



Univerzita  
Pardubice  
Fakulta elektrotechniky  
a informatiky

UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

KATEDRA SOFTWAREVÝCH TECHNOLOGIÍ

---

OBJEKTIVĚ ORIENTOVANÉ PROGRAMOVÁNÍ - IOOP

ZADÁNÍ SEMESTRÁLNÍ PRÁCE

# Lexikální analyzátor

---

*Autor zadání:*  
Karel ŠIMERDA

*Garant předmětu:*  
Karel ŠIMERDA

Verze 2.1  
25.9.2019

# 1 Úvod

## 1.1 Cíl semestrální práce

Cílem semestrální práce je osvěžení znalostí získané v předmětech IZAPR a IPALP z předchozího semestru.

Dalším cílem bude zjistit, do jaké míry studenti mají představu o objektově orientovaného programování. To vše bude ověřeno na malém projektu lexikálního analyzátoru, který studenti vypracují do „značné míry“ samostatně. Některé části programu budou prezentovány na přednáškách a některé postupy řešení budou ukázány na cvičeních.

Studenti v roli „kodéra“ budou realizovat návrh, který je popsán v tomto dokumentu. Studenti nebudou moci uplatnit tvůrčí přístup v návrhu architektury projektu, protože implementace všech tříd musí vyhovět jednotkovým testům. Testy jsou nedílnou součástí zadání semestrální práce. Testy budou předány v projektu, který bude dále obsahovat kostry zdrojových souborů. Úkolem studenta bude doplňovat postupně kostry zdrojových kódů, tak aby vyhověly jednotkovým testům.

## 1.2 Účel dokumentu

Tento dokument je určen pro vyučující a studenty, jako zadání první semestrální práce na cvičení předmětu „Objektově orientované programování“ (IOOP).

Dokument je členěn do několika částí.

Kapitola 2. Vize obsahuje vizi, která je neformálním popisem požadované funkce lexikálního analyzátoru. Vize slouží k přiblížené problematice lexikální analýzy studentům.

Kapitola 3. Konečný automat seznamuje čtenáře s úvodem do teorie konečných automatů. Tato kapitola je zařazena z důvodu lepšího pochopení návrhu lexikálního analyzátoru a usnadnění implementace.

Kapitola 4. Požadavky obsahuje funkční a nefunkční požadavky programu. Funkční požadavky popisují budoucí hlavní vlastnosti aplikace. Nefunkční požadavky předepisují například co se může použít, jak musí ošetřit chyby nebo jaké jsou výkonnostní omezení. Teprve splněním všech požadavků, funkčních a nefunkčních, může být semestrální práce přijata. Než se tak stane, bude studentům práce vrácená k dopracování. Je nutné upozornit na to, že může při nedodržení termínu odevzdání dojít k souběhu druhou semestrální prací, která bude výrazně větší.

Kapitola 5. Návrh lexikálního analyzátoru se věnuje návrhu aplikace lexikálního analyzátoru. Kapitola obsahuje diagramy UML, některé kapitoly o teorii OOP a podrobný popis návrhu aplikace.

Kapitola 6. Postup obsahuje doporučení, jak postupovat v řešení tohoto zadání semestrální práce.

Kapitola 7. Plán konfigurace obsahuje plán konfigurace projektu. Tento plán je návodem jak odevzdávat projekt na úložiště SVN. Plán obsahuje pravidla a procedury práce s úložištěm.

## 2 Vize

Program „**Lexikální analyzátor**“ bude umožňovat zpracování textu ze souboru a zobrazení výsledků v podobě seznamu tokenů na monitoru.

Proč takové zadání?

Pro zpracování textu a převod do dat v počítači byla v minulém semestru využívána třída `Scanner`. Její instance umožňovala bohaté zpracování textu ze vstupního zařízení. Její nevýhodou bylo, že se muselo předpokládat jaký druh informace v textu bude. Když se očekávalo celé číslo musela se použít metoda `nextInt()` a když reálné číslo tak `nextDouble()` nebo `nextFloat()`. Před tím se mohlo zeptat, zda takový formát dat je v textu připraven ke zpracování. K tomu sloužily metody `hasNextInt()`, `hasNextDouble()`, `hasNextFloat()` a tak podobně. To je někdy nepraktické a v případě jednoduché rozboru vstupního textu je výhodnější si naprogramovat vlastní řešení. K takovému řešení se obvykle říká lexikální analyzátor (anglicky scanner).

Lexikální analýza, kterou provádí lexikální analyzátor, rozděluje posloupnost znaků na lexikální elementy (například identifikátory, čísla, klíčová slova, operátory, a pod.). Tímto způsobem překladače zpracovávají naše zdrojové kódy do tzv. tokenů a ty se dál poskytují k dalšímu zpracování v syntaktickém analyzátoru. Se syntaktickou analýzou se někteří studenti setkají na navazujícím magisterském studiu v předmětech kompilátory nebo v teorii jazyků.

My samozřejmě nebudeme realizovat kompilátor, ale pokusíme se o naprogramování jednoduché lexikální analýzy v podobě malého lexikálního analyzátoru.

Náš program rozebere vstupní text na výstupní tokeny. V textu se budou hledat elementy tj. klíčová slova, čísla v desítkové, oktálové nebo hexadecimální soustavě a zbylé části textu se budou považovat za identifikátory, oddělovače nebo operátory. Elementy se budou mezi sebou oddělovat bílými znaky, jako jsou mezera, tabulátor, konec řádku a dále znaky rovnítko, čárky a středník. Čárka, rovnítko a středník budou plnit funkci oddělovačů a zároveň budou tokeny. Dalšími oddělovači budou matematické operátory sčítání, odečítání, násobení a dělení. Výstupem programu bude výpis charakteristik všech tokenů.

### 2.1 Příklad rozboru ukázkového textu

Vstupní text:

```
begin
a,b=100;
c=0x10;
end;
```

Výpis tokenů:

```
KeyToken{klicoveSlovo=Keyword{key=begin}}
SeparatorToken{Separator{bílý znak}}
IdentifierToken{a}
SeparatorToken{Separator{čárka}}
IdentifierToken{b}
SeparatorToken{Separator{rovná se}}
NumberToken{value=100}
SeparatorToken{Separator{středník}}
SeparatorToken{Separator{bílý znak}}
IdentifierToken{c}
SeparatorToken{Separator{rovná se}}
NumberToken{value=16}
SeparatorToken{Separator{středník}}
SeparatorToken{Separator{bílý znak}}
KeyToken{klicoveSlovo=Keyword{key=end}}
SeparatorToken{Separator{středník}}
SeparatorToken{Separator{bílý znak}}
```

### 3 Konečný automat

#### 3.1 Obecně o konečných automatech

Konečný automat přijímá vstupní symboly z konečné množiny zvané *vstupní abeceda*  $I$  a vydává výstupní symboly zvané *výstupní abeceda*  $O$ . Dále má určitý počet *vnitřních stavů*  $Q$  (dále jen stavy). Vstupní symboly, výstupní symboly a stavy se uvažují pouze v rámci diskretních časových okamžiků. Předpokládá se, že počet stavů je konečný, proto se používá název konečný automat.

Popis činnosti automatu: Předpokládejme, že automat je na počátku v některém stavu  $q_i \in Q$ . Při příchodu vstupního symbolu  $I_m \in I$  automat vydá výstupní symbol  $z_k \in O$  a přejde do stavu  $q_j \in Q$ , ve kterém tedy bude v příštím časovém okamžiku. Následující stav  $q_j$  a výstupní symbol  $z_k$  jsou jednoznačně určeny současným stavem a vstupním symbolem.

Praktické využití

- zpracování přirozeného jazyka
- překladače
- hledání výskytu slova v textu
- realizace protokolů

Popis konečného automatu

- stavovým diagramem
- tabulkou
- stavovým stromem.

Příklad implementace konečného automatu se třemi stavy je na výpisu 1.

Listing 1: Ukázka kódu jednoduchého konečného automatu

```

1 public class Inentifier1 {
2     enum Stav { START, ANO, NE };
3     static boolean execute(String text) {
4         System.out.print(text + ":");
5         Stav stav = START;
6         for (int i = 0; i < text.length(); i++) {
7             char znak = text.charAt(i);
8             if (stav == START) {
9                 if (znak == '_' || ('a' <= znak && znak <= 'z')) {
10                     stav = ANO;
11                 } else {
12                     return false;
13                 }
14             } else {
15                 if (!(znak == '_' || ('a' <= znak && znak <= 'z')
16                     || ('0' <= znak && znak <= '9')))) {
17                     stav = NE;
18                     return false;
19                 }
20             }
21         }
22         return stav == ANO;
23     }
24     public static void main(String[] args) {
25         System.out.println(execute("abc_123"));
26         System.out.println(execute("1abc123"));
27         System.out.println(execute("1abc@123"));
28     }

```

## 4 Požadavky

### 4.1 Funkční požadavky

**FR1 Hlavní funkce:** Program musí provádět lexikální analýzu.

**FR2 Zobrazování seznamu:** Program bude vypisovat výsledek lexikální analýzy vstupního textového souboru v podobě seznamu tokenů, tak je naznačeno v kapitole 2.Vize.

**FR3 Spouštěcí příkaz:** Program se bude spouštět z příkazového řádku svým názvem, kdy jeho prvním parametrem bude jméno souboru, případně s cestou k němu.

**FR4 Volba klíčových slov:** Program bude zpracovávat tyto klíčová slova: `begin`, `end`, `for`, `if`, `then`, `else`, `while` a `return`.

**FR5 Identifikátory:** Program bude slova, která budou začínat písmenem a nebudou klíčová, považovat za identifikátory.

**FR6 Číselné literály:** Program bude rozlišovat oktalová, desítková a hexadecimální čísla. U hexadecimální čísel se požaduje prefix `0x`. Oktalová čísla budou začínat znakem nula.

