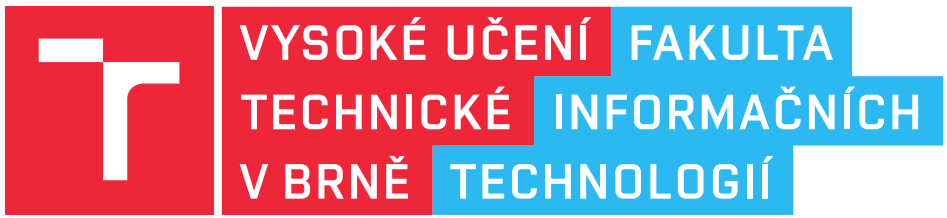
Programovací jazyky a paradigmata, SWIG a práce se starším kódem

Dominika Regéciová, Michal Wiglasz

Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 1/2. 612 66 Brno - Královo Pole
iregeciova@fit.vutbr.cz, iwiglasz@fit.vutbr.cz

Praktické aspekty vývoje software (IVS) 2020



JAZYKY A PARADIGMATA

* Jazyky

Jazyk je systém sloužící jako základní prostředek lidské komunikace.

- Lingvisté uvádí, že existuje zhruba 7 tisíc jazyků
- Z toho 5-6 tisíc jazyků ovládá pouze 5% populace

Jazyky ovlivňují, jak myslíme a jak se chováme.

- Sapir-Whorfova Hypotéza: pojetí reálného světa je vystavěno na jazykových zvyklostech konkrétní dané komunity, jež pak předurčuje určitý výběr interpretace reality

Učením jazyků se dozvídáme více i o kultuře a zvyklostech dané skupiny.

When Americans say...	It means...
Awesome	Good
Fabulous	Good
Amazing	Good
Great	Fine
Fine	Bad
OK	Bad
Not so great	Really bad
Challenging	Driving me completely nuts
Hilarious	Unexpected
For sure	Probably
Forever	30 minutes
Let's get coffee sometime	Goodbye; I like you
Let's stay in touch	Goodbye; I don't like you that much
My friend	A person I know
My best friend	A person I know and also like

gerrycanavan

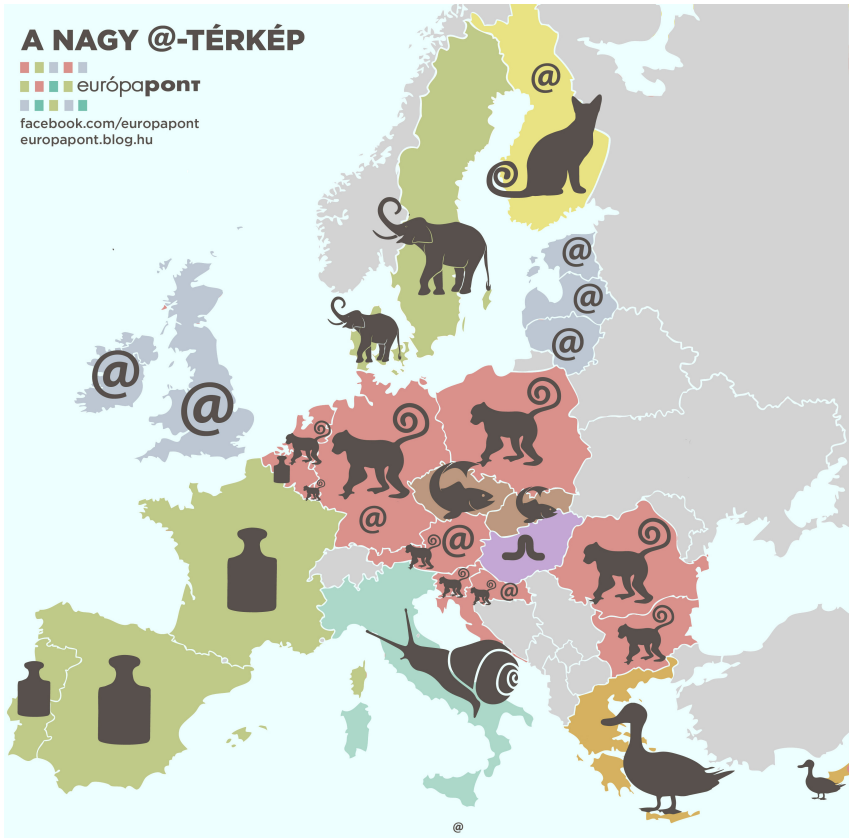
I'm afraid of Americans

lindsayribar

This is super real.

* Jazyky: Odlišné označení symbolu @

finština	miukumauku	mňau mňau
dánština	snabel-a	chobotové a
italština	chocciola	šnek
španělština	arroba	měřice (stará dutá míra)
řečtina	papaki	kachňátko



Nemáme jednu striktní definici programovacího jazyka:

- Systém kódování, které umožňuje vytvořit program, představující určitý úkol, který má vykonávat počítač.

Široké spektrum programovacích jazyků

- různé perspektivy, vlastnosti, podporované konstrukce, způsoby zápisu, ...
- Wikipedia: List of programming languages → cca 700
- HOPL kolekce jazyků: uvádí skoro 9 tisíc jazyků

Jaký jazyk zvolit?

- ...

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 $\rightarrow$ méně chyb
- přenositelné mezi platformami

Imperativní vs. deklarativní

Imperativní = specifikace, jak se problém řeší

- 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, ...

```
output = []
for N in input:
  if N > 10:
    output.append(N * N)
```

```
SELECT N * N
FROM input
WHERE N > 10
```

Statické vs. dynamické

Statické (C, C++, Java, . . .)

- 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 (PHP, Python, JavaScript, . . .)

- 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 (Java, Python, . . .)

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

Slabě typované (Perl, PHP, . . .)

- automaticky přetypuje hodnoty dle potřeby

Dynamické typování $\neq$ slabé typování (a naopak)

Automatická vs. manuální správa paměti

Automatická

- programátor paměť jen alokuje
- nepoužívanou paměť uvolňuje
  - počítaní referencí – neuvolní cykly
  - sledovací algoritmy – přeruší běh programu

Manuální

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

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

Dnes často jeden jazyk umožňuje kombinovat více paradigmat.

