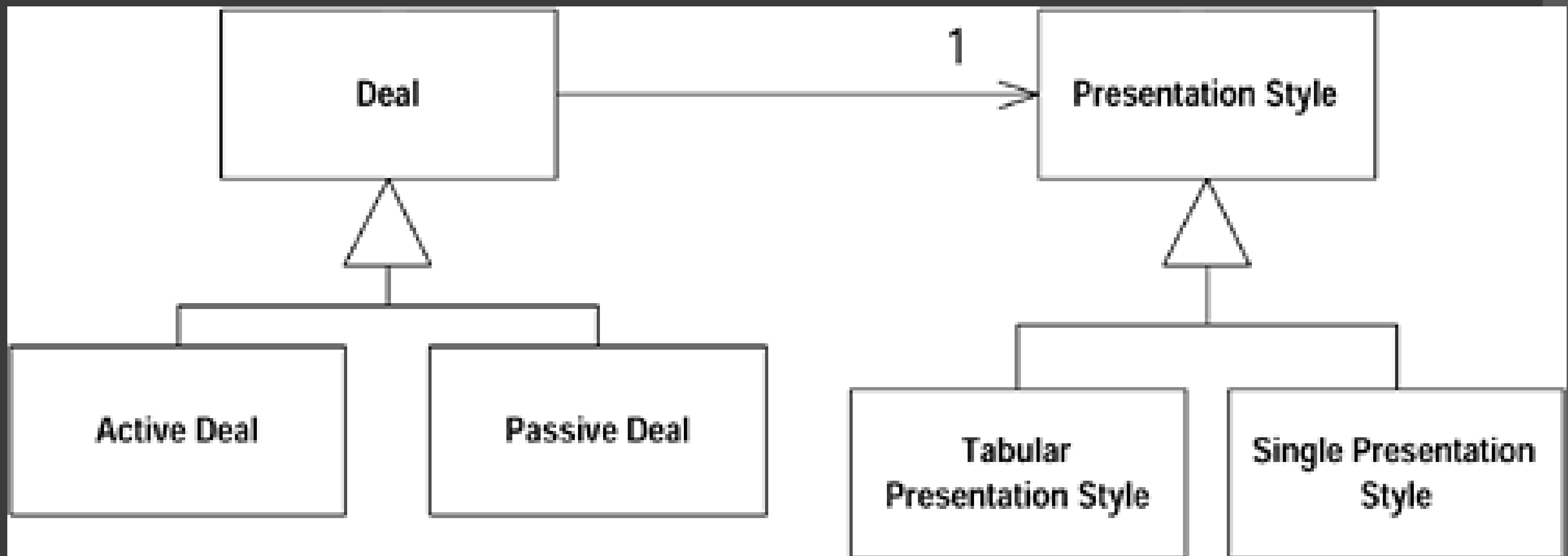
Roztrhnutí dědičnosti (2/3)

- Jednu úlohu vyčleníme do nové třídy s potomky
- Tuto třídu přidáme jako delegáta k původní třídě (třída Deal obsahuje instanci třídy Presentation style)



Roztrhnutí dědičnosti (3/3)

- Z původní třídy pouze voláme funkcionalitu nové třídy.
- Jednoduché přidávání dalších typů obou funkcionalit



Refaktorování a realita

Proč se vývojáři zdráhají refaktorovat?

- Protože neví jak
- Jsou placeni jen za nové funkce
- Na vývoji se podílí více programátorů a jednotlivé komponenty nejsou dostatečně zapouzdřeny
- Proč refaktorovat teď, když užitek přijde až za delší dobu?
- Refaktorování může znefunkčnit program (pokud je provedeno špatně)
- Pokud existuje několik verzí programu, tak musí být zkontrolována funkcionalita každé verze

Nástroje pro refaktorizaci

- Java - Eclipse, Photran, IntelliJ IDEA, NetBeans, Jdeveloper
- Delphi - Embarcadero Delphi
- Visual studio + addons – JustCode, ReSharper, Coderush, Visual Assist (C, C++, C#, .NET,…)
- DMS Software Reengineering Toolkit (VB, VB.NET., C#, C++)
- Xcode (C, C++, Java, Python, Ruby,…)

