AZ-NUGGETS: AN IDE FOR PROGRAMMING BY CONCEPT

by

L. Deepak Yalamanchili

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
July 2013

© Copyright by L. Deepak Yalamanchili, 2013

Table of Contents

List of Figures iv

Abstract ix

List of Abbreviations Used x

Acknowledgments xi

Chapter 1 Introduction 1

1.1 Unintuitive representation 1

1.2 Hidden semantics 2

1.3 Hidden language features 4

1.4 Objectives 5

1.5 Audience 5

Chapter 2 Related Work 7

2.1 Scratch 7

2.2 App Inventor for Android 9

2.3 AgentSheets 11

2.4 Alice 15

2.5 BlueJ 17

2.6 Greenfoot 20

2.7 CSmart 23

2.8 Grammar Cells 24

2.9 Eclipse 26

Chapter 3 Az-Nuggets 28

3.1 Nuggets 28

3.2 Overview 33

3.3 Programming by Concept with Nuggets 35

3.4 Exploration and Presentation of language details 37

3.5 Unfolding Semantics 41

3.6 Putting it altogether 49

Chapter 4 Az-Nuggets: An Application Framework . . . 73

4.1 Overview 73

4.2 Tentative Guidelines 75

4.3 Summary 83

Chapter 5 Conclusion 85

5.1 Future Work 85

Bibliography 88

List of Figures

Figure 2.1 A snapshot of the Scratch Tool. 8

Figure 2.2 The Design view of App Inventor. 9

Figure 2.3 The Block editor of App Inventor (a) Categories in the Block editor (b) Viewing the blocks under a category. 10

Figure 2.4 A rule that defines the action of the Pacman upon cursor-up key (from p05 of [20]). 11

Figure 2.5 Tools in AgentSheets for building domain oriented visual environments (from p67 of [21]). 12

Figure 2.6 Conversational Programming architecture of AgentSheets (from p01 of [18]). 13

Figure 2.7 Demonstration of Conversational Programming (from p02 of [19]) (a) Choosing the ‘frog’ agent in a worksheet (b) Annotated conditions of the selected ‘frog’ agent in the worksheet. 14

Figure 2.8 A snapshot of Alice. 16

Figure 2.9 Java support in Alice 3 (from p03 of [5]) (a) A code snippet in Alice decorated with Java syntax (b) Preference to switch between Alice and Java. 16

Figure 2.10 The main screen of BlueJ. 17

Figure 2.11 Creating an instance of a class by executing its constructor. 18

Figure 2.12 The Object Workbench of BlueJ displaying objects created. 18

Figure 2.13 Interacting with an object from its context menu. 19

Figure 2.14 BlueJ’s code editor. 19

Figure 2.15 The main window of GreenFoot. 21

Figure 2.16 The code editor of GreenFoot. 21

Figure 2.17 Context menu of a class in GreenFoot. 22

Figure 2.18 Greenfoot world hosting GreenFoot objects. 22

Figure 2.19 Annotated code in (from p03 of [6]). 23

Figure 2.20 Information Pane in CSmart (from p03 of [6]). 23

Figure 2.21 Visualizing the printf function of (from p03 of [6]). 24

Figure 2.22 Visualizing an arithmetic operation in C (from p04 of [6]). 24

Figure 2.23 Grammar cells of a language (from p03 of [12]). 25

Figure 2.24 Properties of a constant control (from p03 of [12]). 25

Figure 2.25 A Java class creation wizard in Eclipse. 26

Figure 3.1 Nugget used to create a pointer `p` to an integer `i` (a) A nugget used for declarations in C (b) Pointer `p` initialized to point to an integer variable `i`. 29

Figure 3.2 Nugget used to declare a pointer `fptr` to point to a function of a specific type (a) A nugget showing declaration of a function pointer (b) Declaring `fptr` to point to a function of a specific type. 30

Figure 3.3 A generic *class* nugget. 31

Figure 3.4 A generic nugget to provide single or multiline comments. 33

Figure 3.5 Programming and natural language selection screen. 34

Figure 3.6 A tentative depiction of the layout of the main screen of Az-Nuggets. 34

Figure 3.7 Conceptually working with a pointer variable (a) Context menu displaying the contextual information associated with a pointer variable (b) A wizard displaying the declarative properties of `ptr` (c) Selecting a contextual action, *cast to type* ..., to typecast `ptr`. 36

Figure 3.8 Reverting the access modifier in C++ conceptually (a) A hint icon on `myVar` (b) The context menu of `myVar` guiding the user to interact with *visibility* property (c) The *visibility* wizard pointing to the language feature. 37

Figure 3.9 Conceptually working with a generic class in C++ and exploring its features (a) Creating a generic class from a regular class (b) A wizard that lets the user specify generic parameters (c) Adding a generic parameter `X` that accepts a type defaulted to `int` (d) Wizards to configure each parameter. 39

Figure 3.10 The Rules context menu of `ptr`. 41

Figure 3.11 Static, instantiated and contextualized rules of a naming an identifier in Java (a) Viewing the rules for a valid identifier (b) Feedback provided on an invalid identifier (c) Instantiated rules (d) Contextualizing a rule. 42

Figure 3.12 Static and instantiated rules for method overriding in Java (a) Selecting *rules to override* from the context menu of a *method* nugget (b) Viewing the rules for overriding a method (c) Instantiated rules with a contextualized violated rule. 43

Figure 3.13 Instantiated rules for compile-time call binding (a) The rules-context menu of method call (b) Instantiated and contextualized rule. 45

Figure 3.14 Instantiated rules for Overriding; Contextualizing one of the relevant rules. 46

Figure 3.15 Exploring semantic rules of exceptions associated with methods in Java (a) Rules context menu of `method` of `Interface3` (b) Contextualizing the rule to understand why `method` of `Interface3` cannot throw any exceptions. 47

Figure 3.16 Instantiating and contextualizing rules for runtime call binding (a) Pointer `basePtr` pointing to an object of class `SubChildClass` (b) Rules context menu associated with function call (c) Instantiated rules associated with call binding are also shown. 48

Figure 3.17 Az-Programming editor displaying *package*, *import* and *class* nuggets. 51

Figure 3.18 A way of entering a data member in a class. 51

Figure 3.19 Data member creation wizard. 52

Figure 3.20 The definition of the data member `p`. 52

Figure 3.21 The data member `q` as defined by the user (a) The user typing in the definition of `q` (b) The definition of the data member `q`. 53

Figure 3.22 The Nugget Appearance Configurator window. 53

Figure 3.23 The appearance of *class* nugget A after configuring borders with available options (a) The appearance of *class* nugget A without thick borders and with connectors (b) The appearance of *class* nugget A without thick borders and connectors. 54

Figure 3.24 Inserting an init-block and initializing the data members in it (a) Entering an init-block in a class (b) Initializing p & q in the init-block. . 55

Figure 3.25 Class A in the editor. 55

Figure 3.26 Class B in the editor. 56

Figure 3.27 Creating an instance of an inner class by concept. 57

Figure 3.28 A non-modal message window that is displayed on conceptually creating an instance of a class. 58

Figure 3.29 The user pointing the mouse at the location where the instance has to be created. 59

Figure 3.30 The user clicking at the location where the instance has to be created that brings up the *inner class instance creation* wizard. 60

Figure 3.31 The user creating an instance of the outer class B. 61

Figure 3.32 The *instance creation* wizard. 62

Figure 3.33 Creating an instance of the inner class C. 63

Figure 3.34 Selecting *about* from the context menu associated with class C. 64

Figure 3.35 Details of inner class C. 65

Figure 3.36 Selecting *highlight scope* from the context menu associated with a data member. 66

Figure 3.37 The scope of p highlighted in green. 67

Figure 3.38 The scope of localVar highlighted in green. 68

Figure 3.39 Selecting the *resolving rules* entry from the Rules context menu of System. 69

Figure 3.40 Instantiated and contextualized rules for resolving `java.lang.System`. 70

Figure 3.41 Selecting the *resolving rules* entry from the Rules context menu of `D`. 71

Figure 3.42 Instantiated and contextualized rules for resolving class `D`. 72

Figure 4.1 An abstract layout of the Editor Suite. 74

Figure 4.2 A snapshot of the Editor Suite with the nugget designed to represent a Java class. 76

Figure 4.3 Specifying context menu entries using the Context Menu Specifier. . . 77

Figure 4.4 The code editor displaying the added event handlers. 78

Figure 4.5 Specifying entries of the Rules context menu using the Rules Context Menu Specifier. 79

Figure 4.6 Specifying the entries for the Constructs context menu using the Context Menu Specifier. 80

Figure 4.7 Specifying class resolving rules and their Instantiators and Contextualizers. 81

Figure 4.8 Adding *class* nugget to Nuggets Panel using the Nugget Panel Configurator. 82

Figure 4.9 The Programming Interface displaying the *class* nugget in the nuggets panel. 83

Figure 4.10 Dragging and dropping the *class* nugget onto the Programming Editor and interacting with it to bring up its Constructs context menu. . . . 83

Abstract

Contemporary Integrated Development Environments (IDEs) offer minimal or no features that allow programmers to explore details of a programming language or interact with the program elements at a conceptual level. Programmers have limited means of identifying and contextualizing relevant syntactic or semantic rules. Az-Nuggets attempts to address these limitations by facilitating programming-by-concept and allowing programmers to access, instantiate and contextualize syntactic and semantic rules.

List of Abbreviations Used

IDE	Integrated Development Environment

Acknowledgments

I would like to thank my supervisor, Dr. Philip T. Cox, for his tireless support, endless patience and invaluable feedback. I would also like to thank all my friends for their moral support and invaluable friendship.

A special thanks to Michael Hackett, Ali Daniyal and Faisal Abbas who always had time to have technical discussions with me and never said no. I would always cherish the discussions I have had with them.

Out of all, I am indebted to my parents for *everything*.

Chapter 1

Introduction

An Integrated Development Environment (IDE) is a software application that provides software development tools required for computer programmers under one umbrella. An IDE normally includes a source code editor, build automation tools, a debugger and the like. Well known contemporary IDEs such as Eclipse, Visual Studio and NetBeans include many such useful tools integrated into their development environments. In the context of programming, these IDEs offer assistive features such as syntax highlighting, auto-formatting, project outlining, refactoring and code completion among many others. Despite these features, these IDEs share common limitations which are as follows.

1.1 Unintuitive representation

A programming language's syntax is not necessarily a straightforward representation of the concepts it presents. Consider the following declarations and their semantics from different programming languages.

1) Pointer declaration in C

   ```
   int *p = &i;
   ```

   This statement declares an integer pointer `p` initialized to point to an integer variable `i`.

2) Function pointer declaration in C

   ```
   int * (*fptr) (char, int (*) ( )) ;
   ```

   This statement declares `fptr` as a pointer to a function that returns an integer pointer and takes two parameters: a character and a pointer to a function that takes no arguments and returns an integer.

3) Pure virtual function declaration in C++

```
virtual int getValue() = 0;
```

This statement declares `getValue` as a pure virtual function or an abstract function that has no body and is meant to be defined by a non-abstract derived class.

In these examples, the syntax does not intuitively convey the semantics. Programmers cannot express these concepts if they do not know or remember the syntax of a language. They are expected to deal with the textual syntax of a language as contemporary IDEs offer no features that allow them to program at a conceptual level. Well known features provided by these IDEs to help in writing code including colored syntax highlighting, user-defined code formatting, context-sensitive code assistance and the like are all based on the syntactical correctness of the statement they act upon.

1.2 Hidden semantics

Knowledge of semantic rules is pivotal for the correctness of programs. Despite the well-defined semantics of programming languages, programmers are not presented with essential semantic details in a way that enhances their understanding of a language. Contemporary IDEs expect programmers to know the semantics of a language and mostly offer a 'reactive' approach of providing semantic information rather than a 'proactive' approach that does not lend itself to allowing the programmers to explore the syntactic or semantic details of a programming language. Consider the following few of an endless list of examples which identifies the need for an IDE to provide access to such information.

1) Scope of different types of variables in Ruby

The scope of a variable is the context within which it is defined. Information about the scope of a variable is not provided in contemporary IDEs until the variable is used in a context out of its scope. For example, Ruby has four types of variables, each with its own scope. Programmers new to Ruby have no way of identifying the scope of these variables without having to type them at random places in the code to get semantic feedback from the IDE or referring to the Ruby programming manual.

2) __init__.py files for importing packages in Python

One of the important requirements when importing packages in Python is to include a file called __init__.py in each directory named within the path of the package import statement. Since the semantics of package importing is different in Java and Python, a Java programmer would be oblivious to such requirements in Python without the IDE providing access to such information.

3) Exception declaration semantics in Java

Consider the following example in Java which could be misleading in its exception declaration semantics.

```
package org.sample;

import java.io.IOException;

interface Interface1
{
	void method() throws IOException;
}

interface Interface2
{
	void method() throws InterruptedException;
}

interface Interface3 extends Interface1, Interface2
{
}

public class Example implements Interface3
{
	public void method()
	{
	}

	public static void main(String[] args)
	{
	}
}
```

A commonly known semantic rule in Java is that an overriding method must not throw checked exceptions that are new or broader than those declared by the overridden method. According to this semantic rule, it might appear legal for the method `method` defined in class `Example` to throw either `IOException`, or `InterruptedException` or their sub-

types. However, the semantic rule requires that the set of checked exceptions that a method can throw is the intersection of the sets of checked exceptions that it is declared to throw in all applicable types but not the union. This implies that `method` defined in the `Example` cannot throw any checked exceptions. Such context-based semantic rules cannot be proactively presented to the programmers in contemporary IDEs.

4) Semantics of method overloading in Java involving widening, autoboxing, and var-args

Consider the following code snippet in Java to identify the method that would be invoked upon the call `method(x)` and the semantics behind its invocation.

```
package org.sample;

public class Third
{
	static void method(Integer x)
	{
		System.out.println("Integer");
	}

	static void method(long x)
	{
		System.out.println("long");
	}

	public static void main(String[] args)
	{
		int x = 5;
		method(x);
	}
}
```

The code snippet outputs 'long' because the Java compiler prefers widening over boxing and var-args. Programmers unaware of this semantic rule cannot be sure of why the definition of `method(long x)` is invoked upon the call `method(x)`.

1.3 Hidden language features

Contemporary IDEs do not allow programmers to explore language details at a conceptual level. Programmers oblivious of contextually relevant language features have no way of knowing their existence from within the IDE.

1) Restoring visibility of data members in private inheritance in C++

In C++, all the public and protected members of a parent class are inherited as private members of the derived class when the base class is inherited in private mode. However, the original access specifications for these data members can be restored by using access restore declarations on these members within the derived class. Such a possibility is not hinted at in contemporary IDEs and would therefore go unnoticed by any C++ programmer who is unaware of this language feature.

2) Using default and non-type arguments with generic classes in C++

In C++, default and non-type arguments can be associated as part of the template specification for a generic class. Programmers who are familiar only with a basic template declaration for a class would be unaware of these options even when working with generic parameters in the IDE.

1.4 Objectives

The main objective of Az-Nuggets is to address the limitations mentioned in sections 1.1, 1.2 and 1.3 by offering the following features to the programmers:

- Programming-by-concept
- Exploration/Presentation of language details at a conceptual level
- Inference of syntactic and semantic rules

1.5 Audience

Az-Nuggets is intended to address the aforementioned limitations of contemporary IDEs. It is proposed to be used by programmers in general, irrespective of their expertise levels. However, it is mostly aimed at intermediate programmers who are familiar with programming languages at a conceptual level and are willing to further explore them from within the IDE.

Although Az-Nuggets is yet to be implemented, for the sake of simplicity its features will be explained as if it has been implemented.

Chapter 2

Related Work

This chapter focuses on previous work done on the conceptual programming features offered at different levels of abstraction for domain specific and general purpose programming languages in various IDEs and programming tools. The level at which semantic information is presented to the user in each of these tools is also discussed. This chapter includes a brief description of each related tool/IDE and an analysis of its features in this regard.

2.1 Scratch

Scratch is a visual programming environment that allows users to learn computer programming to create interactive, media-rich projects [14]. A key goal of Scratch is to introduce programming to those with no previous programming experience. It provides a set of general computational concepts such as sequences, loops, parallelism, events, conditionals, operators, and data that can be found in many programming languages. Each concept is provided with a definition and a concrete example from a Scratch project [2]. Programming in Scratch is based on a building-block metaphor against a background called the "stage" to control sprites by snapping colorful command blocks together, which allows users to build scripts much like putting together pieces in a jigsaw puzzle [15]. Users can also import images and sounds, or use a sound recorder and a built-in paint tool to create custom sounds and sprites that can be programmed in the same way. Scratch evolves with every release by adding new features and commands to offer more functionality to its users [14].

The default Scratch interface is a single window that has four panes as shown in Figure 2.1. The top-left pane contains categories and the left pane displays command palettes

corresponding to an active category. The middle pane is the script editor where the users can define behavior for a selected sprite. The top-right pane is where the sprite actions take place as a result of executing the script. The currently active sprite, and a collection of thumbnails of all the sprites in a project, are displayed in the bottom-right pane.

Scratch is aimed at programmers with no previous programming experience by providing simple blocks with intuitive visual syntax that allows these blocks to be snapped together to compose programs. Each block is associated with static help that explains its meaning and usage. Offering more functionality in Scratch involves adding more blocks to its repertoire of blocks, a process which does not scale well with the tool. The goal with every release of Scratch is to keep the number of blocks under a certain threshold value to allow programmers to navigate and find them easily.

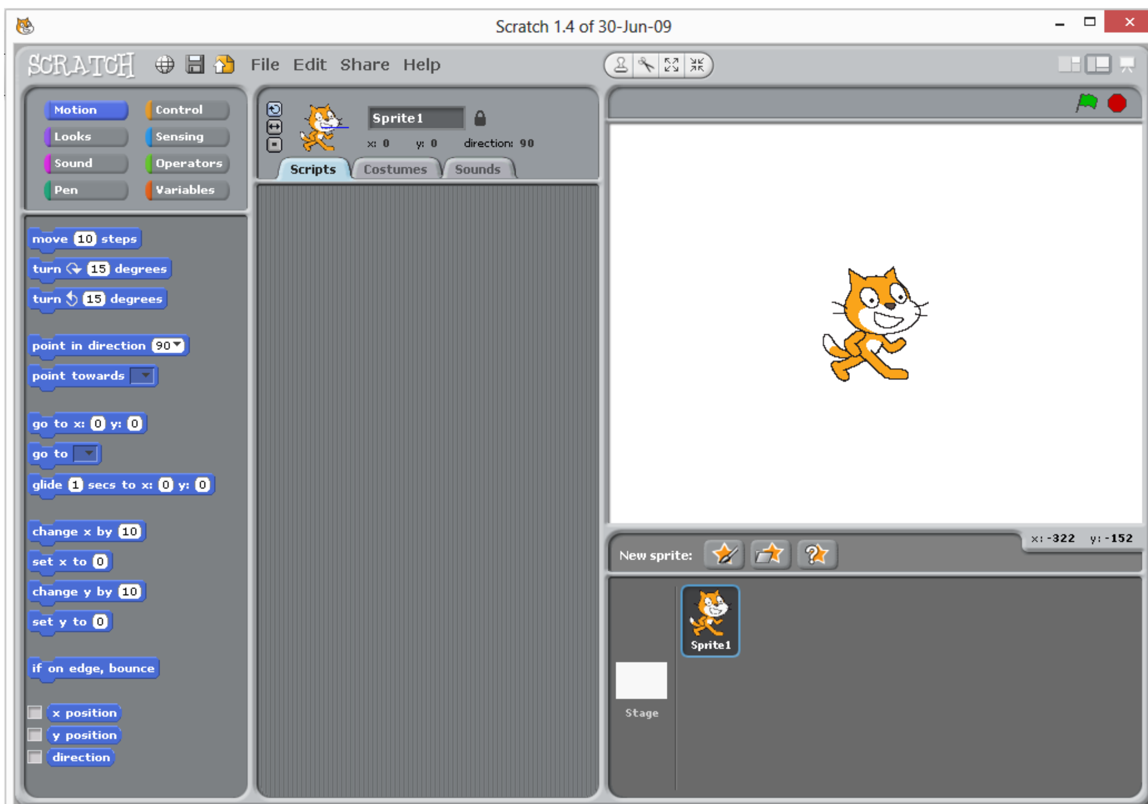


Figure 2.1: A snapshot of the Scratch Tool.

2.2 App Inventor for Android

App Inventor for Android is an application from Google that allows users to design and program Android applications with a Web page and Java interface. App Inventor uses a graphical interface that allows users to drag-and-drop visual objects to create Android applications. Its visual programming framework is similar to that of the Scratch programming language [13]. Simple Android applications can be created with very little programming knowledge and more complex and powerful applications can be created with continuing experience [23].

The App Inventor application programming environment consists of the Design view and the Blocks editor. The design view as shown in Figure 2.2 consists of the five columns, Palette, Viewer, Components, Media and Properties. The Palette column consists of all the components that can be used in a project whereas the Components column displays all the components added to a project. A component can be made active by selecting it from the Viewer column that would then display its properties in the Properties column.

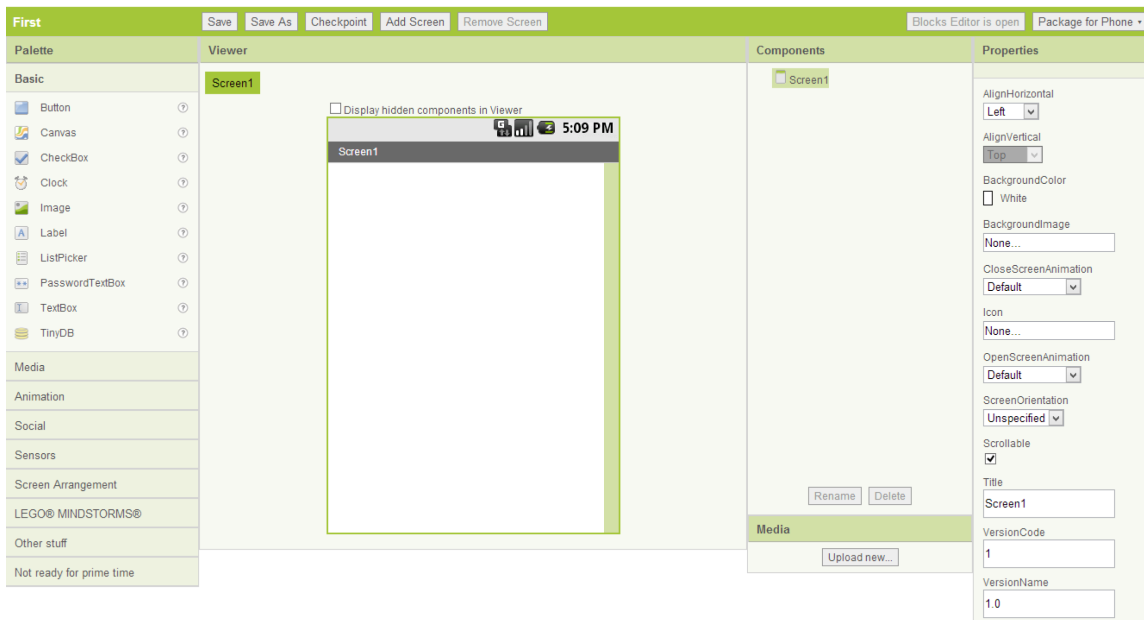
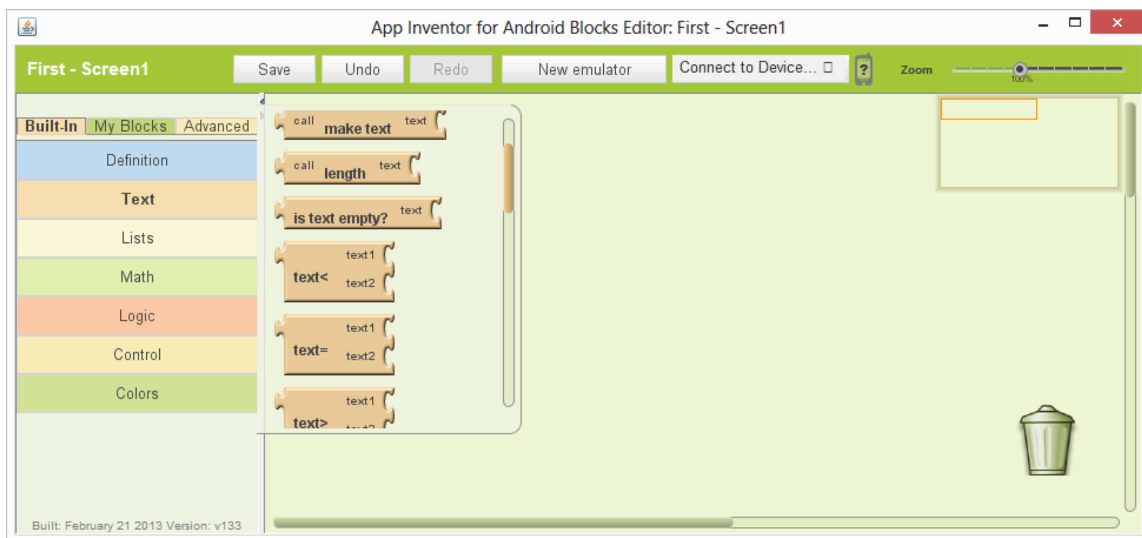


Figure 2.2: The Design view of App Inventor.