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

```
output = []
for N in input:
    if N > 10:
        output.append(N * N)
```

- 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

```
output = [N for N in input
          if N > 10]
```

- Pseudokód pro řadící algoritmus Quicksort:

```
procedure quicksort(List values)
  if values.size <= 1 then
    return values

  pivot = nahodny prvek z values

  list1 = { prvky vetsi nez pivot }
  list2 = { pivot }
  list3= { prvky mensi nez pivot }

  return quicksort(list1) + list2 + quicksort(list3)
```

- Stejný algoritmus v jazyce Haskell:

```
qsort [] = []
qsort (x:xs) = qsort small ++ pivot ++ qsort large
    where
        small = [y | y <- xs, y < x]
        pivot = [y | y <- xs, y == x] ++ [x]
        large = [y | y <- xs, y > x]
```

- Prolog
- založeno na matematické logice
- program = konečná množina axiomů
- výpočet důkaz dotazu uživatele

```
muz(honza). muz(jirka). muz(vilik).
zena(monika). zena(jana).
jeDite(honza,jirka). jeDite(jana,monika).
jeDite(vilik,monika).
jeSyn(X,Y) :- jeDite(X,Y), muz(X).

>>> jeSyn(X,monika)

X = vilik
```

- Erlang, Elixir, ...
- programy jsou popsané jako procesy, které spolu komunikují
- nemusí nutně běžet paralelně

```
-module(tut15).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
   Pong_PID ! finished,
   io:format("ping finished~n", []);

ping(N, Pong_PID) ->
   Pong_PID ! {ping, self()},
   receive
      pong ->
         io:format("Ping rcvd. pong~n", [])
   end,
   ping(N - 1, Pong_PID).
```

```
pong() ->
   receive
      finished ->
         io:format("Pong finished~n", []);
      {ping, Ping_PID} ->
         io:format("Pong rcvd. ping~n", []),
         Ping_PID ! pong,
         pong()
   end.

start() ->
   Pong_PID = spawn(tut15, pong, []),
   spawn(tut15, ping, [3, Pong_PID]).
```

- Program vytváří/modifikuje program (klidně sám sebe)
- Generování kódu
- Šablony v C++
- Anotace v Javě
- Dekorátory v Pythonu
- eval()
- Překladače

```
template <int N>
struct Factorial {
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0> {
    enum { value = 1 };
};

void foo() {
    int x = Factorial<4>::value; // == 24
    int y = Factorial<0>::value; // == 1
}
```

Imperativně:

```
output = 0
for N in input:
  if N > 10:
    output += N*2
```

Funkcionálně:

```
output = sum(
  N*2 for N in input if N > 10
)
```

Funkcionálně:

```
from functools import reduce
sum = lambda x, y: x + y
mul = lambda x: x*2
cmp = lambda x: x > 10
output = reduce(sum, map(mul, filter(cmp, input)))
```

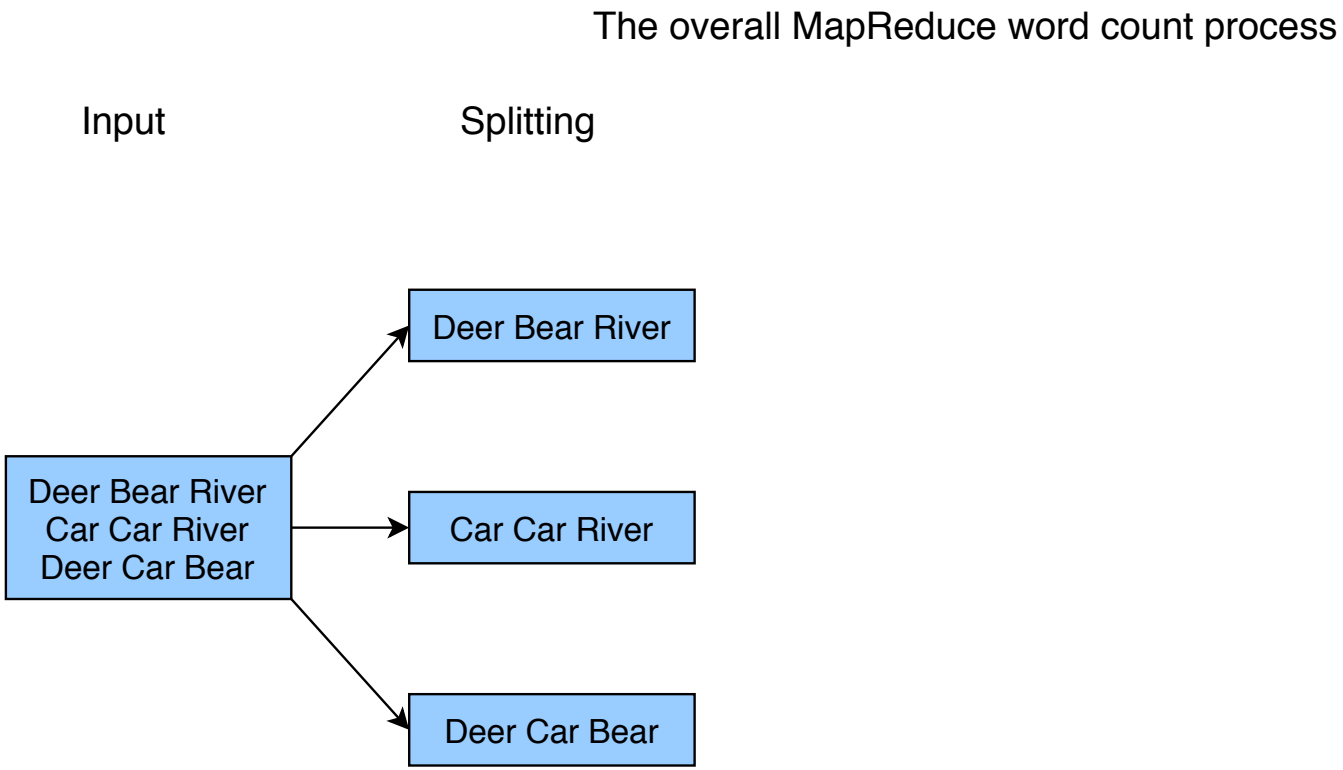
- Metoda pro paralelizaci výpočtu (i pro big data)
- Inspirace funkcionálními jazyky
- Založeno na funkcích *map* a *reduce*
  - *map(in_key, in_value)* $\rightarrow$ *list(out_key, intermediate_value)*
  - *reduce(out_key, list(intermediate_value))* $\rightarrow$ *list(out_value)*
- Programátor dodá jen funkce *map* a *reduce*
- O distribuci mezi výpočetní uzly a tok dat se stará runtime
- Ale ne každý výpočet lze převést na MapReduce