**FR7 Oddělovače:** Program bude rozlišovat tyto oddělovače: rovnítko `„=“`, čárku `„,“`, dvojtečku `„:“` a středník `„;“`.

**FR9 Bílé znaky:** Program bude bílé znaky považovat za oddělovače.

**FR10 Velikost písmen:** Program bude zpracovávat malá i velká písmena.

### 4.2 Nefunkční požadavky

**NR1 Realizace:** Požaduje se, aby realizací semestrální práce bylo rozšíření projektu pro vývojové prostředí NetBeans, které bude uvolněno vyučujícím na cvičení z předmětu IOOP.

**NR2 Ošetření chyb:** Požaduje se, aby chyby ve vykonávání metod byly hlášeny pouze pomocí výjimek, které budou zachyceny až v metodě `main`.

**NR3 Výčtové hodnoty:** Požaduje se, aby místo konstant byly v programu důsledně používány výčty.

**NR4 Jména výčtových hodnot:** Výčty musí umožňovat uložení přirozeného českého jména hodnoty a to malými písmeny, které se potom použijí při výpisech charakteristik tokenů.

**NR5 Hledání klíčových slov:** Klíčová slova se budou vyhledávat ve výčtu klíčových slov.

**NR6 Struktura tokenů:** Požaduje se, aby třídy, které budou reprezentovat speciální tokeny, byly uspořádány do hierarchického stromu dědičnosti s jednou vrcholovou generalizovanou třídou obecného tokenu.

**NR7 Ověření jednotkovými testy:** Požaduje splnění všech testů, který byly součástí zadání.

**NR8 Ověření programu:** Pro ověření aplikace si každý student vytvoří alespoň jeden zkušební textový soubor, kterým bude moci přezkoušet plnění požadavků. Tento soubor bude součástí projektu uloženého na SVN.

**NR9 Soulad s návrhem:** Program musí být v souladu s popisem a s diagramy, které jsou uvedeny v kapitole 5.Návrh lexikálního analyzátoru

**NR10 Kolekce:** Není dovoleno používat knihovny kolekce Javy.

**NR11 Knihovna:**Není dovoleno používat knihovny Javy a to především metody obalových tříd na převod textu na číselnou hodnotu.

**NR12 Termín dokončení** Požaduje se dokončení semestrální práce do čtvrtého týdne po zadání.

## 5 Návrh lexikálního analyzátoru

Návrh programu s lexikální analýzou je popsán diagramy tříd, sekvenčními a stavovými diagramy. Diagramy tříd popisují statickou strukturu programu. Zatímco sekvenční a stavové diagramy modelují jeho dynamické chování.

### 5.1 Statická struktura

#### 5.1.1 Přehledový diagram tříd

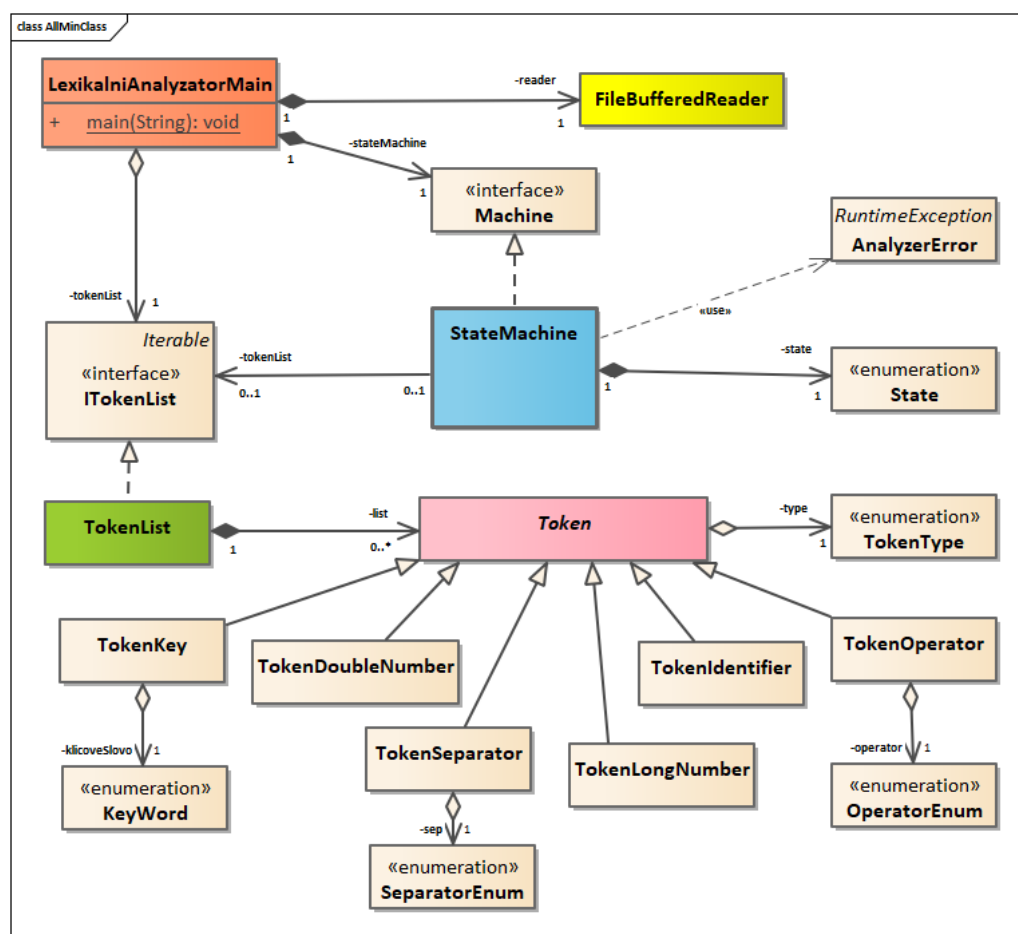
Na obrázku s diagramem tříd 1. Přehledový diagram tříd aplikace lexikálního analyzátoru jsou zobrazeny všechny předměty a relace modelu tříd aplikace lexikálního analyzátoru. Předměty modelu jsou třídy, rozhraní a výčty cílové aplikace. Diagram dále obsahuje vazby mezi předměty modelu, kterými jsou relace a dědičnost.

V tomto diagramu je potlačeno zobrazení podrobností předmětů, abychom se mohli soustředit na celkovou architekturu návrhu aplikace. Podrobnosti předmětů, tj, atributy a metody, jsou uvedeny až v následujících diagramech.

Výklad diagramu využijeme k výuce vizuálního modelovacího jazyka UML. Naučíme se číst diagramy tříd. Co to jsou předměty a vazby mezi nimi.

Následující text kapitoly obsahuje teorii OOP a UML, která je potřebná k porozumění diagramu tříd 1. V kapitole se střídají odstavce s teorií s odstavci s konkrétními jejími aplikacemi.

Ve výkladu budeme postupovat od vrcholových tříd, kterými jsou *LexikalniAnalyzatorMain*, *FileBufferedReader*, *StateMachine* a *TokenList* k ostatním třídám diagramu.



Obrázek 1: Přehledový diagram tříd aplikace lexikálního analyzátoru

### Teoretický úvod do objektového přístupu k programování a do UML

Teorie OOP a UML

Při zkoumání diagramu tříd je důležité vědět, že třídy jsou pouze vzory podle kterých za běhu aplikace vznikají objekty, nebo-li instance tříd. Relace (asociace a její varianty)

mezi třídami se za běhu aplikace transformují do jednoho nebo více spojení mezi objekty. Násobnosti u relací říká kolik spojení objekt jedné třídy může vytvořit s objekty druhé třídy. Teprve po vytvořených spojeních je možné přenášet zprávy z jednoho objektu do druhého. Zprávy v Javě se realizují voláním metod v cílovém objektu, proto Java není čistě objektová. Aby objekt mohl vyvolat metodu v jiném objektu, musí mít k dispozici referenci na cílový objekt, jinak řečeno musí mít spojení. Pro každé spojení s jinými objekty potřebuje objekt po jedné referenci. Z toho logicky plyne, že mezi dvěma objekty lze vytvořit pouze jedno spojení.

Příklad jak číst diagram tříd

**Z diagramu tříd 1 lze sestavit tyto téměř přirozené věty které vyjadřují hlavní myšlenky návrhu:**

1. Lexikální analyzátor **má** čtení ze souboru, **má** konečný automat s lexikální analýzou a **má** seznam tokenů.
2. Konečný automat **má** stav a **má** seznam tokenů.
3. Seznam tokenů **má** tokeny.
4. Každý token **má** typ.
5. TokenKey **je** Token, TokenDoubleNumber **je** Token atd.

**Jednoduché pravidlo pro tvorbu a interpretaci vazeb v diagramu tříd:** Když můžeme použít slovo **má** ve větě, která popisuje vztah mezi třídami, jedná se o relaci (asociaci, agregaci nebo kompozici). Když lze použít mezi názvy tříd slovo **je**, jedná se o vztah zobecnění (dědičnosti) mezi třídami. Je dobré si toto pravidlo zapamatovat, protože to může velmi usnadnit jak najít dobrý objektový návrh SW.

Pravidlo jak vytvořit vazby mezi třídami

**Mezi třídami používáme tři varianty asociační vazby:**

Teorie OOP

### Základní asociaci

Vztah mezi třídami, když objekt dostane spojení na jiný objekt při nebo až po svém vzniku. Oba objekty mohou existovat plně samostatně

### Sdílenou agregaci

Vztah mezi třídami, když objekt (celek) si vytvoří (agreguje) jiný objekt (součást), který potom umožní tuto součást sdílet ještě dalším objektům. Po zániku celku můžou jeho součásti dále existovat.

