Modularita

Karel Richta a kol.

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Objektové modelování, xxB36OMO
10/2016, Modularita

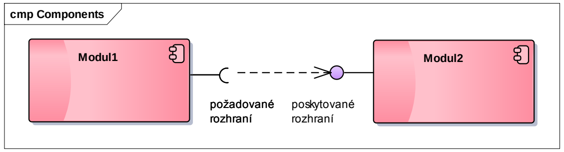
https://cw.fel.cvut.cz/wiki/courses/xxb36omo/start

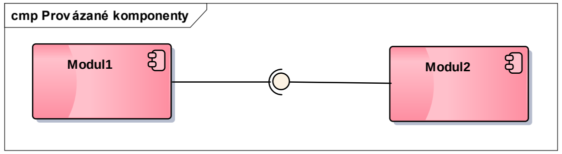
Modularita

- Větší program nelze zpravidla realizovat jako monolitický kus kódu.
- Bylo by to příliš nákladné a často by se opakovalo řešení stejných podproblémů.
- Šikovnější je složit řešení z komponent – modulů.
- Moduly mohou být připraveny předem, nebo být součástí řešení.
- Moduly spolu komunikují (jinak by nemělo smysl je do řešení zahrnovat).
- Na druhé straně by jednotlivé moduly měly být na sobě co nejméně závislé - závislost pouze pomocí rozhraní – „**low coupling**“ (nízká provázanost).
- Samotný modul by měl naopak zahrnovat vše potřebné pro realizaci služeb – „**high cohesion**“ (vysoká soudržnost).

Modularita

- Umožňuje znovupoužívání kódu.
- Lze ji vnímat v různých rovinách abstrakce (metody, třídy, balíčky).
- Pokud je navržený program modulární, pak je obvykle dobře spravovatelný.
- Umožňuje zaměnitelnost jednotlivých “modulů”.
- Výhodou je, že můžeme mít jednu architekturu, která pouhou výměnou jednotlivých modulů může plnit jinou funkci.
- Zpřehledňuje a snižuje množství kódu.
- Závislost mezi moduly je dána počtem použitých rozhraní, jejich složitostí a složitostí informace přenášené přes rozhraní.



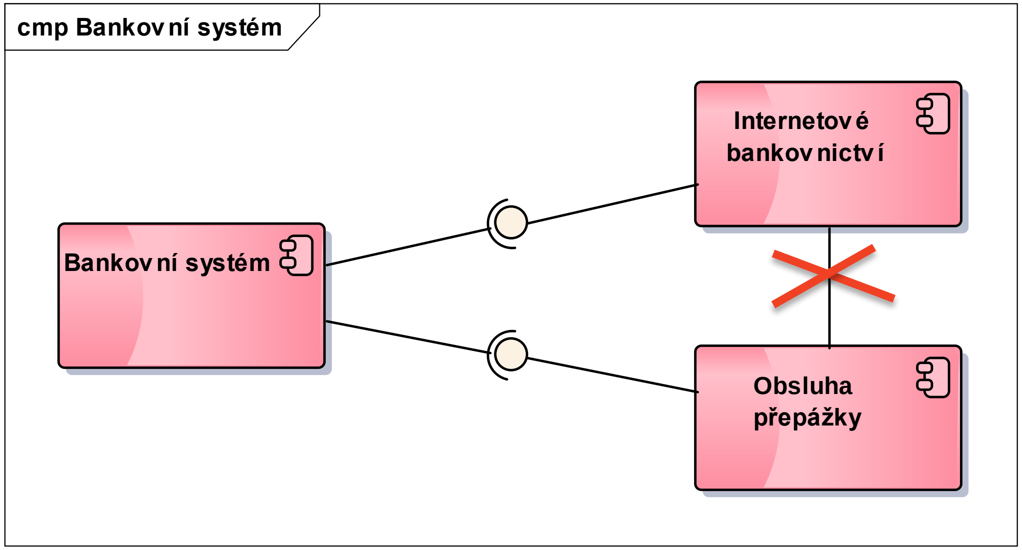


Deméteřin zákon (Law of Demeter)

bohyně plodnosti (Demeter = matka)

Pravidlo, které nám říká následující:

- Každá jednotka by měla mít jen omezené znalosti o dalších jednotkách: pouze jednotky úzce spjaté s touto jednotkou.
- Každá jednotka by měla mluvit pouze se svými přáteli a nemluvit s cizími jednotkami.
- Mluvit jen s bezprostředními přáteli.
- Minimalizuje provázanost jednotlivých modulů.



