

Charles University in Prague  
Faculty of Mathematics and Physics

## MASTER THESIS



Bc. Jonáš Klimeš

## Domain-Specific Language for Learning Programming

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Pavel Parízek, Ph.D.

Study programme: Computer Science

Specialization: Software Systems

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, 12<sup>th</sup> May 2016

Jonáš Klimeš

I would like to express deep gratitude to my supervisor Pavel Parízek for guidance and advices throughout this work. Moreover, I would like to thank Václav Pech from JetBrains for his support with MPS, counselling this work, and possibility to work on this thesis in the JetBrains office.

Název práce: Doménově specifický jazyk pro výuku programování

Autor: Bc. Jonáš Klimeš

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Pavel Parízek, Ph.D.

Abstrakt:

V rámci této práce jsme navrhli jazyk pro výuku programování. Nejprve jsme popsali osm existujících nástrojů pro výuku programování a identifikovali jsme jejich vlastnosti, které jsou důležité pro proces učení. Potom jsme navrhli výukový doménově specifický jazyk Eddie. Eddie je vhodný pro dospívající děti a dospělé, kteří se chtějí naučit programovat. Jazyk používá doménu postavenou na jazyku Robot Karel, ve které uživatelé mohou ovládat postavičku robota ve dvourozměrné mřížce.

Vytvořili jsme prototyp jazyka Eddie pomocí nástroje MPS Language Workbench. Jazyk Eddie postupně představuje cykly, podmínky, proměnné, funkce a objekty. Uživatelské programy mohou být vytvářeny a spouštěny ve vývojovém prostředí Eddie Studio. Eddie Studio také vizualizuje akce robota ve spuštěném programu.

Klíčová slova: výuka programování, doménově specifický jazyk, MPS, nástroj pro tvorbu jazyka

Title: Domain-Specific Language for Learning Programming

Author: Jonáš Klimeš

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Pavel Parízek, Ph.D.

Abstract:

In the scope of this thesis, we designed a language for programming education. At first, we described eight existing tools for learning programming and identified key features in the learning process. Second, we designed an educational domain-specific language Eddie. Eddie is suitable for teenagers and adults who want to learn programming. It uses a domain based on Karel the Robot language, where users can control a robot character in a two-dimensional grid.

We implemented a prototype of Eddie using the MPS Language Workbench and its projectional editor. The Eddie language gradually introduces loops, conditionals, variables, functions, and objects. Eddie programs can be created, executed and visualized in the Eddie Studio IDE.

Keywords: programming education, domain-specific language, MPS, language workbench

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Goals .....	1
1.2	Structure of the Thesis .....	2
<b>2</b>	<b>Existing Languages and Systems</b>	<b>3</b>
2.1	Language Descriptions .....	3
2.2	Language Comparison .....	18
2.3	Key Features .....	22
<b>3</b>	<b>MPS Language Workbench</b>	<b>25</b>
3.1	Projectional Editor .....	25
3.2	Language Design .....	26
3.3	Language Aspects .....	28
3.4	Integrated Development Environment .....	39
3.5	Advantages .....	40
3.6	Limitations .....	41
<b>4</b>	<b>Language Design</b>	<b>42</b>
4.1	Key Requirements .....	42
4.2	Domain .....	43
4.3	Environment .....	44
4.4	Language Constructs .....	46
4.5	Syntax .....	56
4.6	Map and Tutorial Definition .....	61
<b>5</b>	<b>Implementation</b>	<b>65</b>
5.1	Non-Functional Requirements .....	65
5.2	Architecture .....	65
5.3	Language .....	67
5.4	Runtime .....	76
5.5	Eddie Panel .....	79
5.6	Standalone IDE .....	80
<b>6</b>	<b>Tutorial</b>	<b>82</b>
<b>7</b>	<b>Future Work</b>	<b>90</b>
7.1	Language .....	90
7.2	Environment .....	92
7.3	Education .....	93
<b>8</b>	<b>Conclusion</b>	<b>95</b>
	<b>Bibliography</b>	<b>96</b>

<b>Attachments</b>	<b>97</b>
Attachment A List of Karel the Robot Implementations.....	97
Attachment B Content of Enclosed DVD .....	98

# 1 Introduction

Information technology and computing take gradually more and more important part in the lives of people. Smart phones and tablets are spreading. More different devices, from cars and trains to fridges and vacuum cleaners, are controlled by computers. There is a great demand for software engineers and developers. Ability to write small programs is becoming essential also in other fields such as biology and economy. Therefore, many people want to learn programming and to understand how the computing technology works. In addition, governments and educational experts consider that introduction to computer science and algorithmic thinking would be a part of elementary school curriculum. As a consequence, they need tools and languages for learning programming.

For the purpose of teaching people to write their first programs and understand algorithms, currently used general-purpose languages, such as C or Java, are too complex and difficult to use. Although some older and less complex general-purpose languages (e.g. Pascal) or simplified versions of modern languages might be easier to understand, a user-friendly IDE (Integrated development environment) with intelligent code completion is missing in most cases.

We believe that Domain Specific Languages (DSLs) which are designed for learning programming and for which a user-friendly IDE is provided, can be the right way to address this topic.

## 1.1 Goals

The main goal of this work is to design a DSL-based system that would offer safe and intuitive ways to learn programming. The system would consist of a simple language (DSL), a run-time system, and an IDE.

Three tasks have to be fulfilled in order to achieve this goal. The first task is to explore existing educational programming languages, describe their key features, and discuss their strong and weak points. The second task is to identify the key characteristics of tools and languages that have impact on the process of learning programming, and based on these characteristics, choose a suitable domain for our DSL. The third task is to design our own educational DSL and implement a working prototype that would allow people to interactively learn the basics of programming



including some object-oriented concepts. The prototype should consist of a DSL specification, an IDE that would enable users to write their own programs, runtime that would enable to run the programs, and also some tutorial that would help users to understand the key programming concepts. The prototype implementation would be based on the *MPS Language Workbench*[1].

This work aims to support education of writing programs in some language using language constructs. Therefore, we focus on language statements and their syntax, and neglect standard algorithms and data structures.

## **1.2 Structure of the Thesis**

Chapter 2 describes existing educational languages and tools, and compares them. Chapter 3 introduces the MPS Language Workbench and describes how to edit, represent and store the code in MPS and its support for design of languages. Chapter 4 presents our DSL, which is called Eddie. It describes the domain of Eddie, all language constructs and their syntax, and the runtime environment. We also discuss auxiliary languages for definition of map and tutorial. Chapter 5 provides closer look at the prototype implementation of Eddie, how Eddie is decomposed into MPS languages, and how the language, the runtime environment, and the IDE is implemented. Chapter 6 describes a tutorial that we created. Chapter 7 presents ideas for future work and then we conclude in Chapter 8.

## 2 Existing Languages and Systems

This chapter describes eight existing languages and programming tools that either have education of programming as the main goal or they are easy to use for beginners. We consider the following languages: Scratch, Code.org, Karel, Computer programming course at Khan Academy, Peter, Touch Develop, BlueJ, and Greenfoot.

We listed the languages in a specific order. The first four ones are DSLs, while the second four can be considered as general purpose languages. Some of the languages use graphical code representation instead of textual source code. Graphical representation offers safer programming because it prevents users from making syntax errors.

The first two languages, Scratch and Code.org, represents a category of DSL with graphical code blocks instead of text. We choose them because they offer quick introduction to the world of programming for beginners. The next one is Karel, a traditional and very small educational language. The last DSL, Khan Academy, was chosen because it focuses on introducing coding principles and a good practice.

The first of the general-purpose languages is Peter. It represents a complex language with graphical code representation. Touch Develop allows to create mobile applications and makes mobile application development easier for beginners. At last, there are two systems that introduce simplified editor and environment for Java.

All selected languages are described in the next section. Then we compare them in Section 2.2. Finally, in Section 2.3, we describe their key features that are important for a learning process.

### 2.1 Language Descriptions

All language descriptions in the following subsections are of the same paragraph structure. The first paragraph defines what kind of tool or language it is, and the second one describes the main features. The third paragraph describes how the system or language can be used and how does its user interface look like, and the fourth paragraph defines a target group for the language. Finally, the last two paragraphs in each subsection discuss advantages and limitations of the language.

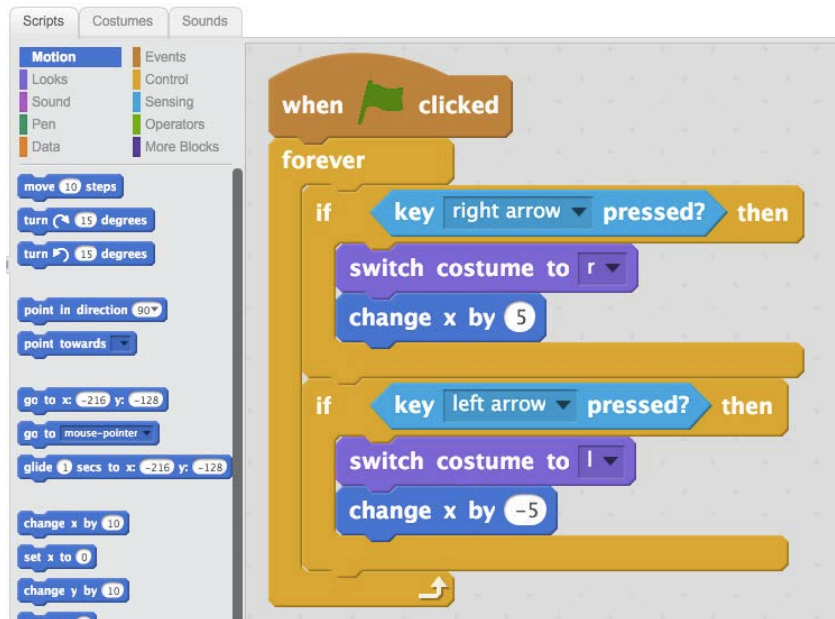
### 2.1.1 Scratch

Scratch[2] is a visual programming tool that allows to program interactive stories, games and animations. Scratch is a project of the *Lifelong Kindergarten Group* led by Mitchel Ransack at the MIT and was initiated in 2003.

Scratch is a DSL that enables to control active objects called *sprites*. Each sprite is displayed as a picture on the screen. Scripts that control sprites are made of graphical code blocks. Figure 2.1 shows a screenshots from the Scratch code editor. On the right side, there is a script made of code blocks. Code blocks look like a jigsaw puzzle, thus users know which constructions fit together according to the shape of the block. This prevents users from making syntax errors. On the left side of the image, there is a library with all available code blocks. The blocks are divided into following groups:

- Control: if-then-else, repeat-until, repeat-n-times, forever, wait, ...
- Motion: move n steps, turn around n degrees, go to coordinates, ...
- Event: when click, when key pressed, when receive message, broadcast message, ...
- Data: initializing and setting variables
- Operators: arithmetic, logical, comparison, ...
- Sensing: what is sprite touching
- Pen: drawing
- Looks: changing an appearance of a sprite
- Sound: playing sound and music

Each group has a different colour, which makes code more clear. Users may create a new project, or they may inspect and modify existing projects of other users.



*Figure 2.1 A screenshot of a Scratch program.*

Source: <https://scratch.mit.edu/projects/102273168/#editor>

Scratch is designed especially for youngsters from 8 to 16 years old, who appreciate its playfulness.

Scratch is available both online and as a desktop application for Windows, Mac OS, and Linux. The source code of Scratch 1.x is available under the GPLv2 license and Scratch Source Code License.

### **Advantages**

The main advantage of Scratch is the representation of code using graphical blocks, which leads to a safe way of creating code. Another benefit is the availability as a web application, thus users may immediately start to code. It is also translated into more than 50 languages. Online-created projects are available to other users to see the code and modify it. Therefore, existing projects may serve also as a source of inspiration and motivation.

### **Limitations**

The main limitation of Scratch is that some users may perceive it more like a game than as a programming language. It does not use the concept that everything is composed from primitive atomic commands. There are some complex commands that could be substituted by others, e.g. `move n steps` instead of `move` used in a `repeat` loop. This can make transition from graphical code blocks to textual

source code of standard languages more difficult. Some commands offered in Scratch can be also confusing for beginners.

### 2.1.2 Code.org

Code.org[3] is a non-profit organization dedicated to computer science education. Their goal is to make computer science available in more schools, and increase involvement of women and students of ethnic minorities. The website code.org contains a few short starter tutorials and education methodologies for teachers with videos, tutorials to learn programming, and activities without computers. There are 20 hours long courses for youngsters from the age of 4, 6, 8, 10 years old and also one accelerated course for students between 10 and 18 years old. For this list of educational languages, we chose the tutorial *Hour of code*, which is the first tutorial that was introduced there.

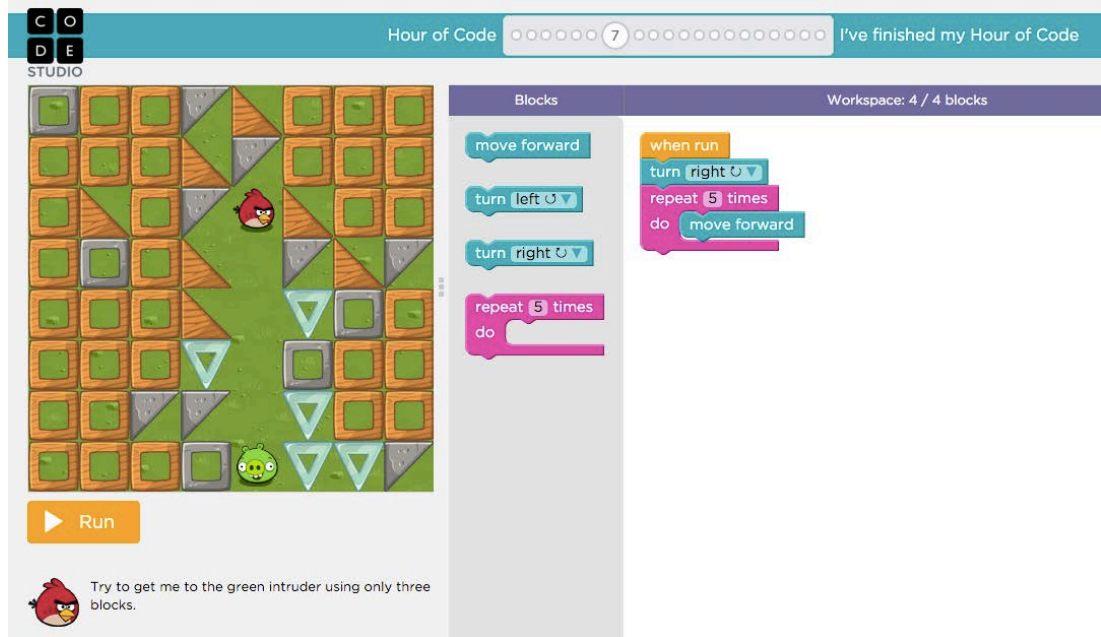


Figure 2.2 A screenshot of Hour of code tutorial.

Source: <https://studio.code.org/hoc/7>

Hour of code is an interactive tutorial that aims to introduce basics of programming in an hour. It uses similar graphical code blocks as Scratch. Figure 2.2 shows how the tutorial looks like. The screen of code.org tutorial is horizontally divided into three sections. There is a map with characters and task description on the left side. In the middle, there are blocks, which can be used to complete the task, and on the right, there is the actual script that should be executed. There are two kinds of blocks. The first ones are blocks to move the character and the second ones are control flow

commands: `repeat times`, `repeat until`, `if-then`, and `if-then-else`. Control flow commands are introduced gradually during the tutorial. Commands are introduced in videos featuring some famous people, including Bill Gates and Mark Zuckerberg.

Hour of code targets to primary school children and teenagers, but since it straightforwardly explains programming basics in an hour, it can be used as a starting point to learn programming also by adults.

It is available online and for free. With the help of volunteers from different countries, it is translated into more than 40 languages.

### **Advantages**

Hour of code aims not only to introduce programming, but it also motivates people to learn about it using stories of famous people both from IT and other fields, e. g. athletes, fashion models. Another benefit is that the tutorials are understandable and easy to use. Graphical code blocks are used, but it is also explained that there is some source code under the hood of each graphical code block. When a user completes a task, she can see the actual source code they created. Tutorial themes such as *Angry Birds* might be interesting for kids.

### **Limitations**

The main limitation of the tutorial is that there is no freedom of what users can program. There are always just the graphical command blocks to complete the task. And when the task is completed users move forward to the next task.

### **2.1.3 Karel**

Karel[4] is an educational DSL created by Richard E. Pattis in 1981. Pattis introduced the language in his courses at Stanford University, California. The language is named after Karel Čapek, a Czech writer who created the word robot.

Karel allows users to control a robot that lives in a two-dimensional grid. It uses standard textual source code. The robot has five basic instructions:

- `move`: Karel moves by one square in the direction he is facing
- `turnleft`: Karel turns around 90° to the left
- `putbeeper`: Karel puts a beeper on the square where he is standing
- `pickbeeper`: Karel lifts a beeper off the square where he is standing

- `turnoff`: Karel switches himself off, the program ends

It supports also three control flow statements – `if` and `while` conditional statements, and an iteration loop with a given count of iterations. For the purpose of conditionals, the robot can perform Boolean queries about its immediate environment. He can ask whether there is a beeper, where he is standing, whether there is enough space to put beeper, whether there is a wall in front of him, and whether he is facing north, east, south, or west. A user can define additional instructions that is composed of the constructs that are mentioned above. Figure 2.3 shows an example of Karel program that contains a custom instruction `turnright`.

```
BEGINNING-OF-PROGRAM

DEFINE turnright AS
BEGIN
  turnleft;
  turnleft;
  turnleft;
END

BEGINNING-OF-EXECUTION
  ITERATE 3 TIMES
  BEGIN
    turnright;
    move
  END
  turnoff
END-OF-EXECUTION

END-OF-PROGRAM
```

*Figure 2.3 An example code of Karel that defines procedure `turnright` and then uses it in the main program.*

Because Karel has a textual syntax, it is suitable both for adults and teenagers, who are able to write code and correct potential syntax errors.

There exist several implementations of Karel, both desktop and online. Implementations can have different syntax and runtime visualization. There is also an implementation of Karel as an example of a sample language in the MPS Language Workbench. In Attachment A, we provide a list of Karel implementations that we found.

## Advantages

The main advantage of Karel is that it uses the concept of a minimal set of instructions. The set contains atomic robot moves, environment queries, and control flow statements. Any other robot commands must be defined using this set. This concept is common in computer science and Karel helps users to understand this concept. In general, a simple domain together with quite a small set of keywords makes it easier to use. Users are not distracted by any advanced features. The topic of controlling a robot is related to programming, which can influence user's motivation to coding.

## Limitations

The fact that Karel is rather a small language may be perceived also as a limitation, because users can learn only programming basics in the standard version of Karel. Another point is that almost all the implementations that we discovered do not have an editor with intelligent code completion. The only exception is *RobotKaja* that is used as a sample language in the MPS Language Workbench.

### 2.1.4 Khan Academy

Khan Academy is an organization that tries to improve education by providing it free for anyone and anywhere. There is also *Computer programming*[5] course at Khan Academy. This course is an interactive tutorial, which uses JavaScript and a domain of 2D drawings.

The course consists of a sequence of chapters, where each chapter contains several tutorials. It starts with drawing basics, and then introduces variables, animation, functions, conditions, loops, and arrays. Then it follows with a chapter about object-oriented programming (OOP), which introduces objects, properties, methods, and inheritance. There are also chapters about debugging, documentation, and how to writing code that is easy to read and understand.

Figure 2.4 shows the tutorial interface. There is a code editor with intelligent code completion and error checking on the right, live preview of the code output on the left, and tabs with language documentation, questions, and tips written by other users where they can also discuss. Some tutorials contain an interactive editor in the form of a voice commentary with a video that shows how the code is being written. However, users can stop the video at any time, edit the code on the screen and see the new output live at the same time. Apart from this interactive tutorial, there are also



short motivation videos describing where are computer programs used and why people should learn about them.

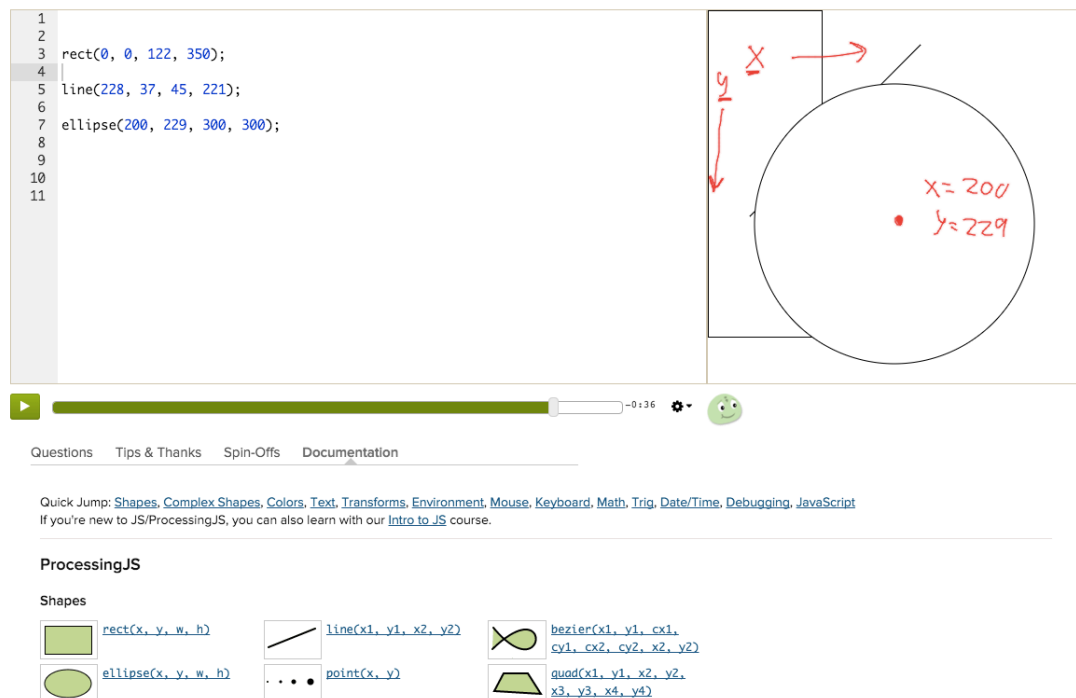


Figure 2.4 A screenshot of Khan Academy Computer programming course. Source: <https://www.khanacademy.org/computing/computer-programming/programming>

The target group for *Khan Academy* are teenagers and adults. It uses textual syntax. Code is more complex than in the case of *Karel* and contains parentheses and semicolons, but code is suggested and corrected automatically in the editor. It is available online and for free.

