Univerzita Karlova
Pedagogická fakulta
Katedra informačních technologií a technické výchovy

BAKALÁŘSKÁ PRÁCE

Porovnání nejpoužívanějších paradigmat pro vývoj aplikací

Comparison of major paradigms for application development

Michael Hartman

Vedoucí práce: PhDr. Josef Procházka, Ph.D.

Studijní program: B7507 – Specializace v pedagogice

Studijní obor: Informační technologie se zaměřením na vzdělávání

2019

Odevzdáním této bakalářské práce na téma Porovnání nejpoužívanějších paradigmat pro vývoj aplikací potvrzuji, že jsem ji vypracoval pod vedením vedoucího práce samostatně za použití v práci uvedených pramenů a literatury. Dále potvrzuji, že tato práce nebyla využita k získání jiného nebo stejného titulu.

V Praze dne 12.7.2019

Děkuji svému vedoucímu práce PhDr. Josefu Procházkovi, Ph.D. za pomoc s vytvořením zadání, laskavé vedení a neuvěřitelnou vstřícnost po dobu tvorby práce. Dále děkuji svému otci Ing. Michalu Hartmanovi, MBA za podporu po celou dobu studia, a to i v komplikovaných časech. Děkuji také svým kolegům z Accenture, kteří mi umožnili se věnovat studiu a pomohli mi zastat mé pracovní povinnosti. Rovněž děkuji své přítelkyni a životní partnerce Ing. Anně Vychytilové za podporu v závěrečné fázi před odevzdáním práce, bez které by její dokončení nebylo možné.

ABSTRAKT

Práce pojednává o nejpoužívanějších paradigmatech pro vývoj aplikací, jmenovitě o paradigmatu procedurálním, objektově orientovaném a funkcionálním.

První kapitola obsahuje stručnou historii vývoje aplikací s ohledem na používaná paradigmata.

Druhá kapitola představuje jejich hlavní principy a konstrukty, na kterých jednotlivá jsou jednotlivá paradigma založená. U složitějších z nich představuje jednoduché případy použití.

Dále práce obsahuje analýzu existujících projektů, které využívají představená paradigma a modelovou implementaci jednoduché aplikace v těchto paradigmatech.

Obsahem celé práce je porovnání popisovaných paradigmat a rozdílů mezi použitými principy a konstrukty. Z pohledu výhod a nevýhod pro různé typy úloh popisovaná paradigmata porovnává čtvrtá kapitola.

Ze závěrů práce vyplývá, že pro jednodušší projekty menšího rozsahu se nejlépe hodí procedurální paradigma, pro větší a složitější projekty je vhodnější objektově orientované paradigma. Funkcionální paradigma je použitelné pro libovolný typ projektu, ale vyžaduje větší zkušenosti a míru abstrakce vývojáře, který daný projekt implementuje.

KLÍČOVÁ SLOVA

Paradigma, programování, vývoj aplikací, procedurální, objektové, funkcionální

ABSTRACT
This thesis deals with comparison of major paradigms for application development, namely procedural, object oriented and functional paradigm.

First chapter consists of brief history of application development with emphasis on the paradigms being used.

Second chapter introduces main principles and constructs, which are these paradigms based on. For some of which, especially the more complicated ones, examples are included to facilitate easier understanding of the subject.

The thesis also contains analysis of existing projects, which are using described paradigms and model implementation of simple application using each of given paradigms.

This whole thesis focuses on comparison of the described paradigms and differences among used principles and constructs. The advantages and disadvantages of analyzed paradigms for different types of projects are discussed in chapter four.

Based on conclusions of this paper, it is best to use procedural paradigm for development of simple and small projects. For more complex projects of larger scale, it is better to use object-oriented paradigm. Functional paradigm could be used for any type of project, but it has higher requirements for the experience and level of abstraction of the developer who implements the project.

KEYWORDS
Paradigm, programming, application development, procedural, object oriented, functional

Obsah

Úvod 8

1 Historie vývoje aplikací 9

2 Přehled nejpoužívanějších paradigmat 10

2.1 Procedurální paradigma 10

Příkaz 10

Procedura 10

2.2 Objektově orientované paradigma 10

Třída 11

Objekt 11

Dědičnost 11

Zapouzdření 12

Polymorfismus 12

Přetěžování metod (method overloading) 12

Přepisování metod (method overriding) 12

2.3 Funkcionální paradigma 13

Čisté funkce (pure functions) 13

Žádné vedlejší efekty (no side effects) 13

Neměnitelné datové struktury (immutable data structures) 13

Rekurze (recursion) 14

Funkce vyššího řádu (higher order functions) 14

Currying 14

Statická typová kontrola (static type checking) 15

Líné vyhodnocování (lazy evaluation) 15

3 Využití různých paradigmat v existujících projektech 16

3.1 SXML 17

3.2 Webový prohlížeč Minx 22

3.3 ShellCheck 24

4 Porovnání výhod a nevýhod vybraných paradigmat pro různé typy úloh 26

Procedurální paradigma 27

Objektově orientované paradigma 27

Funkcionální paradigma 28

5 Modelová implementace aplikace v různých paradigmatech 29

Jazyk C – procedurální implementace 29

Metoda main 29

Procedura loadStudents 30

Procedura parseStudentFromLine 30

Procedura provideMeans 31

Procedura sortStudensByMeans 31

Procedura swapStudents 32

Procedura printStudents 32

Jazyk Java – objektově orientovaná implementace 33

Třída Main 33

Třída Classroom 34

Konstruktor 35

Metoda importStudentsFromFile 35

Metoda sortStudentsByMeans 36

Třída compareStudentsByMeans 36

Metoda printStudents 36

Třída Student 37

Konstruktor 37

Metody setName a getName 38

Metoda addGrade 38

Metoda getMean 38

Metoda toString 39

Jazyk Haskell – funkcionální implementace 40

Inicializace programu a načtení vstupních dat 40

Výpočet aritmetického průměru 40

Řazení studentů dle průměru 40

Výpis studentů na výstup 41

Závěr 42

Seznam použitých informačních zdrojů 43

Úvod

Počátky vývoje aplikaci (respektive programování) sahají až do poloviny minulého století. Za tu dobu vznikla řada programovacích jazyků a také mnoho paradigmat.

Zatímco atributy programovacího jazyka jsou syntax, standardní knihovna a další nástroje umožňující vývojáři řešit určitý problém, paradigma popisuje způsob řešení daného problému. Přestože některé jazyky jsou vhodnější pro použití s určitým paradigmatem a pro jiné se příliš nehodí, řadu jazyků lze použít pro implementaci programu za použití i více různých paradigmat.

Přestože paradigma je tedy zejména o způsobu myšlení a s programovacím jazykem souvisí jen nepřímo, tak se tato práce věnuje porovnání paradigmat v různých jazycích, v závislosti na použití dané kombinace jazyka a paradigmatu.

Paradigmata vývoje aplikací se dělí do dvou hlavních skupin – imperativních a deklarativních. Zatímco při použití imperativních paradigmat za použití programovacího jazyka říkáme počítači, co a jak má dělat, při použití deklarativních paradigmat definujeme pouze požadovaný výsledek.

Práce se zabývá dvěma imperativními paradigmaty (procedurálním a objektově orientovaným), jedním deklarativním paradigmatem (funkcionálním) a jejich vzájemným porovnáním.

1 Historie vývoje aplikací

Implementace vůbec prvních algoritmů se datuje do 19. století (Wexelblat, 1981), kdy se vyjadřovali rovnou strojovým kódem, a to většinou na děrné štítky nebo pásky. Tyto jednoúčelové programy byly nejdříve používány pro zefektivnění výroby na tkacích strojích, později pak ke sčítání lidu v USA, a nakonec i v armádě při vývoji první jaderné bomby. Tyto stroje však byly jednoúčelové a ve srovnání s dnešními počítači byly velmi primitivní.

To se změnilo v první polovině 20. století, kdy Alan Turing vynalezl Turingův stroj – první univerzální programovatelný počítač, jehož vynález měl velký podíl na obratu vývoje druhé světové války a jehož model se v teoretické informatice používá dodnes (Campbell-Kelly, a další, 2004).

Od vynálezu Turingova stroje uplynulo ještě 20 let, než byl představen první moderní programovací jazyk – FORTRAN. Zdrojový kód v tomto jazyce už využíval konceptu klíčových slov (v angličtině), parametrů, funkcí a byl používán pro vývoj aplikací v imperativním paradigmatu. Nejčastěji byl (a stále je) používán pro numericky náročné matematické výpočty.

Jen o rok později byl představen programovací jazyk LISP, jenž se stal zástupcem funkcionálního paradigma a nejčastěji byl využíván pro implementace simulací umělé inteligence. Jako první přišel s abstraktními datovými strukturami, dynamickým typování a funkcemi vyššího řádu.[1]

Objektově orientované paradigma bylo představeno o pár let později s jazykem Simula (Stefik, et al., 1986), který jak už název napovídá, byl využíván zejména k simulacím, modelování procesů a algoritmů.

Procedurální paradigma bylo na počátku historie vývoje aplikací patrně nejpoužívanějším, a to díky jeho přímočarosti a výkonnosti aplikací, v něm implementovaných. Později, s potřebou vývoje složitějších aplikací, se stalo velmi používaným a moderním paradigma objektově orientované. Umožňovalo totiž snadné modelování architektury aplikace, izolaci dílčích komponent a tím rozdělení vývoje mezi mnoho vývojářů.