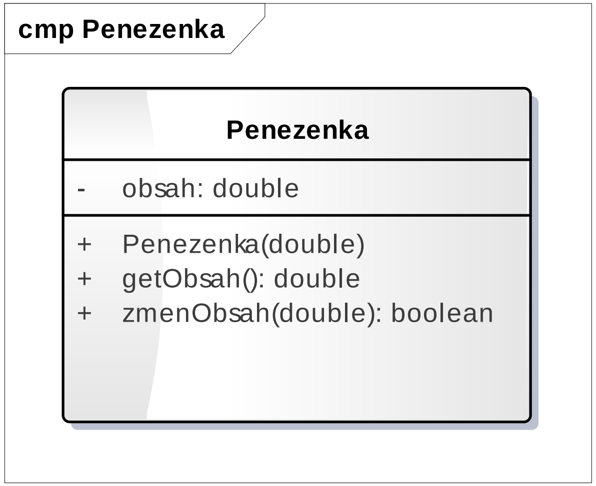
Aplikace Deméteřina zákona na OOP

Z hlediska OOP můžeme toto pravidlo aplikovat pomocí následujících 5-ti pravidel s jakými objekty může pracovat metoda M zaslaná objektu O:

- s touto instancí (this),
- s argumenty metody M,
- s objekty, které metoda vytvoří,
- s komponentami, které jsou součástí O,
- s globálními objekty, které jsou metodě v rámci O přístupné.

Příklad: Práce s peněženkou

```
class Penezenka {
    private double obsah;
    public double getObsah() { return obsah; };
    public void zmenObsah(double kolik) { obsah += kolik; };
    public Penezenka(double pocatek) { obsah = pocatek; }
};
```



Příklad: Práce s objektem samotným (this)

```
class Finance {
    double penize;
    private Penezenka mojePenezenka;
    boolean platPenezenkou(double platba) {
        if (this.mojePenezenka.getObsah() < platba) return false;
        this.mojePenezenka.zmenObsah(-platba);
        return true;
    };
    public void platbaHotovost(double kolik) {
        if (platPenezenkou(kolik)) System.out.println("OK");
        else System.out.println("Chyba, platbu nelze provest");
    };
};
```

Příklad: Práce s argumenty metody

```
class Finance {
    private Penezenka mojePenezenka;
    boolean zmenPenezenku(Penezenka pen, double platba) {
        if (pen.getObsah() < platba) return false;
        pen.zmenObsah(-platba);
        return true;
    };
    public void platbaHotovost(double kolik) {
        if (zmenPenezenku(mojePenezenka,kolik))
                System.out.println("OK");
        else System.out.println("Chyba, platbu nelze provest");
    };
};
```

Příklad: Práce s komponentami objektu

```
public class Penezenka {
    private double obsah;
    private List<Transakce> historie;
    public double getObsah() { return obsah; };
    public List<Transakce> getHistorie() { return historie; };
    public void zmenObsah(double kolik) { obsah += kolik; };
    public Penezenka(double pocatek) { obsah = pocatek; }
};

class Transakce {
    private Date kdy;
    private double castka;
    public Transakce(Date k, double p) { kdy = k; castka = p; };
};
```

Příklad: Práce s komponentami objektu (pokr.)

```
class Finance {
    double penize;
    private Penezenka mojePenezenka;
    boolean platPenezenkou(double platba) {
        if (this.mojePenezenka.getObsah() < platba) return false;
        this.mojePenezenka.zmenObsah(-platba);
        Transakce e = new Transakce(new Date(),-platba);
        this.mojePenezenka.getHistorie().add(e);
        return true;
    };
    public void platbaHotovost(double kolik) {
        if (zmenPenezenku(mojePenezenka,kolik))
             System.out.println("OK");
        else System.out.println("Chyba, platbu nelze provest");
    };
};
```

Příklad: Práce s globálními objekty

```
class Finance {
    public static double hotovost = 1000; // pod matraci
    private Penezenka mojePenezenka;
    boolean zmenPenezenku(Penezenka pen, double platba) {
        if (pen.getObsah() < platba) return false;
        pen.zmenObsah(-platba);
        return true;
    };
    public void platbaHotovost(double kolik) {
        if (zmenPenezenku(mojePenezenka,kolik))
            System.out.println("OK");
        else if (hotovost >= kolik) { hotovost -= kolik;
            System.out.println("OK, ale placeno z hotovosti");
        }
        else System.out.println("Chyba, platbu nelze provest");
    };
};
```

Nástroj pro správnou definici modulů - ADT

- ADT = **Abstract Data Type**
- ADT má specifikaci: vyjadřuje abstraktní vlastnosti modulu, nezabývá se reprezentací a implementací.
- ADT není jen struktura dat:
  - datová struktura + sada konvencí
- Sada konvencí – invarianty reprezentace.
  - Invarianty indikují, zda je datová struktura dobře vytvořena – definují množinu validních stavů datové struktury.
- Abstrahující funkce: převod konkrétní reprezentace do abstraktních hodnot.
  - Říká, jaký je význam datové struktury (abstraktní hodnota), jak má být datová struktura interpretována, jak počítat inverzi, zobrazení z abstraktních hodnot do objektů.