The overall MapReduce word count process

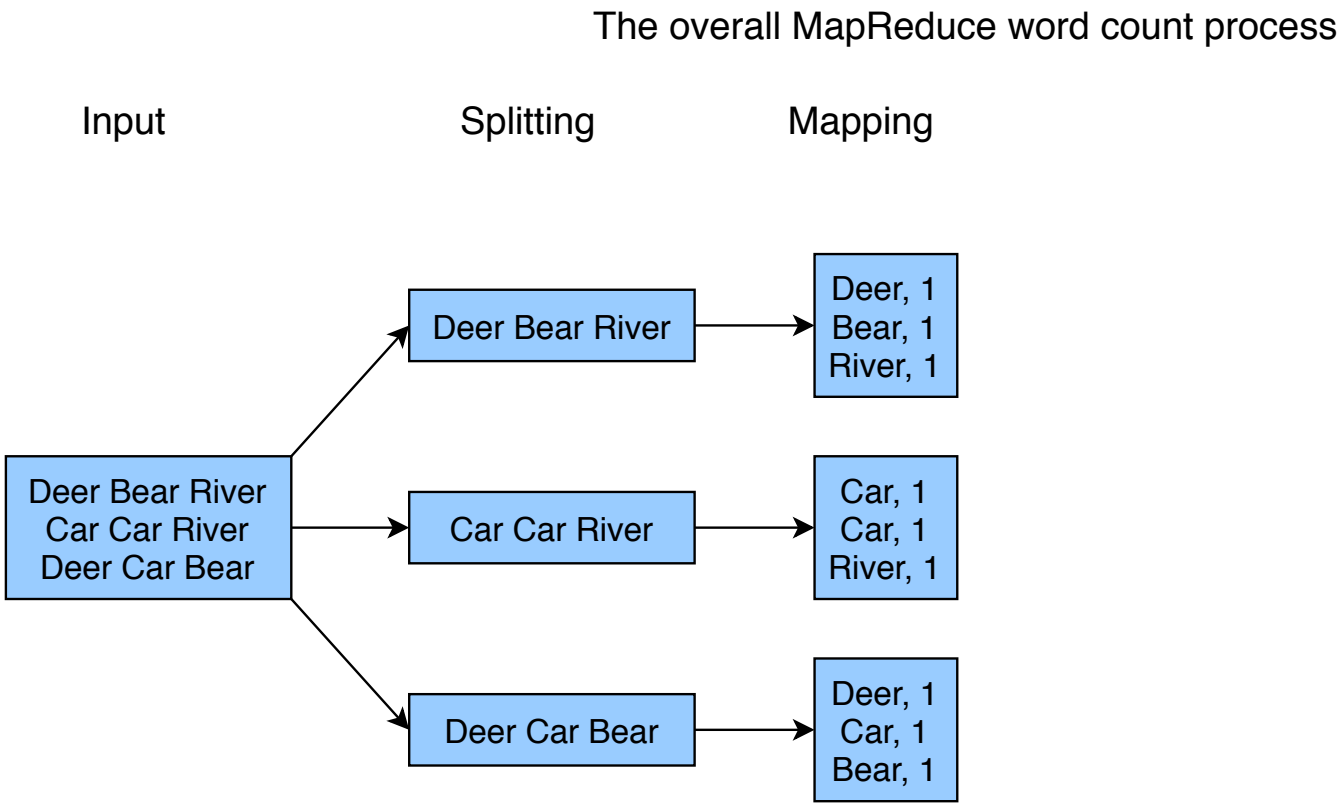
Input

Deer Bear River
Car Car River
Deer Car Bear

Určení počtu jednotlivých slov v kolekci dokumentů

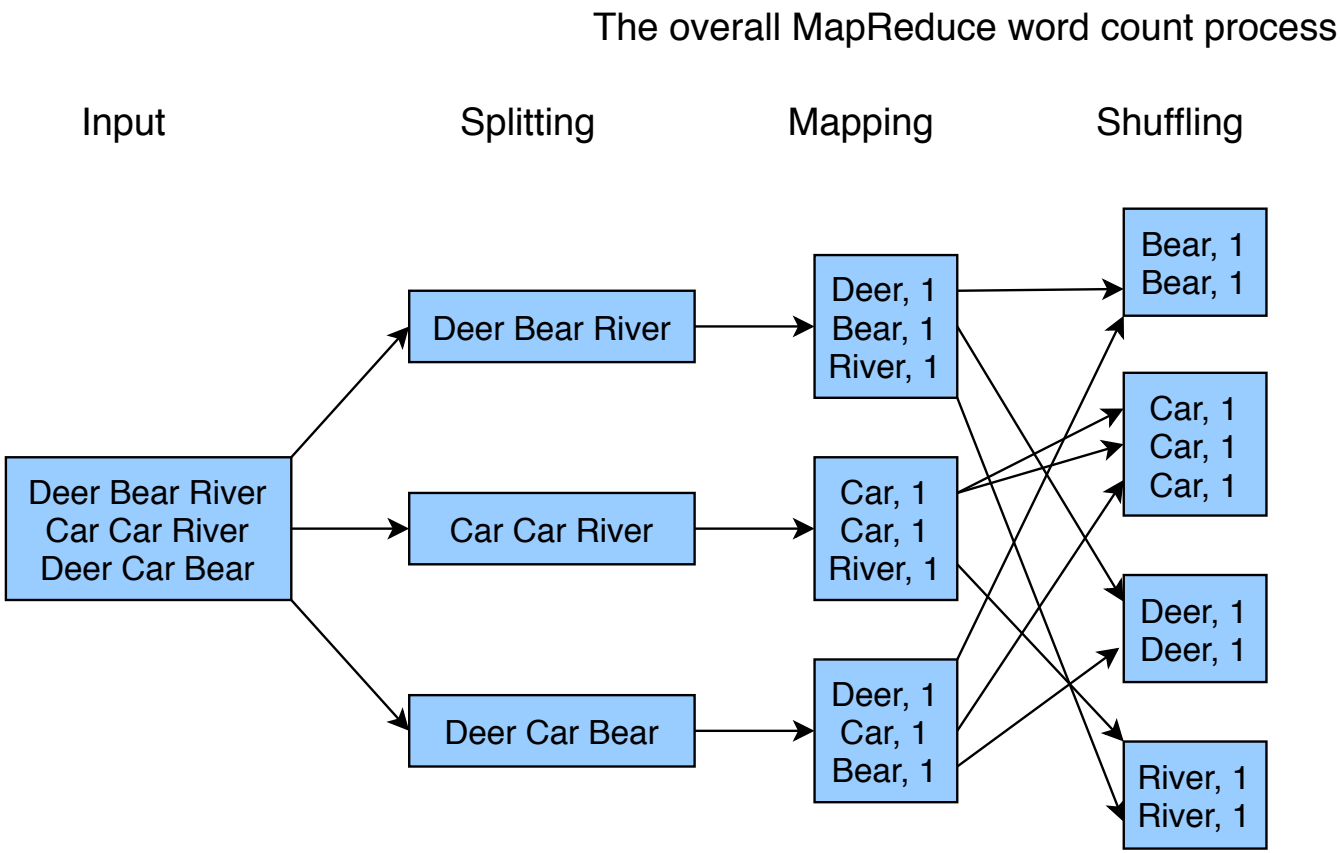


