Refaktorování kódu

Matěj Klouček

Objektově orientované programování

Co je refaktorování kódu?

- Proces restrukturování kódu tak, že se nemění jeho vnější chování
- Cílem je lepší čitelnost a udržitelnost kódu, menší náchylnost k chybám a celkové zefektivnění daného programu.
- Dává smysl především pro složitější projekty, které vyžadují spolupráci více programátorů a jejichž funkčnost je nutná po delší období.
- Neprovádění refaktorování může vést k akumulaci technického dluhu
- Refaktorování, se většinou provádí na podnět nějakého *pachu kódu*
- Většina moderních IDEs má zabudované funkce pro snadné refaktorování

Testování

- Před jakoukoliv úpravou je třeba zajistit, že se vnější chování kódu opravdu nezmění
- Pro automatické testování kódu se používají *unit testy,* které ověřují správnou funkčnost dílčích částí kódu (obvykle třída či metoda)
- Po zavedení unit testů je refaktorování iterativní proces, při kterém provádíme malé změny a vždy kontrolujeme, zda jsou testy splněny

Techniky refaktorování

Extract Function

- Problém: Máme část kódu, kterou lze logicky seskupit dohromady
- Řešení: Vytvoření nové funkce, a nahrazení původního kódu voláním této funkce
- Poznámka: Předcházíme tím zároveň duplikaci, jelikož funkce může být použitá na jiném místě

```
void printOwing() {
  printBanner();

  // Print details.
  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:  " + outstanding);
}
```

Extract Variable

- Problém: Těžko srozumitelný výraz
- Řešení: Rozdělení výrazu do jednotlivých lokálních proměnných s vhodnými jmény
- Poznámka: Mírné zhoršení efektivity kódu, jelikož se vždy zavolají všechny funkce

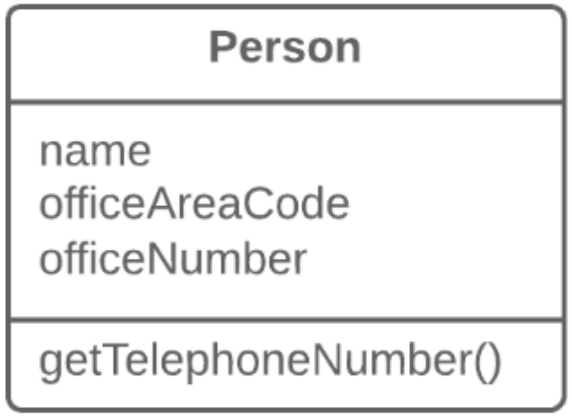
```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1) &&
      (browser.toUpperCase().indexOf("IE") > -1) &&
       wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

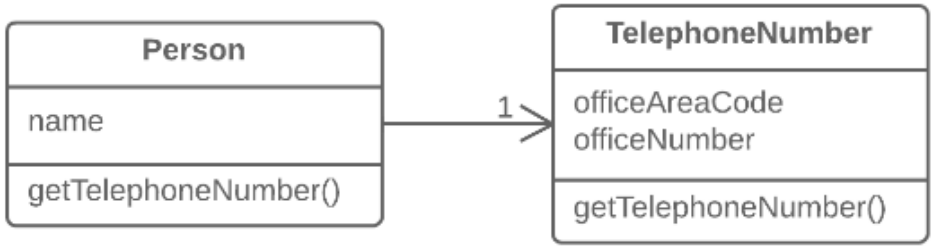
```
void renderBanner() {
  final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
  final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something
  }
}
```

Extract Class

- Problém: Jedna třída zastává práci dvou
- Řešení: Vytvoření nové třídy a přesunutí relevantní funckionality do ní
- Poznámka: Dává smysl se zamyslet jaké atributy / metody by přestaly dávat smysl, kdybych odstranil nějaký jiný atribut / metodu z dané třídy





Inline Function

- Opačná technika k *Extract Function*
- Problém: Z těla funkce je zřejmé co funkce dělá, extrahovaná funkce jen zbytečně přidává složitost
- Řešení: Nahrazení volaní funkce tělem dané funkce a následně její odstranění
- Poznámka: Funkce mohla být dříve složitější, ale postupem času byla osekána

```
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ? 2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

```
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

Inline Variable

- Problém: Zbytečná deklarace dočasné proměnné
- Řešení: Nahrazení odkazu na proměnou samotným výrazem užitým v její deklaraci
- Poznámka: Je třeba zkontrolovat, zda se proměnná nevyužívá jinde

```
boolean hasDiscount(Order order) {
  double basePrice = order.basePrice();
  return basePrice > 1000;
}
```