ADT je určen specifikací

- Specifikace ADT je kolekce procedurálních abstrakcí (nikoliv jen kolekce procedur).
- Procedurální abstrakce poskytují:
  - definici množiny hodnot,
  - všechny způsoby, jak lze množinu hodnot používat.
- Abstrakce pro vytváření (creators) – vytvářejí nové hodnoty.
- Abstrakce pro manipulaci (manipulators) – mění hodnoty.
- Abstrakce pro prohlížení (observers) – umožňují prohlížení hodnot.

Implementace ADT

- Implementaci ADT (např v Javě) zajišťujeme pomocí tříd (classes).
- Při implementaci ADT:
  - volíme reprezentaci instancí,
  - implementujeme operace nad touto reprezentací.
- Při volbě reprezentace respektujeme:
  - aby bylo možné implementovat operace,
  - aby nejčastěji používané operace byly nejvíce efektivní.
- Které to budou?
- To sice nevíme, ale abstrakce nám umožňuje tyto skutečnosti změnit.

Zapouzdření

- Umožňuje nám řídit přístupová práva k metodám, třídám a objektům.
- Je založeno na předpokladu, že správné fungování tříd je založeno na řadě invariantů (vlastnostech, které platí po celou dobu existence objektu).
- Programátor, který třídu používá nemůže tyto invarianty všechny znát.
- Z tohoto důvodu je dobré programátorovi zamezit v přístupu k členským proměnným a metodám, které by mohly nesprávným použitím tyto invarianty porušit.
- Zapouzdřením se snažíme zabránit problému zpřístupnění implementace (Representation Exposure).

Příklad : CharSet - Specifikace

// Overview: A CharSet is a finite mutable set of Characters

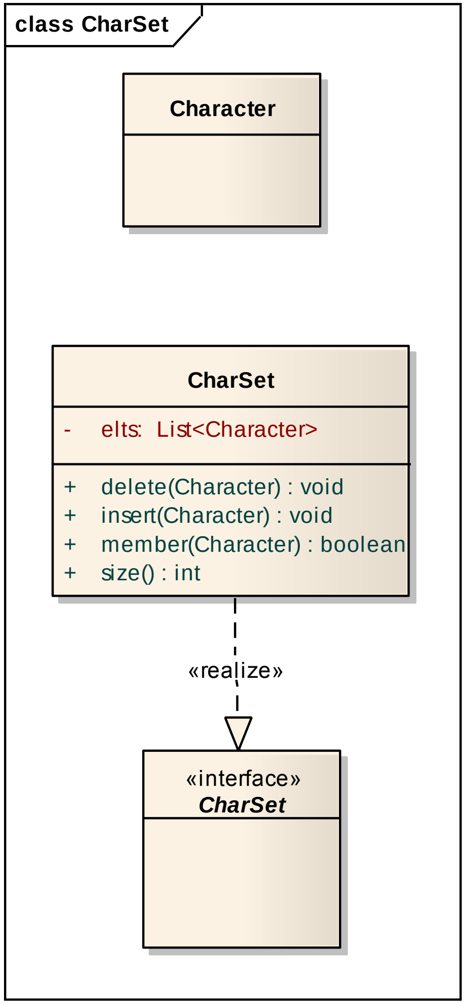
// effects: creates a fresh, empty CharSet
public CharSet ( )

// modifies: this
// effects: $this_{post} = this_{pre} \cup \{c\}$
public void insert (Character c);

// modifies: this
// effects: $this_{post} = this_{pre} - \{c\}$
public void delete (Character c);

// returns: $(c \in this)$
public boolean member (Character c);

// returns: cardinality of this
public int size ( );



Implementace (jedna možná)

```
class CharSet {
    private List<Character> elts = new ArrayList<Character>();
    public void CharSet(){};
    public void insert(Character c) { elts.add(c); }
    public void delete(Character c) { elts.remove(c); }
    public boolean member(Character c) { return elts.contains(c); }
    public int size() { return elts.size(); }
};
```