## Advantages

There is a step-by-step tutorial that is well explained and easy to control. The domain of drawing has several benefits. The first is that it is generally known and it is easier for people to understand programming, when it is explained using a domain that they already know, than to learn both programming and some new domain at once. Another benefit is that it is possible to have a live preview, which is not so easy in other domains (e.g. in the domain of controlling robot). This allows users see how the output changes when they make a little change to the code. The next benefit is that drawing brings space for creativity, which is also important for programming. Tutorials also cover many programming concepts. Finally, it uses JavaScript that is one of the top used languages, which can motivate people to learn it.

## Limitations

On the other hand, the domain of drawing can be little boring for some people, especially at the beginning before loops and conditions are introduced. Beginners, especially children, may have problems to write syntactically correct code. They have a hard time to remember the count and order of parameters of functions and the block structure of an `if` statement. They also forget parentheses and semicolons, so the textual representation may not be a perfect fit for beginners.

### 2.1.5 Peter

Peter[6] is a visual programming tool developed in the Czech Republic in 1999 by Ing. Miroslav Němeček. *Peter* is a general programming language and a complete tool for programming.

It supports basic programming statements, typed variables, functions, mathematics, and also features for 2D and 3D graphics, operations with files, and network communication.

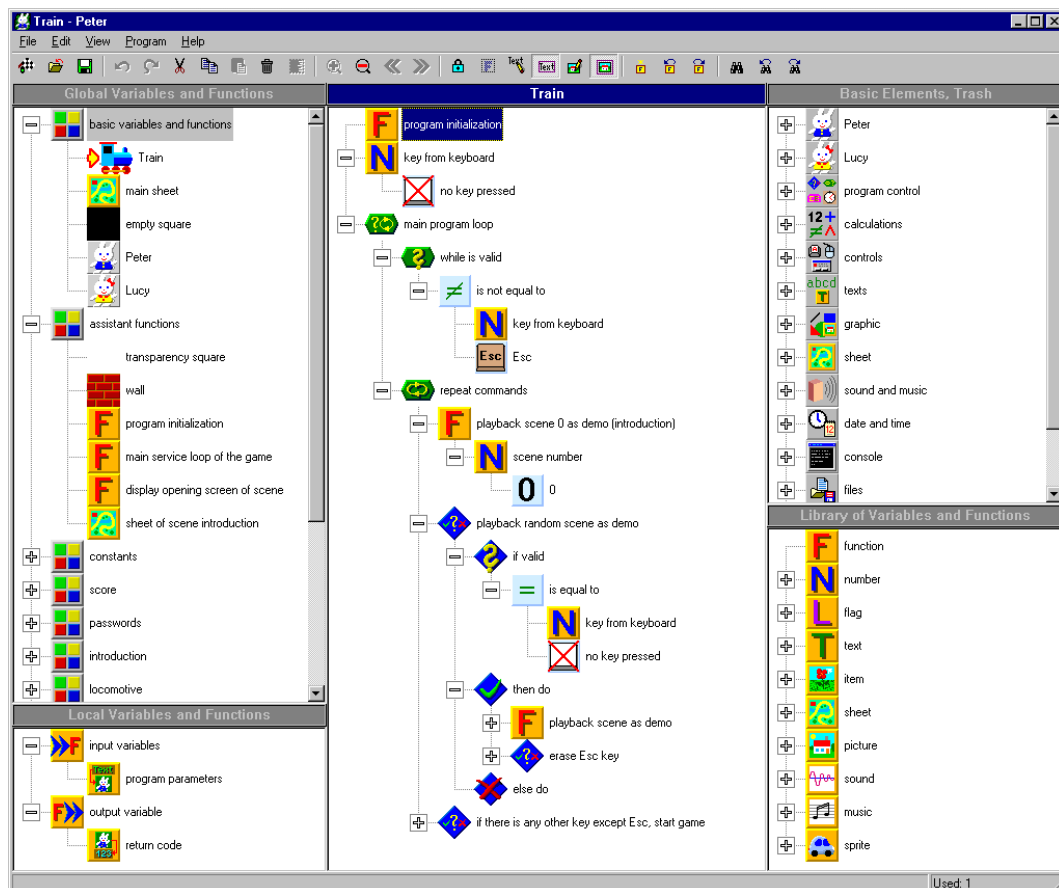


Figure 2.5 A screenshot of Peter IDE.  
Source: <http://www.gemtree.com/peter.htm>

The principal characteristic of Peter is a graphical representation of the program structure (abstract syntax tree). Users create programs using a mouse. They move icons of language constructs from a toolbox to a program. Therefore, a system does not allow creating an element combination that would cause syntax errors. Figure 2.5 shows Peter IDE. It has five windows. A window in the middle contains a graphical representation of current program code. An upper right window contains language constructs that can be dragged and placed into the code in the middle. In a lower right window, there are different types of variables that can be used. Windows on the left display global and local variables that are already declared. There is a simple domain for beginners. In this domain, users can control a rabbit called Peter and move with him around a two-dimensional map. Advanced users can use the features mentioned above to create standalone applications for Microsoft Windows.

*Peter* is accessible to beginners, even for primary school children. It is a desktop application for Windows 95/98/NT/ME/2000/XP/Vista and the last version 2.30 was released in 2009. It was originally commercial product, but since the 1<sup>st</sup> September 2012, it is freeware. Since the 1<sup>st</sup> September 2013, it is open source.

### **Advantages**

The main advantage of Peter results from its graphical code representation. Program is created from code blocks so it is not possible to make syntax errors. Users can also better understand how the code is represented as abstract syntax tree. Beginners can start programming with basics programming concepts such as conditions or loops. There is also a simple domain of controlling a rabbit, which users can use for their first steps to learn programming.

### **Limitations**

Currently, the main limitation of Peter is that it is only for Windows platform and it is no longer supported by the creators. It is not clear whether it is compatible with Windows 8 and later versions. It might be difficult to switch from this graphical code structure to some standard programming language. Peter is more complex in comparison to other tools, such as code.org.

### **2.1.6 Touch Develop**

Touch Develop[7] is an online visual programming language for creating applications for mobile devices. It is developed at Microsoft Research. Touch Develop provides an interactive environment for developing, testing, and running

programs. Users can share their programs and modify programs of other users. It contains also several tutorials.

Touch Develop allows to create complete mobile applications. An application consists of several features; the main features are: functions, objects, global variables, pages, events, picture and music resources, persistent data structures, and libraries. Functions contain executable code, and objects define member fields and functions. Variables are typed; they can be either primitive or instances of objects. Pages define screens of application user interface, they can contain pictures, music and form elements. Events allows to react to a user input, e.g. a user taps on some object on the screen or shakes with the device. Persistent data structures allow keeping data after application is turned off. Data can be stored locally on a device or in the cloud. Data are stored in tables or in key-value indexes. Libraries allow using some prepared external libraries. Figure 2.6 shows Touch Develop user interface. There is a simple menu on the left, next to it, there is a list of all the structures that are in the application. On the right side, there is an editor for the selected structure.

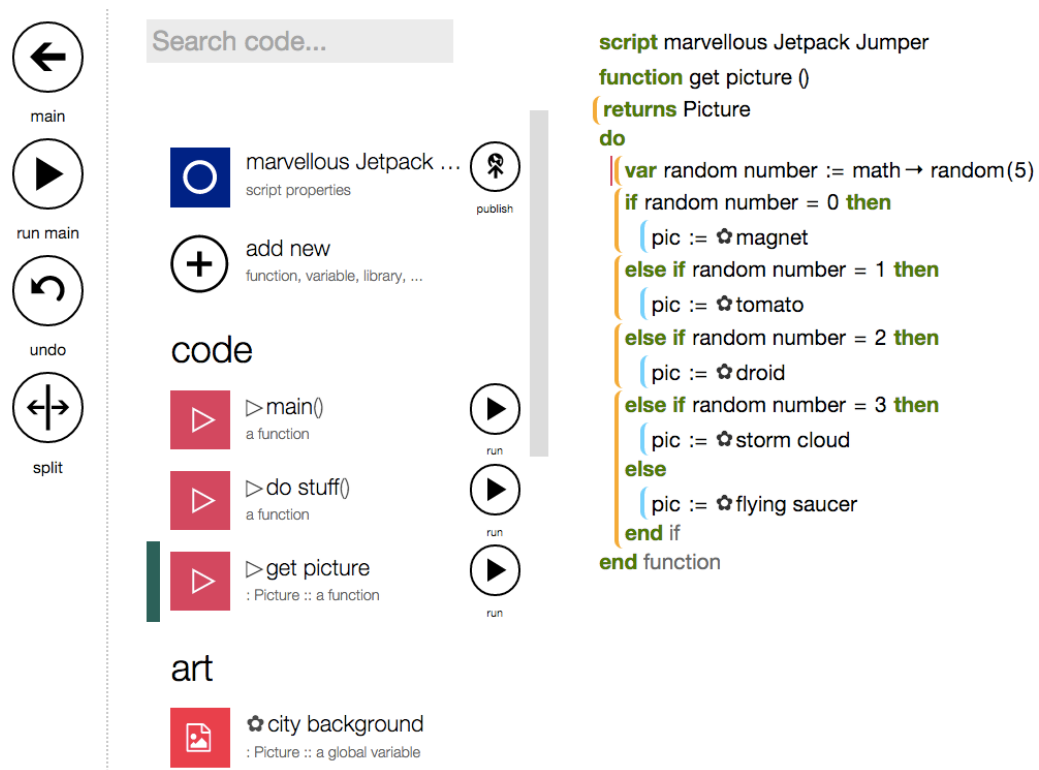


Figure 2.6 A screenshot of Touch Develop IDE. Source: The Marvellous Jetpack Jumper application from <https://www.touchdevelop.com/app/>

Users can choose from three skill levels: Beginner, Coder, and Expert. At the Beginner level, advanced features are hidden and code is displayed and edited in the

form of visual blocks. The Coder uses textual syntax and enables most of the features. The Coder level is a default level. The Expert level enables all the features and uses a syntax that is similar to JavaScript. Code levels change only the presentation layer, so users can switch between them any time, in any project. Touch Develop contains also some interactive tutorials.

Touch Develop is the most suitable for people, who already have some experience and want to create some real applications. Beginners can switch to the Beginner level with code blocks and use the tutorials, but compared to code.org or Khan Academy, the Touch Develop IDE is less intuitive.

Touch Develop is provided as a web application or as a mobile application for Android or Windows Phone. User interface of Touch Develop IDE is designed for mobile devices with touch display. Created applications can be exported as Android, iOS, Windows Phone, or HTML5 applications.

### **Advantages**

Mobile and web applications are very popular these days and many desktop applications are migrated to the web or provided with mobile versions. Unfortunately, for people who want to learn programming for these platforms, it usually requires more knowledge than creating desktop application. Touch Develop breaks this rule and brings a simpler way how to create applications that can be used on all types of smart phones. Users can select one of three skill levels and as they learn more, they can increase their level. Expert level code is similar to JavaScript, so users can learn one of the currently most used languages. Three skill levels make Touch Develop also suitable for a larger group of users. Another good thing about Touch Develop is its availability. It is an online tool, so user does not have to install anything and they can try it immediately. It can be also easily used on tablets or phones with big screen, so users can create their programs in more situations such when they travel in public transport.

### **Limitations**

Touch Develop limitations come from the fact that it is a full language to create mobile applications, so even at the Beginner level, it can be too complicated to use for people without any experience with programming. Next limitation is that the user interface is designed for touch screens and it is less user-friendly to use it with mouse at desktop computers.

### 2.1.7 BlueJ

BlueJ[8, 7] is a simple IDE for Java that is developed for educational purposes. The project was started by Michael Kölling and John Rosenberg at Monash University.

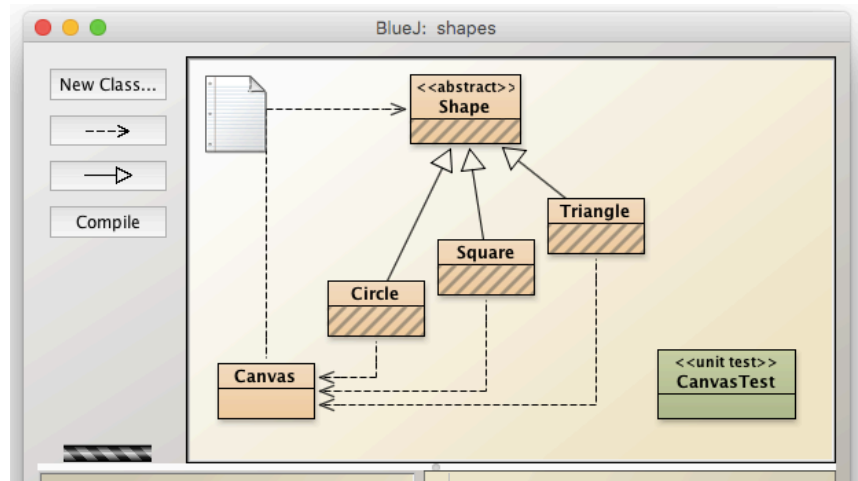
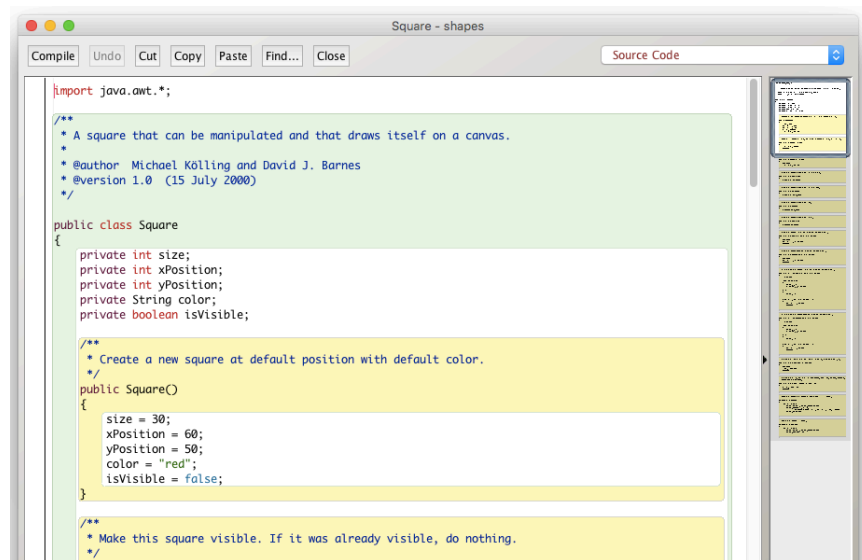


Figure 2.7 Screenshot of BlueJ application structure editor

The main part of the BlueJ IDE is an *application structure view* that visualise the structure of the Java application. Figure 2.7 shows how it is visualised, it looks like an UML diagram. It is possible to manage *classes*, *interfaces*, *enums*, *Java Applets*, and unit tests there. They can be interactively created, deleted, and organized into packages. It is also possible to create dependencies and inheritance relationships among the elements. Every element can be opened in the source code editor. Figure 2.8 shows the class editor with Java syntax highlighting. It also offers a code completion for the methods of current class, but the code completion is very simple and it can be invoked also when a cursor is in a position where it is not allowed to call methods.

Created classes can be instantiated in something like a sandbox. Users then can use a command window to write simple Java commands. Commands can manipulate with the created instances. It is also possible to inspect the state of the created instances. When a code is written, it can be compiled and then methods can be run from the application structure view. Errors are not checked before compilation and they are reported one by one. BlueJ contains also some tutorials and sample projects, e.g. drawing shapes in Java Swing or maintaining a database of people.



*Figure 2.8 Screenshot of BlueJ class code syntax highlighting*

BlueJ is suitable for people who already know programming basics, who are at least familiar with Java syntax, and who want to learn Java and object-oriented programming.

BlueJ is maintained by University of Kent and it is available free under the GNU General Public License. It is distributed as an installation package with Java JDK 8 for Windows and Mac OS, or it can be downloaded and run as a Java application.

### **Advantages**

BlueJ provides a simplified user interface that can be easier to use for beginners in comparison with common Java IDEs (IntelliJ IDEA, Eclipse, Netbeans). Application classes are represented graphically. Users can create instances of the classes in IDE, try custom code with the instances, and inspect state of the instances. Users can observe and easier understand what is happening inside the objects by repeating these steps. This helps to understand the differences between classes and instances, and other object-oriented concepts. Syntax highlighting has different background colour for a class, its methods, and the rest of the code. This makes the class code clearer.

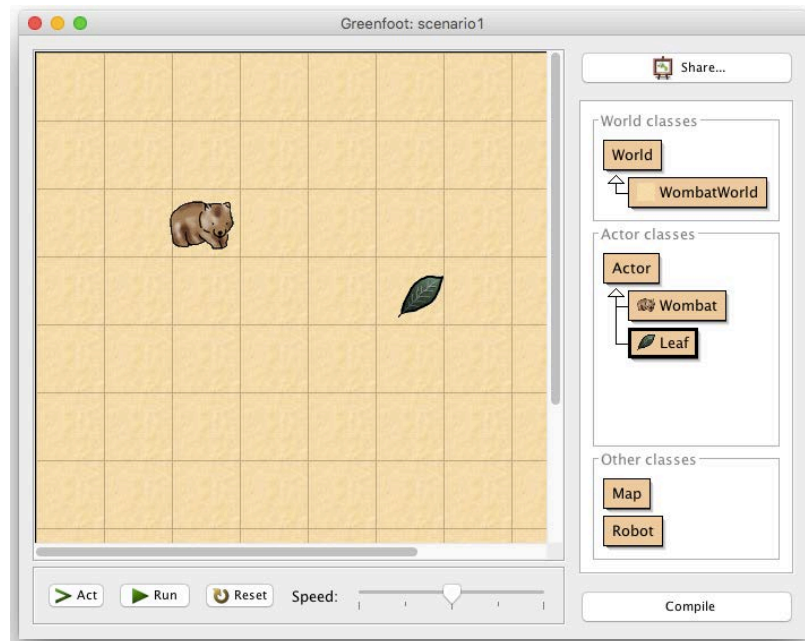
### **Limitations**

Although BlueJ provides simplified IDE, it is generally less user friendly than common Java IDEs. It does not have an intelligent code completion of Java keywords and it does not report errors immediately when they are written, but the

errors are reported one by one after compilation. Most of the sample projects are just fragments of applications and it is not possible to run them. Therefore, it takes quite a lot of time to create some visible output, which can be frustrating for users.

### 2.1.8 Greenfoot

Greenfoot[9] is a simplified Java IDE developed in University of Kent. It is build on the top of BlueJ and it has a prepared environment that allows to create simple visual scenarios.



*Figure 2.9 A screenshot of Greenfoot IDE shows Wombat world with a wombat and a leaf.*

Greenfoot introduces a domain of *world* and *actors* that inhabit the world. Both world and actors are Java classes that extend predefined classes `World` and `Actor`. The world is a two dimensional grid of cells and an actor is an object that exists in the world. It has a location, a graphical appearance, and some behaviour defined in a method named `act`. Users can define their own worlds and actors. Worlds, actors, and other classes can be edited as Java source code. There is the same editor that is used in BlueJ. Figure 2.9 shows the main window of Greenfoot, there are defined classes on the right, world graphical output on the left, and control panel at the bottom. Users can create an instance of defined actors and drag them to some position in the world. Then they can compile the code and run it. Execution in Greenfoot consists of a built-in main loop that repeatedly invokes the `act` method for each actor.



Greenfoot targets to a similar group of people as BlueJ. These people should have already some basic knowledge, but the graphical output of Greenfoot makes it available for little less experienced people, who would appreciate to see some output of their work soon.

Greenfoot, same as BlueJ, is maintained by University of Kent. It is distributed in the same way as BlueJ and the source code is also available under the same general public license.

### **Advantages**

Same as BlueJ, Greenford provides simplified user interface, that can be easier to use for beginners in comparison with common Java IDEs. It has a prepared environment (world and actors), and graphical runtime, so users can quickly see some output of their code. Initialization of the scenario is simple, users just need to create an instance of `Actor` class or its subclass and insert it to some place in the world by mouse.

### **Limitations**

The code editor is same as in BlueJ, so it has the same limitations regarding the code completion and error reporting.

## **2.2 Language Comparison**

We compare existing languages and systems based on the following criteria: whether it is DSL or general-purpose language, how is the code created and edited, theme of the tool (if there is any), how easily can users move from the educational language to some commonly used language, which programming concepts does it support, how it is available and easy to use for beginners, and how does it support learning process. The learning process criteria are: whether there is any tutorial, space for creativity, and freedom. The comparison results are discussed in this section. For the sake of clarity, we created four comparison tables.

First, Table 2-1 contains general criteria. DSLs are more understandable for beginners and usually show users what kind of task they should do. And even general purpose language can introduce some simple domain for beginners. We believe that general-purpose languages are good for users that have some realistic idea what kind of program they want to create. Then they can spend more time coding it in comparison with DSL. However, for the beginners it can be difficult to choose a

program to create that is suitable for their level of skills, they can tend to choose ambitious work and then struggle with it.

General								
	Scratch	Code.org	Karel	Peter	Khan Academy	Touch Develop	BlueJ	Greenfoot
DSL	✓	✓	✓		✓			✓ <sup>1</sup>
General-purpose language				✓		✓	✓	✓
Graphical code structure	✓	✓		✓		✓ <sup>2</sup>		
The theme of the domain is close to programming	✓	✓	✓	✓	✗			
Relationship with common languages: uses common language (***), has similar syntax (**), has similar constructs (*)	✗	*	*	*	***	**	***	***

*Table 2-1 General features of languages comparison*

Graphical code representation is more accessible to beginners, especially children. Code blocks in a shape of jigsaw puzzle can also provide syntax checking. On the other hand, writing textual source code can be more effective than moving graphical blocks using mouse. Textual source code is also closer to standard programming and allows easier transition to standard languages. Touch Develop introduces probably the best way how to deal with this issue. It lets users to switch between graphical blocks and textual code.

DSLs must have some domain, the domain should be understandable, and we think it should be somehow connected with programming. The most of the selected DSLs use programming to control some character that is moving in some space. We believe that from the motivation point of view, this is a good choice of theme in

<sup>1</sup> Greenfoot uses Java, but it introduces its own domain using Java.

<sup>2</sup> Touch Develop has three skill levels, the beginner level uses graphical code blocks.

comparison with drawing in Khan Academy, because when someone wants to draw a picture she can use image-editing software, but when she want to control a robot, it makes sense that they need to know computer programming.

Advanced concepts								
	Scratch	Code.org	Karel	Peter	Khan Academy	Touch Develop	BlueJ	Greenfoot
Variables	✓	✗	✗	✓	✓	✓	✓	✓
Functions	✗	✗	✓	✓	✓	✓	✓	✓
Object-oriented programming	✗	✗	✗	✗	✓	✓	✓	✓
Testing	✗	✗	✗	✗	✗	✓	✓	✓
Debugging	✗	✗	✗	✗	✗	✓	✓	✓

*Table 2-2 Advanced concepts of languages comparison*

Educational languages are good start, but people who want to become software developers once must eventually switch to some standard programming language. In our opinion, when users know they can easily switch to some standard programming language, it increases their motivation to learn. There are several ways how to make this transition easy. The easiest way is to use some standard language in some educational environment. Another way is to use similar syntax or at least use similar language constructs. Among the selected languages, it is probably the hardest to move from Scratch to a standard language, because Scratch contains many building blocks that does not exactly match with constructs of standard languages. Some educational languages also introduce advanced programming concepts and help users to understand them in simplified environment. Table 2-2 shows summary of these advanced concepts in the languages.

