Klíčové koncepty modelování systémů II

- komplexní systémy mají následující vlastnosti:
  - **hierarchická struktura**
    - architektura systému = funkce jeho komponent a hier. vztahu mezi nimi
    - téměř dekomponovatelný systém = mezi komponentami jsou vazby slabé, ale nezanedbatelné
  - **separation of concerns**
    - spolu související části a vlasnosti jsou drženy spolu, odděleny od ostatních
    - aneb nemíchat jablka a hrušky
  - **common patterns**
    - hier. systémy obvykle sestaveny z pár typů komponent, které se opakují v různých kombinacích a uspořádáních
    - obsahují opakující se patterny, ve kterých se přepoužívají komponenty omezeného množství typů
  - **stabilní přechodové formy**
    - komplexní fungující systém – téměř jistě odvozen z jednoduššího fungujícího systému
    - komplexní systém navržený od nuly zpravidla nefunguje a rozfungovat nepůjde ani patchováním

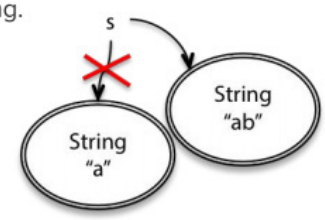
Mutabilita, immutabilita

- typy lze klasifikovat jako:
  - **mutable (měnitelné)** – např. Date (setMonth, getMonth,…)
  - **immutable (neměnitelné)** – např. String (operace vytváří nové objekty)

- mutable typy lze měnit → poskytují operace, které způsobují, že výsledky dalších operací na témže objektu vrací různé výsledky
- StringBuilder je mutable

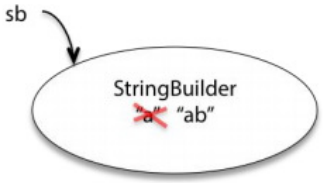
String je immutable, tedy např. při přidání znaků na konec se vždy vytváří nový String.

```
String s = "a";
s = s.concat("b"); // s+="b" and s=s+"b" also mean the same thing
```

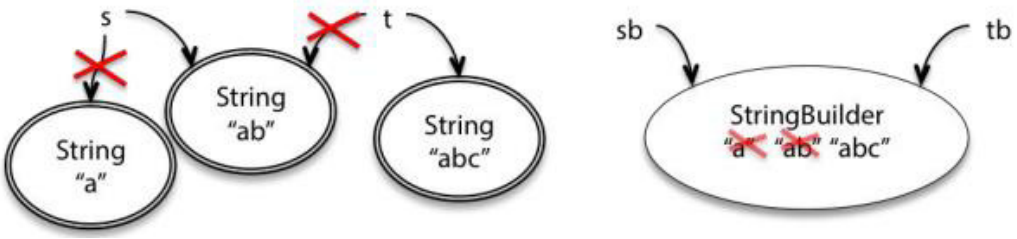


StringBuilder je mutable. Tedy i přidání znaků na konec modifikuje originální objekt.

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```



```
String t = s;
t = t + "c";
StringBuilder tb = sb;
tb.append("c");
```



Rizika mutability

- předávání mutable parametrů (sum(list), sumAbsolute(list) – modifikace původního listu)
- vracení mutable parametrů (příklad s Java Date, externí služba, cachování)

- **mutable** jsou „mocnější“ z hlediska funkcionality
  - většinou mají i lepší performance (immutable vytváří kopie)
  - např. ArrayList, Hashtable apod. (ale Collections API umí i immutable)

- doporučuje se ale používat **immutable**, protože:
  - jsou ***méně náchylné pro vznik bugů*** (absence aliasingu – odkazování z různých míst)
  - kód s IM je ***jednodušší na pochopení*** (není třeba studovat, co se děje na pozadí)
  - je ***jednodušší na upravování*** (je bezpečnější zasahovat do hotového kódu)
  - mutable objekty zesložiťují kontrakt a zhoršují reuse

Klasifikace typů a operací ADT

Operace ADT

Creators

- vytvářejí nové objekty daného typu
- mohou vzít objekt jako argument, nikoliv však stejného typu jako vytvářený

Producers

- vytvářejí nové objekty ze starých objektů daného typu
- např. operace „concat“ ve třídě String – vezme 2 Stringy, vytvoří 1 reprezentující spojení

Observers

- berou objekty ADT, vracejí objekty jiného typu
- např. operace „size“ ve třídě List – vrací int

Mutators

- mění objekty
- např. metoda „add“ třídy List – mění list přidáním elementu na jeho konec

Typy ADT

Schématicky lze zapsat následovně:

- **creator**: t* → T
  - např. konstruktor = new ArrayList(): () → ArrayList
  - nebo statická metoda jako Arrays.asList(), nebo valueOf: () → String

- **producer**: T+, t* → T
  - např. concat: String x String → String

- **observer**: T+, t* → t
  - parametrický – regionMatches: String x boolean x int x String x int → boolean
  - bezparametrický - size: List → int

- **mutator**: T+, t* → void | t | T
  - často vrací void, ale ne vždy
  - např. Component.add(): () → Component

