Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

Studijní program: Aplikovaná informatika

Obor: Podniková informatika

Doplnění vývojového prostředí BlueJ o funkce využitelné v úvodních kurzech programování

DIPLOMOVÁ PRÁCE

Student : Bc. Oleksandr Matviichuk

Vedoucí : Ing. Rudolf Pecinovský, CSc.

Oponent : Ing. Jarmila Pavlíčková

2014

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze kterých jsem čerpal.

V Praze dne 07.05.2014 .....

Oleksandr Matviichuk

Poděkování

Úvodem bych rád poděkoval svému vedoucímu práce, panu Ing. Rudolfu Pecinovskému, CSc., za odborné vedení, rady a pomoc při tvorbě této práce. Také bych rád poděkoval své manželce Katerině za trpělivost a podporu jak při studiu, tak při tvorbě této práce.

Abstrakt

Tato práce se věnuje problematice výuky programování a její podpoře ve vývojovém prostředí BlueJ. Práce se primárně zaměřuje na úvodní kurzy programování, ale její výsledky lze využít i ve vyšších kurzech. Práce se skládá z části teoretické a praktické.

V teoretické části práce jsou nejprve analyzovány různé metodiky výuky programování. Následně se pro metodiku, která byla vybrána jako nejmodernější, provádí analýza její podpory ve výukovém prostředí BlueJ. Na základě této analýzy byl vytvořen seznam nových funkcí, jejichž implementace do prostředí BlueJ by měla výrazně zlepšit podporu výuku podle vybrané metodiky.

Aktuálnost práce spočívá mimo jiné v tom, že při návrhu nových funkcí do prostředí BlueJ byly zohledněny rovněž novinky, se kterými přišla poslední verze jazyka Java, vydaná méně než dva měsíce před odevzdáním této práce. V práci je navrženo přidat do prostředí BlueJ podporu těch novinek z verze Java 8, které je z pedagogického hlediska přínosné začlenit do výuky v úvodních kurzech programování.

V praktické části práce pak byly navržené nové funkce implementovány do prostředí BlueJ. Práce kromě toho obsahuje uživatelskou příručku s návodem, jak nové funkce používat, a programátorskou příručku s popisem provedených změn.

Klíčová slova

BlueJ, výuka programování, Objects First, Design Patterns First, Architecture First

Abstract

This work is devoted to the problems of teaching programming and its support in integrated development environment BlueJ. First of all it focuses on the basic programming courses, but the results can be used at the advanced courses as well. The work consists of the theoretical and practical parts.

In the theoretical part the various methods of teaching programming are analyzed. Then a method which has been selected as the most modern is getting tested by checking its support in the environment BlueJ.

Based on these analysis, it is offered to add to the BlueJ list of new functions which should greatly improve the teaching of programming according to the chosen method.

Relevance of work lies in the propositions as for new possibilities in BlueJ taking into consideration the innovations of the latest version of Java language (which was released less than two months before this work was handed in). This work suggests adding to BlueJ environment support of the innovations of Java 8, which are useful to include to the teaching of introductory programming courses from educational point of view.

In the practical part the suggested new functions have been implemented to the BlueJ environment. The work also includes instructions for users how to use the new methods and instructions for the programmers, which describes the details of the latest changes.

Keywords

BlueJ, teaching programming, Objects First, Design Patterns First, Architecture First.

Obsah

Úvod

Moderní doba se vyznačuje neustálým zrychlováním změn: na trh se uvádějí stále nové výrobky a služby, vznikají nové firmy (i celá odvětví) a jiné zase zanikají, konkurence narůstá. Jelikož se v současné době téměř žádné solidní podnikání neobejde bez podpory IT služeb, snaží se i sektor informačních a komunikačních technologií tomuto trendu přizpůsobit. Vývojáři aplikačního softwaru musí neustále upravovat vyvíjené systémy, tak, aby vyhověli neustále se měnícím požadavkům zákazníka.

Tato situace zvyšuje nároky na vývojáře softwaru: nyní už nestačí jen umět zakódovat řešení v nějakém programovacím jazyce, je třeba umět navrhnout řešení tak, aby počítalo s budoucími úpravami, bylo snadno rozšiřitelné a udržovatelné s minimálními náklady. Jinak řečeno, od dobrého programátora se v současné době očekává, že bude především architekt IT systémů, nikoliv jen jejich kodér [1].

Na tento trend reagují i vzdělávací instituce, které se zabývají přípravou budoucích programátorů. Mění se přístupy k výuce (co a jak učit, a hlavně z čeho vyjít, odkud začít výklad látky [2]), vznikají nové metodiky výuky programování. Spolu s novými metodikami vznikají a vyvíjejí se příslušné podpůrné nástroje, bez nichž by výuka programování byla mnohem komplikovanější. A stejně jako například výcvik pilotů nezačíná lety na skutečných letadlech, ale používají se nejprve různé letecké trenažéry a simulátory, ani výuku programování není dobře začínat v robustních profesionálních prostředích, která studenty zbytečné zaplaví obrovským množstvím nabízených funkcí. Pro začátečníky existují speciální zjednodušená vývojová prostředí, určená primárně na podporu výuky programování v základních kurzech. Například pro jazyk Java, který je v současné době nejvíce poptávaným programovacím jazykem ze strany zaměstnavatelů [3], takových nástrojů pro začátečníky existuje celá řada: DrJava, Alice, Greenfoot, BlueJ, jGRASP [4]. Tato práce se věnuje vývojovému prostředí BlueJ, které se používá v základních kurzech programování na Vysoké škole ekonomické v Praze i na řadě dalších vysokých škol a univerzit.

Cíl a přínos práce

Cílem práce je na základě analýzy současných trendů a metodik výuky programování a možností vývojového prostředí BlueJ navrhnout a implementovat do daného prostředí nové funkce, využitelné ve vstupních kurzech programování.

Přínosem práce bude upravená verze vývojového prostředí BlueJ, rozšířená o nové funkce, které lépe podporují současné trendy v metodice výuky programování, umožňují zlepšit kvalitu výuky a naučit budoucí programátory to, co od nich skutečně očekávají budoucí zaměstnavatelé.

Komu je práce určena

Práce je primárně určena učitelům programování, kteří dostanou k dispozici lepší podpůrný nástroj (upravené a rozšířené prostředí BlueJ), na základě čehož budou schopni upravit své

přístupy k organizaci výukových programovacích kurzů, připravit lepší výukové pomůcky a materiály a modernizovat výukový proces v souladu se současnými trendy v metodice výuky programování.

Další cílovou skupinou jsou lidé, kteří se zabývají vývojem a zlepšováním prostředí BlueJ, jimž tato práce může posloužit jako podklad pro další úpravy. Za ideální výsledek autor považuje začlenění funkcí, navržených a realizovaných v rámci této práce, do jedné z budoucích oficiálních verzí prostředí BlueJ. Autor a vedoucí této práce budou tým vývojářů BlueJ kontaktovat, ale rozhodnutí o začlenění nových funkcí do oficiální verze bude záležet na vizi tvůrců daného prostředí ohledně jeho budoucnosti, a v neposledně řadě také na kvalitě implementací nových funkcí.

Za třetí cílovou skupinu lze považovat vývojáře jiných nástrojů na podporu výuky programování, a to jak pro Javu, tak i pro jiné programovací jazyky. Daný text může posloužit jako zdroj inspirace, na jehož základě mohou vývojáři jiných výukových nástrojů převzít do svých projektů a realizovat některé z nových funkcí, implementovaných v rámci této práce do prostředí BlueJ.

Předpoklady a omezení

Práce předpokládá znalost zásad objektově orientovaného programování a související terminologie. Pojmy jako třída, instance, tovární metoda, návrhový vzor atd. se zde používají bez vysvětlování jejich významu, jelikož se předpokládá, že čtenář pojmy zná.

Dalším předpokladem je středně pokročilá znalost programovacího jazyka Java, nutná pro pochopení implementační části.

Struktura práce

Práce se skládá ze čtyř částí. V první části autor popisuje vývoj přístupů a metodik výuky programování a následně analyzuje možnosti prostředí BlueJ z hlediska podpory moderních trendů v této oblasti.

V druhé části autor navrhuje přidat do prostředí BlueJ nové funkce, které budou lépe podporovat současné přístupy a metodiky ve výuce programování.

Třetí část obsahuje uživatelskou příručku, která popisuje a pomocí snímků obrazovek také graficky znázorňuje, jak nové funkce používat.

Poslední, čtvrtá část obsahuje implementační čili programátorskou příručku, která dopodrobna popisuje veškeré provedené zásahy do zdrojových kódů BlueJ. Tato část je primárně určena pro programátory, kteří budou chtít integrovat nové funkce do dalších verzí BlueJ. Jak již bylo uvedeno v předchozí kapitole, za ideální stav by autor považoval případ, kdy by tvůrci BlueJ jednou začlenili nové funkce, vytvořené v rámci této práce, do svého úložiště zdrojových kódů a tyto funkce se pak staly součástí všech následujících verzí BlueJ. Pokud však autoři BlueJ z jakýchkoliv důvodů nebudou chtít tak učinit, proces integrace změn bude

třeba opakovat s vydáním každé nové verze BlueJ. Podrobný popis všech provedených změn má tedy za cíl tuto práci maximálně usnadnit.

Rešerše a informační zdroje

Rešerše se skládá ze dvou částí, které mají výrazně odlišnou dostupnost informačních zdrojů: analýza metodik výuky programování a analýza vnitřní architektury prostředí BlueJ.

O výuce programování pojednává obrovské množství zdrojů (jak v češtině, tak i v angličtině), a to v podobě učebnic [5][6], závěrečných prací [4], příspěvků na konferencích a v odborných časopisech [1][2]. Problém proto spočívá ve výběru vhodných a relevantních zdrojů z obrovského množství volně dostupných pramenů, jež ale často obsahují duplicitní informace.

S analýzou vnitřní architektury BlueJ je situace naprosto jiná: zdrojů je naopak jen velmi málo. Jediná publikace v češtině, věnovaná dané problematice, je bakalářská práce [7], která ale představuje jen úvod do architektury tohoto projektu, což je sice velmi důležité, ale ne postačující pro rozsáhlejší úpravy. Pro úplnost je však třeba dodat, že kromě samotného textu tato bakalářská práce obsahuje přílohy, v nichž bylo možné dohledat velké množství UML[1] diagramů, které byly následně použity při analýze. V angličtině se podařilo najít dizertační práci [8], v jejímž rámci byla do BlueJ přidána podpora jednotkového testování. O vnitřní architektuře BlueJ se tato práce zmiňuje jen okrajově.

Za této situace nutno konstatovat, že hlavním informačním zdrojem pro analýzu architektury BlueJ se stal samotný zdrojový kód daného projektu.

Závěrem této části je záhodno ještě dodat, že vývoj a úpravy IDE[2] vyžadují od vývojářů poněkud hlubší znalosti než vývoj běžného aplikačního softwaru. Proto bylo občas třeba podívat se do hloubky, například jak fungují v jazyce Java reflexe[3] a debugování[4], dohledat něco přímo ve specifikaci jazyka [9] nebo ve specifikaci virtuálního stroje [10].

[1] UML – grafický modelovací jazyk, viz terminologický slovník
[2] IDE – integrované vývojové prostředí, viz terminologický slovník
[3] Reflexe – získávání informace o třídách a jejich prvcích za běhu programu, viz terminologický slovník
[4] Debugování – proces odstraňování chyb v programu, viz terminologický slovník

1. Analýza metodiky výuky programování a její podpora v BlueJ

1.1 Přístupy k výuce programování: odlišnosti a společné rysy

Programování je dynamická disciplína, která se neustále vyvíjí: prosazují se nová programovací paradigmata a některá jiná jsou opouštěna, vznikají nové programovací jazyky, nová vývojová prostředí, nové frameworky a knihovny. Vzdělávací instituce v oblasti IT na tyto změny reagují tak, že vytvářejí nové metodiky a přístupy k výuce programování. Přístupy k výuce programování lze klasifikovat podle toho, čím vlastní výuka začíná, tedy co který princip upřednostňuje – protože to nejdůležitější by se studenti měli dozvědět co nejdřív [11]:

- Nejdříve hardware (*Hardware-first*)
- Nejdříve algoritmy (*Algorithms-first*)
- Nejdříve příkazy (*Imperative-first*)
- Nejdříve funkce (*Functional-first*)
- Nejdříve objekty (*Objects-first*)
- Nejdříve celkový přehled (*Breadth-first*)

Podrobný rozbor všech přístupů není předmětem této práce. Podstatné je podchytit trend, který spočívá v tom, že novější přístupy obvykle pracují na vyšší úrovni abstrakce a používají větší „stavební kameny“ než jejich předchůdce. Například na začátku počítačové éry vývojáři psali programy na úrovni strojových kódů, a proto výuku programování bylo nutné začínat hardwarem. Bez znalosti „železa“ nebylo možné programovat. V současné době je většina programátorů od hardwarové platformy odstíněna, a ve studijných plánech některých informatických oborů není ani jeden předmět zaměřený na hardware [12]. Pro většinu současných programátorů má z praktického hlediska větší význam znalost knihoven a frameworků než znalost toho, jak funguje procesor v jejich notebooku a jakou má sadu strojových instrukcí.

Bohužel lze ale v této souvislosti konstatovat nepříjemný fakt, že totiž vývoj metodiky výuky programování za vývojem samotného programování zaostává. Na školách je poměrně častým jevem, že se vyučuje určitý styl programování, který sice v době svého vzniku před x lety byl progresivní a perspektivní, ale nyní je už považován za zastaralý a překonaný [11].

Toto tvrzení autor práce může doložit na základě své zkušenosti. V letech 2002–2006 vystudoval střední odbornou školu se zaměřením na informatiku. V době, kdy hlavním proudem v praxi bylo objektově orientované programování (dále jen OOP), roční kurz programování, vyučovaný na této škole, 90 % času věnoval programování strukturovanému. Výuce OOP byly věnovány jen poslední 2–3 přednášky na konci kurzu, tedy navíc v době, kdy většina studentů již spíše přemýšlí o tom, co bude dělat o prázdninách, než aby se soustředila na pochopení nové látky.

Výsledkem takového přístupu k výuce programování bylo, že naprostá většina studentů neuměla OOP vůbec a zbývající část mohla jen – v nejlepším případě – zabalit své strukturované programy do tříd a objektů. Takové programy čistě formálně mohly vypadat zvenku jako napsané v objektově orientovaném stylu. Ve skutečnosti ale zevnitř to byl „klasický“ strukturovaný kód.

Programovat opravdu objektově se pak autor naučil až na Vysoké škole ekonomické v Praze, když absolvoval kurz „4IT101 – Základy programování“, vyučovaný přístupem *Objects First* [13], kterému je věnována další kapitola.

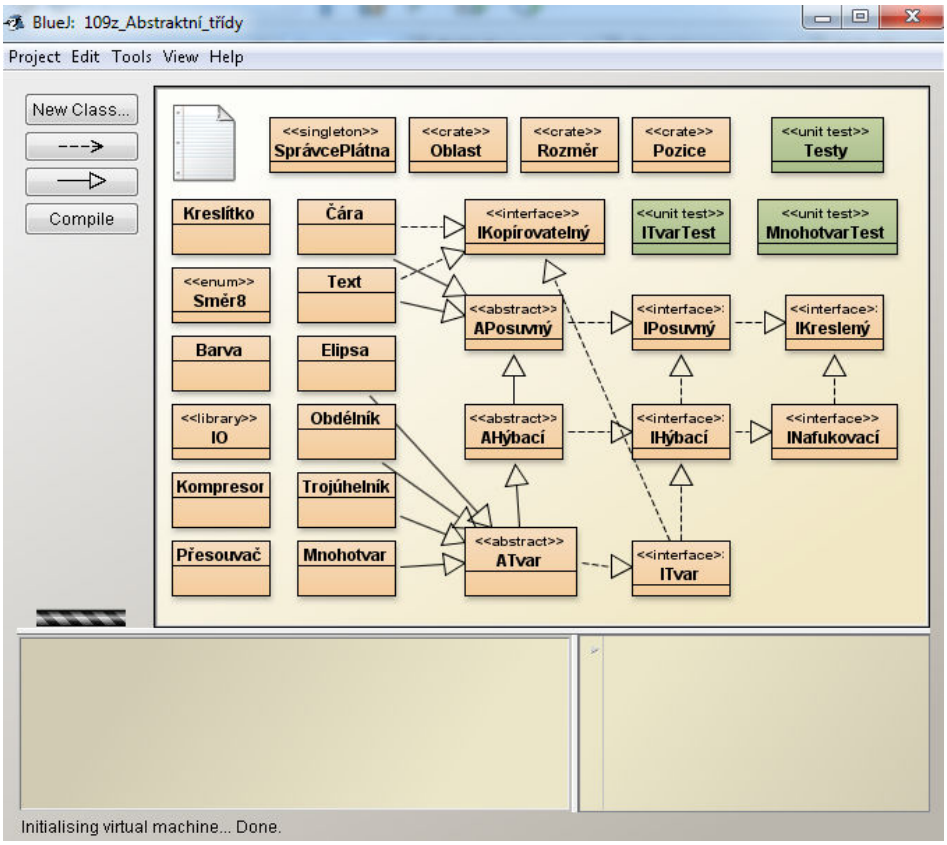
1.2 Metodika Objects First a prostředí BlueJ

Přístup k výuce OOP, při němž se studenti na začátku učí strukturované programování, na které pak navazuje OOP, lze považovat za kontraproduktivní, protože více problémů přináší, než řeší. Studenti, kteří už mají nějakou představu, jak se má programovat, na úrovni podvědomí odmítají vstřebávat novou látku, která je v rozporu s jejich předchozí představou. Je mnohem snadnější naučit OOP od nuly, než přeškolovat studenta-programátora ze strukturovaného na objektově orientované programování. Proces přeškolení na OOP je tím obtížnější, čím více má student zkušeností se strukturovaným programováním [14].

Proto se na přelomu století objevila nová metodika *Objects First*, která začíná výuku programování od vysvětlování pojmů `třída` a `objekt`. Na podporu této metodiky bylo rovněž vytvořeno nové vývojové prostředí BlueJ, jehož náhled je vidět na obrázku 1.

Na obrázku 1 je vidět, že většinu pracovní plochy zabírá zjednodušený diagram tříd, který zobrazuje veškeré třídy aktuálního projektu. Standardní třída je zobrazena pomocí obdélníku, v jehož horní části je uveden název třídy. Některé druhy tříd mají navíc nad názvem třídy ještě popisek v dvojitých ostrých závorkách, takzvaný stereotyp, který upozorňuje na to, že se jedná o speciální případ třídy. Kromě stereotypu lze ještě speciální případ třídy v diagramu zvýraznit pomocí jiné barvy. Ve výchozím nastavení má jinou barvu pouze testovací třída. Pro ostatní stereotypy lze však zvláštní barvy nastavit ručně úpravou konfiguračního souboru *bluej.defs.*

V dolní části plochy se nachází oblast zásobníku odkazů, do něhož se ukládají odkazy na vytvořené objekty. V pravém dolním rohu je umístěn příkazový panel, který uživateli umožňuje psát a vyhodnocovat jednotlivé příkazy v jazyce Java. Tento panel je podle výchozích nastavení skrytý. Je to proto, že základní princip prostředí BlueJ spočívá v tom, aby uživatel komunikoval s programem v interaktivním režimu, který je realizován přes GUI prvky (kontextové nabídky tříd a objektů se seznamem dostupných metod a dialogová okna na zadávání parametrů a zobrazení výsledků), nikoliv psaním textových příkazů. Díky takovému přístupu se programátor-začátečník učí přemýšlet v rovině objektově orientovaných pojmů „třída“, „objekt“, „zaslání zprávy“, a ne v rovině příkazů konkrétního programovacího jazyka. Jinými slovy interaktivní režim nutí studenta v prvé řadě přemýšlet o objektově orientovaném návrhu řešení úlohy, nikoliv o samotném kódování.