Some languages are ready to use online and users can run their first code in a few minutes. Some others require installation and reading manual. It is clear that the first case is better. In addition, an intuitive user interface helps beginners to dive in. These criteria are compared in Table 2-3.

Accessibility								
	Scratch	Code.org	Karel	Peter	Khan Academy	Touch Develop	BlueJ	Greenfoot
Available online	✓	✓	✓ <sup>3</sup>		✓	✓		
Available in more languages	50+	40+		2	6		18	18
Easy to use: very intuitive (***) – needs to use manual (*)	**	*		***	*	**	***	***
How long it takes from visiting website to run some code: 0–10 min (*), 10–30 min (**), more than 30 min (***)	*	*	**	***	*	*	*** <sup>4</sup>	** <sup>4</sup>

*Table 2-3 Accessibility of language comparison*

It is crucial for an educational tool, how it supports and facilitates learning process. Table 2-4 covers these criteria. We believe that the best way, how to do it, is when there is some interactive tutorial. Textual tutorial that are available separately from the tool also supports learning process, but requires more effort from the users. In the Hour of Code tutorial at Code.org, users can use only prepared set of blocks and after they complete the task they cannot play with the scene any more. We believe that in the tutorial a user should be led, but there should be also space to finish the task in her own way, because in programming there is never only one solution. From our experience, a good educational tool should also support creativity of users. Scratch and Touch Develop did the best job here, because they allow creating custom scenarios, animations, and games and they provide galleries of pictures and music where users can find inspirations.

<sup>3</sup> Some implementations of Karel are available online.

<sup>4</sup> It expects to know basics of Java.

Learning process								
	Scratch	Code.org	Karel	Peter	Khan Academy	Touch Develop	BlueJ	Greenfoot
Interactive tutorials		✓			✓	✓		
Text tutorials				✓			✓	✓
Creativity: high (***) – low (*)	***	*	*	**	**	***	*	**
Freedom	✓	✗	✓	✓	✓	✓	✓	✓

Table 2-4 Learning process comparison

## 2.3 Key Features

Based on the analysis and the comparison of languages from the previous sections, we identified some key features that have benefits for learning programming. It is not intended as characteristic of an ultimate educational language, but more like set of points that should be considered when an educational language is designed. Some mentioned features might be incompatible with the others. Features are specified as an itemized list. Each item is marked and some of them are referenced from the Chapter 4 that describes our educational language.

F1 Language is available and easy to use.

F1.1 It is available online and it is prepared for immediate use, which means also that no registration is required.

F1.2 The user interface is intuitive and users do not have to read the manual.

F1.3 Code can be run easily by a single button.

F1.4 It is available in other languages and there is a mechanism that allows volunteers from different countries to translate it into their languages.

F2 It motivates users.

F2.1 It introduces all different aspect of people's life that are influenced by computer programming and how can be things made better using computer science.

F2.2 It contains motivation videos featured some famous people.

F2.3 DSL has interesting themes.

F3 There are prepared tutorials.

- F3.1 The tutorials are understandable and well structured.
- F3.2 Tutorial scenarios are ordered from the easiest one to more difficult ones, but users can move forward without complete the previous tasks if they think it is too easy for them.
- F3.3 Users are not limited how they finish the task.
- F3.4 After finishing the task, users may go through again and try different approaches.
- F3.5 Users should be able to play with current task even after they complete it. They can try different ways or they can observe what exactly happens when they do something wrongly.
- F4 The domain of DSL is well known or easily understandable.
- F5 Code structures are organised into groups (control statements, variables, expression, ...) and they are graphically distinguished, e.g. marked with different colours.
- F6 Language concepts are presented gradually. It starts with introducing basic concepts while the other concepts are hidden (from menu, from code completion, ...). Advanced concepts are also introduced one by one.
- F7 Language editor contains an intelligent code completion.
- F8 Language editor prevents users from making syntax errors and when they make some, it is reported immediately (this can be achieved using graphical code blocks in the shape of jigsaw puzzle).
- F9 Code execution output is visible for users. It is even better if there is live preview of the output, while a user is editing the code.
- F10 It supports creativity of users and inspires them to create something. This can be achieved for example by prepared pictures of characters, background pictures of different places, ...
- F11 A language that uses graphical code representation displays the relationship between the graphical code and the textual source code.
- F12 There is a different code representation for different skill levels, e.g. there is graphical code block for beginners, textual source code with educational syntax for advanced users, and syntax similar to some general used language for expert level.
- F13 It is easy to move from the educational language to some language that is actually used in the industry. One way how this can be achieved is to have similar language concepts to some real language and to display the relationship

between the educational language and the real language. Another way is to use similar syntax to some real language.

F14 It leads users to produce high-quality source code.

F14.1 It introduces code documentation, testing, and refactoring and explains why it is important.

F14.2 It makes users to think about their coding style and shows some good practices.

F14.3 It breaks the stereotype that shorter code is automatically better than longer code, which many programmers believe.

F15 There is space for experimenting with objects and it allows inspecting their runtime state.

F16 For advance concepts such as objects, it offers also different viewpoint than a source code, e.g. UML like diagram in BlueJ.

## 3 MPS Language Workbench

MPS Language Workbench is a tool that allows creating custom extensible domain-specific language (DSL). It is developed by company JetBrains and it licensed under the Apache 2 open-source license. It is based on the JetBrains' open-source platform for building IDEs, called IntelliJ Platform[10], which all Java-based IDEs from JetBrains utilize.

MPS is based on technology of projectional editor. A projectional editor directly manipulates with abstract syntax tree (AST). An MPS language is called projectional language. Each projectional language has three main parts: language structure, a projectional editor, and a generator. Language structure defines the structure of AST, the editor defines language syntax and how can users edit the AST, and the generator defines how is the created code of DSL transformed to some textual output or another projectional language. Details are described in the following sections.

### 3.1 Projectional Editor

Traditional languages environment uses a lexer and a parser to convert input source code in plain text form to AST. The lexer gets text as a sequence of characters and produces sequence of tokens – words used in the language such as keywords and identifiers. The parser then reads the sequence of tokens and creates AST.

A projectional editor does not work with text, but it creates and modifies AST. AST is projected into projectional editor in the way that it looks like textual source code. A user can write only things that are valid in AST. Projectional editor provides intelligent code completion out of the box and MPS contains various mechanisms to allow edit the language like in common text editor.

Figure 3.1 shows MPS IDE. There is a *Logical View* panel on the left side. Under the logical view, there is a window that shows messages from the IDE. On the right side, there is a projectional editor. The editor has two parts. There is the main editor in the top, and an *Inspector* panel below it. The Inspector panel is used to display and edit additional details of MPS language constructs.



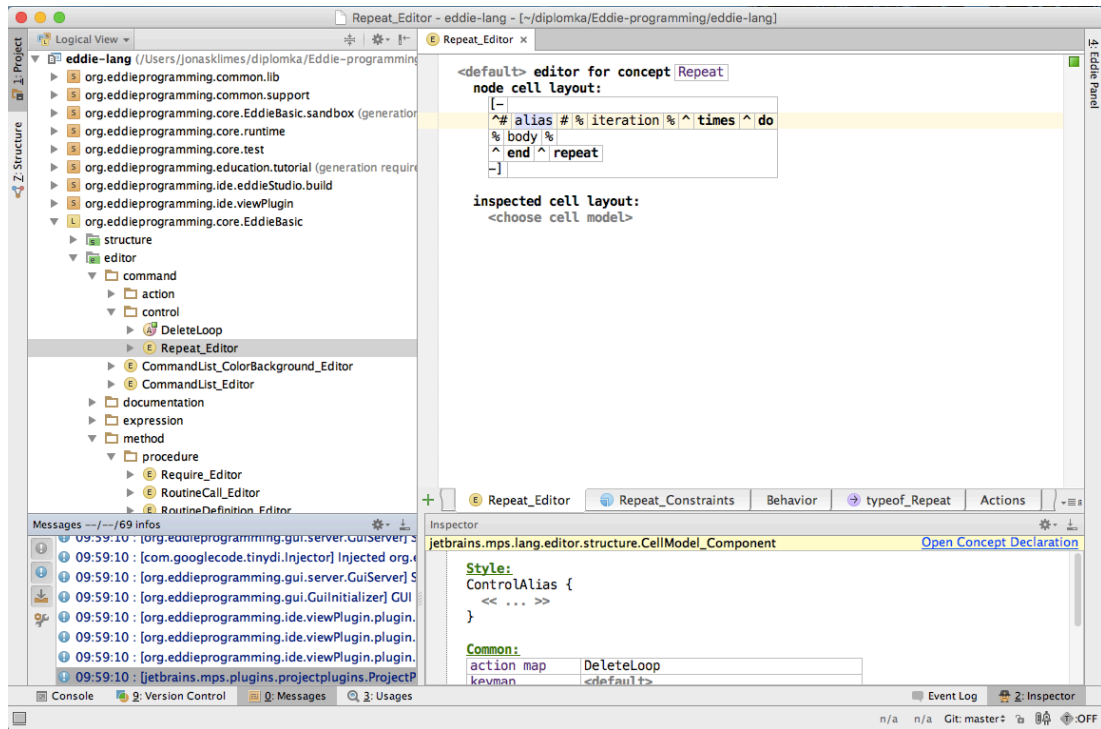


Figure 3.1 A screenshot from MPS IDE

## 3.2 Language Design

When users want to create a projectional language they must create a new MPS project. MPS project consists of *modules*. There are three types of modules: *language*, *solution*, and *DevKit*. The language module contains language definition. The solution module can be either a run-time module or a sandbox module. Run-time solutions are sometimes called libraries and contain arbitrary code written in some MPS language or stubs that allow using some Java classes or libraries. Sandbox solutions contain user programs. DevKit modules allow packing more languages and solutions together so they are easier to use. Modules may depend on other modules and use their code. A language can extend other language.

Figure 3.2 shows MPS project in IDE. There is the Logical View panel on the left which contains modules; language modules have yellow icon with letter 'L', solution modules have orange icon with letter 'S', and DevKits have blue icon with letter 'D'. Modules consists of *models*. Language models are called *aspects*. There is a set of aspects in MPS and every aspect has different role in language definition. In the figure, there are following aspects of org.eddieprogramming.core.EddieBasic language: Structure, Editor, Actions, Constraints, Behavior, Typesystem, Intentions, DataFlow, and Generator. In the following subsection, the main aspects are

explained on an example and then there is detailed description in the following sections.

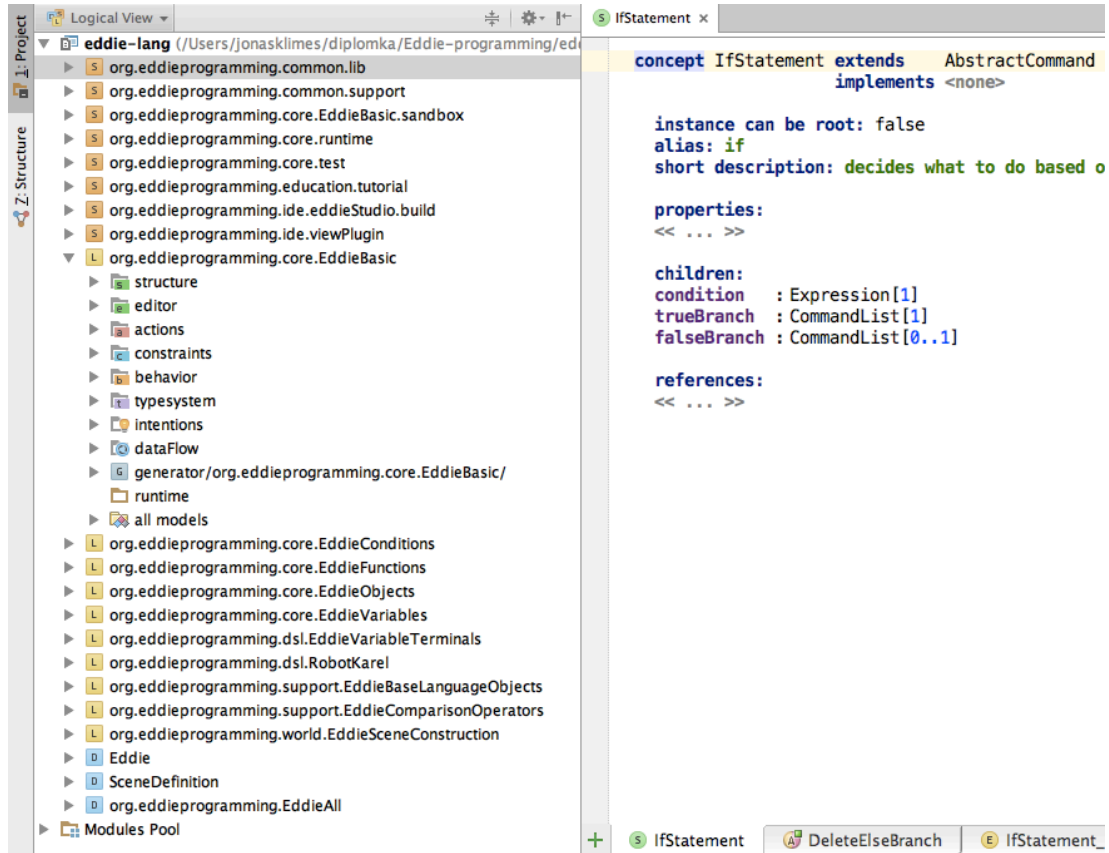


Figure 3.2 A screenshot of an MPS project in the MPS IDE

### 3.2.1 Language Design Example

This subsection illustrates the main aspects of a projectional language design using an example of definition `if-then-else` statement. The main parts are: Structure aspect, Editor aspect, and Generator or TextGen aspect. At first, a *Concept* must be defined in Structure aspect. On the right side of Figure 3.2, there is a definition of the concept. AST of a program is built from *nodes* and each node is an instance of some concept. The `if-then-else` concept must contain an expression that is evaluated to either `true` or `false`, then it has a true branch, and a false branch that are represented as a list of statements. The syntax of the `if-then-else` concept is defined in Editor aspect.

Projectional languages code is usually transformed to either text (e.g. XML, Java) or to some other projectional language. Projectional language transformation to other projectional language is done by Generator aspect and it offers more options than transformation into textual language, which is done by TextGen aspect. For the

purpose of transformation into a projectional language, MPS provides *BaseLanguage*, which is a clone of Java, and it is finally generated into textual Java code. In this example, Generator aspect is used to transform the language into BaseLanguage. There must be defined a template for the concept. The template is written in the output language – BaseLanguage and defines that the content of condition, true branch, and false branch are transformed into their equivalents in BaseLanguage.

## 3.3 Language Aspects

As was mentioned above, language definition in MPS is divided into several models called aspects. The previous section introduced three main aspects: Structure, Editor, and Generator. This chapter describes these aspects and some other language aspects which are important for understanding design of our language Eddie.

### 3.3.1 Structure

Structure aspect defines which nodes can be part of AST and how they can be structured. Structure aspect has 5 concepts:

- Concept
- Interface concept
- Enum Data Type
- Constrained Data Type
- Primitive Data Type Declaration

Concepts define base build blocks of the language. Instances of concepts are nodes in AST. Figure 3.3 shows a definition of *if-then-else* statement concept. A concept can have *properties*, *children*, and *references*. Properties contain simple literals, children define concepts of child nodes in AST, and references point to some existing nodes in language AST. Concept has an *alias* and *short description*. Alias is used to refer to the concept in an editor. Both alias and short description are used as labels in a code completion menu. Concepts have inheritance hierarchy. A concept can extend another single concept and can implement multiple interface concepts. A concept can be marked as abstract; abstract concepts cannot be instantiated. There is a special sort of concepts called *root concepts*. Only instances of root concepts can be created as a new node in the Logical View.

```

concept IfStatement extends AbstractCommand
  implements <none>

instance can be root: false
alias: if
short description: decides what to do based on logical condition

properties:
  << ... >>

children:
  condition : Expression[1]
  trueBranch : CommandList[1]
  falseBranch : CommandList[0..1]

references:
  << ... >>

```

Figure 3.3 Definition of if-then-else statement concept

Interface concepts represent independent traits. They define some functionality that can be reused in more concepts and one concept can implement multiple interface concepts. Same as concepts, they can have properties, children, and references.

The other three concepts *Enum Data Type*, *Primitive Data Type Declaration*, and *Constrained Data Type* allow defining data type for literals as enumeration type, string-based type constrained with some regular expression, or as a new primitive type. Defined types can be used as types of properties.

### 3.3.2 Editor

Editor aspect defines how nodes of concepts defined in Structure aspect are displayed to user and how they can be modified. Each concept can have one or more editors. An editor for concept consists of *Editor Cells*. Main cell types are layout collection, constant, concept property, concept child node, and concept reference. Each cell can have visual style. Styles can be defined in Editor aspect node *Stylesheet*.

```

if wall ahead then
  turn left
else
  step
end if

```

Figure 3.4 Example of if-then-else statement in an editor

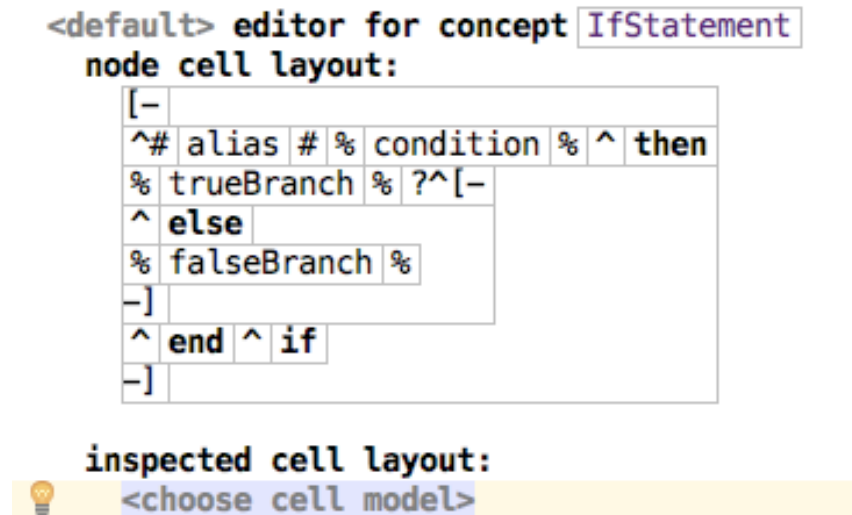


Figure 3.5 An editor definition for if-then-else statement

Figure 3.4 shows how the node of IfStatement concept looks like in a projective editor. Figure 3.5 shows definition of node cell layout for IfStatement concept. Layout definition starts with an indent layout (symbols ‘[-’), which allows to define editor as several Editor Cells. Second line starts with a Property cell that inserts a value of property alias, which is defined in the Concept definition. Next editor cell is Child Cell for the child node condition. It inserts an editor cells defined for the child concept. The last cell in this line is a Constant Cell and it displays constant text ‘then’.

Other editor nodes allow defining actions that can manipulate with AST or editor when some events are triggered. The events are for example node deletion, pasting node from clipboard, or pressing key. It is possible to define more editors for one concept and then switch between these editors in IDE.

### 3.3.3 Behavior

In Behavior aspect there can be defined a Behavior node for each Concept or Interface Concept. Behaviour node contains node constructor, and instance and static methods. Static methods can be called through Concept. Instance methods can be called on nodes that are created as instance of Concepts. Methods can be marked as virtual and then be overridden. Methods in Behavior node of abstract Concept can be abstract. Figure 3.6 An shows an example of a Behavior node that defines methods for common descendants of unary logical operators. It overrides two methods defined in interface concept *ITypeAnnotable*. The second method is defined as static.

```

concept behavior UnaryLogicalOperator {
    constructor {
        <no statements>
    }

    public node<Type> getTypeAnnotation()
        overrides ITypeAnnotable.getTypeAnnotation {
        return getStaticType();
    }

    public static node<Type> getStaticType()
        overrides ITypeAnnotable.getStaticType {
        return concept/BooleanType/.getInstance();
    }
}

```



Figure 3.6 An example of Behavior node

### 3.3.4 Constraints

Constraints aspect can introduce some constraints on concept child nodes, properties, and references. In structure aspect there is defined type of each child or reference node. Constraint aspect allows creating some more complex checks based on current context.

```

concepts constraints IfStatement {
    can be child <none>

    can be parent
        (childConcept, node, childNode, operationContext, link)->boolean {
        return ConstraintsUtil.checkStaticType(childConcept.asConcept, TypeUtil.BOOLEAN, link,
        linkNode/IfStatement : condition());
        }

    can be ancestor <none>

    <<property constraints>>

    <<referent constraints>>

    default scope
    |<no default scope>
}

```

Figure 3.7 An example of constraint aspect node

Figure 3.7 shows constraints defined for if-then-else statement concept. A child of a concept can be restricted by methods can be child, can be parent, and can be ancestor. The methods get a node or a concept as a parameter and decide, whether the given node or the node of the given concept can be a child node, a parent node or an ancestor of the node for which concept is the constraint defined.

The method `can be parent` in the example ensures that a condition expression is of Boolean type. Values that do not qualified are not suggested in the code completion menu and when they are already in code, they are marked as errors.

Properties of node can be validated before they are set and they can have custom getter and setter.

By default, a target of reference can be any node of particular type that is in current model or in the dependencies of current model. This is not always desired, so a referent constraint can define to limit scope for references. That is useful for example to define local variables.

