Programovací jazyky a paradigmata, SWIG a práce se starším kódem

Michal Wiglasz

Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2, 612 66 Brno - Královo Pole
iwiglasz@fit.vutbr.cz

25. 4. 2017
Praktické aspekty vývoje software (IVS)

JAZYKY A PARADIGMATA

Jazyk ovlivňuje co a jak můžeme vyjádřit.

Možná ovlivňuje i naše myšlení

(Sapir-Whorfova hypotéza)

Široké spektrum programovacích jazyků

- různé perspektivy, vlastnosti, podporované konstrukce a různé způsoby zápisu.
- Wikipedia: List of programming languages → cca 700

Jaký jazyk zvolit?

Ideálně dle řešeného problému

Člověk vs. počítač

Na úrovni HW velmi komplikovaný proces a velmi primitivní prostředky – instrukce

Pro člověka je přirozenější pracovat na vyšší úrovni.

Některé jazyky více reflektují povahu počítače, jiné více způsob uvažování člověka.

Efektivita vývoje vs. výsledku

Je třeba zvážit, co je důležitější:

- rychlejší program
- rychlejší / pohodlnější vývoj

Vhodný jazyk či prostředí může výrazně zvýšit efektivitu vývoje.

Mnohdy je lepší srozumitelnější a přehlednější program, i když je o trochu pomalejší.

Někdy je levnější koupit výkonnější HW, než zaplatit více za vývojáře.

Nízkoúrovňové vs. vysokoúrovňové jazyky

Nízkoúrovňové

- strojový kód, assembler
- velmi malá abstrakce od HW
- snadný překlad do instrukcí pro procesor
- mohou být rychlejší a méně náročné na prostředky
- složitější vývoj

Vysokoúrovňové

- větší míra abstrakce
- srozumitelnější, jednodušší vývoj → méně chyb
- přenositelné mezi platformami

Imperativní vs. deklarativní

Imperativní

- specifikace, jak se problém řeší (kuchařka)
- sekvence kroků, které mění stav programu a tím provádějí výpočet

Deklarativní

- specifikace, co se má řešit
- popis problému vhodnými konstrukty
- řešení najde vyhodnocovací mechanismus
- funkcionální a logické paradigma, SQL, …

Statické vs. dynamické

Statické

- co se bude dít, se rozhoduje při překladu
- je obtížné zkoumat a měnit stav programu
- nelze za běhu měnit a přidávat funkce, objekty či typy
- optimalizace při překladu
- edit → compile → run → debug

Dynamické

- co se bude dít, se rozhoduje za běhu
- je jednoduché zkoumat, rozšiřovat, manipulovat
  - monkey patching
- optimalizace při běhu programu

Silně vs. slabě typované

Silně typované

- každá proměnná či operace má striktně určený datový typ, který se nemění

Slabě typované

- automaticky přetypuje hodnoty dle potřeby

Dynamické typování ≠ slabé typování (a naopak)

Automatická vs. manuální správa paměti

Automatická

- programátor paměť jen alokuje
- nepoužívanou paměť uvolňuje *garbage collector*
  - počítání referencí – neuvolní cykly
  - sledovací algoritmy – přeruší běh programu

Manuální

- programátor paměť alokuje i uvolňuje
- obtížné ve složitějších programech – *memory leaks*

Paradigmata

- Imperativní
- Objektově orientované
- Funkcionální
- Logické
- Konkurentní
- Metaprogramování
- …

Imperativní

- C, Pascal, Java, Python…
- Základní prostředky:
  - cykly
  - větvení
  - ukazatele
  - struktury
  - funkce
- Explicitní stav programu měněn sekvencí příkazů
- Reflektuje architekturu počítačů
- Nadstavby: procedurální, strukturované, modulární

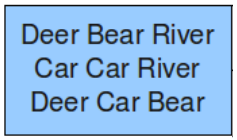
Funkcionální

- Haskell, Lisp, Clojure, F#, Scala…
- Základ v lambda kalkulu
- Základní prostředky:
  - funkce (v matematickém smyslu)
- Specifikace problému v podobě funkcí
- Data proplouvají funkcemi, které je transformují
- Žádná explicitní manipulace s daty
- Žádné vedlejší efekty (kromě I/O)
- Vyhodnocování v libovolném pořadí
- Lze snáze ukázat korektnost programu