Obr. 1 – Náhled vývojového prostředí BlueJ

Bohužel autoři metodiky *Objects First* sami nedocenili sílu takového přístupu a po několika úvodních přednáškách, kde výuka probíhá v interaktivním režimu, začínají klasický výklad konstrukcí programovacího jazyka [1].

Další nedostatek metodiky *Objects First* spočívá v tom, že sice její autoři začínají výkladem tříd a objektů, ale další důležitý OOP pojem „rozhraní“ odkládají až na závěr kurzu, a o návrhových vzorech se nezmiňují vůbec [15]. Právě proto kolektiv pedagogů Vysoké školy ekonomické v Praze přišel s nápadem modifikovat metodiku *Objects First* a na jejím základě vytvořil metodiku *Design Patterns First* [16], která pak byla po určitém historickém vývoji přejmenována na *Architecture First* [1]. Této nové metodice je věnována další kapitola.

1.3 Metodika Architecture First a její podpora v prostředí BlueJ

Metodika *Architecture First* začíná výuku programování rozborem základních principů budování architektury objektově orientovaných programů. Hlavní myšlenkou metodiky je, že zpočátku by studenti na praktických cvičeních neměli sami psát kód, místo nich by to měl dělat generátor kódu, zabudovaný do výukového vývojového prostředí. Metodika se snaží odložit okamžik, kdy studenti začnou psát kód sami, až do doby, kdy složitost probírané látky překročí možnosti generátoru kódu. Takový přístup umožňuje studentům zůstávat co nejdéle v hladině architektury a při uvažování nad řešením úlohy neomezovat své návrhy na známé jazykové konstrukce. Jinými slovy v první fázi metodika učí studenty přemýšlet jenom o

návrhu řešení, nikoliv o tom, jak takový návrh zakódovat. Kódování je v první fázi v kompetenci generátoru kódu.

Další charakteristikou metodiky je časné začlenění návrhových vzorů do výuky, a to už v první fázi, kdy studenti ještě nepíšou kód sami, ale nechávají to na generátoru. Takže s prvním návrhovým vzorem se studenti seznámí dříve než s prvním příkazem. Proto tato metodika původně měla název *Design Patterns First*.

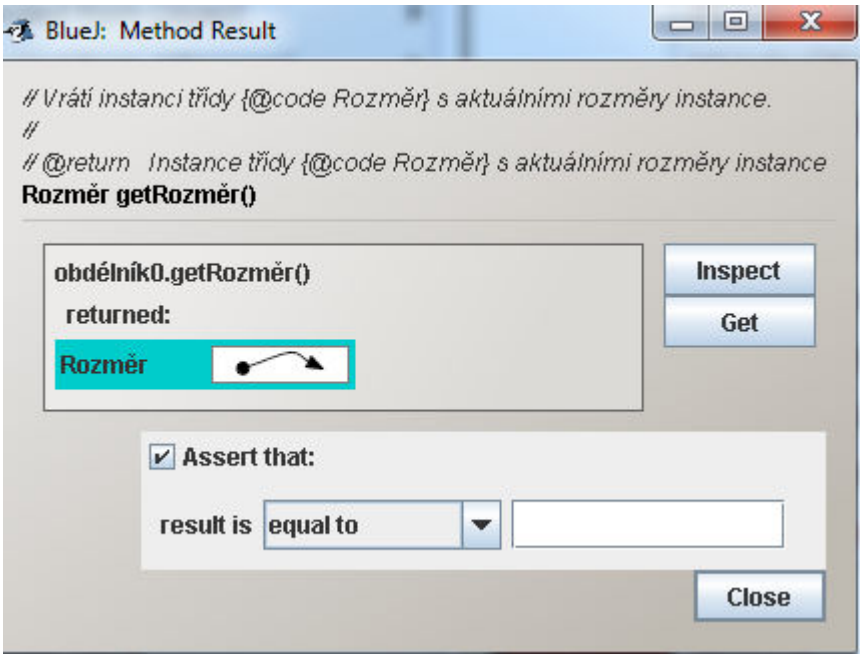
Dále se podíváme, jak prostředí BlueJ podporuje výuku této metodiky z hlediska dvou již uvedených bodů: podpory generování kódu a podpory návrhových vzorů.

Začneme od druhého bodu (který ale historicky byl prvním), tedy od podpory návrhových vzorů. V podstatě lze konstatovat, že téměř žádná speciální podpora pro návrhové vzory v prostředí BlueJ není. Jediné, jak už bylo zmíněno v předchozí kapitole, je zvýraznění některých speciálních druhů tříd v UML diagramu pomocí stereotypů, případně pomocí jiné barvy, definované uživatelem. Mezi takové speciální třídy patři *Výčtový typ* (*Enum*), který je návrhovým vzorem [17].

Co se týče podpory metodiky *Architecture First* z hlediska generování zdrojových kódů, tady je situace mnohem lepší. Sice BlueJ nemá generátor kódu určený přímo na podporu této metodiky, ale v rámci již zmíněné dizertační práce [8] byl do prostředí BlueJ implementován generátor kódu pro podporu jednotkového testování. Autoři metodiky *Architecture First* se ho pokusili přizpůsobit pro své účely. Jelikož tento generátor byl původně navržen pro zcela jiný účel (generování kódu testovacích metod), generování kódu pro metodiku *Architecture First* je podporováno s velkým množstvím omezení, vyplývajících z původního zaměření generátoru:

- metody lze přidávat pouze do testovacích tříd,
- nelze zadávat parametry metod – všechny nové metody jsou bezparametrické,
- nelze zadávat viditelnost metody – všechny nové metody jsou veřejné,
- nelze zadávat návratový typ – všechny nové metody jsou „`void`“,
- nelze generovat statické metody – všechny nové metody jsou instanční,
- nelze přidat deklaraci metody (do rozhraní nebo abstraktní třídy).

Tyto vady autor práce považuje za podstatné, protože výrazně omezují možnosti výuky programování pomocí metodiky *Architecture First.* Kromě toho by chtěl uvést ještě jednu drobnost, která je ovšem povahy spíše jen kosmetické: pokud je BlueJ v režimu nahrávání testovacích metod, pak ve všech dialogových oknech na zobrazení návratových hodnot je dole vidět tak zvaný „assert“ panel (viz obr. 2), určený k ověření návratové hodnoty. Tento panel je nezbytný při nahrávání testovacích metod, ale je irelevantní při nahrávání běžných metod.



Obr. 2 – Dialogové okno s návratovou hodnotou metody a „assert“ panelem

Další kapitola této práce analyzuje, jak odstranit výše uvedené vady, případně přidat nějaké funkce navíc s cílem zlepšit podporu výuky podle metodiky *Architecture First* v prostředí BlueJ.

2 Návrh nových funkcí do prostředí BlueJ pro lepší podporu metodiky Architecture First

2.1 Nové stereotypy pro vybrané návrhové vzory

Pro podporu návrhových vzorů v BlueJ asi nejde vymyslet nic lepšího, než inspirovat se příkladem *Výčtového typu* a označovat je pomocí speciálních stereotypů. Samozřejmě to lze jen pro ty návrhové vzory, které se skládají pouze z jedné třídy. Po dohodě s vedoucím této diplomové práce byly vybrány následující tři návrhové vzory: *Knihovní třída (Library Class)*, *Jedináček (Singleton)* a *Přepravka (Crate)*. Podrobnosti o těchto vzorech lze najít ve specializované literatuře, například [17].

Problém ale je v tom, že literatura obsahuje popis návrhových vzorů v podobě volného textu, určeného pro interpretaci člověkem (čtenářem knihy). Pro strojovou detekci vybraných návrhových vzorů je třeba volný popis každého vzoru převést do podoby několika striktních pravidel, která pak bude možné zakódovat do prostředí BlueJ.

Další problém spočívá v tom, že návrhové vzory lze implementovat více způsoby a je velmi náročné ohlídat je všechny. A navíc implementace různých návrhových vzorů se občas mohou navzájem překrývat. Například vzor *Jedináček* kromě všech způsobů, uvedených v [17], lze ještě implementovat jako *Výčtový typ* s jednou instancí [18]. Ale *Výčtový typ* už má v BlueJ přidělený stereotyp, takže zřejmě nemá cenu v případě pouze jedné instance přepisovat ho jiným, novým stereotypem.

Proto se musíme smířit s tím, že automatická detekce třech vybraných návrhových vzorů nebude pokrývat 100 % všech jejich možných implementací. Autor textu odhaduje, že pokrytí bude kolem 80 %, a považuje to za celkem postačující pro účely výuky v základních kurzech programování.

Nyní pro každý vybraný návrhový vzor uvedeme jeho učebnicovou definici podle [17], a pak striktní pravidla, podle kterých bude fungovat detekce daného vzoru v prostředí BlueJ.

„Knihovní třída slouží jako obálka pro soubor statických metod. Protože k tomu nepotřebuje vytvářet instance, je vhodné jejich vytváření znemožnit.“

- třída má pouze jeden soukromý bezparametrický konstruktor
- třída má pouze statické členy
- třída nemá žádnou tovární metodu a žádný atribut s odkazem na instanci dané třídy

„Jedináček specifikuje, jak vytvořit třídu, která bude mít nejvýše jednu instanci.“

- třída má pouze jeden soukromý konstruktor
- třída má přesně jeden atribut s odkazem na instanci dané třídy
- tento atribut je nesoukromý nebo existuje nesoukromá tovární metoda

„Vzor Přepravka využijeme při potřebě sloučení několika samostatných informací do jednoho objektu, prostřednictvím nějž je pak možno tyto informace jednoduše ukládat nebo přenášet mezi metodami.“

- třída má nesoukromé konstantní instanční atributy
- třída má nesoukromý konstruktor

2.2 Pokročilý generátor kódu

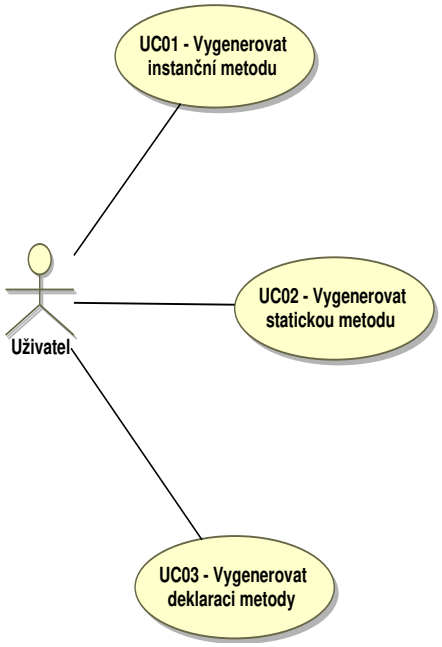
Požadavky na pokročilý generátor kódu vyplývají ze seznamu omezení generátoru testovacích metod, uvedeného v kapitole 2.3. Takže od upraveného generátoru se očekává, že bude schopný:

- přidávat metody do libovolných tříd,
- podporovat generování metod s libovolným počtem parametrů,
- podporovat generování metod s libovolným modifikátorem přístupu,
- podporovat generování metod s návratovou hodnotou,
- podporovat generování statických metod,
- podporovat generování hlaviček metod do abstraktních tříd a rozhraní,
- podporovat generování defaultních metod do rozhraní.

Tyto požadavky v podstatě lze realizovat v rámci třech základních případů užití, zobrazených na obrázku 3. Samozřejmě každý případ užití má kromě hlavního úspěšného scénáře ještě pár alternativních, jako je například zadání chybných vstupních parametrů nebo již existující metoda se stejnou signaturou.

Rozepíšeme nyní případy užití po jednotlivých krocích. Společné předpoklady pro všechny případy:

- aktérem je uživatel,
- systémem je prostředí BlueJ,
- jazyk GUI prostředí je nastaven na angličtinu.



Obr. 3 – Diagram základních případů užití generátoru kódu

UC01 – Vygenerovat instanční metodu

Popis: uživatel nahrává instanční metodu.

Vstupní podmínky: v zásobníku odkazů je minimálně jeden objekt.

Tab. 1: UC01 – Vygenerovat instanční metodu – hlavní scénář

Krok	Role	Akce
1	Aktér	V místní nabídce objektu zvolí položku „*Record method*“.
2	Systém	Zobrazí dialogové okno na zadání vstupních dat: viditelnost, název a jeden parametr metody. Všechna políčka jsou prázdná.
3	Aktér	Vyplní viditelnost a název metody, pak, pokud je metoda bezparametrická, zmáčkne tlačítko „-“, čímž odstraní řádek na zadání vstupního parametru, a pokračuje krokem 5; pokud má metoda více než jeden parametr, zmáčknutím tlačítka „+“ přidá další řádky podle počtu parametrů.
4	Aktér	Pro každý parametr vyplní jeho typ, název a výraz pro počáteční hodnotu.
5	Aktér	Zmáčkne tlačítko „*OK*“.
6	Systém	Provede formální kontrolu vstupních dat: zkontroluje, že všechna políčka jsou vyplněna a názvy metody a parametrů jsou platné Java identifikátory. Pokud jsou chyby, viz **Vygenerovat instanční metodu – alternativní scénář (chybná vstupní data).**
7	Systém	Zkontroluje, že v cílové třídě není metoda se stejnou signaturou. Pokud je, viz **Vygenerovat instanční metodu – alternativní scénář (existující metoda).**

8	Systém	Pokud metoda je parametrická, spustí vyhodnocování výrazů pro počáteční hodnoty parametrů. Pokud vyhodnocování skončilo chybou, viz **Vygenerovat instanční metodu – alternativní scénář (chybná vstupní data).**
9	Systém	Uzavře modální okno, pak přidá do zásobníku odkazů: aktuální instanci pod jménem „`_this`“, všechny atributy cílové třídy včetně soukromých, počáteční hodnoty všech parametrů. Pak přepne prostředí do režimu „*recording*“.
10	Aktér	Provádí interakce, které chce zapsat jako příkazy do těla nahrávané metody. Na konci zmáčkne tlačítko „*End*“.
11	Systém	Zobrazí dialogové okno pro zadání návratové hodnoty. Přednastaveno na „`void`“.
12	Aktér	Zvolí návratovou hodnotu nebo nechá „`void`“. Zmáčkne tlačítko „*OK*“.
13	Systém	Pokud už existuje metoda se stejnou signaturou, nový kód přepíše existující, jinak se nový kód vloží na konec třídy. Ukončí režim „*recording*“.

Tab. 2: UC01 – Vygenerovat instanční metodu – alternativní scénář (chybná vstupní data)

Krok	Role	Akce
1	Systém	Zobrazí chybovou hlášku ve stejném modálním okně a vyzve uživatele k úpravě chybných vstupních dat.
2	Aktér	Opraví chybná data, pak pokračuje krokem 5 hlavního scénáře.

Tab. 3: UC01 – Vygenerovat instanční metodu – alternativní scénář (existující metoda)

Krok	Role	Akce
1	Systém	Zobrazí dialogové okno s upozorněním, že metoda se stejnou signaturou už existuje, a zeptá se, zda ji uživatel chce nahradit nebo ne.
2	Aktér	Pokud uživatel zvolí možnost „*Replace*“, pak pokračuje krokem 8 hlavního scénáře, jinak krokem 3.

UC02 – Vygenerovat statickou metodu

Popis: uživatel nahrává statickou metodu.

Vstupní podmínky: v projektu je minimálně jedna netestovací třída.

Tab. 4: UC02 – Vygenerovat statickou metodu – hlavní scénář

Krok	Role	Akce
1	Aktér	V místní nabídce třídy zvolí položku „*Record static method*“, případně „*Record static/default method*“ u rozhraní.
2–8	Aktér, Systém	Stejně jako v **UC01 – Vygenerovat instanční metodu.**
9	Systém	Uzavře modální okno, pak přidá do zásobníku odkazů: všechny statické atributy cílové třídy, včetně soukromých, a počáteční hodnoty všech parametrů. Pak přepne prostředí do režimu „*recording*“.
10–13	Aktér, Systém	Stejně jako v **UC01 – Vygenerovat instanční metodu**, rozdíl jen v tom, že v případě úspěšného dokončení se metoda vloží do kódu s modifikátorem `static`. U rozhraní systém před vložením vygenerovaného kódu ještě nabídne uživateli možnost vybrat mezi `static` a `default`. Do kódu se pak metoda vloží se zvoleným modifikátorem.

UC03 – Vygenerovat deklaraci metody

Popis: uživatel nahrává statickou metodu.

Vstupní podmínky: v projektu je minimálně jedna abstraktní třída nebo jedno rozhraní.

Tab. 5: UC03 – Vygenerovat deklaraci metody – hlavní scénář

Krok	Role	Akce
1	Aktér	V místní nabídce objektu zvolí položku „*Add method declaration*“ u rozhraní, případně „*Add abstract method declaration*“ u abstraktních tříd.
2	Systém	Zobrazí dialogové okno na zadání vstupních dat: viditelnost, název a jeden parametr metody. Všechna políčka jsou prázdná. U parametru se zobrazují pouze políčka pro typ a jméno, políčko pro počáteční hodnotu se nevykresluje.
3	Aktér	Vyplní viditelnost a název metody, pak, pokud je metoda bezparametrická, zmáčkne tlačítko „-“, čímž odstraní řádek na zadání vstupního parametru a pokračuje krokem 5; pokud má metoda více než jeden parametr, zmáčknutím tlačítka „+“ přidá další řádky podle počtu parametrů.
4	Aktér	Pro každý parametr vyplní jeho typ, název a výraz pro počáteční hodnotu.
5	Aktér	Zmáčkne tlačítko „*OK*“.
6	Systém	Provede formální kontrolu vstupních dat: zkontroluje, že všechna políčka jsou vyplněna a názvy metody a parametrů jsou platné Java identifikátory. Pokud jsou chyby, viz

		Vygenerovat instanční metodu – alternativní scénář (chybná vstupní data) – používá se stejný alternativní scénář jako v UC001.
7	Systém	Zkontroluje, že v cílové třídě není metoda se stejnou signaturou. Pokud je, viz **Vygenerovat instanční metodu – alternativní scénář (existující metoda) – používá se stejný alternativní scénář jako v UC001.**
8	Systém	Vloží do kódu deklaraci nové metody; pokud už existuje metoda se stejnou signaturou, nový kód přepíše existující, jinak se nový kód vloží na konec třídy.