### 3.3.5 Actions

Actions aspect helps to improve editor usability. It allows defining node transformations. The main nodes are: *Node Factories*, *Transform Menu Actions*, and *Node Substitution Actions*. Node Factories defines how a new instance of node is created based on enclosing node or when it is created to replace some other node. Figure 3.8 shows Node Factories definition for if-then-else statement. When an if-then-else statement is created to replace a while loop, a condition and a body of the while loop are preserved as a condition and a true branch of the if-then-else statement.

```
node factories ConditionsNodeFactories

node concept: IfStatement
  description : change while to if
  set-up      : (newNode, sampleNode, enclosingNode, model)->void {
    ifInstanceOf (sampleNode is While whileCmd) {
      newNode.condition.set(whileCmd.condition);
      newNode.trueBranch.set(whileCmd.body);
    }
  }
```

Figure 3.8 An example of a Node Factories node

Transform Menu Actions transform an AST node when a user creates a new node next to the first node or just type some text that matches some pattern. Figure 3.9 describes an action to create an else branch for an if-then-else statement when a word 'else' is written in the editor. There is a condition, when can be the transformation applied. In this case, it can be applied only if else branch is not created yet.

```

side transform actions IfElse

right transformed node: IfStatement tag: ext_1 // after then or afeter the end
condition :
    (operationContext, model, sourceNode)->boolean {
        sourceNode.falseBranch.isNull;
    }

common initializer :
    <no common variables>
    .....
    <no common initializer>

actions :
    add custom items (output concept: AbstractCommand)
    simple item
        matching text
            else
        description text
            <default>
        icon node
            <default>
        type
            <default>
        do transform
            (model, operationContext, sourceNode, pattern, editorContext)->node<> {
                sourceNode.falseBranch.set new initialized(CommandList);
            }

```

*Figure 3.9 Example of Transform Menu Action*

```

node substitute actions localVariableDeclarationWithType

substituted node: AbstractCommand // allows to create local variable declaration by writing type
condition :
    <none>

common initializer :
    <no common variables>
    .....
    <no supplemental initializer>

actions :
    add custom items (output concept: LocalVariableDeclarationCommand)
    wrap Type
        wrapper block
            (nodeToWrap, parentNode, currentTargetNode, childConcept, model, operationContext, editorContext)
            node<LocalVariableDeclarationCommand> declarationCommand =
                nodeToWrap.model.new initialized node(LocalVariableDeclarationCommand , );
            node<LocalVariableDeclaration> declaration = nodeToWrap.model.
                new initialized node(LocalVariableDeclaration , );
            declaration.type.set(nodeToWrap);
            declaration.setDefaultInitiazer();
            declarationCommand.declaration.set(declaration);
            return declarationCommand;
        }
    return small part
    <false>
    selection handler
        <default>

```

*Figure 3.10 An example of a Node Substitution Action*

Node Substitute Action allows substituting a node or wrapping a node with some other node. When MPS creates items for code completion menu, it searches also for these Node Substitute Actions. Figure 3.10 shows an example of a Node Substitute



Action that wraps a node that represents a variable type in a local variable declaration node, so for example when a user writes variable type `int` in the editor, MPS creates a node that declares a local variable of `int` type with a default initializer and with an editor cursor at the place where a variable name can be written.

### 3.3.6 Typesystem

Typesystem aspect allows to compute types and to check them. The main concepts are: *Inference Rule* that allows to define types for concepts, *Subtyping Rule* to define a type hierarchy, and *Non-Typesystem Rule* to define a custom validation for any node.

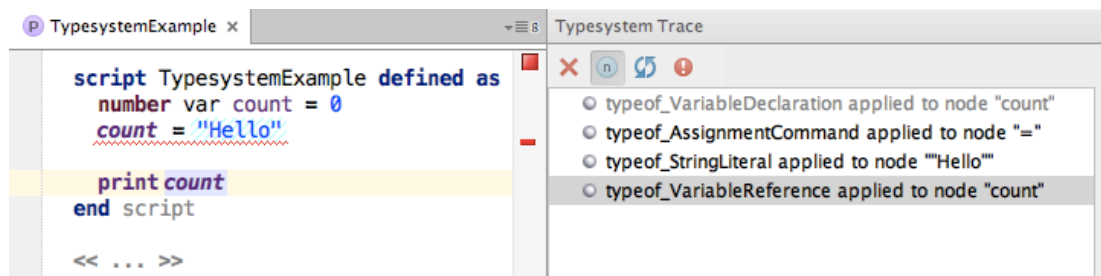


Figure 3.11 An example of typesystem evaluation

Figure 3.11 shows an example of typesystem evaluation. On the left side, there is a code that declares numeric local variable `count` and tries to assign a textual value “Hello” into the variable. This is not allowed, so the assignment results in an error, which is displayed as a red underline. On the right side, there is a sequence of applied typesystem rules to infer and check whether the variable `count` can be assigned with the value “Hello”. Firstly, the rule `typeof_VariableDeclaration` is applied and it evaluates that the variable `count` is declared as the number type. The variable initialization is not important in this situation, because the language is typed statically. Secondly, the rule `typeof_AssignmentCommand` is applied to assignment operator and it checks whether a type of the value (right operand) is a subtype of the variable (left operand). These types have not been known yet, so the evaluation continues. The value “Hello” is a text literal and the rule `typeof_StringLiteral` defines its type as a text. Then a type of the variable reference `count` is evaluated as a number, because the rule `typeof_VariableReference` infers a type from the declaration of the variable and the type of this variable was evaluated in the first step as a number type. Finally, the rule `typeof_AssignmentCommand` has to check whether the text type is a subtype of the number type. Because there are not any

Subtyping Rules that would imply that the text type is a subtype of the number type, this assignment is marked as an error.

Non-Typesystem Rules can check any conditions and set custom info, warning, or error message. They can be used for instance to show an error message if some names are not unique, or an info message if names does not follow some naming convention. Non-Typesystem Rules are also used to display error, warning, and info messages that are found during a static analysis that is executed using Dataflow aspect. The static analysis is described in the next section.

### 3.3.7 Dataflow

Dataflow aspect defines rules for a static analysis. For each concept, a language execution flow can be defined using jumps, conditional jumps, variable reading and writing, and evaluation of dataflow of child nodes.

Figure 3.12 shows an example of dataflow definition for if-then-else statement. The body of the *data flow builder* is separated by comments into three parts: condition, true branch, and false branch. The condition part starts with a statement `code for` to execute dataflow recursively at the node representing the condition expression. It cannot be determined, whether the condition is fulfilled or not, but it should be processed by dataflow, because it might read variables or call functions. On the next line, there is an `ifjump` statement. It defines that the execution may or may not jump after the place in code that is marked with the label `endOfTrue`. Jumping represents the situation that condition is not fulfilled.

```
data flow builder for IfStatement {  
  (node)->void {  
    // condition  
    code for node.condition  
    ifjump after label endOfTrue  
  
    // true branch  
    code for node.trueBranch  
    { jump after node }  
    label endOfTrue  
  
    // false branch  
    if (node.falseBranch.isNotNull) {  
      code for node.falseBranch  
    }  
  }  
}
```

Figure 3.12 Example of dataflow builder

Next part, the true branch, evaluates code for the code block in `trueBrach` of `if-then-else` statement. Statement `jump` represents unconditional jump, after evaluation of true brach program should jump after the `if-then-else` node. The third part, false branch, is executed only if `falseBrach` is present in the `if-then-else` statement. If it is present, it evaluates dataflow for the `falseBrach` code block.

When dataflow is defined for a language, MPS provides means to create and evaluate a dataflow program for some node using defined data flow builders. Figure 3.13 shows an example of a dataflow program. On the left side, there is a method that counts the greatest common divisor for two arguments. On the right side, there is a fragment of a dataflow program created for this method. Created dataflow program allows checking variables, unreachable code, or whether a function return value from all of its execution paths. Checking of variables includes a check whether declared variables are used, whether assignments to the variable are read, or whether variables are initialized before reading from them.

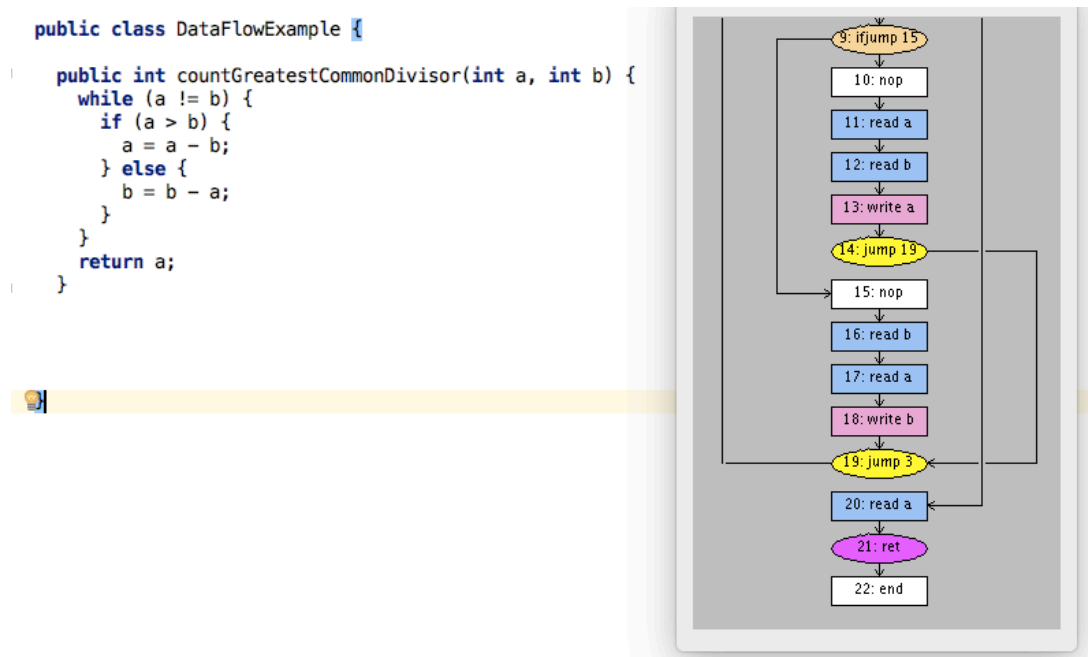
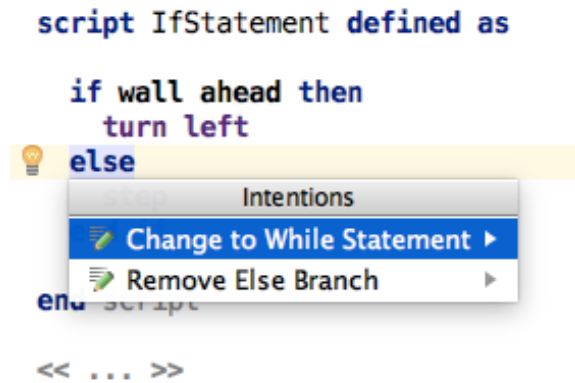


Figure 3.13 An example of dataflow program

### 3.3.8 Intentions

Intentions aspect is concept used in IntelliJ IDEA platform. It is contextual menu invoked by specific keyboard shortcut with some actions connected to the current cursor position in the code. In MPS, it allows to create custom AST node

transformations. Figure 3.14 shows an example of intentions for `if-then-else` statement. The intentions can change the statement to a `while` loop with keeping the condition and body of true branch or they can delete false branch. An intention can be also defined to fix some error.



*Figure 3.14 An example of intentions menu*

### 3.3.9 Generator

Generator aspect allows transforming code from defined language to some other projectional language, typically into `BaseLanguage` – a projectional clone of Java. Generation is done using templates and mapping configuration.

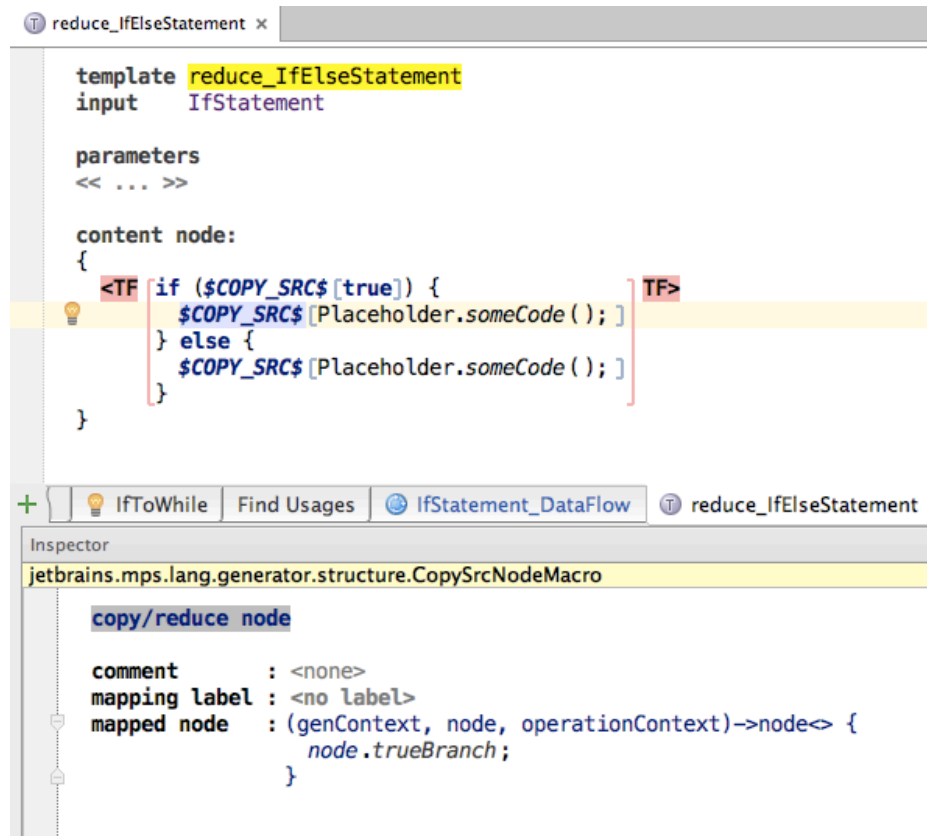


Figure 3.15 Example of generator template

A template is a root node or code fragment written in output language. A template receives a node of input language as a parameter. The parameter influences the output code using macros. Figure 3.15 shows an example of a generator template. In the upper part, there is a template to transform if-then-else statement with else branch. The template code is written in BaseLanguage in the *content node* section. There is a block statement represented by curly braces and inside them, there is an if-then-else statement. If-then-else statement is surrounded with a *template fragment* – represented as '<TF' and 'TF>' symbols and red brackets. Only the code inside the template fragment is included into the output.

The condition in the if-then-else statement is true and code of both branches is Placeholder.someCode(). They serve only as placeholders and there could be any code as long as it is valid. All the placeholders are surrounded with COPY\_SRC macros. In the lower part of the picture, there is a detail of the COPY\_SRC macro in the true branch. The macro defines that placeholder code should be replaced with the output code of node.trueBranch.

Besides the `COPY_SRC` macro, there are for example a `LOOP` macro to iterate over collection of nodes, an `IF` macro to decide between two alternatives, a `SWITCH` macro to decide between different templates, a property macro to replace placeholders with values of properties of nodes, or a reference macro to create references between nodes in output language.

In order to generate an output, there must be also defined, how are templates applied to an input AST. This is defined in a *Mapping configuration* that maps input language concepts to defined templates. Mapping rules can contain conditions, so there can be more templates for one concept and the selection of the right template is based on values of the node.

### 3.3.10 TextGen

TextGen aspect serves to transform language AST into a text. If the output text is source code in some language, it can be then compiled or interpreted with standard tools. Figure 3.16 shows an example from BaseLanguage TextGen that transforms BaseLanguage into Java source code. A TextGen template for a node can apply templates for its child nodes and it has also means to maintain indentation.

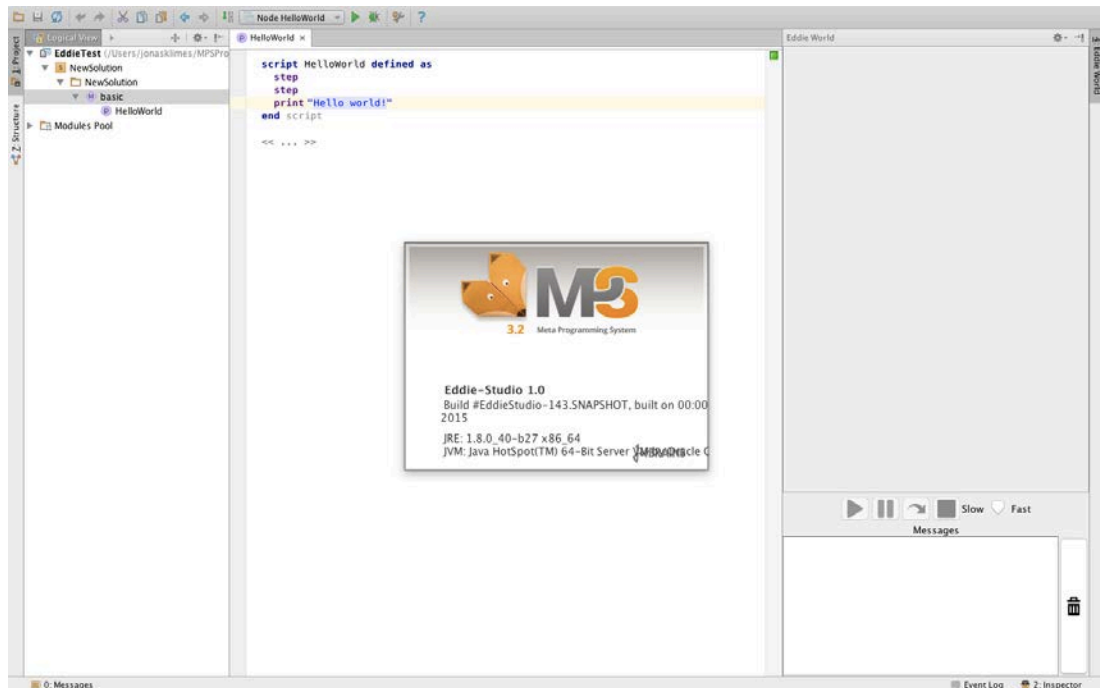
```
text gen component for concept IfStatement {
  (context, buffer, node)->void {
    append \n;
    indent buffer;
    append {if (} ${node.condition} {) {}};
    with indent {
      append ${node.ifTrue};
    }
    append \n {} $list{node.elseifClauses};
    if (node.ifFalseStatement.isNotNull) {
      append { else} ${node.ifFalseStatement};
    }
  }
}
```

Figure 3.16 An example of TextGen from BaseLanguage

## 3.4 Integrated Development Environment

Source code of DSLs designed in MPS can be build and distributed as a language plugin. Languages can be built as a plugin for MPS, a plugin for IntelliJ IDEA, or a Standalone IDE. The plugin for MPS is the simplest solution how to use language. Languages can be imported into other MPS projects and they can be used there. A language that is packed as the plugin for IntelliJ IDEA can be used in IntelliJ IDEA

projects. It requires installing MPS plugin for IntelliJ IDEA. The last option is to create a standalone MPS distribution customized to edit particular languages.



*Figure 3.17 A screenshot of custom MPS IDE*

It is also possible to create custom windows, called *Tools*, for MPS or custom MPS IDE. Tools can provide any functionality GUI component implemented in Java using Swing. These custom Tools can be packed and distributed together with MPS languages in a custom MPS distribution. Figure 3.17 is a screenshot from a custom MPS IDE. On the right side of the screen, there is a custom Tool to control and display execution of a program. MPS is available for Windows, OS X, and Linux and custom IDE can be also created for all of these platforms.

### 3.5 Advantages

The first advantage of MPS is that projectional languages have defined editors with rigid syntax, which may guide the novice programmers. An intelligent code completion can be sophisticated and larger parts of code can be inserted automatically, so syntax can be more explanatory without the need of writing more. Users do not need to take care about inserting special characters such as parenthesis or semicolons. Editor has also pre-defined layout, which takes care about spacing and indentation.

Syntax highlighting can reflect also relations between nodes, not only their syntax. Additional text can be added to the editor and it can vary as user is changing the code, e.g. when some node is missing, a textual hint can be added there to help user to correct it, once the code is correct, the hint disappears. Projectional code does not have to be parsed, so language syntax can have much more options and languages can be freely combined together. There is also support for visual elements such as tables or diagrams.

The fact that projectional editor manipulates directly with AST allows to implement some editor features that are common in mature IDEs with much less effort. MPS provides an intelligent code completion out of the box. Thanks to typesystem and dataflow aspects, it is easier to implement some checks, so some errors can be reported immediately when they are made and it is harder to write invalid code.

## 3.6 Limitations

Most of the limitations come from the fact that MPS language code is not plain text, but each single MPS model, which can contain more root nodes, is stored in a single XML document. Projectional language code can be read and write only in MPS, in IntelliJ IDEA with *MPS plugin*, or in custom MPS IDE. The language must be also packed as a *language plugin* and imported into the IDE. Code from MPS editor can be copied as text, but when it is necessary to insert plain text code back into MPS editor, it requires extra effort to implement a parser for the language.

It is also impossible to do version control without the *MPS VCS plugin*, which understands AST XML representation. A programmer cannot do manual merge and even small changes that does not influence code appearance in editor, can broke some references in AST and make the code invalid. Therefore, a change in implementation can break existing code. In order to allow language evolution MPS provides *Migration aspect* to transform code between different versions of language.

MPS developers plan to provide MPS as web application in futures versions, but it is not certain that MPS will develop in this direction and it could take years. This prevents us from providing the language online with a customized and simplified user interface.



# 4 Language Design

This chapter describes our educational DSL, which we call *Eddie*. We decided to choose this name because it is easy to remember, easy to spell, and it evoke education. This chapter starts with selecting key requirements for the language, then it describes a chosen domain and runtime environment. The following sections describe the language and its syntax. The last section describes how tutorials can be defined.

We believe that in the field of educational programming tools for small children, there are good tools such as *Code.org* or *Scratch*. This tools use graphical code representation. Although MPS supports tables and diagrams, it is mainly designed to create textual languages. Therefore, *Eddie* is strictly textual and we believe that it can help users to switch from *Eddie* to some standard programming language, after they gain some programming skills.

*Eddie* targets to adults, teenagers, and their school teachers, because they are able to use more complex IDE and they may prefer textual language to graphical code blocks. We also believe that some tools for this target group, such as *BlueJ* or *Greenfoot*, have some limitations that *Eddie* could deal with.

## 4.1 Key Requirements

This section defines some key requirements that we want for *Eddie*. It is based on the analysis an discussion of existing languages from Chapter 2. There are also some requirements that we figured out during the language design.

We believe *Eddie* should have these features: well known domain, constructs organized into groups, gradual concept introduction, intelligent code completion, immediate error checking, visible code execution, code representation according to skill level, easy switching to standard programming languages (items F4-F9 and F12-F13 in the list in Section 2.3).

We also believe that tutorials are essential for programming education. More about tutorials is in the following parts. Section 4.6 describes how the tutorials can be defined and Chapter 6 introduces examples of tutorials themselves.

We do not focus on increasing people's motivation. We think that *Code.org* with their videos, campaigns, and popular themes did a great job that we definitively cannot compete.

We believe that accessibility is a key requirement for good educational language and that the best way how to achieve it, is to provide it as a web application. Unfortunately, current version of MPS does not allow deploying projectional editor as a web application.

*Eddie* is intended just for education, so its syntax does not have to be optimized to write code fast, but it should be clear and self-explanatory. Therefore, it should avoid convention over configuration principles, there should not be any default behaviour that is not clear from the code, and it should prefer words to parentheses and other symbols. The only exceptions from this are standard math symbols. In spite of similar syntax with common language is one of the key features that allows users to switch from educational language to common languages, we prefer to have self-explanatory syntax.

Most of the non-textual educational systems lack of an undo operation, which allows users to quickly and safely revert their actions. Undo operation is common in the most of the textual editors and MPS-based systems support this operation too, so it is also in *Eddie*. MPS IDE also allows using standard means of version control, e.g. *Git* or *Subversion*, for all MPS languages including *Eddie*. This gives users the ability to manage gradual evolution of their code base and guarantee safe rollback.