### Kompozici

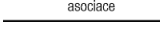
Vztah mezi třídami, když si objekt (celek) vytvoří další objekty, jako své součásti, které nikomu neposkytne, plně je skryje. Při zániku celku zanikají i jeho součásti.

My budeme pro tři výše popsané asociační vazby používat společný pojem *relace*.

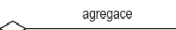
V dvojici tříd, které spojuje nějaká relace, se jedna třída považuje za zdrojovou (*source*) a druhá třída za cílovou (*target*). Za zdrojovou třídu se považuje ta, která obsahuje atribut pro referenci na instanci cílové třídy. Někteří autoři používají obrácené označování tříd a to tak, že atribut s referencí mají v cílové třídě. My budeme používat první možnost.

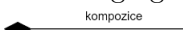
Role tříd ve vazbě

### Vizuální syntaxe relací asociace UML

Na zobrazení relace asociace se používá prostá čára . Čáry bývají doplněny otevřenými šipkami, které určují kdo koho může používat. V naprosté většině případů mají relace jen jednu šipku. Je-li šipka u třídy, tak její instance bude používána instancí druhé třídy. Když není použita ani jedna šipka, tak viditelnost, nebo-li průchodnost, není stanovena. Je-li šipka na obou koncích jedná se vzájemnou viditelnost, což může způsobovat komplikace. Takové třídy jsou na sobě vzájemně závislé.

Teorie UML

Jestliže čáru na jednom konci opatříme malým kosočtvercem, jedná se o další dvě specializace asociace. Nevyplněný kosočtverec nám říká, že se jedná o sdílenou agregaci .

Vyplněný kosočtverec oznamuje, že jedná o kompozici . Šipky u těchto relací mají stejný význam jako u asociace.

**Instancí relace mezi třídami je spojení mezi objekty.**

Teorie OOP

Je důležité upozornit na to, že programovací jazyky typy relací nepodporují. Implementace typu relace je na programátorovi, zda a jak dodržuje zásady dobrého návrhu softwaru [1].

Aplikování zásad návrhu OOP

Tou nejdůležitější zásadou je skrývání implementace (viz LEARN IOOP Téma 6). Dobré skrytí implementace zajistí bezpečnější zdrojový kód, který bude snadno znovupoužitelný (*reuse*).

Skrývání implementace si můžeme ukázat na třídě `TokenList`, která skrývá implementaci vnitřního seznamu `list`. Skrytí je naznačeno tím, že atribut `list` je privátní a že je v diagramu použita kompozice na třídu `Token`. Seznam `list` můžeme implementovat různým způsobem, například pomocí pole, jako spojový seznam nebo požit vhodnou třídu (koleci) z knihovny.

## Popis návrhu vrcholové struktury aplikace

Činnost analyzátoru se zahajuje ve třídě `LexikalniAnalyzatorMain`. Tato instance vytváří nejdříve tři spojení na tyto objekty:

1. Na objekt třídy `FileFufferedReader`, který přebírá znak po znaku z textového souboru. Trvalé spojení na objekt se zajistí uložením reference do privátního atributu `reader`.
2. Na objekt s rozhraním `Machine`, který provádí lexikální analýzu na tokeny. Trvalé spojení na objekt se zajistí uložením reference do privátního atributu `stateMachine`.
3. Na objekt s rozhraním `ITokenList`, který shromažďuje všechny dekodované tokeny do seznamu. Trvalé spojení na objekt se zajistí uložením reference do privátního atributu `tokenList`.

Jestliže spojení vytvoříme (agregujeme objekty) v metodě `main()`, která je metodou třídy (modifikátor `static`), budeme muset získané reference uložit též do proměnných třídy. To není dobré řešení. Znemožňuje to vytvoření více instancí takové třídy.

Proto bude vždy lepším řešením v metodě `main()` vytvořit instanci vlastní třídy. Potom lze reference na spolupracující objekty uložit do instančních proměnných instanční metodou volanou z metody třídy `main()`.

**Poznámka:** *Jedna aplikace může obsahovat více tříd s metodou `main()`, ale jako první může být spuštěna jenom jedna. Která metoda `main()` se spustí se musí zvolit ve spouštěcím příkazu virtuálního stroje (JVM).*

Hlavní kroky činnosti lexikálního analyzátoru:

1. Zpracování vstupního textu bude probíhat ve cyklu ve vhodné metodě ze třídy `LexikalniAnalyzatorMain`.
2. Postupným čtením znaku po znaku ze souboru objektem třídy `FileFufferedReader` se načtený znak nechá zpracovat objektem třídy `StateMachine`.
3. Pokud objekt třídy `StateMachine` dekoduje některý token, uloží ho do objektu třídy `TokenList`.
4. Po převzetí posledního znaku ze souboru se přejde na výpis seznamu tokenů z objektu třídy `TokenList`.

Kroky lexikálního analyzátoru lze zobrazit vizuálně pomocí sekvenčního diagramu na obrázku 2. Sekvenční diagram lexikálního analyzátoru.

Sekvenční diagramy zachycují časově uspořádanou posloupnost zasílání zpráv mezi objekty. Obdélníky se svislými čárkovanými čarami se nazývají životočáry. Do hlavy životočáry je možno zapsat před dvojtečku název objektu a za dvojtečku je možné uvést klasifikátor objektu. Směrem dolů po životočáře plyne čas. Když je životočára ukončena pootočeným křížkem, tak se jedná o ukončení života objektu. Za klasifikátor se používají většinou názvy tříd. Vodorovné čáry představují zasílání zpráv. Zakončením vodorovné čáry různými šípkami se volí, zda se jedná jednoduchou zprávu (otevřená šipka), synchronní (vyplněná šipka) nebo asynchronní zprávu (poloviční otevřená šipka). Svislý obdélníček na životočáře vyznačuje aktivaci objektu. Délka obdélníčku značí dobu aktivace objektu. V rámci jedné aktivace se vykonává jedna operace (metoda) objektu. Zprávy, které končí na hlavě životočáry, jsou konstruktory objektu.

Objekty lze vytvářet jen podle instančních tříd a ne podle interfejsů. Interfejsy obsahují v Javě pouze deklarace metod a definice, tj. těla metod, musí dodat implementační třída.

Návrh aplikace

Tři hlavní objekty  
lexikálního analyzátoru

Chybné místo pro  
vytvoření spojení

Vytváření spojení jedině  
v instanci

Pozor na metodu  
`main()` je vyjimečná

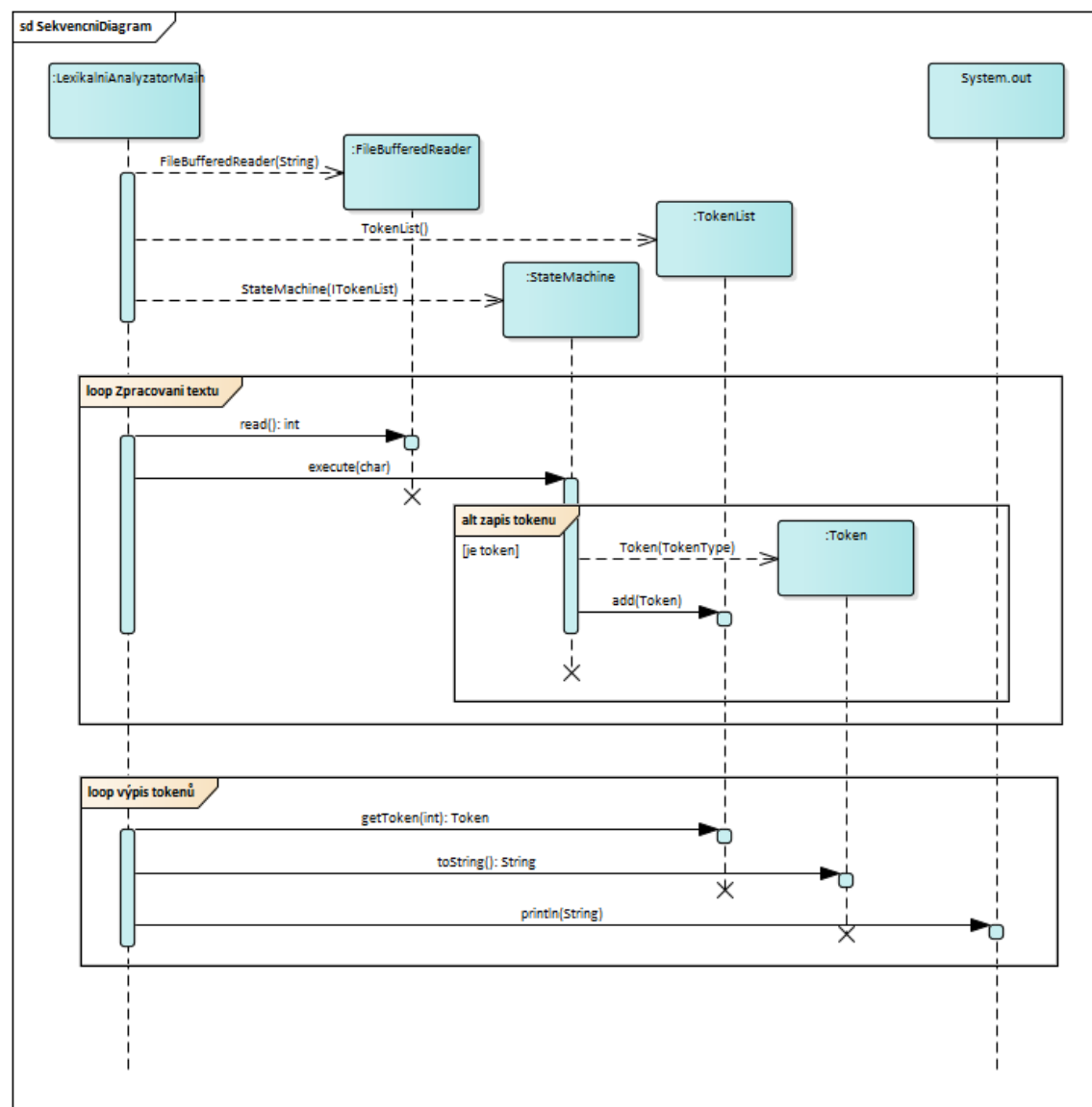
Postup činnosti

Sekvenční diagram  
lexikálního analyzátoru

Teorie UML - Sekvenční  
diagramy  
(zjednodušeně)

Zásada programování  
proti rozhraní





Obrázek 2: Sekvenční diagram lexikálního analyzátoru

Objekty se sice vytvářejí podle instančních tříd `StateMachine` a `TokenList`, ale pro instanci třídy `LexikalniAnalyzatorMain` jsou přístupné jenom metody, které jsou deklarovány v jejich interfejsch. Když se v deklaracích referenčních proměnných jako typ používají interfejsy místo tříd, tak se jedná o programování proti rozhraní (viz LEARN IOOP Téma 5). Výhoda tohoto přístupu je v tom, že můžeme kdykoliv provést záměnu implementačních tříd, aniž bychom museli měnit zdrojový kód ostatních tříd.