… kde „**T**“ = abstraktní typ; každé „**t**“ = nějaký jiný typ; „**+**“ = výskyt 1 až n-krát; „*****“ = 0 až n-krát

Příklady

int = primitivní datový typ; immutable (žádné mutators)

- **creators**: čísla 0,1,2,…
- **producers**: aritmetické operátory +,-,*,/
- **observers**: porovnávací operátory ==, !=, <, >,…
- **mutators**: nemá

List = interface, mutable

- **creators**: ArrayList a LinkedList konstruktory
- **producers**: Collections.unmodifiableList
- **observers**: size, get,…
- **mutators**: add, remove, addAll, Collections.sort,…

String = třída, immutable

- **creators**: String konstruktory
- **producers**: concat, substring, toUpperCase,…
- **observers**: length, charAt,…
- **mutators**: nemá

Jak navrhovat ADT

- lepší minimální množina jednoduchých operací, které lze dobře kombinovat
- každá operace – přesně vymezený účel, 100% fungující (NE sum() v Listu)
- přidávání/odebírání metod a testování, mám-li požadované informace o objektu
- typ buď **generický** (seznam, množina,…) nebo **doménově specifický** (telefonní seznam,…)

Invarianty

- = vlastnost, která je splněna pro jakýkoliv runtime stav programu
- ve všech jeho stabilních stavech
- nezávisí na chování klienta
- měla by být zaručena pro volání public metod

- **příklady**:
  - immutabilita – jakmile je IM objekt vytvořet, drží si tu samou hodnotu
  - BST může mít invariantu, že klíč levého potomka je menší než klíč daného nodu
  - Java List – za každé situace drží pořadí prvků

- **kontrola**:
  - zadokumentováním
  - kontrolou v rámci volané metody
  - speciální metodou, kterou voláme když potřebujeme kontrolovat – checkRep()

```
class Child {
    private void checkRep() {
      assert (0 <= age < 18);
      assert (birthdayge + age == todays_date);
      assert (isLegalSSN(0 <= agsocial_security_number));
    }
}
```

```
class Account {
   public void tranfersMoney(int dstAccount, float amount) {
      // using Java assert syntax
      assert balance ≥ 0 : "balance should be ≥ 0";
      // using junit.framework.Assert (you don't have to be writing a unit test to use this class)
      Assert.assertTrue ("balance should be ≥ 0", balance ≥ 0);
    }
}
```

ADT – immutable invarianty

- správný ADT je odpovědný za zachování svých invariant
- důležitou invariantou vybraných ADT je immutabilita

```
public class Tweet {
   public String author;
   public String text;
   public Date timestamp;

   public Tweet(String author, String text, Date timestamp) {
       this.author = author;
       this.text = text;
       this.timestamp = timestamp;
   }
}
```

Jak zajistit, že jednotlivé tweety jsou immutable?

- 1. problém – objekty mohou přímo modifikovat proměnné třídy (**representation exposure**)
- to kromě invariant ohrožuje i **representation independence** (nezávislost na reprezentaci)

```
public class Tweet {
   private final String author;
   private final String text;
   private final Date timestamp;
   public Tweet(String author, String text, Date timestamp) {
       this.author = author;
       this.text = text;
       this.timestamp = timestamp;
   }
   public String getAuthor() {
       return author;}
   public String getText() {
       return text; }
   public Date getTimestamp() {
       return timestamp; }
}
```

1.) schování proměnných za getter/setter operace
2.) modifikátor final → ani implementátor třídy nemůže upravit stav objektu

---

Vývojář ale může napsat:

```
/** @return a tweet that retweets t, one hour later*/
public static Tweet retweetLater(Tweet t) {
    Date d = t.getTimestamp();
    d.setHours(d.getHours()+1);
    return new Tweet("rbmllr", t.getText(), d);
}
```

… z Tweet vrátil mutable objekt, který modifikoval ve vnějším kódu.

Tomu lze zabránit pomocí tzv. defensive copying.

```
public Date getTimestamp() {
    return new Date(timestamp.getTime());
}
```

Vývojář může ale zase napsat:

```
/** @return a list of 24 inspiring tweets, one per hour*/
public static List<Tweet> tweetEveryHourToday () {
    List<Tweet> list = new ArrayList<Tweet>();
    Date date = new Date();
    for (int i = 0; i < 24; i++) {
       date.setHours(i);
       list.add(new Tweet("John", "You made it!", date));
    }
    return list;
}
```

… čímž všechny instance Tweetu odkazují na ten samý Date objekt.

Problém lze vyřešit kopií objektu Date v konstruktoru.

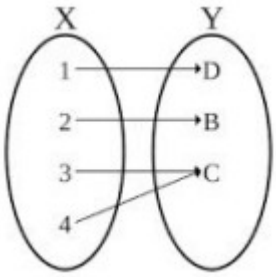
```
public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = new Date(timestamp.getTime());
}
```

- Pokud vše selže, popř. nechceme dělat kopie objektů – nutné alespoň zapsat ve specifikaci!