2.3 Podpora lambda výrazů

V březnu 2014 byla vydána osmá verze jazyka a platformy Java, která přinesla spoustu novinek. Z hlediska jazyka je největší novinkou podpora lambda výrazů [19]. Podle [20] lambda výrazy velmi dobře zapadají do metodiky *Architecture First*, proto přidání podpory pro lambda výrazy do prostředí BlueJ bylo hned po rozšíření generátoru kódu druhou největší prioritou této práce.

Problematika lambda výrazů je velmi zajímavá a rozsáhlá – mohla by být tématem samostatné závěrečné práce. Daná práce se proto ve svém rámci této problematiky dotkne jen velmi stručně, pokusí se pouze objasnit podstatu lambda výrazů, a to v minimálním rozsahu, který je nutný pro pochopení změn zaváděných do prostředí BlueJ.

Lambda výraz umožňuje definovat kus kódu, který poté můžeme použít v jiné části programu jako objekt. Například předat ho jako vstupní parametr metody, uvnitř které se pak tento kus kódu spustí. V Javě lze lambda výrazy zapsat dvěma způsoby. Hlavní způsob zápisu vypadá takto:

```
parametr -> výraz
(parametry) -> výraz
parametr -> { příkazy }
(parametry) -> { příkazy }
```
[20].
Vlevo od šipky se zapisuje seznam parametrů, který může být i prázdný, vpravo se píšou příkazy, které je třeba provést.

Pokud příkaz představuje pouze volání jedné metody, lze použít jiný způsob zápisu, takzvaný odkaz na metodu (method reference):

```
objekt::instančníMetoda
Třída::statickáMetoda
Třída:: instančníMetoda
```
[21].

Lambda výrazy se chovají jako instance funkčních rozhraní. Funkční rozhraní (*Functional Interface*) je rozhraní, které deklaruje právě jednu abstraktní metodu. Pokud je třeba uložit lambdu, realizuje se tento krok jejím uložením do proměnné příslušného funkčního rozhraní.

Nyní můžeme navrhnout, jak by mohla vypadat podpora lambda výrazů v prostředí BlueJ. Lze ji realizovat pomocí dialogového okna, v němž uživatel zadá do jednoho políčka samotný lambda výraz, do druhého políčka zadá funkční rozhraní a zmáčkne tlačítko „*OK*“. Lambda výraz se uloží do zásobníku odkazů jako instance zvoleného funkčního rozhraní. Mělo by to také být v souladu s konceptem práce v interaktivním režimu. Pokud chceme definovat lambda výraz jako odkaz na metodu, prostředí BlueJ by mělo uživateli umožnit tuto metodu vybrat kliknutím myši v místní nabídce příslušné třídy nebo objektu, a pak automaticky vygenerovat odkaz na vybranou metodu.

2.4 Uložení do zásobníku odkazů hodnot primitivních datových typů

Nezřídka je třeba poslat výsledek jedné metody jako vstupní parametr do metody druhé. Například potřebujeme objektu `obdélník1` nastavit stejnou šířku, jakou má `obdélník2`. To lze zakódovat dvěma způsoby:

1) bez použití pomocné proměnné:

```
obdélník1.setŠířka(obdélník2.getŠířka());
```

2) s použitím pomocné proměnné pro uložení mezivýsledku:

```
int šířka = obdélník2.getŠířka();
obdélník1.setŠířka(šířka);
```

Pokud se zamyslíme nad tím, jak by takovou úlohu měli řešit studenti začátečníci na první hodině programování podle metodiky *Architecture First*, dojdeme k závěru, že ji vyřešit nedokážou, i přes to, že je velmi jednoduchá. Varianta bez použití pomocné proměnné vyžaduje ruční zápis kusu kódu, což by studenti na první hodině dělat neměli. Varianta s použitím pomocné proměnné zase narazí na jiný problém. Jde o to, že jediným „úložným prostorem“, kam studenti mohou ukládat výsledky v interaktivním režimu, je zásobník odkazů, do kterého ale nelze vložit hodnotu primitivního typu.

Jelikož Java pro každý primitivní typ definuje příslušný obalový typ (*wrapper*), výše uvedený problém můžeme obejít tak, že výsledky primitivních typů budeme ukládat do zásobníku odkazů pomocí odpovídajících obalových typů. Pak ale vznikne další problém. Obalový typ je plnohodnotný objekt, kterému lze posílat zprávy. Studentům tak zcela unikne pojem „primitivní typ“ a budou se domnívat, že všechno je objekt a všemu je možné posílat zprávy. Toto tvrzení sice platí pro čistě objektové jazyky, jakým je například Scala [22], ale Java takovým jazykem není.

Autor proto navrhuje kompromisní řešení. Pokud metoda vrátí hodnotu primitivního typu, umožníme uložit ji do zásobníku odkazů pomocí příslušného obalového typu, ale současně zajistíme, aby se tento objekt v zásobníku vydával za primitivní. Znamená to, že jako popisek datového typu se místo skutečného objektového typu zobrazí odpovídající primitivní typ a v místní nabídce objektů nebudeme zobrazovat žádné metody. Takové objekty v zásobníku odkazů můžeme pojmenovat jako „falešná primitiva“ (*fake primitives*).

2.5 Změna hodnoty objektu v zásobníku odkazů

Generátor kódu generuje tělo metody jako sekvenci příkazů. V této sekvenci je dost často třeba výsledek jedné metody posílat na vstup metody druhé. V předchozí kapitole jsme pro tento účel použili proměnnou pro uložení mezivýsledku. Pokud později znovu vznikne potřeba uložit mezivýsledek stejného datového typu a ten předchozí již nepotřebujeme, je zbytečné zavádět novou proměnnou. Úplně postačí uložit nový výsledek do existující proměnné (změnit její hodnotu). Aktuální implementace zásobníku odkazů však toto nepovoluje. Pokud už v zásobníku odkazů je objekt se jménem *x* a my se pokusíme uložit pod tímto jménem objekt nový, BlueJ ho automaticky uloží pod jménem *x1*. Pak při dalších pokusech to budou *x2, x3* atd.

Poslední navrhovanou úpravou je právě změna tohoto výše uvedeného chování. Autor se domnívá, že správně by BlueJ v takové situaci měl uživatele upozornit, že objekt se stejným jménem již existuje, a nabídnout na výběr dvě možnosti: buď změnit hodnotu existujícího objektu, nebo uložit nový objekt pod jiným jménem.

3 Uživatelská příručka

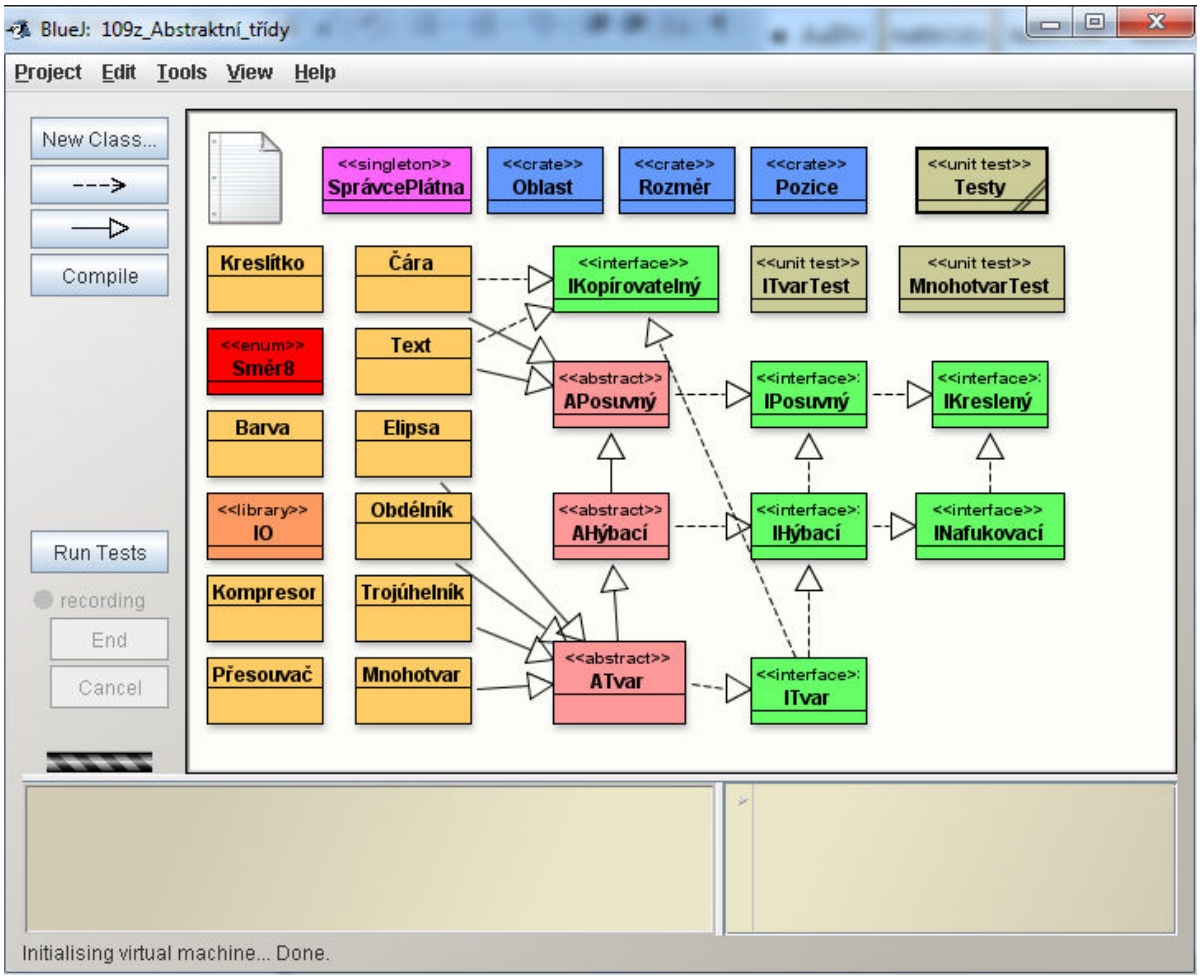
3.1 Nové stereotypy pro vybrané návrhové vzory

Upravená verze vývojového prostředí BlueJ je schopna automaticky rozpoznat, že daná třída realizuje jeden z vybraných návrhových vzorů, a označit ji příslušným stereotypem.

V případě, že uživatel definoval pro daný návrhový vzor speciální barvu, kromě stereotypu bude návrhový vzor v diagramu tříd ještě zvýrazněn touto definovanou barvou; viz příklad na obr. 4. Vlastní barvy lze definovat v konfiguračním souboru *bluej.defs* pomocí barevného modelu RGB. Tabulka 5 ukazuje seznam podporovaných návrhových vzorů. Pro každý podporovaný návrhový vzor je uveden jeho stereotyp a klíč pro definici barvy.

Tab. 5: Podporované návrhové vzory

Vzor	Stereotyp	Klíč pro definicí barvy v bluej.defs
Knihovní třída	«library»	`colour.class.bg.library`
Jedináček	«singleton»	`colour.class.bg.singleton`
Přepravka	«crate»	`colour.class.bg.crate`



Obr. 4 – Nové stereotypy pro návrhové vzory v diagramu tříd

3.2 Generátor zdrojového kódu

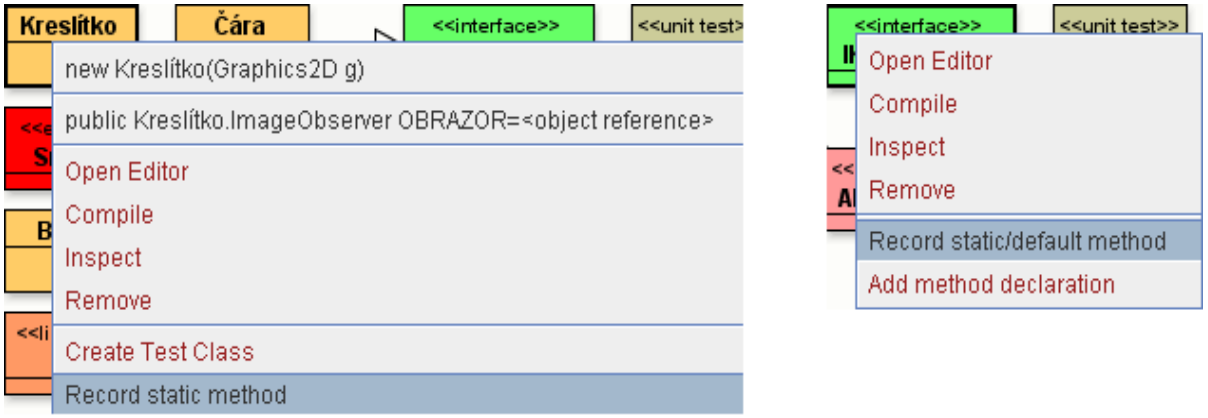
3.2.1 Spouštění generátoru

Generátor se spouští přes položku *Record static method* (nahrát statickou metodu) v místní nabídce třídy v diagramu tříd, nebo přes položku *Record method* (nahrát instanční metodu) v místní nabídce objektu zásobníku odkazů. Položka *Record method* je u objektu dostupná pouze v případě, že se jedná o instanci třídy, jejíž zdrojový kód je k dispozici a lze ho editovat. Obrázek 5 porovnává místní nabídku instance třídy `Obdélník` z aktuálního projektu[5] a místní nabídku instance třídy `String` ze standardní knihovny. Je vidět, že v prvním případě můžeme přidat (nahrát) do třídy novou metodu, ale v druhém případě taková možnost není.

Obr. 5 – Místní nabídka instance třídy z vlastního projektu vs. místní nabídka instance třídy z knihovny

Co se týče možnosti nahrát statickou metodu, ta je dostupná u všech tříd projektu, ale v případě rozhraní je tato položka pojmenována trochu jinak: *Record static/default method* (viz obr. 6). Je to proto, že v Javě 8 je realizována možnost přidávat do rozhraní výchozí (defaultní) implementaci metod [21]. Defaultní implementace je v podstatě implementace instanční metody, proto by položka „nahrát defaultní metodu“ logicky měla patřit do místní nabídky instance v zásobníku odkazů. Ale jelikož u rozhraní nemůžeme přímo vytvořit instanci, možnost nahrávání defaultní metody a nahrávání metody statické byly spojeny do jedné položky místní nabídky rozhraní.

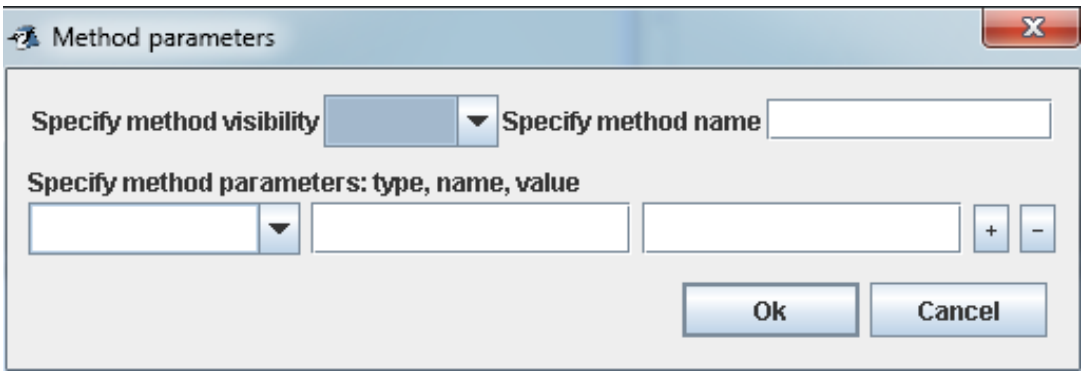
[5] Jedná se o projekt 109z_Abstraktní_třídy z učebnice [6]



Obr. 6 – Místní nabídka standardní třídy vs. místní nabídka rozhraní

3.2.2 Vyplnění vstupních údajů pro budoucí metodu

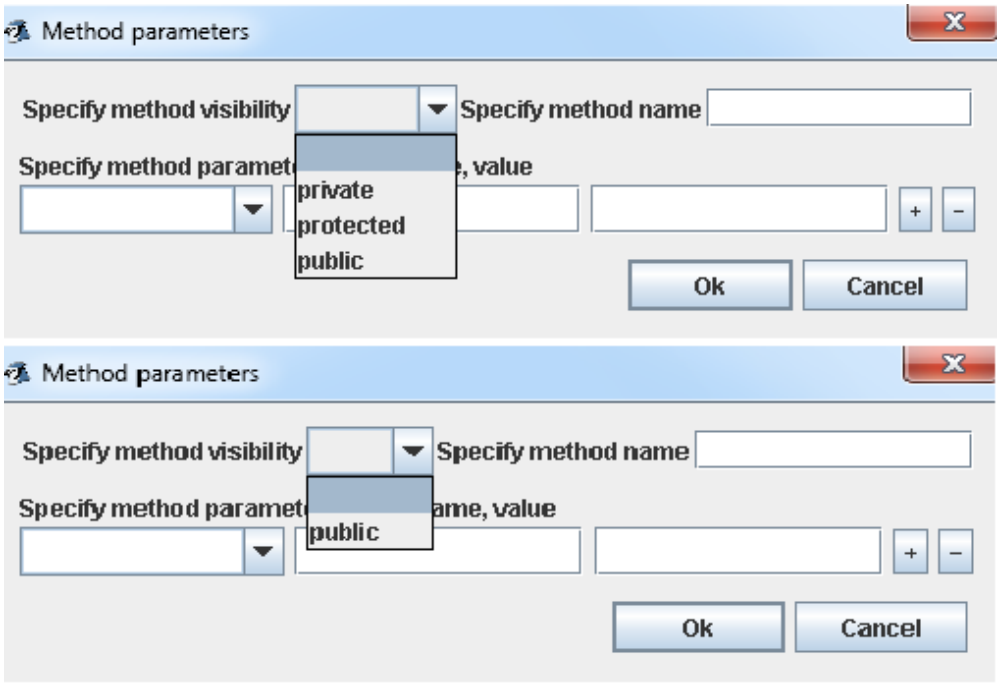
Bez ohledu na to, jakým způsobem byl spouštěn generátor kódu (přes místní nabídku třídy nebo instance), hned po spuštění se zobrazí dialogové okno pro zadání vstupních údajů pro budoucí metodu, jehož výchozí vzhled je vidět na obrázku 7. V tomto okně uživatel může zvolit viditelnost budoucí metody, zadat jméno metody a její parametry.



Obr. 7 – Výchozí vzhled modálního okna pro zadávání vstupních údajů budoucí metody

Nabídka modifikátoru viditelnosti je kontextově závislá na typu třídy, do které novou metodu přidáváme. Tak pro standardní třídu (viz obr. 8 – horní část) jsou k dispozici všechny možné modifikátory přístupu, ale pro rozhraní (viz obr. 8 – dolní část) nejsou k dispozici modifikátory `private` a `protected`. Tím pádem je zajištěno, že student začátečník nevloží do zdrojového kódu modifikátor, který v daném kontextu není povolen a který by mohl způsobit chybu typu „*modifier xxx not allowed here*“ a pád kompilace.

Do políčka „*specify method name*“ uživatel musí zadat platný identifikátor Javy – to se kontroluje při validaci vstupních dat (viz další kapitola). Ale zda je metoda pojmenována správně, tj. podle příslušné jmenné konvence [23], by měl student (nebo jeho učitel) ohlídat sám.