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.insert(a);
s.delete(a);
if (s.member(a)) System.out.println("Chyba");
else System.out.println("OK");
```

Kde je chyba?

- Může být špatně metoda delete.
  - Měla by odstranit všechny výskyty znaku.
- Může být špatně metoda insert!
  - Neměla by vložit znak pokud již v Listu je (množina může jeden prvek obsahovat nejvýše jednou, jinak to není množina, ale tzv. multi-množina, angl. bag).
- Měli bychom se to dozvědět z invariantu reprezentace datového typu, který by měl být součástí specifikace ADT.

Invarianty reprezentace (Representation Invariants)

- Specifikují vlastnosti stavu ve kterém platí, že je datová struktura dobře utvořená.
- Zachycují informaci, která musí platit napříč všemi metodami.
- Můžeme ho specifikovat např. takto:

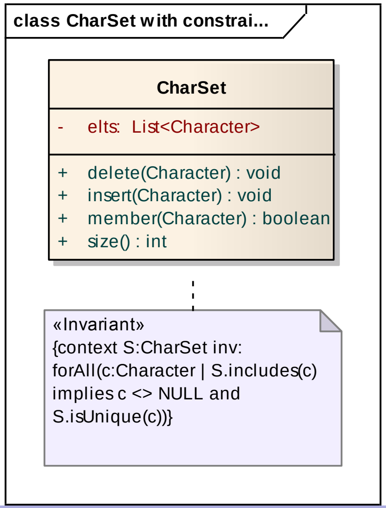
```
class CharSet {
  // Rep invariant: elts has no nulls and no duplicates
  private List<Character> elts;
  ...
```

- Nebo formálněji:

$\forall$ indices i of elts . elts.elementAt(i) $\neq$ null

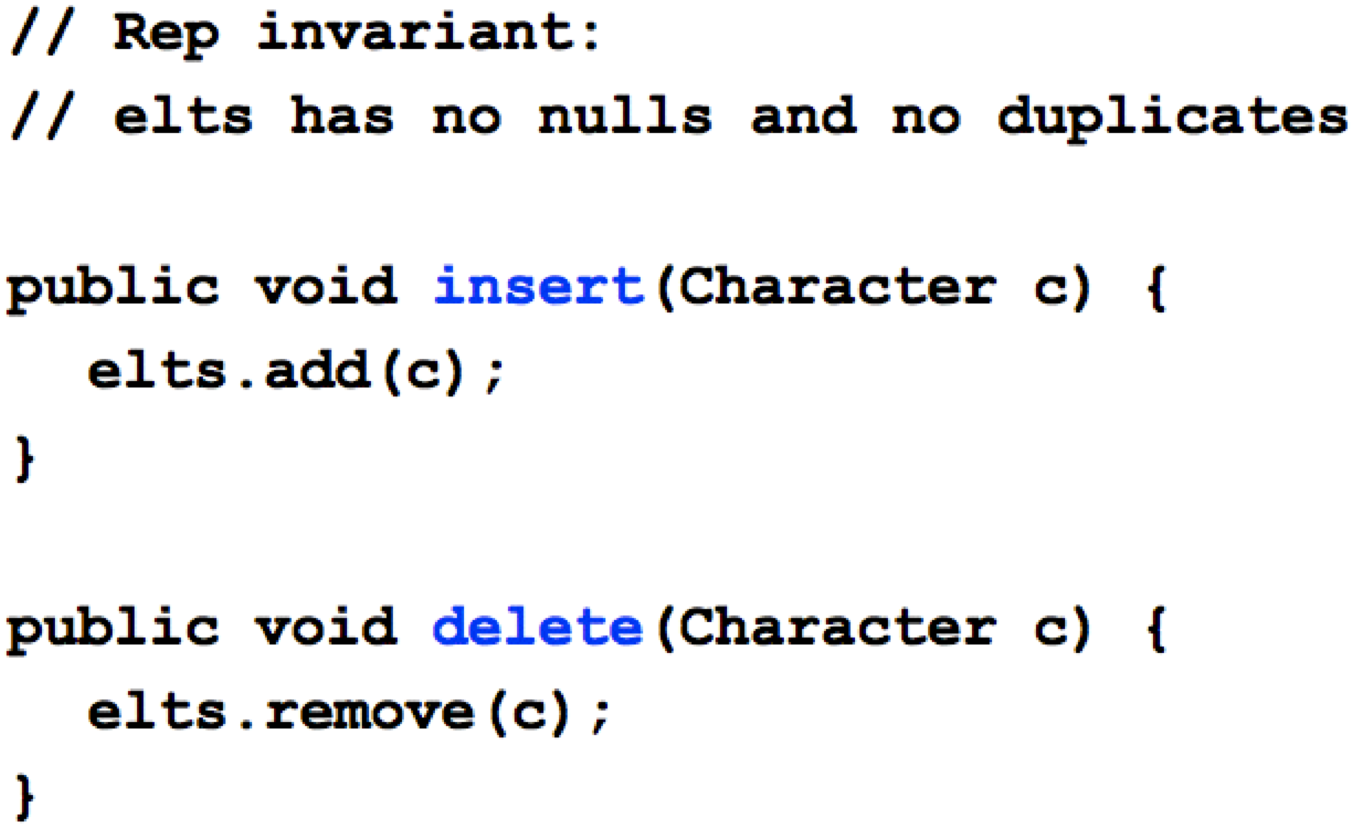
$\forall$ indices i, j of elts .

$\quad$ i $\neq$ j $\Rightarrow \neg$ elts.elementAt(i).equals(elts.elementAt(j))