## 4.2 Domain

Before an educational DSL can be designed, a domain must be chosen. We did not find any other suitable domain except the two that are used in the selected existing languages. They are the following: controlling some character, and 2D drawing. They can be also combined to control some character that draws as it moves, then its path can draw some shapes.

We decided to choose the domain of the one of the existing languages – *Karel the Robot*. This domain and its language allow programmers to control a robot. It is described in Section 2.1.3. We chose it because it is tightly connected to the world of programming, as was discussed in Section 2.2. We believe that controlling robot can

be interesting for both children and adults. Some programmers and IT teachers are also familiar with this domain, which can help Eddie to get their attention.

We call a code unit of Eddie that can be executed as an Eddie program or program. The two-dimensional grid, which the robot inhabits, we call Eddie map or map. Constructs that manipulates with the robot are called actions. Eddie contains a set of predefined actions and users can define their own custom actions.

Besides the domain Karel the Robot, we designed also a domain *Computer Terminals*, which allows to place computer terminals in the map and use them to store some information. We added computer terminals, because we think that Karel the Robot domain does not provide sufficient means to practise manipulation with variables. For example, computer terminals can be used in a scenario where a user needs to read some secret value from one terminal into a variable and then write the secret into another terminal to complete the task.

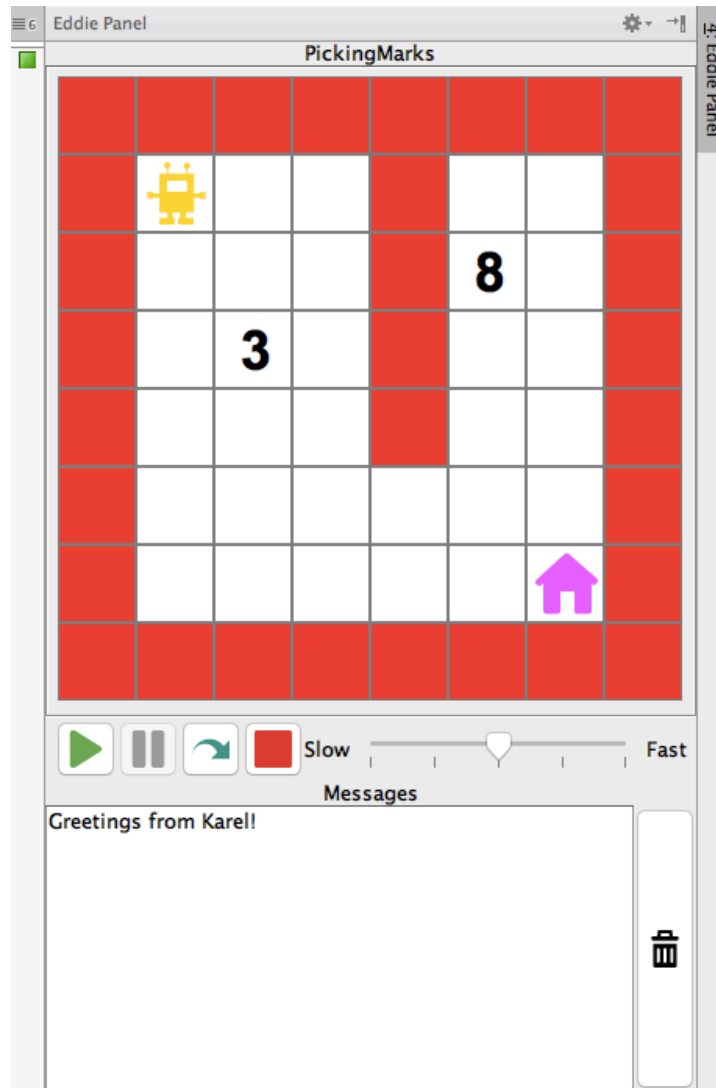
## 4.3 Environment

Eddie is distributed within its own IDE called *Eddie Studio*. Eddie Studio allows creating, modifying, and running Eddie programs. One of the key features for educational language is that code output is visible for users. Program execution is visualised in one window of Eddie Studio, called *Eddie Panel*. Inside the panel, there is an Eddie map.

The Eddie map consists of cells and each cell can contain some objects. A cell without any object is called an *empty cell*. Objects also can have labels, which are displayed as tooltip texts. The objects are:

- *Robot*: an object which is controlled by the program
- *Wall*: the only place that robot cannot visit.
- *Marks*: a cell with one to eight marks. Robot can pick existing marks or put marks until the limit of eight marks is reached.
- *Home*: an object that can be used in tutorials as a target destination for some robot task.
- *Computer Terminal*: an object that can be used to store some value. Each terminal has a variable type and a value of that type. Each type has different colour and both type and values are displayed in the label of the Terminal.

We decided always to display current value in order to make debugging of programs easier.



*Figure 4.1 A screenshot of Eddie Panel*

Figure 4.1 shows Eddie Panel, upper part contains visualisation of Eddie map. Eddie map is displayed as two-dimensional grid with yellow robot, white empty cells, red walls, black numbers, which represents marks, and a violet house of the robot. Below the map, there is a control panel. Panel contains buttons to run the program, to pause the running program, to do just a single step, and to cancel the execution of the program. It is also possible to choose one of the five execution speeds. Under the controls, there is a space where it is possible to print messages from the program. A button with dustbin allows deleting all of the printed messages. Panel can also display a popup window with some message. When a popup window opens, the program execution is suspended until a user closes the window. Popup window can

be invoked from Eddie program and it is also used to display Eddie runtime errors. *Eddie runtime errors* are errors caused by users, such as when the robot hits a wall.

## 4.4 Language Constructs

Our language Eddie has a set of language constructs. One language construct is created by one or more MPS concepts. Eddie's constructs can be divided into the following groups:

- Roots
- Actions
- Control statements
- Expressions (including functions and variables)
- Documentation

For easier understanding, the basic parts of Eddie without object-oriented constructs are described first and they are organized into these groups. All the object related construct are described at the end as one group, although they could fit also into these groups. This section does explain neither names that are used for constructs nor their syntax. That is described in Section 4.5.

### 4.4.1 Roots

Every code must be written in some kind of document. An instance of root concept represents a single file in Logical View in MPS. Eddie contains two root concepts: Program and Library. The Program concept is an entry point for every Eddie program and it can be directly executed. When a user runs the program, it is transformed into a Java class with a `main` method and it is started as a standalone Java application, which then communicates with Eddie Panel.

Figure 4.2 shows an example of program. There is a body of the program and definitions of a custom action and a function. The body is a list of statements, which is executed after a user runs the program. A program can contain arbitrary number of custom actions or functions.

The Library concept serves to define custom actions or functions. Libraries can be imported into programs using `require` statement. `Require` statement can be used only directly in the body of a program. When a library is imported, any of its custom

actions or functions can be used in the program. Eddie does not allow importing a library into another library in order to avoid potential cyclic dependencies.

```
program ProgramExample defined as
  step
  turn left
end script

action turn around defined as
  turn left
  turn left
end turn around

function say
  with parameters:
    text message
  with return type none
  defined as
    print "Karel: " + message
end say
```

*Figure 4.2 An example of a program*

#### 4.4.2 Actions

As is mentioned in Section 4.2, Eddie contains two types of actions – actions defined by the language and custom actions defined by users. Figure 4.3 shows a program with all the actions. There are basic actions to control robot Karel: `step`, `turn left`, `pick mark`, and `put mark`, and there are also actions `print` and `alert` that serves to post some message into Eddie Panel. The `print` action writes a message into the area for messages in Eddie Panel. The `alert` action displays a popup window with the message and the execution of the program is suspended until a user closes the popup window.

At the bottom of the picture, there is a custom action `turn right`, which is defined as turning left three times. Custom actions can be defined in a program or a library. Each custom action has a name and a body. Its name is used as a reference to it in the place where it is supposed to be invoked. The name must be unique in the program or the library where it is defined. If there are two actions with the same name in a program and in an imported library, projectional editor offers both of them and a user selects the reference, which she wants to use. The body of custom action is a standard list of statements.

```

program Actions defined as
  step
  turn left

  pick mark
  put mark

  print "Hello world in messages."
  alert "Message in popup window."
end script

action turn right defined as
  turn left
  turn left
  turn left
end turn right

```

*Figure 4.3 An example of predefined actions and a custom action*

### 4.4.3 Control Statements

We designed four control statements in Eddie: repeat times loop, for loop, if-then-else statement, and while loop. Figure 4.4 shows a code of repeat loop. It consists of a list of statements that should be executed and a number that specifies the count of repetitions. The count can be any expression of integral numeric type.

Figure 4.5 shows a Pascal variant of for loop. It has an iteration variable, start index, end index, and a body. It can have optional keyword down before the word to, which means that iteration variable is being decremented instead of incremented.

```

repeat 5 times do
  ...
end repeat

```

*Figure 4.4 Repeat times loop*

```

for number val i = 1 to 5 do
  ...
end for

```

*Figure 4.5 For loop*

When we designed Eddie, we were thinking, what kind of fixed iterations loop should we used. We considered a C language variant of for loop with the structure for (initialization; condition; increment/decrement), but we decided to use simple repeat times as probably the simplest control statement. Then we decided to introduce also the Pascal variant of for loop, because users might need to work with the iteration variable. We did not introduce

the C variant of `for loop`, because it is too complicated, and it allows writing code that is hard to understand.

Figure 4.6 shows a conditional `if-then-else` statement, it has three parts: condition, true branch, and false branch. The false branch is optional. Condition can be any expression of Boolean type. True and false branches are lists of statements. If the condition is evaluated as truth, the true branch is executed. If the condition is not truth and there is false branch, the false branch is executed.

Figure 4.7 shows a `while loop`, it contains a condition and a body. While the condition is satisfied, the body is executed. The condition is evaluated before the first loop and after every iteration. We preferred `while loop` to `do while` and `repeat until`, because the construct with condition at the beginning is more common in nowadays standard languages.

```
if wall ahead then
  turn left
else
  step
end if
```

*Figure 4.6 If-then-else statement*

```
while mark do
  pick mark
end while
```

*Figure 4.7 While loop*

#### 4.4.4 Expressions

Next group of language constructs are expressions. This group contains literals, operators, variables, functions, and map queries. Expressions have types. We decided to use statically typed variables, because static typing meets our requirement that the language does not contain any implicit behaviour that users cannot see in the code. Eddie has three basic types: text, number, and logical. Text is a sequence of characters, number is a signed integer, and logical is a Boolean value. We tried to use names that are understandable to beginners in programming world. We decided not to have `char` type as a representation of a single character and `string` type as a sequence of zero or more characters. We believe that `text` type can cover all users' needs at this point. We designed also `void` type that is used only as a return type of functions that does not return any value. Eddie uses name `none` instead of `void`.



## Literals

The simplest expressions are literals. Eddie contains integer literals, logical values `true` and `false`, and text literals, which can be any character sequence enclosed in quotation marks.

## Operators

Next group of expressions are operators. Eddie has logical operators, arithmetic operators, and text concatenation operator. Logical operations are negation, conjunction, and disjunction. Arithmetic operations are addition, subtraction, multiplication, and integer division. Text concatenation operator can be used also to concatenate text with number or logical.

We used simplified implementation for expressions. As a result, Eddie does not contain any parenthesis in expressions and comparison operators. Details of this implementation and reasons for this decision it are described in Section 5.3.4.

## Variables

Modern programming languages such as Scala prefer to have variables immutable. Older languages like Java or C# also allows declaring immutable variables, but default behaviour is mutable and many programmers do not pay attention to it and leave them mutable. We decided to introduce mutability in Eddie, because we believe that it is good practice to use mutable variables only where it is needed.

```
text val name = "Karel"  
number var count = 0  
logical var isWall = wall ahead  
  
print name
```

*Figure 4.8 Local variable declarations and a variable reference*

Figure 4.8 shows three local variable declarations and a variable reference in a `print` action. Variable declarations consist of a type, a keyword `var` or `val`, a name, and an initialization. A variable declared as `val` is immutable and its value cannot be modified. `val` is the default option. A variable declared as `var` is mutable and its value can be changed. If there is not explicit initialization, the types `number`, `logical`, and `text` are initialized with number zero, `false`, resp. empty text. Variables are referenced by their names. Variables are defined as local; they are visible in the

current list of statements from the point of declaration to the end of the list including nested list of statements. It is not allowed to have two declared variables with the same name in one place, so it is not possible to overlap a local variable from an ancestor list of statements with a new declaration with same name. A local variable must be always initialized. Besides local variables, Eddie contains also function parameters and member fields of objects, which are described later.

## Functions

Functions can be defined in the same way as custom actions, but functions are perceived as expressions, because they can return a value and they can have parameters. Figure 4.9 shows definition of the function `doSafeSteps`, which accepts one numeric parameter with name `count` and which returns a logical value. Function parameter declarations contain type and name and they are always immutable. If a function returns a value, it must contain a return statement with a value of selected type at each exit point of the function's body.

```
function doSafeSteps
  with parameters:
    number count
  with return type logical
  defined as

    repeat count times do
      if not wall ahead then
        step
      else
        # we stopped early
        return false
      end if
    end repeat

    # we finished all steps
    return true
  end doSafeSteps
```

*Figure 4.9 Definition of function `doSafeSteps`*

Functions can be called in programs and libraries in same way as custom actions. A function can be called from any place where an expression can be or it can be called as a single statement. In code completion menu, each function is displayed with a list of its parameters and with its return type. When a function call is inserted into code, its signature with names and types of parameters is prepared.

Figure 4.10 shows a program with two function calls. The first one shows a call of function `doSteps` just after it was selected in the code completion menu. There is a red text `'enter: number'`, which is displayed when a parameter value is not set. It should help to insert a value of the right type. The second call is the function `reportStatistics` with the following parameters: text `robotName`, number `pickedMarks`, and logical `showPopup`. When parameters are set, their types are hidden.

```
program Functions defined as
doSteps ( count = enter: number )
reportStatistics ( robotName = "Karel", pickedMarks = 5, showPopup = false )
end program
```

*Figure 4.10 A program with function calls*

## Map Queries

Map queries are predefined simple functions that provides information about objects in the map around the robot. They are not defined as functions, which were introduced in previous the section, but they are part of the language, so they can be introduced to a user, who does not know functions yet. Map queries returns a value of logical type.

Eddie contains robot queries that are introduced in Karel language description in Section 2.1.3. They are the following: `mark`, `full`, `wall ahead`, and `looking north`, `south`, `east`, or `west`. We designed also a query `at home`, to check whether robot is in the position where his home is in the map.

## Computer Terminals

Computer Terminals are described in Sections 4.2 and 4.3. We designed three operations with terminals. They are displayed in Figure 4.11. The first one is a map query, which checks whether there is a terminal of selected type in the position of robot and assigns this information into a logical variable `logicalAvailable`.

```
logicalAvailable = available logical in terminal
write text "Some text" to terminal
count = read number from terminal
```

*Figure 4.11 Operations with computer terminals*

Other operations can be considered as functions, but they do not follow the syntax of functions, in order that they can be introduced to users before functions. The second operation is a function that writes a text ‘Some text’ into terminal, at the position of the robot in the map. The third operation reads a number value from the current terminal and assigns it into the variable `count`. If there is neither a terminal of given type nor a terminal at all, Eddie runtime error is thrown and the program is terminated.

#### 4.4.5 Documentation

In order to create easily understandable code, it is good practice to document it. Statements in code can be separated by empty lines to increase code readability. In projectional languages, code is stored directly as AST, so empty lines must be also represented by some concept. We included the empty line concept into documentation concepts, because same as them, it is ignored when program is compiled and executed.

Eddie offers two ways how to document code: line comments and documentation blocks. Figure 4.12 shows a function `countMarks` with a documentation block before the function declaration. It can be multiline text. Documentation block can be at any program, library, custom action, or function. Users can add the documentation block using an *Intention* contextual menu. In the body of the function, there is a line comment before each of the loops. A line comment can be at any line where a statement can be written.

```

###
# Counts all marks by picking them and putting them back.
###
function countMarks
  with parameters:
    << ... >>
  with return type number
defined as
  number var count = 0

  # pick all marks and count them
  while mark do
    pick mark
    count = count + 1
  end while

  # put the same amount back
  repeat count times do
    put mark
  end repeat

  return count
end countMarks

```

*Figure 4.12 A function with a documentation block and line comments*

We believe that in Eddie should be possible to comment out any code that a user does not want to use at that time. MPS 3.3 provides means to implement it, but version that we used at the time when this was implemented, did not support commenting out.

#### 4.4.6 Objects

The object-oriented fragment of Eddie implemented in the prototype contains simple objects without inheritance. We designed two root nodes: class and singleton class. Class defines structure of the object. It can contain three types of members: a constructor, fields, and methods. Class can have only one constructor and it is always used when an instance is created. A constructor can have parameters.

Fields and methods have access modifiers, which can be either `public` or `private`. Public members can be accessed from anywhere, private members only from the node of the class. Field definitions are similar to local variable definitions, but they do not have to be explicitly initialized. Immutable fields can be also initialized in a constructor.

```

class Dog defined as
  fields:
    private text val name

  constructor:
    constructor with parameters
      text name
    defined as
      this.name = name
    end constructor

  methods:
    public method bark
      with parameters:
        << ... >>
      with return type none
    defined as
      print this.name + ": woof-woof"
    end bark

end Dog

```

*Figure 4.13 Definition of class Dog*

Method definitions are same as functions. Object members are accessed via a dot operator. Fields and methods can be accessed from their own class using `this` construct. Figure 4.13 shows an example of class definition, there is a class `Dog` with private text field `name`, constructor with parameter `name`, and public method `bark`. Unlike in common languages, class definition in Eddie has defined sections for fields, constructor, and methods and members of different kind cannot be mixed together. All classes, singletons, and class members can also have a documentation comment.

Classes introduce also a new type: a class type that represents an instance of a class. A variable of a class type is initialized using constructor. Constructor is called with keyword `create` and name of the class. If the constructor has parameters, they are set in the same way as they are set in functions or methods. Figure 4.14 shows a declaration of a local variable `dog` of type `Dog` that is initialized with an instance of class `Dog` with name 'Laika'. On the second line, there is a call of public method `bark` on the variable `dog`.

```

Dog val dog = create Dog ( name = "Laika" )
dog.bark ( )

```

*Figure 4.14 Instantiation of class and a public method call*

We decided to hide memory allocation and `null` value checking from users, because we think it is too complicated for beginners. Class type variables must be always initialized. If a user forgets to initialize some class type variable, the program ends with a runtime error. Initialization of local variables is enforced by the editor. This check is complicated in class fields, where it is not necessary to initialize field at its declaration, but it can be initialized in constructor. This check is not implemented in prototype, but it could be implemented in the future using DataFlow aspect, so it is expected that in the final implementation, these runtime errors will be avoided.

We also considered an option that instance would be created immediately when the variable is declared, but this would either disallow having class member fields of other class types or it could lead to infinite loop of initialization just by defining two or more classes with references between them and initializing one of them. This problem can still occur when a user initializes these references with new instances, but at least the cause of this problem would be explicitly in the code.

The second object-oriented root concept is singleton class. We decided to create language support for a singleton instead of a static class. Our singleton is very simple, so in some way it has same limitations as static classes. However, using singletons is nowadays preferred to using static classes or members and we wanted to go in the right direction. We believe it is also easier to understand the singleton concept to someone who already knows classes, than introducing static members.

Definition of singleton is similar to standard class, the only differences are that the node starts with keywords `singleton class` instead of `class`, and its constructor cannot have any parameters.

Singleton instance is referenced only by the name of the singleton class. It is not allowed to declare a variable of singleton class type. This is because we wanted to make Eddie simple and without inheritance, there are not many situations, where it makes sense to store singleton instance in another variable.

## 4.5 Syntax

Language constructs and their syntax are introduced in the text and the figures of the previous section. This section describes some general decisions about language syntax and syntax highlighting. For the basic language constructs, we tried to replace

some typical programming keywords with words that are more common in the real world, in order to make it easier to learn for beginners. On the other hand, for advanced users can be better to learn common programming keywords, so in the object-oriented fragment of Eddie, we tried to use the same keywords that are used in standard object-oriented languages.

We considered words `procedure` or `routine` for a definition of custom actions, but finally, we used simply word `action`, because we think it fits into the category of basic language construct, where we wanted to use names from the real world. For basic types we used `text` instead of `string`, and `number` instead of `integer` or `int`, because we believe that integral numbers are enough to learn programming basics. As a name for `logical`, which is equivalent to `Boolean` type, we considered also `boolean`, `flag`, and `condition`.

Many language constructs contain block statements and we thought how to unify them. We considered three possible syntax for code block: surrounding with `begin` and `end` keywords, surrounding with braces, or custom keywords for each construct with some unified element. `Begin` and `end` are clear for users, but with more nested block statements, it results in visually chaotic code. The braces are more convenient for experienced programmers, but for beginners it can be confusing to use symbols instead of words.

Customized code block allows to distinguish ends of different control statements (e.g., `if – fi`, `do – done`), but this introduces new keywords that can distract users. We decided to use customized code combined with benefits of projectional editor. When a user inserts some control statement in projectional editor, its whole structure is created and it is possible to use arbitrary text in the code block.

Figure 4.15 shows the code block syntax. Opening keyword is chosen to help understand the language construct. Block endings uses keyword `end` and after it, there is a name of the construct. Block endings of actions, functions, and methods, contain their identifier in the block ending. In order to minimize distraction potential of keywords and names in the block ending, they have grey colour.



```
script CodeBlock defined as
  if full then
    end if
    repeat 6 times do
    end repeat
  end script
action turn right defined as
end turn right
```

*Figure 4.15 Syntax of code blocks*

Projectional editor does not need to parse the code, so it does not have limitations for naming conventions of identifiers. We decided to have predefined action names with all small letters and with spaces between words. We did not want to distract users with different notations, so we used the most natural way how to write it. We kept the same naming convention also for custom actions. On the other hand, names of functions do not allow spaces in their identifiers. There are two reasons for that, firstly functions can be used inside expressions and spaces there could worsen their readability, and secondly we think that when users have enough experiences to learn functions, they can also learn about camel notation.

Beside syntax of the concepts and naming conventions, each word in the code can have some font style. We use normal, bold, italics, and underlined font style and different colours. We designed some general rules, how we applied these styles. Definitions of identifiers are in normal font and references to them are in italics. Keywords are usually in bold, but keywords that are not a primary part of some language construct can be in normal font in order to highlight the primary keywords. Underlined text is used only in class nodes to separate different kind of class members.