**Poznámka:** Je dobré rozlišovat pojmy *interfejs* a *rozhraní*. Pojem *rozhraní* použijeme pro souhrn všech veřejných metod třídy. Pojem *interfejs* budeme používat pro speciální třídy, které obsahují jenom abstraktní metody. Java pro takovou abstraktní třídu zavedla speciální konstrukci, pro kterou využívá klíčové slovo `interface`. Platí pravidlo, že třída musí implementovat všechny metody připojeného interfejsu, ale naopak rozhraní třídy může mít další veřejné metody.

interfejs versus rozhraní

Kvalita objektového návrhu je dána, jak jsou použity zásady dobrého návrhu a další techniky návrhu. Těmi dalšími technikami mohou být návrhové vzory, faktoring, čistý kód atd.

Kvalita návrhu

Zásada o soudržnosti (viz LEARN IOOP Téma 9) doporučuje, aby metoda nebo celá třída se zabývala pouze jedním úkolem a případné ostatní úkoly delegovala na jiné metody nebo jiné třídy.

Zásada návrhu OOP:  
Soudržnost

Náš návrh vrcholové struktury aplikace přiděluje hlavní úkoly třem třídám. Tím, že jsme požadovanou funkčnost rozdělili do tří tříd, budeme moci některé třídy využít i v jiných

Zásadní architektonické  
rozhodnutí

projektech bez dodatečných úprav. Nebo naopak budeme moci v tomto našem projektu snadno zaměnit některou třídu jinou. Kandidátem na výměnu bude třída `TokenList`, kdy ji budeme moci zaměnit třídou z druhé semestrální práce.

Z hlediska celkové funkce lexikální analýzy by bylo jedno, zda by všechny tři odpovědnosti (úkoly) byly implementovány v metodě `main()` třídy `LexikalniAnalyzatorMain`. Takový přístup k architektuře aplikace není výjimečný. Velmi často se vyskytuje v projektech studentů, tak mnohdy i u zkušenějších programátorů, kteří nerespektují objektový přístup a dobré zásady návrhu. Ukázku takového kódu máme i v tomto dokumentu, viz 1. Ukázka kódu jednoduchého konečného automatu.

Špatný příklad architektury

Je-li celý nebo podstatná část zdrojového kódu jedné metodě nebo ve všeobíhající třídě, tak takový kód nelze snadno znovu použít (tzv. *reuse*). Když k tomu dojde, musíme vynaložit značné a zbytečné úsilí na vyjmutí potřebného kódu. Doprovodným efektem takového přístupu je vznik duplicit.

### Detailní návrh třídy `StateMachine`

Třída zajišťuje hlavní funkčnost aplikace tj. lexikální analýzu. Třída implementuje interfejs `Machine`, který má dvě abstraktní metody a dvě metody s tělem.

Od verze 8 Javy je možné i v interfejsech definovat u metod i těla. Tyto metody se musí opatřit modifikátorem `default`, mají značné omezení a jejich vlastnosti budou vysvětleny v průběhu semestru na přednáškách. Zde si ukážeme jenom zdrojový kód interfejsu `Machine` 2, z kterého bude patrné o co se jedná.

Listing 2: Interfejs pro konečný automat `Machine`

```

1 public interface Machine {
2     void execute(char character);
3     State getState();
4     default void execute(char... list) {
5         for (char character : list) {
6             execute(character);
7         }
8     }
9     default void execute(String str) {
10        execute(str.toCharArray());
11    }
12 }
```

Instance třídy `StateMachine` postupným zpracováním sekvence vstupních znaků postupně mění svůj vnitřní stav. Konkrétní stav si objekt uchovává do okamžiku příchodu charakteristického znaku pro začátek jiného tokenu. Objekt třídy `StateMachine` si stav uchovává v privátní atributu `state`. Když objekt dokončí načtení sekvence znaků jednoho tokenu, vytvoří si token zjištěného typu a uloží ho do seznamu tokenů.

Seznam tokenů `TokenList` spravuje podle diagramu 1 pouze objekty typu `Token`. Třída `Token` je abstraktní a proto nelze z ní vytvořit instanci. Do seznamu se ale může uložit reference na instance potomků abstraktní třídy `Token`. Tomu jevu se říká substituční princip (wiki), jehož autorkou je Barbara Liskov

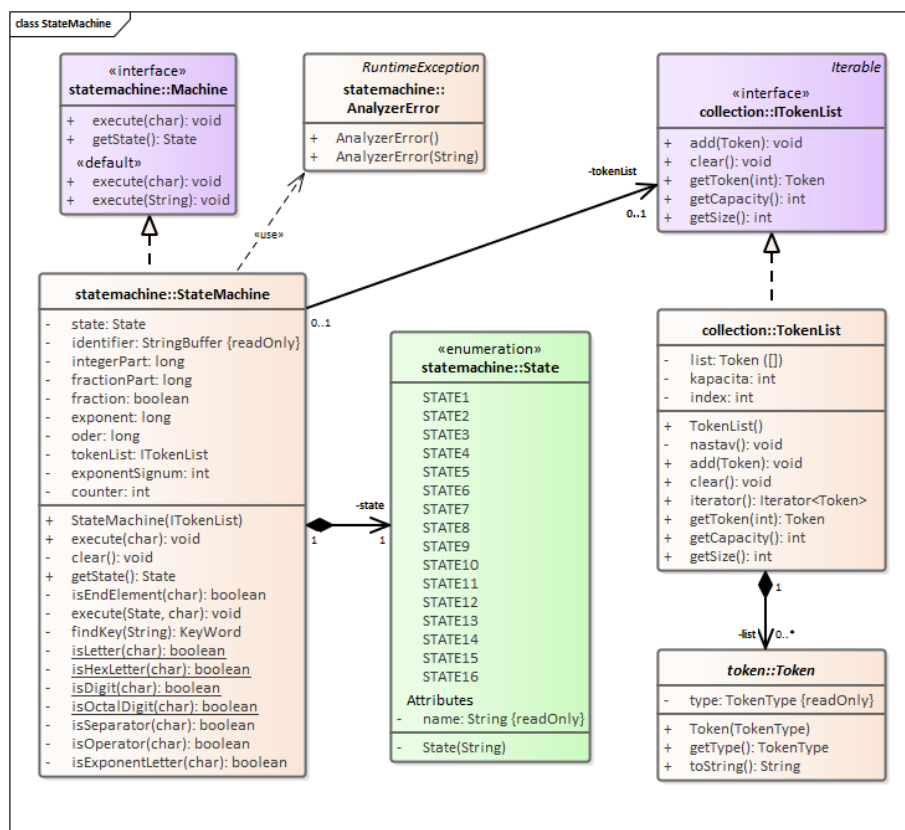
Na obrázku 3 je detailní diagram tříd konečného automatu `StateMachine`. Předměty diagramu (třídy, interfejsy a výčty) mají vypsány kromě veřejných i všechny privátní členy (členy = atributy, metody a výčtové hodnoty).

Privátní členy naznačují jaká byla zvolená implementace vyučujícím. Každý student si může zvolit jiné privátní členy podle své implementace. Povinné je dodržet implementaci interfejsu `Machine`.

Jestliže dojde v metodách `execute(...)` k chybě, požaduje se hlásit chybu vystavením výjimky `AnalyzerError`.

Konstruktor třídy `StateMachine` má jeden parametr, v kterém se bude předávat reference na objekt se seznamem tokenů `TokenList`.

Podtržení členů třídy znamená, že se jedná o členy třídy. To znamená, že tyto členy jsou v Javě opatřeny modifikátorem `static`.