Code obfuscation je obrácený code refactoring

Code obfuscation

```
var gapi=window.gapi=window.gapi||{};gapi._bs=new Date().getT
var b=[],c;for(c in a)S(a,c)&&b[v](c);return b},T=function(a,
(0<e.c[D]?"#"+e.c[K]("&"):"")};var Ca=function(a,b,c){if(L[b+
var Pa=function(a){return a[K](",")[x](/\./g,"_")[x](/-/g,"_"
a,e)}if(!(e=c[Y.g]))if(e=Da(na[H]),!e)throw"Bad hint";var h=e
a,U)}}q&&q();return 1};0<d&&(m=L.setTimeout(function(){B=l;u(
if(!d)throw"Bad hint:"+h;f=d=d[x]("__features__",Pa(I))[x](/\
a.split("/"),b=Z(),c=0,e=a[D];b&&"object"===typeof b&&c<e;++c
session_index:e,session_prefix:d!==j&&d!==n&&""!==d?"u/"+d+"/
d);d=f[x](eb,kb);f={};T(e,f);f.hl=$("lang")||"en-US";f.origin
typeof g&&(0<c&&g>=c)&&(f.ic="1");var g=/^#|^fr-/,c={},q;for(
"___"+a+"_"+ib[a]++,b.id=q);b=R();b[">type"]=a;T(e,b);k[A]("d
q=Ba(q,m,g);m=R();T(cb,m);m.name=m.id=c;T(b.attributes,m);m.s
j;"number"===typeof a?b=a:"string"===typeof a&&(b=parseInt(a,
h[G]("class")))&&qb[ca](h))&&(k=h[1]),i=k&&(rb[k]||sb[k])&&(!
e)}},zb=function(a){var b=N(V,a,{});b.go||(b.go=function(b){r
Na[v]([Y.o,function(a,b,c){pb=c;b&&tb[v](b);for(b=0;b<a[D];b+
f[i],q=0;q<g[D];++q)h.src&&0==h.src[y](g[q])&&d[v](h);0==d[D]
j;g&&e[v](g)}f=Ya("cd");e=0;for(d=f[D];e<d;++e)$a(Z(),f[e]);f
O,p)):L.attachEvent&&(L.attachEvent("onreadystatechange",func
gapi.load("plusone",{callback:window["gapi_onload"],_c:{"anno
```

Ukázka ze zdrojového kódu tlačítka Google +1

Účel code obfuscation

- Skrýt účel kódu nebo jeho logiku
- Kvůli zamezení manipulací
- Ztížit reverzního inženýrství
- Vytvoření hlavolamu

- Často používán pro kód který kontroluje licenci softwaru
- Využíván škodlivým softwarem

Metody v code obfuscation

- Odstranění komentářů a dokumentace v kódu
- Vymazání veškerého “bílého” místa a odsazení
- Přejmenovaní identifikátorů proměnných, funkcí, typů, objektů a dalších struktur
- Generování řetězců a jiných hodnot proměnných kódem
- Přetypování
- Spuštění hodnot proměnných jako kód
- Definování funkcí a dalších nadbytečných struktur pro účely code obfuscation a jejich náhodné použití v různých částech kódu

```
#include <stdio.h>
int main() {    printf("Hello World\n"); }
```

```
#include <stdio.h> #define THIS printf(
#define IS "%s\n"
#define OBFUSCATION ,v);
double h[2]; int main(_, v) char *v; int _; { int a = 0; char f[32]; h[2%2] =
21914441197069634153456391018824026170709523170177760997320759459436800394
07307212501870429040900672146338833938303659439237740635160500855813030357
49237268288785805461648960544158982974043306599507665022915207988359711097
3562880.0 00000; h[4%3] = 1867980801.569119;  switch (_) { case 0: THIS IS OBFUSCATION
break; default: main(0,(char *)h); break; } }
```

```
#define _ F-->00 || F-OO--;
long F=00,OO=00;
main(){F_OO();printf("%1.3f\n", 4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_
            _-_-_-_
}
```

Program na výpočet π

Zdroje a doporučená literatura

- Refactoring: Improving the Design of Existing Code, 1999, Martin Fowler
- Čistý kód - Návrhové vzory, refaktorování, testování a další techniky agilního programování, 2009, Robert C. Martin
- http://refactoring.com/catalog/index.html
- http://www.cise.ufl.edu/~manuel/obfuscate/obfuscate.html