Obr. 8 – Modifikátory viditelnosti pro metodu třídy (nahoře) a pro metodu rozhraní (dole)

Podle výchozích nastavení modálního okna se předpokládá, že metoda bude mít jeden vstupní parametr. Pomocí tlačítka „+“ může uživatel přidat políčka pro zadávání dalších parametrů, pomocí tlačítka „-“ odebrat stávající a tím pádem definovat metodu jako bezparametrickou.

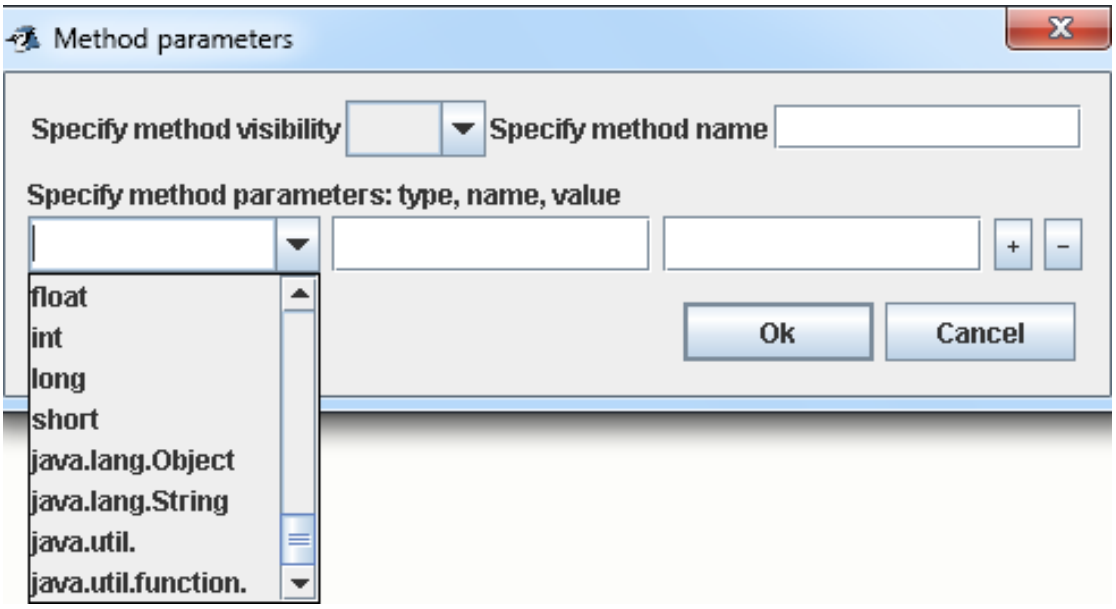
Pro každý parametr by měl uživatel zadat jeho typ, název a hodnotu, která bude použita při přidání daného parametru do zásobníku odkazů (viz kapitola 3.2.5).

Typ parametru uživatel může vyplnit ručně, nebo vybrat z nabídky v rozbalovacím menu (viz obr. 9). Tato nabídka obsahuje všechny třídy aktuálního projektu kromě testovacích, všechny primitivní typy, třídy `java.lang.String` a `java.lang.Object`, začátky plných jmen tříd z balíčku `java.lang.util` a `java.lang.function`. Například pokud uživatel potřebuje parametr typu `java.util.Date`, může vybrat z nabídky „`java.util`.“ a pak ručně dopsat „`Date`“.

Do políčka pro jméno parametru je třeba zadat platný Java identifikátor. Jmenná konvence není kontrolována (stejně jako u jména metody). V případě, že metoda má více parametrů, jméno každého parametru musí být unikátní v rámci dané metody.

Do políčka pro hodnotu parametru můžeme jako hodnotu zadaného typu uložit přímo jméno objektu ze zásobníku odkazů, nebo výraz, jehož vyhodnocení by mělo vrátit hodnotu příslušného datového typu. Například pro parametr typu `java.lang.String` můžeme zadat:

- `"lorem ipsum"`
- `string1`
- `IO.odháčkuj("žlutý kůň")`



Obr. 9 – Nabídka možných datových typů pro parametry metody

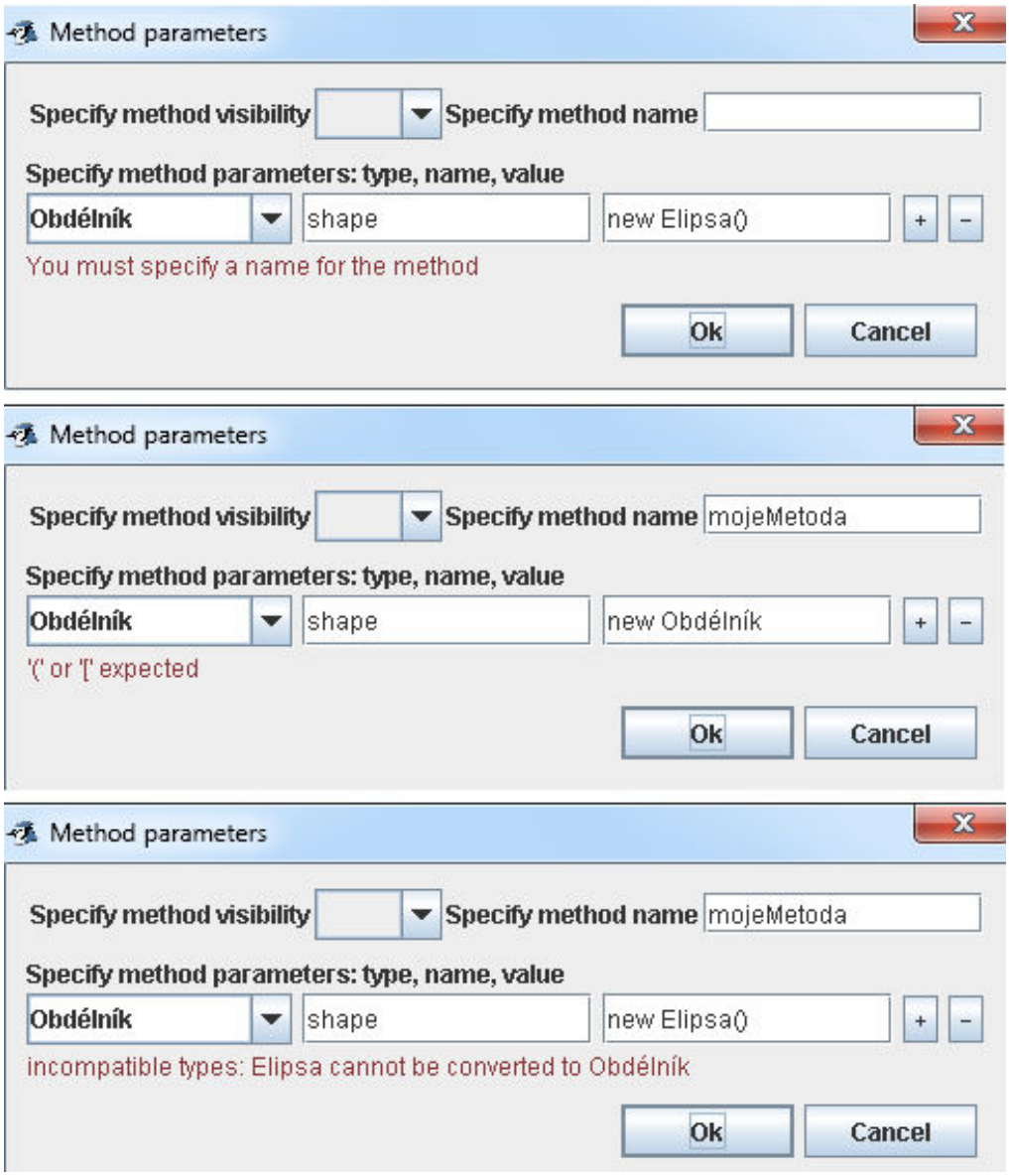
V prvním případě je to přímé vložení hodnoty, v druhém odkaz do zásobníku odkazů (předpokládá se, že tam je instance třídy `java.lang.String` se jménem `string1`) a ve třetím je to výraz k vyhodnocení, který by měl vrátit výsledek typu `java.lang.String`. Samozřejmě metodika *Architecture First* předpokládá, že studenti začátečníci budou zadávat hodnotu přímo nebo budou používat objekt ze zásobníku odkazů, nikoliv psát do políčka kód k vyhodnocení.

3.2.3 Validace vstupních dat

Po zmáčknutí tlačítka „*OK*“ v dialogovém okně na zadávání vstupních dat metody spouští BlueJ validaci zadaných uživatelských dat. Tato validace kontroluje následující body:

- žádné políčko není prázdné,
- název metody a názvy všech parametrů jsou platné Java identifikátory,
- vyhodnocení všech výrazů pro hodnoty parametrů proběhlo bez chyb a výsledky odpovídají očekávanému datovému typu.

Pokud data neprošla úspěšně kontrolou, v dolní části dialogového okna se zobrazí hláška s textem chyby. Na obrázku 10 jsou zobrazeny různé výsledky neúspěšné validace: nevyplněno jméno metody, nesprávný výraz (chybí závorky), výsledek vyhodnocení není kompatibilní s typem parametru.

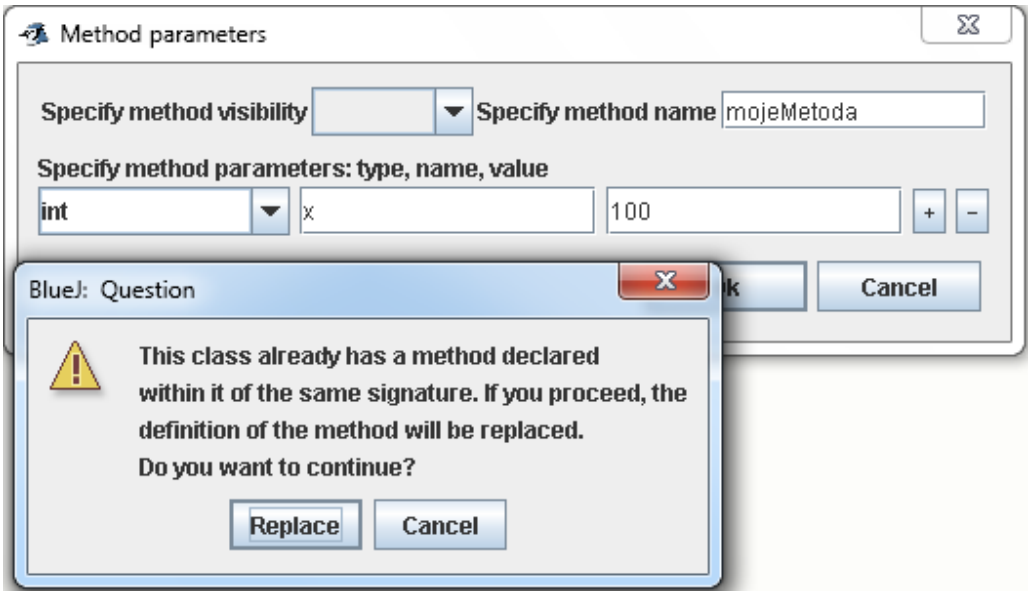


Obr. 10 – Různé typy validačních chyb

3.2.4 Kontrola signatury metody

Kromě uvedené validace BlueJ také ověřuje, zda cílová třída, do níž se chystáme přidat novou metodu, již neobsahuje metodu se stejnou signaturou, tzn. se stejným jménem a seznamem parametrů stejného typu ve stejném pořadí. Pokud taková metoda ve třídě již existuje, BlueJ zobrazí upozornění, že v případě pokračování nově vygenerovaný kód přepíše již existující (viz obr. 11).

Pokud se uživatel rozhodne pokračovat (volba „*Replace*“), pak začne nahrávání nové sekvence příkazů, která následně přepíše kód existující metody. Pokud se uživatel rozhodne jinak (volba „*Cancel*“), vrátí se do dialogového okna na zadávání vstupních dat, v němž má možnost signaturu metody upravit (změnit jméno nebo parametry metody) a zkusit znovu projít kontrolou. Případně v tomto okně může také zmáčknout „*Cancel*“, pokud se rozhodne vůbec novou metodu nenahrávat.



Obr. 11 – Upozornění, že metoda se stejnou signaturou již ve třídě existuje

3.2.5 Příprava zásobníku odkazů pro nahrávání nové metody

Pokud validace vstupních dat proběhla úspěšně, BlueJ připraví zásobník odkazů pro nahrávání nové metody. To znamená, že zajistí, aby tam bylo všechno, s čím uživatel může pracovat v kontextu metody, a nebylo nic, s čím pracovat nemůže.

Technicky je to realizováno tak, že BlueJ nejprve zcela vyklidí zásobník odkazů, a pak do něj vloží všechno to, k čemu má uživatel v kontextu dané metody přístup. Pro instanční metodu to znamená:

- odkaz na aktuální instanci, uloženou pod jménem `_this`[6],
- všechny atributy (instanční a statické) bez ohledu na viditelnost[7],
- všechny parametry metody.

Pro statickou metodu bude seznam o něco kratší, a sice:

- všechny statické atributy bez ohledu na viditelnost,
- všechny parametry metody.

Obrázek 12 ukazuje stav zásobníku odkazů při nahrávání instanční metody (nahoře) a statické metody (dole).

[6] Při uložení do zásobníku odkazů není možné jako jméno objektu použít klíčové slovo jazyka Java, proto místo „`this`“ se použije „`_this`“. Při vygenerování kódu pak proběhne opačná transformace a veškeré výskyty „`_this`” budou nahrazeny „`this`“.

[7] Dostupnost soukromých atributů se nastaví přes *`java.lang.reflect.AccessibleObject#setAccessible()`*.

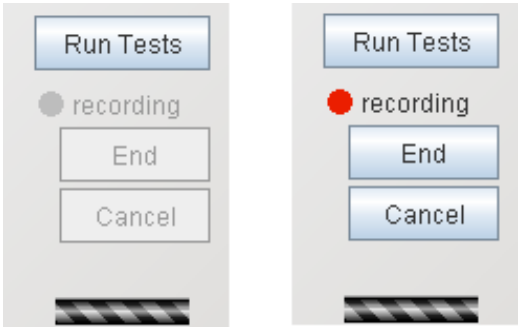
Obr. 12 – Stav zásobníku odkazů při nahrávání instanční metody (nahoře) a statické metody (dole)

V obou případech se jedná o přidávání metody do stejné třídy `Text` a se stejným seznamem parametrů (`String param1, int param2`). Jak je vidět, jediný rozdíl je v tom, že ve statickém kontextu nejsou dostupné instanční atributy `barva`, `název`, `font`, a odkaz na aktuální instanci `_this`.

3.2.6 Nahrávání příkazů

Po dokončení přípravy zásobníku odkazů (viz předchozí kapitola) se prostředí BlueJ přepne do režimu nahrávání. To lze poznat podle stavu ovládacích prvků v panelu „*testovací nástroje*“[8] (viz obr. 13). V tomto režimu prostředí BlueJ zaznamenává veškeré interakce uživatele, jako je volání konstruktorů a metod, vytvoření lambda výrazů (viz kapitola 3.3) nebo změna hodnot objektu v zásobníku odkazů (viz kapitola 3.5).

Po ukončení režimu nahrávání (viz další kapitola) se z každé zaznamenané interakce stává příkaz zdrojového kódu.

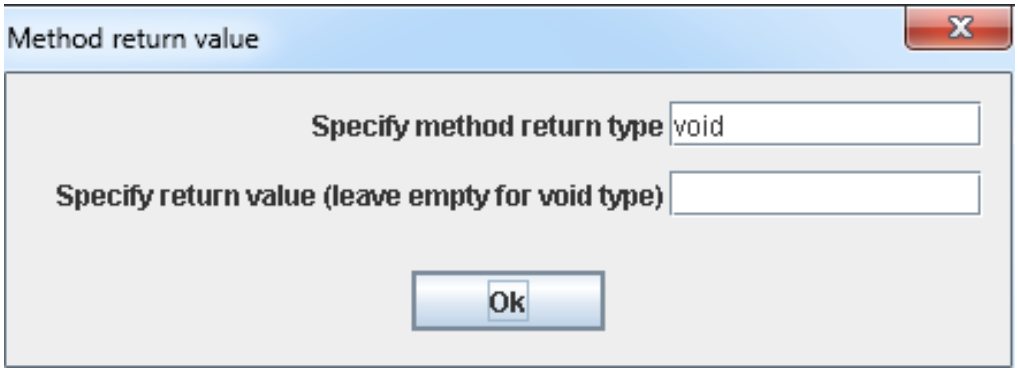


Obr. 13 – Stav panelu „testovací nástroje“ v běžném režimu (vlevo) a v režimu nahrávání (vpravo)

3.2.7 Ukončení nahrávání metody

Po dokončení poslední interakce, která patří do kódu nové metody, zmáčkne uživatel tlačítko „*End*“ v panelu „*testovací nástroje*“. BlueJ zobrazí okno pro zadání návratové hodnoty metody, které je podle výchozího stavu nastaveno na typ „`void`“ (žádná návratová hodnota), viz obr. 14.

[8] Generátor kódu metod je postaven nad generátorem kódu jednotkových testů a sdílí s ním některé společné prvky jak na front-endu, tak i na back-endu aplikace.

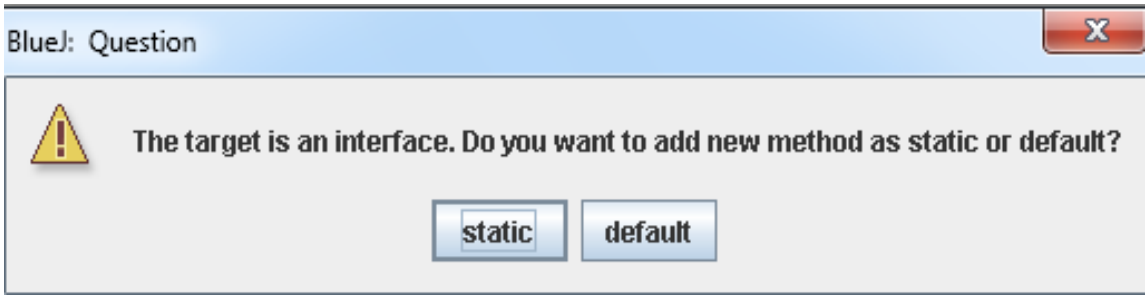


Obr. 14 – Výchozí stav dialogového okna pro zadání návratové hodnoty metody

Zadat návratovou hodnotu může uživatel samozřejmě i ručně, ale preferovanější je klik myši na příslušný objekt v zásobníku odkazů. V tomto případě BlueJ automaticky doplní do příslušných políček název vybraného odkazu a jeho datový typ. Pokud uživatel nějakou návratovou hodnotu vyplní, generátor kódu přidá na konec metody příkaz „`return`“ a do hlavičky metody doplní příslušný datový typ.

Po zmáčknutí tlačítka „*OK*“ se vygenerovaná metoda vloží do zdrojového kódu třídy. Pokud ve třídě již existovala metoda se stejnou signaturou, nová metoda bude vložena na její místo a přepíše existující kód. V opačném případě se nová metoda uloží na konec třídy.

