Doporučené postupy v programování

Návrh tříd

Abstrakce · Zapouzdření · Dědičnost vs. kompozice · Polymorfizmus · Immutability

Lubomír Bulej

KDSS MFF UK

Abstrakce

Zjednodušený pohled na složité věci

- eliminace (pro daný kontext) nepodstatných detailů

Třída jako nositel abstrakce

- reálné a abstraktní objekty z domény problému
- abstraktní datové typy + dědičnost a polymorfizmus

Poznámky:

Steve McConnel:

> Způsob uvažování při programování se vyvíjel spolu se složitostí programů, které bylo nutné vytvářet. Zatímco v dávných dobách programátor přemýšlel o jednotlivých příkazech a jejich sekvenci, v 70. a 80. letech 20. století začal pracovat s konceptem rutin. V 90. letech se rozšířilo (vniklo již dříve) objektové programování, které stále představuje hlavní paradigma pro tvorbu programů. Dnes tedy programátoři pracují s konceptem tříd a objektů.
>
> Třída představuje spojení dat a rutin, které slouží nějakému dobře definovanému účelu. Třída se obejde i bez dat – v takovém případě představuje sdružení služeb, které opět pojí nějaký společný účel. Klíčem k efektivitě programátora je maximalizace části programu, kterou je možné pustit z hlavy aniž by to mělo negativní důsledky tu na část, se kterou pracuje. V tomto ohledu třídy představují hlavní nástroj, jak toho dosáhnout.

Bez ohledu na podporu v konkrétmím jazyce třídy představují pouze technický prostředek k zápisu programů. Pokrok v programování však přicházel vždy především se zvyšováním úrovně abstrakce, na které se o programu přemýšlí. Tento posun v abstrakci představují abstraktní datové typy, které popisují zároveň data i operace, které s těmito daty pracují. Objektově-orientované programování je tedy primárně o práci s abstraktními datovými typy.

Abstraktní datové typy?

~~Abstraktní matematické struktury~~

- ~~mějme množinu ..., prvky mají následující vlastnosti ...~~
- ~~definujme operace ..., předpokládejme ..., lze ukázat ...~~
- ~~definice ..., lemma ..., věta ..., důkaz ..., složitost ...~~

Nástroj pro práci v jazyce problému

- definují data a operace pro manipulaci s nimi
- umožňují manipulovat s "reálnými" entitami, i když ty nemusí být nutně hmatatelné
  - `HttpRequest`, `Player`, `Shape`, `Font`
- při přemýšlení o problému umožňují nezabývat se implementačními detaily – místo vložení položky do seznamu se bavíme o
  - vložení buňky do tabulky, přidání nového typu okna do seznamu typů, přidání vagónu k soupravě při simulaci vlaku, atd.

Poznámky:
Slovo "reálné" je v uvozovkách, protože problém, který řešíme, se reality mimo počítač vůbec nemusí týkat. Třeba třída `HttpRequest` nepředstavuje nic opravdu hmatatelného, je to (obvykle) jen sekvence bajtů zaslaná po síti.

Příklad: práce s fonty

Ad-hoc řešení

```
currentFont.size = 16
currentFont.size = PointsToPixels (12);

currentFont.sizeInPixels = PointsToPixels (12);

currentFont.attribute |= 0x02;
currentFont.attribute |= FONT_ATTRIBUTE_BOLD;

currentFont.bold = true
```

Abstraktní datový typ

```
currentFont.setSizeInPixels (sizeInPixels);
currentFont.setSizeInPoints (sizeInPoints);
currentFont.setWeight (FontWeight.BOLD);
currentFont.setTypeFace (typeFaceName);
currentFont.setStyle (FontStyle.ITALIC);
```

Hlavní výhody ADT

Skrytí implementačních detailů

- omezuje složitost, se kterou je nutno pracovat
- umožňuje výměnu implementace
- omezuje šíření změn programem

Informativní rozhraní

- správnost programu je zjevnější
- operace jsou samovysvětlující

Související věci jsou pohromadě

- není nutné např. předávat struktury po celém programu

Třída jako nositel abstrakce

Abstrakce je určena rozhraním třídy

- abstrahuje od implementačních detailů, které skrývá
- důležité je vytvářet dobré, konzistentní abstrakce
- rozhraní je tedy nejdůležitější částí návrhu třídy

Jak se pozná dobrá abstrakce?