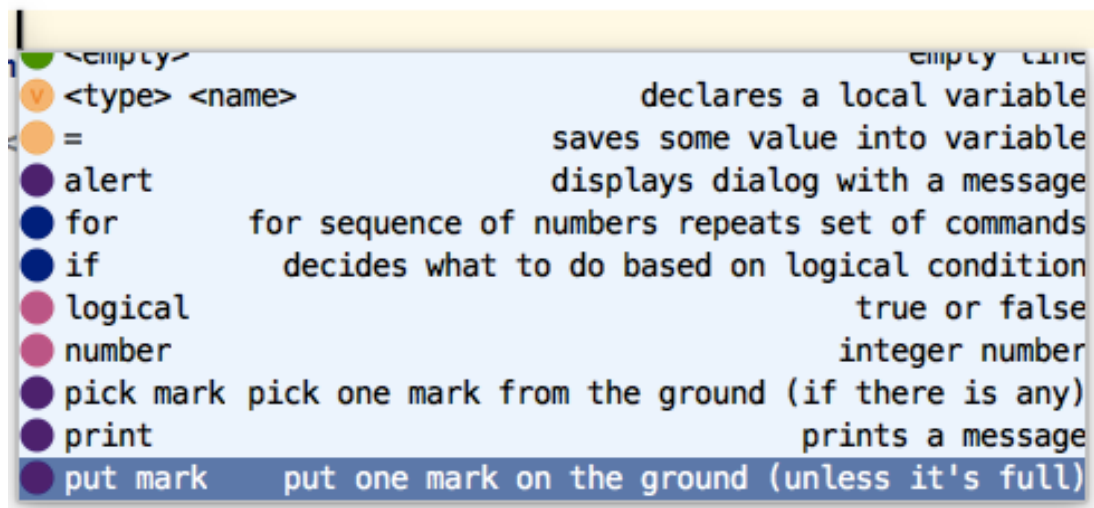


Figure 4.16 Colour icons in code completion menu

One of the key requirements for our language is to organize language constructs into groups and mark them with different colours. Languages with graphical code blocks have a toolbox, where language constructs are organized, and users can drag them from there and place them in the code. In Eddie, language constructs are created using the code completion menu that does not allow organizing constructs into groups, so we tried at least distinguish them by different syntax highlighting styles and different colour icons in the code completion menu, as is shown in Figure 4.16. For this purposes we selected following style groups:

- *Control statements* are dark blue. Primary keywords are in bold, secondary keywords, which have usually documentary character, are in normal font weight.
- *Actions* are dark magenta. Predefined actions are in bold, since they are part of the language. User defined actions are in italics because they are references to their definition.
- *Types* are crimson. Predefined types are in bold and class types are in normal font weight.
- *Expressions* are large group so it is divided into subgroups with different styles.
  - *Literals* are light blue. Most IDEs uses different colours for string, number, and Boolean literals. We believe that same colour for all literals can help beginners to orient among different programming language concepts.
  - *Operators* have same style as control statements.

- Other expressions are black. This group contains mainly variable references and function calls.
- *Modifiers* are grey. Eddie contains access modifiers for class members and mutability modifiers for variable declarations. We think that modifiers are less important than constructs that they modify, so we chose some bland colour.
- *Documentation elements* are dark green and in bold. We believe that documentation should be distinct, because they are meant for users.

MPS allows creating *editor hints*, which allow using a different editor for some concepts. We created a prototype of an editor for class definition that has colour background. It is inspired by the editor of BlueJ and Greenfoot and we believe that it can help beginners with orientation in the class definition code. Users can decide whether they use it or not. When they want to activate it, they need to invoke contextual menu in the editor of a class node, choose option *Push Editor Hints*, and check a checkbox *ColorBackground* as is shown in Figure 4.17.

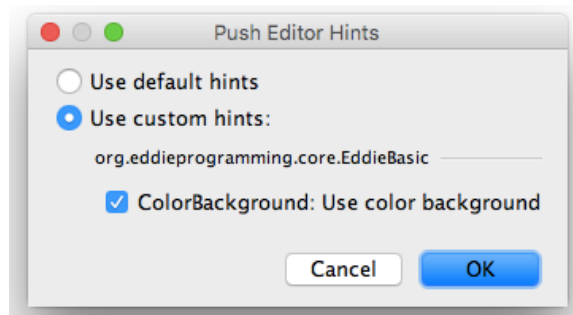


Figure 4.17 *Push Editor Hints* window

Figure 4.18 shows how a class node looks like in the editor with colour background. Unfortunately, MPS was not designed to use the editor like this and there are some limitations. White boxes in the picture have some space around and we believe that it makes them clearer, but it is not possible to set this appearance simply by some padding attributes. It is possible to set padding, but it is ignored around the colour boxes. Instead of it, we created it using additional horizontal and vertical collections and empty cells, which clutter editor source code.

```

class TracingRobot defined as
  fields:
    private text val robotName
    private logical val trace

  constructor:
    constructor with parameters
      text name
    defined as
      this.robotName = name
    end constructor

  methods:
    public method doStep
      with parameters:
        << ... >>
      with return type none
      defined as
        if this.trace then
          this.trace ( message = "I am doing step." )
        end if
        Karel.step ( )
      end doStep

    private method trace
      with parameters:
        text message
      with return type none
      defined as
        print this.robotName + ": " + message
      end trace

```

Figure 4.18 Colour background editor

## 4.6 Map and Tutorial Definition

One of our requirements for Eddie is that it should have some tutorial. A tutorial consists of scenarios, each scenario is connected with one Eddie program, and users have to do some coding tasks in the program. Scenarios are built in Eddie and they can be viewed as levels in a game. At each level, a user should program robot Karel to complete some tasks. The tutorial should gradually introduce programming constructs and concepts such as variables, arithmetic, functions, or object-oriented programming.

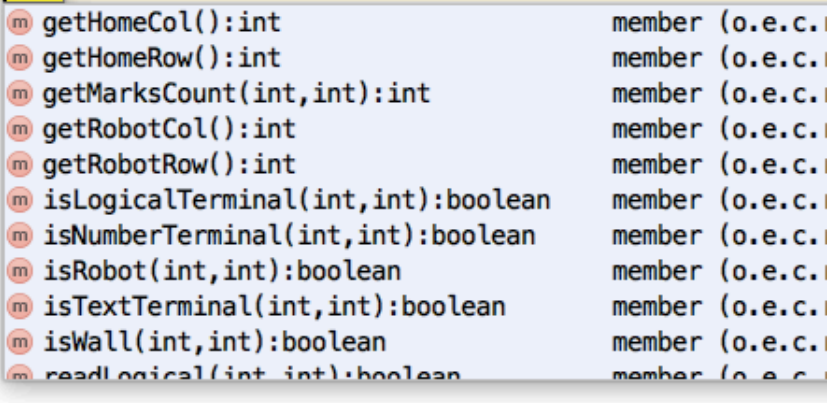
We designed languages that extends Eddie and defines two main concepts: a map and a scenario. The map describes objects in the Eddie map and the scenario defines tasks for users. A map and a scenario are assigned with one program; the program serves as an entry point for users. The languages for tutorial definition are intended to be used by teachers or experienced users and their editors have less helper features

than the editor for the rest of Eddie. These languages provide all necessary means to define maps and scenarios. The languages are: EddieSceneConstruction, EddieBaseLanguageObjects, and EddieComparisonOperators. EddieSceneConstruction allows defining maps and scenarios. EddieBaseLanguageObjects allows to access specific classes defined in BaseLanguage as a singleton class in Eddie. It is used to access the *Map API* that is written in BaseLanguage. EddieComparisonOperators adds following comparison operators to Eddie: equals, greater or equals, greater then, less or equals, less then. The operators are applicable to numbers. Eddie does not contain these comparison operators; the reason for this is described in Section 5.3.4. Some features for scenario definitions needs to compare numbers, so EddieComparisonOperators allows it, even though using these operators in expressions is less user-friendly.

```
Map MapExample {
  width: 10
  height: 10

  robot position:
    row: 1
    col: 1
    direction: east

  createMap {
    Scene.setMarks ( row = 4, col = 5, count = 6 )
    Scene.
  }
}
```



*Figure 4.19 Map definition*

EddieSceneConstruction language contains the two root concepts: the map and the scenario. The map defines its size, initial position of the robot, and its content. Figure 4.19 shows a map definition. Each map has walls around itself. Coordinates are indexed from zero, but the walls at the borders are also counted into map size, so the upper left corner position, where it is allowed to put the robot has coordinates (1, 1).

A content of the map is defined in `createMap` method. Inside the method, it is possible to use Eddie language and Map API. Methods of Map API are displayed in the code completion menu in Figure 4.19.

Figure 4.20 shows a scenario definition. It contains a reference to a map definition, a scenario description, which is displayed to users, an initial program, and a list of events. Initial program is an Eddie program that defines initial state in the user program for this scenario. Program can contain some initial code to help user to finish the task or some comments to describe the task. User program for the scenario contains button Reset that allows users to discard their solution, when they are not satisfied, and start again from the beginning with the initial code. When Reset button is pressed, the content of current program node is deleted and the content of the initial program is copied into the user's program. If the initial program uses some libraries, the libraries are also reverted to the initial state. Reset feature is not implemented for classes and singleton classes in this prototype.

```
Scenario ScenarioExample
  map TestScene
  initProgram: TestInitProgram
  description:

  ###
  # Task: Put five marks in front of the house.
  ###
  events:
    Event
      description: All marks were put in the right place
      On condition: Scene.getMarksCount ( row = 2, col = 2 ) == 5
      do action: {
        alert "Well done! Missing accomplished!"
      }

    Event
      description: <no description>
      On condition: Scene.getMarksCount ( row = 2, col = 2 ) == 4
      do action: {
        alert "Almost there."
      }

  }
```

*Figure 4.20 Scenario definition*

An event has a description, a condition, and an action. The event description is optional and it is used only as a private documentation. The condition is a logical expression in Eddie. The action is a body of custom action in Eddie. When a program with the scenario is executed, conditions of scenario's events are checked after every

action that influences current state of the map. If the condition is satisfied, the defined action is executed. For example, the condition checks the task of the scenario and if it is completed, action `alert` is called to display a success message in Eddie.

Each program can have references to a map and a scenario. This is the only way, how users can connect a program to a scenario or a map. The referenced map defines how the map for the program looks like, when the program starts. The scenario defines tasks for the program. Scenario should have defined a map. If both map and scenario are set, the map from the scenario is used. If neither map nor scenario are set, default empty map is used. References to a map or a scenario are set in Inspector Tool, so they are usually hidden from users.

We designed three helper classes for map and scenario definition. They are defined in `BaseLanguage` and they can be accessed using `EddieBaseLanguageObjects` language. The first one is *Random* that has methods to generate random number, random logical value, and random name of an animal as text. It can be used in scenarios, whose goals are to find some value and write it into a computer terminal, e.g. count all marks and write the number down. This value can be generated randomly so users cannot simply count marks on the screen and then just create program that writes this value as a constant.

When a scenario uses a random value that should be read and written by the robot and the scenario checks whether the values are read correctly, the randomly generated values must be stored somewhere. This is a task for *Storage* helper class, which allows passing the generated value from the map definition to an event condition in the scenario definition. Storage is a key-value data structure, where the key is some text and the value can be number, logical, or text. For each type it has methods to put and get values.

The last helper class is *Statistics*. Statistics contains statistical information about current program such as number of commands in the body of the program or number of commands in the whole program. It is possible to set and check limits for the number of commands in the scenario. It allows, for example, forcing users to use a loop instead a sequence of several same commands.

# 5 Implementation

Previous chapter describes Eddie language and Eddie Studio from the high-level point of view. This chapter describes it from a lower-level point of view. It describes some architectural decisions, technical design, and implementation details of the language, runtime environment, Eddie Panel, and Eddie Studio.

## 5.1 Non-Functional Requirements

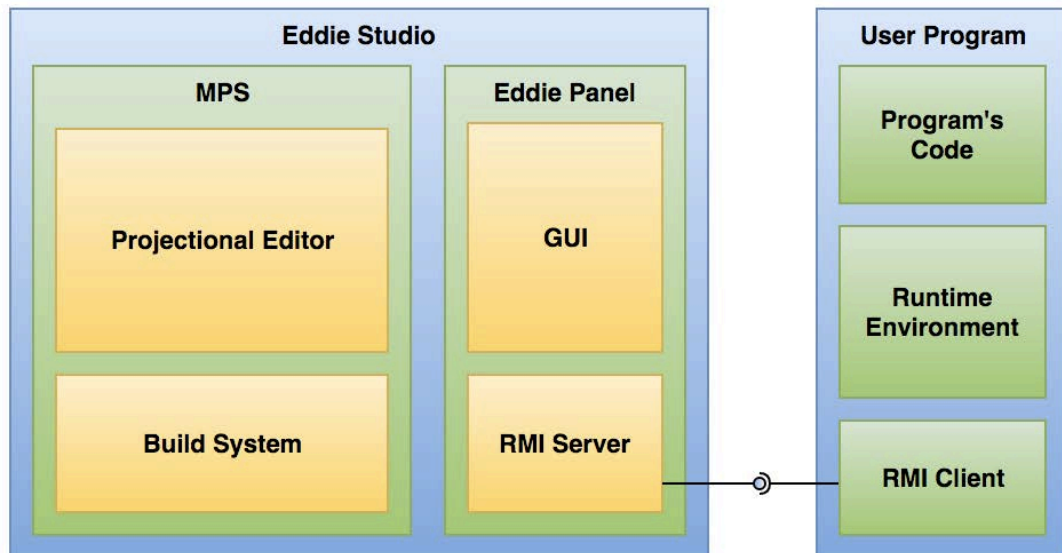
We specified two main non-functional requirements that are important for us. The first one is extendibility. It should be possible to create a new language constructs as a new MPS language that would be *language extension* of Eddie. It should be possible to create a different domain which can be used instead of domain Karel the Robot. A new domain language would require also to add a new visualisation in Eddie Panel. This should be possible without modifications of Eddie Panel.

One of the key features of educational programming tools identified in Section 2.3 is to provide it in more human languages. The second requirement is that it is possible to translate language and Eddie Panel into other languages.

## 5.2 Architecture

Figure 5.1 shows a runtime architecture overview of the whole system. There are *Eddie Studio* and *User Program*. Both of them run in separated Java Virtual Machine (JVM) and communicate via Java RMI. Eddie Studio is customized MPS application. It contains standard MPS modules and custom Eddie Panel. The main parts of MPS are the projectional editor and the build system. The projectional editor uses Eddie language definition to create and edit programs in Eddie. The build system uses Generator aspect of Eddie to transform Eddie programs into runnable Java classes. These classes are displayed in the right side of the picture as User Program. They contain transformed code from Eddie, runtime environment that represents actual state of Eddie map, and RMI client. The last part of the picture is Eddie Panel. It contains Eddie GUI and RMI server. RMI server allows programs to connect to the Eddie GUI and visualise Eddie programs there. Besides map visualisation, Eddie GUI contains also a panel to control program execution. Details are described in following sections.





*Figure 5.1 Eddie Studio architecture*

### 5.2.1 Requirements Implementation

In order to fulfil the requirement for extendibility, Eddie is defined as a modular language and Eddie Panel provides a layer of abstraction to display objects in the two-dimensional map. This layer is independent on the Karel the Robot domain. Language implementation is described in Section 5.3 and Eddie Panel abstraction is described in Section 5.5.

Both Eddie language and Eddie Panel are designed for localization into other human languages. The most of the text labels in Eddie and all the text labels in Eddie Panel are prepared for localization. The localized language text labels are mostly warning and error messages, and intention labels.

The text labels of Eddie Panel are stored in Java property file and are referenced using text property key. Here is an example:

```
controls.messages.label=Messages
controls.button.stop.tooltip=Stop
```

Some text in the language cannot be localized in MPS, because it can be either defined only as a text literal (e.g. alias and short description of concepts) or it is not defined in the language at all (e.g. some language error messages). MPS does not allow including text files into MPS project, so language text labels are stored in a BaseLanguage class `Labels` in a key-value map. In final version of Eddie Studio, they could be stored also as Java property file, which can be packed and imported as

JAR archive, or all localization files could be stored at predefined path in user home directory. Language selection is not implemented in the prototype.

## 5.3 Language

This section describes how is Eddie defined in MPS. First, it describes the modular design of Eddie and then its implementation details. Implementation description is organized by MPS language aspects. The following text uses some generalizations; it refers to all custom actions, functions, class methods, and class constructors as methods and to all local variables, parameters, and class fields as variables.

Eddie language is defined as ten modular languages. We created five core languages, two domain languages, and three languages that are used to define maps and scenarios. Some of the languages extend others. Decomposition of Eddie core languages allows using them in the tutorial one by one, so users do not see language constructs that they have not known yet in the code completion menu. Decomposition also allows extending Eddie languages in order to add some new features. The core languages are the following:

- *EddieBasic* is the main language, it contains program and library root concepts, `print` and `alert` actions, custom actions, `repeat` loop, literals (number, logical, and text), line comments, and documentation blocks. It also contains abstract foundation for types, variables, and methods, which are implemented in other language extensions.
- *EddieConditions* language introduces `if-then-else` statement, `while` loop, and logical operators (negation, conjunction, disjunction).
- *EddieVariables* language introduces local variables, assignment expression, arithmetic operations (addition, subtraction, multiplication, and division), text concatenation, and `for` loop. `For` loop uses iteration variable, so it cannot be introduced earlier with other control statements in *EddieBasic*.
- *EddieFunctions* language contains functions with parameters and return values and expressions to call these functions.
- *EddieObjects* introduces classes, singleton classes, and concepts to create instances and manipulates with them.

Domain languages are separated from the core languages, so it is possible to create a completely new domain and use it with Eddie. Users can import only domain languages that they want to use and language constructs from other domain

languages will not appear in the code completion menu. The domain languages are the following:

- *RobotKarel* language contains actions to control the robot and map queries to observe the area around the robot.
- *EddieTerminals* language brings the computer terminals feature. Computer terminals can be part of the map and the robot can check, if there is a terminal of particular type or it can read and write variables from and into terminal.

The languages for map and scenario definition are:

- *EddieSceneConstruction* allows to define maps and scenarios that can be used for programs.
- *EddieBaseLanguageObjects* allows to access some BaseLanguage objects in Eddie as singleton objects.
- *EddieComparisonOperators* language contains comparison operators that are needed to check some conditions in scenarios.

### 5.3.1 Language Aspects

Implementation of each Eddie language is divided into language aspects and follows standard principles as is described in Chapter 3. This section aims to describe parts of Eddie that uses some concepts that are not sufficiently described in Chapter 3 or they are more complex. In order to make it simpler, it refers to Eddie as one language and does not distinguish the modular languages that are described in previous section. Typesystem and DataFlow aspects refer to some more complex issues and so they are described separately in the next sections.

#### Structure and Behavior

Structure aspect contains concepts that define AST structure of any Eddie code. Constructs of Eddie language are described in Section 4.4 and they are divided into language modules as is described in the previous section. We created two main abstract concepts: *AbstractCommand* and *Expression*. *AbstractCommand* is a common ancestor for all statements and *Expression* is a common ancestor for all expressions. A sequence of statements is represented as *CommandList*. *CommandList* represents a body of programs, control statements, and all kind of methods.

Behavior Aspect defines methods for nodes and concepts that are used in other aspects, e.g. every expression must implement method `getTypeAnnotation`, which returns expression's type.

## Editor

Editors for concepts are defined in standard way using *Concept Editor* nodes as is generally described in Section 3.3.2. Editor parts that can be reused in other concepts are defined as reusable *Editor Components*, which can be used in Concept Editor nodes. For example, `MethodDefinitionParams_Component` defines how parameters are defined and it is used in concept editor for both function definition and class method definition.

We used also *Cell Action Maps*, *Stylesheet*, and *Editor Hint*. Cell Action Maps defines delete actions in Eddie. For example, we created actions for Binary Operators. When a cursor is at the position of an operator and `DELETE` key is pressed for the first time, the right operand is deleted. When it is pressed for the second time, the whole operator is deleted and replaced with its left operand. When a cursor is at the position of an operator and `BACKSPACE` key is pressed for the first time, the left operand is deleted and for the second time the whole operator is deleted and replaced with the right operand.

Stylesheet contains visual styles for each group of language constructs, e.g. control statements, actions, expressions. We defined styles for opening and closing keywords of code block using *Matching Labels*, so when either opening or closing keyword is selected, both of them are highlighted. Figure 5.2 shows two `if-then-else` statements. On the left side, the first `if` keyword is highlighted because a user clicked on the grey `if` keyword in the end. On the right side, the `end` keyword is highlighted, because a user placed a cursor at `then` keyword. This feature can help with orientation in code with more nested control statements.



```
if wall ahead then
  turn left
else
  step
end if
```

Figure 5.2 Highlighting of matching opening and closing keywords

Editor aspect contains also an Editor Hint *ColourBackground*. It allows switching to alternative editors that highlights some language concepts with different background colours. These editors are implemented only for some concepts. If a concept does not have an editor for this Editor Hint, a default editor is used.

## **Constraints**

Constraints aspect defines some constraints over nodes' children, properties, and references. Basic AST structure is defined in Structure aspect, all child nodes and references have specified their concept. Constraints aspect specifies more precise rules about AST structure, e.g. *ReturnStatement* can be defined only in a function or method body. Properties values are validated, e.g. validation of identifiers' names. Constraints aspect also restricts the scope of references, e.g. scope of local variables, custom actions, and methods.

## **Actions**

Actions aspect makes manipulation with AST easier. We used three main Actions aspect concepts in Eddie: Node Factories, Node Substitution Actions, and Node Transform Actions. The Node Factories define for a particular concept, how a node of the concept is initialized. If a new node is created to replace some existing node, the new node can use values of the existing node in its initialization, e.g. *BasicNodeFactories* node defines that when a binary operator is changed to other binary operator, its operands are preserved, or when a loop is changed to other loop, its body is preserved.

Node Substitute Actions provide automatic substitution of AST fragment, and are used to calculate possible options in the code completion menu. Each Node Substitute Action defines a concept that can be substituted. MPS contains different kind of substitutions. We used two of them in Eddie: adding *Node Wrapper* and adding *Parameterized Items*. When a user writes a name of a type in Eddie in a place, where a statement is possible to write, it creates a local variable declaration. This is done using Node Wrappers. They define how a newly created node can be wrapped by other new nodes (e.g. a type node is wrapped in a local variable declaration node), so it is possible to create a concept at the place where it cannot be directly inserted. Parameterized Items are used to create custom items in code completion menu. For example, we created *AbstractCall\_Expression* that defines all types of method calls. It substitutes an *Expression*, which means that this item is displayed in the code completion menu anywhere where it is possible to write an

Expression. At first, it finds all accessible methods and then it uses the method definitions as a parameter to create alias, short description, icon, and the output node. The output node is an `AbstractCall` node with a method signature that contains all the parameters of the method.

## Intentions

Intention aspect adds a special *Intention contextual menu* to some node in the editor. Intention menu can trigger arbitrary action, e.g. *AddComment* intention adds a documentation block to a node that can have a documentation block. Intentions can be also designed to fix some error, e.g. intention *FixConstructorParams* is offered when the count of parameters of a method call differs from the count in the method definition. When this intention is invoked, it recreates the proper call signature and copies the parameter values from the original call. These error intentions have red icons.