[1] Funkce vyššího řádu je taková funkce, která konzumuje jako parametr jednu nebo více funkcí, případně vrací funkci jako návratovou hodnotu (Contracts for higher-order functions, 2002).

Funkcionální paradigma bylo historicky využíváno méně především z důvodu menší výkonnosti takto implementovaných programů. To se ale poslední dobou začíná měnit díky efektivnějším kompilátorům a výkonnějšímu hardwaru.

Existují i další paradigmata pro vývoj aplikací, například logické, strukturované, flow-based a další, těmi se ale tato práce nezabývá.

2 Přehled nejpoužívanějších paradigmat

Následující kapitola popisuje základní principy a stavební bloky procedurálního, objektově orientovaného a funkcionálního paradigma.

2.1 Procedurální paradigma

Procedurální paradigma je postavené na využití procedur, jejich přepoužívání a kompozici. Oproti čistě imperativnímu paradigmatu, které staví na pouhé posloupnosti příkazů, umožňuje přepoužití stejného zdrojového kódu na více místech v programu a dává vývojáři možnost program lépe organizovat (Govender, 2010). Procedury rovněž mohou být vyvíjeny a testovány nezávisle na sobě.

Příkaz

Příkaz je nejmenší samostatný prvek programu vyjadřující nějakou činnost, která má být provedena – například provedení výpočtu, změnu hodnoty proměnné, nebo výpis textu na obrazovku (Seema Kedar, 2009).

Procedura

Procedura je blok programu, který zajišťuje spuštění předem definovaných příkazů v daném pořadí. Každá procedura může být volána (spuštěna) z jakéhokoliv místa programu, včetně jiných procedur. V kontextu programování jsou často zaměňovány termíny procedura a funkce. Formálně oproti proceduře musí mít funkce navíc návratovou hodnotu. Pro účely této práce není třeba tyto termíny rozlišovat.

2.2 Objektově orientované paradigma

Základem objektově orientovaného paradigma je použití objektů a systém předávání zpráv mezi těmito objekty. Objekt je samostatně implementovatelná a testovatelná funkční jednota

programu. Může mít vlastní metody a pole, jejichž obsah reprezentuje stav, ve kterém se daný objekt nachází. Využití objektů umožňuje větší abstrakci, než je možná v případě využití procedurálního paradigma a tím dává vývojáři možnost modelovat architekturu programu dle reálného světa.

Třída

Třída je model pro množinu objektů, na kterých je objektově orientované paradigma založené. Definuje, jaké metody a pole budou dané objekty obsahovat, jejich názvy, parametry, a jejich přístupnost. Aby mohl být vytvořen objekt, musí nejdříve existovat odpovídající třída, která objekt definuje (Eckerdal, et al., 2005).

Objekt

Objekt je instancí třídy a základní funkční jednotkou programu implementovaném v objektově orientovaném paradigmatu. Na základě definice třídy má určité, jemu vlastní, funkce, kterým říkáme metody a také proměnné, respektive pole. Aktuální hodnota proměnných daného objektu určuje jeho stav.

Na základě nastavení přístupnosti těchto funkcí a polí rozlišujeme, jestli k nim může přistupovat (číst je a modifikovat), pouze daný objekt nebo jsou přístupná i z jiných částí programu.

Polím a metodám, ke kterým může přistupovat pouze daný objekt říkáme privátní. Metodám dostupným z jiných částí programu říkáme veřejné a společně tvoří rozhraní (API) daného objektu, respektive třídy. Proměnné se zpravidla jako dostupné mimo daný objekty nevystavují. Některá pole a metody mohou být sdílená mezi více objekty, o takových říkáme, že jsou statická (Bloch, 2008).

Dědičnost

Součástí objektově orientovaného paradigma je koncept dědičnosti – možnost definice vztahu dvou tříd jako rodiče a potomka. Potomek v takovém případě dědí všechny vlastnosti (metody a proměnné) rodiče, ke kterým může volitelně přidávat libovolné další, a dokonce upravovat chování zděděných metod (viz níže).

Dědičnost silným nástrojem pro pokročilou abstrakci architektury programu, v některých případech však může vést k nečekaným chybám, kterým je třeba předcházet řádnou dokumentací. Taková dokumentace by v případě, že je daná třída navržena pro rozšiřování formou dědičnosti, měla obsahovat i nezbytné informace o implementaci dané třídy, které jsou v jiných případech naopak nežádoucí. Pokud třída pro rozšiřování navržena a zdokumentována není, doporučuje se raději využít kompozice.

Zapouzdření

Zapouzdření je seskupení souvisejících metod a polí do jedné třídy, které spolu s nastavením přístupnosti umožňuje programátorovi využívajícímu danou třídu soustředit se pouze na řešení svého problému a odstínit se od implementace dané třídy, podpůrných metod a ostatního neveřejně přístupného zdrojového kódu. Obecně se doporučuje jako veřejné vystavovat minimální možné rozhraní potřebné pro kompletní práci s danou třídou.

Polymorfismus

Polymorfismus je vlastnost, která umožňuje určitou metodu volat s různými parametry nebo změnit chování dané metody. Rozlišujeme dva druhy polymorfismu – přetěžování metod (method overloading) a přepisování metod (method overriding).

Přetěžování metod (method overloading)

Přetěžování metod dovoluje existenci více metod se stejným jménem, které se liší pouze signaturou. Odlišnou signaturou myslíme rozdílný počet nebo ty parametrů dané metody. Která z daných metod bude v době běhu programu spuštěna určují parametry, se kterými je volána. O výběru konkrétní metody rozhoduje kompilátor v době překladu zdrojového kódu na strojový (spustitelný) kód.

Přepisování metod (method overriding)

Přepisování metod umožňuje v případě využití dědičnosti třídě, která je potomkem, deklarovat odlišnou implementaci funkce, než je ta, kterou daná třída zdědila od své

rodičovské třídy. Přepisování funkcí se provádí deklarací ve třídě potomka a poskytnutím příslušné implementace. Ve výběru konkrétní metody, která bude spuštěna a vykonána, se rozhoduje až v době běhu programu.

2.3 Funkcionální paradigma

Funkcionální paradigma se od předešlých dvou představených paradigmat zásadně liší. Jde o paradigma deklarativní, což znamená, že zdrojový kód implementace aplikace definuje, co má být výsledkem operace, ale nepředepisuje, jakým způsobem má být výsledku dosaženo (Ford, 2014). Veškerá logika je v projektech využívajících funkcionální paradigma realizována funkcemi a jejich kompozicí. Pokud povaha projektu vyžaduje práci s stavem, pak je tento držen samostatně, mimo funkční logiku zbytku aplikace.

Čisté funkce (pure functions)

Základním prvkem funkcionálního paradigma jsou funkce. Funkcionální paradigma navíc vyžaduje, aby všechny funkce byly takzvaně čisté. Čistá funkce (pure function) je taková funkce, která při volání se stejnou uspořádanou n-ticí argumentů vždy vrátí stejný výsledek. Aby byl tento výsledek vždy zaručen, nesmí mít čistá funkce žádné vedlejší efekty.

Žádné vedlejší efekty (no side effects)

Vedlejší efektem funkce rozumíme takovou akci, která modifikuje hodnotu statické proměnné, proměnné definované mimo danou funkci, hodnotu proměnné, jejíž reference byla dané funkci předána jako argument nebo nějaký externí vstupní proud. Výše popsané chování je v případě funkcionální paradigma nepřípustné a může vést k neočekávanému chování nebo dokonce i pádu aplikace.

Neměnitelné datové struktury (immutable data structures)

Ve funkcionálním paradigma nelze upravovat hodnotu žádných proměnných. V případě potřeby modifikace a použití proměnné s upravenou hodnotou je na místo toho vytvořena nová instance dané proměnné s požadovanou hodnotou a použita místo původní proměnné. Tím je zabráněno skrytým vedlejším efektům, které by jinak pro aplikaci mohly mít fatální následky.

Rekurze (recursion)

Rekurzivní funkce je taková funkce, která volá sama sebe (Rubio-Sanchez, 2018). Tímto způsobem lze efektivně nahradit použití cyklu, což je ve funkcionálním paradigmatu obzvláště důležité, neboť je možné tímto způsobem skrýt změny stavu. Při použití funkcionálního paradigma jsou proto cykly nahrazovány právě rekurzí.

Funkce vyššího řádu (higher order functions)

Funkce vyššího řádu je taková funkce, která splňuje alespoň jednu z následujících podmínek:

- Jedním nebo více z jejích argumentů je funkce
- Její návratová hodnota je funkce

Ve funkcionálním paradigma se často využívá přístupu, kdy jsou složitější funkce složeny z více jednodušších funkcí. Tyto jednodušší funkce často zajišťují určitou jednoduchou obecnou funkčnost (například procházení seznamu, filtrování atd.), která využívá několik složitějších funkcí vyššího řádu. Tím je zajištěna opětovná použitelnost kódu a zamezeno opakování stejného kódu ve více funkcích. Jednodušším funkcím, které jako argument ani návratovou hodnotu nemají jinou funkci, říkáme funkce prvního řádu.