```
boolean hasDiscount(Order order) {
  return order.basePrice() > 1000;
}
```

Encapsulate Variable

- Problém: Přímý přístup k datům
- Řešení: Vytvoření getteru a setteru pro přístup k datům
- Poznámka: Větší flexibilita, můžu přidat validační logiku při manipulaci s daty

```
class Range {
  private int low, high;
  boolean includes(int arg) {
    return arg >= low && arg <= high;
  }
}
```

```
class Range {
  private int low, high;
  boolean includes(int arg) {
    return arg >= getLow() && arg <= getHigh();
  }
  int getLow() {
    return low;
  }
  int getHigh() {
    return high;
  }
}
```

Rename Method / Variable

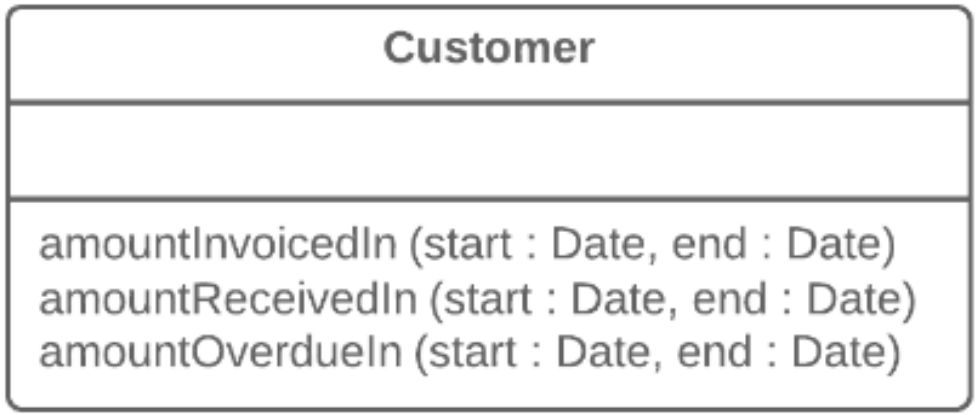
- Problém: Název funkce / proměnné nepopisuje její účel
- Řešení: Lepší pojmenování

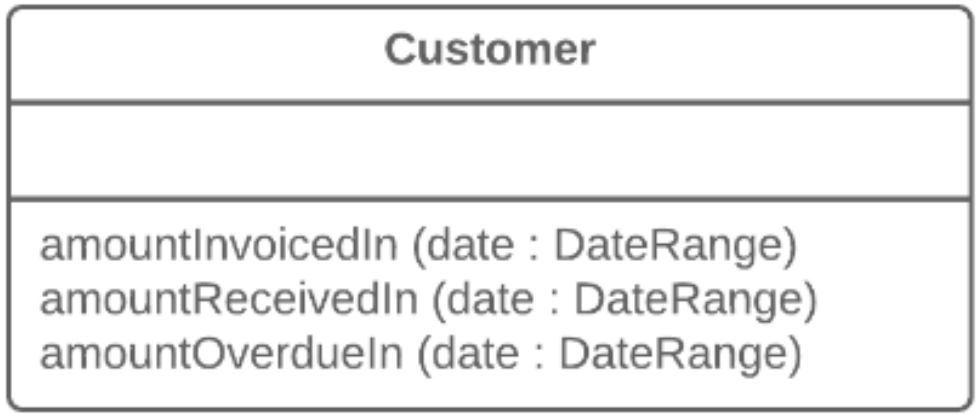
```
double a = height * width;
```

```
double area = height * width;
```

Introduce Parameter Object

- Problém: Různé funkce opakovaně používají stejnou skupiny parametrů
- Řešení: Nahrazení této skupiny parametrů objektem
- Poznámka: V rámci daných funkcí pak můžeme využít metod dané třídy pro manipulaci s daty





Combine Functions into Class

- Problém: Skupina funkcí pracují se stejnými daty
- Řešení: Přesunutí funkcí do nové metody
- Poznámka: Vyhneme se tak, zbytečnému předávání parametrů

```
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}
```

```
class Reading(Order order) {
  base() {…}
  taxableCharge() {..}
  calculateBaseCharge() {…}
}
```

Split Phase

- Problém: Část kódu, která řeší 2 věci současně... pro změnu jedné části se musím zabývat i tou druhou
- Řešení: Rozdělení na dvě navazující fáze