- orientace na problém
- konzistence

Zásady pro návrh třídy

Jasně definujte povinnosti/zodpovědnost

- třída by měla dělat jednu věc, a dělat ji dobře
  - *Třída `ErrorMessages` představuje seznam chybových hlášení, reprezentovaných třídou `Message`.*
  - pozor na "božské" třídy, které všechno ví a všechno umí
- uvažujte na úrovni abstraktních datových typů
  - Jaký ADT třída reprezentuje?

Specifikujte kontrakt objektu

- invarianty
- interakce s okolím
- kontrakty jednotlivých metod

Příklad: třída reprezentující program

Rozhraní plné různých konceptů

```
public class Program {
  ...
  public void initializeCommandStack ();
  public void pushCommand (Command command);
  public Command popCommand ();
  public void shutdownCommandStack ();
  ...
  public void initializeReportFormatting ();
  public void formatReport (Report report);
```

```
    public void printReport (Report report);
    ...
}
```

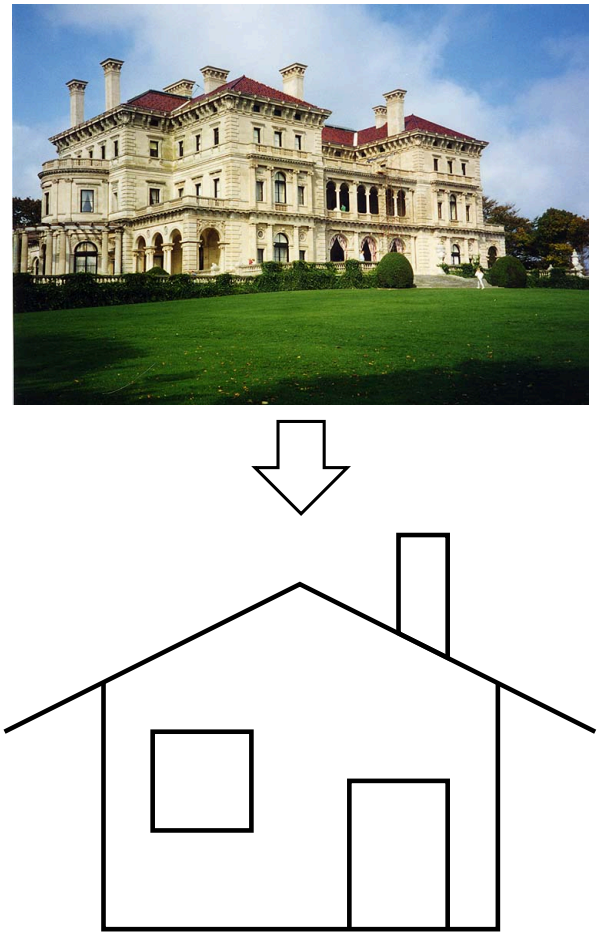
Zjednodušené rozhraní

```
public class Program {
  ...
  public void initializeProgram ();
  public void shutdownProgram ();
  ...
}
```

Zásady pro návrh třídy

Vytvářejte konzistentní abstrakce

- poskytuje zjednodušený pohled na složité věci
  - umožňuje členit inherentní a odfiltrovat zavlečenou složitost
  - umožňuje zabývat se pouze (pro daný kontext) podstatnými detaily
- u software eliminuje nutnost znalosti implementačních detailů
  - projevuje se na všech úrovních návrhu
- řada objektů v reálném světě představuje nějakou formu abstrakce
  - dům, dveře, klika, auto, ...



Poznámky:

Abstrakce samozřejmě nejsou samospasitelné, většina z nich má nějaké díry a k jejich efektivnímu používání většinou potřebujeme vědět, co abstrahují:

- Joel Spolsky: The Law of Leaky Abstractions

Což však z pohledu návrhu zas tolik nevadí, důležité je, že nám umožňují se některými nepodstatnými detaily nezabývat, pokud to není potřeba.

Zásady pro návrh třídy

Dbejte na jednotnou úroveň abstrakce v rozhraní

- sledujte kohezi metod poskytovaných v rozhraní
  - nízká koheze indikuje špatnou abstrakci – opačně to však neplatí
- nepřidávejte do rozhraní metody, které nejsou konzistentní s poskytovanou abstrakcí
  - eroze rozhraní v důsledku modifikace – broken windows
  - pozor na použití dědičnosti – přebírá rozhraní base class
- nepřidávejte do rozhraní metody jen proto, že používají pouze jeho veřejné metody

Poznámky:

Eroze v tomto případě znamená tendenci přidávat do objektů různé pomocné metody, které se hodí uživatelům objektu v různých jiných částech kódu. Zejména u větších projektů s více programátory se často stává, že přidané metody nesouvisí s abstrakcí představovanou objektem a jeho zodpovědností. Vzniká tak zamotaný, nestrukturovaný kód.