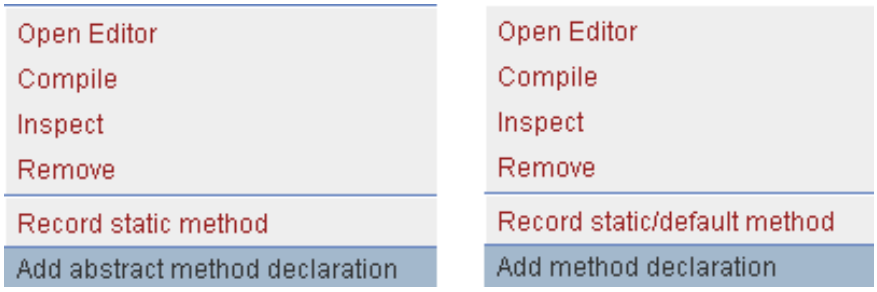
U rozhraní se generátor před vložením vygenerované metody do kódu ještě zeptá, zda chce uživatel metodu přidat jako statickou, nebo jako defaultní (viz obr. 15).



Obr. 15 – Dotaz před vložením vygenerované metody do rozhraní

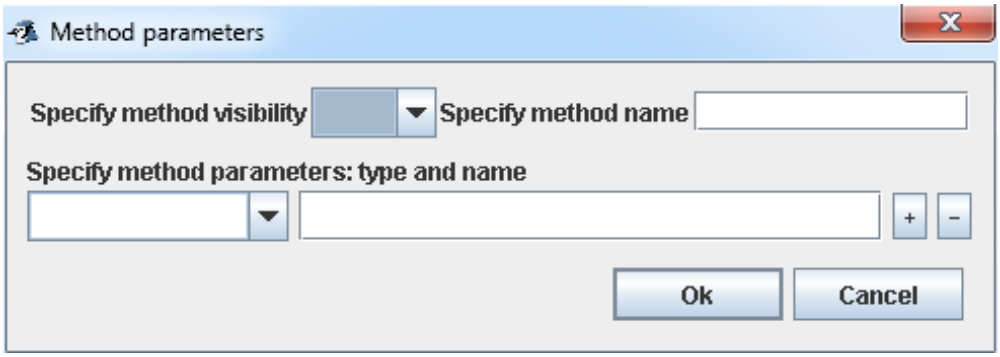
3.2.8 Přidání deklarace metody

Kromě plnohodnotného nahrávání metod nový generátor kódu umožňuje také generování deklarací metod (hlavičky metod bez implementace). Tato funkce je dostupná pro abstraktní třídy a rozhraní přes jejich místní nabídky (viz obr. 16).



Obr. 16 – Místní nabídka abstraktní třídy (vlevo) a rozhraní (vpravo)

Vygenerování deklarace opět začíná dialogovým oknem pro zadávání vstupních dat, které ale u parametrů metody nevyžaduje zadání jejích hodnot (viz obr. 17).



Obr. 17 – Dialogové okno na zadávání vstupních dat pro novou deklaraci metody

Po zmáčknutí tlačítka „*OK*“ proběhne validace dat (téměř stejně jako při nahrávání metody, jen se nekontrolují výrazy pro hodnoty parametrů). Vzápětí po úspěšné validaci se objeví dialogové okno na zadání typu návratové hodnoty, které vypadá obdobně jako dialogové okno zobrazené na obrázku 14, rozdíl je ale v tom, že políčko pro zadávání návratové hodnoty je nedostupné. Uživatel může zadávat pouze typ návratové hodnoty, nikoliv hodnotu samotnou.

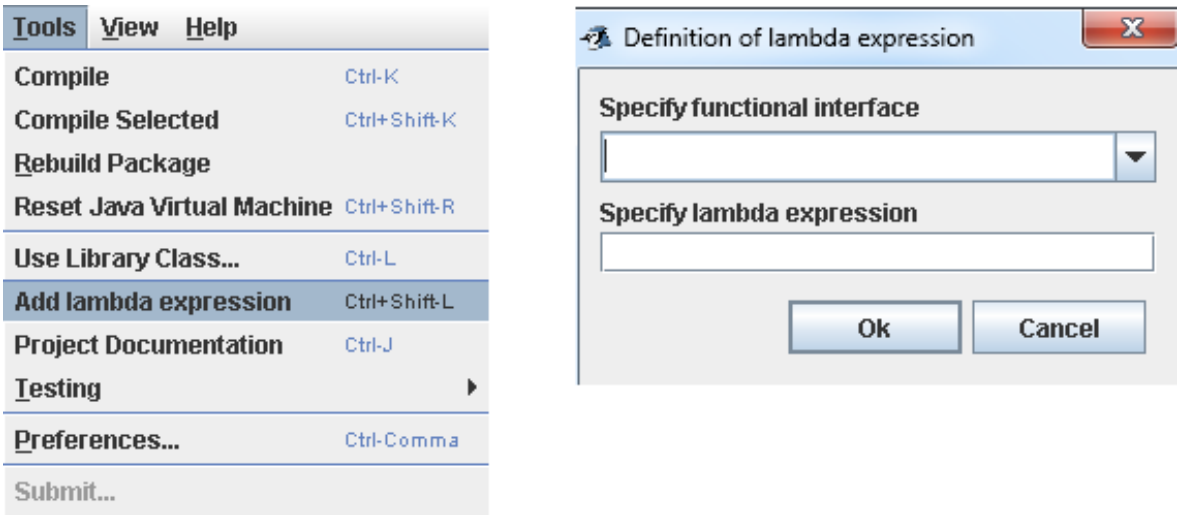
Po vyplnění typu návratové hodnoty uživatel zmáčkne „*OK*“ a deklarace se následně vloží do kódu. Pokud cílová třída je abstraktní, nová deklarace se vloží s modifikátorem `abstract`.

3.3 Podpora lambda výrazů

Přidat nový lambda výraz lze přes položku menu *Tools->Add lambda expression* (viz obr. 18 vlevo), která otevře dialogové okno pro definici lambda výrazu (viz obr. 18 vpravo).

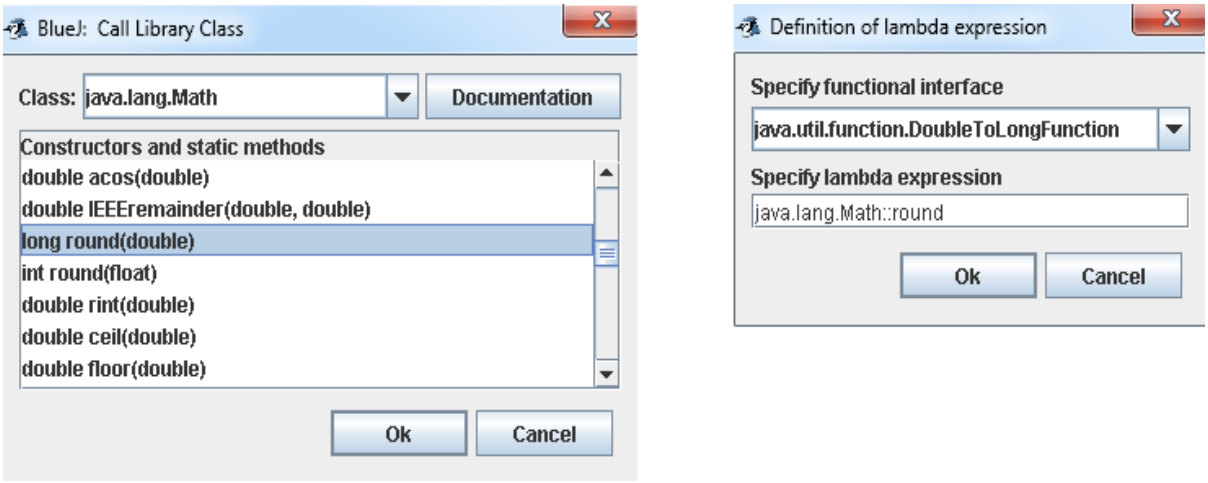
Okno obsahuje dvě políčka: do horního se zadává funkční rozhraní, pod kterým se lambda výraz uloží do zásobníku odkazů, do dolního se zadává lambda výraz samotný.

Funkční rozhraní může uživatel zadat ručně jako text, nebo vybrat z nabídky, která obsahuje rozhraní `java.lang.Runnable`, `java.lang.Comparable`, `java.util.Comparator` a všechna rozhraní z balíčku `java.util.function`.



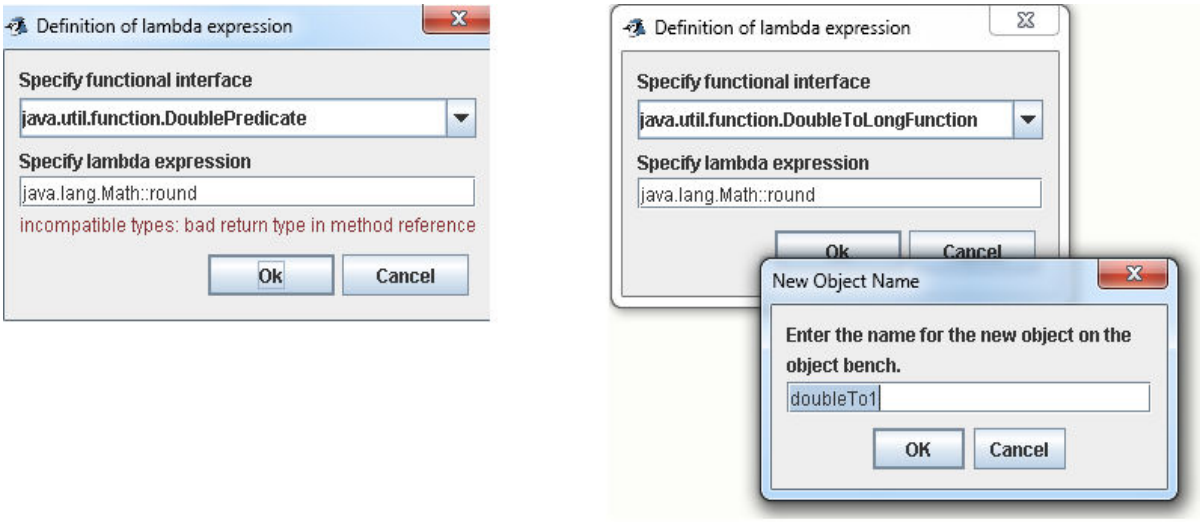
Obr. 18 – Definice nového lambda výrazu

Samotný lambda výraz může uživatel také zadat ručně jako text, nebo použít funkci na vygenerování odkazu na metodu. Funguje to tak, že pokud je otevřené okno na definici lambda výrazu, výběr jakékoliv metody v místních nabídkách tříd a objektů místo klasického spuštění metody pouze vygeneruje odkaz na vybranou metodu do políčka *„Specify lambda expression“*. Totéž platí i pro výběr metody z knihovní třídy přes okno *„Use Library Class“* (viz obr. 19).



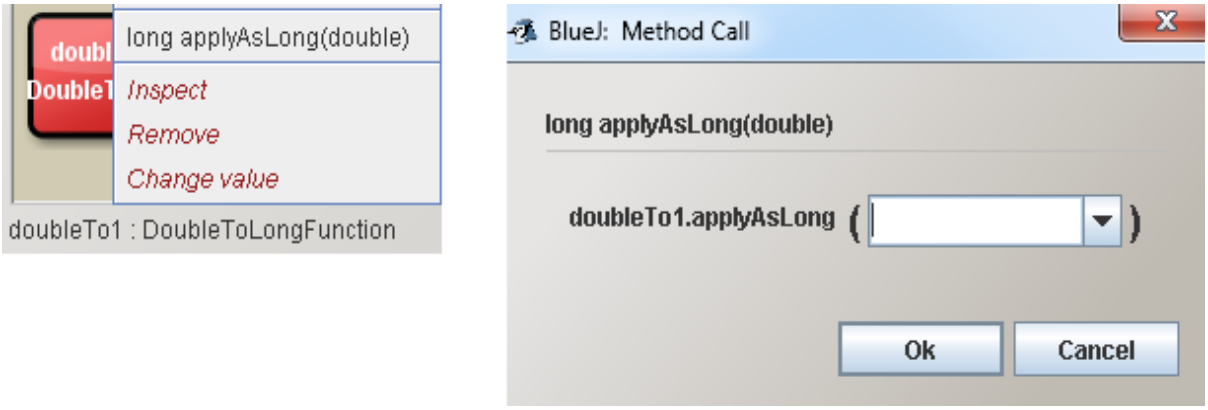
Obr. 19 – Výběr metody (vlevo) a vygenerovaný odkaz na metodu (vpravo)

Po zmáčknutí tlačítka „*OK*“ se spustí vyhodnocování lambda výrazu. Pokud lambda výraz obsahuje chybu nebo není kompatibilní s vybraným funkčním rozhraním, chyba se zobrazí ve stejném dialogovém okně (viz obr. 20 vlevo). Při úspěšném výsledku vyhodnocení se BlueJ zeptá na jméno (viz obr. 20 vpravo), pod nímž se pak lambda výraz uloží do zásobníku odkazů.



Obr. 20 – Výsledek vyhodnocení lambda výrazu: neúspěšný (vlevo) a úspěšný (vpravo)

Do zásobníku odkazů se lambda výraz uloží jako instance funkčního rozhraní (viz obr. 21 vlevo), se kterou lze pracovat stejně jako s instancí jakékoliv „normální“ třídy: volat metody z místní nabídky[9] (viz obr. 21 vpravo), posílat jako vstupní parametr do metody nebo vracet při nahrávání metod jako návratovou hodnotu.

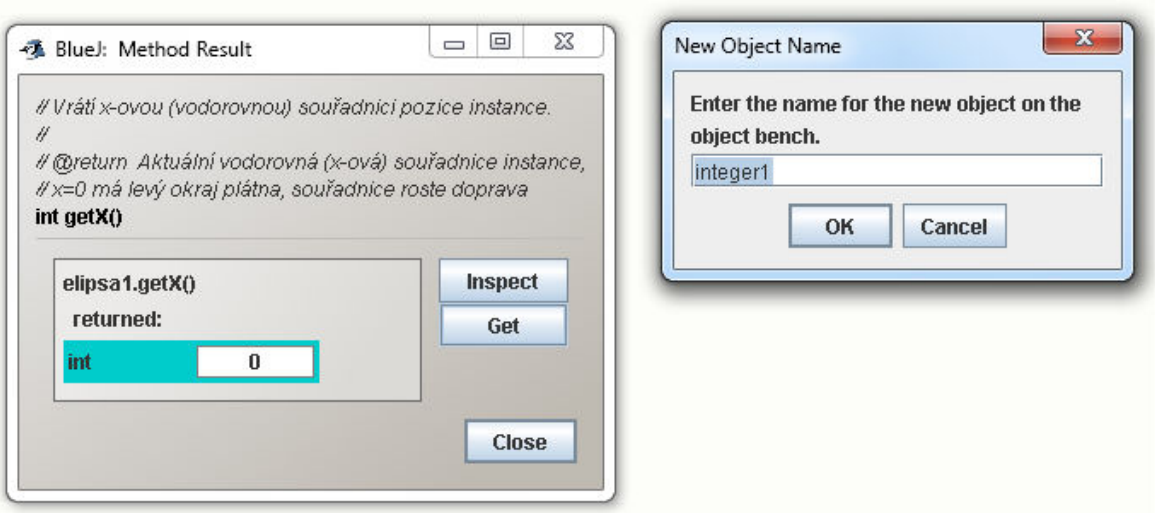


Obr. 21 – Lambda výraz v zásobníku odkazů (vlevo) a spuštění metody lambda výrazu (vpravo)

3.4 Uložení primitivních hodnot do zásobníku odkazů

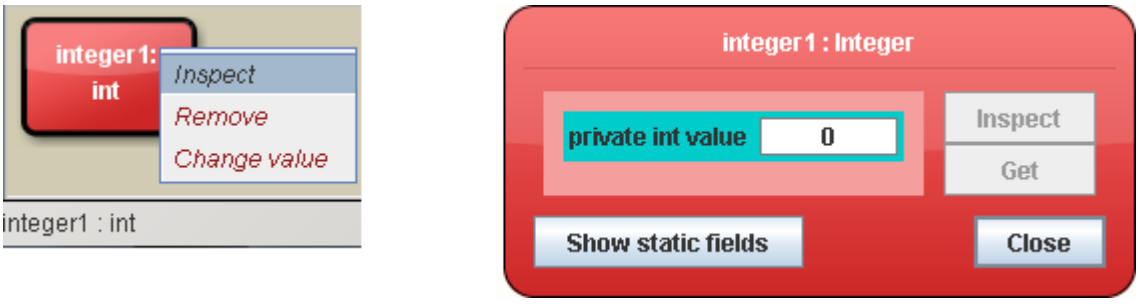
Upravená verze BlueJ nově umožňuje ukládat do zásobníku odkazů hodnoty primitivních datových typů. Viz obr. 22 – metoda sice vrátila výsledek primitivního datového typu, ale tlačítko „*Get*“, které umožňuje uložení do zásobníku odkazů, je dostupné. Podle jména, které BlueJ nabízí, lze odhalit, že se do zásobníku odkazů ukládá ve skutečnosti instance příslušného obalového typu, tj. `Integer` místo `int`.

[9] V místní nabídce lambda výrazu je vždy minimálně jedna metoda, jejíž implementaci vyžaduje dané funkční rozhraní. Ale pokud rozhraní má kromě toho ještě nějaké defaultní metody, ty se zobrazí v místní nabídce také.



Obr. 22 – Výsledek volání metody s primitivním typem návratové hodnoty a jeho uložení do zásobníku

V zásobníku odkazů se ale u příslušného objektu zobrazuje primitivní datový typ a místní nabídka neobsahuje žádnou metodu (viz obr. 23 vlevo). Skutečný datový typ prozradí inspektor objektu (viz obr. 23 vpravo). Při generování kódu se použije „falešný“ primitivní typ.