Figure 2.3a shows the Block editor of the App Inventor, which looks similar to Scratch in its user interface. Users can find main categories in the left pane, each of which contains a set of command blocks. Clicking on a category displays its set of command blocks in an extension pane as shown in Figure 2.3b.

(a) Categories in the Block editor



(b) Viewing the blocks under a category

Figure 2.3: The Block editor of App Inventor.

The blocks offered by App Inventor are specific to Android application development. These blocks are simple and self-explanatory just like the blocks offered in Scratch. App Inventor suffers from the same drawbacks as Scratch when more functionality is offered which increases the number of blocks to manage.

2.3 AgentSheets

"AgentSheets is a visual design and programming environment that allows users to create agent-based simulations, visualizations and applications" [16]. Agents in AgentSheets are programmed using a rule-based tactile visual programming language called Visual AgenTalk which offers graphical user interface items for the language components to minimize the problems caused by the textual syntax in traditional programming languages. Commonly known graphical user interface elements like pop-up menus, checkboxes, text fields are used to represent the rule components, conditions and actions [16]. Figure 2.4 shows one of the many rules that define the action of the Pacman when the cursor-up key is pressed. It sets a new direction for the pacman and changes its depiction to face up.

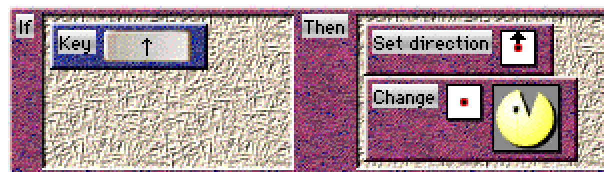


Figure 2.4: A rule that defines the action of the Pacman upon cursor-up key (from p05 of [20]).

Figure 2.5 shows some of the task-specific specializations that are part of the AgentSheets environment. The icon editor in the top left corner is used to edit the look of the agents. The worksheet in the top right is used to create an environment by selecting agents in the gallery and placing them in the worksheet. The worksheet is a grid and every agent placed in the worksheet occupies exactly one grid square. The gallery in the lower left serves as a palette of agents whereas the class browser shown in the lower right is used to inspect the agent class hierarchy [21].

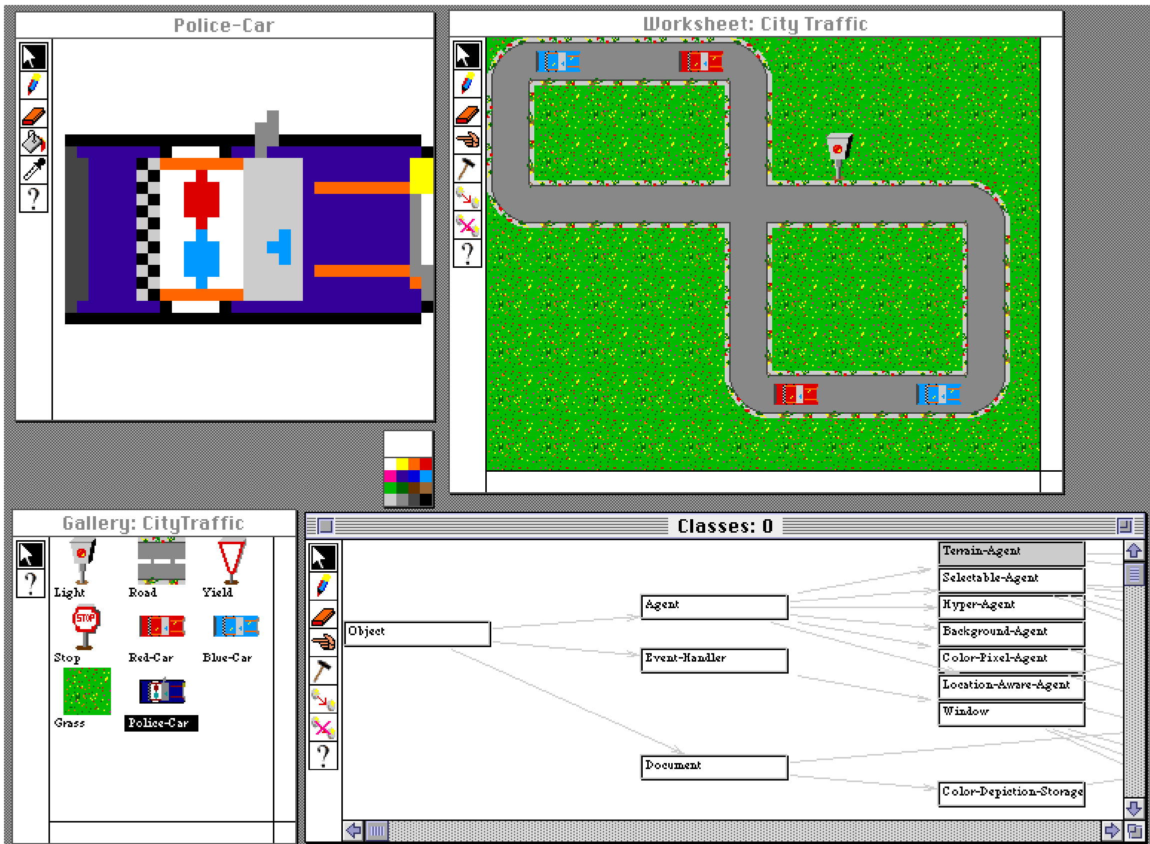


Figure 2.5: Tools in AgentSheets for building domain oriented visual environments (from p67 of [21]).

Conversational programming of AgentSheets employs computational agents to execute parts of a program to provide real-time semantic feedback to a programmer. It is found to be effective because of its proactive nature that displays the outcome of the test of a condition without the need for an execution step and its high degree of parallelism that annotates all relevant code in real time. Figure 2.6 shows the Conversational Programming architecture of AgentSheets in which a conversational programming agent executes a program to annotate it semantically and interpret the situation [18].

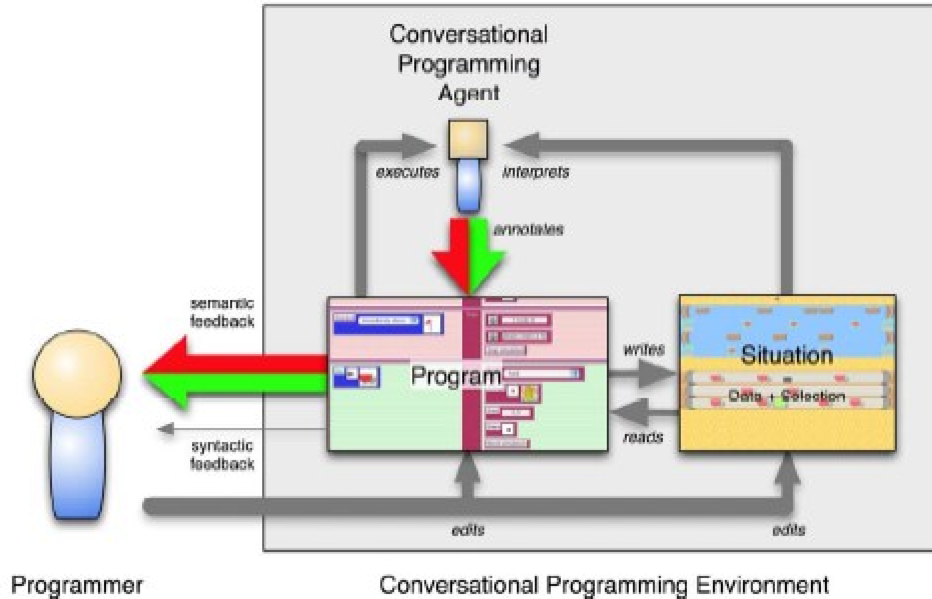
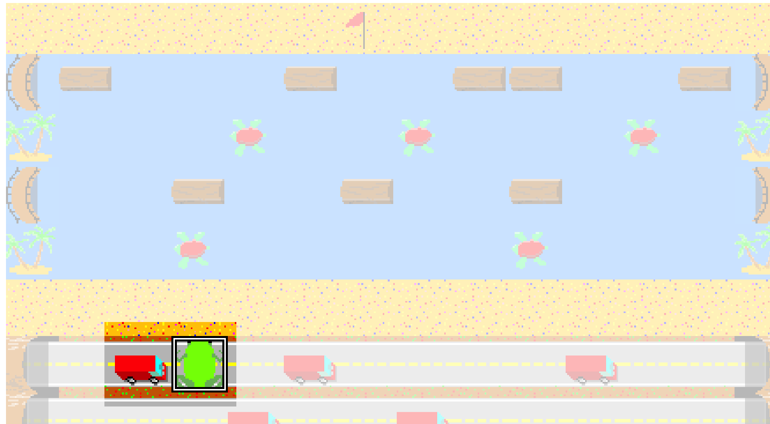


Figure 2.6: Conversational Programming architecture of AgentSheets (from p01 of [18]).

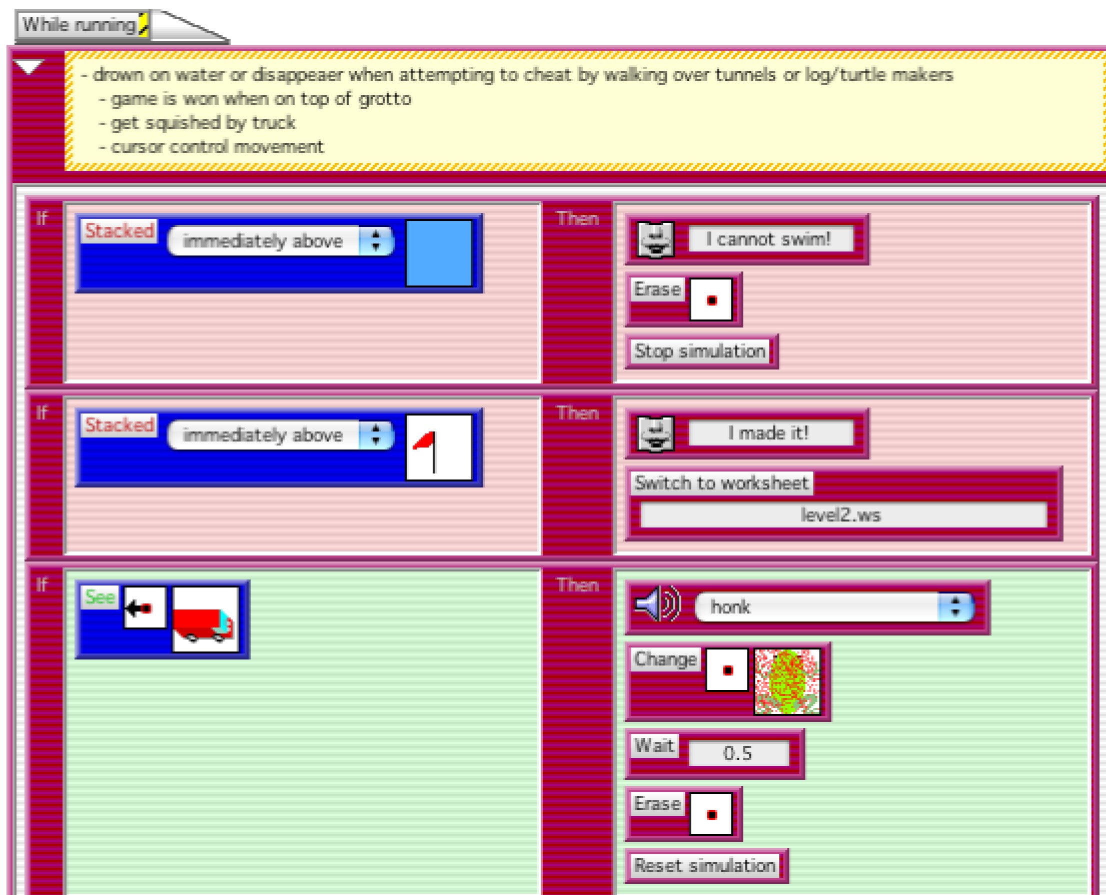
A conversational programming agent visualizes the outcome of running the program of a selected agent one step into the future. Figure 2.7 illustrates the functionality of a conversational programming agent. Figure 2.7a shows a worksheet designed to simulate a frog crossing a street and a lake to get to the flag. The truck agents shown in the figure are defined to move along the streets. Figure 2.7b shows the rules that define the behavior of the frog agent in the simulation. The first rule checks if the frog agent is on top of the water agent which ends the simulation displaying a message that the frog cannot swim. The second rule checks if the frog agent is on top of the flag agent which concludes the current level and loads another level. The third rule checks if the truck agent is immediately to the left of the frog agent, in which case the truck horn honks and the truck runs over the frog killing it. The last rule defines the behavior of the frog agent on pressing the left-arrow key.

The functionality of a conversational programming agent acting on a selected 'frog' agent in Figure 2.7a is shown in Figure 2.7b where the outcome of conditions that apply on the selected frog agent is visualized. The conditions for first and second rules are annotated in red and the one for third rule is annotated in green. Green denotes a rule which, in cur-

rent circumstances, will be executed, since its condition is satisfied, while red indicates a rule that will not be executed.



(a) Choosing the 'frog' agent in a worksheet



(b) Annotated conditions of the selected 'frog' agent in the worksheet

Figure 2.7: Demonstration of Conversational Programming (from p02 of [19]).

The visual blocks offered by Visual AgenTalk are similar to the blocks offered in Scratch and App Inventor in their simplicity and straightforwardness. AgentSheets provides the user with helpful semantic guidance of applicable rules of a selected agent. However the semantic guidance provided is on the rules specified by the user rather than the Visual AgentTalk language itself.

2.4 Alice

Alice is a programming environment designed to help novices understand the concepts of object oriented programming through interactive three-dimensional virtual worlds. "It was designed by studying how novices try to describe the motions of objects in a 3D world, and then modifying Alice to reflect these observed expectations" [17]. It provides a programming environment where the users can create animations by programming 3D objects. Programming is done with a drag-and-drop smart editor with the emphasis on visual formatting of the code blocks. Studies conducted on its usability have shown that the gain from this visual editing mechanism is a reduction in complexity where the subjects focused on the concepts of objects and encapsulation rather than dealing with the frustration of textual syntactic nuances. Additional features, including the visualization of program execution, enable students to gain immediate feedback by allowing them to correlate individual programming statements with the behavior of the objects in the virtual world.

Figure 2.8 shows a snapshot of Alice. The upper left panel in the interface displays a set of objects in the current world that is displayed in the inset located in the upper center. The bottom left panel allows users to work with the properties of the objects and their corresponding methods that could be dragged to the code editor shown in the lower right [3].

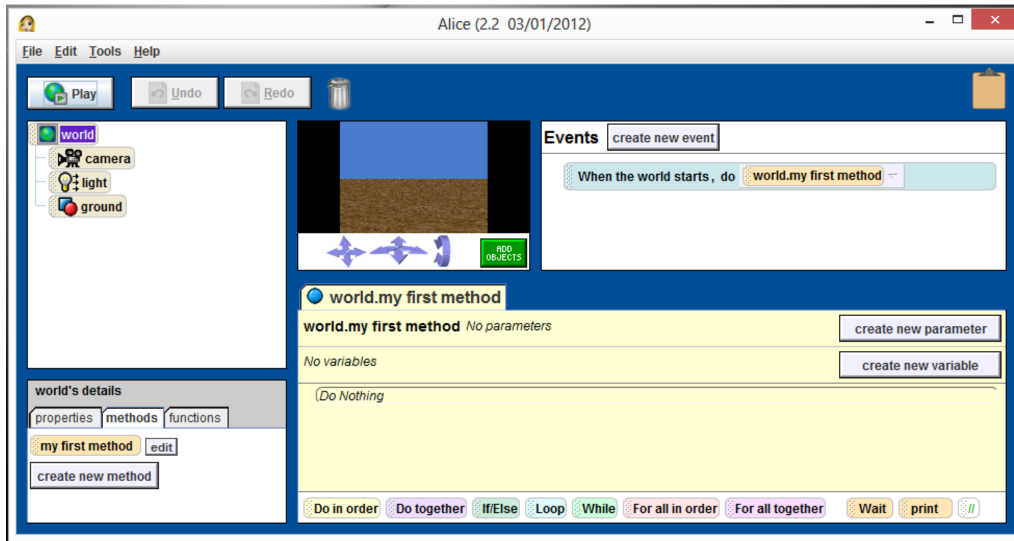
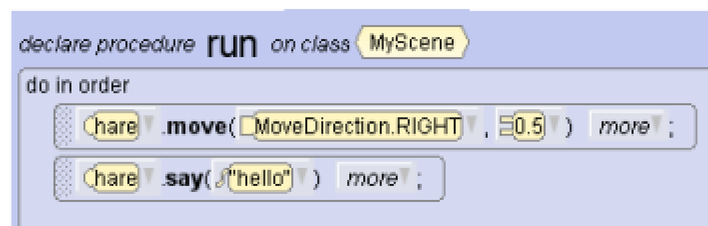
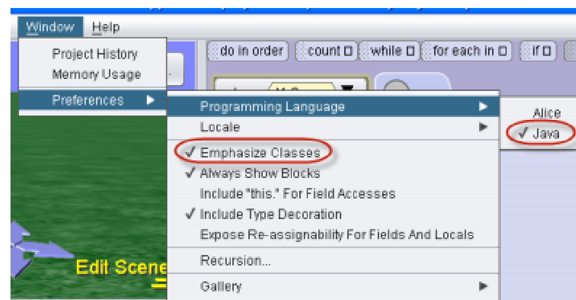


Figure 2.8: A snapshot of Alice.

Alice 3 provides options that allow the code in the code editor to be decorated with Java syntax to help users get familiarized with the syntax details of Java. Figure 2.9a shows a snippet of code in the code editor decorated with Java syntax with necessary parentheses, quotes, commas and semicolons. Figure 2.9b shows the option for switching between Alice and Java.



(a) A code snippet in Alice decorated with Java syntax



(b) Preference to switch between Alice and Java

Figure 2.9: Java support in Alice 3 (from p03 of [5]).

Alice does not offer features that the programmers can explore to view the syntax or semantic rules of the Java programming language. Rules cannot be inferred in the context of a program in a way that they can be contextualized on request by the programmer. Elements in a program are not associated with contextual information that allows programmers to interact with them at a conceptual level.

2.5 BlueJ

BlueJ is an IDE for the Java programming language, developed mainly for teaching introductory object-oriented programming [1]. It facilitates the discussion of object-oriented design with an "objects first" methodology [11].

BlueJ offers a text-based Java IDE which presents the class structure of an application graphically as a UML diagram. Figure 2.10 shows the main screen that displays the class structure of an application under development. Double clicking on a class brings up a text-based code editor as shown in Figure 2.14 that allows users to write code for the class. Users can create objects by interacting with a class icon and executing its constructor as shown in Figure 2.11. The objects created are added to the object bench located at the bottom of the main window as in Figure 2.12.

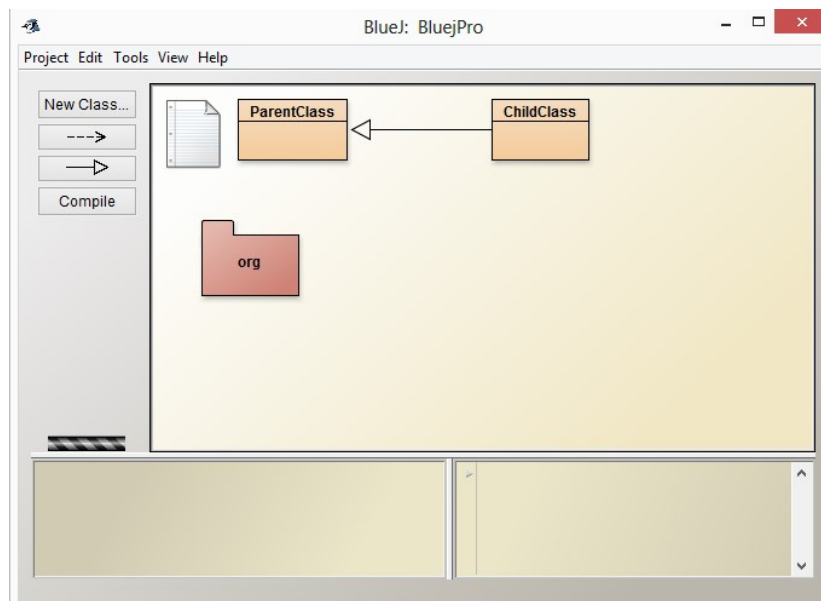


Figure 2.10: The main screen of BlueJ.

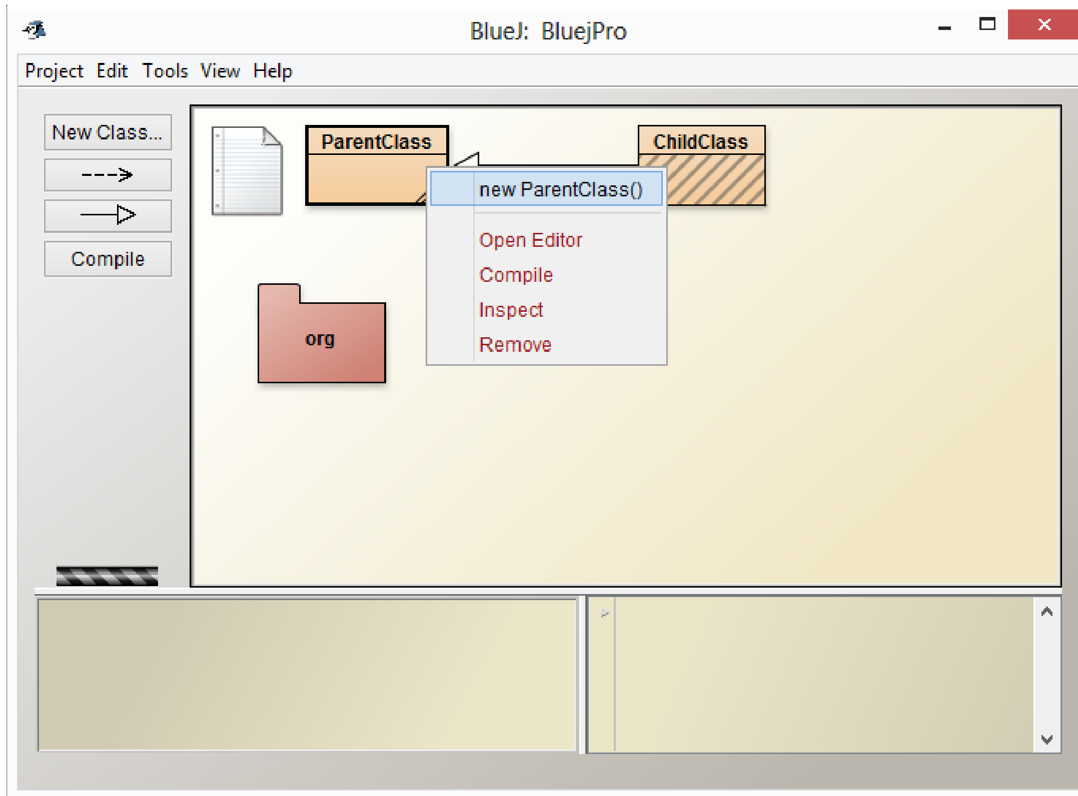


Figure 2.11: Creating an instance of a class by executing its constructor.