```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```

```
const orderRecord = parseOrder(orderString);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values = aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```

Pachy kódu

Duplicated Code

- Často vzniká, když více programátorů pracuje na různých částech stejného programu najednou nebo při copy-paste programování
- Pokaždé, když narazím na duplikovaný kód, musím precisně porovnávat, zda je opravdu duplikovaný - náročné
- Snadno řešitelné pomocí *Extract Function*
- Pokud se jedná o metody různých tříd - vyvořím super třídu těchto tříd a společnou metodu umístím do ní

Long Function / Large Class

- Dlouhé funkce a třídy jsou těžko srozumitelné a tak těžko udržitelné
- Obvykle vznikají postupem času s vývojem programu
- Často v sobě zároveň skrývají duplikovaný kód
- Pokud je zapotřebí komentářů k vysvětlení funkcionality, je vhodné funkci / třídu fragmentovat, pokud možno, pomocí *Extract Function / Class*
- Zhoršení efektivnosti je zanedbatelné oproti benefitům čistého kódu

Global and Mutable Data

- Globální data:
  - mohou být modifikována (pokud to nejsou konstanty) odkudkoliv z kódu a je těžké zjišťovat odkud – vede k těžko odhalitelným bugům
  - První pomocí je vždy *Encapsulate Variable,* čímž omezíme přístup k datům pouze přes přístupové metody
- Proměnlivá data:
  - Zdroj častých problémů: změněním dat můžeme rozbít jinou část kódu, která je očekává v jiném tvaru
  - Funkcionální programování
  - Pro větší kontrolu je vhodné použít *Encapsulate Variable*

Shotgun Surgery

- Ideálně: Pro změnu jedné funkcionality najít konkrétní bod v kódu, který je třeba změnit
- Realita: Musíme měnit mnoho zdánlivě nesouvisejících funkcí a tříd, abychom dané změny docílili
- Je snadné pak nějakou změnu přehlédnout, což vede k bugům
- Pokud možno, je vhodné přesunout dané funkce do jednoho modulu
- Pokud navíc funkce pracují na stejných datech, je vhodné je seskupit do třídy
- Často pomůže i *Inline Function*

Divergent Change

- Související (ale v jistém smyslu opačný) pach k Shotgun Surgery
- Opakované měníme stejnou část kódu z odlišných důvodů
- Například pokud musíme měnit vnitřní chování programu, pokaždé když z databáze načítáme nové objekty
  -> Interakce s databází by se měla oddělit od vnitřních procesů programu například pomocí *Split Phase* nebo *Extract Class*

Feature Envy

- V objektově orientovaném programování běžně strukturujeme kód do částí (souborů, tříd), abychom maximalizovali interakce uvnitř a minimalizovali interakce vně těchto částí
- Feature Envy je případ, kdy funkce komunikují více s funkcemi / daty vně jejich modulu více než s funkcemi / daty ve svém vlastním modulu
- Můžeme řešit přesunutím funkce (nebo její části) do modulu s kterým komunikuje nejvíce

Data Clumps

- Různé části kódu často obsahují stejné shluky proměnných
- Pokud by skupinka proměnných nedávala smysl, pokud by se jedna z nich odstranila, je vhodné zapouzdřit tyto skupiny proměnných do třídy

Primitive Obsession

- Omezování se na primitivní typy (int, string, …), přestože by bylo jednodušší si vytvořit vlastní datový typ pomocí nové třídy
- Vede k zbytečnému nabobtnávání existujících tříd - přidáváme mu více a více primitivních atributů místo jedné třídy
- Příklad: Ukládání telefonního čísla jako string - pomocí třídy bychom si mohli vytvořit metody pro jeho korektní zobrazování jinde v programu

Speculative Generality

- Ve vývoji se často vytváří kód, který by měl sloužit jako základ pro budoucí potencionální funkcionality, které ale občas nejsou nikdy implementovány
- V kódu tak zůstávají abstraktní nevyužité / málo využívané třídy a funkce
- Je vhodné se jich zbavit buď spojením podtříd a supertříd, nebo úplným odstraněním nevyužívaného kódu

Middle Man

- Zapouzdření může mít za důsledek vznik třídy, jejíž hlavní (nebo dokonce jediná) funkce je delegování práce na jinou třídu
- V takovém případě je vhodné se prostředníka zbavit a komunikovat pouze s cílovou třídou

Data Class

- Třídy které obsahují pouze atributy s daty, gettery a settery pro přístup k těmto datům
- Slouží tedy jenom jako úložiště dat
- Je vhodné najít funkce, které s těmito daty pracující a přesunout je sem jako metody

Zdroje

- Fowler, M.
  Refactoring: Improving the Design of Existing Code, Second Edition.
  Addison-Wesley, 2018.
- Refactoring guru
  https://refactoring.guru/refactoring