ADT – Rep(rezentační) invarianty

- na ADT se lze dívat jako na vztah dvou prostorů hodnot
- prostor abstraktních hodnot **„A“** a prostor reprezentací **„R“**
- abstraktní prostor je to, co by měl ADT implementovat a vystavit

- **abstraktní funkce**
    - = surjektivní mapování hodnoty z R do hodnot A (abstr. hodnot, které reprezentují)
    - AF: R → A
    - surjekce = „na“ zobrazení = na celou množinu = každý prvek cílové množiny má alespoň jeden vzor



- **rep invarianty**
    - = mapuje hodnoty z R na boolean (vlastnosti, které se drží a které ne)
    - RI: R → boolean

Příklady

Komplexní čísla

- R = množina objektů třídy Complex
- A = množina komplexních čísel

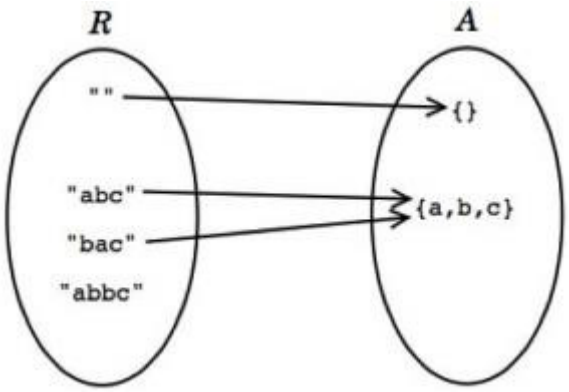
```
class Complex {
  private double real;
  private double imag;
 // The abstraction function is
 //   real + i * imag
}
```

Implementace ADT CharSet

- pro vnitřní reprezentaci použijeme String
- pak R obsahuje Stringy a A je matematická reprezentace množiny znaků
- každá hodnota z A je mapovaná na nějakou hodnotu z R
    - musíme být schopni vytvářet a manipulovat s veškerými možnými hodnotami A
    - a zároveň být schopni je reprezentovat
- některé abs. hodnoty jsou mapovány na více reprezentačních hodnot

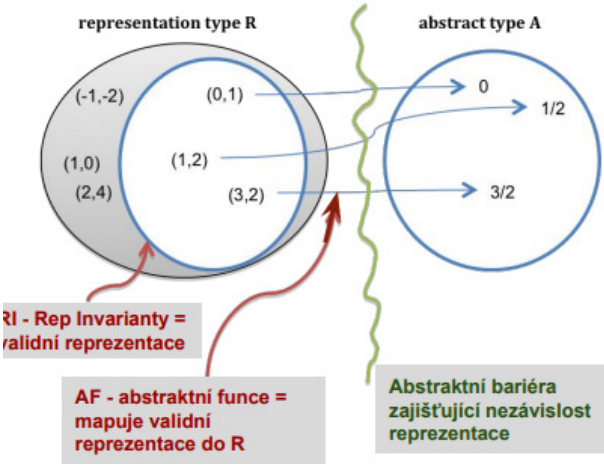
- ne všechny rep hodnoty jsou mapované
    - pokud string nemá duplicity, můžeme např. ukončit remove(x) po 1. instanci

```
public class CharSet {
   private String s;
   ...
}
```



Příklad ADT pro reprezentaci racionálních čísel =>

- Páry jako např. *(2,4)* a *(18,12)* nepatří do RI prostoru (mezi Rep Invarianty), protože nejsou v prvočíselném rozkladu jak je vyžadováno (**numer/denom is in reduced form)**




```
public class RatNum {
    private final int numer;
    private final int denom;
    // Rep invariant:
    //   denom > 0
    //   numer/denom is in reduced form
    // Abstraction Function:
    //   represents the rational number numer / denom
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }
    public RatNum(int n, int d) throws ArithmeticException{
        // reduce ratio to lowest terms
        int g = gcd(n, d);
        n = n / g;
        d = d / g;
        // make denominator positive
        if (d < 0) {
            numer = -n;
            denom = -d;
        } else {
            numer = n;
            denom = d;
        }
        checkRep();    }
}
```

Shrnutí

- **invarianta** = vlastnost, která je vždy splněna na dané instanci ADT po celý jeho lifecycle
- dobrý ADT si drží své vlastní invarianty
- Creators/Producers nejdříve inicializují invarianty a Observers/Mutators je pak drží

- **Rep invarianty** specifikují validní hodnoty reprezentace; jsou kontrolovány v runtime speciální metodou checkRep(), která +- nahrazuje preconditions
- abstraktní funkce mapují konkrétní reprezentace na abstraktní hodnoty, které reprezentují

- **representation exposure** je ohrožením pro nezávislost reprezentace (**r. independence**) a zachování invariant (**inveriant preservation**)

- ADT rep invarianty = kuchařka, jak vytvářet abstraktní datové typy:
    - aby se minimalizovaly možnosti vzniku chyb
    - kód byl srozumitelný
    - kód byl připravený na změny