Currying

Currying je technika, při jejímž použití je definována funkce, která pro výpočet finální návratové hodnoty potřebuje několik argumentů, které jsou ale funkci zadávány postupně po jednom a v každém mezikroku daná funkce vrací funkci, která vykonává část operace.

Tento, poněkud abstraktní, koncept ilustruje následující ukázka v jazyce JavaScript.

```
function soucet (a) {
  return function (b) {
    return a + b;
  }
}

var prictiSest = soucet(6);

pristiSest(4); //vrací návratovou hodnotu 10
```

V ukázce výše je nejdříve definována funkce *soucet*, který má jeden argument. Tato funkce vrací jako návratovou hodnotu další funkci, která má opět jeden argument a až tato funkce vrací jako návratovou hodnotu součet čísel, které byly postupně zadávány.

Statická typová kontrola (static type checking)

Typová kontrola je proces, který vyhodnocuje, jestli typ proměnné, argumentu nebo návratové hodnoty funkce odpovídá stavu očekávanému vývojářem. Pokud taková kontrola probíhá v době překladu zdrojového kódu do strojového kódu (tedy v době kompilace), pak jde o statickou typovou kontrolu. Pokud je tato kontrola prováděna až v době běhu programu, jde o kontrolu dynamickou. Kontrola v době překladu pomáhá odladit část chyb už během vývoje aplikace (Chauhan, 2013). Tyto by jinak byly nalezeny až při testování.

Statická typová kontrola pro uplatnění funkcionálního paradigma není nutná, ale vzhledem k využití častého přetypování je z výše uvedeného důvodu velmi vhodná. Z tohoto důvodu ji například programovací jazyk Haskell, který byl vyvinutý speciálně pro vývoj za použití funkcionálního paradigma, podporuje již v základu.

Některé jazyky, které podporující více paradigmat, přidávají pro usnadnění využití funkcionálního paradigma statickou typovou kontrolu pomocí knihoven. Například pro jazyk JavaScript je možné využít knihovnu flow[2].

Statickou typovou kontrolu často uplatňují i jazyky, které nepodporují funkcionální paradigma, například C a Pascal. Jedním z dalších důvodů pro využití statické typové kontroly je totiž lepší výkonnost aplikace při běhu, kdy už není nutné typy kontrolovat (doba překladu je naopak pomalejší).

Líné vyhodnocování (lazy evaluation)

Líné nebo také odložené vyhodnocování je vyhodnocovací strategie, jejímž účelem je optimalizace doby běhu programu. Využívá zákonů logiky v situacích, kdy není třeba znát (tudíž ani vyhodnocovat, respektive počítat) všechny termy[3] logického výrazu ke správnému vyhodnocení jeho hodnoty.

[2] https://flow.org/

[3] Term je individuální proměnná nebo výsledek logické operace reprezentující logickou hodnotu.

Nejčastěji se líné vyhodnocování využívá v logických výrazech obsahujících konjunkci nebo disjunkci, což ilustruje ukázka níže.

```
function a() {
    //nějaký výpočet
    return false;
}

function b() {
    //nějaký výpočet
    return false;
}

function c() {
    //nějaký výpočet
    return true;
}

if ( a() && b() ) {...} //bude volána jen jedna z funkcí

if ( a() || c() ) {...} //bude volána jen funkce c()
```

Ukázka výše je velmi triviální, v praxi se tato technika nejlépe uplatní u složitějších výrazů, který obsahuje několik podvýrazů, z nichž se vyhodnotí vždy jen ty, které jsou zrovna potřeba pro celkový výsledek.

Použití této techniky je možné pouze v případě, že funkce použité v logickém výrazu nemají žádné vedlejší efekty. Pokud by nějaké vedlejší efekty měly, bylo by třeba vždy volat všechny funkce, případně zajistit vykonání příslušných vedlejších efektů jinou technikou, jinak by se výsledný program choval nedeterministicky.

3 Využití různých paradigmat v existujících projektech

Tato kapitola se zabývá identifikací využití výše popsaných paradigmat v existujících projektech a zkoumá jejich vhodné využití. Jako zkoumané projekty autor zvolil minimalistický parser SXML, mobilní webový prohlížeč Minx a nástroj pro statickou kontrolu Bash skriptů ShellCheck.

3.1 SXML

SXML je minimalistický XML parser implementovaný v programovacím jazyce C (norma ANSI C89) (Caprino, 2014). Rozsah implementace je přibližně 400 řádků zdrojového kódu, který v kontextu této práce poslouží jako ukázka použití procedurálního paradigma. Jazyk C byl pro použití procedurálního paradigma navržen a pohodlné použití jiného z paradigmat, kterým se tato práce věnuje, ani neumožňuje.

Zpracovávání (parsování) dat pomocí SXML se zahajuje voláním funkce sxml_parse(), která jako argumenty zpracovává ukazatel na vlastní datovou strukturu sxm_t, ukazatel na vstupní data ve formátu XML a další pomocné parametry. Výstupem funkce je informace o výsledku operace. V případě úspěchu jsou XML elementy přístupné pro další zpracování ve vstupně výstupním argumentu tokens.

Hlavička funkce sxml_parse:

```
sxmlerr_t sxml_parse(sxml_t *parser, const char *buffer, unsigned bufferlen,
sxmltok_t* tokens, unsigned num_tokens);
```

Argumenty funkce:

Jméno argumentu	Typ argumentu	Účel argumentu
parser	ukazatel na strukturu sxml_t	Pomocná datová struktura sloužící k udržení stavu parseru
buffer	ukazatel na konstantní pole znaků	Pole se vstupními daty pro parser ve formátu XML
bufferlen	neznaménkové celé číslo	Délka vstupních dat
tokens	ukazatel na strukturu sxmltok_t	Vstupně výstupní argument s datovou strukturou pro XML tokeny reprezentující vstupní XML strukturu
num_tokens	neznaménkové celé číslo	Počet tokenů ve struktuře tokens

Datové struktury sxml_t, sxml_tok a jejich použití mohou připomínat třídy a objekty popsané v předchozí kapitole této práce, dokonce jsou tak autorem programu SXML

nazývány v dokumentaci. Jedná se však o triviální datové struktury bez zapouzdření a dalších charakteristik specifických pro OOP, které slouží pouze ke zjednodušení práce s daty, která program SXML zpracovává a ke zpřehlednění zdrojového kódu. Stejným způsobem je možné seskupit například argument buffer s argumentem bufferlen a argument tokens a argumentem num_tokens.

Seskupení argumentů je v těle programu, patrně z důvodů popsaných výše skutečně použito – autor programu používá strukturu sxml_args_t k předávání všech argumentů mezi dalšími funkcemi programu.

```
        sxml_t temp= *state;
        const char* end= buffer + bufferlen;

        sxml_args_t args;
        args.buffer= buffer;
        args.bufferlen= bufferlen;
        args.tokens= tokens;
        args.num_tokens= num_tokens;
```

Další funkce a jejich volání během zpracování vstupních dat znázorňuje graf níže. Z něj je patrné, že veškeré zpracování dat zajišťuje několik málo funkcí. V první fázi program zpracuje kořenovou část XML struktury, ve druhé fázi zbývající část vstupních dat. Ve které fázi se program nachází zjišťuje pomocí maker a pomocné proměnné, ve které si program uchovává svůj vnitřní stav.

```
#define ROOT_FOUND(state)  (0 < (state)->taglevel)
#define ROOT_PARSED(state)((state)->taglevel == 0)
```

Z formálního hlediska jde o deterministický konečný automat, který přijímá jazyk definovaný standardem XML, vnitřní stav si udržuje v pomocné proměnné a přechodové funkce k určení následujícího stavu využívají tento stav a právě jeden následující znak ze vstupních dat.

```
while (!ROOT_FOUND (&temp))
        {
                sxmlerr_t err;
                const char* start= buffer_fromoffset (&args, temp.bufferpos);
```

```
		const char* lt= str_ltrim (start, end);
		state_setpos (&temp, &args, lt);
		state_commit (state, &temp);

		if (end - lt < TAG_MINSIZE)
			return SXML_ERROR_BUFFERDRY;

		/* --- */

		if (*lt != '<')
			return SXML_ERROR_XMLINVALID;

		switch (lt[1])
		{
		case '?':	err= parse_instruction (&temp, &args);	break;
		case '!':	err= parse_doctype (&temp, &args);	break;
		default:	err= parse_start (&temp, &args); break;
		}

		if (err != SXML_SUCCESS)
			return err;

		state_commit (state, &temp);
	}
```

Druhá část programu, která zpracovává zbytek XML struktury vypadá na první pohled velmi podobně, část logiky je dokonce totožná, bylo by tedy patrně možné tyto dvě části kódu sloužit a upravit logiku, případně některé části kódu vyčlenit do samostatných funkcí a přepoužít v obou částech kódu. Vzhledem k malé složitosti a rozsahu kódu autor práce považuje i současný kód za kvalitní, a to s důrazem na přehlednost, které bylo právě tímto rozdělení dosaženo.