```
«Invariant»
{context S:CharSet inv:
forAll(c:Character | S.includes(c)
implies c <> NULL and
S.isUnique(c))}
```

Nyní můžeme nalézt chybu



```
// Rep invariant:
// elts has no nulls and no duplicates

public void insert(Character c) {
   elts.add(c);
}

public void delete(Character c) {
   elts.remove(c);
}
```

Další ukázka invariantu

```
class Account {
    private int balance;
    // history of all transactions
    private List<Transaction> transactions;
    …
}
```

// real-world constraints:

$balance \geq 0$

$balance = \Sigma_i$ transactions.get(i).amount

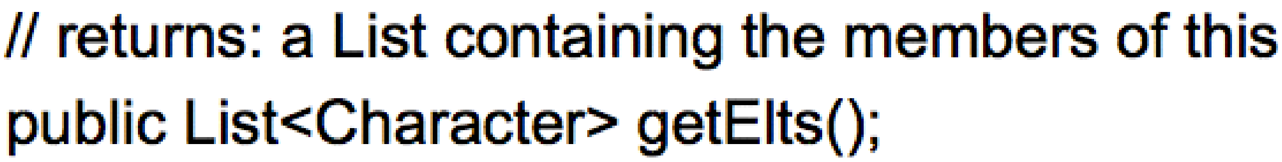
// implementation-related constraints:

transactions $\neq$ null

no nulls in transactions

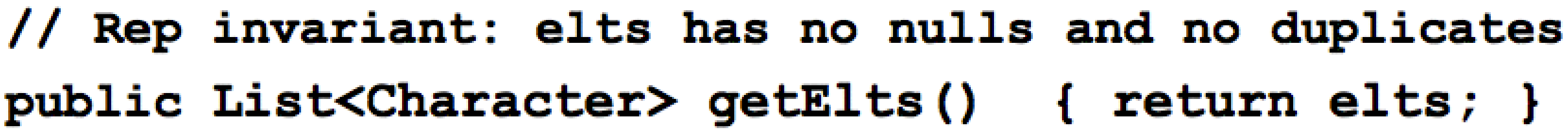
CharSet pokračování

- Uvažujme přidání následující metody do třídy CharSet!



```
// returns: a List containing the members of this
public List<Character> getElts();
```

- S následující implementací!

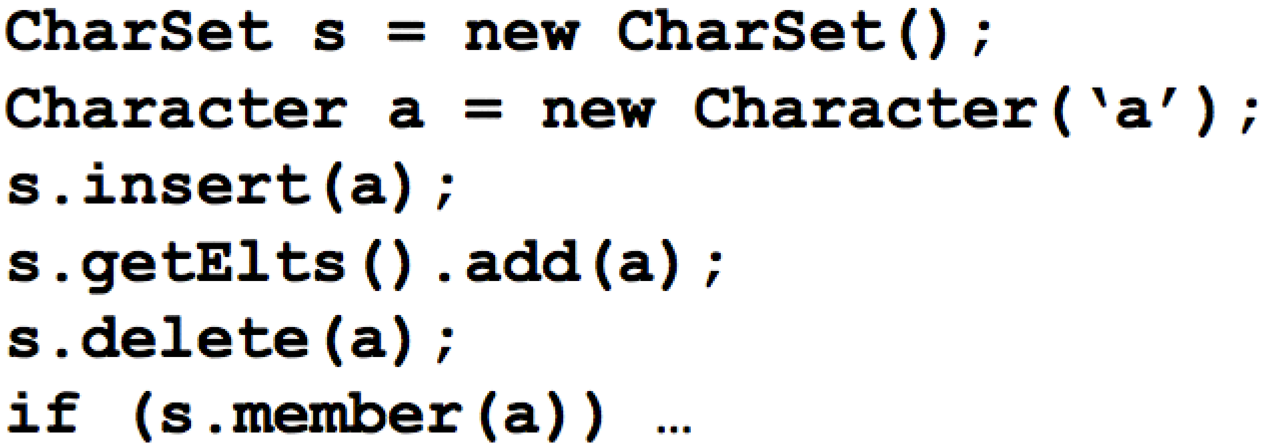


```
// Rep invariant: elts has no nulls and no duplicates
public List<Character> getElts()  { return elts; }
```

- Dodržuje implementace getElts() invariant reprezentace?

Representation Exposure

- Uvažujme následující kód:

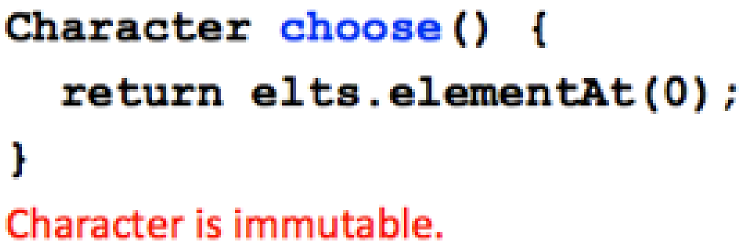