1. Vstupní data (dokumenty) jsou rozdělena mezi výpočetní uzly

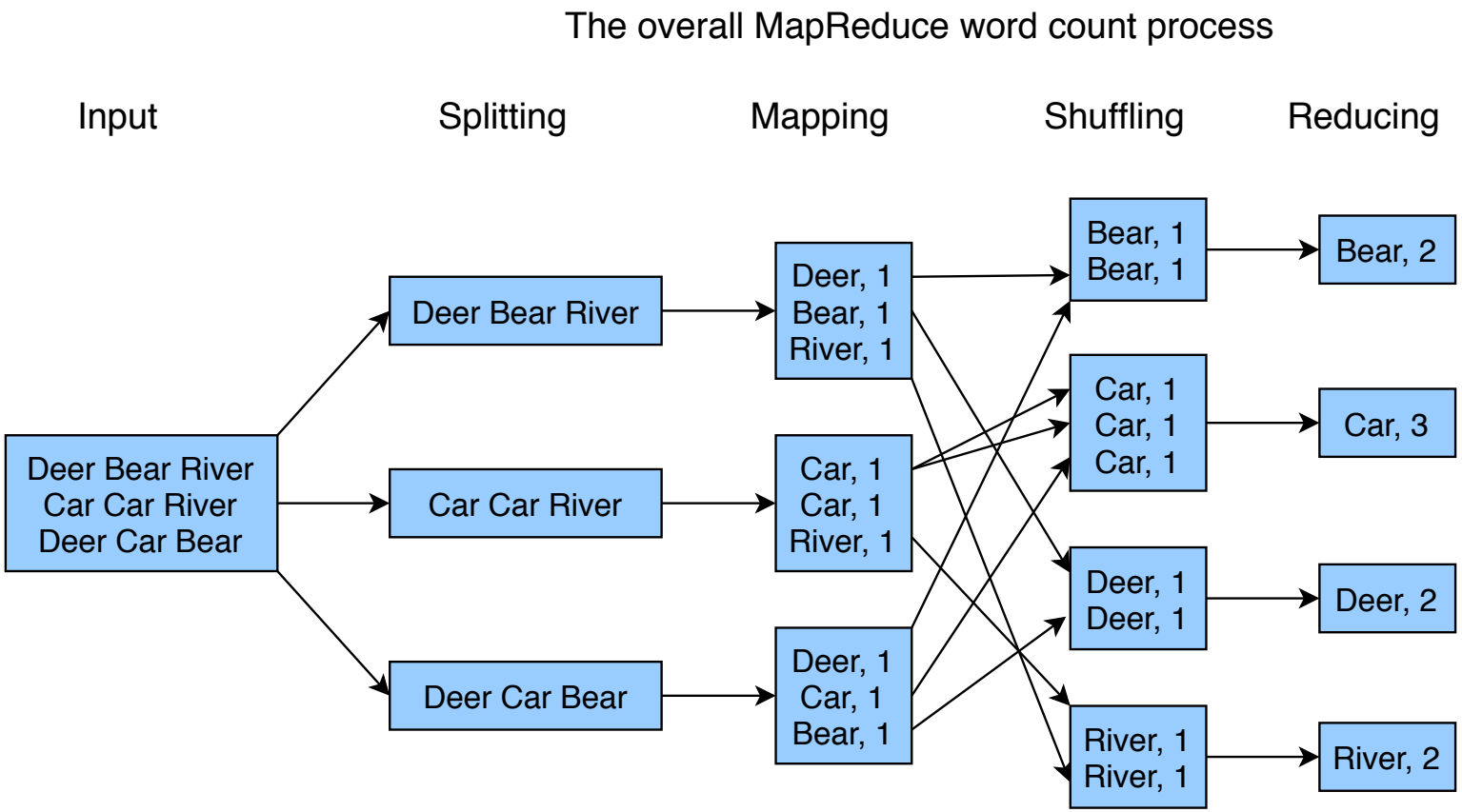


2. Každý uzel provede:

map(<klíč>, dokument) $\rightarrow$ *[(slovo, 1), (slovo, 1), (slovo, 1), . . .)*