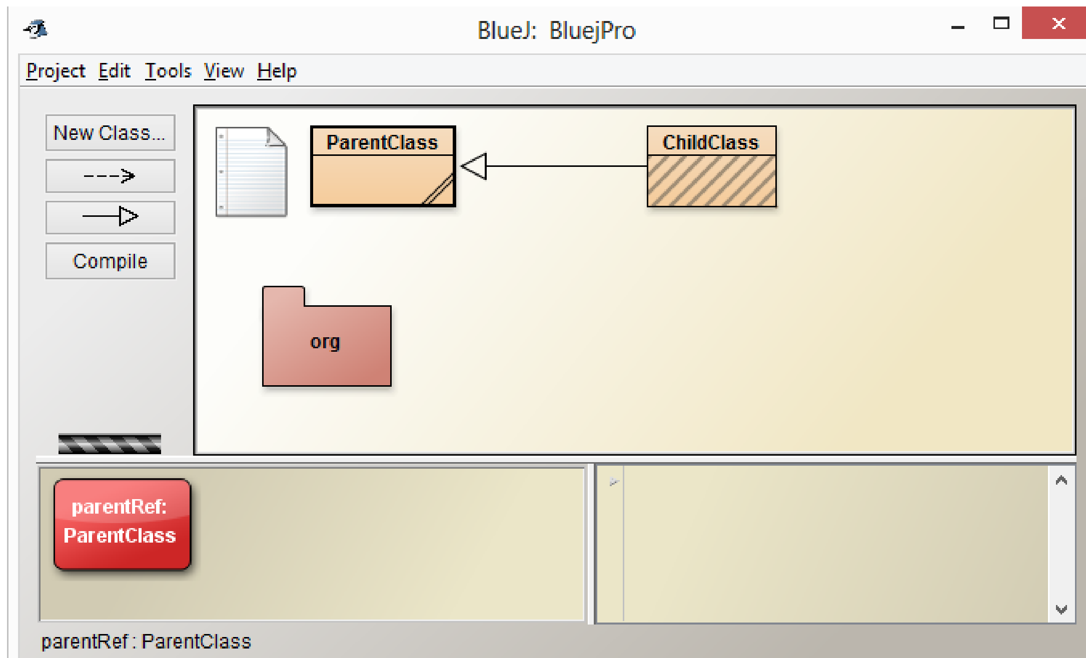


Figure 2.12: The Object Workbench of BlueJ displaying objects created.

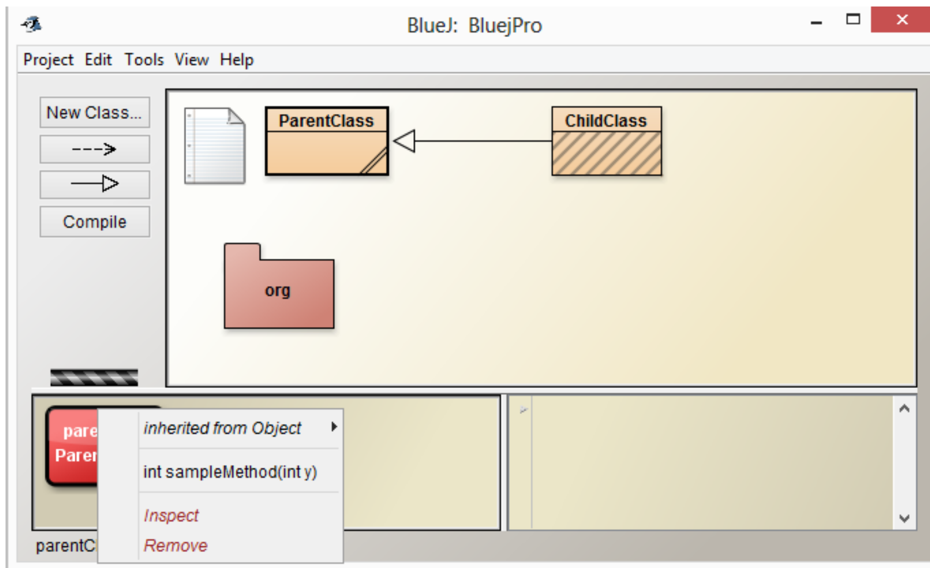


Figure 2.13: Interacting with an object from its context menu.

Users can also interact directly with objects from the object bench by right-clicking on an object of interest and bringing up its context menu that contains options related to object interaction. As shown in Figure 2.13, the context menu contains an entry for each public method defined on an object in addition to options that allow objects to be inspected and removed. The code editor of BlueJ is shown in Figure 2.14 [10].

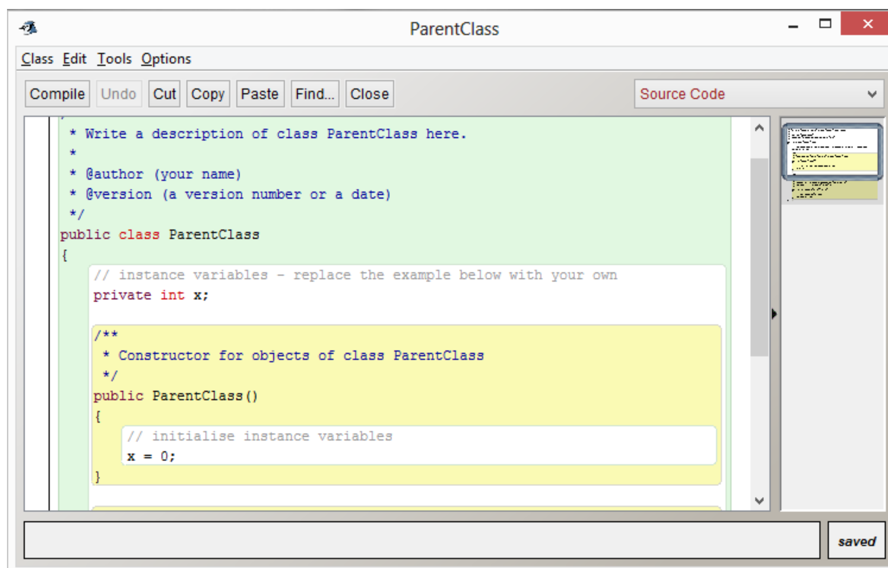


Figure 2.14: BlueJ's code editor.

BlueJ allows programmers to interact with classes and objects through their context menus. It allows programming by concept by allowing instances of classes to be created from their context menus. This functionality, however, is limited to classes and objects and is not extended to other constructs and program elements. Programmers are also not presented with the syntax and the semantic rules of the language to infer and contextualize them from within the IDE.

2.6 Greenfoot

Greenfoot is an interactive Java development environment that allows easy development of applications ranging from simple games to highly sophisticated simulations of complex systems [8]. The main window of Greenfoot displays the Greenfoot world of customizable size that contains objects from a scenario. Figure 2.15 shows the main window that displays the Greenfoot world where scenarios are executed. A Greenfoot world displayed in the main window is a grid similar to a worksheet in AgentSheets. The panel to the right is a class browser which visualizes the classes and their inheritance relations used in a scenario. Double clicking on a class icon opens up its code in a text editor showing its source code as shown in Figure 2.16 [9].

Greenfoot extends the idea of the object bench of BlueJ to an object world where all objects have a graphical appearance and a position in the Greenfoot world. Objects can be created by interacting with the classes as shown in Figure 2.17. An object behavior can be observed directly by invoking methods on the object and observing changes in its position and appearance in the Greenfoot world. Figure 2.18 shows a grid of cells in the Greenfoot world which hosts individual greenfoot objects which are interactive [7].

Greenfoot attaches contextual information to the objects in the Greenfoot world and to the classes in the class browser but does not extend this functionality to actual programming where no contextual information is associated with the elements in a program. It does not offer the features that would allow programmers to infer syntax or semantic rules of a language from within its editor.

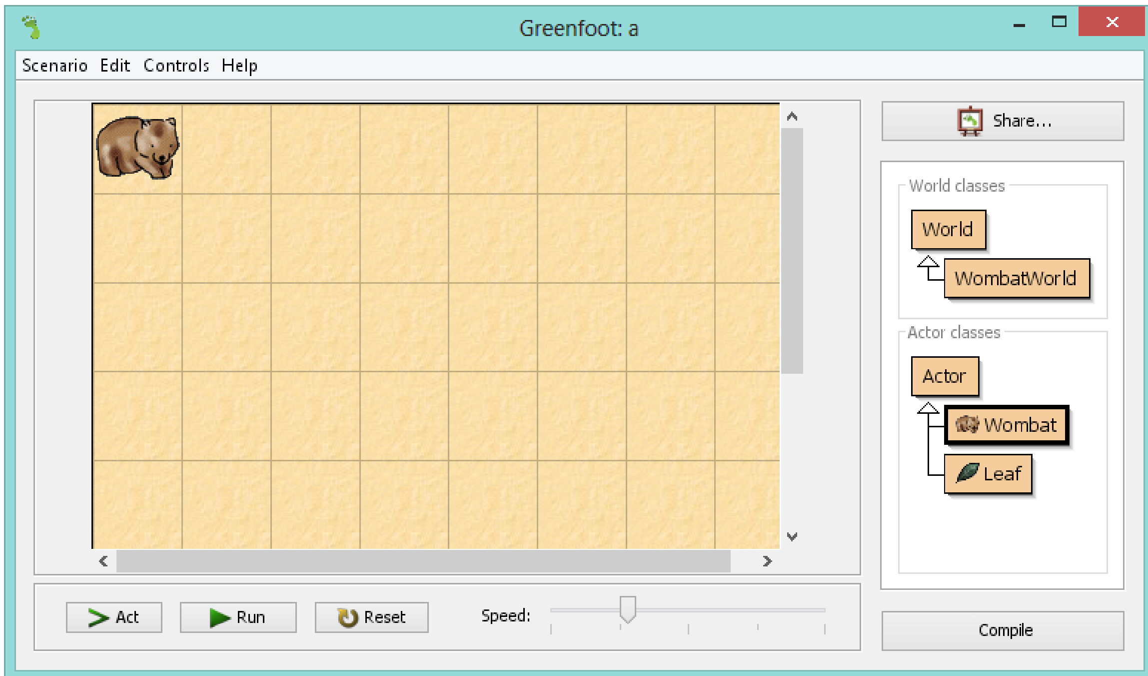


Figure 2.15: The main window of GreenFoot.

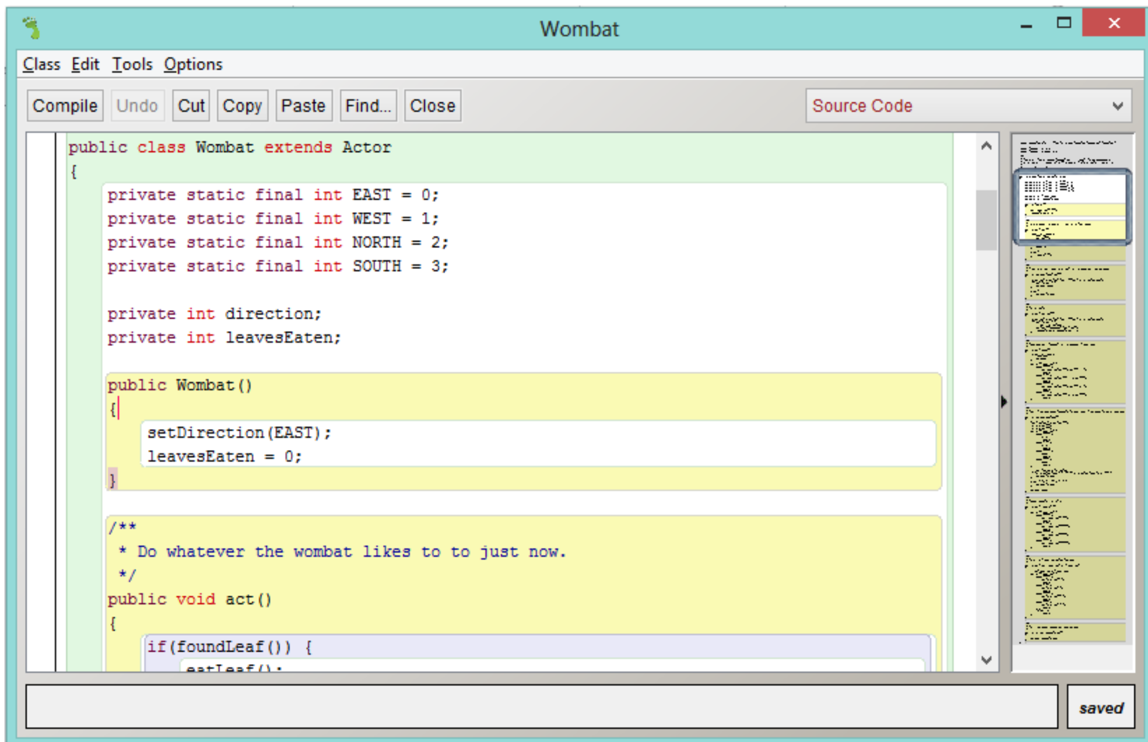


Figure 2.16: The code editor of GreenFoot.

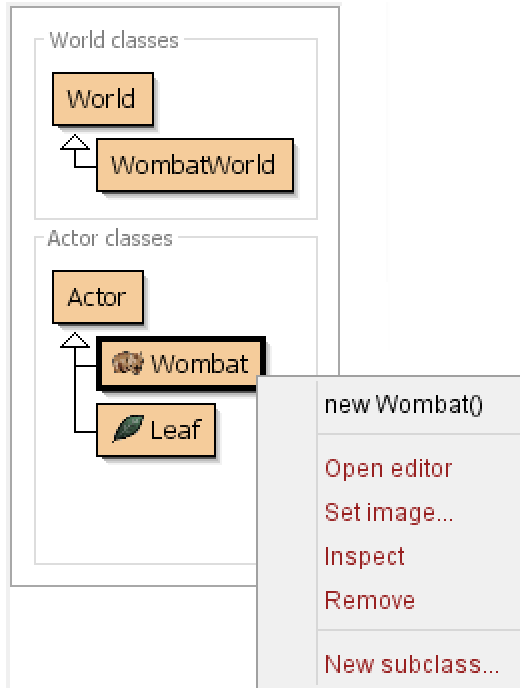


Figure 2.17: Context menu of a class in GreenFoot.

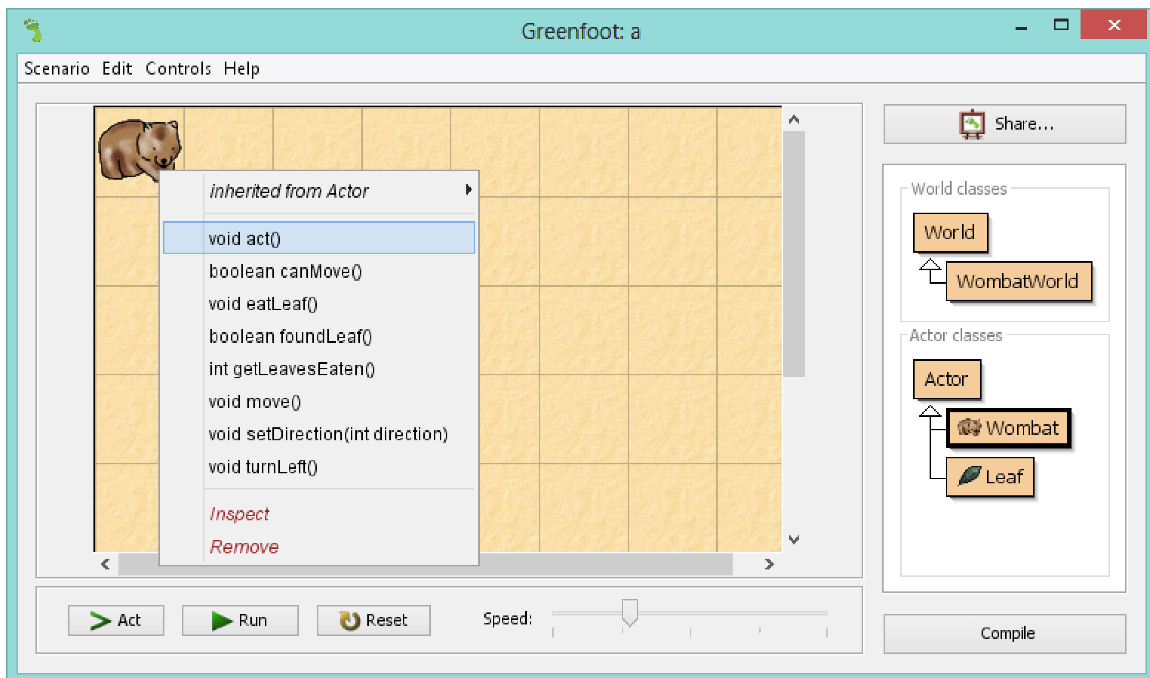


Figure 2.18: Greenfoot world hosting GreenFoot objects.

2.7 CSmart

CSmart is considered to be a learner's integrated development environment (L-IDE) proposed to increase the pedagogical value of the source code of sample programs by transforming them into self-explaining tutorials. It emphasizes teaching by example where the syntax and the semantics of a language are taught by presenting samples of annotated program code prepared by experienced teachers. Tutorials are created dynamically by parsing the example source code files which are then presented in a rich format that aids comprehensibility. Figure 2.19 shows how a line of code is parsed into an informal instruction within CSmart's editor. Rendered code in CSmart is annotated with the comments added to the example source code by the authors of the tutorial, who are expected to ensure that the explanations are appropriate.

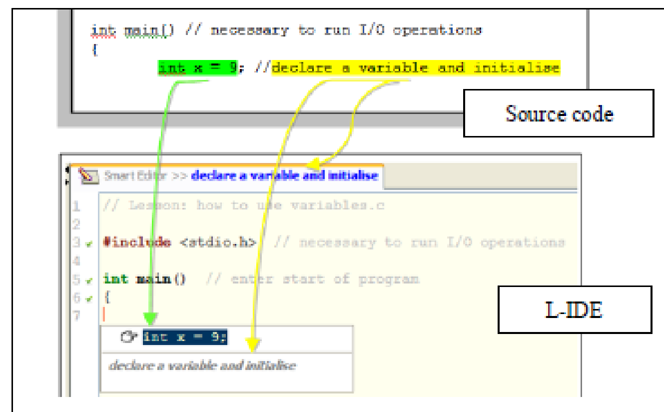


Figure 2.19: Annotated code in (from p03 of [6]).

Figure 2.20 shows a pane next to the editor area that displays additional information on keyword and standard functions to augment the learning process of learners.

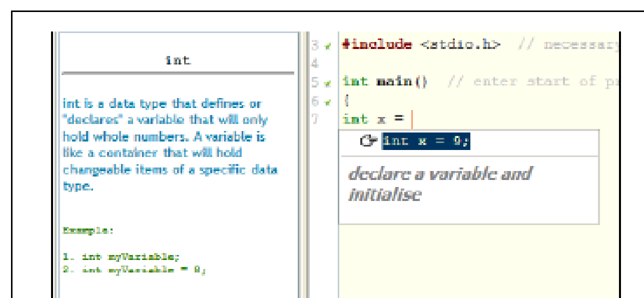


Figure 2.20: Information Pane in CSmart (from p03 of [6]).

CSmart also generates visual explanations of code constructs automatically. Figures 2.21 & 2.22 show how this feature is presented to the learners [6].

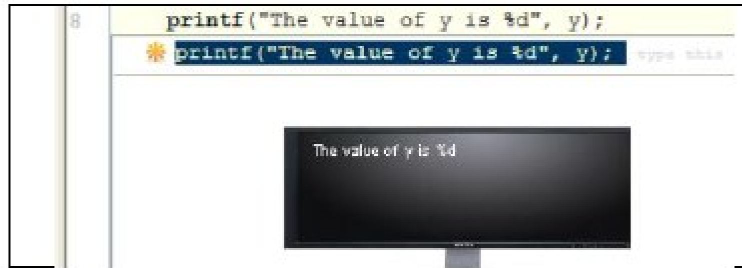


Figure 2.21: Visualizing the printf function of (from p03 of [6]).

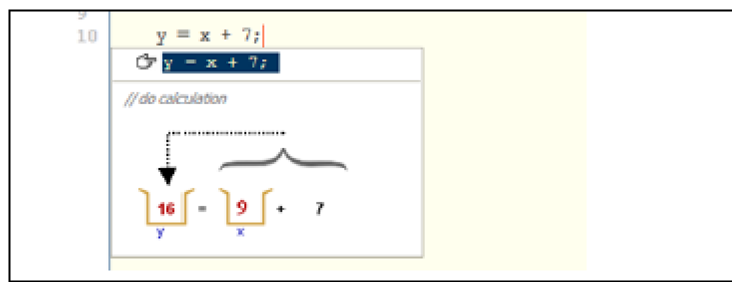


Figure 2.22: Visualizing an arithmetic operation in C (from p04 of [6]).

CSmart requires the examples to be manually annotated with syntactic and semantic information. This is required for every example created by the instructor. The annotations added by the instructor are confined to the example to which they are added. The IDE offers no provision for the instructor to add these rules once and allow the learners to infer them across all examples when relevant. Learners are limited to inferring the annotations provided by the instructor with no options to instantiate and contextualize them.

2.8 Grammar Cells

In another approach to providing syntactic guidance in an IDE, controls are associated with the language elements. Figure 2.23 displays the aspects of a programming language including its grammar presented as basic controls called Grammar Cells [12]. Each control has its own set of properties that the users can set as shown in Figure 2.24. The re-

search, however, focuses only on properties associated with grammar cells but not on the contextual actions applicable to the elements of a program. Presenting and contextualizing the semantic rules of a programming language are also not considered in this research.

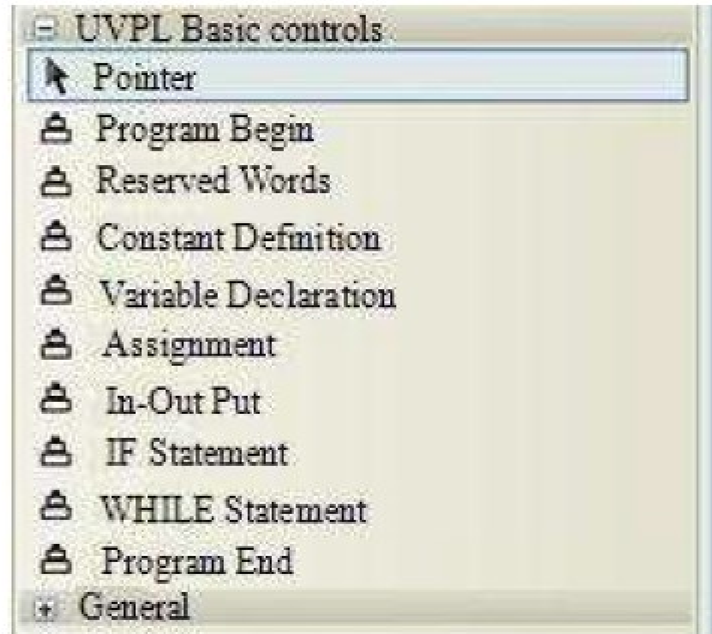


Figure 2.23: Grammar cells of a language (from p03 of [12]).