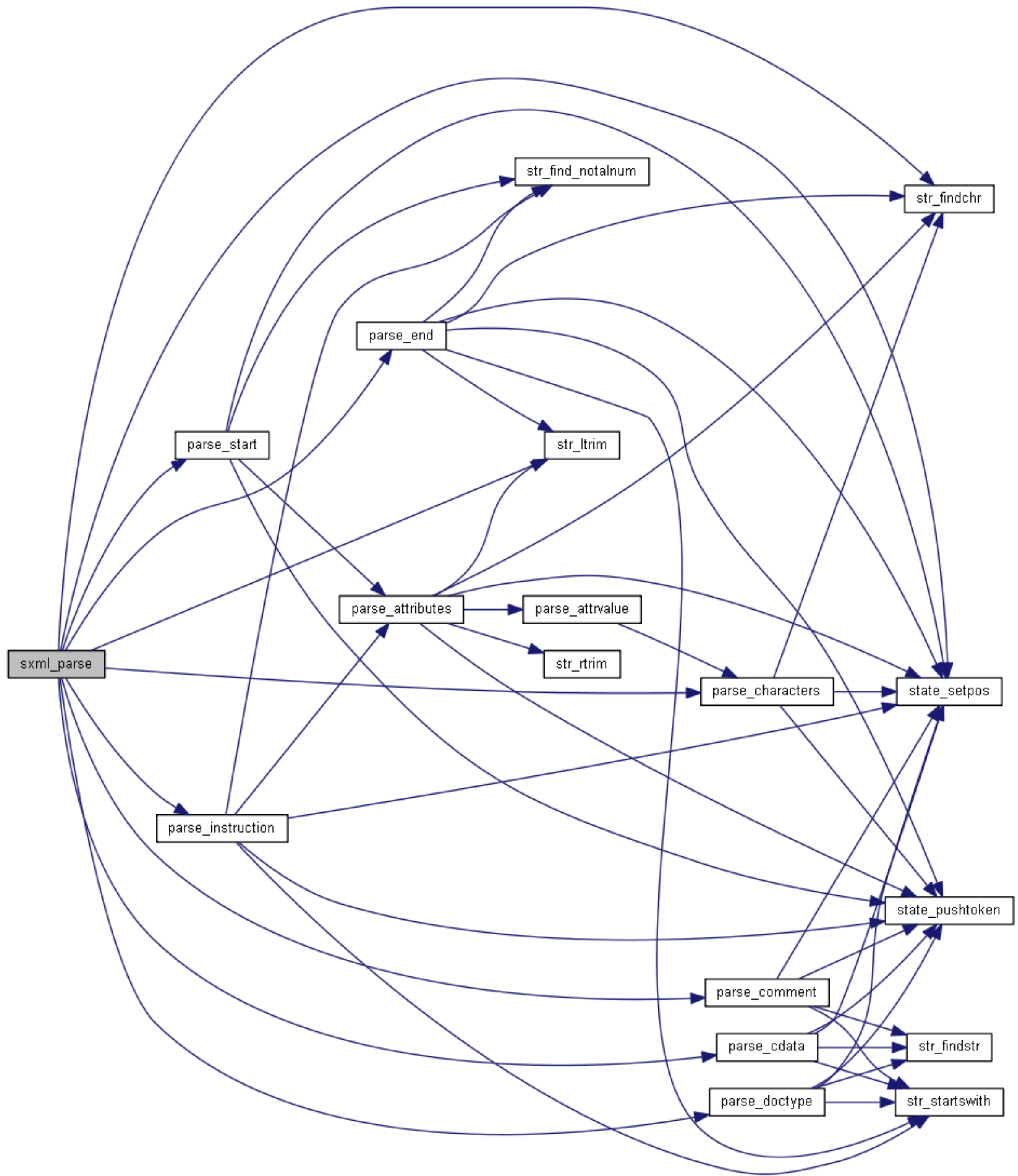
```
while (!ROOT_PARSED (&temp))
	{
		sxmlerr_t err;
		const char* start= buffer_fromoffset (&args, temp.bufferpos);
		const char* lt= str_findchr (start, end, '<');
		while (buffer_fromoffset (&args, temp.bufferpos) != lt)
		{
			sxmlerr_t err= parse_characters (&temp, &args, lt);
			if (err != SXML_SUCCESS)
				return err;

			state_commit (state, &temp);
		}
```

```
        /* --- */

        if (end - lt < TAG_MINSIZE)
                return SXML_ERROR_BUFFERDRY;

        switch (lt[1])
        {
        case '?':       err= parse_instruction (&temp, &args);
                        break;
        case '/':       err= parse_end (&temp, &args);
                        break;
        case '!':       err= (lt[2] == '-') ? parse_comment (&temp, &args)
                            : parse_cdata (&temp, &args);
                        break;
        default:        err= parse_start (&temp, &args); break;
        }

        if (err != SXML_SUCCESS)
                return err;

        state_commit (state, &temp);
}
```

Patrně nejefektivnějším způsobem pro grafické znázornění základní struktury programu využívajícího procedurální paradigma je graf volání funkci (call graph). Je z něj názorné, jaké funkce jsou v programu implementované a rovněž základní flow[4] programu.

Graf níže byl vygenerován dokumentačním nástrojem Doxygen (Heesch, 2018) znázorňuje jeden průběh volání funkce sxml_parse(). Je z něj patrné, že o zajištění běhu programu se stará jen několik málo funkcí, z nichž některé mají na starost řízení běhu programu a aktualizaci jeho vnitřního stavu, další slouží k rozhodování jakým způsobem má být zpracována další část vstupních dat, a nakonec jsou data zpracována a proces začne znovu. Tento proces se opakuje, dokud jsou na vstupu ještě nějaká nezpracovaná data.

[4] V kontextu vývoje aplikací se jako flow nebo také control flow označuje pořadí volání funkcí a vyhodnocování výrazů za běhu daného programu.


str_find_notalnum
str_findchr
parse_end
parse_start
str_ltrim
parse_attributes
parse_attrvalue
str_rtrim
sxml_parse
parse_characters
state_setpos
parse_instruction
state_pushtoken
parse_comment
parse_cdata
str_findstr
parse_doctype
str_startswith

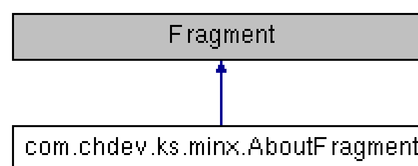
3.2 Webový prohlížeč Minx

Minx je minimalistický webový prohlížeč pro Android pro zobrazování textového obsahu webu (Sharma, 2015). Projekt je realizován v jazyce Java a principy objektově orientovaného paradigma utilizuje také při použití standardní knihovny (například dědičnost).

Jedním ze základních principů jazyka Java je, že všechny třídy jsou potomky java.lang.Object, čímž je zajištěno, že všechny objekty disponují několika stejnými základními metodami. Jedná se například o metody equals() a hashCode() (Bloch, 2008). Existence (a správná implementace) těchto metod zajišťuje (mimo jiné) možnost ukládání objektů v datových strukturách, které jsou poskytovány jako součást standardní knihovny.

V programu Minx je tohoto principu také využito. Součástí kolekce Android Jetpack je třída Fragment, která reprezentuje uživatelské rozhraní nebo jeho část a jeho chování. Uživatelské rozhraní webového prohlížeče Minx je kompletně zrekonstruované z objektů, které jsou potomky třídy Fragment, čímž je zajištěna jednotnost rozhraní, efektivní vývoj a aktualizace i přehlednost kódu.

Příkladem budiž třída AboutFragment, která reprezentuje obrazovku About se základními údaji o aplikace a jejím autorovi. Grafická reprezentace níže znázorňuje výše popsaný vztah tříd AboutFragment a Fragment, totiž že třída AboutFragment je přímým potomkem třídy fragment.



Ve zdrojovém kódu je tento vztah popsán klíčovým slovem extends. Díky využití principů objektově orientovaného paradigma celá implementace obrazovky About zabírá jen necelý 60 řádků zdrojového kódu.

```
public class AboutFragment extends Fragment {

    Button btnGithub, btnGooglePlus, btnMail;
    ListView listViewLibrary;

    @Nullable
```

```
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_about, container, false);
    }

    @Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
      /* tělo implementace metody onActivityCreated */
    }
}
```

Ze zdrojového kódu výše je dále zřejmé, že se na obrazovce About nachází tři tlačítka, btnGithub, btnGooglePlus, btnMail a že třída Fragment obsahuje metody onCreateView() a onActivityCreated(), které třída AboutFragment přepisuje a chování metod rodiče nahrazuje vlastní implementací. Informace o přepisování metod je zde znázorněna anotací @Override. Ta slouží jak vývojářům pro rychlou orientaci v kódu, tak překladači k verifikaci úmyslu autora daného kódu.

V praxi se může stát, že vývojář zamýšlí nějakou metodu přepisovat, ale ve skutečnosti ji přetíží, kvůli chybné specifikaci signatury dané metody. V takovém případě je pak vývojář na tuto skutečnost překladačem laskavě upozorněn.