Asi jediný způsob, jak se erozi vyhnout, je disciplína programátorů a případně jednoznačné vlastnictví kódu (kdy vlastník nepovolí nesystematický zásah do svého kódu).

Příklad: třída reprezentující tým sportovců

Rozhraní s různými úrovněmi abstrakce

```
public class Team extends ArrayList <Athlete> {
  ...
  // public methods
  public void setName (TeamName teamName);
  public void setCountry (CountryCode countryCode);
  ...
  // public methods inherited from ArrayList
  public void add (Athlete athlete);
  public void clear ();
  public boolean isEmpty ();
  public void ensureCapacity (int minCapacity);
  ...
}
```

Příklad: třída reprezentující tým sportovců

Rozhraní s konzistentní úrovní abstrakce

```
public class Team {
  ...
  // public methods
  public void setName (TeamName teamName);
  public void setCountry (CountryCode countryCode);
  ...
  public void addMember (Athlete athlete);
  public void removeMember (Athlete athlete);
  ...
  private List <Athlete> members;
}
```

Příklad: třída reprezentující zaměstnance

Původní rozhraní (před modifikací)

```
public class Employee {
  ...
  public Employee (...);
```

```
    public FullName getFullName ();
    public Address getAddress ();
    public PhoneNumber getWorkPhone ();
    public PhoneNumber getHomePhone ();
    public TaxId getTaxId ();
    public JobClassification getJobClassification ();
    ...
}
```

Příklad: třída reprezentující zaměstnance

Eroze rozhraní při modifikaci

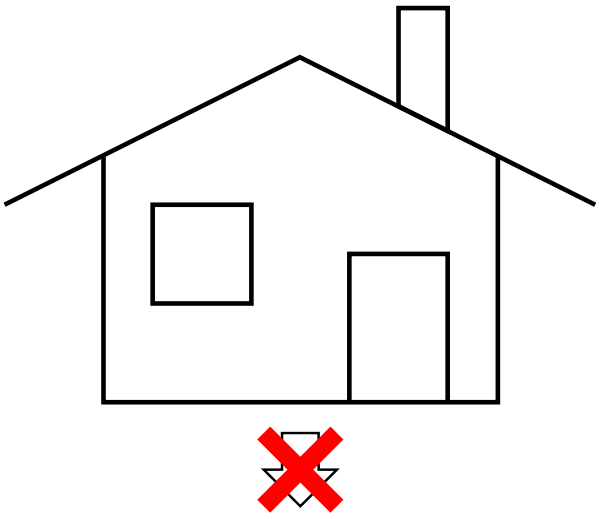
```
public class Employee {
  ...
  public Employee (...);
  public FullName getFullName ();
  public Address getAddress ();
  public PhoneNumber getWorkPhone ();
  public PhoneNumber getHomePhone ();
  public TaxId getTaxId ();
  public JobClassification getJobClassification ();
  ...
  public boolean isJobClassificationValid (JobClassification job);
  public boolean isZipCodeValid (Address address);
  public boolean isPhoneNumberValid (PhoneNumber phoneNumber);
  ...
  public SqlQuery getQueryToCreateNewEmployee ();
  public SqlQuery getQueryToModifyEmployee ();
  ...
}
```

Zapouzdření

Zapouzdření

Doplněk k abstrakci

- abstrakce odfiltruje nepodstatné detaily



Zapouzdření znemožňuje opuštění poskytované abstrakce

- skrývání vnitřních informací o objektu před okolím
- důsledkem je volnější vazba ke zbytku systému
  - poskytuje větší flexibilitu implementaci
  - umožňuje nezávislé testování malých celků
- WYSIWYG → WYSIAYG (What You See Is All You Get)

Jak dosáhnout zapouzdření?

Minimalizujte viditelnost všeho

- veřejné třídy by neměly mít žádné veřejné atributy
- modifikátor protected používejte po pečlivém zvážení
- přístup se dá uvolnit vždy, omezit už málokdy

Přístup k atributům tříd

- mutable atributy znemožňují vynucování invariantů a jsou thread-unsafe
- atributy jsou součástí rozhraní, nelze je např. přesunout do nadtřídy
- použijte pomocné metody – getters/setters, accessors
- analogie ke SmallTalkovské komunikaci pomocí zpráv

Jak dosáhnout zapouzdření?