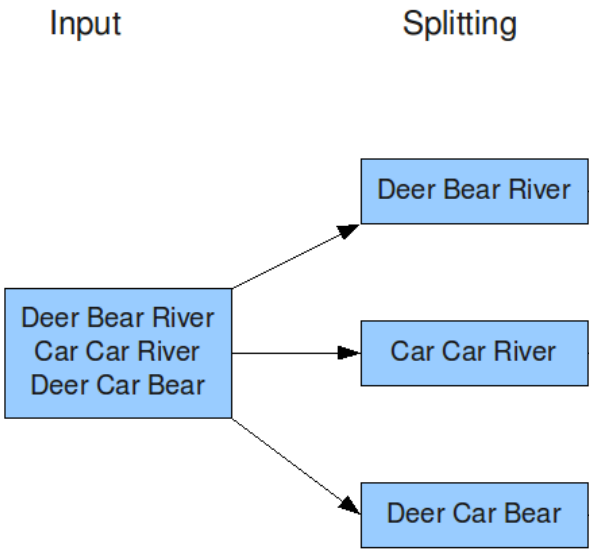
- Metoda pro paralelizaci výpočtu (i pro big data)
- Inspirace funkcionálními jazyky
- Založeno na funkcích *map* a *reduce*
  - *map(in_key, in_value) → list(out_key, intermediate_value)*
  - *reduce(out_key, list(intermediate_value)) → list(out_value)*

1. vstupní data jsou rozdělena mezi výpočetní uzly
2. každý uzel provede *map*
3. mezivýsledky jsou rozděleny mezi uzly dle klíčů *out_key*
4. každý uzel provede *reduce*
5. výsledky se sebírají a uloží na výstup

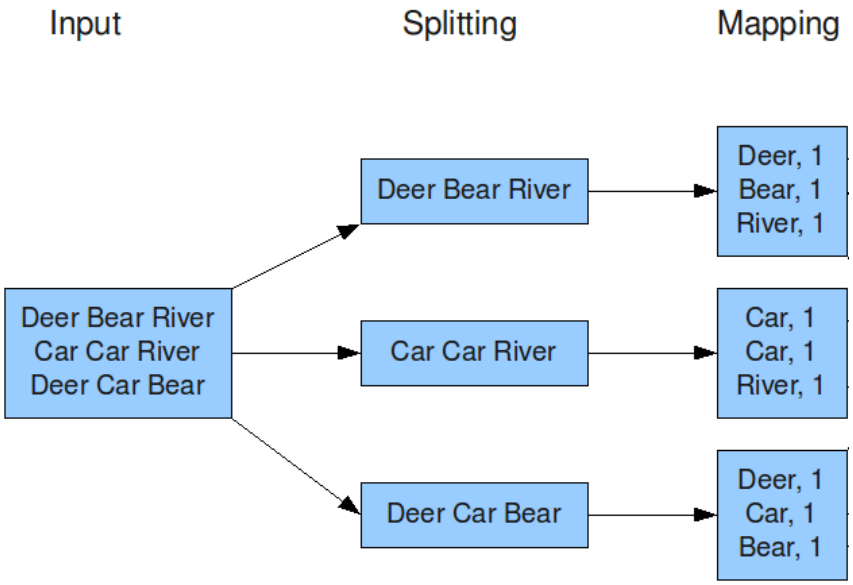
Input



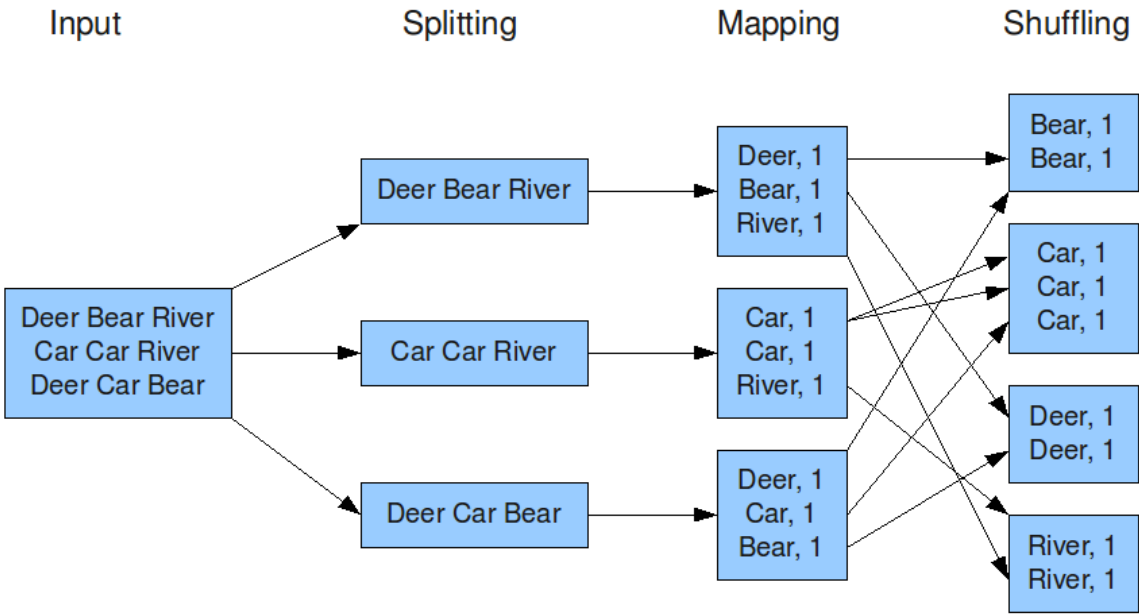
1. vstupní data jsou rozdělena mezi výpočetní uzly