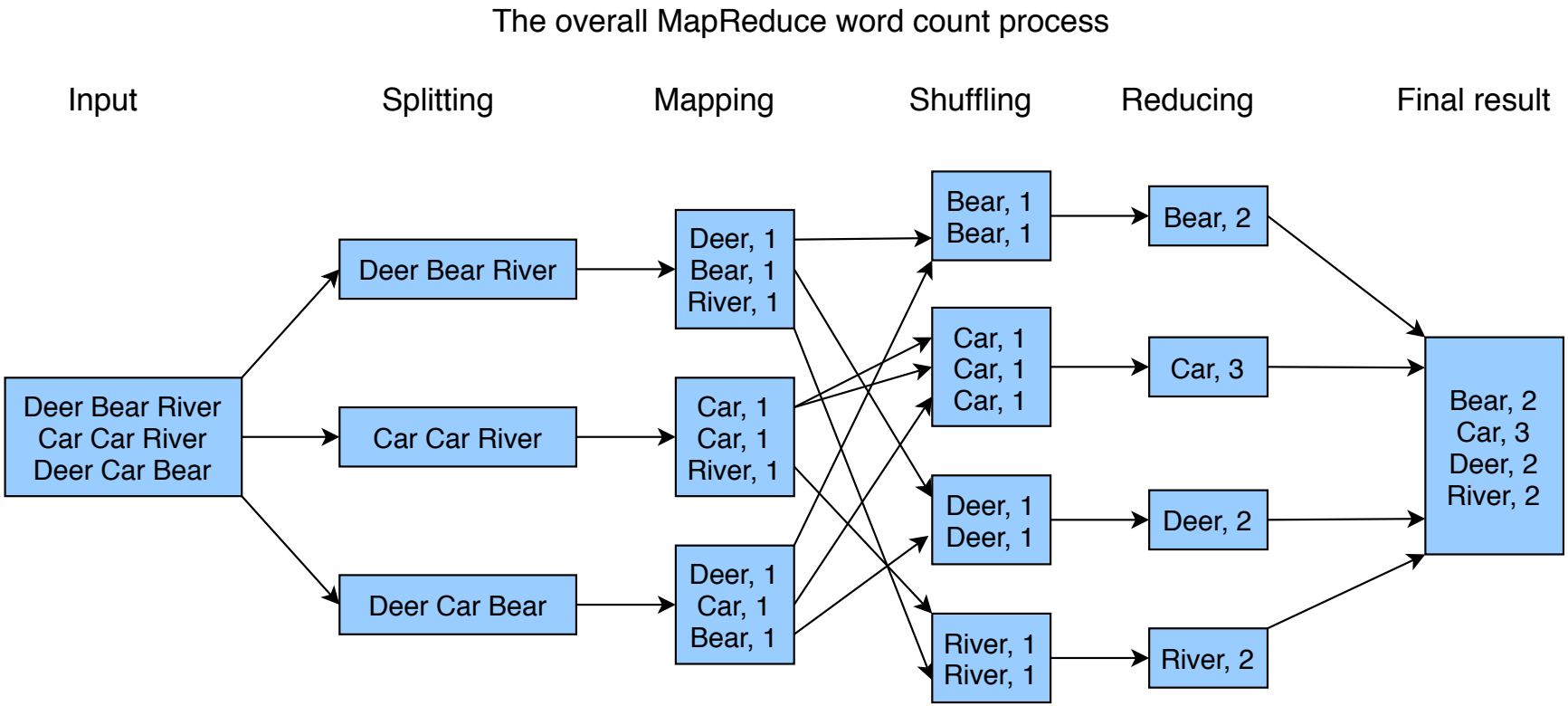


3. Mezivýsledky jsou rozděleny mezi uzly dle klíčů *out_key* (zde *slovo*)



4. Každý uzel provede:

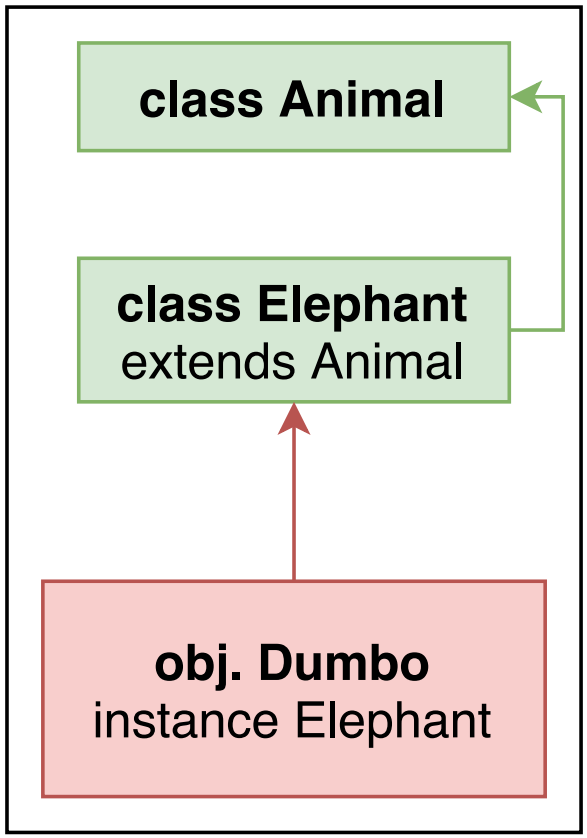
reduce(slovo, [počet, počet, počet, . . .]) → (slovo, počet)