```
CharSet s = new CharSet();
Character a = new Character(‘a’);
s.insert(a);
s.getElts().add(a);
s.delete(a);
if (s.member(a)) …
```

- Zpřístupnění implementace je tedy externí přístup k členské proměnné.
- Zpřístupnění implementace je téměř vždy problém (může dojít k porušení invariantů)! Snažíme se mu vyhnout.

Způsoby jak zabránit Representation Exposure

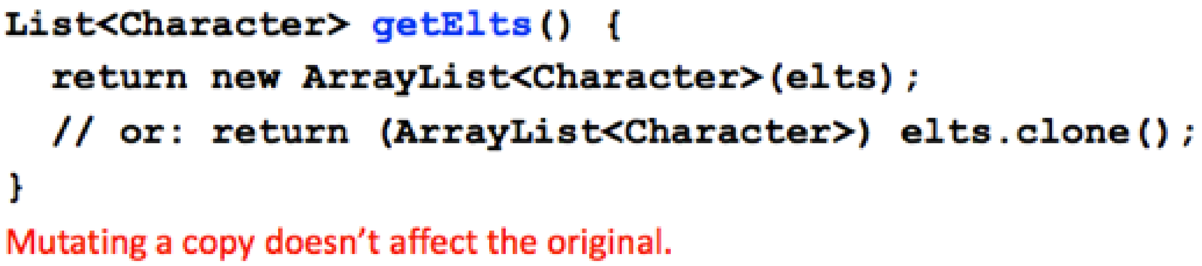
1. Využít immutabilitu



```
Character choose() {
  return elts.elementAt(0);
}
```

Character is immutable.

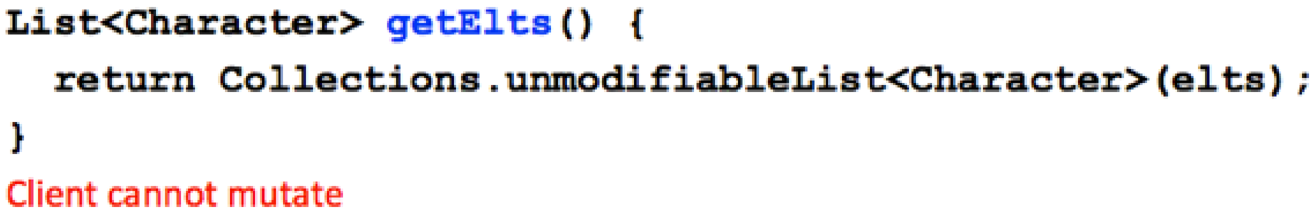
2. Vytvořit kopii



```
List<Character> getElts() {
  return new ArrayList<Character>(elts);
  // or: return (ArrayList<Character>) elts.clone();
}
```

Mutating a copy doesn't affect the original.

3. Vytvořit immutable kopii



```
List<Character> getElts() {
  return Collections.unmodifiableList<Character>(elts);
}
```

Client cannot mutate

Je třeba invarianty reprezentace testovat?

- Určitě, pokud to není nákladné.
- Ale i když to nákladné je, při ladění je to nutné.
- Může být těžké testování za běhu vypnout.
- Některé privátní metody není nutno testovat (proč?).

Testování invariantů reprezentace

- Standardní pravidlo – testuj při vstupu a při výstupu (proč?).

```
/* testuje jedinecnost reprezentace - invariant */
private void CheckRep() {
    for (int i = 0; i < elts.size(); i++) {
         assert elts.indexOf(elts.elementAt(i) == i);
    }
}

/* delete s testy */
public void delete(Character c) {
    CheckRep();
    elts.remove(c);
    CheckRep();
}
```

Praktické programování s ochranou

- Předpokládejte, že budete dělat chyby.
- Vytvořte a používejte kód pro jejich zachycení.
- Na vstupu:
  - Testujte invarianty reprezentace.
  - Testujte „preconditions“ (klazule „requires“).
- Při výstupu:
  - Testujte invarianty reprezentace.
  - Testujte „postconditions“ (klazule „provides“).
- Testování invariantů reprezentace pomůže při hledání chyb.
- Porozumění invariantům reprezentace pomůže vyhnout se chybám.
- Nebo dokažte, že chyby nemohou nastat.

Invarianty reprezentace omezují strukturu, nikoliv význam

Nová implementace metody insert, která zajišťuje invariant:

```
/* nova implementace insert */
public void insert(Character c) {
    Character cc = new Character (encrypt(c));
    if (!elts.contains(cc)) elts.addElement(c);
}
```

Program je ale stále špatně:

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.insert(a);
s.delete(a);
if (s.member(a)) System.out.println("Chyba");
else System.out.println("OK");
```

Abstrahující funkce: reprezentace → význam

- Abstrahující funkce mapuje konkrétní reprezentaci na abstraktní hodnotu, kterou reprezentuje.
  - AF: Object → abstract value
  - AF(CharSet this) = { c | c je obsažen v this.elts }
  - “množina znaků, obsažená v this.elts”
- Abstrahující funkce typicky není vykonatelná.
- Abstrahující funkce nám umožňuje pochopit význam z hlediska uživatele.

Abstrahující funkce: reprezentace → význam

- Náš cíl je splnit specifikaci metody insert:

  // modifies: this

  // effects: $\text{this}_{post} = \text{this}_{pre}$ U {c}

  public void insert (Character c);

- Musíme zkoumat, kde by mohla vzniknut chyba.
- Aplikace abstrahující funkce na výsledek volání metody insert dává:

  AF(elts) U {encrypt(‘a’)}

- Co když použijeme následující abstrahující funkci?

  AF(this) = { c | encrypt(c) je obsaženo v this.elts }

  AF(this) = { decrypt(c) | c je obsaženo v this.elts }

Řešení abstrahující funkce

- Náš základní cíl je splnit specifikaci metody insert:

  // modifies: this

  // effects: this_{post} = this_{pre} U {c}

  public void insert (Character c);

- AF nám vysvětluje význam:

  AF(CharSet this) = { c | c je obsaženo v this.elts }

- Uvažme volání metody insert:

  Na vstupu je význam AF(this_{pre}) ≈ elts_{pre}

  Na výstupu je význam AF(this_{post}) = AF(this_{pre}) U {encrypt(‘a’)}

- Pak můžeme použít abstrahující funkci:

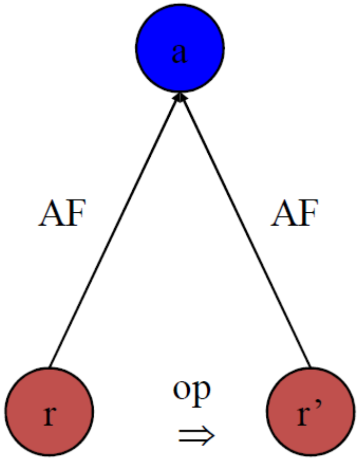
  AF(this) = { c | encrypt(c) je obsaženo v this.elts }

  AF(this) = { decrypt(c) | c je obsaženo v this.elts }

Povolené vedlejší efekty

- Jiná implementace metody member:

```
boolean member(Character c1) {
    int i = elts.indexOf(c1);
    if (i == -1) return false;
    // optimalizace
    Character c2 = elts.elementAt(0);
    elts.set(0, c1);
    elts.set(i, c2);
    return true;
}
```



- Optimalizace (move-to-front) urychluje test na členství.
- Modifikuje reprezentaci, ale nemění abstraktní hodnotu.

Abstrahující funkce je funkce

Otázka: Proč AF mapuje konkrétní objekty na abstraktní a ne obráceně?

- Obrácené zobrazení nemusí být funkce.
- A nedá se zpravidla použít vždy.

Jak psát abstrahující funkce

- Doménou jsou všechny reprezentace, které splňují invarianty.
- Obor hodnot: může být někdy obtížné stanovit.
- Pro matematické entity (jako množiny) je to jednoduché.
- Pro složitější abstrakce: rozdělte specifikaci na části (fields).
- AF definuje hodnotu pro každou takovou část.
- Přehledová část specifikace by měla poskytovat návod na definici abstraktních hodnot.
- Tištěný výstup se hodí pro ladění.