Přepisování metody onActivityCreated() je zde použité právě kvůli tlačítkům na obrazovce About, respektive kvůli konfiguraci jejich chování. Příkladem budiž konfigurace tlačítka btnGithub, které slouží k předání informace o umístění repozitáře zdrojových kódů aplikace Minx na portálu Github.

```
@Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        btnGithub= (Button) getView().findViewById(R.id.btn_github);
        btnGooglePlus= (Button) getView().findViewById(R.id.btn_google_plus);
        btnMail= (Button) getView().findViewById(R.id.btn_mail);

        btnGithub.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent=new Intent(Intent.ACTION_VIEW);

intent.setData(Uri.parse(getResources().getString(R.string.github_website)));
```

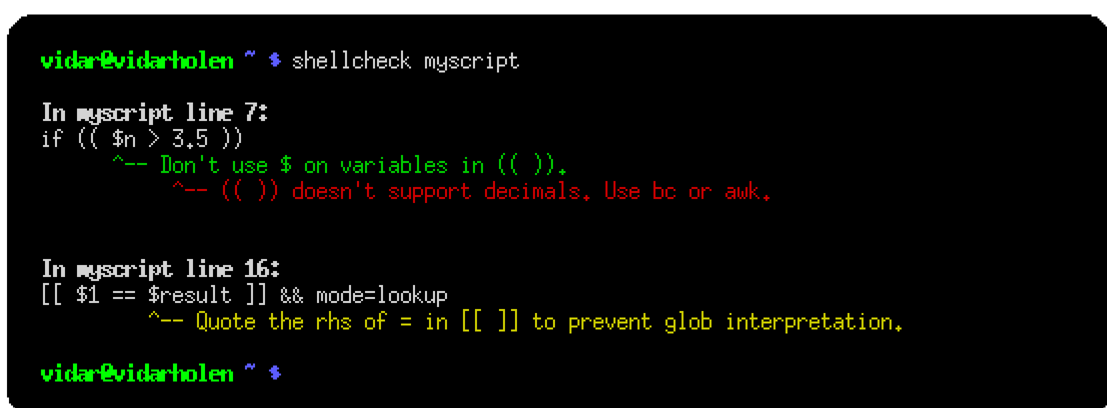
```
                startActivity(intent);
            }
        });
```

Samotné tlačítko btnGithub je instancí třídy Button, která je potomkem třídy TextView a ze stejných důvodů uplatňuje stejné principy, jako třída AboutFragment popsaná výše. Třída TextView je potomkem třídy View. Podobné vztahy a principy jsou v implementaci celé aplikace a použitých knihoven velmi hojně využívané, což svědčí o vhodném použití objektově orientovaného paradigma.

3.3 ShellCheck

ShellCheck je nástroj pro statickou analýzu skriptů napsaných v jazyce Bash. Jazyk Bash je jeden z nejpoužívanějších jazyků používaných příkazovými řádkami v operačních systémech Linux a Unix.

Cílem projektu ShellCheck je analyzovat skript napsaný v jazyce Bash a označit potenciálně problémové části kódu, včetně návrhu řešení. Nezkušeným a začínajícím uživatelům jazyka Bash může pomoci odladit jejich skript a vyvarovat se typických začátečnických chyb. Zkušenějším uživatelům může pomoci odhalit komplikovanější problémové oblasti, omezení vyplývající z použití kombinace některých funkcí nebo složitějších konstruktů a upozorní na potenciálně nežádoucí chování v mezních případech.



Celý projekt ShellCheck je realizován v jazyce Haskell a uvolněn pod licencí GPLv3. Jeho zdrojové kódy jsou tedy volně k dispozici.

Hlavním modulem aplikace je modul Analyzer, který se stará o přebírání a kontrolu vstupních dat a následnou exekuci další logiky programu, která kontroluji vstupní Bash skript a vyhodnocuje potenciálně problémová místa.

```
analyzeScript :: AnalysisSpec -> AnalysisResult
analyzeScript spec = newAnalysisResult {
    arComments =
        filterByAnnotation spec params . nub $
            runAnalytics spec
            ++ runChecker params (checkers params)
}
  where
    params = makeParameters spec

checkers params = mconcat $ map ($ params) [
    ShellCheck.Checks.Commands.checker,
    ShellCheck.Checks.Custom.checker,
    ShellCheck.Checks.ShellSupport.checker
    ]

optionalChecks = mconcat $ [
    ShellCheck.Analytics.optionalChecks
    ]
```

Z ukázky zdrojového kódu výše je patrné, že program si nedrží žádný vnitřní stav, nevytváří žádné pomocné datové struktury a vše je realizováno funkčním voláním a předáváním argumentů mezi funkcemi.

Jednou z kontrol, kterou program ShellCheck provádí v průběhu analýzy Bash skriptu je kontrola použití kulatých závorek pro vyhodnocení matematického výrazu, místo zastaralé syntaxe s hranatými závorkami. Tato kontrola je realizována funkcí checkDollarBrackets.

```
prop_checkDollarBrackets1 = verify checkDollarBrackets "echo $[1+2]"
prop_checkDollarBrackets2 = verifyNot checkDollarBrackets "echo $((1+2))"
checkDollarBrackets _ (T_DollarBracket id _) =
    style id 2007 "Use $((..)) instead of deprecated $[..]"
checkDollarBrackets _ _ = return ()
```

Další částí kódu, která dobře ilustruje vhodné použití funkcionálního paradigma je kontrola správného použití uvozovek s utilitou Trap, o tu se stará funkce checkTrapQuotes.

```
prop_checkTrapQuotes1 = verify checkTrapQuotes "trap \"echo $num\" INT"
prop_checkTrapQuotes1a= verify checkTrapQuotes "trap \"echo `ls`\" INT"
```

```
prop_checkTrapQuotes2 = verifyNot checkTrapQuotes "trap 'echo $num' INT"
prop_checkTrapQuotes3 = verify checkTrapQuotes "trap \"echo $((1+num))\" EXIT
DEBUG"
checkTrapQuotes = CommandCheck (Exactly "trap") (f . arguments) where
    f (x:_) = checkTrap x
    f _ = return ()
    checkTrap (T_NormalWord _ [T_DoubleQuoted _ rs]) = mapM_ checkExpansions
rs
    checkTrap _ = return ()
    warning id = warn id 2064 "Use single quotes, otherwise this expands now
rather than when signalled."
    checkExpansions (T_DollarExpansion id _) = warning id
    checkExpansions (T_Backticked id _) = warning id
    checkExpansions (T_DollarBraced id _ _) = warning id
    checkExpansions (T_DollarArithmetic id _) = warning id
    checkExpansions _ = return ()
```

Jednou z výhod funkcionálního paradigma vyplývající z absence stavu jednotlivých komponent aplikace, je možnost jednotlivé funkce jednoduše samostatně testovat (Eekelen, 2007). Projekt ShellCheck této výhody plně využívá, jak je vidět na předchozích dvou ukázkách kódu. Oba bloky kódu obsahují testy dané funkce. Test začínající klíčovým slovem Verify zajišťuje pozitivní validaci, že funkce potenciálně problémové místo ve vyhodnocovaném skriptu jako potenciálně problémové skutečně vyhodnotí. Test začínající klíčovým slovem VerifyNot naopak kontroluje, že korektní části skriptu nejsou chybně vyhodnoceny jako potenciálně problémové.

Projekt ShellCheck je rozdělen do několika modulů, z nichž každá má svou specifickou a izolovanou oblast, jejíž řešení zajišťuje. Zejména se jedná o moduly Analytics, Analyzer, Checker, Data, Fixer, Parser a Regex. Každý z modulů je realizován sadou funkcí, které jsou samostatně testovatelné a jednotkové testy daných funkcí jsou vždy součástí daného modulu. Díky dodržení principů funkcionálního programování a jeho vhodného použití je projekt ShellCheck snadno rozšiřitelný a implementovaná funkčnost je použitelná v jiných projektech pro řešení jiného problému.

4 Porovnání výhod a nevýhod vybraných paradigmat pro různé typy úloh

Tato kapitola se zabývá porovnáním popisovaných paradigmat z pohledu jejich využitelnosti pro různé typy úloh a rozsah projektů. Škála úloh řešitelná za pomocí výpočetní techniky je

obrovská a velkou část těchto úloh je možné řešit různými způsoby, tato kapitola má za cíl popsat vhodnost analyzovaných paradigmat podle obecných parametrů řešené úlohy.

Procedurální paradigma

Z pohledu komplexnosti je nejjednodušším paradigmatem, kterým se tato práce zabývá, paradigma procedurální. Z toho také plynou jeho hlavní výhody.

Nejlépe se hodí pro malé projekty a jednoduché úlohy. Na rozdíl od složitějších paradigmat není třeba pro triviální projekty zavádět zbytečné abstrakce nebo navrhovat robustní architekturu projektu , který by se stejně nikdy nevyužila a akorát by zbytečně prodlužovala a zdražovala vývoj. Nevýhodou, která z výše uvedeného přímo vyplývá pak je obtížná rozšiřitelnost takového projektu a složitější udržovatelnost zdrojového kódu a budoucí modifikace.

Díky své jednoduchosti je také praktické pro využití při tvorbě nových programovacích jazyků pro speciální účely, například vestavěné systémy. Vzhledem k absenci složitějších konstruktů je návrh jazyka využívajícího procedurální paradigma a implementace překladače pro tento jazyk jednodušší, zároveň tedy i rychlejší a levnější než v případě využití komplexnějšího paradigma.