Skryjte implementační detaily

- implementace perzistence, low-level výjimky, ...
- výjimečně možno lehce porušit – "Leaky abstraction"
- pozor na dědičnost – porušuje zapouzdření

Poznámky:

K porušování skrývání implementačních detailů: Existuje mnoho toolkitů obalujících část Win32 API pro tvorbu GUI. Typicky jde o hierarchii tříd, na jejímž konci se nacházejí třídy pro jednotlivé ovládací prvky. Málokdy je možné a vhodné implementovat do obalující třídy všechny vlastnosti, které dotyčný ovládací prvek ve Win32 API má – třída by až příliš narostla. Pragmatické řešení je porušit zapouzdření a zveřejnit handle ovládacího prvku z Win32 API. Kdo bude potřebovat, může vlastnosti ovládacího prvku, které zapouzdřující třída neobsahuje, používat přímo pomocí handle a Win32 API funkcí. Takto byl problém například vyřešen v Borland Delphi.

Sémantické porušení zapouzdření

Vyhněte se sémantickému porušení zapouzdření

- používání znalostí o fungování vnitřností třídy, které nejsou uvedeny v jejím kontraktu
  - spoléhání na automatické zavření souboru při destrukci příslušného objektu
  - předpoklady o platnosti ukazatelů vrácených metodou objektu, který už není viditelný
- zrádné – kompilátor ani jiný nástroj nezkontroluje
- porušujete kdykoliv se koukáte na kód třídy místo její dokumentace

Poznámky:
K porušování sémantického zapouzdření: Pokud jste nuceni se podívat do zdrojového kódu používané třídy, znamená to, že třída má buď špatnou úroveň abstrakce nebo nedostatečnou dokumentaci. Správnou akcí je v tuto chvíli donutit autora třídy k nápravě, je-li to možné.

Dědičnost vs. kompozice

Kompozice

Objekt obsahuje více jiných objektů

- A "has a" B
- *A car has an engine and four wheels.*

```
class Car {
  private Engine engine;
  private Wheel[] wheels;
}
```

Agregace vs. kompozice

agregace
: objekt obsahuje jiné objekty jako části, ty však mohou být součástí více objektů a často mohou existovat i bez samotného agregátu

kompozice
: objekty mohou být součástí pouze jednoho objektu, jejich existence bez kompozitu často nedává smysl

Dědičnost

Reprezentace vztahu typu "is a"

- objekt A je specializací objektu B – A "is a" B

Generalizace vs. specializace

generalizace
: vymezení společných vlastností nějaké množiny objektů

specializace
: vymezení podmnožin v nějaké množině objektů

Dědění rozhraní vs. dědění implementace

dědění rozhraní
: po nadtřídě dědíme jen signaturu rozhraní, nikoliv kód

dědění implementace
: po nadtřídě dědíme signaturu rozhraní i kód

Poznámky:
Dědičnost tříd umožňuje definovat implementaci jednoho objektu pomocí implementace jiného objektu. Jedná se tedy o mechanizmus pro sdílení kódu a reprezentace. Oproti tomu dědičnost rozhraní (subtyping) popisuje kdy může být jeden objekt použit na místo jiného.

Hlavní výhody dědičnosti

Omezení duplicity kódu

- společné rysy skupiny třídy sdílejí definici [a implementaci]
  - base class definuje společné rysy na jednom místě
  - odvozené třídy definují své specifické rysy

Stejné zacházení s více třídami

- pro některé operace stačí rysy, které vykazuje společná nadtřída
- mechanizmus pro realizaci polymorfizmu

Použití dědičnosti

Substituční princip (Barbara Liskov)

- 1. nejdůležitější zásada dědičnosti
- použití dědičnosti pouze pokud je (is-a) podtřída skutečně specializací nadtřídy
- test – Hunt & Thomas
  - Všude, kde lze použít nějaká třída, musí jít použít i její podtřída, aniž by uživatel poznal rozdíl.
- netýká se jen syntaxe, ale i sémantiky
  - podtřída musí dodržovat kontrakt nadtřídy (může oslabit pre-conditions a posílit post-conditions)
  - Příklad: `Square` vs `Rectangle`