Obrázek 3: Diagram tříd konečného automatu `StateMachine`

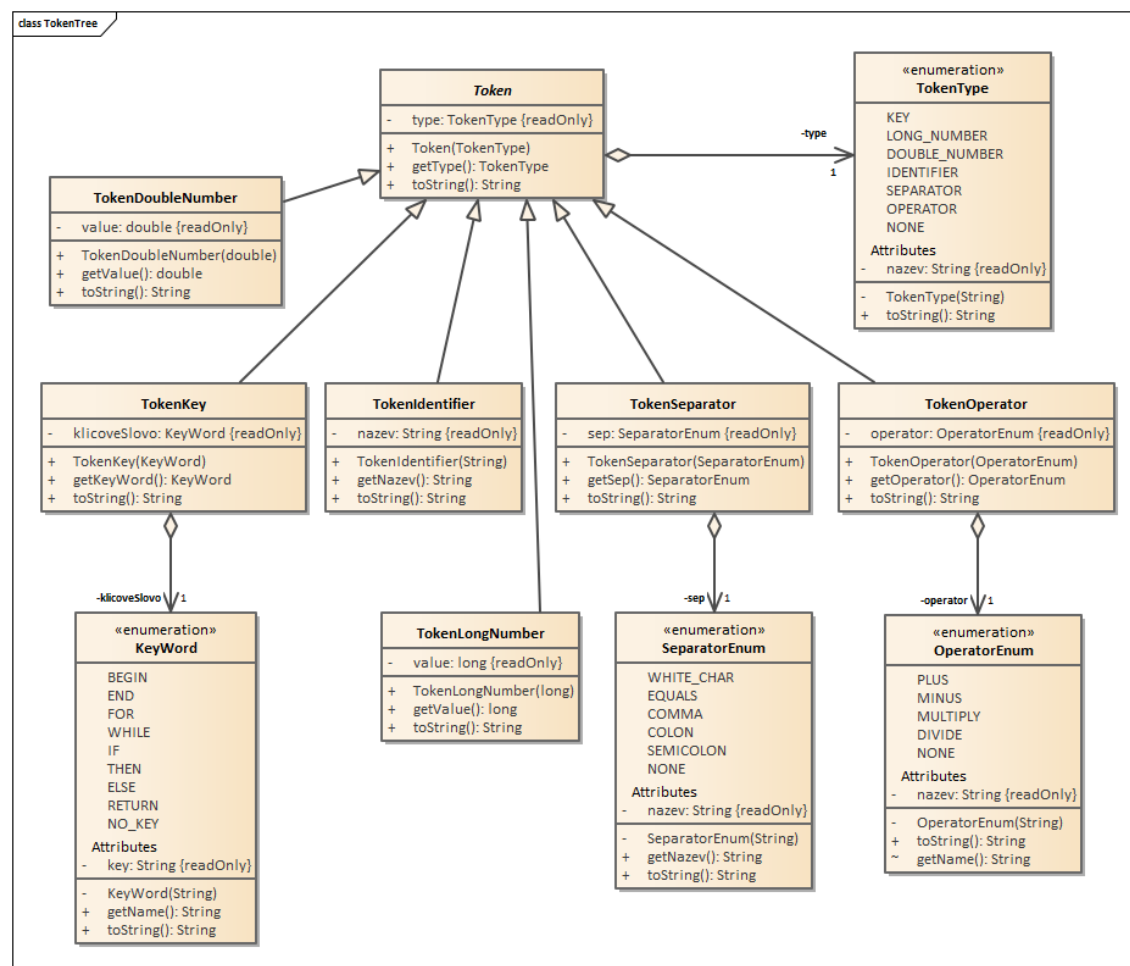
### Strom dědičnosti tokenů

Na obrázku 4 je diagram se stromem dědičnosti všech typů tokenů. Název třídy `Token` má v diagramu skloněno písmo (je vysázen kurzívou). Kurzíva v názvu třídy v UML znamená, že třída je abstraktní.

Dobrou praktikou je mít v abstraktní třídě atribut s typem potomka. V našem případě je to privátní atribut `type` do kterého se ukládá jedna z hodnot výčtu `TokenType`. Proč je to dobré? Znalost typu potomka umožňuje se vyhnout testování kdo byl potomek. Místo testování kdo byl potomek můžeme použít přepínač podle typu. Když známe typ potomka můžeme pomocí přetypování získat přístup k rozhraní potomka.

Atribut `type` má v digramu uvedeno omezení `readOnly`, což znamená, že se jedná o „neměnnou proměnnou“, kterou po naplnění již nelze změnit. Hodnotu lze nastavit pouze v definici proměnné nebo, což je náš případ, konstruktorem.

Potomci abstraktní třídy `Token` obsahují atributy s typem, který je dán konkrétním typem tokenu. Potomci mají polymorfní metodu `toString`, která vrací textový řetězec s charakteristikou konkrétního typu tokenu.



Obrázek 4: Diagram tříd se stromem tokenů

## 5.2 Dynamický model

Modelování dynamického chování požadované funkce je důležitou součástí návrhu softwaru. Na modelu si můžeme dopředu ujasnit, které řešení je nejlepší. Pokud hledání řešení děláme až v okamžiku implementace, můžeme se dostat do situace, že musíme opustit již naprogramovaný kód a vytvářet nový. Dokonce některé CASE nástroje dokáží vykonávat stavové diagramy ještě před jeho implementací.

UML poskytuje několik diagramů na modelování chování. My si ukážeme jen dva diagramy a to sekvenční a stavový.

Se sekvenčním diagramem jsme se setkali na obrázku 2. Tento diagram se hodí k modelování interakcí mezi objekty.

Druhým diagramem, který zde použijeme, je stavový diagram. Tento diagram se hodí k modelování chování jednoho objektu. Pro konečný automat s algoritmem lexikálního analyzátoru je vypracování takového stavového diagramu nezbytností. Důvodem je velký počet stavů a přechodů mezi nimi. Stavový diagram lexikální analýzy je na obrázku 5. Tento diagram obsahuje všechny stavy automatu a má potlačeno zobrazení detailních informací o stavech. Podrobnější informace o stavech jsou zobrazovány v následujících dílčích diagramech.

Největší skupinou stavů jsou stavy, kterými se rozpoznávají formáty a soustavy číselných literálů. Soustavy jsou tři: desítková, oktalová a hexadecimální. Formáty jsou také tři: celé číslo, číslo s desetinou tečkou a semilogaritmický.

Stavy jsou označeny anglický slovem **State** a číslem. Číslování stavu je běžné, protože to usnadňuje orientaci v diagramu. Hodnota čísla nemá žádný význam, slouží pouze k odlišení stavů.

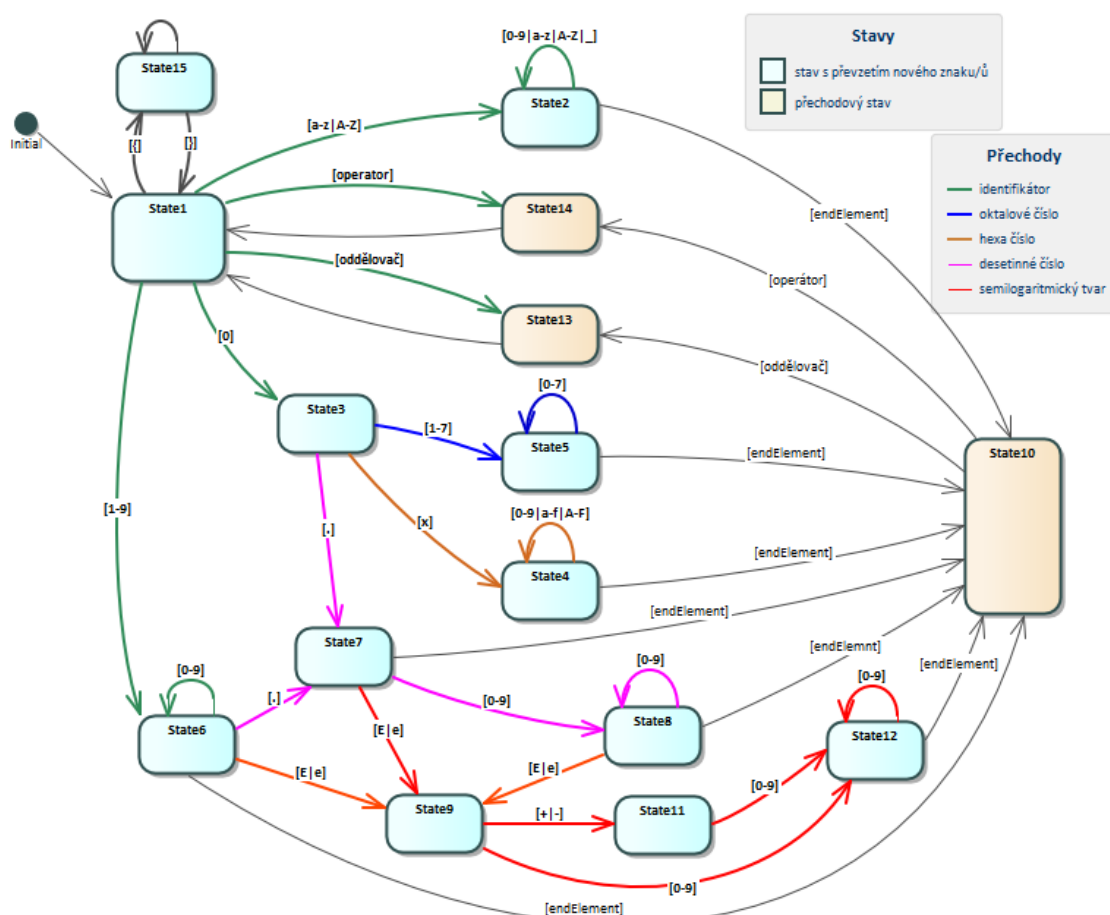
### 5.2.1 State1 - Výchozí stav

Výchozím stavem je stav `State1`, do kterého se automat dostane

1. při inicializaci
2. po dočtení komentáře
3. po načtení operátoru nebo oddělovače.

Z tohoto stavu se vychází když

1. přijde levá složená závorka, přejde se na čtení komentáře,
2. přijde malé nebo velké písmeno,
3. přijde operátor, zpracuje se operátor,
4. přijde oddělovač, zpracuje se oddělovač,
5. přijde znak nula, přejde se na čtení čísla v oktalové, hexadecimální soustavě nebo na číslo s desetinnou částí v desítkové soustavě,
6. přijde znaky „1“ až „9“, přejde se na čtení čísla v desítkové soustavě.