Procedurální paradigma bývá rovněž využíváno pro výuku programování. Využíván k tomu byl například programovací jazyk Pascal – velmi jednoduchý procedurálně orientovaný jazyk, který se studenti učili často s jediným cílem, a to naučit se základům programování a algoritmizace. Později pak nabyté dovednosti využili při studiu nějaké komplexnějšího paradigma nebo složitějšího procedurálně orientovaného jazyka s širšími možnostmi použití.

Objektově orientované paradigma

O něco složitějším paradigmatem, které z procedurálního paradigma s trochu nadsázky vychází, je paradigma objektově orientované. Z pohledu historie se v některých oblastech stalo jeho přímým nástupcem, zavádí složitější konstrukty a řeší hlavní nevýhody procedurálního paradigma.

Jeho použití pro malé projekty je možné, více se však hodí na projekty většího rozsahu nebo takové, u kterých je očekávána modifikace či rozšíření v budoucnu. Pro tvorbu triviálních

projektů se příliš nehodí, protože čas věnovaný návrhu projektu a abstrakci nebude reálně zužitkován.

Hlavním přínosem objektově orientovaného paradigmatu je zavedení konceptu objektu, které bylo inspirováno reálným světem. Velmi dobře se tak hodí na všechny typy úloh, který řeší problém konkrétních předmětů z reálného světa a jejich komunikace.

Praktickou výhodou objektově orientovaného paradigma oproti funkcionálnímu je jeho rozšířenost a dostupnost specialistů, kteří jej umí efektivně využít. Často je totiž vyučováno na školách a přechod z procedurálního paradigma na objektově orientované je výrazně jednodušší než přechod na funkcionální.

Vzhledem k možnostem abstrakce projektu a izolace jeho jednotlivých částí, které objektově orientované paradigma nabízí, je využitelné pro řešení prakticky jakékoliv běžně řešené úlohy a je tak na uvážení tvůrce projektu, jestli zvolí právě objektově orientované paradigma nebo nějaké jiné vyhovující paradigma.

Funkcionální paradigma

Funkcionální paradigma definuje mnoho abstraktních prvků a zavádí řadu omezení. Svou složitostí je z tohoto pohledu srovnatelné s objektově orientovaným paradigmatem, nicméně neinspiruje se svými konstrukty v reálném světě, nýbrž je více abstraktní. Z tohoto důvodu může být komplikovanější na naučení a následné použití.

Z pohledu velikosti projektu je funkcionální paradigma dobře použitelné pro libovolně velké projekty. Funkcionální zápis algoritmu je často kratší, než imperativní a na rozdíl od objektově orientovaného paradigma není třeba vytvářet složitou strukturu projektu, ale stačí psát jen efektivní kód řešící daný problém.

Nevýhodou funkcionálního paradigma může být jeho vysoká úroveň abstrakce a s tím spojená složitost použití (Alvin, 2017). S tím také souvisí menší množství specialistů, kteří toto paradigma umí efektivně využít.

Velmi dobře se však hodí k řešení logických a matematických úloh, z tohoto důvodu je velmi oblíbené v akademické sféře a výzkumu. Zde je jeho použití často naopak jednodušší než použití nějakého imperativního paradigmatu, protože vstupem řešené úlohy může být popis problému blízký funkcionálnímu zápisu, a tak jeho řešení může být spíš otázkou syntaxe než

skutečné algoritmizace. Problém tady totiž skutečně řeší až stroj, což je obecně nespornou výhodou všech deklarativních paradigmat.

5 Modelová implementace aplikace v různých paradigmatech

Jako modelová aplikace pro ukázkou použití analyzovaných paradigmat byl pro účely této práce svolen triviální srovnávač studijních výsledků. Vstupem aplikace je CSV[5] soubor se jmény studentů a jejich studijními výsledky. Aplikace provede analýzu studijních výsledků jednotlivých studentů, spočítá jejich aritmetický průměr, studenty seřadí dle spočteného průměru od nejlepšího po nejhorší a výsledek uživateli vypíše na obrazovku.

Jazyk C – procedurální implementace

Základní struktura modelové implementace v jazyce C přesně kopíruje principy procedurálního paradigma. Po spuštění programu je volána hlavní metoda, ve které jsou deklarovány potřebné proměnné a následně dochází k volání procedur, které se starají o funkční vykonání požadovaného algoritmu.

Metoda main

Metoda main(), která je volána operačním systémem po zavedení aplikace do paměti, jako první kontroluje, že program byl zavolán se správným počtem parametrů a pokud ne, tak vypíše chybovou hlášku a ukončí se.

Dále se metoda pokouší o otevření souboru, jehož název byl programu předán jako parametr, v případě neúspěchu opět vypíše chybovou hlášku a ukončí se.

V případě, že byl program spuštěn korektně, je deklarována datová struktura pro uložení studentů, která je dále předávána procedurám, které zajistí funkční chování programu.

```
int main ( int argc, char ** argv )
 {
      if ( argc < 2 )
       {
```

[5] CSV – comma separated values – definuje formát textového souboru s daty oddělenými čárkou (Shafranovich, 2005)

```
            printf("Error: Not enough arguments\n");
            printf("Usage: %s input_file_name.csv\n", argv[0]);
            return 1;
      }


      FILE * inputFile = fopen ( argv[1], "r" );
      if (inputFile == NULL)
     {
            printf("Error while opening the file.\n %s\n", strerror(errno)
);
            return 2;
     }

      struct Students students[STUDENT_COUNT];
      loadStudents ( students, inputFile );
      provideMeans ( students );
      sortStudensByMeans ( students );
      printStudents ( students );
      fclose(inputFile);
      return 0;
 }
```

Procedura loadStudents

Procedura loadStudents má dva parametry: datovou strukturu students a soubor se vstupními daty. Stará se o načtení dat o studentech do datové struktury programu pro další zpracování. Pro zpracování vstupu používá pomocnou proceduru parseStudentFromLine, které předává pro zpracování jednotlivé řádky, které reprezentují dané studenty.

```
void loadStudents ( struct Students students[STUDENT_COUNT], FILE * inputFile
)
 {
       char line[BUFFER_SIZE];
       for ( int i = 0; i < STUDENT_COUNT; ++i )
       {
             fgets ( line, BUFFER_SIZE, inputFile );
             students[i] = parseStudentFromLine ( line );
       }
 }
```

Procedura parseStudentFromLine

Jedná se o pomocnou proceduru, zajišťuje parsování řádku reprezentujícího studenta a uložení do programové datové struktury, kterou vrací jako svůj výstup.

```
struct Students parseStudentFromLine ( char line[BUFFER_SIZE] )
{
	struct Students student;

	int i = 0;
	for ( ; line[i] != FIELD_SEPARATOR; ++i )
	{
		student.name[i] = line[i];
	}
	student.name[i] = '\0';

	for ( int j = 0; j < EXAM_COUNT; i+=2 )
	{
		student.exams[j++] = line[i+1] - '0';
	}

	return student;
}
```

Procedura provideMeans

Tato procedura provádí datové operace nad programovou strukturou reprezentující studenty a počítá jejich průměr, který rovnou ukládá do dané programové struktury.

```
void provideMeans ( struct Students students[STUDENT_COUNT] )
{
	for (int i = 0; i < STUDENT_COUNT; ++i )
	{
		int sum = 0;
		for ( int j = 0; j < EXAM_COUNT; ++j )
		{
			sum += students[i].exams[j];
		}
		students[i].mean = (double) sum / EXAM_COUNT;
	}
}
```

Procedura sortStudensByMeans

Tato procedura řadí studenty podle průměru spočítaného procedurou provideMeans. Jako řadící algoritmus implementuje bubble sort, který postupně prochází všechny studenty a v případě, že narazí na posloupnost dvou studentů, kteří nejsou seřazeni podle průměrů, pak tyto prohodí. K tomu používá pomocnou proceduru swapStudents. Studenti jsou procházeni ve dvou vnořených cyklech, algoritmus má tedy asymptoticky kvadratickou složitost.

```
void sortStudensByMeans ( struct Students students[STUDENT_COUNT] )
{
		for ( int i = 0; i < STUDENT_COUNT; ++i )
		{
			for ( int j = 0; j < STUDENT_COUNT - 1; ++j )
			{
				if ( students[j].mean >  students[j+1].mean )
				{
					swapStudents ( j, j + 1 , students );
				}
			}
		}
}
```

Procedura swapStudents

Pomocná procedura pro prohození dvou studentů v programové datové struktuře použitá procedurou sortStudentsByMeans popsané výše.

```
void swapStudents ( int idxA, int idxB,  struct Students
students[STUDENT_COUNT] )
{
	struct Students tmp = students[idxA];
	students[idxA] = students[idxB];
	students[idxB] = tmp;
}
```

Procedura printStudents

O výpis výsledku zpracovaného výše popsanými částmi programu se stará procedura printStudents s jediným argumentem, a to datovou strukturou s daty o studentech a jejich výsledcích. Výpis je realizován jednoduchým cyklem, jehož každá iterace zajišťuje výpis informací o jednom studentovy. Tento je realizován dalším vnořeným cyklem, který v každé iteraci vypíše jednu specifickou informaci o daném studentovi. Výpis je proveden na standardní výstup, odkud může být dále přesměrován nástroji operačního systému dle požadavků uživatele.

```
void printStudents (  struct Students students[STUDENT_COUNT] )
{
	for (int i = 0; i < STUDENT_COUNT; ++i )
	{
		printf("%s", students[i].name);
		for ( int j = 0; j < EXAM_COUNT; ++j )
		{
			printf(";%d", students[i].exams[j]);
		}
```

```
            printf(";%.2f\n", students[i].mean);
        }
}
```

Jazyk Java – objektově orientovaná implementace

Implementace modelové aplikace v Jazyce Java reprezentuje typický objektově orientovaný přístup. Flow programu je řízeno ve třídě Main, aplikace dále obsahuje implementaci dvou dalších tříd – Classroom, která reprezentuje klasickou školní třídu se studenty a Student, která reprezentuje jednotlivé studenty. Všechny třídy v jazyce Java dědí od obecné třídy Object; třídy v této implementaci jsou dokonce jejím přímým potomkem, neboť nerozšiřují žádnou jinou třídu.