Vztah (is-a) musí být trvalý

- třída `Employee` dědí z třídy `Person`
- třída `Supervisor` dědí z třídy … ?
- `Employee` a `Supervisor` mohou být role

Zásady pro práci s dědičností

Třídu navrhněte pro dědění, nebo jej zakažte

- 2. nejdůležitější zásada dědičnosti
- k návrhu patří i dokumentace, tj. jakým způsobem se má využít rozhraní dědičnosti
- zákaz dědičnosti: `final` (Java), `sealed` (C#)

Návrh pro dědičnost = netriviální rozhodnutí a odpovědnost

- nutno navrhnout vnější rozhraní a rozhraní pro dědičnost
- rozhraní pro dědičnost je náchylné k porušení zapouzdření

Poznámky:
Viz *Effective Java: Programming Language Guide, Item 15*.

Příklad: počet přidání elementu do množiny

```
public class CountingHashSet <E> extends HashSet <E>
  private int addCount = 0;

  public CountingHashSet () {}

  public CountingHashSet (int initCap, float loadFactor) {
    super (initCap, loadFactor);
  }

  @Override public boolean add (E e) {
    addCount++;
    return super.add (e);
  }

  @Override public boolean addAll (Collection <? extends E> c) {
    addCount += c.size ();
    return super.addAll (c);
  }

  public int getAddCount {
    return addCount;
  }
}
```

Příklad: počet přidání elementu do množiny

Co zobrazí následující kód?

```
CountingHashSet <String> s = new CountingHashSet <String> ();
s.addAdll (Arrays.asList ("Snap", "Crackle", "Pop"));
System.out.println ("Element additions: "+ s.getAddCount ());
```

Návrh pro dědičnost

Co vše je nutné zvážit?

- Jaká má být viditelnost atributů, metod a dalších prvků?
- Které metody mají být virtuální?
  - virtuální metoda = extension point
  - pozor na jazykové defaults (Java vs C#)
- Které metody mají být abstraktní?
- Použít abstraktní bázovou třídu a šablonové metody? Jaké?
- Nesnížíme příliš flexibilitu pokud dědičnost zakážeme?
  - Příklad: `String` v Javě

Poznámky:

K virtuálním metodám: Každá virtuální metoda je extension point, do kterého může svůj kód umístit odvozená třída. Ta může "vyvádět psí kusy", porušit některé invarianty, způsobit reetrantnost, na kterou nejste připraveni apod. V praxi naštěstí zdá se k těmto problémům příliš nedochází – kvůli nim si zatím nikdo příliš nestěžoval, že v Javě jsou všechny metody implicitně virtuální.

Často se však objevují stížnosti z důvodu výkonu. Zavolání virtuální metody trvá o něco déle než zavolání obyčejné, takže ve výkonnostně kritických aplikacích (např. hry) nebo úsecích kódu se

někdy vyplatí nad virtuálními metodami uvažovat i z tohoto hlediska. Nejjistější je jako obvykle takové rozhodnutí založit na nějakých faktech.

Ke snižování flexibility: Tím, že zabráníme uživateli dědit od nějaké třídy, zabraňujeme mu vytvořit si její upravené verze, ve kterých si např. dopíše různé pomocné metody. To je velký problém u tříd v knihovnách Javy, které jsou často `final` a přitom jsou navrženy poměrně minimalisticky. Důsledek je, že skoro v každém větším projektu v Javě se najde spousta pomocných tříd (např. `StringUtils`), které potřebné metody implementují jako statické.

Jednou ze zajímavých vlastností C# 3.0 je, že obsahuje mechanizmus, který zajišťuje, že se tyto pomocné statické metody dají volat, jako by to byly metody původní třídy. Něco podobného se již dá najít i v řadě jiných jazyků, protože to umožňuje zachovat původní typ malý a potřebná rozšíření nechat na uživatelích.

Zásady pro práci s dědičností

Vyhněte se příliš složitým hierarchiím

- Více jak 3 úrovně ⇒ co je opravdu cílem?
- Někdy pomůže návrhový vzor Decorator

Společné věci přesuňte v hierarchii co nejvýše

- usnadňuje jejich použití v podtřídách
- pozor na zachování konzistence abstrakce

Pozor na třídy s jen jednou podtřídou

- signalizuje "přemýšlení dopředu"
- nemusí platit pro knihovny

Poznámky:

K třídám s jen jednou podtřídou: Podobně platí i pro rozhraní s jen jednou implementací, nicméně tam může být situace trochu odlišná. Rozhraní může být použito k omezení rozhraní poskytovaného třídou nebo k vytvoření určitého pohledu (facet) na třídu.