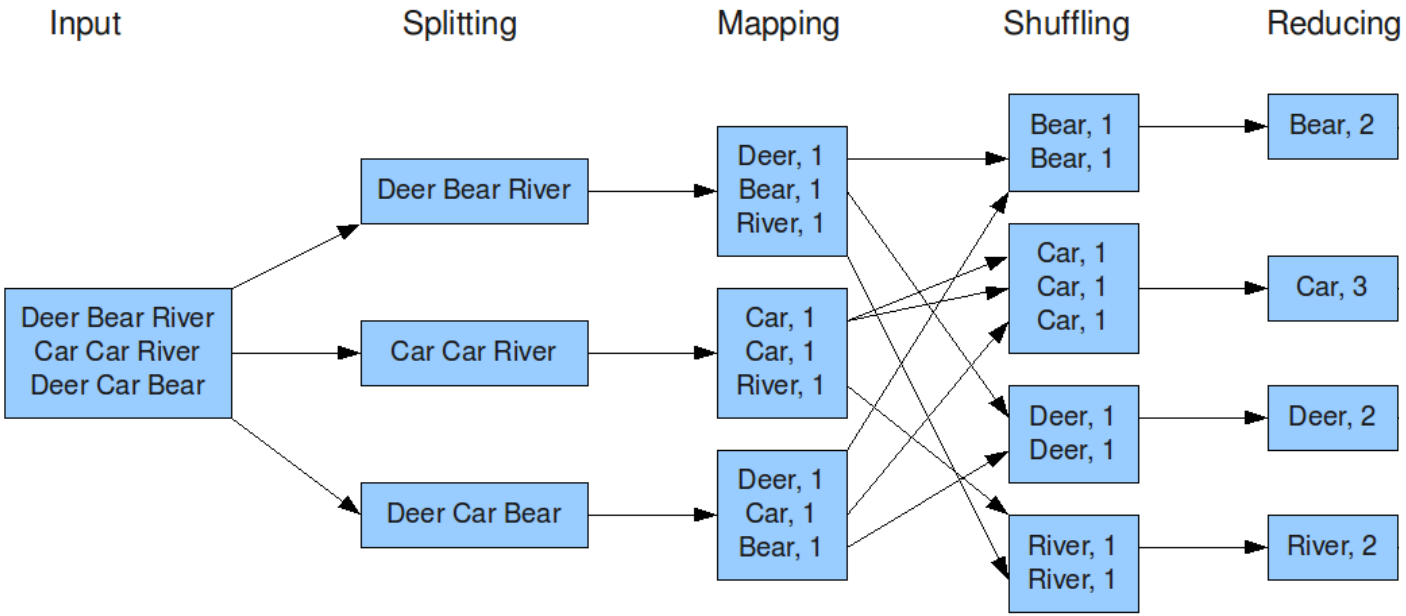
2. každý uzel provede *map*



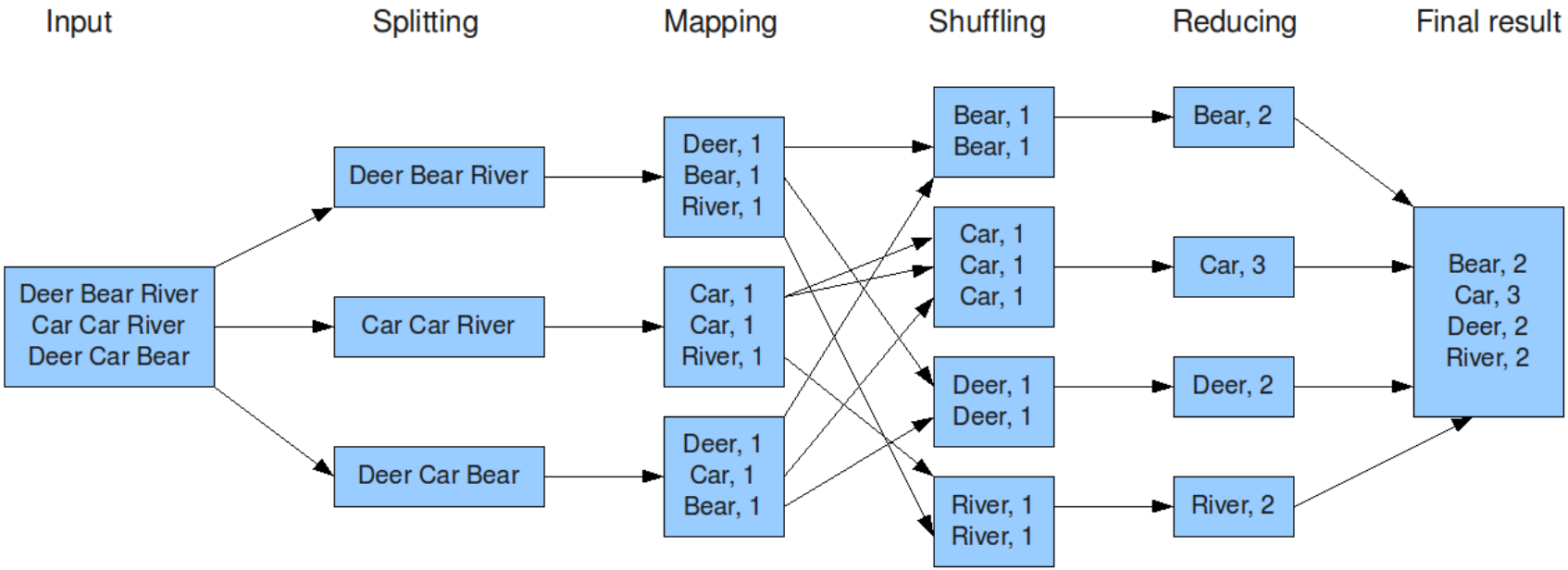
3. mezivýsledky jsou rozděleny mezi uzly dle klíčů *out_key*



4. každý uzel provede *reduce*



5. výsledky se sebírají a uloží na výstup



Objektově orientované programování

- C++, Java, Python, Ruby, JavaScript…
- Prakticky všechny moderní imperativní jazyky
- Objekty (zapouzdření) a zprávy (komunikace)
- Objekty = data + kód
- Dva přístupy:
  - třídy – nejčastější
  - prototypy

Třídy

- Abstrakce podstaty všech podobných objektů
- Předpis, jak vyrobit objekt
- Objekt = instance třídy
- V některých jazycích je třída zároveň objektem
  - třída = instance metatřídy
- Organizace do hiearchie dědičnosti
  - jednoduchá dědičnost → strom
  - vícenásobná dědičnost → orientovaný acyklický graf
  - v kořeni často obecná třída *object*
  - specializace, generalizace

Prototypy

- JavaScript, Self, Lua
- Každý objekt je jedinečný
- Objekty sdílejí určité rysy (traits)
- Delegace namísto dědičnosti

```
var Animal = function(name) {
        this.name = name;
};
Animal.prototype = {
        getName: function() {
                return this.name;
        }
};
var mici = new Animal('Mici');
var kevi = new Animal('Kevi');
```