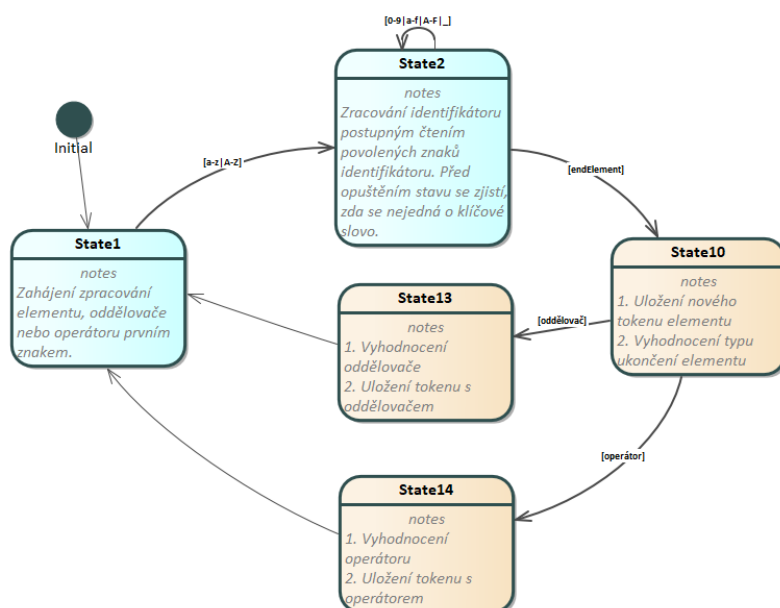


Obrázek 5: Stavový diagram konečného automatu s lexikální analýzou

### 5.2.2 State2 - Čtení identifikátoru

Rozpoznaní identifikátoru je velmi snadné (viz diagram 6). Načítání identifikátoru začíná ve stavu `State1`, když přijde znak s malým nebo velkým písmenem. Dalšími znaky můžou

být zase malé nebo velké písmeno, číslice a podtržítka. Načítání končí příchodem operátoru nebo oddělovače, což v diagramu je naznačeno hranou s podmínkou `endElement`. Před ukončením stavu ještě vyhodnotí, zda se nejedná o klíčové slovo. Pokud je zjištěno, že identifikátor je klíčové slovo, tak se vytvoří token ze třídy `TokenKey`, jinak se vytvoří od třídy `TokenIdentifier`. Uložení do seznamu tokenů se provede až při vstupu do přechodového stavu `State10`.



Obrázek 6: Stavový diagram čtení identifikátoru

### 5.2.3 State3 - Zjistění, zda bude číslo v oktalové, v hexadecimální soustavě nebo desetinné

Pokud ve stavu `State1` přijde číslice nula, budou se očekávat znaky v číselné soustavě oktalové nebo hexadecimální. Která to bude, rozhodne až následující znak. Stav se opouští příchodem dalšího znaku a tím může být písmeno x, číslice „0“ až „7“ nebo tečka. Když přijde tečka, přejde se do stavu načítání desetinné části čísla. Když přijde písmeno x, přejde se do načítání hexadecimálního čísla. A nakonec, když přijdou znaky „0“ až „7“ přejde se na načítání oktalového čísla. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`. Stav při opuštění nic neukládá.

### 5.2.4 State4 - Načítání čísla v hexadecimální soustavě

Do stavu se přejde ze stavu `State3`, když přišel znak „x“. V tomto stavu se očekávají číslice „0“ až „9“ a malá a velká písmena „a“ až „f“ nebo „A“ až „F“, které se převádějí na binární celočíselnou hodnotu. Stav se opustí, když přijde operátor nebo oddělovač, což v diagramu je naznačeno hranou s podmínkou `endElement`. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`. Při opuštění stavu se vytvoří token celého čísla `TokenIntegerNumber`. Uložení do seznamu tokenů se provede až při vstupu do přechodového stavu `State10`.

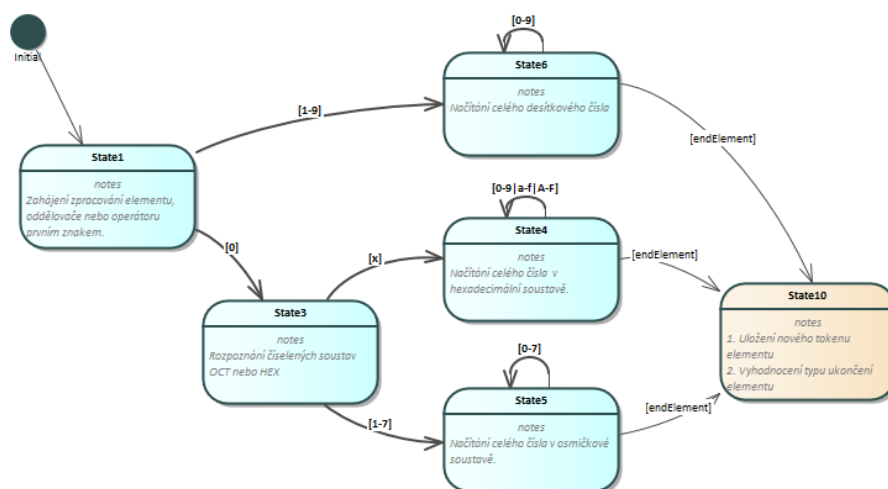
### 5.2.5 State5 - Načítání čísla v oktalové soustavě

Do stavu se přejde ze stavu `State3`, když přišly znaky číslic „1“ až „7“. V tomto stavu se očekávají číslice „0“ až „7“, které se převádějí na binární celočíselnou hodnotu. Stav se opustí, když přijde operátor nebo oddělovač, což v diagramu je naznačeno hranou s podmínkou `endElement`. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`. Při opuštění stavu se vytvoří token celého čísla `TokenIntegerNumber`. Uložení do seznamu tokenů se provede až při vstupu do přechodového stavu `State10`.

### 5.2.6 State6 - Načítání čísla v desítkové soustavě

Do stavu se přejde ze stavu `State1`, když přišly znaky číslic „1“ až „9“. V tomto stavu se očekávají číslice „0“ až „9“, které se převádějí na binární celočíselnou hodnotu. Stav se

opustí, když přijde operátor nebo oddělovač. Pokud ovšem přijde-li znak tečky, přejde se do stavu `State7`, kde se zahájí načítání desetinné části čísla s pevnou čárkou. Když přijde jeden ze znaků malé „e“ nebo velké „E“, zahájí načítání exponentu ve stavu `State9`. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`. Při opuštění stavu se vytvoří token celého čísla `TokenIntegerNumber`. Uložení do seznamu tokenů se provede až při vstupu do přechodového stavu `State10`.



Obrázek 7: Stavový diagram zpracování celých čísel

### 5.2.7 State7 - Zjištění, zda bude načítání desetinné části čísla nebo exponentu

Stav čeká na další znak, podle kterého se rozhodne jaký stav bude následovat. Když přijde znak z rozsahu číslic „0“ až „9“, tak se bude načítat desetinná část čísla ve stavu `State8`. Když přijde jeden ze znaků malé „e“ nebo velké „E“, zahájí načítání exponentu ve stavu `State9`. Stav se opustí, když přijde operátor nebo oddělovač, což v diagramu je naznačeno hranou s podmínkou `endElement`. V tom případě před ukončením stavu se vytvoří token podle třídy `TokenDoubleNumber`, který se uloží do seznamu tokenů až při vstupu do přechodového stavu `State10`. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`.

### 5.2.8 State8 - Načítání desetinné části desítkového čísla

Stav načítá znaky „0“ až „9“ a převádí je do desetinné části čísla. Když přijde jeden ze znaků malé „e“ nebo velké „E“, zahájí načítání exponentu ve stavu `State9`. Stav se opustí, když přijde operátor nebo oddělovač, což v diagramu je naznačeno hranou s podmínkou `endElement`. V tom případě před ukončením stavu se vytvoří token podle třídy `TokenDoubleNumber`, který se uloží do seznamu tokenů až při vstupu do přechodového stavu `State10`. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`.

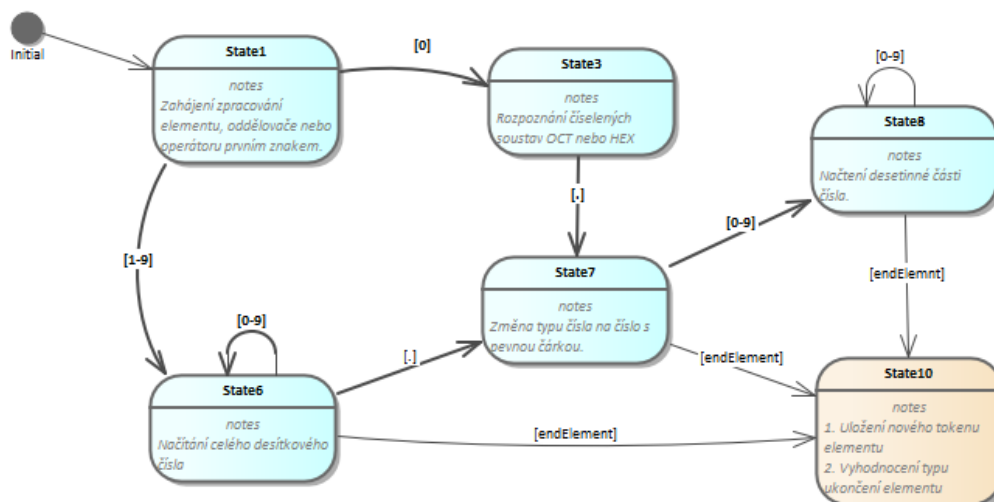
### 5.2.9 State9 - Zjištění, zda bude exponent mít znaménko

Ve stavu se rozhoduje podle došlého znaku, zda bude exponent se znaménkem nebo bez něj. V případě, že došel znak plus „+“ nebo minus „-“ se přejde se do stavu `State11`. Když došly znaky číslic „0“ až „9“ přejde se do stavu `State12`, který zahájí načítání hodnoty exponentu. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`.