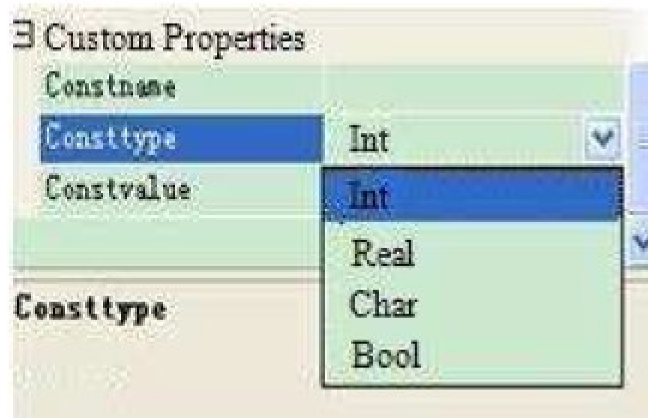


Figure 2.24: Properties of a constant control (from p03 of [12]).

2.9 Eclipse

Eclipse is a multi-language IDE and an integration platform that serves the domain of software development tools [4]. Figure 2.25 shows a Java class creation wizard in Eclipse which presents the user with a set of properties of a class. When creating a class, the user can specify its name, the package it belongs to, its parent class, and a set of interfaces that it implements and whether it is an outer class or an inner class, amongst other details. This wizard is limited to the creation of high level Java constructs/concepts like classes, interfaces and enums but is not extended to other constructs and elements of a Java program that constitute the content of these constructs. For example, the class wizard does not provide a means for specifying the methods of a class, and there is no method wizard for defining properties of a method. This scenario is also true of other languages supported in Eclipse and other similar multi-language IDEs like NetBeans, Visual Studio, etc.

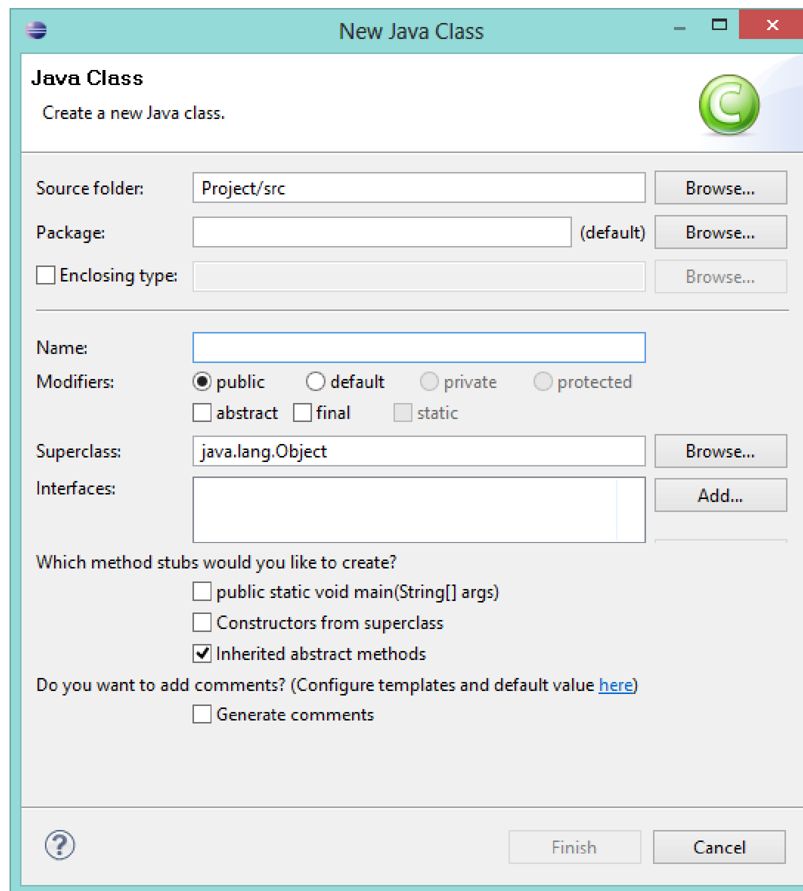


Figure 2.25: A Java class creation wizard in Eclipse.

As we have noted in examples 4 and 5 of section 1.2, these IDEs offer no features to allow the user to infer the syntactic or the semantic rules of a language from a program that conforms to these rules.

Chapter 3

Az-Nuggets

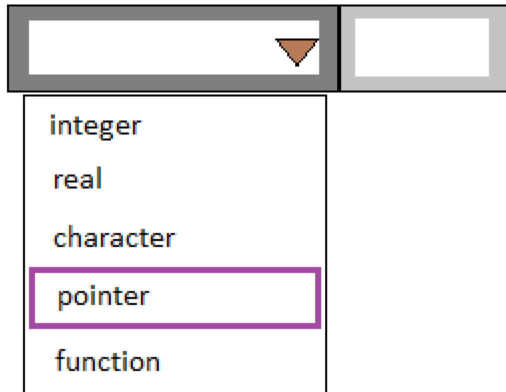
Az-Nuggets is a pseudo-visual programming IDE that offers language-specific visual abstractions called nuggets. Programs in Az-Nuggets are composed with nuggets which are interactive graphical widgets representing elements or constructs of a programming language. Nuggets are similar to visual blocks provided by tools like Scratch and App Inventor but in addition, they have contextual and semantic information associated with them. The contextual information associated with a nugget is a list of contextual entries with each entry representing either a property of the nugget or an action that can be specified on the nugget. Each entry is optionally associated with semantic information pertaining to the entry as defined by the language. These features enable programming by concept and the exploration of semantic details of a programming language. This chapter is dedicated to the discussion of these features of Az-Nuggets from the perspective of application developers.

3.1 Nuggets

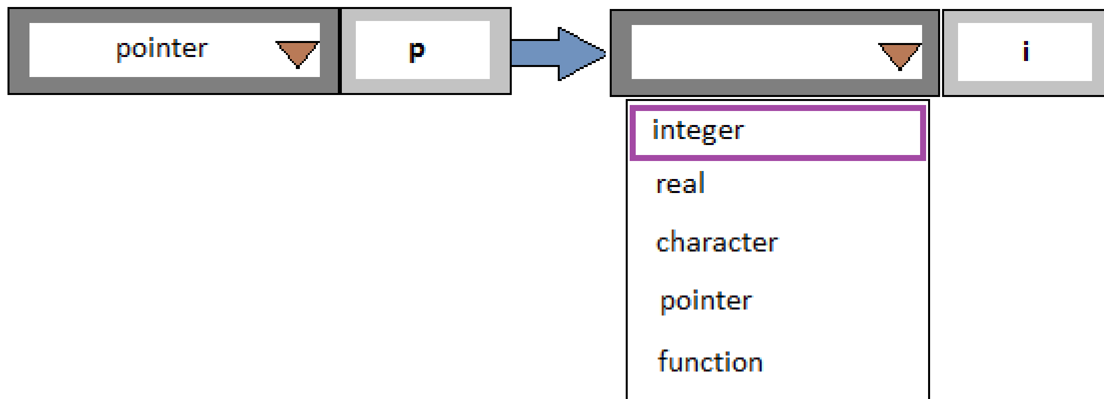
A nugget is an interactive visual abstraction of a programming language construct or an element of a program that is associated with contextual and semantic information. Nuggets can be instrumental in bridging the gap between the syntax and the semantics of a language. Programmers can interact with nuggets without necessarily having to deal with the textual syntax of a programming language.

Consider, for example, the first two pointer declarations in C listed under section 1.1. Figures 3.1 and 3.2 illustrate a nugget that can be used to declare variables/constants/functions in C. Figure 3.1a shows the introduction of a pseudo-type called

pointer in C to declare a pointer variable that can point to a specified type. Figure 3.1b shows that upon choosing pointer as the type for the variable p, an arrow and a drop-down appears to its right allowing the user to choose integer as the type that p can point to. The user can optionally initialize p at the point of declaration as shown in Figure 3.1b where it is initialized to point to an integer variable i.



(a) A nugget used for declarations in C

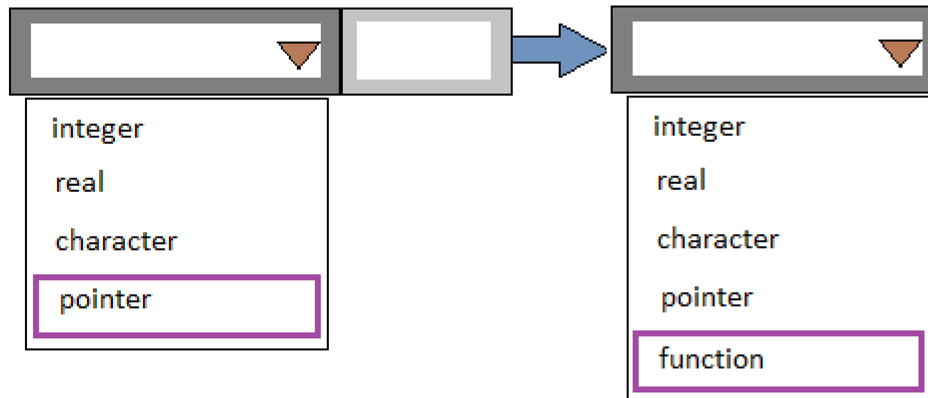


(b) Pointer p initialized to point to an integer variable i

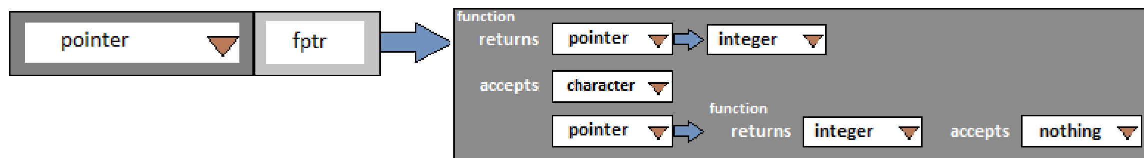
Figure 3.1: Nugget used to create a pointer p to an integer i.

A pointer to a function can be declared in a similar way by choosing a pseudo-type function from the drop-down as shown in Figure 3.2a. The drop-down is replaced by a nug-

get that allows the user to declare the types of arguments and the return type of the function that the pointer can point to. Figure 3.2b shows the declaration of a function pointer `fptr` that is declared to point to a function that returns a pointer to an integer and accepts two arguments: a character and a pointer to a function that returns an integer and accepts no arguments.

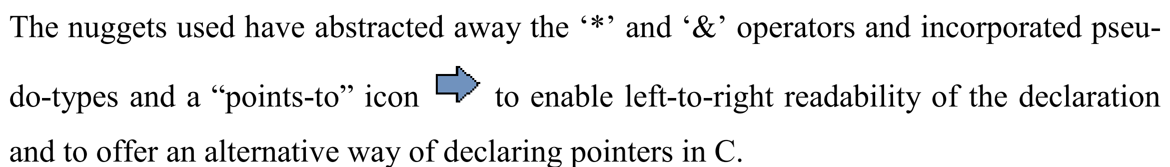


(a) A nugget showing declaration of a function pointer



(b) Declaring `fptr` to point to a function of a specific type

Figure 3.2: Nugget used to declare a pointer `fptr` to point to a function of a specific type.

The nuggets used have abstracted away the '*' and '&' operators and incorporated pseudo-types and a "points-to" icon  to enable left-to-right readability of the declaration and to offer an alternative way of declaring pointers in C.

With nuggets, the user does not have to focus on petty details of the textual syntax of a language removing the burden of memorizing and dealing with the textual syntax to be able to program in that language. This also leads to a possibility of having a consistent representation of common language concepts across different programming languages.

Consider the declarations of a class `SampleClass` in the following programming languages:

Objective-C:

```
@interface SampleClass : NSObject
@end
```

Java:

```
class SampleClass
{
}
```

C++:

```
class SampleClass
{
};
```

Python:

```
class SampleClass:
```

Ruby:

```
class SampleClass
end
```

In these declarations tokens like '`:`', '`;`', '`{`', '`}`', '`end`', '`@end`' can be categorized as "noise tokens" as they have no inherent meaning. Nuggets can be used to abstract away these noise tokens to provide a consistent representation of concepts across languages. Assuming that the change in the declaration of the class construct of Objective-C from `@interface` to `class` is consistent with the language, the *class* nugget shown in Figure 3.3 can be used to represent the class construct consistently across these languages.

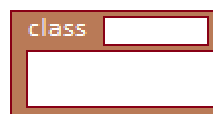


Figure 3.3: A generic *class* nugget.

Consider the example of single and multiline comments in different programming languages:

In Objective-C, Java, C and C++, single line comments are provided by preceding the text of the comment with `//` whereas a multiline comment is provided by enclosing the text of the comment between `/*` and `*/`.

Eg:

```
// This is a single line comment
/*
      This is a
      multiline comment
*/
```

In Python and Ruby, single line comments are provided by preceding the text of the comment with a `#`

Eg:

```
# This is a single line comment
```

In Python, multiline comments are provided by enclosing the multiline comment in triple quotes, which can be either single or double quote marks.

Eg:

```
"""
      This is a
      Multiline comment
"""
```

In Ruby, multiline comments are provided by enclosing the multiline comment in a `=begin =end` block:

Eg:

```
=begin
      This is a
      Multiline comment
=end
```

In Ant scripts, single or multiline comments are created by enclosing the text of the comment between `<!--` and `-->`

Eg:

```
<!-- Single or multiline comments goes here -->
```

The differences in the ways of providing comments across these languages can be abstracted away with the usage of the nugget depicted in Figure 3.4 to be commonly used across all the languages for both single and multiline comments.

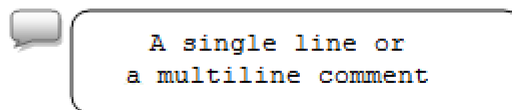


Figure 3.4: A generic nugget to provide single or multiline comments.

Nuggets can be associated with a variety of properties that allows them to be presented in interesting ways to convey concepts of a programming language. Properties like visual cues, natural language choices, audio feedback, icons etc. can be tagged with nuggets and manipulated in numerous ways to create new avenues for conveying semantics.

3.2 Overview

Upon launching Az-Nuggets, the user is provided with a list of languages enabled in Az-Nuggets that can be chosen from as shown in Figure 3.5. The figure also shows a drop-down that contains a list of natural languages for letting the user choose a natural language for the IDE. The text in the IDE is displayed in the natural language selected at the time of launch. However, the choice of language can be switched to any other natural language supported by the IDE. Figure 3.6 shows a tentative depiction of the layout of the main screen of Az-Nuggets. It displays a categories panel in the top-left corner that is used to display the categories that group the nuggets of the language. The nuggets panel in the left is used to display the nuggets of a selected category. The categories and the nuggets in these panels are specific to the programming language chosen by the user. The

programming editor on the right is where the nuggets can be dragged and dropped to compose a program. The nuggets in the programming editor are instances of their corresponding nuggets from the nuggets panel.

Figure 3.5: Programming and natural language selection screen.

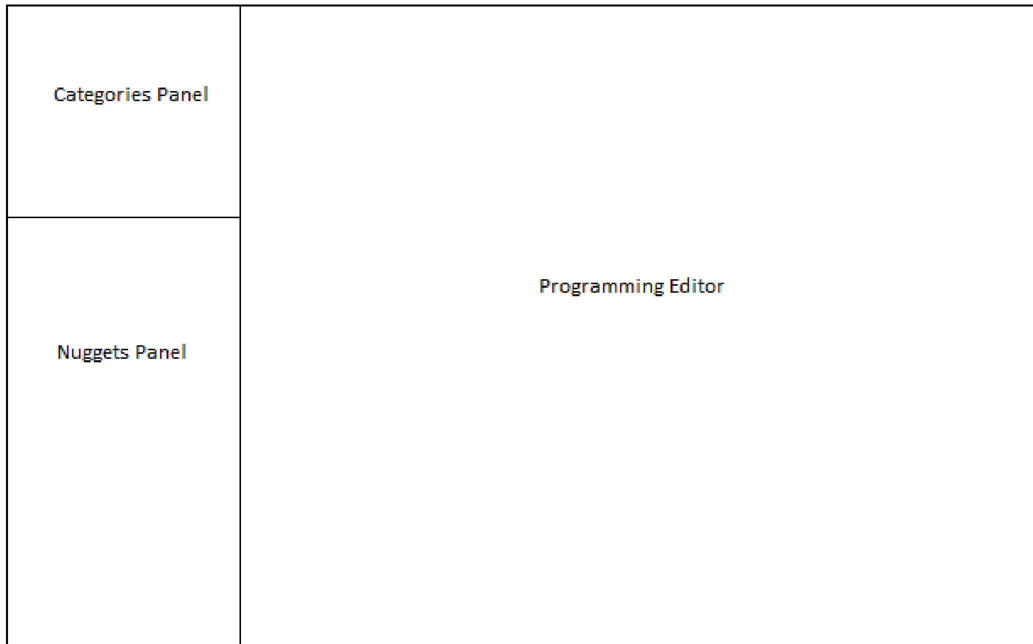
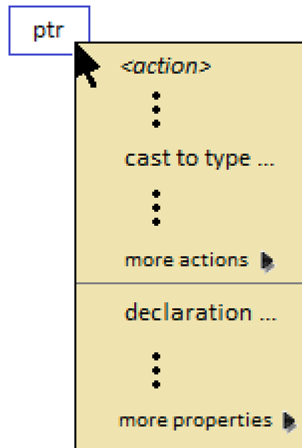


Figure 3.6: A tentative depiction of the layout of the main screen of Az-Nuggets.

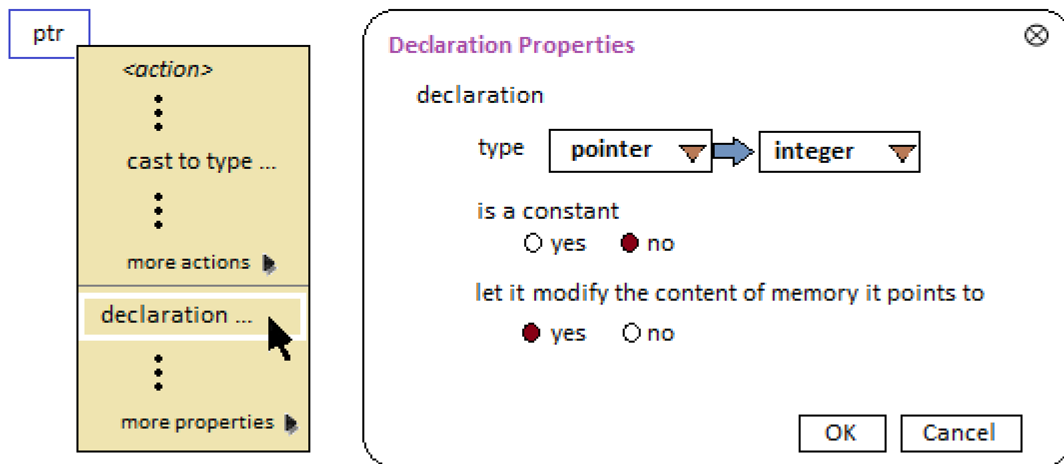
3.3 Programming by Concept with Nuggets

A nugget is associated with contextual information that allows users to interact with it at a conceptual level without having to know the exact syntactic abstraction to express a concept in a particular programming language. This allows the users to work with nuggets while knowing "what to do" (the concept) rather than "how to do it" (the exact syntax to convey the concept). The contextual information associated with a nugget can be properties attributable or actions applicable on the nugget. This information facilitates the exploration of the language concepts associated with a nugget. The following examples from different programming languages illustrate the contextual information associated with nuggets and how it enables programming by concept. Note that the list of contextual concepts attached to a nugget in the examples pertain only to the example and is not an exhaustive list of possible concepts that can be associated with the nugget. Although the contextual information associated with a nugget is presented as a contextual menu, it could as well be presented in various other ways.

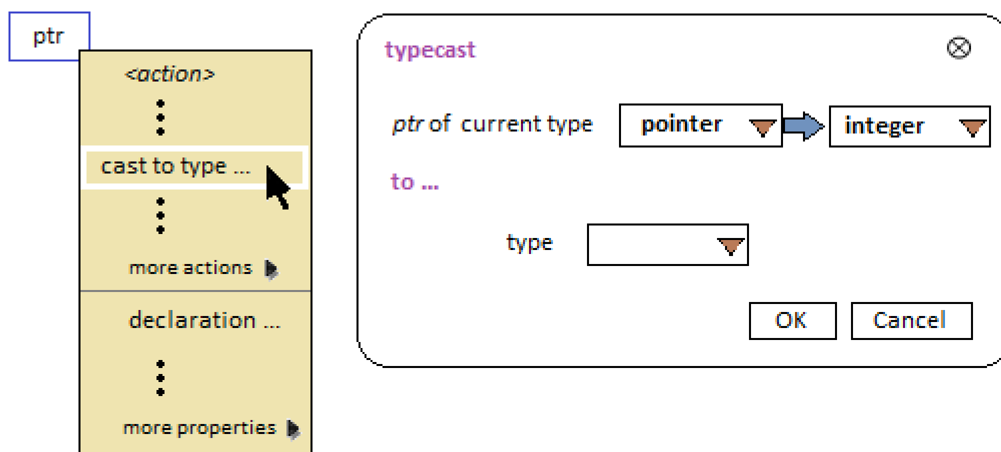
Consider declaring a constant pointer to a constant integer in C/C++. Assuming a pointer variable `ptr` to an integer is already declared in the code, Figure 3.7a shows a reference of `ptr` in the editor and its context menu displaying contextual entries pertaining to `ptr`. These contextual entries are a list of properties and actions that are relevant to `ptr`. Selecting the *declaration* entry from the context menu of `ptr` displays a wizard as shown in Figure 3.7b that contains the declarative properties of `ptr`. Note that `ptr` is initially declared as a non-constant pointer to a non-constant integer. Setting the option *is a constant* to *yes* and the option *let it modify the content of memory it points to* to *no* adjusts the declaration of `ptr` to make it a constant pointer to a constant integer at the point of its declaration. This change would simultaneously be made in the *declaration properties* wizard as well. Figure 3.7c is an example of choosing a contextual action on `ptr` which shows a similar approach to casting `ptr` to a desired type. Choosing *cast to type ...* from the context menu of `ptr` displays a wizard that lets the user choose the type to cast `ptr` to. Confirming the cast inserts into the program the constructs required to typecast `ptr`.



(a) Context menu displaying the contextual information associated with a pointer variable.