Příklady jazyků

C

Chef

Ruby

LOLCODE

brainfuck

Scala

Shakespeare

JavaScript

Bash

Haskell

Prolog

Lisp

C++

Malbolge

Whitespace

OSTRAJava

Clojure

Basic

Python

Erlang

Java

PHP

Intercal

Swift

Pascal

Mathematica

Příklady jazyků

C

Ruby

LOLCODE

brainfuck

Chef

Scala

Shakespeare

JavaScript

Haskell

Bash

Prolog

Lisp

C++

Malbolge

Whitespace

OSTRAJava

Clojure

Basic

Python

Erlang

Java

PHP

Intercal

Swift

Pascal

Mathematica

IMPERATIVNÍ

Příklady jazyků

C

Chef

Ruby

LOLCODE

brainfuck

Scala

Shakespeare

JavaScript

Haskell

Bash

Prolog

Lisp

C++

Malbolge

Whitespace

OSTRAJava

Clojure

Basic

Python

Java

Erlang

PHP

Intercal

Swift

Pascal

Mathematica

OBJEKTOVĚ ORIENTOVANÉ

Příklady jazyků

C

Chef

Ruby

LOLCODE

brainfuck

Scala

Shakespeare

JavaScript

Bash

Haskell

Prolog

Lisp

C++

Malbolge

Whitespace

OSTRAJava

Basic

Clojure

Python

Erlang

Java

PHP

Intercal

Swift

Pascal

Mathematica

FUNKCIONÁLNÍ

Příklady jazyků

C

Chef

Ruby

LOLCODE

brainfuck

Scala

Shakespeare

JavaScript

Haskell

Bash

Prolog

Lisp

Malbolge

C++

Whitespace

OSTRAJava

Clojure

Basic

Python

Erlang

Java

PHP

Intercal

Swift

Pascal

Mathematica

LOGICKÉ

Příklady jazyků

C

Ruby

LOLCODE

brainfuck

Chef

Scala

Shakespeare

JavaScript

Haskell

Bash

Prolog

Lisp

C++

Malbolge

Whitespace

OSTRAJava

Basic

Clojure

Python

Erlang

Java

PHP

Intercal

Swift

Pascal

Mathematica

EZOTERICKÉ

Příklady jazyků

C

Chef

Ruby

LOLCODE

brainfuck

Scala

Shakespeare

JavaScript

Bash

Haskell

Prolog

Lisp

C++

Malbolge

Whitespace

OSTRAJava

Clojure

Basic

Python

Erlang

Java

PHP

Intercal

Swift

Pascal

Mathematica

SKRIPTOVACÍ

Příklady jazyků

C

Chef

Ruby

LOLCODE

brainfuck

Scala

Shakespeare

JavaScript

Haskell

Bash

Prolog

Lisp

C++

Malbolge

Whitespace

OSTRAJava

Clojure

Basic

Python

Erlang

Java

PHP

Intercal

Swift

Pascal

Mathematica

PRO WEB

Podle čeho volit jazyk?

Podle úlohy

- web lze napsat i v Lispu, ale proč?