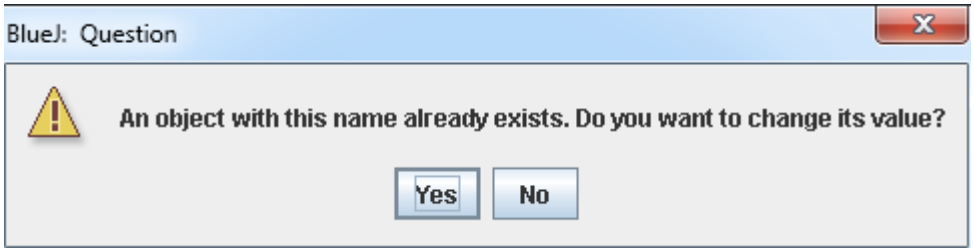


Obr. 23 – „Falešná“ primitivní hodnota v zásobníku odkazu (vlevo) a její skutečný typ (vpravo)

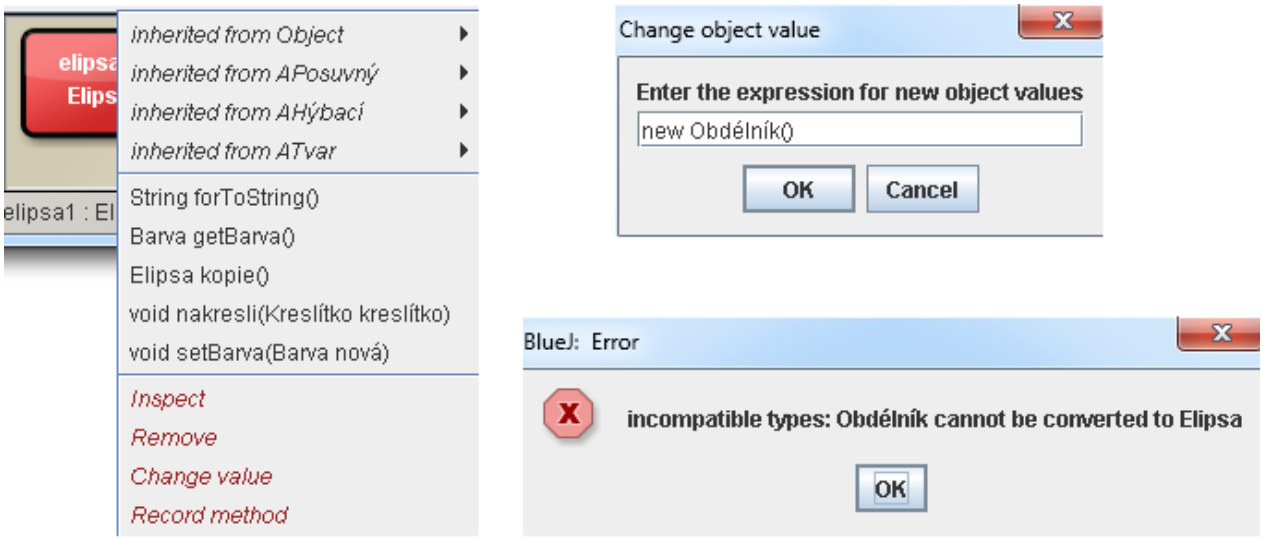
3.5 Změna hodnoty objektu v zásobníku odkazů

Změnit hodnotu objektu v zásobníku odkazů lze dvěma způsoby:

- Při uložení nového objektu zadat jméno již existujícího a potvrdit, že skutečně chceme změnit existující hodnotu (viz obr. 24).
- V místní nabídce vybrat položku *„Change value“* (viz obr. 25 vlevo) a v dialogovém okně zadat výraz pro novou hodnotu (viz obr. 25 vpravo nahoře). Pokud vyhodnocování výrazů proběhne úspěšně, hodnota v zásobníku se změní, jinak se zobrazí chybová hláška.



Obr. 24 – Uložení nové hodnoty do objektu



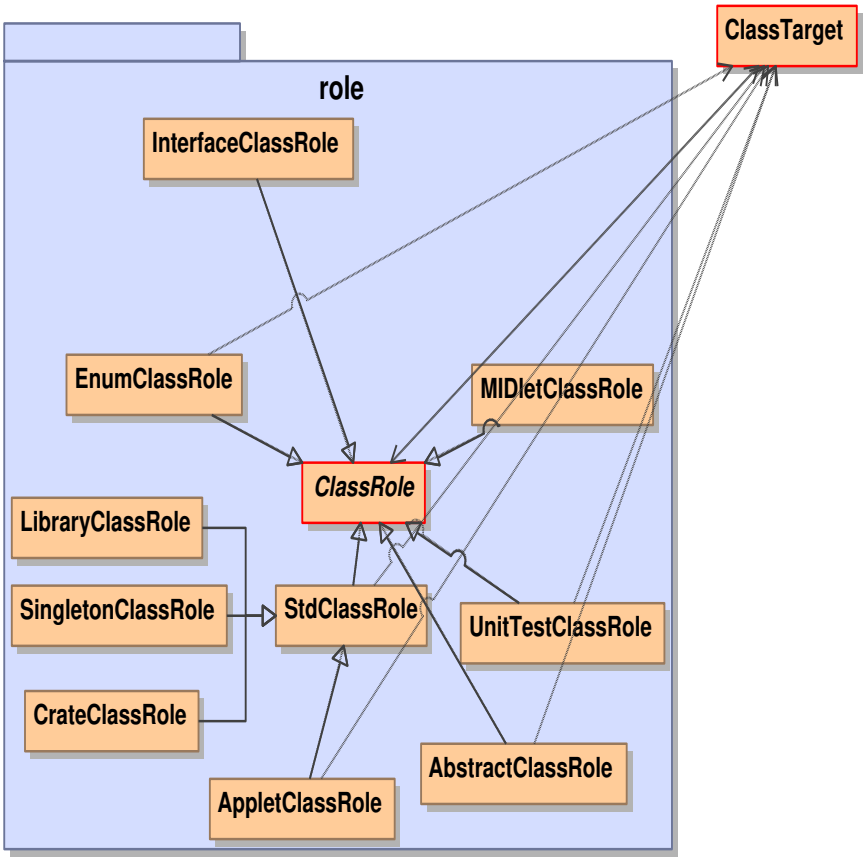
Obr. 25 – Změna hodnoty objektu v zásobníku odkazů

Bez ohledu na způsob změny hodnoty se při nahrávání metody do zdrojového kódu tato operace zapíše v podobě `jmenoPromenné = nováHodnota;`.

4 Implementační příručka

4.1 Přidání nových stereotypů

Každá třída projektu v IDE BlueJ se vykresluje do UML diagramů v podobě obdélníku s názvem třídy a případným stereotypem. Na implementační rovině za tímto obdélníkem stojí instance třídy `ClassTarget`. Ale to, jak třída v diagramu vypadá a co všechno umí, záleží na tom, jakou má třída roli. Právě role určuje, jaký má třída stereotyp (případně nemá žádný), jaká barva se používá pro vyplnění obdélníku v diagramu a jaké položky má třída v místní nabídce. Standardní třída má roli `StdClassRole`, abstraktní třída má roli `AbstractClassRole`, rozhraní má roli `InterfaceClassRole` atd. Všechny role jsou potomky abstraktní role `ClassRole`, která zapouzdřuje funkcionalitu společnou pro všechny role.



Obr. 26 – Diagram tříd balíčku bluej.pkgmgr.target.role. Zdroj: upraveno podle [7]

Takže přidat nové stereotypy pro tři návrhové vzory na implementační rovině znamenalo přidat tři nové role. Jelikož jedináček, knihovní třída a přepravka jsou v podstatě speciální případy standardní třídy, tyto tři nové role přidáme jako potomky třídy `StdClassRole`, viz diagram tříd na obrázku 26.

Přiřazování konkrétní role se odehrává v metodě `ClassTarget.determineRole(Class)`. Pokud třída je přeložená (kompilovaná), detekce role probíhá na základě analýzy class-objektu třídy přes reflexi. Jinak na základě analýzy zdrojového kódu. Například to, že třída je výčtovým typem, se v prvním případě pozná podle výsledku metody class-objektu `isEnum()`, v druhém

případě podle toho, že deklarace třídy ve zdrojovém kódu obsahuje klíčové slovo „enum“ místo „class“.

Pro naši úlohu to znamená, že pravidla na detekci návrhových vzorů (viz kapitola 2.1) je třeba implementovat bohužel dvakrát: přes reflexi (analýza class-objektu) a přes parsování (analýza zdrojového kódu). Analýza přes reflexi se provádí přímo v ClassTarget, zatímco analýza na základě zdrojového kódu se provádí ve třídě InfoParser (viz Obr. 30) a zjištěná informace se pak předává do ClassTarget přes přepravku ClassInfo. Shrnutí provedených úprav je uvedeno v tabulce 6.

Tab. 6: Shrnutí provedených úprav pro podporu vybraných návrhových vzorů

Třída/soubor	**Popis úprav**
bluej.pkgmgr.target.role.LibraryClassRole	- (nová třída pro návrhový vzor Knihovní třída)
bluej.pkgmgr.target.role.SingletonClassRole	- (nová třída pro návrhový vzor Jedináček)
bluej.pkgmgr.target.role.CrateClassRole	- (nová třída pro návrhový vzor Přepravka)
bluej.pkgmgr.target.ClassTarget	Upravena metoda determineRole(), vytvořeny nové metody isLibraryClass), isSingletonClass() a isCrateClass()
bluej.parser.InfoParser	Upravena vnitřní třída MethodDesc (přidána podpora pro modifikátory metod), přidána nová vnitřní třída FieldDesc (deskriptor atributů), přidán atribut fieldDesc (seznam deskriptorů atributů analyzované třídy), upraveny metody resolveComments(), gotModifier(), gotField(), gotSubsequentField(), přidána nová metoda addFieldDesc()
bluej.parser.symtab.ClassInfo	Přidány atributy isLibrary, isSingleton, isCrate a příslušné „gettery“ a „settery“.

4.2 Generátor kódu

Generování kódu je z hlediska implementace poměrně složitá funkčnost, která je v prostředí BlueJ realizována pomocí desítek tříd. Je to velmi těžké uchopit jako celek, proto je popis provedených změn rozdělen do dílčích oblastí:

- nová dialogová okna pro interakci s uživatelem při nahrávání metod,
- práce s debuggerem pro získávání hodnot všech atributů cílové třídy do zásobníku odkazů,
- parsování cílové třídy s cílem zjistit místo pro vložení vygenerovaného kódu,
- integrace všech prvků do prostředí BlueJ.

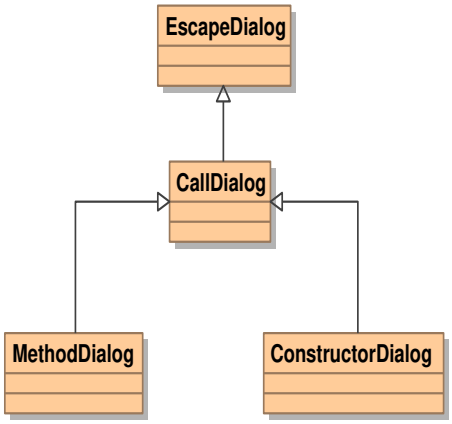
Pokročilý generátor kódu, vytvořený v rámci této práce, nebyl postaven na zelené louce. Naopak byl tento nový generátor postaven nad již existujícím generátorem pro podporu jednotkového testování. Při implementaci se autor snažil co nejvíce využít existující funkcionality a minimalizovat zásahy do zdrojového kódu projektu BlueJ. Znamená to, že pokud nějaká existující funkcionalita (například nahrávání uživatelských interakcí do sekvencí příkazů) vcelku vyhovovala pro nové účely a nebylo nutné do ní zasahovat úpravami, autor tuto oblast dále nezkoumal. Proto v této části práce není kompletní popis toho, jak je v BlueJ implementován generátor kódu, ale je zde jen popis provedených změn, které umožnily rozšířit existující funkcionalitu pro nové účely.

4.2.1 Nová dialogová okna

Při návrhu nových dialogových oken pro zadávání vstupních dat a návratové hodnoty metody použil autor jako zdroj pro inspiraci stávající dialogová okna BlueJ pro zadávání parametrů metod a konstruktorů. Cílem bylo nejen použít stávající funkcionalitu a minimalizovat množství nového kódu, ale také zajistit konzistenci nových oken se zbytkem aplikace, a to jak z hlediska vzhledu, tak i z hlediska funkčnosti. Dialogová okna pro zadávání parametrů metod a konstruktorů umí spolupracovat se zásobníkem odkazů. Pokud uživatel při zadávání parametrů klikne myší na objekt v zásobníku, jméno objektu se přenese do dialogového okna do textového políčka, které má v danou chvíli fokus. Stejné chování bylo třeba zajistit i pro nová dialogová okna.

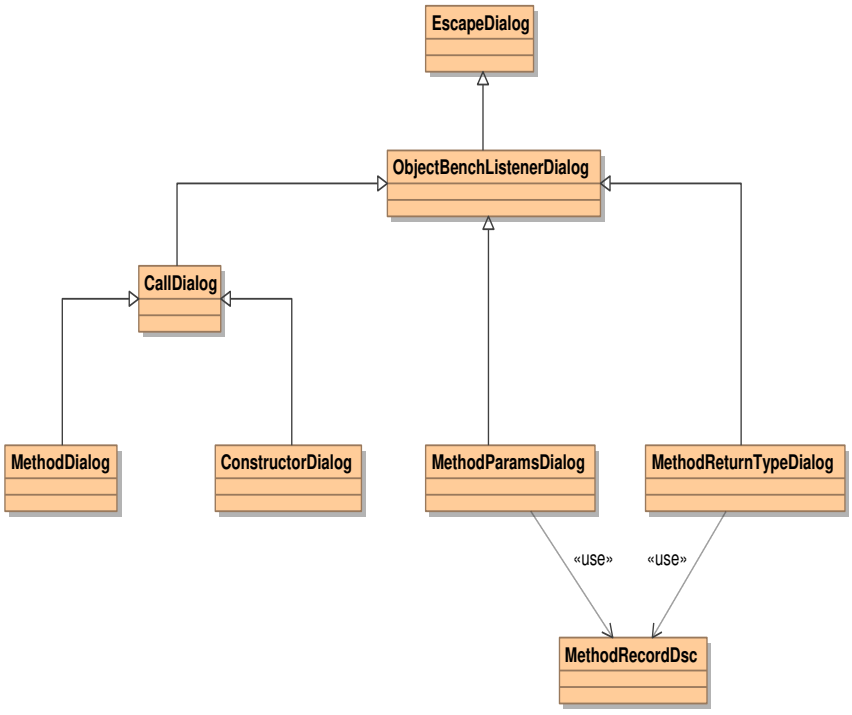
Analýzou zdrojového kódu bylo zjištěno, že téměř všechna dialogová okna v prostředí BlueJ jsou potomky třídy `EscapeDialog`, která pouze zajišťuje zavření okna po stisknutí tlačítka *Esc*. Třída má desítky potomků; diagram tříd na obr. 27 zobrazuje pouze jednu větev hierarchie – tu, která je relevantní pro účely této práce.

Třída `CallDialog` zapouzdřuje společnou funkcionalitu pro spouštění konstruktorů a metod. Zároveň tato třída také zajišťuje výše popsanou spolupráci se zásobníkem odkazů, implementovanou podle návrhového vzoru *Posluchač* (*Listener*) [17].



Obr. 27 – Diagram tříd dialogových oken (výchozí stav). Zdroj: autor

Odvodit třídy pro nová dialogová okna od třídy `CallDialog` bylo však nemožné, protože tato třída byla příliš specializovaná právě na spouštění metod a konstruktorů. Proto se autor rozhodl tuto třídu refaktorovat: nechat pouze funkcionalitu na spouštění metod a konstruktorů, funkčnost pro spolupráci se zásobníkem odkazů přesunout do nové třídy `ObjectBenchListenerDialog`, tu začlenit do hierarchie tříd mezi třídy `EscapeDialog` a `CallDialog` a nová dialogová okna pak definovat jako její potomky. Jelikož při práci s novými okny je třeba přenášet množství údajů o nahrávané metodě (název, viditelnost, parametry), byla zavedena přepravka `MethodRecordDsc`. Diagram tříd na obr. 28 znázorňuje stav po refaktoringu a zavedení tříd pro nová dialogová okna. V textové podobě tyto provedené změny shrnuje tabulka 7.



Obr. 28 – Diagram tříd dialogových oken (konečný stav). Zdroj: autor

Tab. 7: Shrnutí provedených úprav pro nová dialogová okna

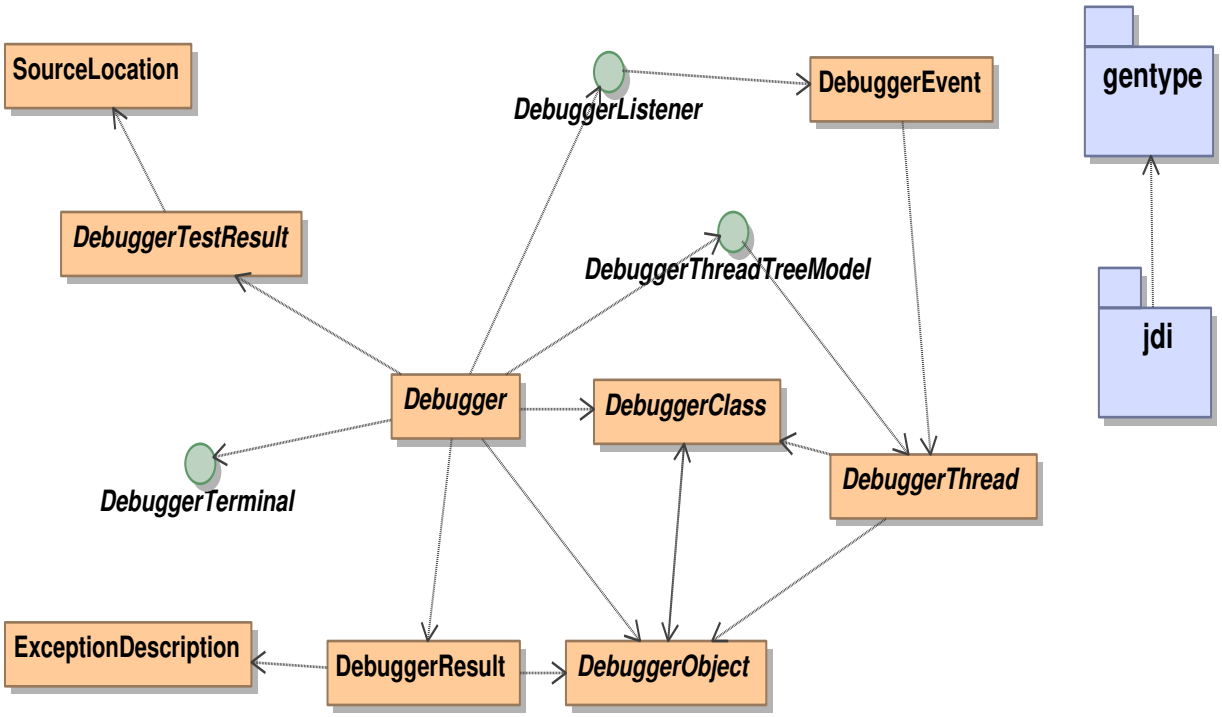
Třída/soubor	Popis úprav
bluej.debugmgr.MethodParamsDialog	- (nová třída pro dialogové okno na zadávání vstupních údajů o nové metodě)
bluej.debugmgr.MethodReturnTypeDialog	- (nová třída pro dialogové okno na zadávání návratové hodnoty metody)
bluej.debugmgr.ObjectBenchListenerDialog	- (nová třída pro dialogové okno, které poslouchá na události v zásobníku odkazů)
bluej.debugmgr.MethodRecordDsc	- (nová třída – přepravka pro údaje o nové metodě)
bluej.debugmgr.CallDialog	Změna předka z EscapeDialog na ObjectBenchListenerDialog a přesunutí do něj veškeré funkčnosti na sledování událostí v zásobníku odkazů
lib/english/labels	Texty pro nové dialogy v angličtině
lib/czech/labels	Texty pro nové dialogy v češtině
lib/english/dialogues	Texty pro nové dialogy v angličtině
lib/czech/dialogues	Texty pro nové dialogy v češtině

4.2.2 Příprava zásobníku odkazů

Před spuštěním nahrávání metody je třeba zajistit, aby se zásobník odkazů naplnil hodnotami atributů cílové třídy, se kterými uživatel může pracovat v kontextu dané metody (statické nebo instanční). Víceméně stejnou činnost provádí generátor kódu jednotkových testů při inicializaci testovacího přípravku – proto se autor práce rozhodl jít v jeho stopách.