Třída Main

Je základní třídou programu a v tomto případě obsahuje jedinou metodu, a to veřejnou statickou metodu main, která je volána operačním systémem po zavedení aplikace do paměti. Má stejné rozhraní, jako modelová implementace v procedurálním paradigma, přijímá tedy jeden argument – jméno souboru s daty o studentech.

Po spuštěný vytváří instanci třídy Classroom a volá metodu importStudentsFromFile této instance, které předává jako parametr jméno souboru pro načtení dat o studentech. Dále volá metodu sortStudentsByMeans bez parametrů, a nakonec metodu printStudents, rovněž bez parametrů.

Ve srovnání s procedurální implementací je na první pohled vidět, že díky objektovému přístupu a zapouzdření je vývojář využívající třídu Classroom úplně izolován od datových struktur, které daná třída používá, její vnitřní implementace a je pro něj důležité pouze rozhraní, pomocí kterého se třídou komunikuje.

V jazyce Java je na začátku každé třídy definován balíček, do kterého daná třída patří. V případě všech tříd v této modelové implementaci je to balíček `cz.cuni.pedf.kittv.hartmami`.

```java
package cz.cuni.pedf.kittv.hartmami;

public class Main {

    public static void main(String[] args) {
           Classroom classroom = new Classroom();
           classroom.importStudentsFromFile(args[0]);
           classroom.sortStudentsByMeans();
           classroom.printStudents();
    }
}
```

Třída Classroom

Třída reprezentující skupinu studentů, respektive klasickou školní třídu. Importuje a používá několik tříd ze standardní knihovny Java. Jako rozhraní vystavuje konstruktor bez parametrů, metodu importStudentsFromFile s jedním parametrem pro název souboru a metody sortStudentsByMeans a printStudents bez parametrů. Pro funkční realizaci rovněž obsahuje soukromou strukturu students datového typu List<Students> pro ukládání a manipulaci s daty o studentech. Dále pak soukromou třídu compareStudentsByMeans. Všechny tyto dílčí části třídy Classroom jsou popsány dále v textu.

```java
package cz.cuni.pedf.kittv.hartmami;

import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Vector;

public class Classroom {
    public Classroom();

    private List<Student> students;

    public void importStudentsFromFile(String file);

    private class compareStudentsByMeans implements Comparator<Student>;

    public void sortStudentsByMeans ();

    public void printStudents();
}
```

Konstruktor

Třída Classroom má vlastní implementaci konstruktoru, probíhá v ní jediná operace – inicializace datové struktury pro ukládání dat o studentech. Ta je provedena vytvořením nové instance třídy Vector<Student> a přiřazením do privátní proměnné students.

```
public Classroom() {
    this.students = new Vector<Student>();
}
```

Metoda importStudentsFromFile

Veřejná metoda určená pro použití programátorem využívající třídu Classroom, má jeden argument typu String, ve kterém očekává jméno csv souboru s daty o studentech. Vstupní soubor prochází o řádcích, kde z každého řádku na základě oddělovače identifikuje data pro uložení a předává je k provedení této operace třídě Student – například metoda student.setName nebo student.addGrade.

V souladu s principy objektově orientovaného paradigma tedy ke vnitřním datovým strukturám třídy Student vůbec nepřistupuje, ale nechává tuto práci na metodách třídy Student. Toto je zásadní rozdíl oproti modelové implementaci v procedurálním paradigma.

V případě, že dojde k chybě při čtení vstupního soboru (například pokud soubor neexistuje nebo k němu program nemá dostatečná oprávnění) metoda vyhodí výjimku a vypíše strackTrace, podle kterého uživatel může identifikovat nastalý problém.

```
public void importStudentsFromFile(String file) {

        Path path = Paths.get(file);
        try (BufferedReader bufferedReader = Files.newBufferedReader(path) ) {
            String line;
            while ( ( line = bufferedReader.readLine() ) != null ) {
                String[] attributes = line.split(";");
                Student student = new Student();
                student.setName(attributes[0]);
                for (int i = 1; i < attributes.length; i++) {
                    student.addGrade(attributes[i]);
                }
                students.add(student);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
```

Metoda sortStudentsByMeans

Jednořádková metoda bez parametrů zajišťující řazení studentů dle jejich průměru. Takto krátká a jednoduchá je právě díky použití objektově orientovaného paradigma a použití objektů ze standardní knihovny jazyka Java. Pro řazení využívá třídu Collections a její metodu sort. Jako parametr je jí předávána datová struktura students typu List<Student> a komparátor compareStudentsByMeans. Konkrétní řadící algoritmus použitý metodou sort je v režii autorů standardní knihovny jazyka Java a autorovi modelové implementace stačí zajistit správnou implementaci komparátoru compareStudentsByMeans.

```
public void sortStudentsByMeans () {
        Collections.sort(students, new compareStudentsByMeans());
    }
```

Třída compareStudentsByMeans

Triviální třída implementující rozhraní definované standardní knihovnou jazyka Java Comparator (McCloskey, a další, 2003) a to pro třídu Student. Implementace obsahuje jedinou metodu, veřejnou metodu compare, která jako argumenty přijímá dva studenty a vrací celé číslo určující, jestli má být první student zařazen před druhého nebo mají být studenti prohozeni, případě mají stejný průměr a na jejich pořadí tedy nezáleží. Průměr studentů metoda získává pomocí metody třídy Student getMean. Je tedy izolovaná od vnitřní datové reprezentace třídy student a o výpočet průměru se nemusí starat.

```
private class compareStudentsByMeans implements Comparator<Student> {
        public int compare(Student a, Student b)
        {
            return (int) (a.getMean() - b.getMean());
        }
    }
```

Metoda printStudents

Díky principům objektového paradigma opět velmi jednoduchá, krátká a snadno pochopitelná metoda starající se o výpis studentů na standardní výstup. V jednoduchém cyklu iteruje přes všechny studenty v datové kolekci a v každé iteraci předává metodě println instanci třídy Student. Překladač použije variantu metody println, která má jako argument instanci třídy Object ze standardní knihovny a na této instanci zavolá metodu toString. Jak je popsáno níže v kapitole popisující třídu Student, tato třída implementuje vlastní metodu

toString, čímž přepisuje implementaci této metody z nadřazené třídy Object a tím zajišťuje požadovaný formát výstupu.

```
public void printStudents() {
        for (Student student : students ) {
            System.out.println(student);
        }
    }
```

Třída Student

Tato třída reprezentuje jednotlivé studenty, má několik privátních atributů a vystavuje veřejné rozhraní pro práci s nimi.

```
package cz.cuni.pedf.kittv.hartmami;

import java.util.List;
import java.util.Vector;

public class Student {
    private String name;
    private List<Integer> exams;
    private Double mean;
    private static char FIELD_SEPARATOR = ';';

    public Student();

    public void setName(String name);

    public String getName();

    public void addGrade (String grade);

    public void addGrade (Integer grade);

    public double getMean();

    public String toString();
}
```

Konstruktor

Třída student má vlastní implementaci konstruktoru. V jeho těle jsou vykonány dvě operace – inicializace datové struktury pro známky studenta a inicializace proměnné určené pro studentův průměr. Pro uložení známek studenta je vytvořena nová instance třídy

Vector<Integer>. Studentův průměr je inicializován na hodnotu null (speciální hodnota symbolizující fakt, že studentův průměr ještě není známý).