Podle cílové platformy, přenositelnosti

- pokud to má běžet pod JVM, nebudu psát v C++

Podle složitosti vývoje

- některé věci se rychleji naprogramují v C, jiné v PHP

Podle požadavků na výkon programu

- JavaScript bude nejspíš pomalejší než C

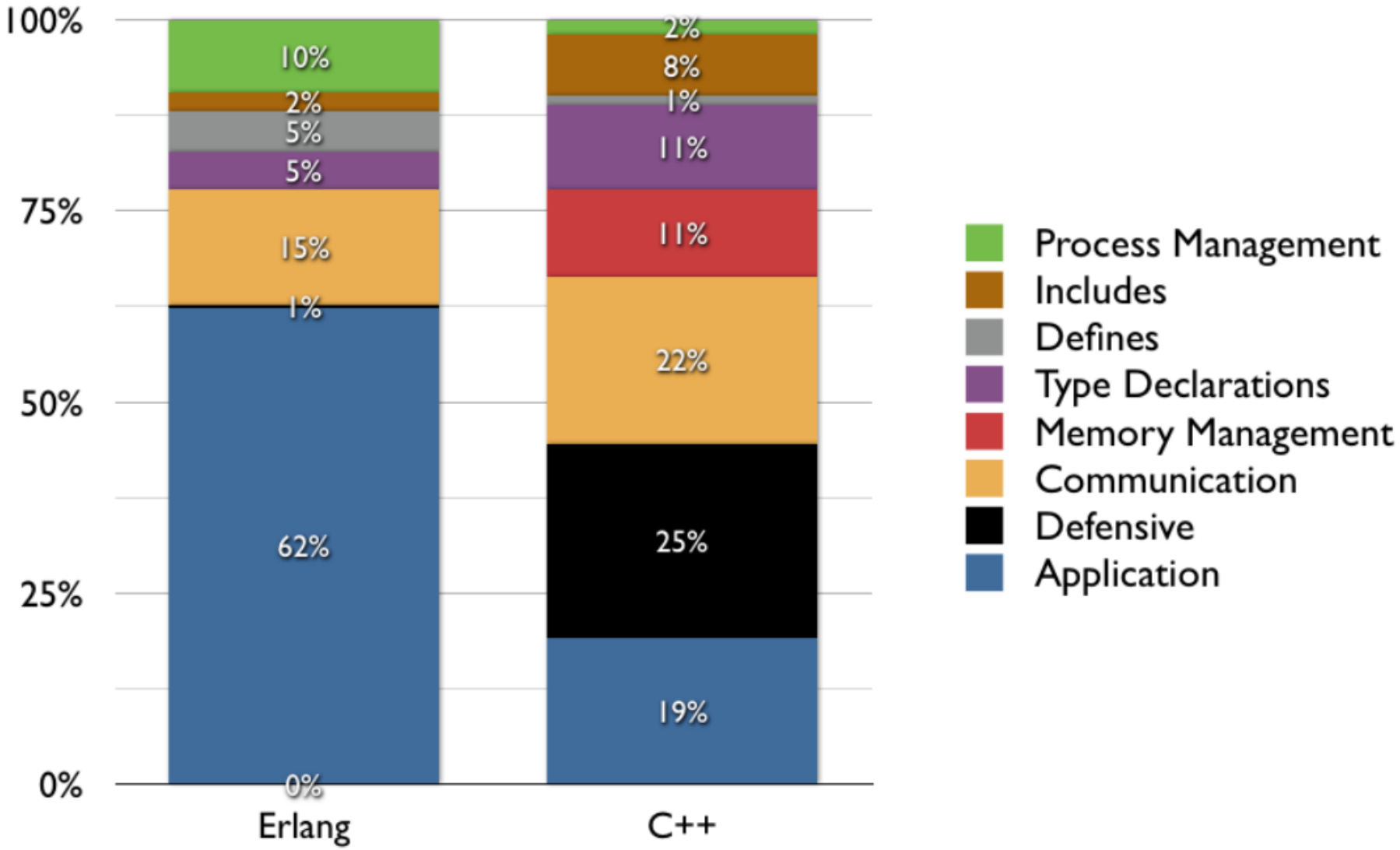
Podle velikosti komunity („popularity“)

- čím je větší, tím spíš už někdo řešil můj problém

Podle osobních preferencí

- někdo má rád Haskell, někdo Python, někdo…

Příklad – proporce kódu



Simplified Wrapper and Interface Generator

SWIG

Někdy se kombinuje více jazyků v jedné aplikaci

Například:

- uživatelské rozhraní ve vyšším jazyce
- časově kritické operace v C/C++

Mnoho jazyků podporuje moduly napsané v C/C++

- ale je třeba vytvořit rozhraní na míru vyššímu jazyku
- nebo jej lze (polo)automaticky vygenerovat

- Simplified Wrapper and Interface Generator
- Propojení kódu v C/C++ s vysokoúrovňovými jazyky
- Generuje adaptéry (wrappers) nad deklaracemi z hlavičkových souborů C/C++

Podpora pro 23 jazyků:

Allegro CL	*Go*	*Mzscheme*	*R*
C#	*Guile*	*OCAML*	*Ruby*
CFFI	*Java*	*Octave*	*Scilab*
CLISP	*Javascript*	*Perl*	*Tcl*
Chicken	*Lua*	*PHP*	*UFFI*
D	*Modula-3*	*Python*	

SWIG – příklad

```c
/* example.c */

#include <time.h>

double My_variable = 3.0;

int fact(int n) {
        if (n <= 1) return 1;
        else return n*fact(n-1);
}

int my_mod(int x, int y) {
        return (x%y);
}

char *get_time() {
        time_t ltime;
        time(&ltime);
        return ctime(&ltime);
}
```

```
/* example.i */

%module example
%{
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}


extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

SWIG – příklad

```
/* example.c */

#include <time.h>

do

int

);
}

int

}

ch
        time_t ltime;
        time(&ltime);
        return ctime(&ltime);
}
```

vše uvnitř %{ ... %} bude zkopírováno

```
/* example.i */

%module example
%{
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}
```

deklarace, ke kterým se vygeneruje wrapper