(b) A wizard displaying the declarative properties of `ptr`



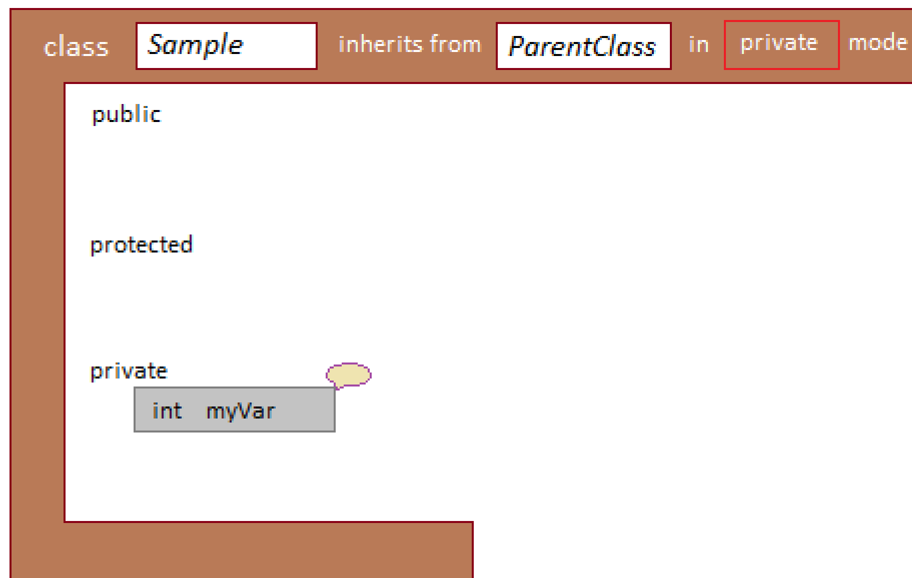
(c) Selecting a contextual action, *cast to type* ..., to typecast `ptr`

Figure 3.7: Conceptually working with a pointer variable.

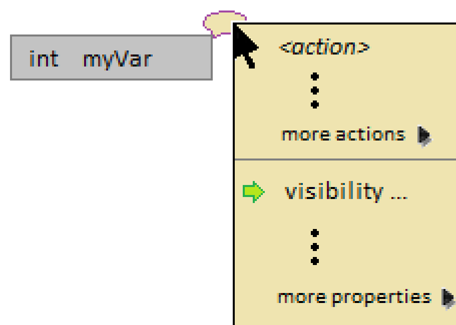
It can be observed from both these cases that working with the contextual information associated with `ptr` allow the user to program using concepts, without knowing the exact syntactic details of the programming language.

3.4 Exploration and Presentation of language details

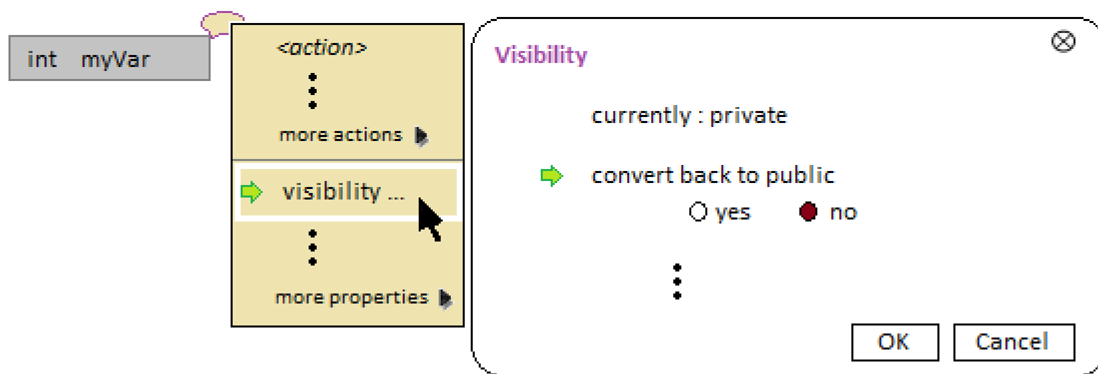
Consider a class `Sample` in C++ that inherits from the class `ParentClass` in private mode. `ParentClass` is defined to have only one public integer member `myVar`. In private inheritance mode, all the public and protected members of the parent are inherited as private members of the child class. However, C++ lets the access modifiers of these inherited members in the child class be reverted to their original values as in the parent class. The hint icon on the inherited member `myVar` as shown in Figure 3.8a guides the user to this language feature. Clicking on the hint icon displays the context menu of `myVar` as shown in Figure 3.8b with a pointer guiding the user to interact with the *visibility* property of `myVar` that offers the details about the hint. Figure 3.8c shows a wizard that points to the language feature that allows the user to apply it by selecting *yes* for the *convert back to public* option. Confirming this change inserts into the program necessary constructs to revert the access modifier on `myVar` in `Sample`.



(a) A hint icon on `myVar`



(b) The context menu of `myVar` guiding the user to interact with *visibility* property

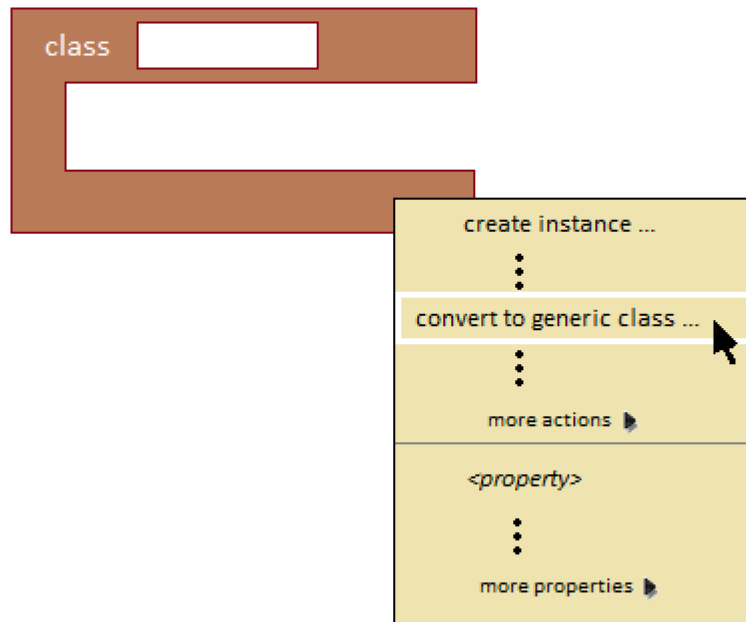


(c) The *visibility* wizard pointing to the language feature

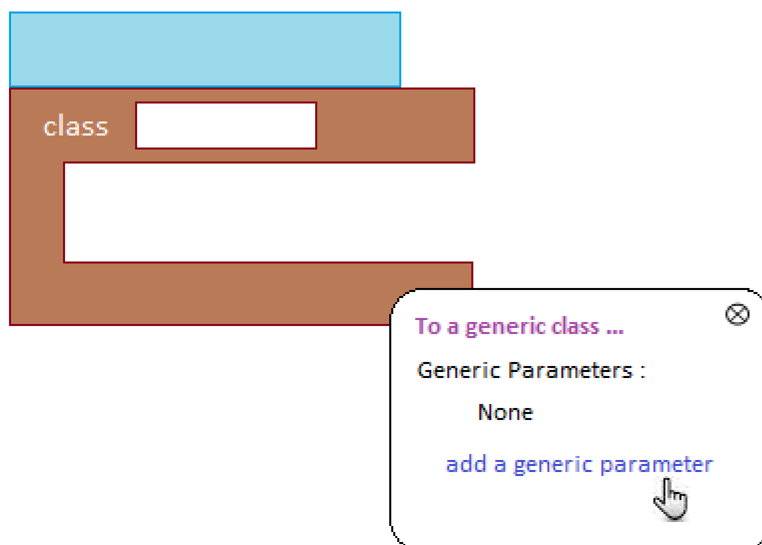
Figure 3.8: Reverting the access modifier in C++ conceptually.

Consider the following example of how a user who is unfamiliar with the creation of a generic class in C++ can work with it conceptually and explore its features. Figure 3.9a shows the context menu of a class in C++ from which the user can select the *convert to a generic class* ... option to bring up a wizard that allows generic parameters to be added and configured. It also changes the appearance of the class as shown in Figure 3.9b to create an extension on top of the *class* nugget to hold its generic parameters and to indicate that it is a generic class. The user then selects the option, *add a generic parameter*, to add a generic parameter `X` and set it to accept a type as an argument and defaults it to `int` as shown in Figure 3.9c. Similarly, the user adds another generic parameter `Y` and sets it to accept an integer value defaulted to `10`. Figure 3.9d shows that the user can pull up the

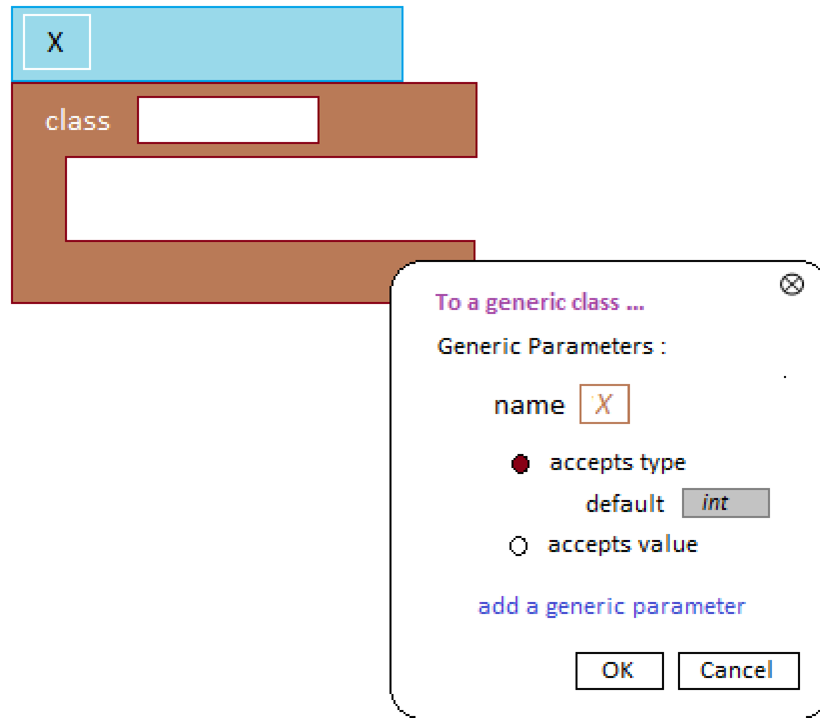
wizards for the generic parameters X and Y at any time to reconfigure them if needed. The *create instance* option from the context menu of the class can be used to create an instance of the generic class. The sequence of steps discussed in this example allowed the user to create a generic class and to specify generic parameters. It also helped the user to understand that a generic parameter can be declared to accept a type or a value of a particular type, and provided with a default type or a value.



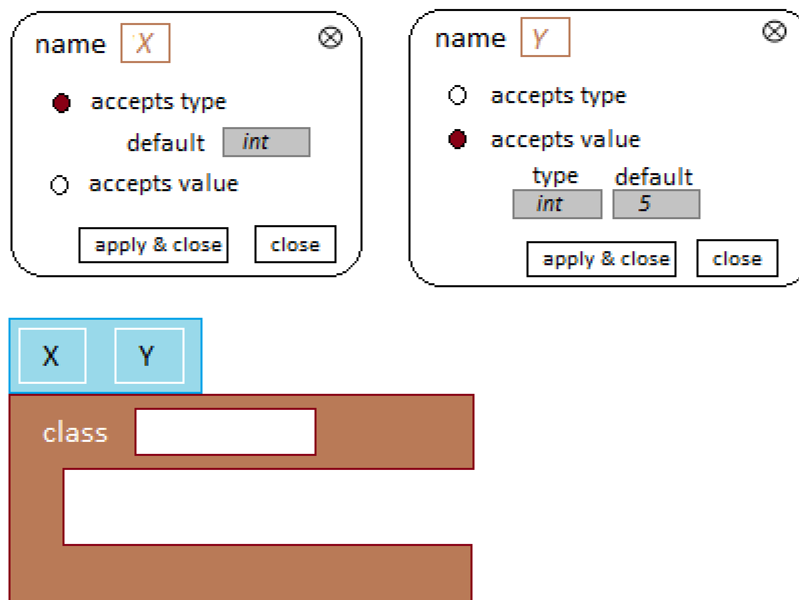
(a) Creating a generic class from a regular class



(b) A wizard that lets the user specify generic parameters



(c) Adding a generic parameter `X` that accepts a type defaulted to `int`



(d) Wizards to configure each parameter

Figure 3.9: Conceptually working with a generic class in C++ and exploring its features.

3.5 Unfolding Semantics

A contextual property or action associated with a nugget can be tagged with syntactic or semantic information. The user can access this information from the Rules context menu associated with a nugget which can be invoked on a nugget by bringing up its context menu while pressing a modifier key. Figure 3.10 shows the rules-context menu associated with `ptr` that was discussed in Figure 3.7. When an instance of a nugget appears in a program, the information tagged to it is instantiated depending on the context in which the nugget occurs. The following examples show how the user can access the static information about syntax or semantics, and the cases when this information is instantiated and contextualized.

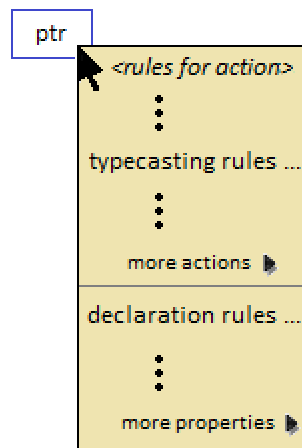
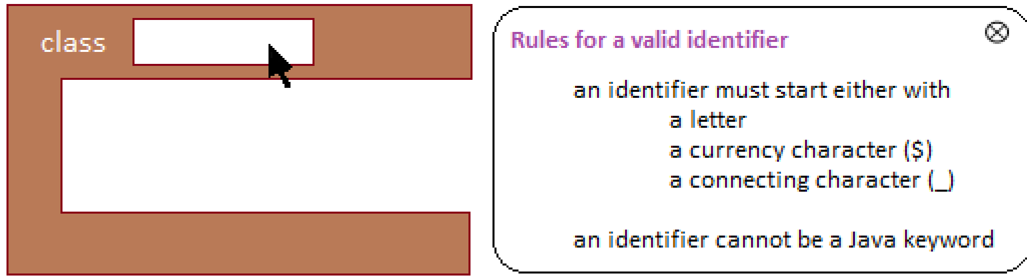
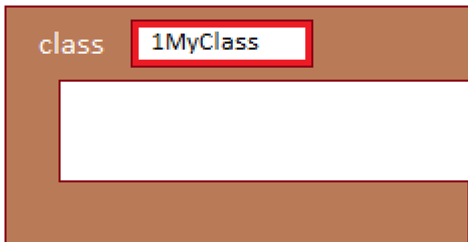


Figure 3.10: The Rules context menu of `ptr`.

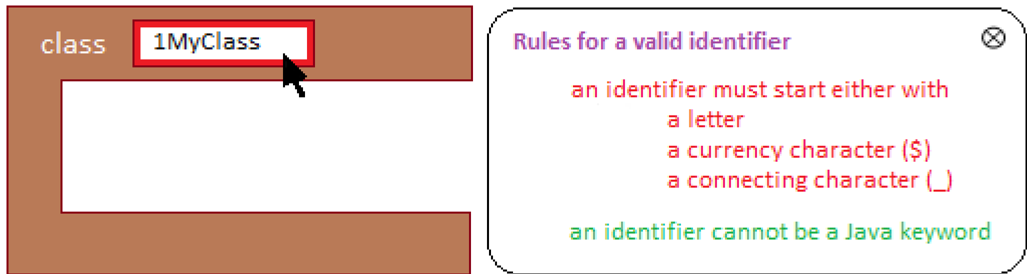
Consider a simple example of identifying the rules for naming a legal identifier for a class in Java. Figure 3.11a shows the user interacting with a *class* nugget in the programming editor by pointing the mouse at the identifier textbox and pressing the modifier key to bring up the rules in a window. The user then closes the rules window and attempts to name the class, `1MyClass`, which violates one of the rules for valid identifiers. This attempt highlights the identifier textbox in red as shown in Figure 3.11b. Depending on the user settings, the verification and the feedback are either provided as the user types in the identifier in the textbox or after the identifier is typed in, as is the case in this example.



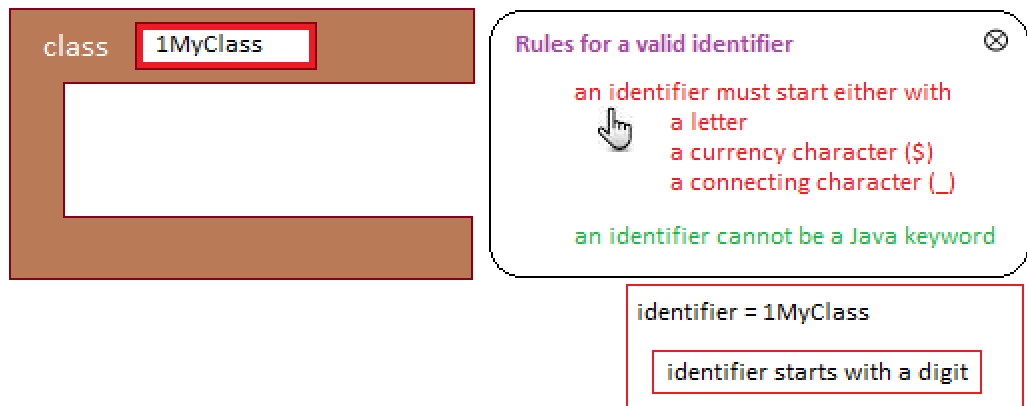
(a) Viewing the rules for a valid identifier



(b) Feedback provided on an invalid identifier



(c) Instantiated rules

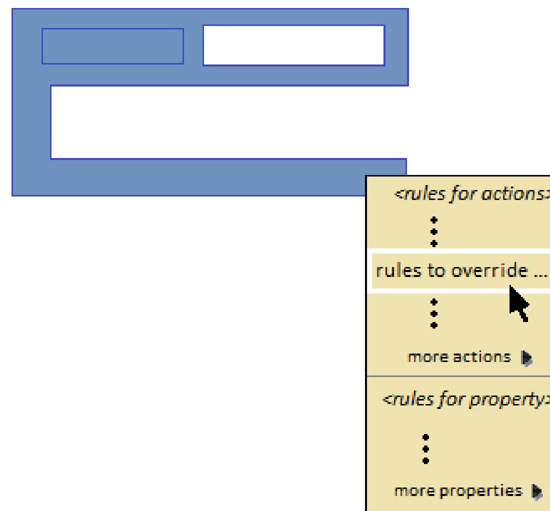


(d) Contextualizing a rule

Figure 3.11: Static, instantiated and contextualized rules of a naming an identifier in Java.

Figure 3.11c displays the window with instantiated rules as the user hovers the mouse over the highlighted textbox. The instantiated rules are contextually relevant rules highlighted to provide accurate feedback to the user. Note that the violated rule in Figure 3.11c is highlighted in red whereas the passed rule is highlighted in green. Figure 3.11d shows the user contextualizing the violated rule by clicking on it to display it in an inset below.

The rules associated with a property or an action of a nugget are instantiated when the nugget is in a context where the property or action is relevant. Figures 3.12a and 3.12b shows the user selecting the *rules to override* item in the Rules context menu of an uninitialized *method* nugget in the programming editor, to view the rules pertaining to overriding a method. Note that the overriding rules displayed in the window are uninitialized as the method is not overriden. Consider an example of an overriding method `methodA` of a class `ChildClass` in Java that extends and overrides a protected method `methodA` of class `ParentClass`. As `methodA` of `ChildClass` overrides `methodA` of `ParentClass`, the rules associated with the *overrides* property of `methodA` of `ChildClass` are instantiated. Figure 3.12c shows the instantiated rules after an attempt to set the access modifier of `methodA` of `ChildClass` to private. The figure also shows the violated rule contextualized in the inset. The information in the inset maps the subjects of the violated rule with the program elements from the context and displays a scale of access modifiers of Java followed by a summary of why the rule is violated.



(a) Selecting *rules to override* from the context menu of a *method* nugget

Rules to override a method

Overriding method must be defined in a child class of the class enclosing Overridden method

Overriding method should have the same name as the Overridden method

Overriding method's argument list must exactly match Overridden method's argument list

Overriding method's return type should be same or a subtype of Overridden method's return type

Overriding method's access level cannot be more restrictive than Overridden method's access level

Overriding method's must not throw new or broader checked exceptions than the Overridden method

⋮

(b) Viewing the rules for overriding a method

Rules for the overriding method

Overriding method must be defined in a child class of the class enclosing Overridden method

Overriding method must have the same name as the Overridden method

Overriding method's argument list must exactly match Overridden method's argument list

Overriding method's return type should be same or a subtype of Overridden method's return type

Overriding method's access level cannot be more restrictive than Overridden method's access level

Overriding method's must not throw new or broader checked exceptions than the Overridden method

⋮

Overriding method = **methodA** *of* ChildClass

Overriding method's access level = **private**

Overridden method = **methodA** *of* ParentClass

Overridden method's access level = **protected**

most restrictive — least restrictive

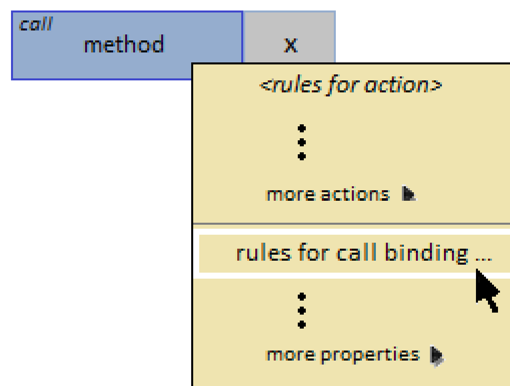
	private	in-package	protected	public
Overridden method			⇧	
Overriding method	⇧			

methodA of ChildClass has restrictive access level than methodA of ParentClass

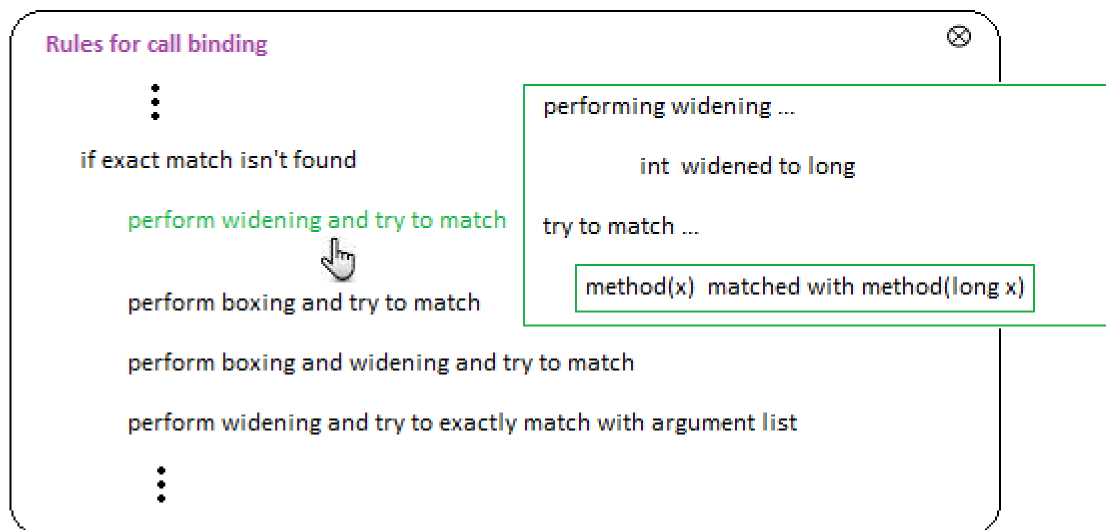
(c) Instantiated rules with a contextualized violated rule

Figure 3.12: Static and instantiated rules for method overriding in Java.

We now consider the fifth example in section 1.1 to show how the user can access semantic information from a syntactically and semantically correct program. Figure 3.13a shows the user clicking on the *rules for call binding* item in the Rules context menu associated with the method call to view the instantiated rule(s) used to bind the method call to one of the method definitions. Figure 3.13b shows the instantiated rule used in this context and the user contextualizing it by clicking on it to display more information on the rule in the inset below. The information in the inset shows that the argument `x` in the call `method(x)` is widened to `long` to bind the call to the method definition of `method(long x)`.