### 5.3.2 Type System

Typesystem aspect defines a type for expressions and provides some checks that lead to info, warning, or error notifications. The notification is reported at the affected node in the editor. Error is also displayed in a popup window every time when the code is built.

Eddie recognizes the following types, organized in a hierarchy:

- Type
  - VariableType
    - SimpleType
      - IntegerType
      - BooleanType
      - StringType
    - AbstractClassType
      - ClassType
      - SingletonType
  - VoidType

`VoidType` is used only as a return type of methods; it cannot be used as a type of variables. `VariableType` can be used as a type of variables and they can be either `SimpleType` or `AbstractClassType`. We created three subtypes of `SimpleType`:

`IntegerType`, `BooleanType`, and `StringType` that are displayed as `number`, `logical` resp. `text` in Eddie and represented as `int`, `boolean`, resp. `String` in Java. We created two representatives of `AbstractClassType`: `ClassType` and `SingletonType`. `ClassType` represents a variable that holds instance of a custom class. Class name is used as a name of the type. `SingletonType` is used in references to singletons. Each `AbstractClassType` are not represented a node of the concept and they have a reference to an Eddie class or singleton definition.

We created *Typesystem Inference Rules* that covers all expression concepts and specifies their result type. Types of literals, map queries, and result of the most of the operators, are defined directly as `SimpleType`. Types of variable declarations, variable references, this expression, and dot operator are defined recursively. Types of some binary operators are defined using *Overloaded Operation Rules* concept. It defines type of `PlusOperator` between two numbers as a number, and type of `PlusOperator` between text and any other type as text, because in the second option `PlusOperator` does text concatenation.

For other concepts that contain expressions as child nodes, e.g. `count` in `repeat loop`, `condition` in `if-then-else` statement, we defined checking rules which checks whether the child nodes types are correct.

Type system in MPS has a limitation; it operates on created nodes and cannot work at the level of concepts. Therefore, even when some concept has always same type, it is not possible to check the type before a node is created and forbid to create any node of incompatible type. For example, `repeat loop` has an attribute `count`, which can be only number. However, if it were restrained only by *Typesystem Rules*, a text literal would be offered in the code completion menu for the attribute `count` too. This is because the MPS type system works only with created nodes, not with concepts. After the text literal is inserted as a `count` attribute of `repeat loop`, it is correctly reported as type system error. This error could be completely prevented if the code completion menu did not offer the text literal at all. What is displayed in code completion menu can be restricted by *Constraints* aspect. Restriction must be based only on concepts, so a type of a potential child node must be determined by some other means than the MPS type system, which works only with nodes.

In order to deal with this, Eddie introduces also a system of static types. It is used only in *Constraints* aspect to prevent to display concepts with an incompatible type in

the code completion menu. It is implemented using static methods in Behavior definition of concepts. The method returns the static type or `null`. If the static type of a potential child node is incompatible with the expected type, it is not offered in code completion menu. In all other situations, including when it returns `null`, static type is ignored and concept can be displayed in code completion menu. After the node is created, it is checked with normal MPS type system. The static types can be used only for literals, some operators limited to a single type, or map queries. Static types are essential at the beginning, when users use only EddieBasic and EddieConditions languages and types are not introduced yet.

### 5.3.3 Static Analysis

In order to find and report some errors before program is built, Eddie contains static analysis, which is implemented using DataFlow aspect. It is implemented in standard way as it is described in Section 3.3.7. DataFlow analysis looks for several situations. The following ones are reported as errors:

- There is unreachable code after a `return` statement in a method.
- A method that has a return type does not return any value in some method's exit point.

The following situations are reported as warnings:

- A variable is assigned with a value that is never read.
- A value is read from a variable that is not previously assigned.
- There is a variable that is never referenced.

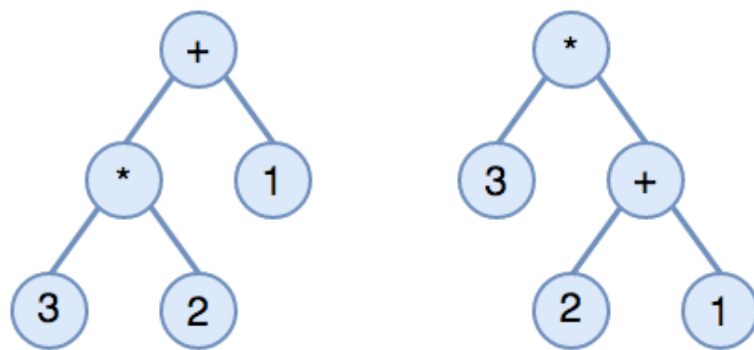
Static analysis is checked using Non-Typepесystem rules in Typesystem aspect. Because creating the dataflow program for large code can be slow, and it is executed in the same thread that maintains the editor, there is a limit for the program's length, and larger programs are not analysed. Analysis is invoked separately for the bodies of programs, custom actions, functions, object methods, or constructors.

### 5.3.4 Expressions and Operators

In Chapter 4, there is mentioned that we decided to use simplified implementation of expressions. As a result, Eddie does not contain parenthesis and comparison operators. This section addresses this issue in detail. In code, expressions are written and displayed in infix notation, which can lead to ambiguity, unless priorities are defined, or parentheses are used. In AST, expressions are represented unambiguously

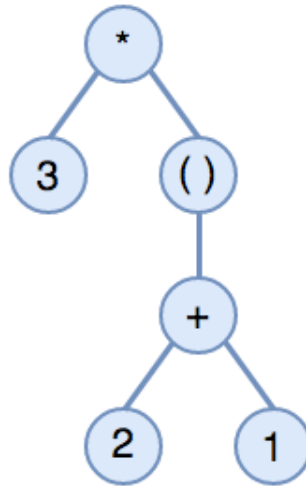


as binary expression tree without operator priorities or parenthesis. In BaseLanguage, when an expression is edited in projectional editor, operator priorities and written parenthesis are evaluated, and the nodes of the expression in AST are rotated if it is necessary. Parenthesis in AST serves only as a marker that they should be displayed in the editor or generated into textual source code. When a user adds parenthesis to the expression, the expression tree structure is adjusted to reflect changed priorities. This implementation, as is in BaseLanguage, would be time-consuming and would not bring anything new, so we decided not to implement it in the prototype of Eddie. Instead of that, we chose a solution that is very simplified, but we believe that it is sufficient for educational purposes.



*Figure 5.3 Expression trees for an expression  $3 * 2 + 1$  in Eddie*

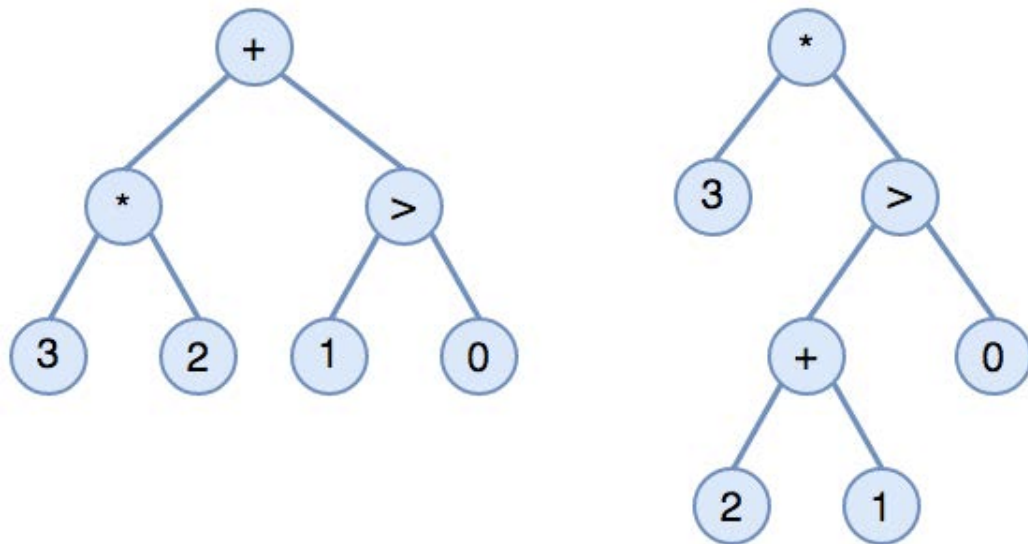
Our solution is that Eddie does not allow writing parenthesis and does not explicitly care about operator priorities. An expression is in Eddie always represented in AST in the same way that was written in the editor, so one expression can have more representations. Figure 5.3 shows two expression trees, which both were written as an expression `'3 * 2 + 1'`. The tree on the right side is not a correct representation of `'3 * 2 + 1'`, it should be displayed as `'3 * (2 + 1)'`, and this tree representation could not be created in BaseLanguage using editor. BaseLanguage editor would transform the expression `'3 * 2 + 1'` into the tree on the left side. A tree representation of the expression `'3 * (2 + 1)'` would have parenthesis in BaseLanguage, as is shown in Figure 5.4.



*Figure 5.4 An expression tree for the expression  $3 * (2 + 1)$  in BaseLanguage*

Expressions in Eddie AST are not actually represented as expression trees, but they are expressions in infix notation stored as trees. When program is executed, expression written in Eddie is generated into BaseLanguage. Generator does not use the editor for BaseLanguage, which would transform the expression into the correct form, but it creates AST of the expression in the same order as it is in Eddie. This can result in AST fragment that is not possible to create using the BaseLanguage editor. This fragment is finally transformed into Java source code as text in infix notation, so the result is again correct.

Operator priorities are also important in situations, when an expression contains expressions of different types, e.g. when there is an expression `'3 * 2 + 1'`, and it is transformed it to an expression `'3 * 2 + 1 > 0'` by writing `' > 0'` behind the original expression. It results in type system error in both representations displayed in Figure 5.3. Figure 5.5 shows expression trees after writing `' > 0'` behind the expression, without any transformations. Both situations lead to a type system error, because the result of the 'greater than' operator is of logical type, and it is not possible to add resp. multiply a number with an expression of logical type.



*Figure 5.5 Expression trees of expression with mixed types which results in typesystem errors*

Therefore, if arithmetic and logical operators had been mixed together in Eddie, it would not have been possible to edit them like in a text editor. Types can be mixed without problems only in character string concatenation operator, because in this case the operand that is evaluated as number or logical is automatically converted to text, so a type system error does not occur.

## 5.4 Runtime

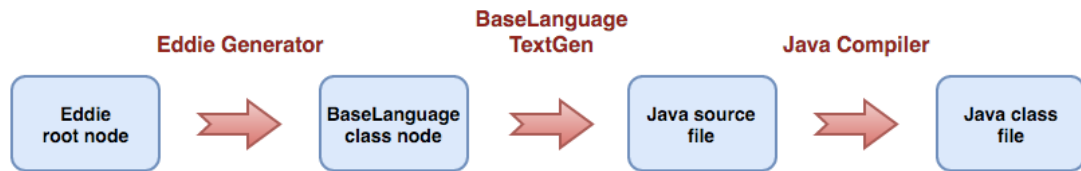
When users write some code in Eddie, it must be somehow executed. There are two possible approaches how to execute created code. First option is to use Generator aspect. Generator defines how code in Eddie is transformed into BaseLanguage and the result can be executed as Java. The second option is to create an interpreter that would interpret Eddie. We decided to use generator. The following sections describe both approaches and discuss why we chose generator approach.

### 5.4.1 Generator

MPS allows transforming code of projectional language into another projectional language using Generator aspect. Generator aspect defines rules how are nodes of input AST transformed into nodes of output AST, details are described in Section 3.3.9.

Users needs to explicitly run the program in order to see its output. When a user hits *Run* button, Eddie Studio builds the program node and all the root nodes in the same

model. Figure 5.6 shows build process in Eddie. It starts with an Eddie root node, which can be program, library, class, or singleton class. The Eddie root node is transformed into a BaseLanguage class node using Generator aspect in Eddie languages. The BaseLanguage class node is then transformed into textual source file of a Java class using BaseLanguage’s TextGen aspect and then the Java source file is compiled using Java Compiler. The Eddie program node, which is supposed to be run, is transformed into a Java class with a main method, and then it is executed in Java. It runs in a different Java Virtual Machine (JVM), so after it is transformed and started, it cannot directly interact with the editor. The executed program communicates with Eddie Studio over a network connection.



*Figure 5.6 Build process in Eddie*

In the prototype, there is a limitation for Eddie that it is not possible to reference libraries, classes, or singletons defined in imported models. They can be used only from the model where they are defined, so it is not possible to create something like a library that would be distributed to other users, because the code of the library would be in a separate model. This is caused by the chosen initial implementation of how references in Generator are maintained. Cross-model references need some special effort. We believe that it should not limit users in common usage of Eddie. It is also possible to change the implementation in the future. On the contrary, map and scenarios definitions should be hidden from users in tutorials, so they have different implementation of maintaining references and they can be referenced from other models.

### 5.4.2 Alternative Approach

We also consider an alternative approach to Generator. We would design a language *interpreter* that would read and interpret AST of Eddie. MPS provides *Open API* [11] that allows accessing AST of models, but there is not any support for interpreter in MPS. Interpreter would maintain call stack and current context. Each concept would have a method called `interpret`, which would do particular action for the concept. Methods would be implemented in Behavior aspect. Methods would

accept current context as a parameter so they could change the context. Interpreter would be executed from the editor, so it could interact with the editor.

We created a very simple proof of concept of interpreter for Karel the Robot domain. It is inspired by *Building an interpreter cookbook* [12]. The cookbook uses a language *Shapes*, which allows drawing 2D objects, and shows how a result of drawing code can be displayed directly in the editor. Our proof of concept is based on MPS example language *RobotKaja* and reuses its code. This proof of concept is not part of the Eddie project.

Figure 5.7 shows a screenshot from our proof of concept. There is a program on the left and a map on the right. The black symbol ‘>’ in the map represents the robot, other symbols are same as in Eddie. We implemented only three concepts: `step`, `turnLeft`, and `repeat`. The main difference from our solution in Eddie is that the map is directly in the editor, and each time a user changes the code, the program is executed from the beginning and animated in the map.

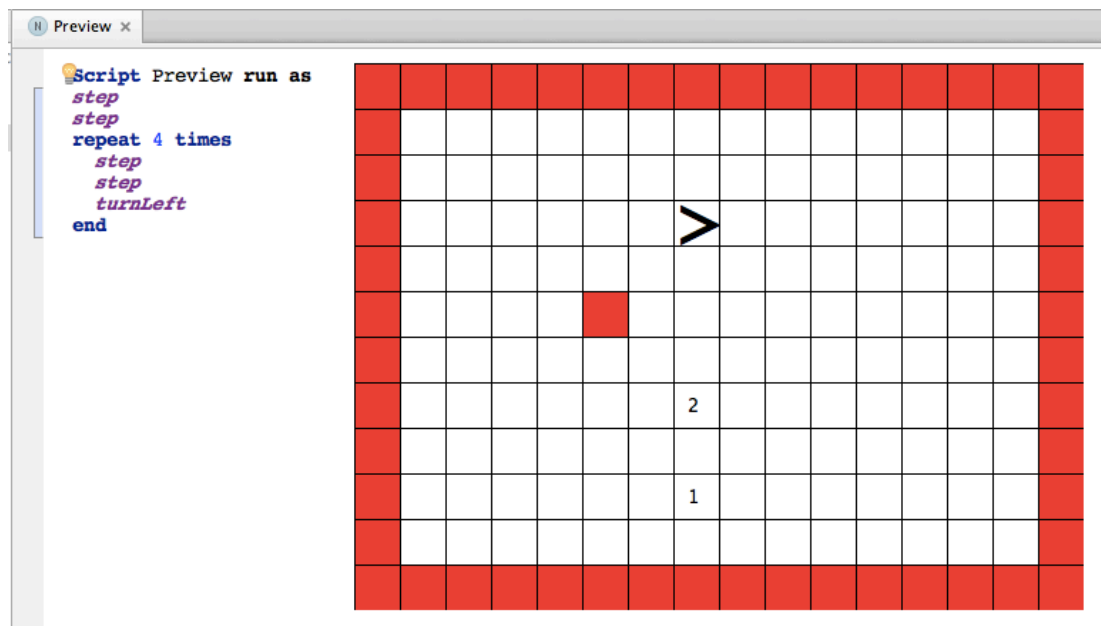


Figure 5.7 A screenshot of proof of concept of interpreted Karel language

### 5.4.3 Comparison

The limitations of generator approach are that it runs in different JVM, so it can communicate with Eddie Studio only over the network. It has also different Java *classpath*, so it is not possible to create parameters for network communication at runtime and pass them in a file in shared classpath. As a result of this, it would be difficult to run more instances of Eddie Panel in the same time, e.g. in Eddie Studio

and in MPS, because a port and a service name must be hardcoded in common libraries or stored in a configuration file in user home directory.

When users want to run their script, they need to click with the right button at the program node in and click on *Run*. The program is being built before it is executed, which can take some time, and users must wait.

The advantage of generator is that it is a standard approach in MPS, and Generator aspect makes it easier to be implemented. MPS also provides support for code debugging of languages that are generated into BaseLanguage. This could be used in future versions of Eddie language.

Interpreter approach could offer immediate program execution, without waiting to build the program. Its execution could be started either when code is changed or by pressing some *Run* button. The *Run* button could be part of the editor. Interpreter could also interact with the editor, e.g. it could highlight currently executed line of code or draw a path, how the robot would move in the map, when the current program was executed.

The main limitation of the interpreter approach is that its implementation would be much more difficult than using prepared Generator aspect. Implementation of advance concepts such as local variables, functions, and references to objects would take much more effort than in generator. This is why we finally decided to use generator.

We had an idea to implement the interpreter only for EddieBasic language and use it as program preview. It is discussed as an idea for future work in Chapter 7.

## 5.5 Eddie Panel

Eddie Panel serves to visualise actions in Eddie programs. Eddie Panel is implemented completely in Java. It is embedded into Eddie Studio as a custom window, which is called *Tool* in MPS. An Eddie program that is being executed communicates with Eddie Panel over the network. Eddie Panel provides abstract layer to manipulate with the Eddie map. The map consists of a two-dimensional grid and a set of things that can be placed at some coordinates in the grid. Both grid cells and things have some visual appearance. One cell can contain more things, but only the one that was added last, is displayed.

Network communication uses remote procedure call (RPC). The Eddie program is a client and Eddie Panel is a server. Communication is always initialized at the client, the server only responses. It is implemented using Java RMI (Remote Method Invocation).

Eddie Panel has defined a life cycle. At first, a map must be initialized. Then, instructions how to set and change the world are sent to the server in the form of steps. Each step has defined speed. When all the changes in the step are visualised, Eddie Panel waits some time before the next step is executed. The waiting time is defined by step speed.

Eddie Panel has the following states:

- *Disconnected*: no client has been connected yet, and Eddie map has not been created.
- *Scene construction*: client has been connected, and the map is being created. In this state all steps are executed without waiting, so it is possible to use the same client commands to prepare the world that are later used with animations in the program execution.
- *Ready for run*: the world has been created and Eddie Panel is waiting for a user to presses *Run* button.
- *Running*: Eddie program is running automatically.
- *Pause*: Eddie program is paused, it can be resumed by *Run* button, or it can be traced by *Step* button.
- *Stopped*: Program execution finished or it was stopped by a user.

## 5.6 Standalone IDE

Eddie is distributed as a standalone customized MPS IDE, called Eddie Studio. Figure 5.8 shows a screenshot of Eddie Studio. It looks similarly to MPS, but it contains also all the Eddie Languages, implemented tutorial, and Eddie Panel.

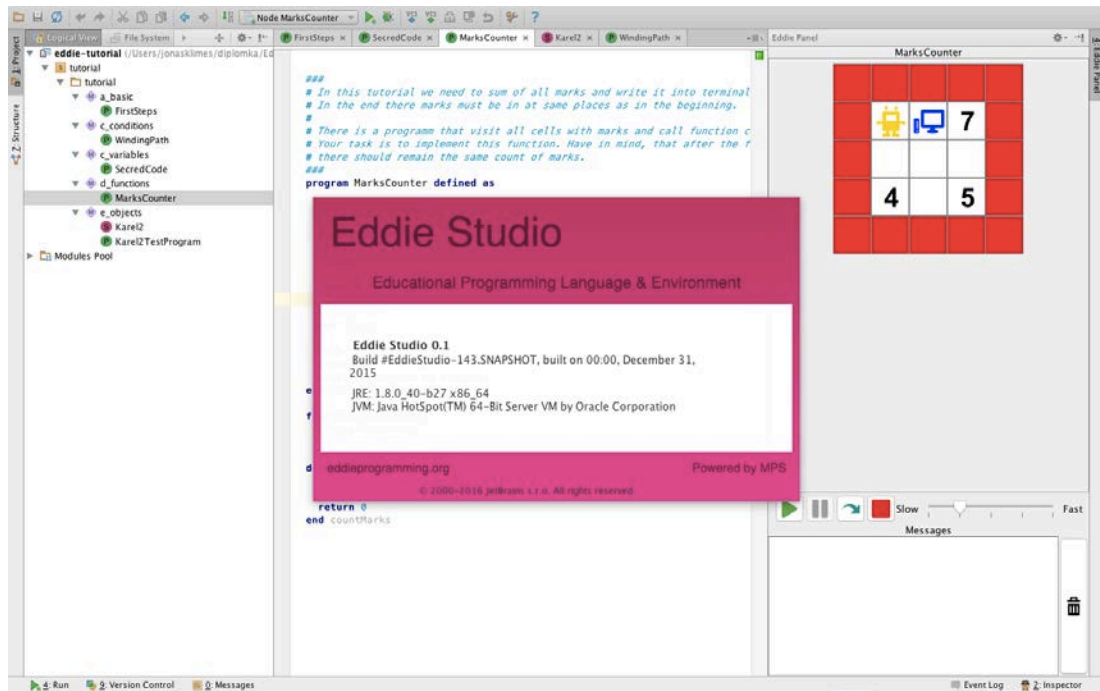


Figure 5.8 Screenshot from Eddie Studio

Eddie Studio can be build from *Eddie-lang* MPS project. Configuration for Eddie Studio is in the solution `org.eddieprogramming.ide.EddieStudio.build`. We created two build scripts in this solution: `EddieStudio` and `EddieStudioDistribution`. `EddieStudio` scripts generate Eddie languages and solutions and compile them into a plugin for IntelliJ IDEA platform. `EddieStudioDistribution` creates platform specific distributions of Eddie Studio from MPS application and the IDEA plugin created by `EddieStudio` script. `EddieStudioDistribution` script creates tar.gz package for Linux, OS X application bundle for Mac OS, and a generic distribution as ZIP archive. ZIP archive contains a Windows Batch file `mps.bat` to run it in Windows and a Shell script `mps.sh` to run it in Unix-like systems. `EddieStudio` build requires Java 8 JDK and MPS Generic Distribution in version 3.3.