### 5.2.10 State10 - Přechodový stav, který zjistí zda byl oddělovač nebo operátor

Tento stav je přechodový. On totiž nečeká na další znak, protože on už byl načten v předchozím stavu. Mohl být znak operátoru, oddělovače nebo nepovolený znak. Pokud byl v předchozím stavu vytvořen nějaký token, tak ho uloží do seznamu tokenů tj. do objektu třídy `TokenList`. Při výstupu rozhoduje, který ze dvou stavu bude následovat, zda stav `State13`, tj. zpracováním oddělovače, nebo stav `State14`, tj. zpracováním operátoru. Pokud to nebyl oddělovač a ani operátor, vystaví se výjimka `AnalyzerError`.





Obrázek 8: Stavový diagram zpracování čísla s desetinou tečkou

### 5.2.11 State11 - Rozpoznání znaménka exponentu

Stav rozkóduje znaménko exponentu a zapamatuje si ho pro stav `State12`. Po příchodu dalšího znaku z množiny číselných znaků „0“ až „9“, přejde do stavu `State12`. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`.

### 5.2.12 State12 - Načtení hodnoty exponentu

Ve stavu se bude načítat znaky z množiny číselných znaků „0“ až „9“, které se budou převádět na hodnotu exponentu.

Stav se opustí, když přijde operátor nebo oddělovač, což v diagramu je naznačeno hranou s podmínkou `endElement`. V tom případě před ukončením stavu se vytvoří token podle třídy `TokenDoubleNumber`, který se uloží do seznamu tokenů až při vstupu do přechodového stavu `State10`. číselná hodnota reálného se sestaví z dílčích hodnot získaných ve stavech `State6`, `State8` a `State6`. Pokud přijde jiný znak, vystaví se výjimka `AnalyzerError`.

### 5.2.13 State13 - Zpracování oddělovače

Jedná se o přechodový stav, v kterém se nečeká na načtení dalšího znaku, protože znak oddělovače byl již načten. Odpovědností stavu je zjistit o jaký oddělovač se jedná, vytvořit objekt s tokenem podle třídy `TokenSeparator` a do seznamu tokenů uložit jeho referenci.

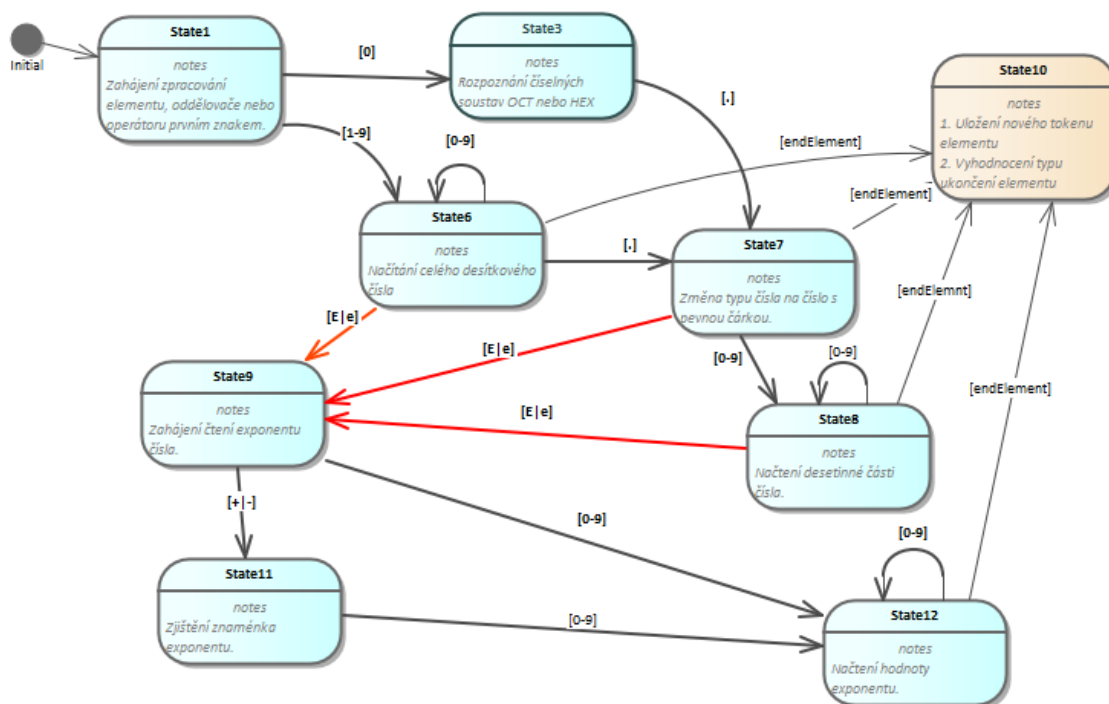
### 5.2.14 State14 - Zpracování operátoru

Jedná se o přechodový stav, v kterém se nečeká na načtení dalšího znaku, protože znak operátoru byl již načten. Odpovědností stavu je zjistit o jaký operátor se jedná, vytvořit objekt s tokenem podle třídy `TokenOperator` a do seznamu tokenů uložit jeho referenci.

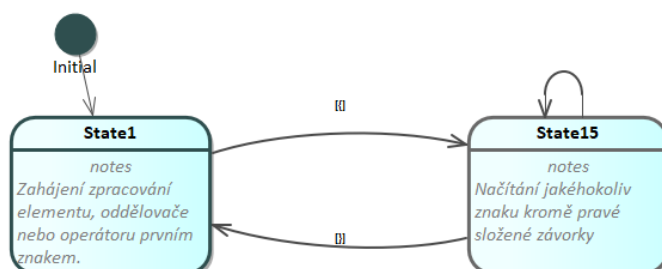
### 5.2.15 State15 - Čtení komentáře

Nejednodušší je rozpoznání komentáře viz diagram 10. Je-li automat ve stavu `State1` a přijde-li znak levá složená závorka `{` jsou další znaky až do příchodu pravé složené závorky `}` ignorovány `}`.





Obrázek 9: Stavový diagram zpracování čísla s exponentem



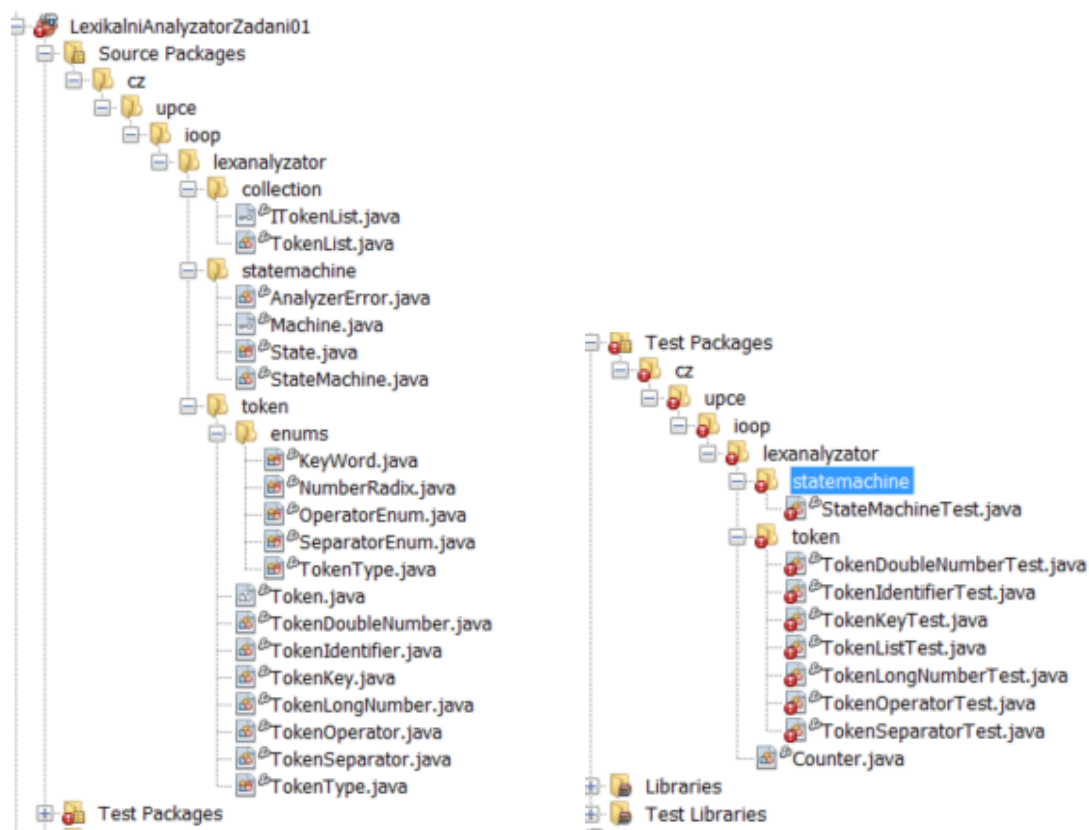
Obrázek 10: Stavový diagram čtení komentáře

## 6 Postup

Součástí tohoto zadání semestrální práce je zkomprimovaná složka s projektem.

Struktura projektu je na obrázku 11. V jeho levé části 11a jsou zobrazeny třídy lexikálního analyzátoru a v pravé části 11b je uveden seznam testů.

Seznam testů 11b má u jednotlivých testovacích tříd červené terčíky, které signalizují chyby v kódu. Ty chyby nejsou chybami testovacích tříd, ale jsou způsobeny tím, že třídy s tokeny nemají nastavenou správnou dědičnost.