(a) The Rules context menu of method call



(b) Instantiated and contextualized rule

Figure 3.13: Instantiated rules for compile-time call binding.

The fourth example of section 1.1 can be explored by the user in a similar way to understand relevant semantic rules. Figure 3.14 shows a window displaying the instantiated rules associated with the *overrides* property of the method `method` defined in the class `Example`. Note how the rules are all highlighted in green implying that `method` of `Example` has overridden `method` of `Interface3` with no errors. The figure also shows the rule related to exceptions contextualized in an inset informing the user that `method` of `Example` cannot throw any exceptions since `method` of `Interface3` does not throw any exceptions. To understand why the overridden method `method` of `Interface3` does not throw any exceptions, the user brings up the rules for exceptions from the Rules context menu of `method` of `Interface3` as shown in Figure 3.15a. Figure 3.15b displays a window with the relevant semantic rule instantiated and contextualized to further inform why `method` of `Interface3` cannot throw any exceptions despite its declarations in `Interface1` and `Interface2` to throw exceptions.

Rules for the overriding method

Overriding method must be defined in a child class of the class enclosing Overridden method

Overriding method must have the same name as the Overridden method

Overriding method's argument list must exactly match Overridden method's argument list

Overriding method's return type should be same or a subtype of Overridden method's return type

Overriding method's access level cannot be more restrictive than Overridden method's access level

Overriding method's must not throw new or broader checked exceptions than the Overridden method

⋮

Overriding method = **method** *of* Example

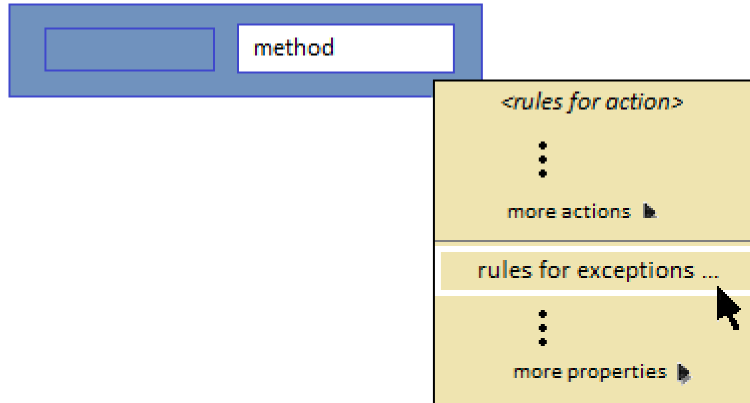
Overridden method = **method** *of* Interface3

Exceptions thrown by Overridden method = **NONE**

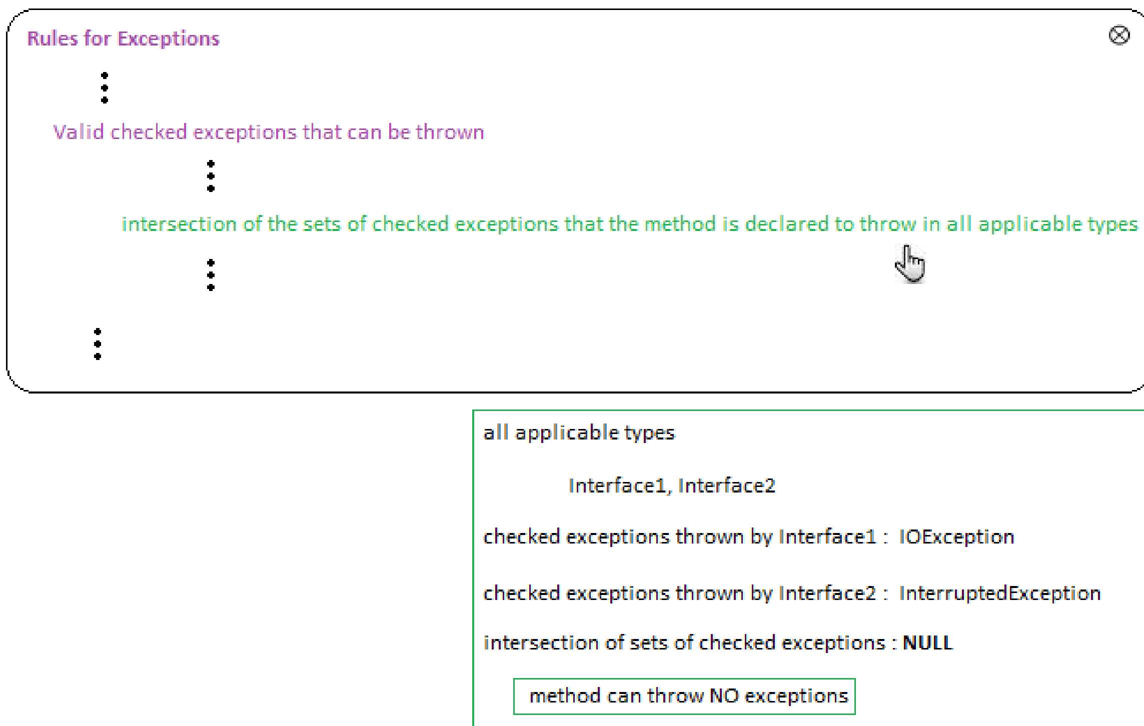
Overriding method
cannot throw new exceptions than NONE => cannot throw ANY => can throw NONE
or
cannot throw broader exceptions than NONE => cannot throw ANY => can throw NONE

Exceptions that Overriding method can throw = **NONE**

Figure 3.14: Instantiated rules for Overriding; Contextualizing one of the relevant rules.



(a) Rules context menu of `method` of `Interface3`



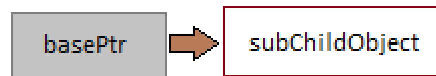
(b) Contextualizing the rule to understand why `method` of `Interface3` cannot throw any exceptions

Figure 3.15: Exploring semantic rules of exceptions associated with methods in Java.

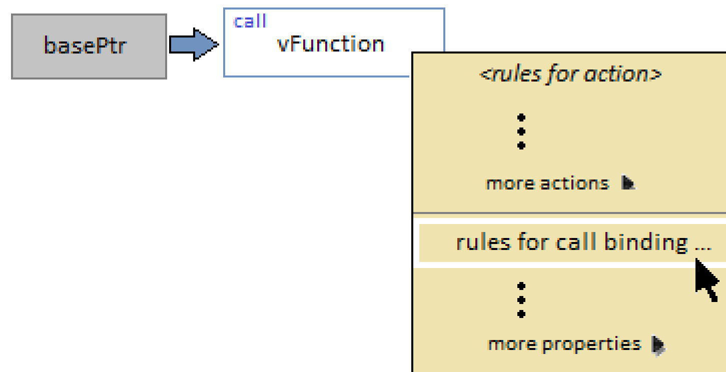
Consider the following example in C++ that demonstrates how this feature can also be used to infer runtime semantics. In this example, we assume a class hierarchy in which the class `ChildClass` inherits from the class `ParentClass` and the class `SubChildClass` inherits from `ChildClass`. Figure 3.16a shows a pointer `basePtr` pointing to `subChil-`

`dObject`, an instance of `SubChildClass`. Figure 3.16b shows `basePtr` invoking a virtual function `vFunction` defined only in `ParentClass` and the context menu associated with the call to `vFunction`. Clicking on the *rules for call binding* item in the Rules context menu after executing the program displays a window with instantiated rules as shown in Figure 3.16c that were relevant in the most recent invocation of `vFunction` by `basePtr`. The figure also shows the rules contextualized in an inset below.

Other limitations mentioned under Limitation B of chapter one can be addressed in a similar way by using the features discussed in the examples above. For example, the user can find out the scope of a variable in a program in Ruby by choosing the *highlight scope* contextual action from its context menu that highlights parts of code where the variable can be accessed. Similarly, semantics about a module in Python can be explored from the Rules context menu associated with a module.



(a) Pointer `basePtr` pointing to an object of class `SubChildClass`



(b) Rules context menu associated with function call

Rules for method binding

⋮

runtime binding [in the most recent invocation]

find the type (class) of the object that the pointer points to

look for the method definition in this class

if not found, traverse up the class hierarchy to find the definition

if definition found, bind the call to the method definition

⋮

type (class) of object : SubChildClass

method looking for: vFunction

looking for method in SubChildClass [not found]

[going up the hierarchy]

looking for method in ChildClass [not found]

[going up the hierarchy]

looking for method in ParentClass [found]

call bound to vFunction defined in ParentClass

(c) Instantiated rules associated with call binding are also shown

Figure 3.16: Instantiating and contextualizing rules for runtime call binding.

3.6 Putting it altogether

This section discusses the features of Az-Nuggets in the context of a single program. Consider a user composing and exploring the following Java program in Az-Nuggets.

```
package org.az.sample;

import org.az.another.*;
```

```java
class A
{
	private int p;
	protected int q;

	{
		p = 10;
		q = 20;
	}

	A()
	{
		System.out.println("In A's constructor");
		System.out.println(p + " "  + q);
	}

	public void method()
	{
		int localVar = 5;

		System.out.println("In method" + localVar);
	}
}

public class B extends A
{
	D dReference;

	B()
	{
		System.out.println("In B's constructor");
	}

	class C
	{
		public void cMethod()
		{
			System.out.println("In Inner Class");
		}
	}

	public static void main(String[] args)
	{
		B b = new B();
		C c = b.new C();
	}
}
```

Figure 3.17 shows the programming editor in which the user has dragged and dropped instances of the *package*, *import* and *class* nuggets and placed the cursor inside the class nugget.

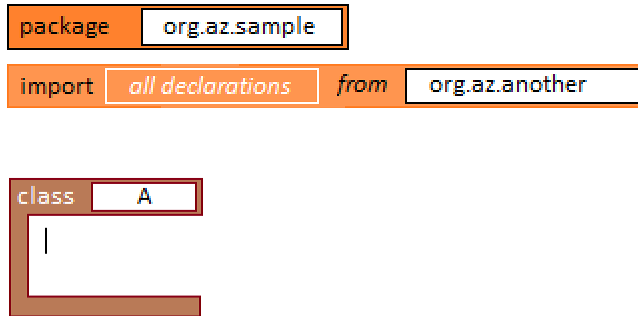


Figure 3.17: Az-Programming editor displaying *package*, *import* and *class* nuggets.

The user then hits a special key to bring up a Constructs context menu that displays a list of constructs that can be inserted at the location of the cursor as shown in Figure 3.18. Alternatively, the user can right click within `class A` to bring up the Constructs context menu to insert a construct at the point of right-click. The Constructs context menu helps the user to identify the features available in a language and further explore them within the IDE by browsing their properties, actions and rules.

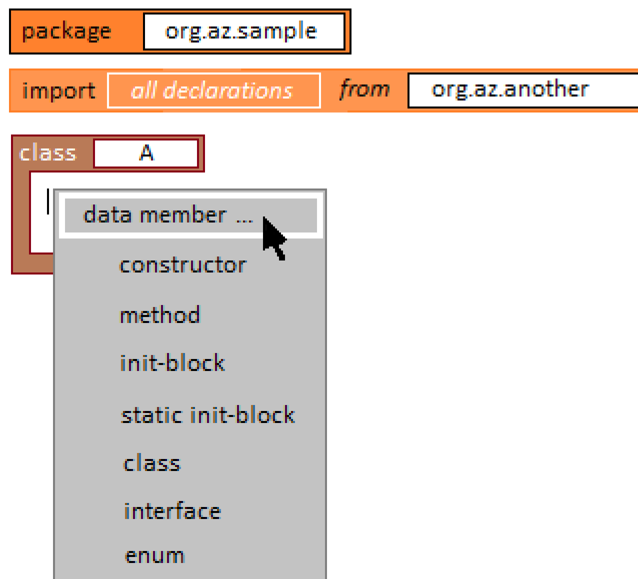


Figure 3.18: A way of entering a data member in a class.

The user proceeds with defining the non-static data member `p` by selecting the *data member* item in the Constructs context menu as shown in Figure 3.18 to access the *data member declaration* wizard as shown in Figure 3.19. The definition of `p` as it appears in `class A` is shown in Figure 3.20. Once `p` is defined, the user hits the ENTER key and simply types in a declaration for member `q`, as shown in the Figure 3.21, as an alternative to inserting it via the menu. The syntactic and the semantic validation are performed on the user input in the wizards and the programming editor to ensure the correctness of a program. The user can reconfigure the declaration attributes of `p` and `q` either directly in the programming editor or by bringing up the respective wizards through their context menus. For example, the user can select the *declaration* property item from the context menu of `p` to bring up the *data member declaration* wizard and reconfigure the properties of `p`.

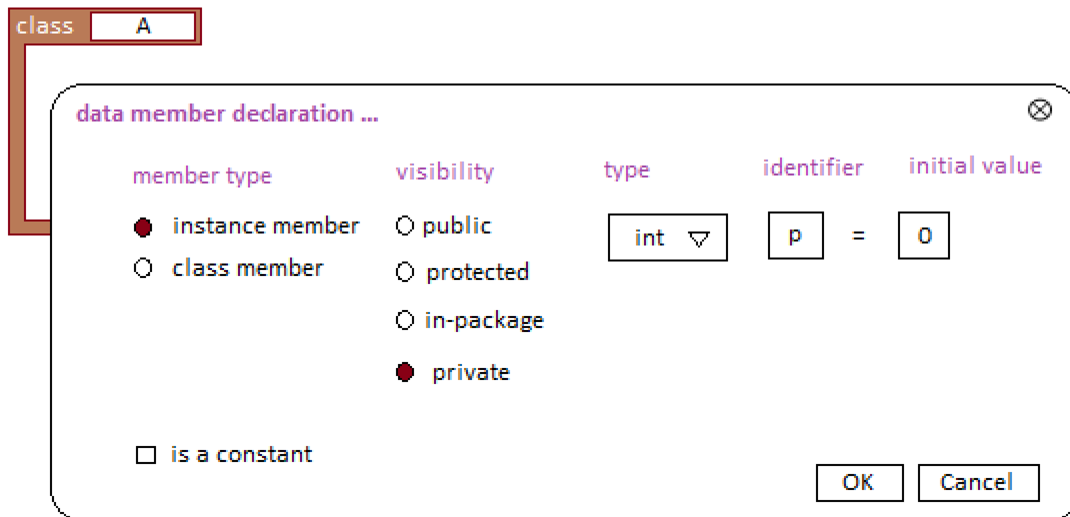


Figure 3.19: Data member creation wizard.

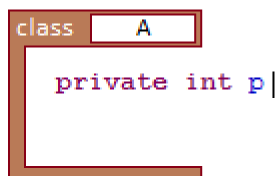


Figure 3.20: The definition of the data member `p`.

```
class  A
  private int p
  protected |
```

(a) The user typing in the definition of `q`.

```
class  A
  private int p
  protected int q |
```

(b) The definition of the data member `q`.

Figure 3:21: The data member `q` as defined by the user.

The user configures the appearance of the nuggets to *show thick borders only on interaction* and opts to *use no connectors* as shown in Figure 3.22 by setting the options in the *Nugget Appearance Configurator* provided by the IDE. Figures 3.23a and 3.23b show the appearance of *class* nugget `A` with no thick borders and with and without connectors respectively.

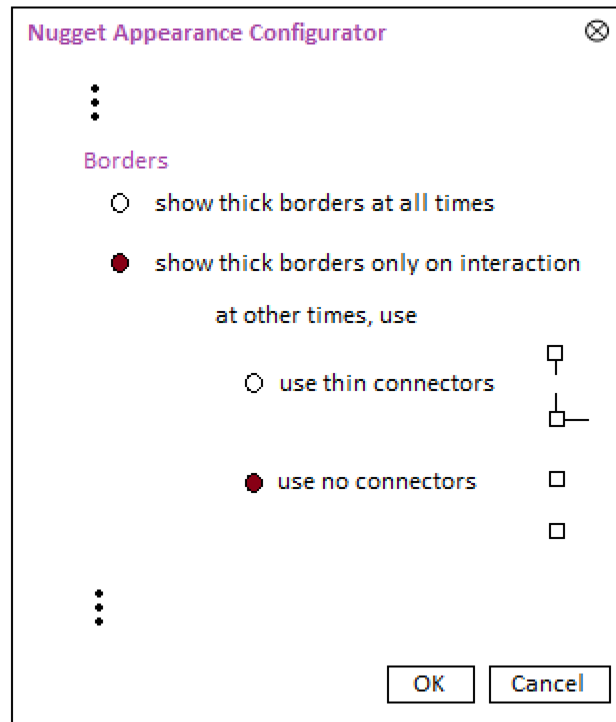
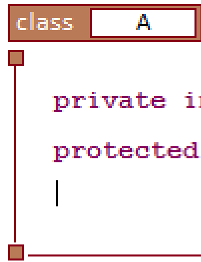
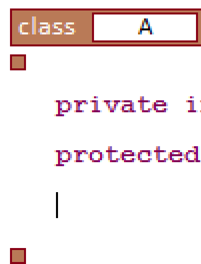


Figure 3.22: The Nugget Appearance Configurator window.



(a) The appearance of *class* nugget `A` without thick borders and with connectors



(b) The appearance of *class* nugget `A` without thick borders and connectors

Figure 3.23: The appearance of *class* nugget `A` after configuring borders with available options.

The user proceeds to insert an init-block in `class A` from the Constructs context menu as shown in Figure 3.24a and initializes `p` and `q` as shown in Figure 3.24b by typing in the assignment statements. The user then inserts a default constructor in `class A` and realizes the initial default actions performed upon the constructor invocation. The initial action involves calling the default constructor of the parent class. The user can click on the down arrow to replace the initial call with a call to an overloaded constructor of `class A` or the parent class. This is followed by executing the code in the initializer blocks. The user continues to compose the program to define `class A` and `class B` in the programming editor as shown in Figures 3.25 and 3.26 respectively. The user then reconfigures the appearance of nuggets to *use thin connectors*.

class A

private int p

protected int q

data member ...
constructor
method
init-block
static init-block
class
interface
enum

(a) Entering an init-block in a class

class A

private int p

protected int q

instance data member initializer block

p = 10
q = 20

(b) Initializing `p` & `q` in the init-block

Figure 3.24: Inserting an init-block and initializing the data members in it.

class A

private int p

protected int q

instance data member initializer block

p = 10
q = 20

default constructor of A

call default constructor of Object

run instance data member initializer blocks

System.out.println("In A's constructor")

System.out.println(p + " " + q)

public void method

int localVar = 5

System.out.println("In method" + localVar)

Figure 3.25: Class `A` in the editor.

```
class B extends A

    D   dReference

    default constructor of B
        call default constructor of A
        run instance data member initializer blocks
        System.out.println("In B's constructor")

    class C
        public void cMethod
            System.out.println("In Inner Class")

    public static void main String[] args
```

Figure 3.26: Class `B` in the editor.

The user, who is unfamiliar with the details involved with the creation an instance of an inner class, attempts to create an instance of `class C` conceptually by selecting the *create instance* action item from the context menu of `class C` as shown in Figure 3.27. Figure 3.28 shows a message displayed in a non-modal window in the editor, asking the user to click at a point in the code where an instance of `class C` is to be created. The user moves the mouse pointer to the beginning of the `main` method as shown in Figure 3.29 and clicks at this location to place the cursor and bring up a wizard, as shown in Figure 3.30, that guides the user through the creation of an instance of `class C`.

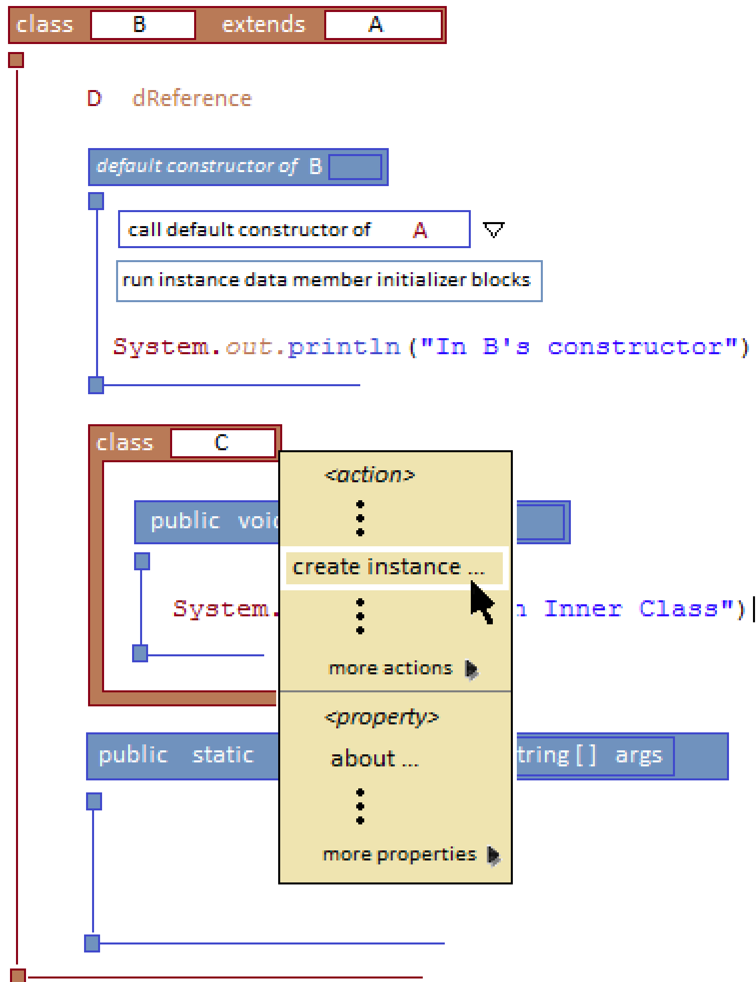


Figure 3.27: Creating an instance of an inner class by concept.

The user proceeds to create the instances of classes `B` and `C` by following the arrow indicators as shown in Figures 3.30 & 3.33 respectively. As guided by the arrow indicator shown in Figure 3.30, the user initially creates an instance of `class B` by interacting with the *create one* link as shown in Figure 3.31. This presents the user with the *instance creation* wizard, as shown in Figure 3.32, which allows an instance to be assigned to a reference or to be anonymous. The user proceeds with the default selection of assigning the instance to a reference by typing `b` in the textbox as the name of the. The user clicks the OK button closes the wizard, inserts the code to create an instance of `class B` and assign it to the reference `b` and moves the arrow indicator to the next step in the *instance creation* wizard, and to the next line in the programming editor as shown in Figure 3.33. In a

similar way, the user completes the creation of an instance of `class C` using the instance of `class B` created earlier.