Analýza zdrojových kódů ukázala, že při práci v interaktivním režimu spouští BlueJ veškerý uživatelský kód na vzdáleném virtuálním stroji, se kterým pracuje přes rozhraní *JDI* (*Java debugger interface*). Znamená to například, že v zásobníku odkazů uživatel ve skutečnosti komunikuje ne s objektem samotným, ale s jeho zástupcem (skutečný objekt je na vzdáleném stroji). Pro usnadnění práce *s JDI* používá BlueJ třídy z balíčku `debugger`, viz obr. 29.

V okamžiku, kdy je třeba inicializovat testovací přípravek, spouští se metoda `Debugger.runTestSetup()`, která přes `VMReference.invokeTestSetup()` spouští na vzdáleném stroji metodu `ExecServer.runTestSetUp()`. Tato metoda provede přes reflexi inicializaci příslušné testovací třídy (vytvoří instanci testu), pak spouští metodu testu `setupUp()`, a nakonec prochází všechny atributy třídy, každému nastavuje přístup `setAccessible(true)` a ukládá do pole objektů informace ve formátu `nazevAtributu1, hodnotaAtributu1, nazevAtributu2, hodnotaAtributu2` atd. Pole se pak vrací opačným směrem do `Debugger.runTestSetup()`, kde se z každé dvojice `nazevAtributu+hodnotaAtributu` vytváří instance třídy `DebuggerObject` (zástupce objektu na vzdáleném stroji), která se ukládá do zásobníku odkazů.



Obr. 29 – Diagram tříd balíčku bluej.debugger. Zdroj [7]

Pro účely přípravy zásobníku odkazů pro nahrávání metod bylo třeba výše uvedený postup poněkud upravit:

- není třeba spouštět metodu `setUp()`,
- při nahrávání statických metod je třeba vyfiltrovat jenom statické atributy,
- je třeba odlišovat hodnoty atributů primitivních datových typů.

Pro řešení posledního bodu bylo navrženo rozšířit množství informací, které *ExecServer* o každém atributu cílové třídy posílá `Debuggeru`: do dvojice `nazevAtributu+hodnotaAtributu` přidat ještě třetí položku `jePrimitivní`. V případě, že položka `hodnotaAtributu` je obalového typu, například `java.lang.Integer`, podle položky `jePrimitivní` lze poznat, jakého typu je daný atribut ve skutečnosti: `java.lang.Integer` nebo `int`.

Pokud ve skutečnosti jde o hodnotu primitivního datového typu, pouze zabalenou do objektu, pak je třeba samozřejmě zajistit správné zobrazení takového objektu v zásobníku odkazů. Toto je ale předmětem kapitoly 4.4.

Tabulka 8 shrnuje, jak byl implementován upravený postup pro získání hodnot atributů cílové třídy na vzdáleném stroji.

Tab. 8: Shrnutí úprav, zajišťujících získávání hodnot atributů cílové třídy

Třída/soubor	Popis úprav
bluej.debugger.Debugger	Přidána deklarace metody getDeclaredFields()
bluej.debugger.jdi.JdiDebugger	Přidána implementace metody getDeclaredFields()
bluej.debugger.jdi.VMReference	Přidána metoda getDeclaredFields()
bluej.runtime.ExecServer	Přidána metoda getDeclaredFields(), přidána konstanta GET_FIELDS

4.2.3 Parsování kódu cílové třídy

Parsování zdrojového kódu je v BlueJ realizováno pomocí hierarchie tříd zobrazené na Obr. 30. V čele hierarchie stojí třída `JavaParser`. Ta postupně prochází zdrojový kód, a když najde nějakou syntaktickou konstrukci, zavolá příslušnou metodu. Například zavolá metodu `beginMethodBody()`, když najde ve zdrojovém kódu začátek těla metody, a pak zavolá `endMethodBody()`, když najde konec těla metody.

Ve třídě `JavaParser` jsou metody typu `beginXxx()` a `endXxx()` implementované jako prázdné, takže nedělají vlastně nic. Konkrétní potomci některé z metod překrývají, a tak definují svoji reakci na tento prvek kódu v závislosti na zaměření daného parseru.

Například `UnitTestParser` má za cíl zmapovat názvy metod třídy a rozmístění jejich těl ve zdrojovém kódu. Podle vytvořené mapy pak lze dohledat v kódu blok (začátek a konec), který patří metodě s určitým jménem, a podle potřeby nahradit tělo nalezené metody novým kódem.

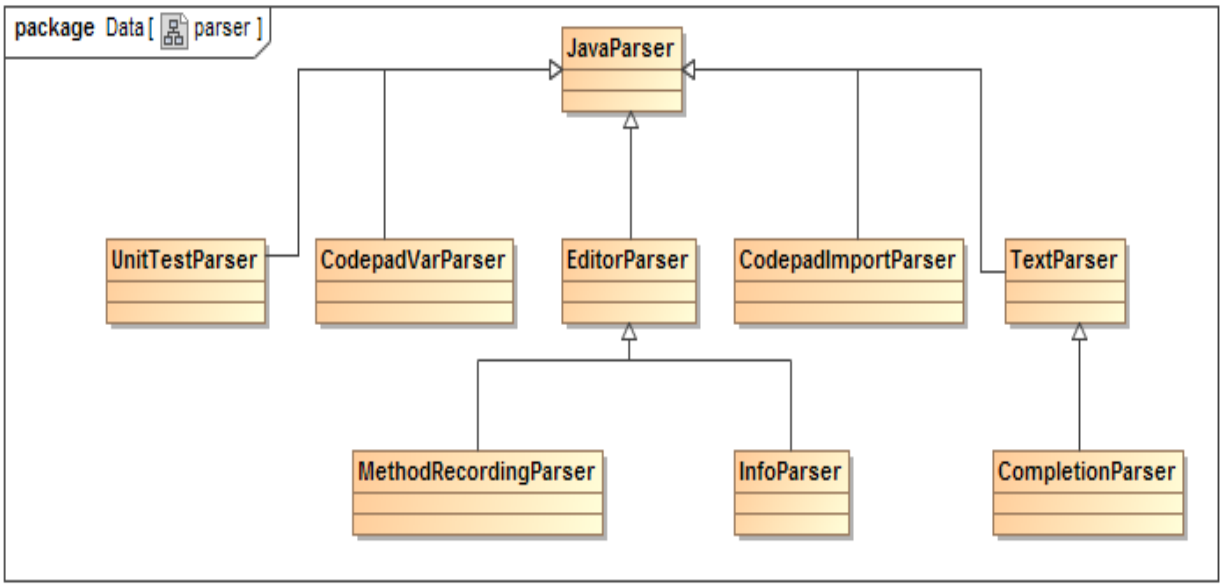
Pro účely pokročilého generátoru kódu je třeba udělat v podstatě totéž. Rozdíl spočívá jen v tom, že generátor kódu jednotkových testů podporuje pouze bezparametrické metody, takže jako klíč mapy stačilo použít jméno metody.

Pokročilý generátor kódu podporuje metody s parametry, proto by mapa měla jako klíč používat signaturu metody (jméno a seznam typů parametrů). Pro určení typů parametrů je třeba předem provést poněkud hlubší analýzu zdrojového kódu, než jen základní, jakou provádí `JavaParser`. Takovou analýzu poskytuje třída `EditorParser`, proto `MethodRecordingParser` byl navržen jako její potomek.

Shrnutí úprav je uvedeno v tabulce 9.

Tab. 9: Shrnutí úprav (parser pro pokročilý generátor kódu)

Třída/soubor	Popis úprav
bluej.parser.MethodRecordingParser	- (nová třída – parser pro pokročilý generátor kódu)
bluej.parser.MethodRecordingAnalyzer	- (nová třída – pomocná třída, která usnadňuje práci s parserem MethodRecordingParser)



Obr. 30 – Diagram tříd parserů kódu v BlueJ. Zdroj: autor

4.2.4 Integrace komponent nového generátoru kódu do prostředí BlueJ

Po dokončení všech komponent nového generátoru kódu bylo třeba je integrovat do prostředí BlueJ. Technicky to znamenalo přidat nové komponenty, především nová dialogová okna, do hlavního okna aplikace `PkgMgrFrame` a správně je propojit mezi sebou, aby se zobrazovaly ve správný okamžik a ve správném pořadí.

Dále bylo třeba přidat akce na spuštění generátoru kódu do všech příslušných místních nabídek a zajistit jejich korektní zobrazení v příslušném kontextu. Například akce *Add method declaration* by se měla dostat pouze do místních nabídek rozhraní a abstraktních tříd, přičemž v druhém případě by se měla správně zobrazovat s popiskem *Add abstract method declaration*. V porovnání s předchozí kapitolou v této části úprav bylo sice třeba provést zásah do většího množství tříd, ale jednalo se hlavně o práci s komponentami grafické knihovny *Swing*, což je mnohem jednodušší než bádání v *Java Debugger Interface* nebo *Java Reflection API.*

Souhrn provedených úprav znázorňuje tabulka 10.

Tab. 10: Shrnutí provedených úprav pro integraci nového generátoru kódu do BlueJ IDE

Třída/soubor	Popis úprav
`bluej.pkgmgr.PkgMgrFrame`	Přidány nové atributy `methodRecordDsc` a `methodReturnTypeDialog`, nové metody `methodRecordingStart()`, `setMethodRecordDsc()` a `getMethodRecordDsc()`, upraveny metody `doEndTest()` a `doCancelTest()`

bluej.debugmgr.objectbench. RecordMethodAction	- (nová třída – akce *Record method*)
bluej.debugmgr.objectbench. AddDeclarationAction	- (nová třída – akce *Add method declaration*)
bluej.pkgmgr.target.role.StdClassRole	Přidána akce *Record static method* do místní nabídky
bluej.pkgmgr.target.role.AbstractClassRole	Přidány akce *Record static method* a *Add abstract method declaration* do místní nabídky
bluej.pkgmgr.target.role.InterfaceClassRole	Přidány akce *Record static/default method* a *Add method declaration* do místní nabídky
bluej.pkgmgr.target.role.EnumClassRole	Přidána akce *Record static method* do místní nabídky
bluej.debugmgr.objectbench. ObjectWrapper	Přidána akce *Record method* do místní nabídky
lib/english/labels	Texty pro nové komponenty v angličtině
lib/czech/labels	Texty pro nové komponenty v češtině

4.3 Podpora pro lambda výrazy

Implementaci podpory pro lambda výrazy lze rozdělit na čtyři dílčí části:

- přidání příslušných prvků do GUI aplikace,
- zajištění vygenerování odkazů na metody,
- vyhodnocení lambda výrazu a přidání výsledku do zásobníku odkazů,
- správné zobrazení lambda výrazu v zásobníku odkazů.

První část byla nejjednodušší: pro zadávání lambda výrazů bylo přidáno nové dialogové okno `LambdaDefinitionDialog` jako potomek `EscapeDialog`, na zobrazení nového okna byla přidána akce *DefineLambdaAction* jako potomek `PkgMgrAction`, nové komponenty pak byly začleněny do hlavního okna aplikace `PkgMgrFrame`.

Pro druhou část bylo třeba projít aplikaci a nalézt všechna místa, kde se spouští volání metody po jejím výběru v místní nabídce třídy nebo objektu[10], s cílem zajistit, že pokud je dialogové okno `LambdaDefinitionDialog` viditelné, namísto spuštění metody proběhne vygenerování odkazu na tuto metodu. Taková místa naštěstí byla jen dvě: `ObjectWrapper.executeMethod()` pro spouštění instančních metod a `PkgMgrFrame.callMethod()` pro spouštění metod statických. Do začátku každé metody byl vložen malý `if` blok:

[10] Případně také po výběru metody ze standardní knihovny v dialogu „*Call library class*“, viz obr. 19 v uživatelské příručce.

```
if(pmf.checkLambdaDialog()) {
        pmf.getLambdaDialog().insertLambdaExpression(method, this);
        return;
}
… //další příkazy pro normální spuštění metody.
```

V podmínce `if` příkazu se testuje, zda okno pro definici lambda výrazu je právě viditelné. Pokud ano, spouštění metody se přeruší a místo něj proběhne vložení odkazu na danou metodu do příslušného políčka v dialogu na definici lambda výrazu. Pokud není viditelné, proběhne normální spuštění vybrané metody.

Pro třetí část (vyhodnocení lambda výrazů) byla použita instance třídy `Invoker`, která se stará v BlueJ o volání metod a konstruktorů [7], ale také má metodu `doFreeFormInvocation()` pro vyhodnocení libovolných Java příkazů. Tato metoda byla původně navržena pouze pro vyhodnocení kusů kódu, které uživatel zadává do panelu příkazů. V rámci této práce byla metoda použita pro několik dalších účelů:

- vyhodnocení výrazů pro hodnoty parametrů v dialogu na zadávání vstupních údajů o metodě (`MethodParamsDialog`),
- vyhodnocení lambda výrazů v dialogu `LambdaDefinitionDialog`,
- vyhodnocení výrazů pro novou hodnotu objektu v zásobníku odkazů (viz kapitola 4.5).

Vyhodnocování lambda výrazů probíhá ve dvou krocích:

```
Invoker invoker=new Invoker(pmf,null,expressionField.getText(),new LambdaWathcer());
invoker.doFreeFormInvocation(interfaceCombo.getSelectedItem().toString());
```

Nejdříve se vytvoří nová instance třídy `Invoker`, která dostává jako vstupní parametry odkaz na hlavní okno[11], text výrazu k vyhodnocení a instanci třídy `LambdaWatcher` pro zpracování výsledku vyhodnocování.

V dalším kroku se pak uskuteční vyhodnocení výrazu voláním metody `doFreeFormInvocation()`, která jako jediný vstupní parametr očekává datový typ očekávané návratové hodnoty. V tomto případě je to uživatelem zvolené funkční rozhraní.

Zpracování výsledků vyhodnocení má na starosti třída `LambdaWatcher`, konkrétně implementace rozhraní `ResultWatcher`. Toto rozhraní definuje metody pro různé události, které mohou nastat při vyhodnocování uživatelského kódu: vyhodnocení proběhlo úspěšně, vyhodnocení spadlo na chybu, kód se nepodařilo přeložit atd. Třída, která implementuje rozhraní `ResultWatcher`, přes implementaci jednotlivých metod definuje svoji reakci na tyto události, případně může tělo metody nechat prázdné, a tím pádem příslušnou událost ignorovat.

[11] Přes odkaz na hlavní okno instance třídy `Invoker` získává přístup do zásobníku odkazů, což je důležité, protože objekty ze zásobníku mohou být použity ve výrazu, který se vyhodnocuje.

V našem případě třída `LambdaWatcher` na neúspěšné vyhodnocení lambda výrazu reaguje zobrazením chybové hlášky a na úspěšný výsledek reaguje zobrazením dialogu pro uložení lambda výrazu do zásobníku odkazů, viz obr. 20 z uživatelské příručky.

Poslední problém, který bylo třeba řešit v BlueJ v souvislosti s přidáním podpory pro lambda výrazy, souvisel s tím, že BlueJ chybně identifikoval jejich datový typ, viz obr. 31.

Obr. 31 – Nesprávná detekce datového typu u lambda výrazů

Analýza zdrojového kódu ukázala, že BlueJ se snaží zjistit datový typ z názvu třídy, který dostane přes *JDI* rozhraní (`com.sun.jdi.ObjectReference.referenceType().name()`). Pro lambda výrazy to bude textový řetězec ve stylu „`A$$Lambda$2/21865865`“, kde A je název třídy, uvnitř které byl definován daný lambda výraz. Bylo nutné toto chování upravit tak, aby se u lambda výrazu jako jeho datový typ používalo příslušné funkční rozhraní.

Pak vznikl další problém: podle čeho přesně poznat, že daný objekt je lambda výraz. Autor očekával, že v nějakém *API* najde metodu `isLambda()`, obdobně jako *Java Reflection API* obsahuje metody `isEnum()`, `isInterface()` atd. Hlubší zkoumání různých specializovaných zdrojů, například [9] a [10], bohužel nepřineslo žádné použitelné výsledky. Proto se autor rozhodl využít výše uvedenou textovou podobu názvu třídy u lambda výrazů a přidal do třídy `JdiUtils` vlastní metodu `isLambda()` , kterou implementoval takto:

Výpis 1: Metoda `isLambda()`

```
public boolean isLambda(String className) {
       return className.contains("$$Lambda$");
}
```

V budoucnu, až se objeví lepší způsob, jak rozpoznat lambda výraz, bude třeba tuto metodu přepsat.

Tabulka 11 obsahuje krátký přehled všech změn, které byly provedeny v rámci implementace do BlueJ podpory pro lambda výrazy.

Tab. 11 Shrnutí úprav pro podporu lambda výrazů

Třída/soubor	Popis úprav
`bluej.pkgmgr.PkgMgrFrame`	Přidány nové atributy `defineLambdaAction` a `lambdaDefinitionDialog`, nové metody `defineLambda()`, `checkLambdaDialog()` a `getLambdaDialog()`, upraveny metody

	callMethod(), setupMenus() a setupActionDisableSet()
bluej.pkgmgr.actions.DefineLambdaAction	- (nová třída – akce *Define lambda expression*)
bluej.debugmgr. LambdaDefinitionDialog	- (nová třída – dialog pro definici lambda výrazů)
bluej.debugger.jdi.JdiUtils	Nová statická metoda isLambda()
bluej.debugger.jdi.JdiDebugger	Upravena metoda guessNewName()
bluej.debugger.jdi.JdiObject	Upraven konstruktor
bluej.debugmgr.objectbench. ObjectWrapper	Upraveny metody findIType() a executeMethod()
lib/english/labels	Texty pro nové komponenty v angličtině
lib/czech/labels	Texty pro nové komponenty v češtině
lib/english/dialogues	Texty pro nové dialogy v angličtině
lib/czech/dialogues	Texty pro nové dialogy v češtině

4.4 Uložení primitivních hodnot do zásobníku odkazů

V předchozí kapitole bylo zmíněno, že třída Invoker se stará o vyhodnocení uživatelských výrazů a o volání metod a konstruktorů. Pro další úpravy nyní potřebujeme prozkoumat, jak přesně daná třída zajišťuje tyto funkce. Hlubší analýza ukázala, že třída Invoker je v podstatě generátor kódu, který při každém volání vygeneruje kostru dočasné třídy – potomka třídy bluej.runtime.Shell.

Dovnitř této dočasné třídy se pak zabalí kód, který je třeba spustit, například volání konstruktoru nebo metody. Invoker tuto dočasnou třídu uloží, přeloží a pak přes bluej.debugger.Debugger spouští na vzdáleném stroji.