Zásady pro práci s dědičností

Pozor na prázdnou předefinovanou metodu

- Možné narušení sémantiky (absence chování)
- Příklad: `Stream.flush` a `MemoryStream.flush`

Nevolejte virtuální metody z konstruktoru

- atributy, které metoda používá, nemusí být ještě inicializovány
  - např. je možné vidět dva různé stavy `final` atributu
- týká se i metod `clone()` nebo `readObject()` v Javě

Poznámky:

K prázdné předefinované metodě: Příkladem budiž abstraktní třída `Stream`, představující nějaký výstupní datový proud. Mezi jejími metodami je i metoda `flush`, která zapíše na disk data držená v bufferu. V memory-based streamu `MemoryStream` taková metoda pochopitelně bude prázdná. To je špatně. Správné by bylo zjemnit hierarchii tříd, rozdělit je na bufferované/nebufferované nebo disk-based/memory-based a metodu `flush` přidat jen do té větve hierarchie, kde ji třídy budou opravdu implementovat.

Zásady pro práci s dědičností

Vyhněte se vícenásobné dědičnosti implementace

The one indisputable fact about multiple inheritance in C++ is that it opens up a Pandora's box of complexities that simply do not exist under single inheritance. – Scott Meyers

- málokdy opravdu potřeba, výjimkou jsou např. mixiny
- vícenásobná dědičnost rozhraní je OK

Poznámky:

K mnohonásobné dědičnosti: Dědičnost rozhraní je v C++ realizována ryze abstraktními třídami, v Javě a C# pak pomocí interfaců.

Příklad: kompozice vs. dědičnost

Chceme definovat třídy `Real` a `Complex`, představující reálná a komplexní čísla – jaký má být mezi nimi vztah?

1. `Complex` dědí od `Real`
   - Myšlenka: Komplexní čísla rozšiřují reálná čísla
   - Ale: Je-li vyžadováno reálné číslo, můžeme dosadit i komplexní
   - Ale: Je-li vyžadováno komplexní číslo, nemůžeme dosadit reálné
2. `Real` dědí od `Complex`
   - Myšlenka: reálné číslo "is a" komplexní číslo
   - `Real` bude jednoduše mít nulovou komplexní složku
   - Ale: Nulová komplexní složka zabírá paměť
   - Ale: Co když budeme potřebovat `Quaternion`?

Co je správně?

Příklad: kompozice vs. dědičnost

Správně není ani jedno!

Nepoužijeme dědičnost, ale kompozici

- komplexní číslo "has a" reálné číslo, a to dvě instance
- rozhraní pro společné operace

Proč tolik povyku kolem dědičnosti?

Dědičnost má tendenci dělat věci složitější

- nadužívána pro technické vlastnosti
- porušuje zapouzdření, často sémanticky

Přemýšlejte v pojmech "is a" a "has a"

- nepoužívejte dědičnost jen kvůli ušetření kódu
  - `Properties extends Hastable` nebo `Stack extends Vector` v Javě
- dědičnost propaguje rozhraní nadtřídy (včetně nedostatků)
- kompozice vyžaduje forwarding metod, ale umožňuje definovat vlastní rozhraní

Preferujte kompozici před dědičností

- (typicky) nemá přímou jazykovou podporu – pracnější při prvním použití

Poznámky:
Viz *Effective Java: Programming Language Guide, Item 14*.

Příklad: počet přidání elementu do množiny

1. část řešení: obecná forwardovací třída

```
public class ForwardingSet <E> implements Set <E>
  private final Set <E> target;

  public ForwardingSet (Set <E> target) { this.target = target; }

  public void clear () { target.clear (); }
  public boolean contains (Object obj) { return target.contains (obj); }
  ...
  public boolean add (E element) { return target.add (element); }

  public boolean addAll (Collection <? extends E> elements) {
    return target.addAll (elements);
  }
  ...
  @Override public boolean equals (Object obj) { return target.equals (obj); }
  @Override public int hashCode () { return target.hashCode (); }
  @Override public String toString () { return target.toString (); }
}
```

Poznámky:

Viz *Effective Java, 2nd Edition: Item 17*.