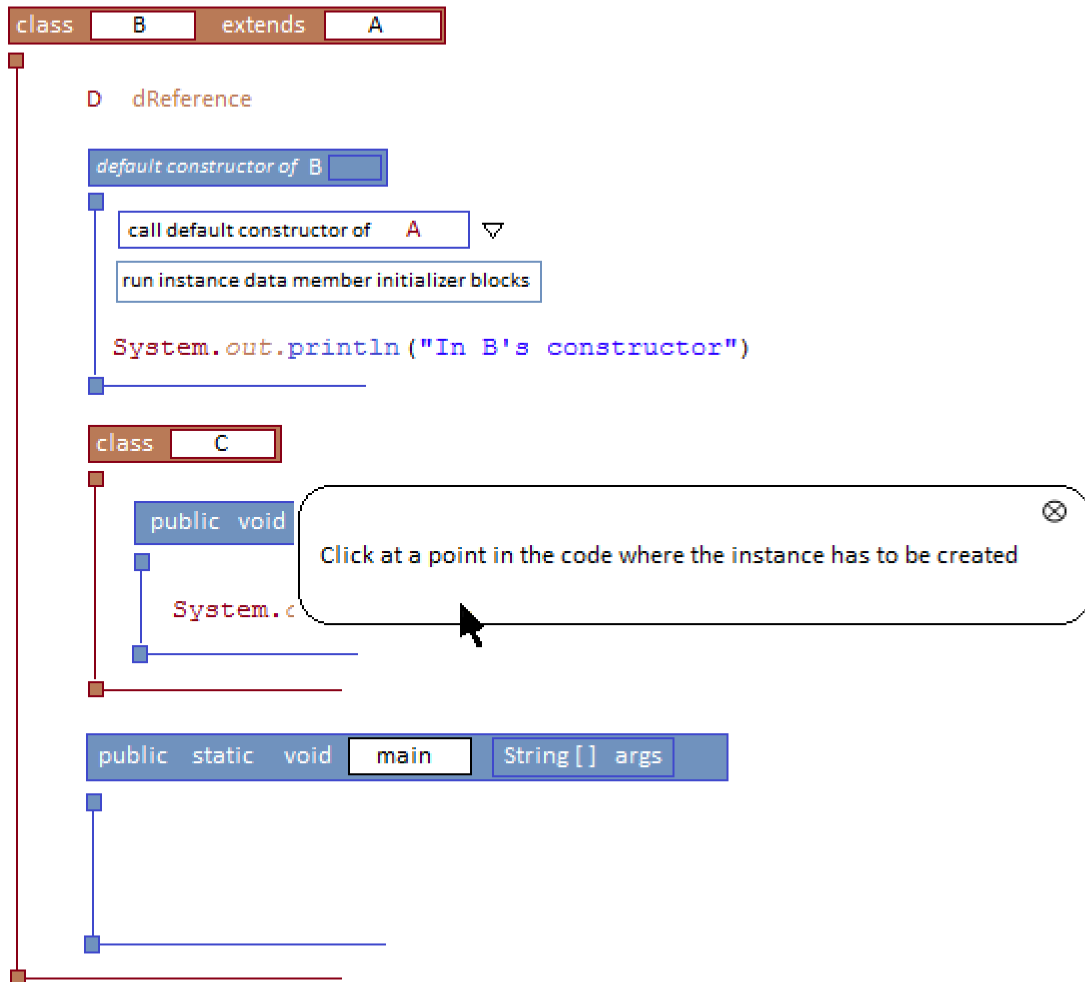


Figure 3.28: A non-modal message window that is displayed on conceptually creating an instance of a class.

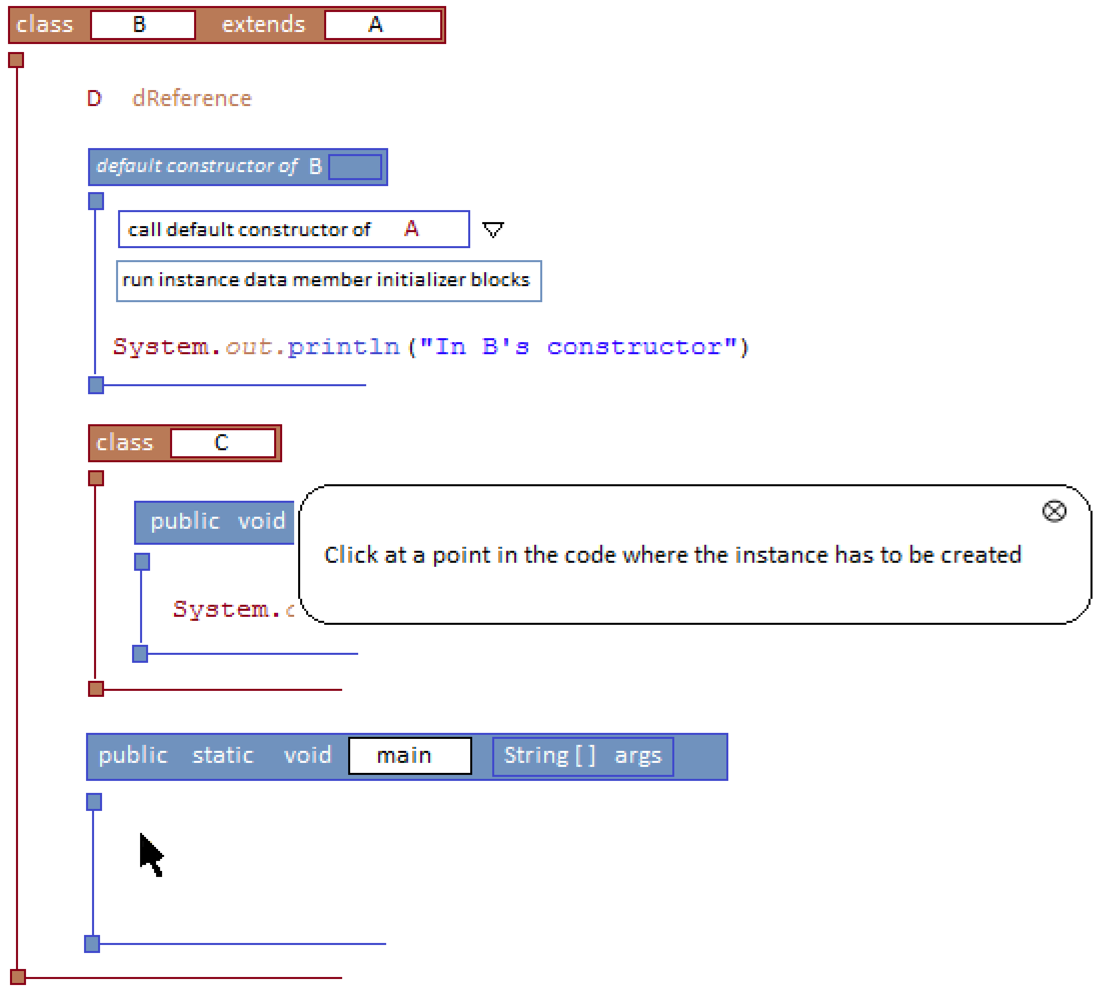


Figure 3.29: The user pointing the mouse at the location where the instance has to be created.

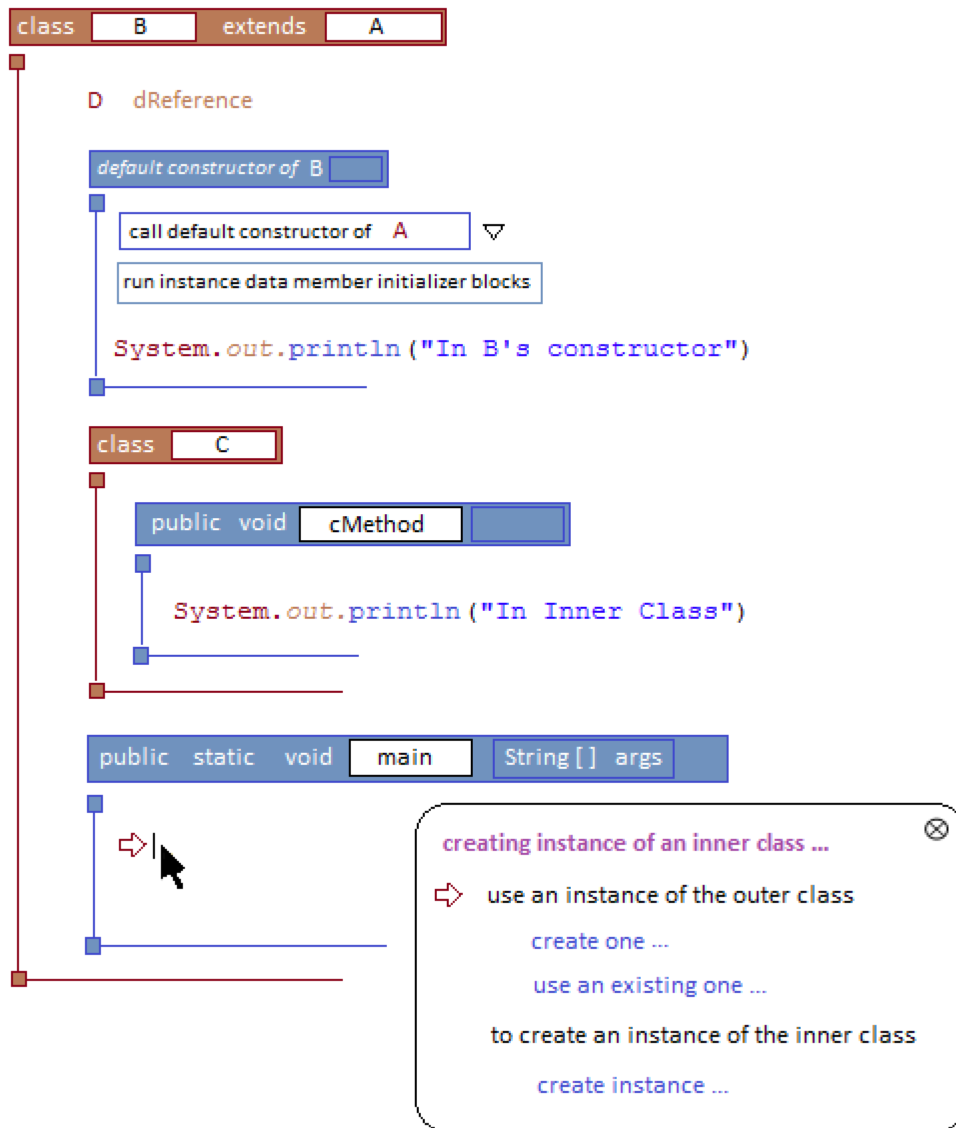


Figure 3.30: The user clicking at the location where the instance has to be created, to bring up the *inner class instance creation* wizard.

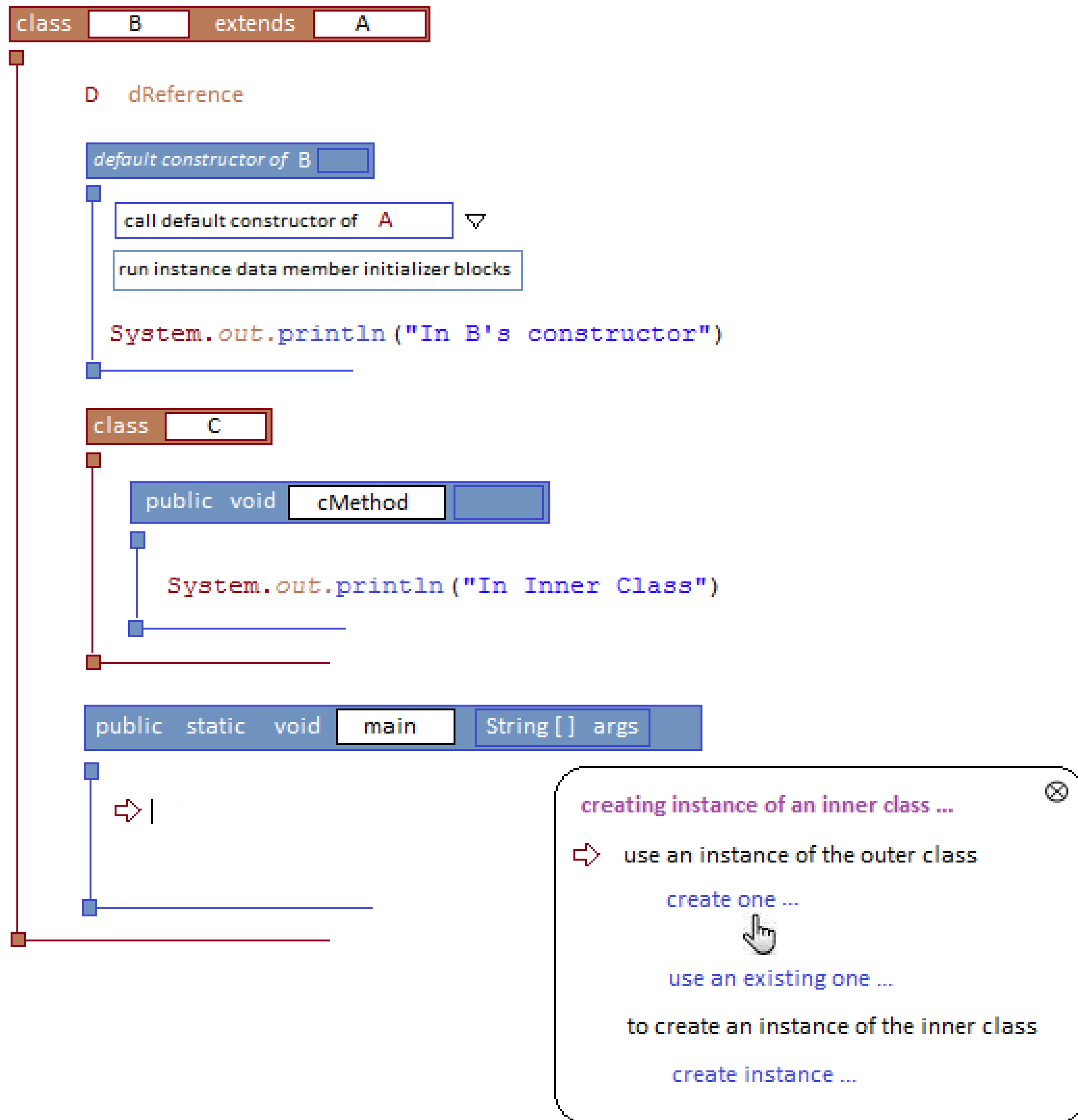


Figure 3.31: The user creating an instance of the outer class B.

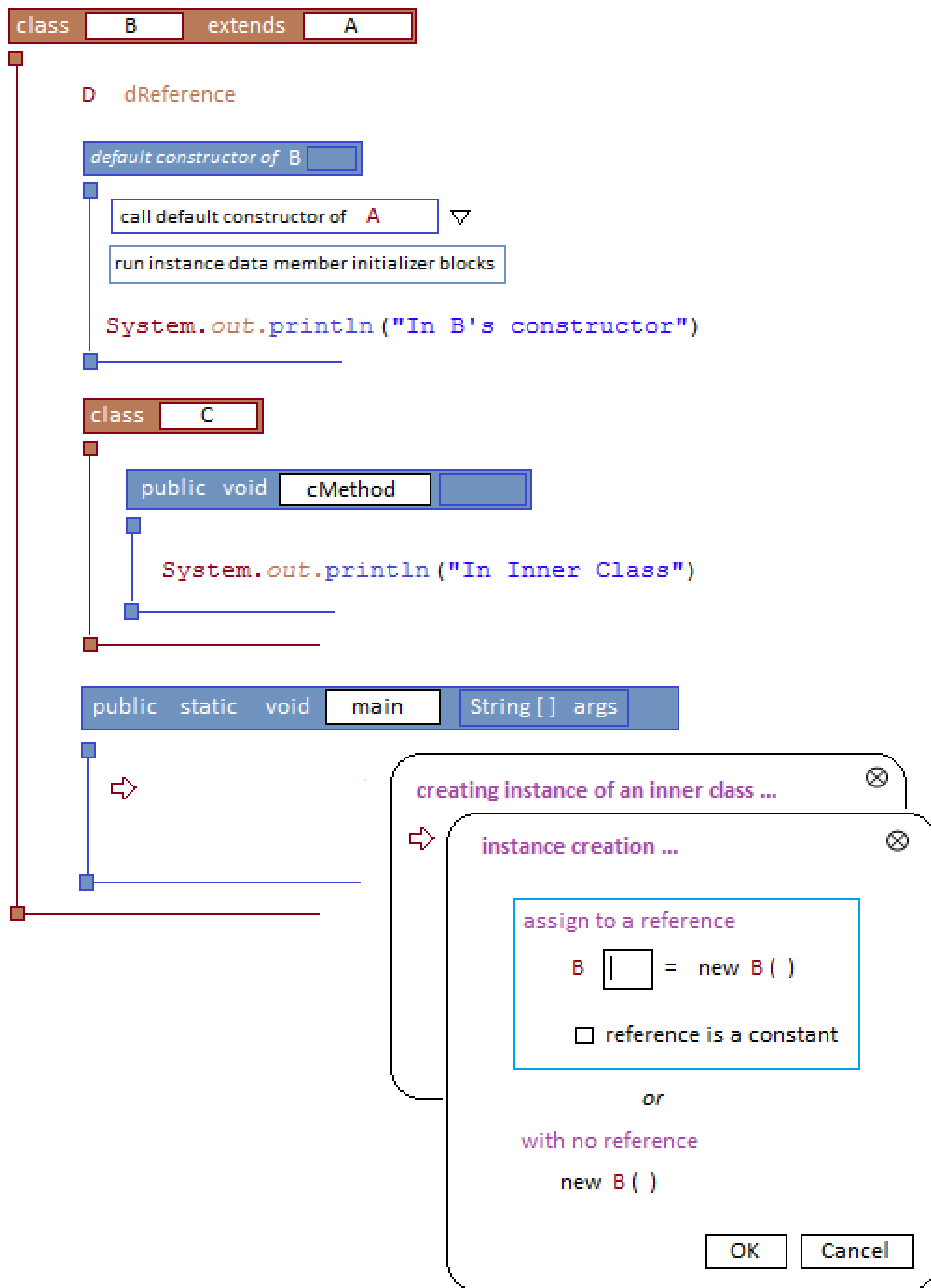


Figure 3.32: The *instance creation* wizard.

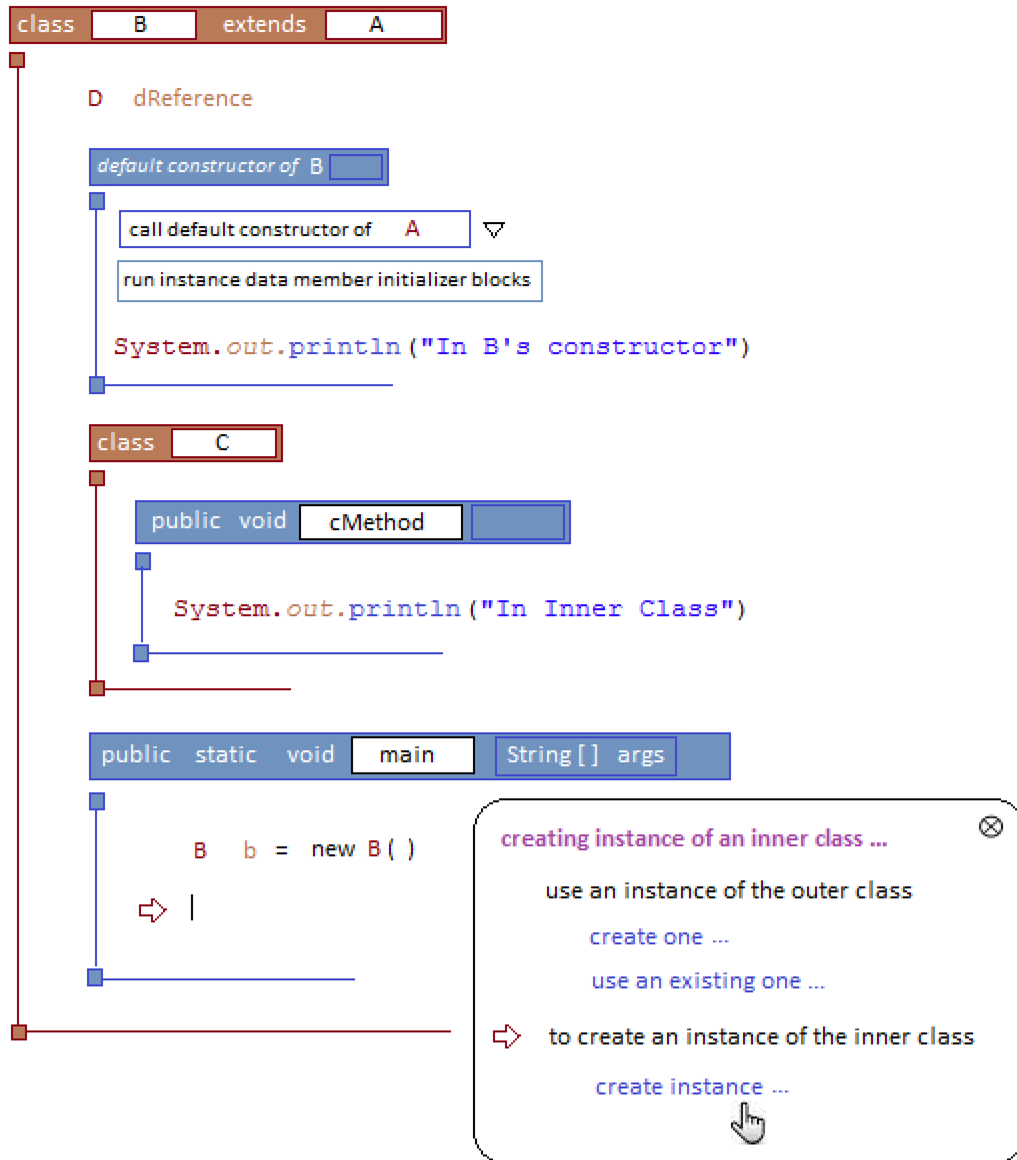


Figure 3.33: Creating an instance of the inner class `C`.

The user then decides to explore the details of `class C` by bringing up its context menu and selecting the *about* entry as shown in Figure 3.34 to view its details in a window as shown in Figure 3.35, including the fully qualified name of `class C` and the name by which it is saved on the file system. The user can also view the byte code for class c in a byte code editor by interacting with the *view byte code* link.

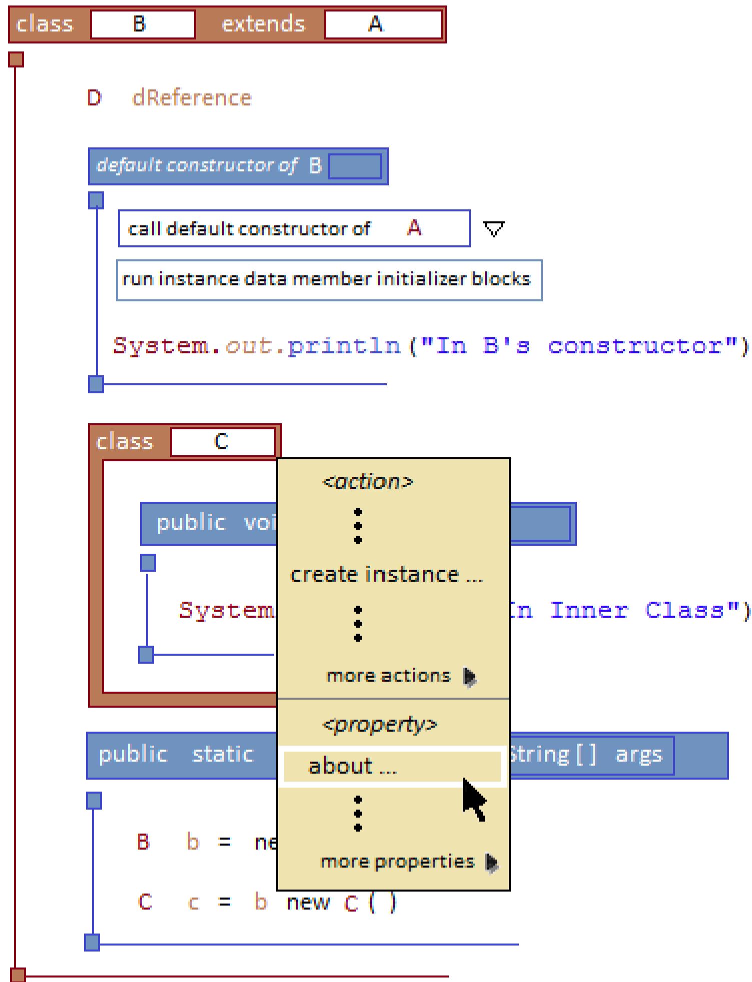


Figure 3.34: Selecting *about* from the context menu associated with `class C`.