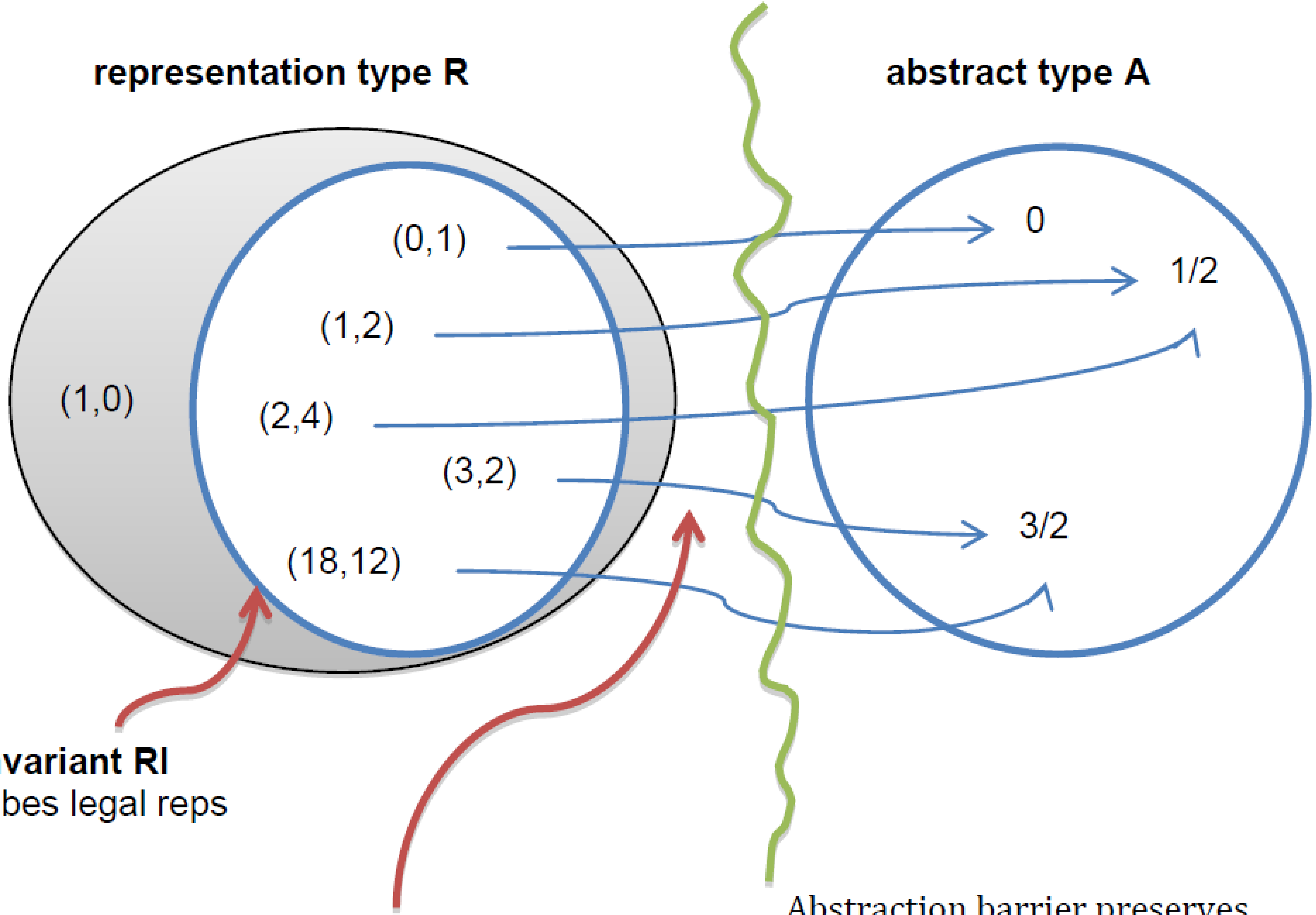
Příklad: ADT pro racionální čísla

```
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
 /** Make a new Ratnum == n. */
    public RatNum(int n) {
       numer = n;
       denom = 1;
       checkRep();
    }
```

Příklad: ADT pro racionální čísla (pokr.)

```
/** Make a new RatNum == (n / d).
 * @throws ArithmeticException if d == 0 */
    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();
    }
}
```

Reprezentace versus abstrakce



rep invariant RI
describes legal reps

abstraction function AF
maps legal reps to abstract values

Abstraction barrier preserves
representation independence

Příklad: test pro racionální čísla (pokr.)

```
// Check that the rep invariant is true
// Warning: this does nothing unless you turn on assertion
// checking by passing -enableassertions to Java
    private void checkRep() {
       assert denom > 0;
       assert gcd(numer, denom) == 1;
    }
// Return a string representation of this rational number
// This effectively implements the abstraction function
    public String toString() {
        return (denom > 1) ? (numer + "/" + denom) : (numer + "");
    }
```

Shrnutí

Invarianty reprezentace:

- Které konkrétní hodnoty reprezentují jaké abstraktní hodnoty.

Abstrahující funkce:

- Pro každou konkrétní hodnotu určí jakou konkrétní hodnotu reprezentuje.

Dohromady to zajišťuje modularizaci implementace:

- Můžete zkoušet jednotlivé operátory.
- Žádný není součástí abstrakce (ADT)

V praxi:

- Vždy definujte invarianty reprezentace.
- Pokud je to zapotřebí, definujte abstrahující funkci.
- Pro triviální případy ji udělejte jen informativní, formální definice je mnohem težší.

Literatura (zdroj)

- http://courses.cs.washington.edu/courses/cse331/11wi/lectures/lect05-af-ri.pdf

The End