Příklad: počet přidání elementu do množiny

2. část řešení: rozšíření forwardovací třídy

```
public class CountingSet  <E> extends ForwardingSet <E>
  private int addCount = 0;

  public CountingSet () {}

  public CountingSet (Set <E> target) { super (target); }

  @Override public boolean add (E element) {
    addCount++;
    return super.add (element);
  }

  @Override public boolean addAll (Collection <? extends E> elements) {
    addCount += elements.size ();
    return super.addAll (elements);
  }

  public int getAddCount {
    return addCount;
  }
}
```

Polymorfizmus

Co je to polymorfizmus?

Schopnost vystupovat v různých formách

- specificky v OOP schopnost jazyka zpracovávat objekty různými způsoby v závislosti na jejich typu

Mechanizmy pro implementaci polymorfizmu

- přetěžování metod
- dědičnost rozhraní
- dědičnost rozhraní a implementace
  - bázová třída specifikuje rozhraní
  - podtřídy v rámci něj implementují odlišné chování
  - v kódu jednotlivé podtřídy nerozlišujeme
  - technicky: virtuální metody, pozdní vazba

Polymorfizmus a příkaz `switch`

Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself. – Scott Meyers, Effective C++

Switch je promarněná příležitost k polymorfizmu

- Na každý příkaz `switch` (nebo ekvivalentní `if`) se dívejte s podezřením a přemýšlejte, zda není lepší ho nahradit polymorfismem
- Jedna z nejdůležitějších zásad OOP!
- Podstata návrhových vzorů State a Strategy

Příklad: promarněná příležitost k polymorfizmu

Explicitní změna chování podle druhu objektu

```
class Shape {
  public void drawRectangle ();
  public void drawShape ();
}

class Graphics {
  public void drawShapes (Collection <Shape> shapes) {
    for (Shape shape : shapes) {
      draw (shape);
    }
  }

  private void draw (Shape shape) {
    if (shape.kind == ShapeKind.RECTANGLE) {
      shape.drawRectangle ();
    } else if (shape.kind == ShapeKind.CIRCLE) {
      shape.drawCircle ();
    } else {
      throw new AssertionError ("unexpected shape: "+ shape.kind);
    }
  }

}
```

Příklad: promarněná příležitost k polymorfizmu

Využití polymorfizmu

```
interface Shape {
  public void draw ();
}

class Rectangle implements Shape {
  public void draw () {
    // draw rectangle
  }
}

class Circle implements Shape {
  public void draw () {
    // draw circle
  }
}

class Graphics {

  public void drawShapes (Collection <Shape> shapes) {
    for (Shape shape : shapes) {
      shape.draw ();
    }
  }
}
```

Příklad: promarněná příležitost k polymorfizmu

Využití polymorfizmu + composite patternu

```
interface Shape {
  public void draw ();
}
```

```
class Rectangle implements Shape {
  public void draw () {
    // draw rectangle
  }
}

class Circle implements Shape {
  public void draw () {
    // draw circle
  }
}

class Graphics implements Shape {
  Collection <Shape> shapes;
  public void draw () {
    for (Shape shape : shapes) {
      shape.draw ();
    }
  }
}
```

Immutability

Poznámky:
Viz *Effective Java: Programming Language Guide, Item 13*.

Definice immutability

Třída je *immutable* právě tehdy, pokud po vytvoření instance nejdou data instance žádným způsobem změnit.

- Všechna data jsou tedy zafixována v konstruktoru.

Poznámky:
Výraz "immutable" zde nepřekládám, protože neznám žádný překlad, který by nezněl divně.
Pokud vás nějaký napadne, rád o něm uslyším.

Jak vyrobit immutable třídu?

Prostředky jazyka

- všechny atributy `private` a `final` (Java)
- žádná metoda třídy nemění data
- žádné metody nejdou předefinovat v podtřídách
  - jsou `final`, nebo je `final` celá třída (Java)

Vnitřní struktura objektu

- není možné změnit obsažené objekty
  - exkluzivní přístup
  - defenzivní kopie
  - jsou rovněž immutable

Poznámky:

Obdobou javovského `final` je v C# modifikátor `sealed`, který je potřeba použít buď na třídu, nebo na metody, které byly označeny `virtual`. V případě atributů je možné použít `readonly` (na rozdíl od `final` umožňuje více zápisů). V C++ lze podobného efektu dosáhnout pomocí `const`.