# 6 Tutorial

As is described in Chapter 4, Eddie allows creating a tutorial. The tutorial consists of tutorial scenarios. This chapter describes examples of tutorial scenarios that we made. We created five scenarios, one scenario for each core language. We suppose that for introduction and practising all the concepts of Eddie it would be necessary to have

5–10 scenarios for each core language.

## First Steps

The first scenario is called *First Steps*, and it is the first scenario at all. It is designed to introduce two of the main robot commands: `step` and `turn left`. Figure 6.1 shows a map of the scenario. There is a robot and a house, and the scenario’s task is to navigate the robot into his house. Figure 6.2 shows the initial program for the scenario, which is the starting point for users. There is a description of the scenario and a program with empty body, where users are supposed to write code. In the top right corner, there is a button *Reset*, which resets the initial state of the scenario. Figure 6.3 show code that solves the task.

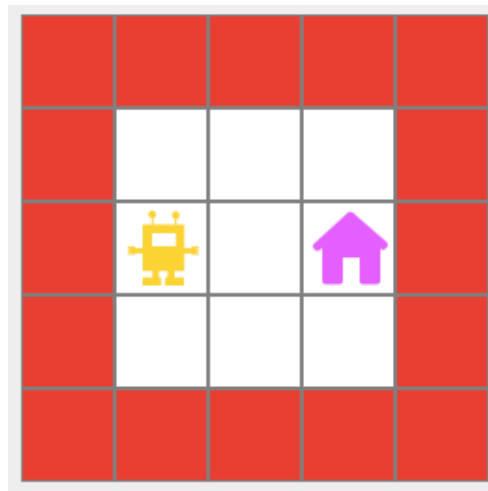


Figure 6.1 A map of *First Steps* scenario

Reset

```

###
# Learn how to move with robot.
# Use step and turnLeft commands to get robot to home.
#
# Run the program first to see the map.
###
program FirstSteps defined as
|
end program

```

<< ... >>

Figure 6.2 An initial program for First Steps scenario

```

program FirstSteps defined as
  turn left
  step
  step
end program

```

<< ... >>

Figure 6.3 A solution of First Steps scenario

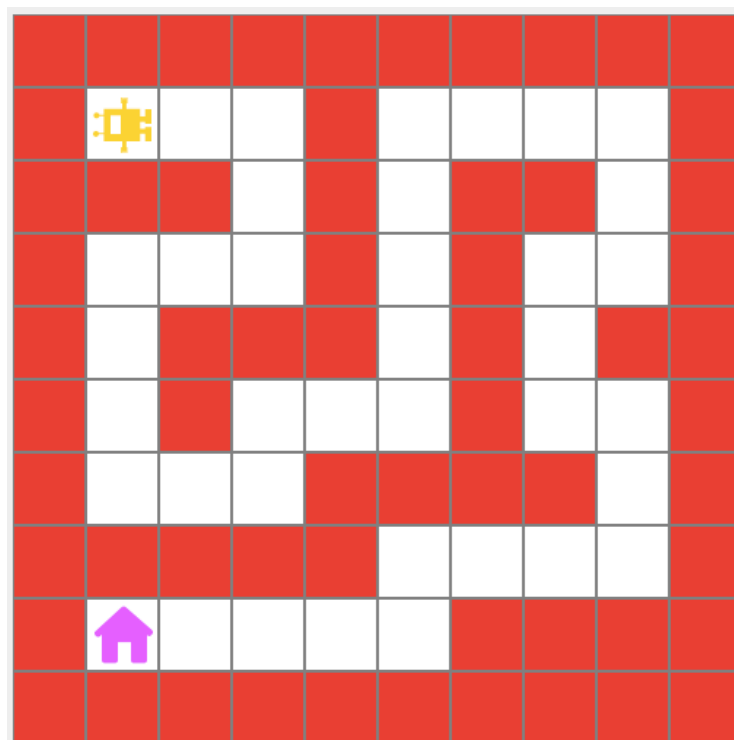


Figure 6.4 A map of Winding Path scenario

## Winding Path

The second scenario is for EddieConditions language, and it is called *Winding Path*. It contains a winding path that should the robot walk through with minimum commands. It is necessary to use while loops and if-then-else statements in order to minimize the number of commands. Figure 6.4 shows the map of the scenario. There is only one possible path, but it is too long and irregular to be effectively finished without using conditional control statements. The initial point for users is empty program. Figure 6.5 shows an example of a program that completes the task.

Reset

```
###
# Navigate robot Karel to its home.
# What is the lowest number of command that is necessary to do it?
###
program WindingPath defined as
  while not at home do
    # go straight on as far as possible
    while not wall ahead do
      step
    end while
    # turn left to check whether there is a free way
    turn left
    # check if there is a free way on the left
    if wall ahead then
      # there is not free way on the left so turn to the oposite direction
      turn left
      turn left
    end if
  end while
end program
```


 << ... >>

Figure 6.5 A solution for Winding Path scenario

## Secret Code

The third scenario is for EddieVariables language, and it is called *Secret Code*. Figure 6.6 shows a map for this scenario. There are two textual computer terminals. Figure 6.7 shows a solution of this scenario. Users are supposed to navigate the robot to the terminal in the upper left corner and to read a secret word from the terminal into a variable. Then the secret word from the variable should be written into the second computer terminal. The secret word is an animal name, randomly chosen in each program execution. The solution code uses local variables, computer terminals, and constructs from EddieBasic language.

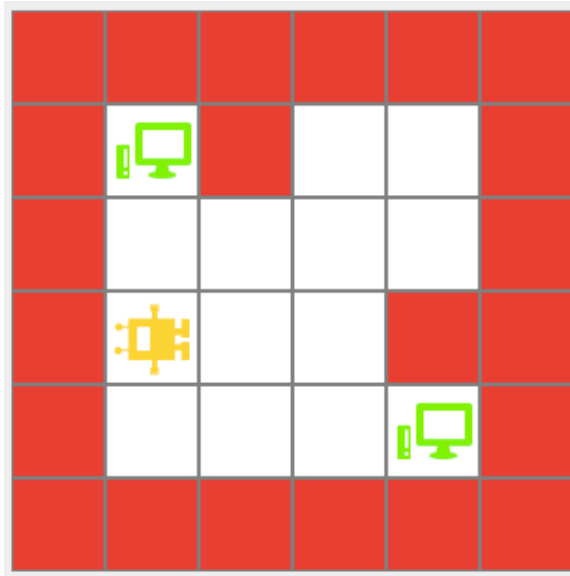


Figure 6.6 A map for Secret Code scenario

```

###
# There are two computer terminals. The first in the upper left corner
# contains a secret password. Use Karel and terminal operation to read
# the password from the first terminal and write it to the second
# terminal in the lower right corner.
#
###
program SecretCode defined as
  turn left
  repeat 2 times do
    step
  end repeat

  text val password = read text from terminal
  print password

  turn around

  repeat 3 times do
    step
  end repeat
  turn left
  repeat 3 times do
    step
  end repeat

  write text password to terminal
end program

action turn around defined as
  turn left
  turn left
end turn around

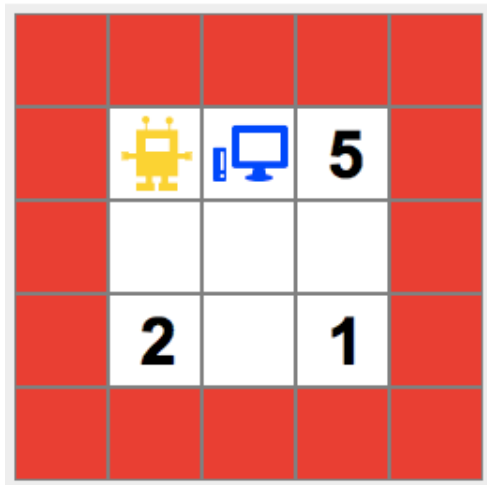
```

Figure 6.7 A solution of Secret Code scenario

## Marks Counter

The fourth scenario is for EddieFunctions language, and it is called *Marks Counter*. It introduces functions with return values. Figure 6.8 shows a map for this scenario; there are three cells with marks and numeric computer terminal. The count of the marks in each cell is randomly generated. The goal is to count all the marks and write their count into the computer terminal. In the end, the number of marks in the cells must be the same as was at the beginning.

Figure 6.9 shows an initial program for this scenario. It makes robot to visit each cell with the marks, call function `countMarks` there, add the result to a variable `sum`, and finally, write the `sum` into the numeric terminal. The task for users is to implement the body of `countMarks` function. Users need to pick all the marks, count each mark that is picked, and then put the same amount back. Figure 6.10 shows an example of implementation of this function.



*Figure 6.8 A map for Marks Counter scenario*

```

###
# In this tutorial we need to sum of all marks and write it into terminal.
# In the end there marks must be in at same places as in the beginning.
#
# There is a programm that visit all cells with marks and call function callMakrs.
# Your task is to implement this function. Have in mind, that after the function finishes,
# there should remain the same count of marks.
###
program MarksCounter defined as

    number var sum = 0

    repeat 3 times do
        step
        step

        sum = sum + countMarks ( )

        turn left
    end repeat

    step
    print "Sum: " + sum
    write number sum to terminal
end program

function countMarks
    with parameters:
        << ... >>
    with return type number
    defined as
        # TODO: implement function body

    return 0
end countMarks

```

Figure 6.9 An initial program for Marks Counter scenario

```

###
# Counts all marks by picking them and putting them back.
###
function countMarks
    with parameters:
        << ... >>
    with return type number
    defined as
        number var count = 0

        # pick all marks and count them
        while mark do
            pick mark
            count = count + 1
        end while

        # put the same amount back
        repeat count times do
            put mark
        end repeat

        return count
    end countMarks

```

Figure 6.10 A solution of Marks Counter scenario

## Karel 2.0

The fifth scenario aims to practise object-oriented programming and it is called Karel 2.0. The task is to create a new version of robot with some advanced commands as a singleton object. Scenario contains an initial singleton for improved robot. Figure 6.11 shows a fragment of this prepared singleton. It has prepared methods with empty body, and a user should implement these methods. The robot has the following methods:

- `doSteps(number count)`
- `turnLeft()`
- `turnRight()`
- `turnAround()`
- `putMarks(number count)`
- `pickMarks(number count)`
- `countMarks()` returns number
- `collectMarksInRectangle(number width, number height)`
- `writeStatistics()`

Method `collectMarksInRectangle` should collect all marks in a rectangular area in the map under the robot with given width and height. Robot should also count number of steps, turns, and marks that picked and put. Method `writeStatistics` should write all the statistics into a computer terminal in predefined text form. The solution should be well decomposed and properly documented.

```

singleton class Karel2 defined as
  fields:
    << ... >>

  constructor:
    constructor defined as

    end constructor

  methods:
    public method doSteps
      with parameters:
        number count
      with return type none
    defined as
      # TODO: implement body
    end doSteps

    public method turnLeft
      with parameters:
        << ... >>
        ...

```

*Figure 6.11 A fragment of skeleton for advanced robot singleton*

It is not possible to check by the means of the scenario, whether a user fulfil all the task. Some tasks must be evaluated by users themselves or by a teacher if there is any. The scenario contains a program that uses robot interface to complete several tasks in a large map. If a user implements all the methods of the robot correctly, the program successfully walks through the map and completes all the tasks there. The final task for users is to come up with their own ideas how to improve the robot and implement them.



# 7 Future Work

This chapter describes some ideas how to continue in developing Eddie and increase its educational impact. Ideas are divided into three groups: the language itself, the whole Eddie environment, and educational possibilities of Eddie.

## 7.1 Language

We have some other ideas that could be part of Eddie language, which we did not implement in the prototype, but we believe that they are also important to learn.

### Object Inheritance

The first idea is object inheritance. We considered two approaches for inheritance. The first one is standard class and interface inheritance as is in Java (version 7 and below) or C#. The second option is to have only classes. Classes could be abstract and have abstract methods. It would be possible to have multiple inheritance with linearization. This prevents multiple inheritance diamond problem, by specifying the exact order of ancestors.

The advantage of standard inheritance is that users would learn it in the same way as it is in traditional languages. The second approach is farer from the inheritance concepts of traditional languages (e.g. Java, C#), but these principles are used in some modern programming languages, whose popularity grows (e.g. Scala). It makes inheritance more simple, and users do not need to understand the difference between interfaces and abstract classes.

If Eddie supported inheritance, the Eddie map could be reimplemented fully in Eddie, and users could create a new objects or extends the current objects (e.g. extends wall to a magical wall that can be opened by entering secret code, or create teleport objects). Implementing Eddie map would also require an array or list, which are a part of the next suggested language extension.

### Collections

Collections are not only useful for Eddie map reimplementation, but they are also essential part of programming languages, so we believe people should learn about them. Eddie could support the following data structures: list, queue, stack, set, bag, and dictionary. The list is implemented as dynamic array indexed by numbers and it

serves also as a substitute for an array, the queue is standard first-in-first-out collection, the stack is last-in-first-out collection. The set represents a collection of elements without duplicities, where the keys for searching are the elements themselves. The bag is like the set, but allows duplicities. The dictionary is a key-value collection.

### **Constants**

Next feature that could be useful are global constants. They could be implemented as a new root concept called constants, which would contain a sequence of defined constants.

### **Documentation Comments**

Eddie allows adding a documentation block to programs, libraries, custom actions, functions, classes, singletons, and class members. In the language prototype, the documentation blocks are just unstructured multiline text. The documentation could be extended to document also parameters and return values of methods. Then, we could create a panel that would allow browsing the structure of created code and displaying these documentation comments as formatted text.

### **Debugging**

We would like also to introduce debugging in Eddie. MPS has support for custom languages debugging. It would require specifying which concepts of Eddie can hold breakpoints. It would use MPS debugger, which is similar to a debugger in IntelliJ IDEA.

### **Parallelism**

Next feature that could be implemented is parallelism and synchronization. Users could program two robots that would work together to complete some task. The robots could also interact with each other. We would create also language constructs for locking and the robots would need to manipulate with some resource that requires synchronization. There would be a special execution mode that would analyse the program, and if it found potential race condition or deadlock, it would execute parallel instructions of both robots with timing that would lead to this race condition or deadlock. This would allow users to experience behaviour that can occur in parallel environment, but its occurrence is quite rare.

### **Alternative Editors**

MPS Editor Hints allow creating alternative editors for a language. This is already used in editor with colour background described in Section 4.5.

We could create also an editor for the whole language that use a syntax similar to some standard programming language (e.g. Java, C#...). Users then could switch between this and the default editor. It could help to move from Eddie to some standard programming language.

### **Better Scene Definition Language**

Creating scenes for Eddie is not much user friendly. A user cannot see, how the map will look before she uses it in some program and run that program. We would like to use options of MPS projectional editor to create map definition as a table. Every cell of the table in projectional editor would correspond to a cell in Eddie Map definition. In each cell users can choose, what kind of object they want to place there. This would provide more intuitive way to create maps.

## **7.2 Environment**

This section describes ideas how to improve the environment of Eddie, mainly Eddie Panel and Eddie Studio.

### **Localization**

Eddie is prepared for localization as much as it is possible in MPS. In the future, there could be more supported languages, and also Eddie keywords could be translated into other languages. The best way, how to translate keywords, could be to suggest multilingual support as a new feature in the following versions of MPS. Otherwise, it might be implemented using an alternative editor for each language.

### **Program Preview**

When we decided to use generator approach instead of *interpreter*, we lost possibilities to implement some features. One of that features is immediate preview of executed code without generating and running the whole program. In the future a simple interpreter could be added. It would immediately display the path of the robot in the current program. The path would be visualised in Eddie Panel in the map. Interpreter could support only simple languages without variables, functions, or

objects. This would simplify the implementation. Program preview could help beginners to learn how to control the robot without waiting for program to be built.

Interpreter would also allow displaying some information directly in the editor. Program could highlight the line that is currently executed. If a program did only operations which are not visualized in Eddie Panel, it would be too fast and users could not see them, but we suppose that most of the operations would somehow manipulate with the map, so the execution of a line of code would be slow enough to see the highlighted line. Next information that could be displayed is dynamic state of the program. At the end of each line of the code, state of the program at that moment when the line was executed could be displayed. For example, there could be information about current position and direction of the robot. Lines that are executed more times in loops would have several columns to display the state for each iteration.

### **Objects Graphic Representation and Inspection**

Our prototype does not allow inspecting runtime state of objects. We identified this as one of the key features (item F15 in Section 2.3). If changes suggested in Section 7.1, to reimplement runtime map in Eddie, were done, Eddie objects could contain a description how they should be displayed in Eddie Panel. Objects could provide paint method, which could be overridden. Eddie Panel could also allow inspecting all object in the map and visualizing their internal state. When users click to any object in the map, it would show what kind of class it is, what fields it has, and the values of these fields.

### **Eddie Web Application**

In Section 4.1, it is discussed that educational tools provided as a web application are much more accessible, and what are the limitations of MPS in this issue. In long-term plan of MPS development, there is a goal to provide a web editor in future versions of MPS. If so, Eddie Studio could be transformed into a simpler web application IDE.

## **7.3 Education**

This section describes how to improve the impact of Eddie in the field of programming education in the future.

## **Evaluation**

It would be good to have some feedback from users before further development of Eddie. This could be done at some practical programming workshops with Eddie. Workshop participants would learn basics of programming with Eddie, and in the end, they would fill in evaluation forms.

## **Tutorial**

We prepared some examples of tutorial scenarios, which are described in Chapter 6. In order to learn programming using Eddie, complete tutorial is needed. There should be a sequence of scenarios would introduce every single concept and how the concepts work together.

In our prototype, we did not cover the feature to lead users to produce high quality code (item F14 in Section 2.3), so some scenarios should also focus on good coding practice.

## **Seminar Methodology**

Tutorial is useful in self-education, but for the situations where there is a trainer and a group of people, who wants to learn programming, there might be better options how to learn programming. In order to gain the most of these options, a methodology for a seminar could be prepared. It would describe a lecture plan for the trainer. The lecture plan would benefit from the facts that there is a group of participants, e.g. they can share ideas, they work in groups, or the trainer can work with motivation of the participants.

## 8 Conclusion

This chapter discuss how we fulfilled our goals. We explored eight existing educational programming tools. We compared their approaches, described their benefits and limitations, and identified their key features that, in our opinion, influence programming learning process.

We designed an educational DSL Eddie based on some key features of the existing tools. We choose domain of Karel the Robot, where users control a robot in a two-dimensional grid. This domain can be easily understood and visualised. We implemented a prototype of Eddie with IDE called Eddie Studio. Eddie Studio and all source files are on enclosed DVD. Content of the DVD is listed in Attachment B. We believe that Eddie can guide users from beginners to a level, when they can easily start to learn some standard programming language.

Eddie allows to do intuitive and safe first steps in programming due to the tutorial, intelligent code completion, syntax errors protection, and visualised program output. It is also close to standard programming languages in the way there is written code and similar language constructs.

Eddie offers guidance by the tutorial, but also a freedom in the way to solve the tasks or how to experiment with the program, after the task is completed. We believe Eddie is suitable for both self-educating individuals and groups with teachers. Teachers can design their own tutorials.

Eddie prototype still lacks some common features such as object inheritance or collections. Object inheritance could be added into EddieObjects language in its future versions. Other features can be implemented as language extensions.

# Bibliography

- [1] JetBrains s.r.o., *Meta Programming System*, [Online]. Available: <http://www.jetbrains.com/mps/>.
- [2] Lifelong Kindergarten Group in the Media Laboratory at the Massachusetts Institute of Technology, *Scratch*, [Online]. Available: <https://scratch.mit.edu/>. [Accessed 13th July 2014].
- [3] Code.org, *Hour of Code*, [Online]. Available: <https://studio.code.org/hoc/1>. [Accessed 13th July 2014].
- [4] R. E. Pattis, *Karel The Robot: A Gentle Introduction to the Art of Programming*, John Wiley & Sons, 1981.
- [5] Khan Academy, *Computer Programming*, [Online]. Available: <https://www.khanacademy.org/computing/computer-programming/>. [Accessed 13th July 2014].
- [6] M. Nemecek, *Peter*, [Online]. Available: <http://www.breatharian.eu/Petr/en/index.htm>. [Accessed 13th July 2014].
- [7] Microsoft Research, *Touch Develop*, [Online]. Available: <https://www.touchdevelop.com/>. [Accessed 13th October 2014].
- [8] M. Kölling and J. Rosenberg, *BlueJ*, [Online]. Available: <http://www.bluej.org/>. [Accessed 13th October 2014].
- [9] Computing Education Research Group at the School of Computing, University of Kent in Canterbury, *Greenfoot*, [Online]. Available: <http://www.greenfoot.org/>. [Accessed 13th October 2014].
- [10] JetBrains s.r.o., *IntelliJ Platform*, [Online]. Available: <http://www.jetbrains.org/pages/viewpage.action?pageId=983889>.
- [11] JetBrains s.r.o., *Open API - accessing models from code*, [Online]. Available: <https://confluence.jetbrains.com/display/MPSD32/Open+API+-+accessing+models+from+code>. [Accessed 20 5 2014].
- [12] JetBrains s.r.o., *Building an interpreter cookbook*, [Online]. Available: <https://confluence.jetbrains.com/display/MPSD32/Building+an+interpreter+cookbook>. [Accessed 20th May 2014].

# Attachments

## Attachment A      List of Karel the Robot

### Implementations

- <http://karel.oldium.net/> – Online tool
- <http://www.karel1981.cz/> – Application for Android and iOS
- <http://joymononline.in/apps/karel/karel.htm> – Online Karel Simulator in HTML5
- [http://is.muni.cz/th/365368/fi\\_m/](http://is.muni.cz/th/365368/fi_m/) – Diploma Thesis: Creation of Web Interpreter for Language Karel
- <http://karel.webz.cz/> – 3D version of Karel
- <http://xkarel.sourceforge.net/eng/> – 3D version of Karel
- <https://www.cs.mtsu.edu/~untch/karel/> – Standard implementation in C++
- <https://www.jetbrains.com/mps/> – Example implementation in MPS
- <http://karel.cloudmakers.eu/> – Application for iPad



## Attachment B      Content of Enclosed DVD

The enclosed DVD has the following directory structure:

- **application**
  - **Eddie StudioEddieStudio-143.SNAPSHOT-linux.tar.gz** – Eddie Studio for Linux
  - **EddieStudio-143.SNAPSHOT-macos.zip** – Eddie Studio for MacOS
  - **EddieStudio-143.SNAPSHOT.zip** – generic distribution of Eddie Studio, which can be used on Windows or Unix-based systems
- **documentation**
  - **Domain-Specific\_Language\_for\_Learning\_Programming.pdf** – this thesis in PDF format.
  - **Eddie\_Studio\_Manual.html** – short description of Eddie Studio and installation manual
  - **Build\_Manual.html** – description how to build Eddie Studio from sources
- **sources**
  - **eddie-gui** – source code of Eddie Panel
  - **eddie-lang** – source code of Eddie languages created in MPS
  - **eddie-tutorial** – MPS project with the tutorial that we created