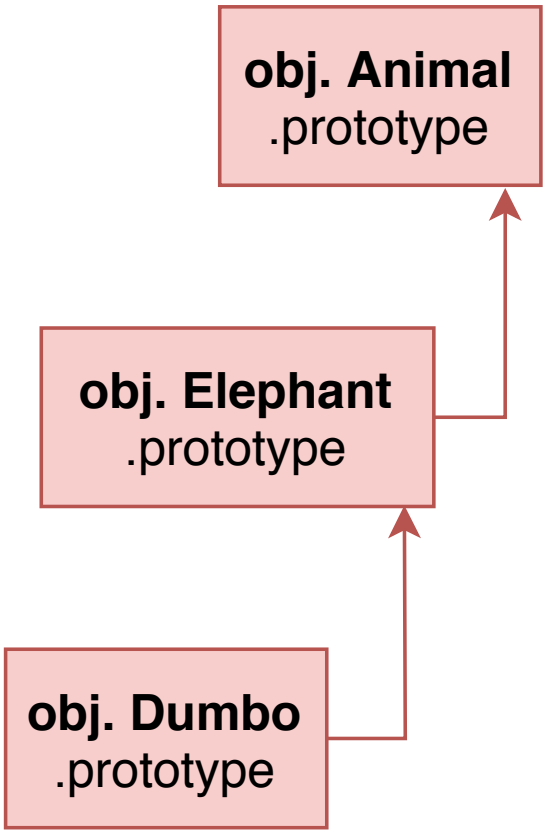
5. Výsledky se sesbírají a uloží na výstup

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

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



- JavaScript, Self, Lua
- Každý objekt je jedinečný
- Objekty sdílejí určité rysy (traits)
- Delegace namísto dědičnosti
- `https://www.zdrojak.cz/clanky/oop-v-javascriptu-i/`



Příklady jazyků

C

Ruby

LOLCODE

brainfuck

Scala

Chef

Haskell

JavaScript

Shakespeare

Lisp

Bash

Malbolge

Whitespace

Prolog

C++

Swift

Basic

Java

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

Příklady jazyků

C

Ruby

LOLCODE

brainfuck

Chef

Scala

JavaScript

Shakespeare

Haskell

Lisp

Bash

Malbolge

Prolog

C++

Whitespace

Basic

Java

Swift

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

IMPERATIVNÍ JAZYKY

Příklady jazyků

C

Ruby

LOLCODE

brainfuck

Chef

Scala

JavaScript

Shakespeare

Haskell

Lisp

Malbolge

Bash

Whitespace

C++

Prolog

Basic

Java

Swift

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

OBJEKTOVĚ ORIENTOVANÉ JAZYKY

Příklady jazyků

C

Ruby

LOLCODE

brainfuck

Chef

Scala

JavaScript

Shakespeare

Haskell

Lisp

Malbolge

Bash

Whitespace

C++

Prolog

Swift

Basic

Java

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

FUNKCIONÁLNÍ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

LOLCODE

Scala

Chef

Haskell

JavaScript

Shakespeare

Lisp

Bash

Malbolge

Whitespace

C++

Prolog

Swift

Basic

Java

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

LOGICKÉ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

LOLCODE

Scala

Chef

Haskell

JavaScript

Shakespeare

Lisp

Bash

Malbolge

Whitespace

C++

Prolog

Swift

Basic

Java

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

SKRIPTOVACÍ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

LOLCODE

Scala

Chef

JavaScript

Shakespeare

Haskell

Lisp

Malbolge

Bash

Whitespace

Prolog

C++

Swift

Basic

Java

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

WEBOVÉ JAZYKY

Příklady jazyků

C

JRuby

brainfuck

LOLCODE

Scala

Chef

JavaScript

Shakespeare

Haskell

Lisp

Malbolge

Bash

Whitespace

C++

Prolog

Swift

Basic

Java

Pascal

Clojure

Jython

Intercal

PHP

Erlang

Mathematica

OSTRAJava

JAZYKY NAD JVM

Příklady jazyků

C

Ruby

brainfuck

LOLCODE

Scala

Chef

Haskell

JScript

Shakespeare

Lisp

Bash

Malbolge

Whitespace

C++

Prolog

Swift

Basic

Java

Pascal

ClojureCLR

IronPython

Intercal

Erlang

PHP

Mathematica

OSTRAJava

JAZYKY NAD .NET (CLI)

Příklady jazyků

C

Ruby

LOLCODE

brainfuck

Scala

Chef

JavaScript

Shakespeare

Haskell

Lisp

Bash

Malbolge

C++

Whitespace

Prolog

Swift

Basic

Java

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

EZOTERICKÉ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

```
++++++++++[>+++++++>++++++++++>++
+>+<<<<-]>++.>+.+++++++..+++.>++.<<++
++++++++++++.>.+++.------.--------.>+.>.
```

Chef

JavaScript

Shakespeare

Lisp

Bash

Malbolge

Whitespace

Prolog

C++

Swift

Basic

Java

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

EZOTERICKÉ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

LOLCODE

```
HAI
CAN HAS STDIO?
VISIBLE "HAI WORLD!"
KTHXBYE
```

Chef

JavaScript

Sh

ala

Haskell

Lisp

Bash

Malbolge

Prolog

C++

Whitespace

Swift

Basic

Java

Pascal

Clojure

Python

Intercal

PHP

Erlang

Mathematica

OSTRAJava

EZOTERICKÉ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

Chef

```
Hello World Cake with Chocolate sauce.
[...]
Ingredients.
33 g chocolate chips
100 g butter
[...]
Method.
Put chocolate chips into the mixing bowl.
Put butter into the mixing bowl.
Put sugar into the mixing bowl.
```