Následující výpis ukazuje vygenerovaný kód dočasné třídy pro volání metody getX() objektu třídy Elipsa v zásobníku odkazů:

Výpis 2: Dočasná třída SHELL1

```
public class __SHELL1 extends bluej.runtime.Shell {
public static java.lang.Object run() throws Throwable {
final bluej.runtime.BJMap __bluej_runtime_scope =
getScope("D:\\tmp\\109z_Abstraktn\u00ed_t\u0159\u00eddy");
final Elipsa elipsa1 = (Elipsa)__bluej_runtime_scope.get("elipsa1");
try {
return makeObj(elipsa1.getX()
);}
finally {
}
}}
```

Je vidět, že výsledek metody `elipsa1.getX()` je zabalen do metody `makeObj()`. Třída `bluej.runtime.Shell` obsahuje verzi této metody pro každý primitivní datový typ. Další výpis obsahuje variantu metody `makeObj()` pro typ `int`:

Výpis 3: Metoda `makeObj()`

```
protected static Object makeObj(final int i)
{
	return new Object() {
		@SuppressWarnings("unused")
		public int result = i;
	};
}
```

Metoda `makeObj()` vytváří objekt-přepravku a do jeho atributu `result` ukládá skutečný výsledek spouštění cílového kódu. Tento trik umožňuje při zaslání výsledku ze vzdáleného stroje na lokální nepřijít o jeho skutečný datový typ. Bez tohoto zabalení by debugger nepoznal, zda výsledek ve skutečnosti byl primitivního typu nebo nebyl.

Autor se rozhodl přidat do objektu-přepravky pro primitivní datové typy ještě atribut `wrappedResult` odpovídajícího obalového typu. Takový trik umožní přes atribut `result` zachovat informaci, že ve skutečnosti je výsledek primitivního datového typu, ale zároveň přes atribut `wrappedResult` získat odkaz, který lze vložit do zásobníku odkazů. Následující výpis ukazuje upravenou verzi metody `makeObj()`:

Výpis 2: Modifikovaná metoda `makeObj()`

```
protected static Object makeObj(final int i)
{
	return new Object() {
		@SuppressWarnings("unused")
		public int result = i;
		public Integer wrappedResult = i;
	};
}
```

Samozřejmě stejná úprava byla provedena pro všechny varianty metody `makeObj()`.

Dále podpora pro tento trik na vzdáleném stroji byla přidána do odpovídajících tříd na straně lokálního stroje. Tak například třída `ResulInspector`, která zajišťuje zobrazení výsledku volání metody, byla upravena tak, aby zobrazovala jako výsledek hodnotu atributu `result`, ale při kliku na tlačítko „*Get*“ poslala do zásobníku odkazů hodnotu atributu `wrappedResult`.

Třída `ObjectWrraper`, která se stará o vykreslování objektu v zásobníku odkazů, byla upravena tak, aby u hodnot takovýchto „falešných“ primitivů zobrazovala místo skutečného objektového typu odpovídající primitivní typ. To je umožněno díky tomu, že třída má dvě

odlišné vlastnosti `className` (název skutečného datového typu) a `displayName` (název typu pro zobrazení), jimž lze nastavit odlišné hodnoty. Také je zajištěno, aby se u „falešných“ primitivů nezobrazovaly v místní nabídce žádné metody.

Dále do třídy `JavaPrimitiveType` bylo doplněno dvousměrné mapování mezi názvy primitivních datových typů a odpovídajících obalových. Shrnutí všech změn je uvedeno v tabulce 11.

Tab. 12: Shrnutí úprav pro vložení hodnoty primitivních typů do zásobníku odkazů

Třída/soubor	Popis úprav
`bluej.pkgmgr.PkgMgrFrame`	Upravena metoda `putObjectOnBench()`
`bluej.runtime.Shell`	Upravena metoda `makeObj()` pro všechny primitivní datové typy
`bluej.debugmgr.inspector.ResultInspector`	Upravena metoda `listElementSelected()`
`bluej.debugger.gentype.JavaPrimitiveType`	Přidána mapování `primitiveToWrapper` a `wrapperToPrimitive`, přidána nová metoda `getPrimitiveByName()`
`bluej.debugger.DebuggerObject`	Přidána deklarace nové metody `isReallyPrimitive()`
`bluej.debugger.jdi.JdiObject`	Nový logický atribut `reallyPrimitive`, implementace metody `isReallyPrimitive()` a nový „setter“ `setReallyPrimitive()`
`bluej.debugger.jdi. JdiArray`	Metoda `isReallyPrimitive()` natvrdo vrací `false`
`bluej.debugmgr.objectbench. ObjectWrapper`	Nový logický atribut `fakePrimitive`, upraveny metody `getGenType()`, `getTypeName()`, `createMenu()`, nová metoda `fakePrimitive()`

4.5 Změna hodnoty objektu v zásobníku odkazů

Tato funkce byla implementována dvěma způsoby: jako uložení výsledku volání metody nebo konstruktoru do již existujícího objektu v zásobníku odkazů a jako přímá možnost změnit hodnotu objektu v jeho místní nabídce. V obou případech ve skutečnosti operace probíhá ve dvou krocích:

- odstranění ze zásobníku odkazů stávajícího objektu s daným jménem,
- přidání do zásobníku odkazů nového objektu pod stejným jménem.

Je to dáno tím, že v zásobníku odkazů je pouze zástupce objektu, který je ve skutečnosti na jiném virtuálním stroji a ke kterému se přistupuje přes debugger `bluej.debugger.Debugger`. Debugger nabízí pouze metody na přidání a odstranění objektu: `addObject()` a `removeObject()`.

Kromě samotné změny hodnoty objektu bylo třeba zajistit, aby se tato operace korektně zapsala do sekvencí příkazů při generování kódu. Bylo proto nutné lehce upravit několik potomků třídy `InvokerRecord`.

Shrnutí provedených změn je uvedeno v tabulce 13.

Tab. 13: Shrnutí úprav pro změnu hodnoty objektu v zásobníku odkazů

Třída/soubor	**Popis úprav**
`bluej.pkgmgr.PkgMgrFrame`	Upraveny metody `callMethod()` a `putObjectOnBench()`, nová metoda `askForReplacement()`
`bluej.debugmgr.objectbench.ObjectWrapper`	Upravena metoda `createMenu()`, nová metoda `changeValue()`
`bluej.testmgr.record.MethodInvokerRecord`	Upraveny metody `benchDeclaration()` a `benchAssignmentTypecast()`
`bluej.testmgr.record.ConstructionInvokerRecord`	Upravena metoda `toTestMethod()`, přidána implementace metody `setBenchName()`
`lib/english/labels`	Texty pro nové komponenty v angličtině
`lib/czech/labels`	Texty pro nové komponenty v češtině
`lib/english/dialogues`	Texty pro nové dialogy v angličtině
`lib/czech/dialogues`	Texty pro nové dialogy v češtině

Závěr

Cílem této práce bylo prozkoumat a zanalyzovat nestandardní možnosti vývojového prostředí BlueJ využitelné ve vstupních kurzech programování, a na základě toho navrhnout a následně realizovat nové funkce pro rozšíření možností tohoto prostředí.

Práce se nejprve zaměřuje na analýzu metodik výuky programování (především v úvodních kurzech) a poté i jejich podpory ve výukovém prostředí BlueJ. Na základě analýzy bylo zjištěno, že za nejmodernější metodiku výuky programování je možno označit *Architecture First*, ale bohužel prostředí BlueJ nenabízí pro výuku podle této metodiky postačující podporu. Metodika *Architecture First* předpokládá, že studenti v první fázi nebudou psát programy ručně, ale budou používat generátor kódu. Stávající generátor kódu BlueJ měl ale velmi omezené možnosti, protože původně byl navržen pouze pro nahrávání jednotkových testů, a tedy umožňoval generovat jen veřejné bezparametrické `void` metody, a to pouze do testovacích tříd.

Jeden z hlavních přínosů této práce spočívá tedy právě v tom, že na základě původního omezeného generátoru kódu byl postaven plnohodnotný generátor kódu, který umožňuje přidávat do jakékoliv třídy metody s libovolným počtem parametrů, libovolnou návratovou hodnotou a libovolným modifikátorem přístupu. Navíc nový generátor kromě generování kódu celé metody umožňuje generovat kód i pro pouhé deklarace nové metody.

Kromě pokročilého generátoru kódu byly implementovány další nové funkce, jako například zvýraznění vybraných návrhových vzorů v diagramu tříd nebo možnost měnit hodnotu objektu v zásobníku odkazů. Za důležité autor považuje také přidání podpory pro vybrané novinky Java 8, a to pro lambda výrazy a defaultní implementace metod v rozhraní.

Výsledkem této práce je upravené a rozšířené prostředí BlueJ s lepší podporou výuky podle metodiky *Architecture First*. Tímto lze považovat cíle práce za splněné. Je ale na místě upozornit, že vývoj projektu tím nekončí. Až se upravená verze BlueJ začne používat v reálné výuce, lze očekávat zpětnou vazbu v podobě požadavků na opravu případných drobných chyb, a také možných návrhů na další rozšíření.

Terminologický slovník

Termín	Zkratka	Význam [Zdroj]
Back-end	-	pozadí aplikace; část, která nekomunikuje s uživatelem [autor]
Datový typ		Datový typ definuje v programování druh nebo význam hodnot, kterých smí nabývat proměnná (nebo konstanta) [24].
Debugování (ladění)	-	Metodický postup pro nalézání a snižování množství chyb v počítačových programech nebo elektronického hardware tak, aby fungoval, jak se předpokládá [25].
Framework		Softwarová struktura, která slouží jako podpora při programování a vývoji a organizaci jiných softwarových projektů. Může obsahovat podpůrné programy, knihovny API, podporu pro návrhové vzory nebo doporučené postupy při vývoji [26].
Front-end		Front-end: přední část aplikace komunikující přímo s uživatelem [autor].
Getter		Metoda pro získání vlastnosti třídy, obvyklý formát getXxx(), kde Xxx – název vlastnosti [autor].
Graphical User Interface	GUI	Grafické uživatelské rozhraní, které umožňuje ovládat počítač pomocí interaktivních grafických ovládacích prvků [27].
Integrated Development Environment	IDE	Software usnadňující práci programátorů, většinou zaměřené na jeden konkrétní programovací jazyk. Obsahuje editor zdrojového kódu, kompilátor, případně interpret a většinou také debugger [28].
Java Debugger Interface	JDI	Rozhraní pro poskytující informace o stavech spuštěné JVM [7]
Jednotkové testování		Činností související s vývojem aplikačních programů. Pod pojem „jednotkové testování“ („unit testing“) se zahrnují nástroje, metodika a činnost, jejímž cílem je ověřování správné funkčnosti dílčích částí neboli jednotek zdrojového kódu [29].
Kompilace		Převod algoritmu zapsaného v nějakém programovacím jazyce do strojového kódu překladačem [30].
Parsování		Proces analýzy posloupnosti formálních prvků s cílem určit jejich gramatickou strukturu vůči předem dané (byť ne nutně explicitně vyjádřené) formální gramatice [31].
Red, green, blue	RGB	Aditivní způsob míchání barev používaný ve všech monitorech a projektorech (jde o míchání vyzařovaného světla), tudíž nepotřebuje vnější světlo (monitor zobrazuje i v naprosté tmě) na rozdíl např. od CMYK modelu [32].
Reflexe		Reflexe umožňuje získávat informace o atributech, metodách a konstruktorech načtených tříd a s objekty daných částí dále pracovat [7]

Setter		Metoda pro nastavení vlastnosti třídy, obvyklý formát setXxx(arg), kde xxxx – název vlastnosti, arg – nová hodnota pro danou vlastnost [autor].
Signatura metody		Unikátní identifikátor metody v rámci typu [33].
Stereotyp UML		Stereotypy umožňují použití existujících prvků UML a vytvoření z nich nových. Pro jeho označení se používají francouzské lomené uvozovky [34].
Unified Modeling Language	UML	Grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů [35].
Validace		Kontrola vstupních údajů při zadávání dat [36].

Použité zdroje

[1] PECINOVSKÝ, Rudolf. Architecture First. In: *Journal of Technology and Information Education.* Olomouc: Palacký University, 2013, s. 107-117. ISSN 1803-537X. Dostupné z: http://jtie.upol.cz/clanky_1_2013/JTIE-1-2013.pdf

[2] PECINOVSKÝ, Rudolf. Skvělý program nemusí být skvěle prodejný. In: *Objekty*. Ostrava, 2007. Dostupné z: http://vyuka.pecinovsky.cz/prispevky/2007-OB_Skvely_program_nemusi_byt_skvele_prodejny.pdf

[3] TAFT, Darryl. Top 10 Programming Languages for Job Seekers in 2014. *Eweek.com* [online]. 2014 [cit. 2014-05-01]. Dostupné z: http://www.eweek.com/developer/slideshows/top-10-programming-languages-for-job-seekers-in-2014.html/

[4] BAUER, Tomáš. *Programy pro podporu výuky programování v OOP*. Praha, 2009. 57 s. Bakalářská práce. Vysoká škola ekonomická v Praze.

[5] BARNES, D J. -- KÖLLING, M. *Objects first with Java : a practical introduction using BlueJ.* Boston: Pearson, 2012. ISBN 978-0-13-283554-1.

[6] PECINOVSKÝ, R. *Java 7 - učebnice objektové architektury pro začátečníky.* 1. vyd. Praha: Grada Publishing, 2012. 495 s. ISBN 978-80-247-3665-5.

[7] BATELKA, Martin. *Analýza open-source verze programu BlueJ*. Praha, 2009. 42 s. Bakalářská práce. Vysoká škola ekonomická v Praze.

[8] Patterson, A., Tool Support for Introductory Software Engineering Education. PhD Thesis, Monash University, Australia, 2002

[9] GOSLING, James, Bill JOY, Guy STEEL, Gilard BRACHA a Alex BUCKLEY. ORACLE AMERIKA. *The Java® Language Specification: Java SE 8 Edition*. Redwood City, 2014. Dostupné z: http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

[10] LINDHOLM, Tim, Frank YELLIN, Gilard BRACHA a Alex BUCKLEY. ORACLE AMERIKA. *The Java® Virtual Machine Specification: Java SE 8 Edition*. Redwood City, 2014. Dostupné z: http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

[11] PECINOVSKÝ, Rudolf. Současné trendy v metodice výuky programování. In: *Počítač ve škole.* Nové město na Moravě, 2006. Dostupné z: http://vyuka.pecinovsky.cz/prispevky/2006-PS_Soucasne_trendy_v_metodice_vyuky_programovani.pdf

[12] VYSOKÁ ŠKOLA EKONOMICKÁ V PRAZE. *Učební plán pro bakalářský obor Informatika*. 2013. Dostupné z: http://fis.vse.cz/wp-content/uploads/2013/06/Informatika_bc_2013_www.pdf

[13] 4IT101 - Základy programování. *Výuka programování a softwarového inženýrství na KIT VŠE* [online]. 2013 [cit. 2014-05-01]. Dostupné z: http://java.vse.cz/Main/HomePage

[14] PECINOVSKÝ, Rudolf. Jak při výuce Javy opravdu začít s objekty. In: *Objekty*. Praha, 2004. Dostupné z: http://vyuka.pecinovsky.cz/prispevky/2004-OB_Jak_pri_vyuce_Javy_opravdu_zacit_s_objekty.pdf

[15] PECINOVSKÝ, Rudolf. Začlenění návrhových vzorů do výuky programování. In: *Objekty*. Ostrava, 2005. Dostupné z: http://vyuka.pecinovsky.cz/prispevky/2005-OB_Zacleneni_navrhovych_vzoru_do_vyuky_programovani.pdf

[16] PECINOVSKÝ, Rudolf, PAVLÍČKOVÁ Jarmila, PAVLÍČEK Luboš. Let's Modify the Objects-First Approach into Design-First. Bologna 26.06.2006 – 28.06.2006. In: ITICSE06 [CD-ROM]. New York : ACM, 2006, s. 188–192. ISBN 1-59593-346-8.

[17] PECINOVSKÝ, Rudolf. *Návrhové vzory.* Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[18] Why Enum Singleton are better in Java. *Javarevisited* [online]. 2013 [cit. 2014-05-02]. Dostupné z: http://javarevisited.blogspot.cz/2012/07/why-enum-singleton-are-better-in-java.html

[19] JELÍNEK, Lukáš. Java 8: lambda výrazy, nové API pro datum a čas, paralelní řazení polí. In: *Linuxespres.cz* [online]. 2014 [cit. 2014-05-02]. Dostupné z: http://www.linuxexpres.cz/novinky/java-8-lambda-vyrazy-nove-api-pro-datum-a-cas-paralelni

[20] PECINOVSKÝ, Rudolf. Functional Programming Constructs in Java 8 and Their Integration into Lessons of Object Oriented Architecture . In: *Federated Conference on Software Development and Object Technologies* . Jihlava, 2013. Dostupné z: http://edu.pecinovsky.cz/papers/2013_OBen_Functional_constructs_in%20Java_8_and_education_of_sw_architecture.pdf

[21] HORSTMANN, Cay S. *Java SE 8 for the really impatient*. Addison-Wesley, 2014, ISBN 03-219-2776-1.

[22] Scala (programovací jazyk). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2009 [cit. 2014-05-02]. Dostupné z: http://cs.wikipedia.org/wiki/Scala_(programovac%C3%AD_jazyk)

[23] 4IT101 - Konvence pro psaní a odevzdávání programů v Javě. *Výuka programování a softwarového inženýrství na KIT VŠE* [online]. 2013 [cit. 2014-05-01]. Dostupné z: http://java.vse.cz/4it101/Konvence

[24] Datový typ. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Datov%C3%BD_typ

[25] Ladění (programování). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Lad%C4%9Bn%C3%AD_%28software%29

[26] Framework. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Framework

[27] Grafické uživatelské rozhraní. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Grafick%C3%A9_u%C5%BEivatelsk%C3%A9_rozhran%C3%AD

[28] Vývojové prostředí. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/V%C3%BDvojov%C3%A9_prost%C5%99ed%C3%AD

[29] Unit testing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Unit_testing

[30] Kompilace. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Kompilace

[31] Syntaktická analýza. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Parsov%C3%A1n%C3%AD

[32] RGB. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/RGB

[33] JANÁČEK, Ondřej. Signatura metody. *Dotnetportal.cz* [online]. 2014 [cit. 2014-05-05]. Dostupné z: http://www.dotnetportal.cz/blogy/12/Ondrej-Janacek/5372/C-Internals-Method-Overloading

[34] Další prvky UML, stereotypy. *Objekty.vse.cz* [online]. 2005 [cit. 2014-05-05]. Dostupné z: http://objekty.vse.cz/Objekty/MetodikyANotace-UMLStereotypy

[35] Unified Modeling Language. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-05]. Dostupné z: http://cs.wikipedia.org/wiki/Unified_Modeling_Language

[36] Pojem validace. *Slovnik-cizich-slov.abz.cz* [online]. 2013 [cit. 2014-05-05]. Dostupné z: http://slovnik-cizich-slov.abz.cz/web.php/slovo/validace

Seznam obrázků

Seznam tabulek

Přílohy

Na přiloženém CD-ROM se nachází:

1. Upravená verze programu BlueJ
2. Upravené zdrojové kódy programu (dostupné i online: https://www.assembla.com/code/bluej_advanced/git/nodes)