(a) Seznam tříd

(b) Seznam testu

Obrázek 11: Struktura projektu

Doporučený postup:

1. Správně nastavit dědičnost mezi třídami s tokeny. Tím by se měly odstranit chyby v testech.
2. Ověřit všechny potomky třídy `Token` testy.
3. Implementovat třídu `TokenList` a ověřit testem `TokenListTest`.
4. Implementovat třídu `StateMachine` postupně tak, aby test `StateMachineTest` hlásil co možná nejméně chyb. Nejlépe žádnou.
5. Podle sekvenčního diagramu 2. Sekvenční diagram lexikálního analyzátoru doplňte a implementujte třídu `LexikalniAnalyzatorMain`.
6. Podle požadavku 4.2.NR8 Ověření programu: ověřte činnost programu.

## 7 Plán konfigurace

### 7.1 Nástroje pro podporu řízení konfigurací

V projektu se mohou využívat tyto verzovací nástroje:

1. Subversion (Windows 32/64-bits) - Klient verzovacího systému „Subversion (SVN)“ pro příkazový řádek
2. Apache™ Subversion® - Stránky se softwarovým projektem verzovacího systému „Subversion“. Tuto stránku mohou využít ti, kteří nepoužívají operační systém Windows.
3. VisualSVN Server (Windows 32/64-bits) - Tento server je nainstalován na serveru

`https:\\fei-svn.upceucebny.cz/`

4. TortoiseSVN - Klient Apache™ Subversion (SVN) ®pro operační systém Windows, který je implementován jako rozšíření „Windows Shell“. To znamená, že je integrován do operačního systému tak, že ho může využívat jakýkoliv prohlížeč souborů. Při použití tohoto prostředku je nutné dbát na to, aby se neukládaly dočasné soubory, jako jsou například soubory s rozšířením `class`.

### 7.2 Základní pravidla a procedury

Na fakultě je provozován server SVN (Subversion) pro účely studentů bakalářského studia. Každý student bude moci využívat úložiště tohoto serveru po celou dobu studia. Jenom na některých předmětech bude toto používání povinné. Jinak studenti mohou toto úložiště používat k uchování softwarových projektů i s jiných předmětů. Není dovoleno ukládat rozsáhlé binární soubory, jako jsou obrázky, videa apod.

Pro připojení z domova nebo přes Eduroam je nutné mít počítači nainstalovaného a spuštěného klienta VPN UPCE

#### 7.2.1 Prohlížení složek studenta na úložišti SVN

Kdy se tento postup použije:

- K získání platné adresy pro konfiguraci připojení projektu v NetBeans s úložištěm.
- K prohlédnutí struktury souborů na úložišti SVN studenta.
- Ke kontrole, zda došlo k uložení správného obsahu zdrojových souborů projektu.

Postup:

1. V internetovém prohlížeči nastavte adresu úložiště SVN:  
`https://fei-svn.upceucebny.cz/svn/bsxx/prijmeni_jmeno_ixxxxx`  
kde `prijmeni_jmeno_ixxxxx` je příjmení, jméno a identifikační číslo studenta. Příjmení a jméno jsou bez diakritiky.  
Když si nejste jisti vaším identifikačním číslem, můžete začít na adrese  
`https://fei-svn.upceucebny.cz/`  
a pokračovat ve vyhledání své složky na úložišti.
2. Po zadání adresy se objeví okno k ověření studenta. Je nutné zadat přihlašovací údaje do domény UPCE.
3. Případně se objeví okno s potvrzením, zda přijímáte nedůvěryhodný certifikát. Odsouhlaste a zaškrtněte, že to přijímáte trvale.
4. Po zobrazení vaší složky, zkontrolujte, zda vaše složka obsahuje větve `ipalp`, `izapr` a `iop` a případně další větve.

5. Přejděte na větev `ipalp` a tuto adresu vložte do schránky. Tuto adresu lze poté použít při nastavení přístupu k verzovacímu serveru v ostatních programech. Především se to týká nastavení v **NetBeans**.
6. Jakýkoliv problém s úložištěm SVN nahlaste učiteli na cvičení nebo na e-mail správce serveru `Karel.Simerda@upce.cz` nebo `Michael.Bazant@upce.cz`

### 7.2.2 Import – První vložení souborů projektu v NetBeans do úložiště SVN

Kdy se tento postup použije:

- Pouze k prvnímu odeslání souborů projektu v NetBeans do úložiště SVN. Po té se již používají ostatní příkazy jako jsou `checkout`, `commit` nebo `update`.

Postup:

1. Ve vývojovém prostředí **NetBeans** vytvořte nový projekt.
2. Pravým tlačítkem myši na kořenu nově vytvořeného projektu vyvolejte kontextové menu a zvolte položku **Versioning**.
3. Dále zvolte v otevřeném podmenu příkaz **Import into Subversion Repository...**
4. Po otevření okna **Import** ve třech krocích inicializujte připojení k úložišti a import souborů projektu.
  - (a) V kroku **Subversion Repository** nastavte URL lokalizaci vaší složky na fakultním serveru SVN a přihlašovací údaje vašeho univerzitního účtu. Doporučuje se povolit uložení přihlašovacích údajů.
  - (b) V kroku **Repository Folder** nastavte vyberte nebo přepište složku do které chcete uložit soubory projektu. Aby se povolil další krok, musíte vyplnit okno **Specify the Message** vhodným textem o importu.
  - (c) V třetím kroku se v novém okně vypíše seznam souborů, které se budou nově přenášet do úložiště. Zde je možné provést poslední úpravy, které souboru se budou přenášet. Nedoporučuje se tento seznam měnit, pokud nejsou ve složkách projektu nějaké vaše pracovní soubory. NetBeans si své pracovní soubory odebírá z obsahu složek projektu automaticky.
5. Odevzdání souborů do úložiště potvrďte stisknutím tlačítka **Finish**. Tímto tlačítkem můžete tento třetí krok vynechat a provést přenos souborů projektu do úložiště bez kontroly seznamu souborů.

### 7.2.3 Commit – Průběžné odevzdání souborů projektu do úložiště SVN

Kdy se tento postup použije:

- Kdykoliv když student uzná z vhodné uložit danou verzi zdrojových souborů.
- Když končí práci na dané počítači, tj. na cvičení nebo mimo něj.

Postup:

1. Ve vývojovém prostředí **NetBeans** otevřete rozpracovaný projekt.
2. Pravým tlačítkem myši na kořenu nově vytvořeného projektu vyvolejte kontextové menu a zvolte položku **Subversion**.
3. Dále zvolte v otevřeném podmenu příkaz **Commit**.
4. Po otevření okna **Commit** překontrolujte seznam modifikovaných souborů určených k odeslání na úložiště SVN.
5. Případně doplňte zprávu o důvodu odeslání modifikovaných souborů do úložiště.
6. Odešlete modifikované soubory stisknutím tlačítka **Commit**.

### 7.2.4 Update – Občerstvení pracovní složky s projektem obsahem úložiště SVN

Kdy se tento postup použije:

- Když chceme získat aktuální stav souborů z úložiště SVN. To může být typicky v takovém případě, že jsme na jednom počítači uložili stav projektu a potom jsme přešli k jinému počítači, kde již máme tentýž projekt připojen k úložišti.
- Tento příkaz se v praxi využívá k aktualizaci projektu, když na projektu současně pracuje více programátorů. V našem případě se to může stát, když učitel do projektu konkrétního studenta vloží své připomínky.

Postup:

1. Ve vývojovém prostředí **NetBeans** otevřete rozpracovaný projekt.
2. Pravým tlačítkem myši na kořenu nově vytvořeného projektu vyvolejte kontextové menu a zvolte položku **Subversion**.
3. Dále zvolte v otevřeném podmenu příkaz **Update** a dále ještě proveďte další volbu a to již příkaz **Update to HEAD**, tím dojde ke stažení modifikovaných souborů z úložiště SVN o proti lokálnímu stavu projektu.

### 7.2.5 Checkout – Připojení pracovního adresáře k úložišti SVN

Kdy se tento postup použije:

- Když chceme pracovní složku (adresář) na svém počítači připojit ke složce na úložišti SVN a stáhnout z úložiště všechny soubory.
- Když chceme v **NetBeans** se připojit k uloženému projektu na úložišti SVN.

Postup:

1. V **NetBeans** rozvineme menu **Team** a zvolíme položku **Subversion** a nakonec zvolíme položku **Checkout...**
2. Po otevření okna **Checkout** vyplníme textové pole **Repository URL** adresou odkud chceme stáhnout projekt.
3. Vyplníme nebo změníme přihlašovací údaje.
4. Tlačítkem **Next** přejdeme na druhý krok zadávání příkazu **Checkout** kde vyplníme nebo vybereme složku na úložišti SVN s projektem a dále případně změníme umístění lokální složky, kam se požadovaný projekt stáhne.
5. Celý příkaz **Checkout** ukončíme stisknutím tlačítka **Finish**.
6. Nakonec se ještě samotný **NetBeans** zeptá, zda má daný projekt nebo projekty otevřít.

## 7.3 Jmenné konvence

Požaduje se, aby v názvu projektu bylo za názvem produktu bylo za podtržítkem příjmení studenta, který projekt vypracoval. To je z toho důvodu, aby učitel mohl jednoznačně odlišit projekty studentů.

## Reference

- [1] PECINOVSKÝ, Rudolf. *Návrhové vzory: 33 vzorových postupů pro objektové programování*. Computer Press, 2007. ISBN 978-80-251-1582-4.
- [2] PECINOVSKÝ, Rudolf. *Java 8: úvod do objektové architektury pro mírně pokročilé*, Praha: Grada Publishing, 2014. Knihovna programátora (Grada). ISBN 978-80-247-4638-8
- [3] ARLOW, Jim a Ila NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. 2. aktual. a dopln. vyd. Přeložil Bogdan KISZKA. Brno: Computer Press, 2011. ISBN 978-80-251-1503-9.
- [4] LEARN IOOP