```
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

SWIG – příklad

```c
/* example.c */

#include <time.h>

double My_variable = 3.0;

int fact(int n) {
        if (n <= 1) return 1;
        else return n*fact(n-1);
}

int my_mod(int x, int y) {
        return (x%y);
}

char *get_time() {
        time_t ltime;
        time(&ltime);
        return ctime(&ltime);
}
```

```
/* example.i */

%module example
%{
/* Includes the header
   in the wrapper code */
#include "example.h"
%}

/* Parse the header file
   to generate wrappers */
#include "example.h"
```

SWIG – příklad

```
# vytvoreni wrapperu swigem (vznikne example_wrap.c)
$ swig -python example.i

# preklad modulu
$ gcc -c example.c example_wrap.c -fPIC $(pkg-config --cflags python)



# linkovani do dynamicke knihovny _example.so
$ ld -shared example.o example_wrap.o -o _example.so

# import z pythonu a spusteni
$ python
[...]
>>> import example
>>> example.get_time()
'Mon Apr 17 12:00:44 2017\n'
>>> exit
```

SWIG – příklad

```
# vytvoreni wrapperu swigem (vznikne example_wrap.c)
$ swig -perl example.i

# preklad modulu
$ gcc -c example.c example_wrap.c \
      $(perl -MConfig -e 'print join(" ", @Config{qw(ccflags
optimize cccdlflags)}, "-I$Config{archlib}/CORE")')

# linkovani do dynamicke knihovny example.so
$ ld -shared example.o example_wrap.o -o example.so

# import z perlu a spusteni
$ perl
use example;
print exaple::get_time();
```

LEGACY CODE

Kód který:

- je těžké upravovat
- nepřehledný
- bez testů
- je zděděný po někom jiném
- bychom nejradši nechali zmizet
  - ale nemůžeme, protože je důležitý a užitečný
  - pokud by nebyl důležitý, už by se dávno nepoužíval

Nemusí to být každý kód, který:

- vypadá škaredě
- napsal někdo jiný
- je starý

Výhody „starého“ kódu?

- program je ověřen praxí jako funkční
- uživatelé jsou na něj zvyklí

Ale co když je do něj třeba zasáhnout?

- nová funkcionalita
- oprava chyby
- refactoring
- optimalizace

Jak upravit legacy code?

Metoda č. 1:

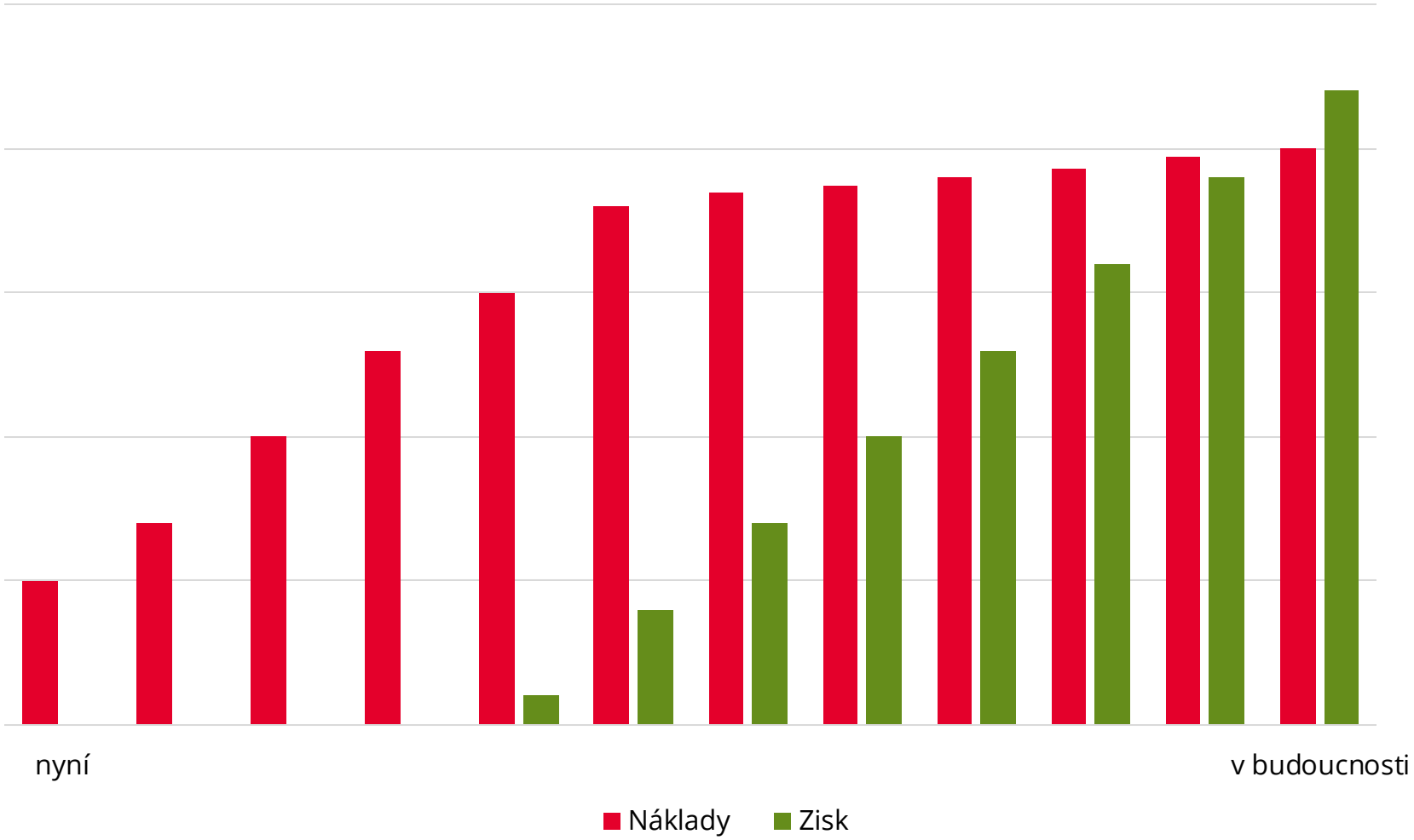
1. Udělej změny
2. Modli se

Metoda č. 2

1. Vše zahoď
2. Napiš to znova

Zahodit a přepsat od nuly?

Dosažení původní funkcionality trvá dlouho a je drahé.



Metoda č. 3

1. Najdi místa, která je třeba změnit
2. Najdi místa, kde je otestovat
3. Zlikviduj závislosti
4. Napiš testy
5. Udělej změny

Zádrhel:

- je třeba mít testy, aby šlo (bezpečně) změnit kód
- je třeba změnit kód, aby šlo vytvořit testy

- Je třeba být velmi opatrný
- Pokud možno nezhoršit situaci
- Nový kód co nejvíce izolovat od starého a otestovat

Sprout method:

- vytvořit novou funkci a pokrýt ji testy
- do starého kódu přidat volání této funkce

Wrap method:

- původní funkci přejmenovat
- namísto ní vytvořit wrapper, který volá starý kód

Sprout method

```
public void postEntries(List entries) {
    for (Entry entry : entries) {
            entry.postDate();

    }
    transaction.getListManager().addAll(entries);
}
```

Sprout method

```
public void postEntries(List entries) {
    List entriesToAdd = new LinkedList();
    for (Entry entry : entries) {
        if(!transaction.getListManager().contains(entry)) {
            entry.postDate();
            entriesToAdd.add(entry);
        }
    }

    transaction.getListManager().addAll(entriesToAdd);
}
```

Sprout method

```
public void postEntries(List entries) {
    for (Entry entry : entries) {
            entry.postDate();

    }
    transaction.getListManager().addAll(entries);
}
```

Sprout method

```
public void postEntries(List entries) {
    List<Entry> filteredEntries = uniqueEntries(entries);
    for (Entry entry : filteredEntries) {

            entry.postDate();


    }

    transaction.getListManager().addAll(filteredEntries);
}
```

Wrap method

```
public void postEntries(List entries) {
    for (Entry entry : entries) {
            entry.postDate();

    }
    transaction.getListManager().addAll(entries);
}
```

Wrap method

```
public void doPostEntries(List entries) {

    for (Entry entry : entries) {

            entry.postDate();


    }

    transaction.getListManager().addAll(entries);
}

public void postEntries(List entries) {
    doPostEntries(uniqueEntries(entries));
}
```

Odkazy

- https://www.slideshare.net/nashjain/working-effectively-with-legacy-code-presentation
- http://www.netobjectives.com/system/files/WorkingEffectivelyWithLegacyCode.pdf
- https://speakerdeck.com/nhpatt/working-effectively-with-legacy-code
- https://speakerdeck.com/paultaykalo/working-with-legacy-codebase
- https://speakerdeck.com/cbushell/working-effectively-with-legacy-code

iwiglasz@fit.vutbr.cz