Výhody immutability

Redukce počtu možných stavů objektu na jeden

- invarianty stačí ohlídat v konstruktoru

Inherentně thread-safe

- nemůže dojít ke kolizím při změnách stavu

Dobré klíče hashovacích tabulek

- `hashCode` objektu se nesmí měnit, dokud je použit jako klíč

Poznámky:

Ke klíčům hashtabulek: Jedním z požadavků na klíč hashtabulky v Javě je, aby se po dobu, co je klíčem, nezměnila hodnota, kterou vrací jeho metoda `hashCode`. Tato hodnota je v drtivé většině případů počítána z atributů objektu. V případě immutable objektů se atributy nikdy nezmění a tím pádem se nezmění ani hodnota vracená metodou `hashCode` objektu. Podmínka kladená na klíč hashtabulky je tedy automaticky splněna.

Výhody immutability

Snadné sdílení objektů

- není nutné kopírování, klidně ho i zakázat

Snadné sdílení vnitřností

- příklad: `BigInteger.negate`

Snadné cacheování

- cache nikdy nebude neaktuální

Poznámky:

K snadnému sdílení vnitřností: Třída `BigInteger` ze standardní knihovny Javy implementuje "velká čísla". Technicky jsou implementována pomocí pole. Metoda `negate` vrátí nové velké číslo, které bude negací toho, na kterém byla zavolána. Protože instance třídy `BigInteger` jsou immutable,

původní i znegované číslo můžou sdílet pole s číslicemi bez obav, že bude přepsáno. Každá instance musí mít jen svoji informaci o znaménku. Ušetří se tak paměť.

Výhody immutability

Umožňuje použít techniky funkcionálního programování

- transformace objektů na objekty
- žádné vedlejší efekty
- důsledkem opět redukce stavového prostoru

Poskytuje dobré "stavební bloky"

- složitější data složená z jednodušších
- v důsledku zjednodušuje i návrh mutable tříd

Příklad: dobré "stavební bloky"

Immutable `DateInterval` postavený z *mutable* tříd `Date`

```
class DateInterval {
  private Date begin;
  private Date end;

  // JE nutné používat defenzivní kopie.

  public Date getBegin() { return begin.clone(); }
  public Date getEnd()   { return end.clone();   }

  public DateInterval(Date begin, Date end) {
    this.begin = begin.clone();
    this.begin = end.clone();
  }
}
```

Podobně problematické použití mutable třídy `Data`

```
Date date = new Date ();
Scheduler.scheduleTask (task1, date);
date.setTime (d.getTime() + ONE_DAY);
Scheduler.scheduleTask (task2, date);
```

Příklad: dobré "stavební bloky"

Immutable `DateInterval` postavený z *immmutable* tříd `Date`

```
class DateInterval {
  private Date begin;
  private Date end;

  // NENÍ nutné používat defenzivní kopie.

  public Date getBegin() { return begin; }
  public Date getEnd()   { return end;   }

  public DateInterval(Date begin, Date end) {
    this.begin = begin;
    this.begin = end;
  }
}
```

Nevýhody immutability

S každou změnou atributu vzniká nová instance

- vadí při mnoha malých operacích za sebou
  - alokace objektů téměř nic nestojí
  - problémem je zvýšený tlak na garbage collector
- možno vyřešit dočasným použitím "mutable counterpart"
  - `String` vs. `StringBuilder`

Paradoxně mutability může vést ke stejné situaci

- `Dimension` vracená metodou `Component.getSize ()`
- použití mutable tříd ke stavbě immutable třídy

Kdy má být třída immutable?

Pokud nemáte velmi dobrý důvod, aby byla mutable

- mutable třídy by se měly měnit co nejméně
- immutabilitu zdokumentovat

```
/**
 * Class which stores information about timing of the experiments
 * in the regression analysis.
 *
 * The class is immutable and especially Misho should never ever
 * try to make it mutable :-)
 *
 * @author David Majda
 */
public class SchedulerInfo implements Serializable {
  /* ... */
}
```

Poznámky:
Příklad je převzat z našeho softwarového projektu.

Častí kandidáti na immutabilitu

Obecně malé "value objects"

- identifikátory
- data, časy
- intervaly, dvojice, trojice,...
- geometrické útvary (bod, úsečka,...)
- třídy popisující layout/strukturu něčeho (dokument, GUI,...)
- třídy vzešlé z DSL
- uzly v AST
- metadata o nějaké entitě (soubor, proces,...)

Nevhodní kandidáti na immutabilitu

- velké objekty, kontejnerové objekty
- postupně konstruované objekty