Metody setName a getName

Standardní metody pro přístup (nastavování a čtení) soukromých atributů třídy, v tom případě atributu name. V anglickém názvosloví má tento typ metod speciální název – setter (pro metody, které nastavují hodnotu atributu) a getter (pro metody, které čtou a vracejí hodnotu atributu).

```
public void setName(String name) {
    this.name = name;
}
```

```
public String getName() {
    return name;
}
```

Metoda addGrade

Tato metoda slouží pro ukládání známek daného studenta do vnitřní datové struktury exams. Metoda je přetížená a v jedné variantě má parametr typu String a ve druhé variantě parametr typu Integer – programátor používající třídu Student si může vybrat, kterou použije, v závislosti na datech, která má k dispozici.

```
public void addGrade (String grade) {
    exams.add(Integer.valueOf(grade));
}

public void addGrade (Integer grade) {
    exams.add(grade);
```

Metoda getMean

Tato metoda zajišťuje výpočet a vrácení studentova průměru. Metoda nejdříve provede kontrolu stavu proměnné mean, která reprezentuje studentův průměr. Pokud je hodnota null (nastavení inicializací v konstruktoru), tak provede výpočet a hodnotu průměru vrátí volajícímu. Pokud už jednou výpočet průměru proběhl, je v proměnné mean jiná hodnota,

než null a metoda rovnou vrátí již dříve vypočítanou hodnotu průměru. Tím je zajištěna optimalizace využití výpočetního výkonu. Jde o běžnou techniku zvanou líné vyhodnocování (lazy evaluation), detailněji popsanou v kapitole Funkcionální paradigma (ve kterém se často využívá).

```
     public double getMean() {
        if (mean == null) {
            mean = exams.stream()
                    .mapToDouble(value -> value).average()
                    .orElseThrow(() -> new IllegalStateException("List of
grades is empty."));
        }
        return mean;
    }
```

Metoda toString

Metoda sloužící k výpisu údajů o studentovi a jeho prospěchu. Využívá třídu StringBuilder, pomocí které připraví všechna data v požadovaném formátu – nejdříve studentovo jméno, dále všechny jeho známky a nakonec průměr – a vrátí je jako textový řetězec. Metodu toString implementuje třída Object, která je základní třídou v jazyce Java, od které všechny ostatní třídy přímo nebo nepřímo dědí, touto implementací je tedy provedeno přepsání původní zděděné implementace. Tato technika je často používána, protože díky jednotnému rozhraní různých tříd je vývojáři umožněno zpracování jednotným způsobem, což má za následek úsporu času, kratší zdrojový kód a menší náchylnost na chyby.

```
    public String toString () {
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.append(name).append(FIELD_SEPARATOR);
        for (Integer grade : exams) {
            stringBuilder.append(grade).append(FIELD_SEPARATOR);
        }
        stringBuilder.append(mean);
        return stringBuilder.toString();
    }
```

Jazyk Haskell – funkcionální implementace

Zdrojový kód modelové implementace je na první pohled v porovnání s ostatními analyzovanými paradigmaty velmi krátký. Krátký zápis je jednou z výhod funkcionálního paradigma.

Inicializace programu a načtení vstupních dat

Stejně jako v případě předchozích dvou paradigmat je první funkcí volanou operačním systémem po zavedení aplikace do paměti funkce main. V tomto případě se stará a převzetí argumentu s názvem vstupního souboru s daty o studentech, jeho převedení do programové datové struktury a předání další funkci ke zpracování. V případě, že při načítání vstupních dat dojde k chybě, tak program toto uživateli oznámí a ukončí se.

```
import System.Environment (getArgs)
import Text.CSV

main = do
      [inputFile] <- getArgs
      putStrLn inputFile
      parseResult <- parseCSVFromFile inputFile
      case parseResult of
            Left errmsg    -> putStrLn "Error when parsing input!"
            Right inputData -> processData inputData
```

Výpočet aritmetického průměru

O výpočet průměru studentova prospěchu se stará funkce avg, definovaná jako suma známek dělená jejich počtem, což je zároveň matematickou definicí aritmetického průměru. Pomocná funkce sum je součástí standardní knihovny jazyka Haskell,

```
avg grades = sum grades / List.genericLength grades
```

Řazení studentů dle průměru

Pro seřazení studentů podle jejich průměru je použita knihovní funkce sortBy s vlastním komparátorem, který řadí studenty dle jejich průměru.

```
sortBy (compare `on` studentMeans) $ students
```

Výpis studentů na výstup

O výpis studentů na výstup se stará funkce printStudents, ve které je možné definovat formát uživatelského výstupu.

```
let prettyPrint (Entry a b c d) = putStrLn $ a ++ ", " ++ show b ++ ", " ++
show c ++ ", " show d
```

Protože data o studentech jsou v programové paměti reprezentovány dvourozměrnou strukturou, je potřeba funkci prettyPrint volat s funkcí pro práci s dvourozměrnou strukturou, v tomto případě jde o mapu, je tedy použita funkce mapM_.

```
mapM_ printStudents students
```

Funkcionální implementace srovnávače studijních výsledků v jazyce Haskell je velmi krátká a jednoduchá. V porovnání s objektově orientovanou implementací ale není zdaleka tak robustní ani tak jednoduše rozšiřitelná. V tomto ohledu je srovnatelná s implementací v procedurálním paradigmatu. Oproti tomu je však náročnější z pohledu kompozice použitých funkcí a program se celkově může jevit jako hůře čitelný.

Závěr

V úvodní části práce byla stručně představena historie programování a nejpoužívanějších paradigmat pro vývoj aplikací. Práce se věnuje analýze dvou imperativních paradigmat, procedurálnímu a objektově orientovanému. Dále se práce věnuje analýze funkcionálního paradigma, které řadíme do skupiny deklarativních paradigmat.

Ve druhé kapitole bylo každé z jmenovaných paradigmat důkladně popsáno. Jsou popsány hlavní principy daných paradigmat, konstrukty specifické daným paradigmatům a techniky, které je využívají.

Třetí kapitola práce představuje tři projekty s otevřeným zdrojovým kódem. Každý z těchto projektů reprezentuje vhodné užití jednoho z analyzovaných paradigmat. Kapitola obsahuje ukázky zdrojových kódů daných projektů, na kterých toto vhodné užití demonstruje. Procedurální paradigma je demonstrováno na projektu SXML, což je minimalistický parser XML souborů implementovaný v jazyce C. Objektově orientované paradigma je demonstrováno na projektu Minx, což je mobilní webový prohlížeč pro operační systém Android, implementovaný v jazyce Java. Funkcionální paradigma je demonstrováno na projektu ShellCheck, což je nástroj pro statickou kontrolu Bash skriptů implementovaný v jazyce Haskell.

Analyzovaná paradigmata jsou vzájemně porovnávána napříč celou prací, závěry z pohledu vhodnosti jejich použití pro různé typy úloh jsou shrnuty v kapitole čtyři. Z těchto závěrů vyplývá, že procedurální paradigma se je vhodné zejména pro realizaci jednodušších projektů menšího rozsahu. Objektově orientované paradigma je naopak vhodnější pro realizaci větší a komplikovanějších projektů. Funkcionální paradigma lze vhodně použít pro libovolný typ projektu, jeho použití je však v porovnání s předchozími dvěma paradigmaty složitější a klade větší nároky na vývojáře využívající toto paradigma.

Práce obsahuje modelovou implementaci triviálního srovnávače studijních výsledků v každém z analyzovaných paradigmat. Na jednotlivých implementacích jsou demonstrovány hlavní principy těchto paradigmat s důrazem na jejich vzájemnou rozdílnost.

Práce představuje ucelený přehled nejpoužívanějších paradigmat pro vývoj aplikací, tato paradigmata popisuje teoreticky a demonstruje na praktických ukázkách.

Seznam použitých informačních zdrojů

Alvin, Alexander. 2017. *Functional Programming, Simplified.* 2017.

Bloch, Joshua. 2008. *Effective Java.* USA : Addison-Wesley, 2008. ISBN:0321356683.

Campbell-Kelly, Martin a Aspray, William. 2004. *Computer: a history of the information machine.* místo neznámé : Westview Press, 2004.

Caprino, Mario. 2014. SXML. *GitHub.* [Online] 2014. https://github.com/capmar/sxml.

Chauhan, Sharad. 2013. *Programming Languages - Design and Constructs.* Pune : University Science Press, 2013. ISBN-10: 9381159416.

Contracts for higher-order functions. **Findler, Robert Bruce a Felleisen, Matthias. 2002.** 9, 2002, Sigplan Notices, Sv. 37, stránky 48-59.

Eckerdal, Anna a Thuné, Michael. 2005. *Novice Java programmers' conceptions of "object" and "class", and variation theory.* 2005. stránky 89-93. Sv. 37.

Eekelen, Marko van. 2007. *Trends in Functional Programming.* Bristol : Gutenberg Press, 2007. ISBN-10: 9781841501765.

Ford, Neal. 2014. *Functional Thinking: Paradigm Over Syntax.* USA : O'Reilly, 2014. ISBN-13: 978-1449365516.

Govender, Irene. 2010. *From procedural to object-oriented programming (OOP) - an exploratory study of teachers' performance.* 2010. Sv. 46.

Heesch, Dimitri van. 2018. *Doxygen.* [Online] 2018. http://www.doxygen.nl.

McCloskey, Robert, Beidler, John a Bi, Yaodong. 2003. *CS2 and Java's comparator interface.* 2003. stránky 349-361. Sv. 19.

Rubio-Sanchez, Manuel. 2018. *Introduction to Recursive Programming.* USA : CRC Press, 2018. ISBN-13: 978-1498735285.

Seema Kedar, Sanjay Thakare. 2009. *Principles of Programming Languages.* Pune : Technical Publications, 2009. ISBN 8184315775.

Shafranovich, Y. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. October 2005. RFC 4180.

Sharma, Kartik. 2015. Minx. *GitHub.* [Online] 30. Prosinec 2015. [Citace: 23. Únor 2019.] https://github.com/crazyhitty/minx.

Stefik, Mark J. a Bobrow, Daniel G. 1986. *Object-oriented programming: Themes and variations.* 1986. stránky 40-62. Sv. 6.

Wexelblat, Richard L, [editor]. 1981. *History of Programming Languages.* Pennsylvania : Academic Press, 1981. ISBN-13: 978-0201895025.