JavaScript

Malbolge

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

EZOTERICKÉ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

LOLCODE

Scala

Chef

JavaScript

Shakespeare

Haskell

Lisp

Bash

Malbolge

Whitespace

C++

```
(=<`#9]~6ZY32Vx/4Rs+0
No-&Jk)"Fh}|Bcy?`=*z]K
w%oG4UUS0/@-ejc(:'8dc
```

Basic

Java

Swift

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

EZOTERICKÉ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

LOLCODE

Scala

Chef

Haskell

JavaScript

Shakespeare

```
Romeo, a young man with a remarkable patience.
Juliet, a likewise young woman of remarkable grace.
[...]
                    Act I: Hamlet's insults and flattery.
                    Scene I: The insulting of Romeo.
[Enter Hamlet and Romeo]
Hamlet:
You lying stupid fatherless big smelly half-witted coward! You
are as stupid as the difference between a handsome rich
brave hero and thyself!
```

Malb **itespace**

Pascal

Cl

hon

PHP

OSTRAJava

EZOTERICKÉ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

LOLCODE

Scala

Chef

JavaScript

Shakespeare

Haskell

Lisp

Bash

Malbolge

Whitespace

C++

Prolog

Basic

Java

Swift

Pascal

Clojure

Python

Intercal

Erlang

PHP

Mathematica

OSTRAJava

EZOTERICKÉ JAZYKY

Příklady jazyků

C

Ruby

brainfuck

Chef

JavaScript

Shakesp

ell

Malbolge

Prolog

C++

ace

Basic

ascal

Clojure

Intercal

Erlang

PHP

```
DO ,1 <- #13
PLEASE DO ,1 SUB #1 <- #238
DO ,1 SUB #2 <- #108
DO ,1 SUB #3 <- #112
DO ,1 SUB #4 <- #0
DO ,1 SUB #5 <- #64
DO ,1 SUB #6 <- #194
DO ,1 SUB #7 <- #48
PLEASE DO ,1 SUB #8 <- #22
DO ,1 SUB #9 <- #248
DO ,1 SUB #10 <- #168
DO ,1 SUB #11 <- #24
DO ,1 SUB #12 <- #16
DO ,1 SUB #13 <- #162
PLEASE READ OUT ,1
PLEASE GIVE UP
```

AJava

EZOTERICKÉ JAZYKY

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

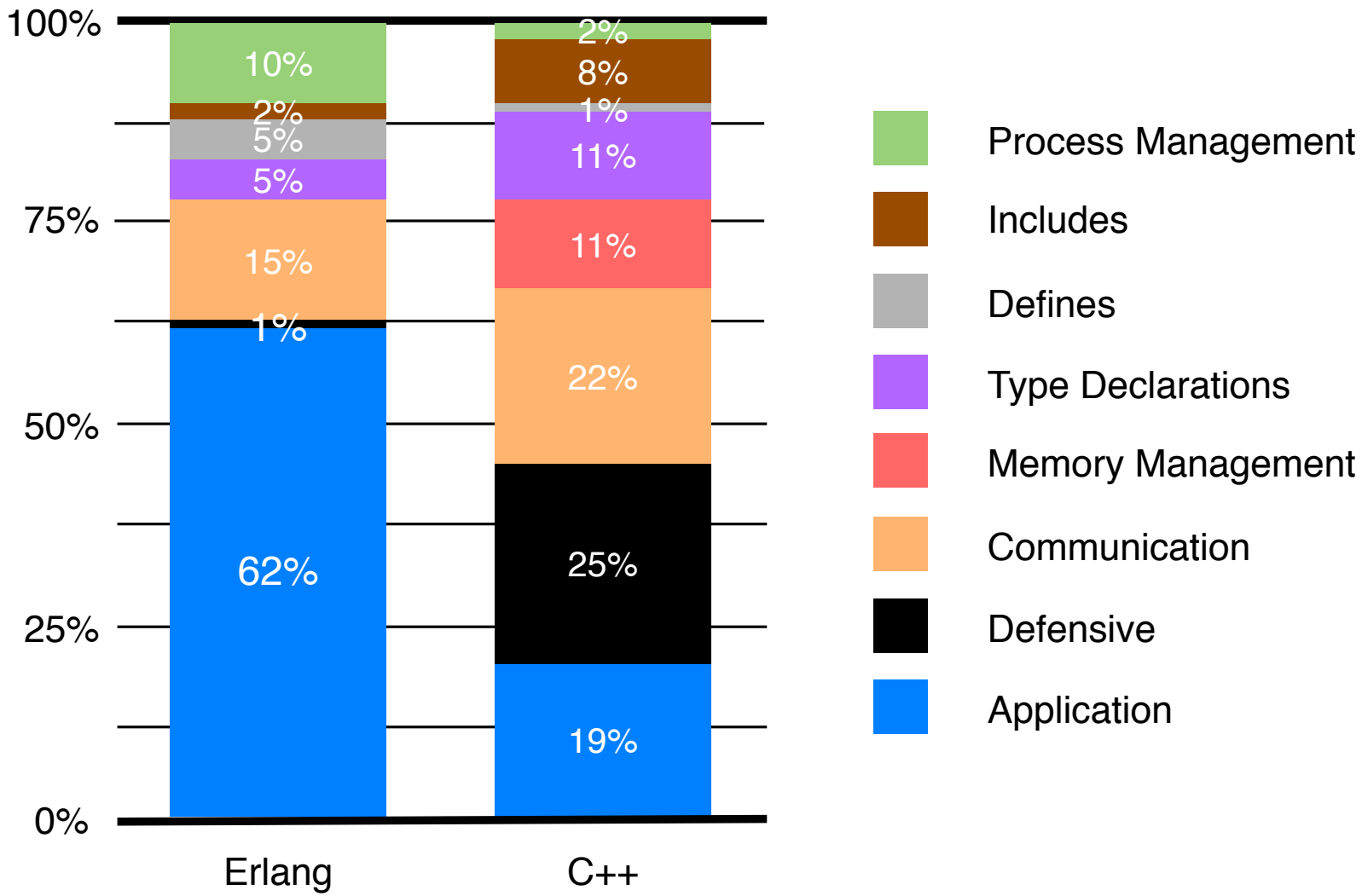
Podle velikosti komunity ("popularity")

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

Podle osobních preferencí

- někdo má rád Haskell, někdo Python, ...

Příklad - proporce kódu



Simplified Wrapper and Interface Generator

SWIG

Propojení jazyků

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(intx, inty) {
    return (x % y);
}

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

```c
/* example.i */

%module
%{
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 */
```

vše uvnitř %{ ... %} bude zkopírováno

```
}
```

deklarace, ke kterým se vygeneruje wrapper

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

```
/* example.i */

%module
%{
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

```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(intx, inty) {
    return (x % y);
}

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

```c
/* example.i */

%module
%{
/* Include the header
   in the wrapper code */
#include <example.h>
%}

/* Parse the 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 $(pgk-config - -cflags python)

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

# import z pythonu a spustetni
$ 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 \
        $(perl -MConfig -e 'print join("", {qw(ccflags optimize cccdlflags)},
"-I$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;
printexaple::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?

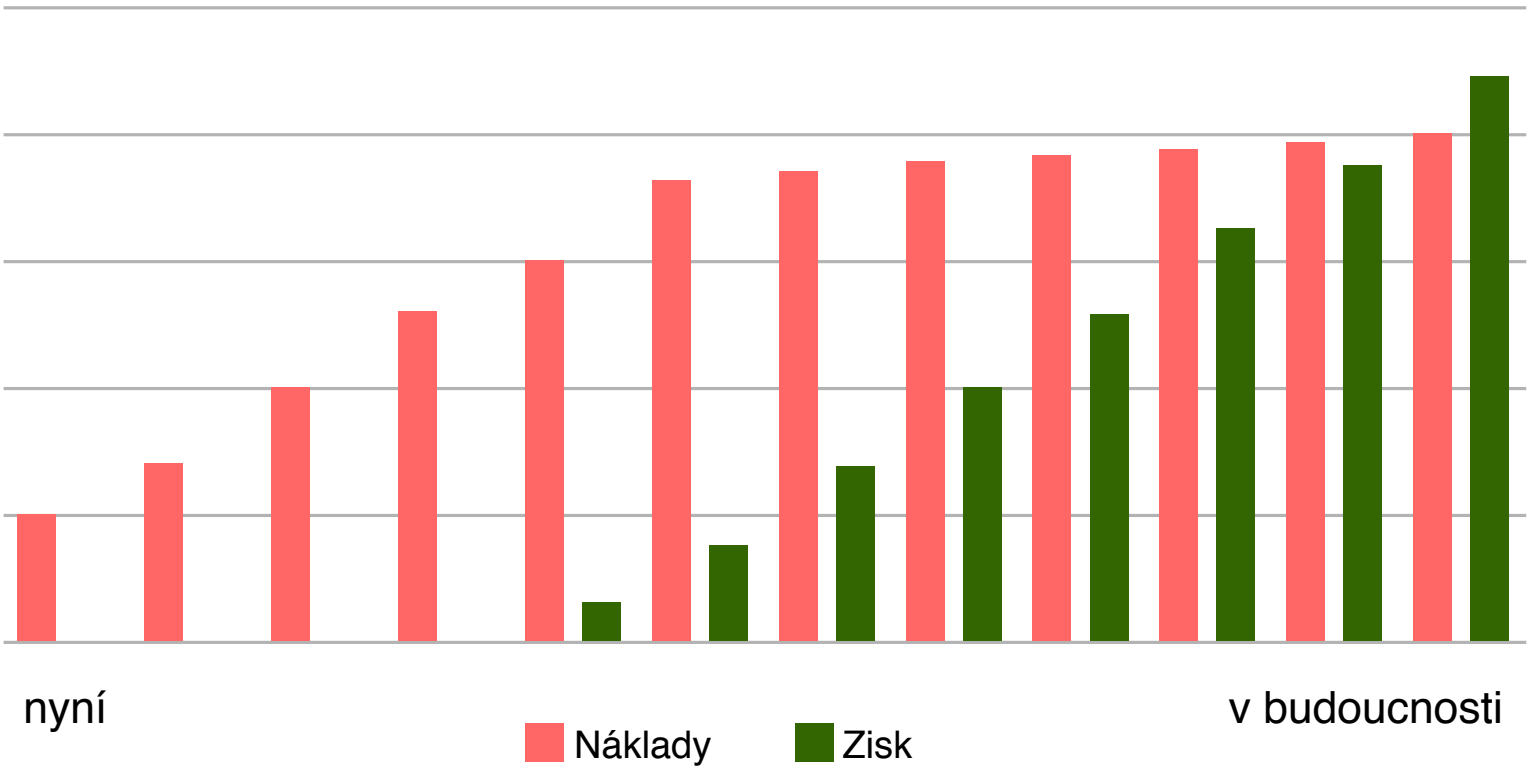
Metoda č. 1 - dát výpověď

Metoda č. 2 - hodit to na někoho jiného (konzultanty, původní autory, . . .)

Metoda č. 3 - začít úplně znova (vše staré bez milosti zahodit)

Zahodit a přepsat od nuly?

Dosažení původní funkcionality trvá dlouho a je to drahé.



Jak upravit legacy code?

Metoda č. 4 - Edit and Pray

- pořádně se seznámit se současným kódem
- pečlivě rozmyslet nutné změny
- začít měnit
- průběžně kontrolovat, jestli se něco nerozbilo
- nasadit na produkci
- čekat na odezvu zákazníků

Jak upravit legacy code?

Metoda č. 5

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));
}
```

Zdroje

- Jan Průcha: Interkulturní komunikace. Grada Publishing. 2010.
- Symbol @ není vždy zavináč, aneb jak to mají jinde ve světě?
- http://www.slideshare.net/nashjain/working-effectively-with-legacy-code-presentation
- https://github.com/ontiyonke/book-1/blob/master/[PROGRAMMING][WorkingEffectively.withLegacyCode].pdf
- http://accorsi.net/docs/WorkingEffectivelyWithLegacyCode.pdf
- http://speakerdeck.com/nhpatt/working-effectively-with-legacy-code
- http://speakerdeck.com/paultaykalo/working-with-legacy-codebase
- http://speakerdeck.com/cbushell/working-effectively-with-legacy-code