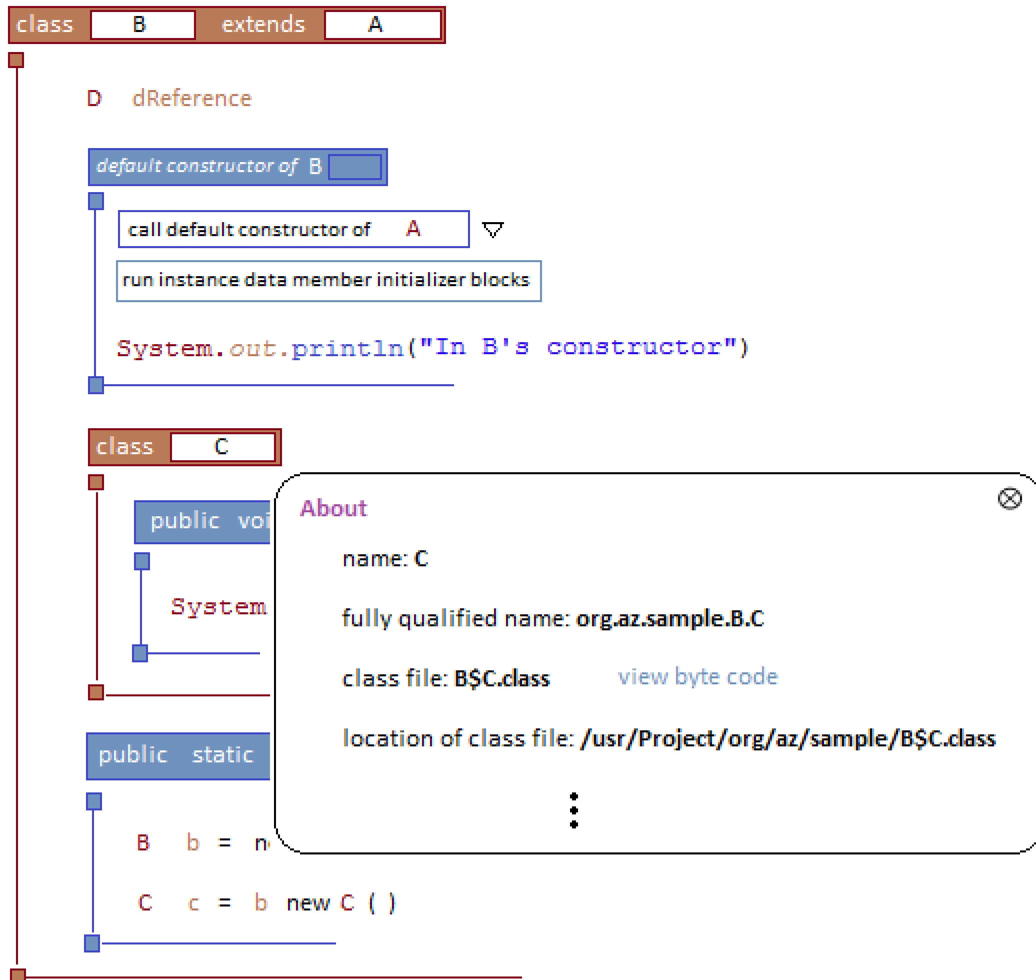


Figure 3.35: Details of inner class `C`.

The user continues to explore other elements in the program by interacting with the non-static data member `p` in `class A` to bring up its context menu and selecting the *highlight scope* contextual action as shown in Figure 3.36. This action highlights in green the parts of the code where `p` is visible and accessible as shown in Figure 3.37. The message displayed in a non-modal window shown in the figure lets the user know that the highlighting can be cleared by hitting the ESC key. The user hits ESC to clear the highlighting and selects the same action item on the local variable `localVar` to highlight its scope as shown in Figure 3.38 before clearing it.

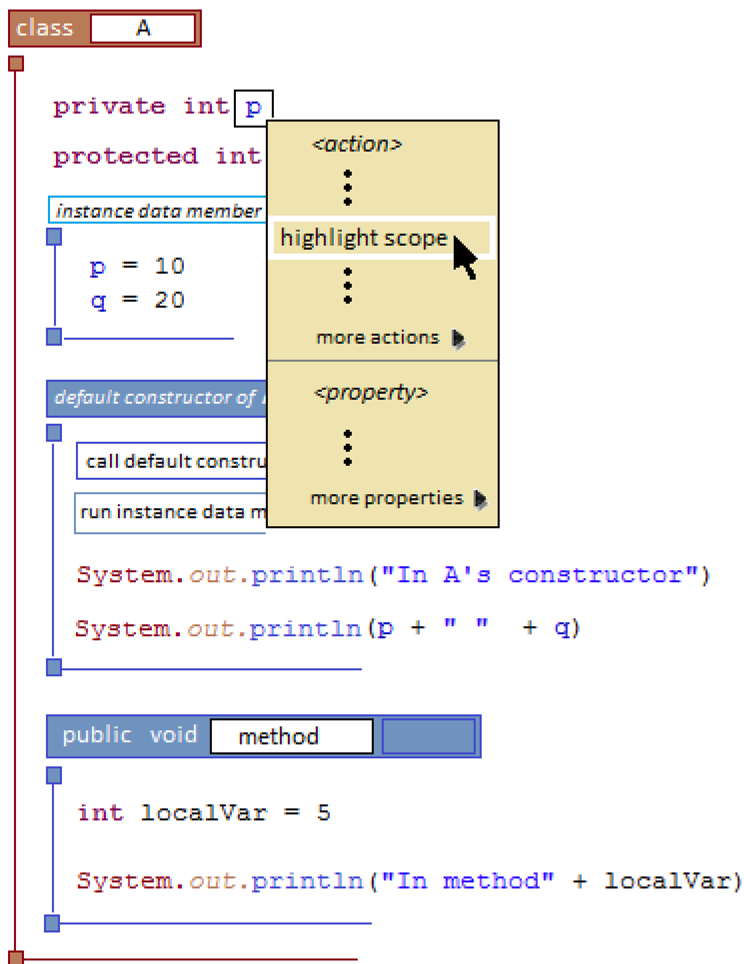


Figure 3.36: Selecting *highlight scope* from the context menu associated with a data member.

class A

```
private int p

protected int q
```

instance data member initializer block

```
p = 10
q = 20
```

Press ESC to clear highlighting

default constructor of A

call default constructor of Object

run instance data member initializer blocks

```
System.out.println("In A's constructor")

System.out.println(p + " " + q)
```

public void method

```
int localVar = 5

System.out.println("In method" + localVar)
```

Figure 3.37: The scope of `p` highlighted in green.

```
class A

    private int p
    protected int q

    instance data member initializer block
        p = 10
        q = 20

    default constructor of A
        call default constructor of Object
        run instance data member initializer blocks
        System.out.println("In A's constructor")
        System.out.println(p + " "  + q)

    public void method
        int localVar = 5
        System.out.println("In method" + localVar)
```

Press ESC to clear highlighting

Figure 3.38: The scope of `localVar` highlighted in green.

In order to understand the class resolution rules, the user selects the resolving rules action item from the Rules context menu of the System class in the program, as shown in Figure 3.39. Figure 3.40 shows the instantiated rules presented to the user, and the user further contextualizing them to understand how the `System` class is resolved by the Java compiler. The user learns from the contextualized rules that the class file of `System` is located in `rt.jar` under the directory `/usr/java/jdk/jre/lib/ext`. Figures 3.41 and 3.42 show similar interactions aimed at revealing the rules that govern the resolution of `class D`.

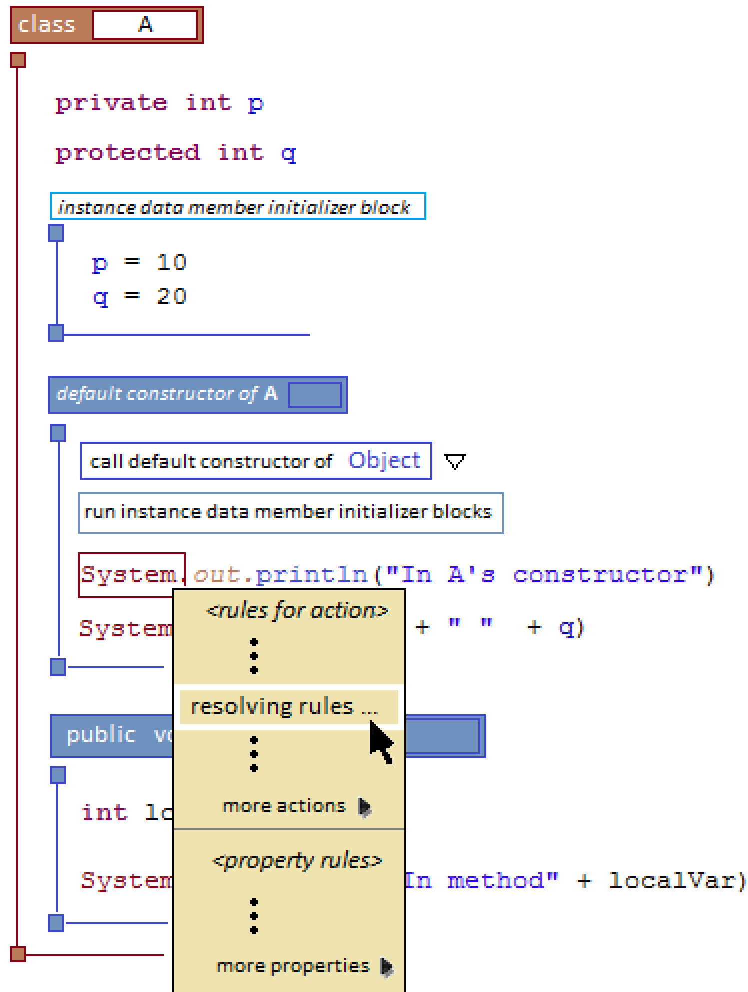


Figure 3.39: Selecting the *resolving rules* entry from the Rules context menu of `System`.

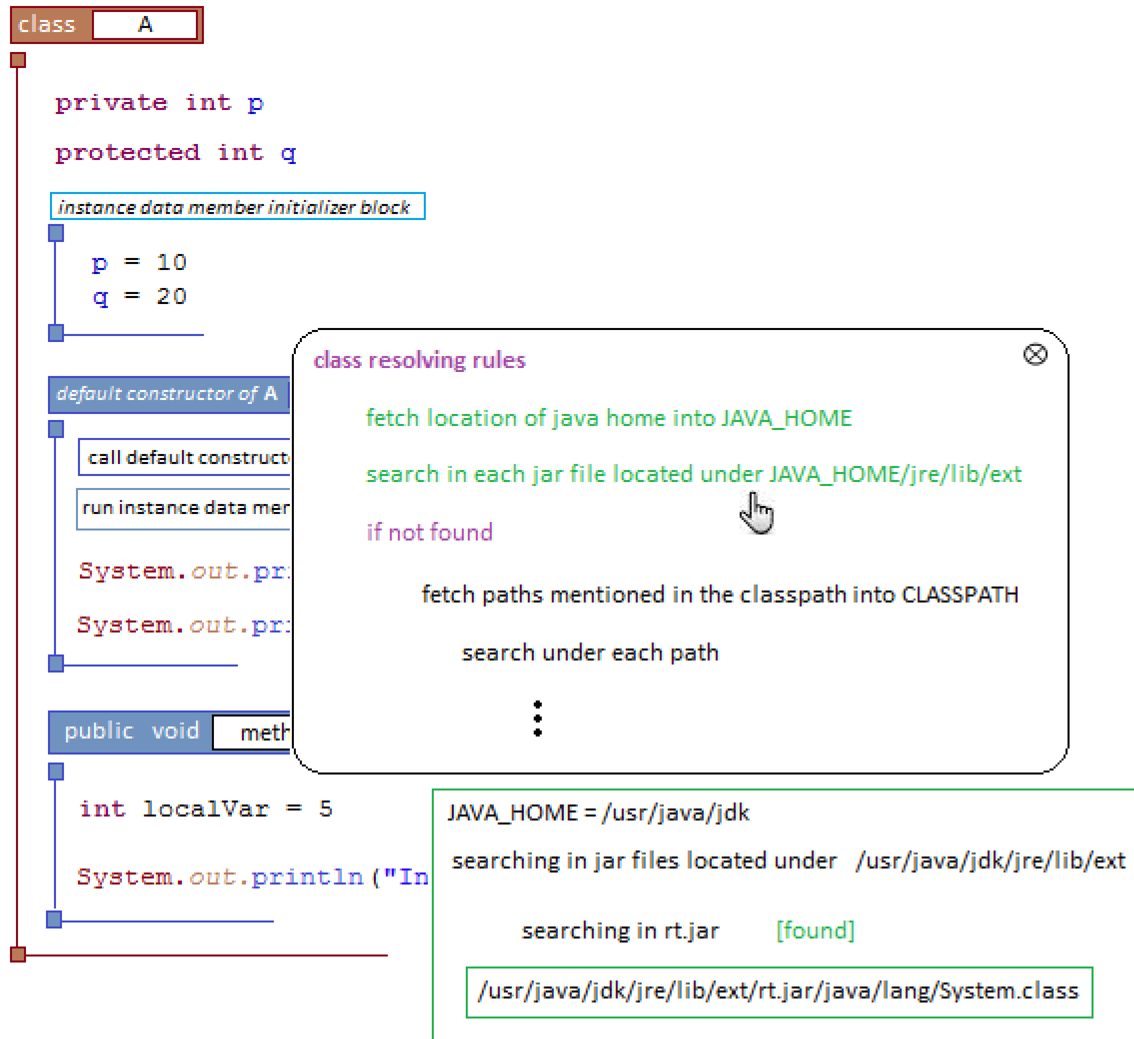


Figure 3.40: Instantiated and contextualized rules for resolving `java.lang.System`.

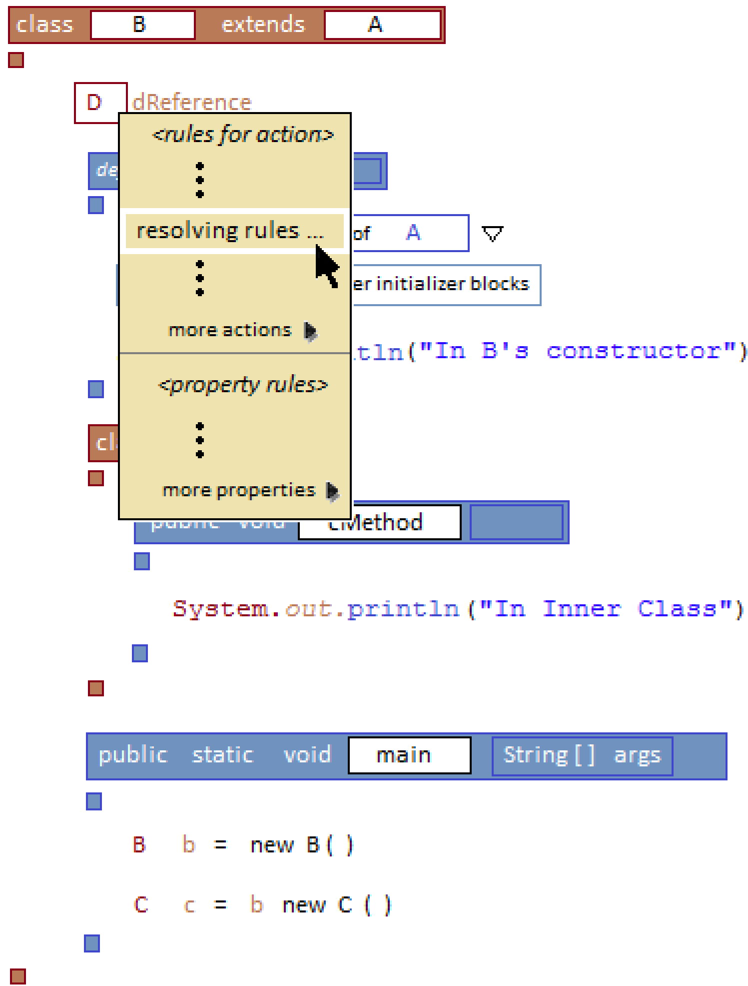
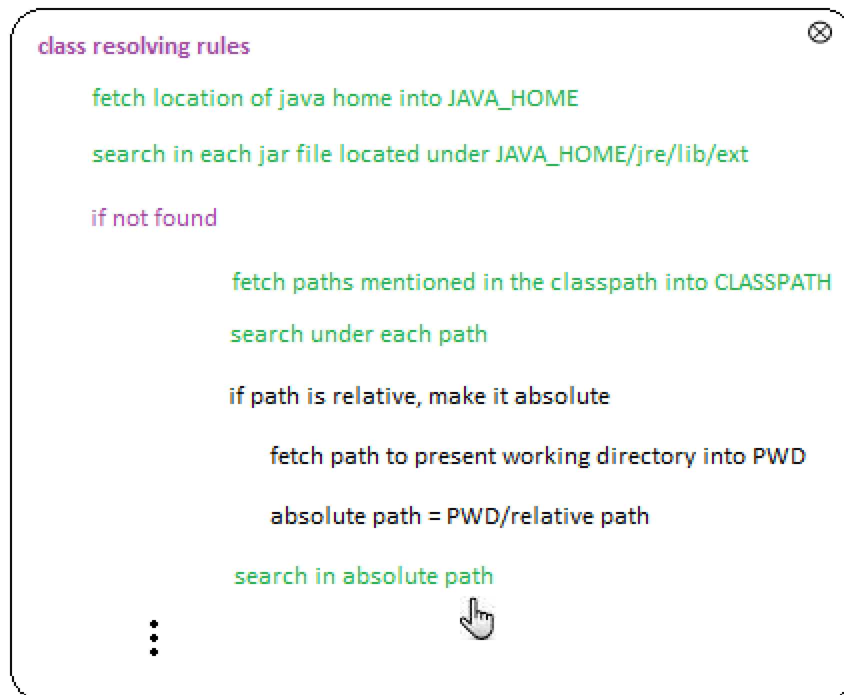


Figure 3.41: Selecting the *resolving rules* entry from the Rules context menu of D.



JAVA_HOME = /usr/java/jdk

searching in jar files located under /usr/java/jdk/jre/lib/ext [not found]

searching in rt.jar [not found]

⋮

CLASSPATH = /usr/Sample; /usr/Project

searching in absolute path /usr/Sample [not found]

searching in absolute path /usr/Project [found]

/usr/Project/org/az/another/D.class

Figure 3.42: Instantiated and contextualized rules for resolving class D.

Chapter 4

Az-Nuggets: An Application Framework

As mentioned in the previous chapters, Az-Nuggets could be implemented as an IDE for any one of a range of languages. Az-Nuggets IDE could support several different languages, and allow the user to program in any of them. To facilitate this, in this section we propose Az-Nuggets as an application framework [22] within which an "enabler" can incorporate a language into the Az-Nuggets IDE.

4.1 Overview

The framework of Az-Nuggets consists of the following main components that allow an enabler to design and activate nuggets for a programming language.

- The Editor Suite

  The Editor Suite is the interface to the framework of Az-Nuggets that lets an enabler visually design and activate nuggets for a language. It provides a Tool palette, a GUI editor and a code editor. The Tool palette provides the building blocks that allow nuggets to be designed in the GUI editor and saved to files. The code editor is used to define classes that attribute properties and behavior to nuggets. An abstract layout of the Editor Suite is shown in Figure 4.1.

- Context Menu Specifier

  Context menus associated with a nugget are specified using the Context Menu Specifier. A nugget is uniquely identified by the name of its class that is defined by the enabler in the Code Editor. Once a nugget of a language is defined, it can be opened in the Context Menu Specifier to specify the context menus associated with it. These include a regular context menu enumerating contextual properties

and actions of the nugget, a Rules context menu that allows the user to access rules related to contextual properties and actions, and a Constructs context menu that allows nuggets of other constructs to be inserted at a point of click within a nugget.

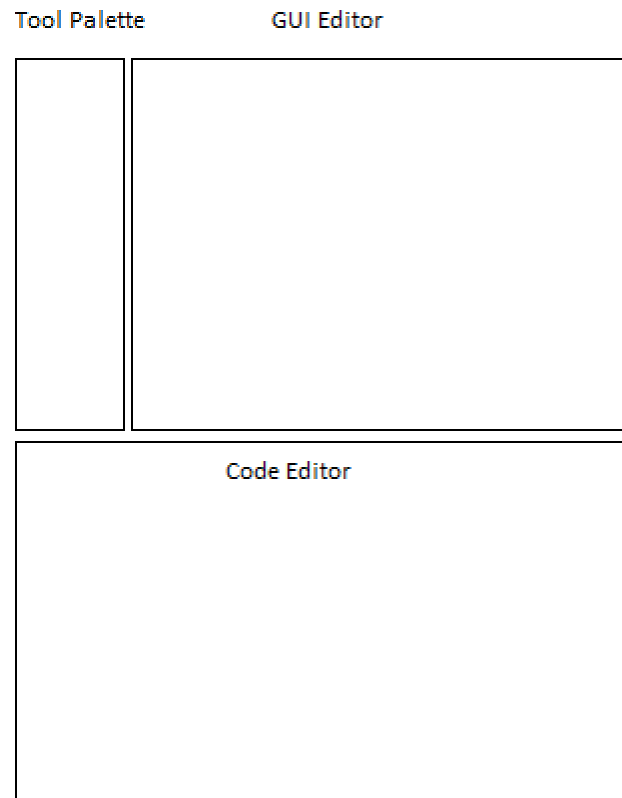


Figure 4.1: An abstract layout of the Editor Suite.

- Rules Specifier

  With the Rules Specifier, the enabler specifies rules in text that explain the syntax or the semantics of a contextual property or action associated with a nugget. The rules specified for a nugget are mapped to methods defined in its class to provide the instantiation and contextualization features to the user. The details of how the enabler performs this mapping are discussed in the following section.

- Nugget Panel Configurator

  The Nugget Panel Configurator helps the enabler add nuggets to the Nuggets Panel which is a part of the programming interface presented to the user upon launching the IDE. The Nugget Panel Configurator also allows the nuggets to be categorized by creating and adding categories to the category panel and attaching groups of nuggets to these categories accordingly.

- Screen Manager

  Instances of corresponding nugget classes are created by the framework when the nuggets are dragged and dropped onto the programming editor. Handles to these instances are passed to the Screen Manager which maintains a mapping between the nuggets in the programming editor to the corresponding instances in a way that any interaction with a nugget in the programming editor is communicated to the right instance.

4.2 Tentative Guidelines

A tentative set of guidelines that an enabler follows to implement a language in the framework are as follows.

1) Use the Editor Suite to design nuggets for the language.
2) Associate relevant context menus to the nuggets using the Context Menu Specifier.
3) Deposit syntactic/semantic rules into the Rules Specifier.
4) Complete the definitions of the classes generated by the framework to represent the nuggets.
5) Map the rules specified in step 3 to the methods from the classes defined in step 4 to instantiate and contextualize the rules for the user.
6) Use the Nugget Panel Configurator to add the activated nuggets to the Nuggets Panel to be used by the user.

Consider an example that illustrates how the enabler designs and activates a nugget for the class construct of Java by following the guidelines above.

1) The enabler uses the rectangle tool from the tool palette to design the *class* nugget as shown in Figure 4.2, adds a label 'class' to the nugget and a text box for the user to input the name of the class. The enabler saves the nugget by the name 'class' and the class that represents it by the name 'AzJavaClass' causing the framework to generate a class `AzJavaClass` as shown in the code editor. The framework instantiates `AzJavaClass` when the user drags & drops or inserts the *class* nugget into the programming editor.

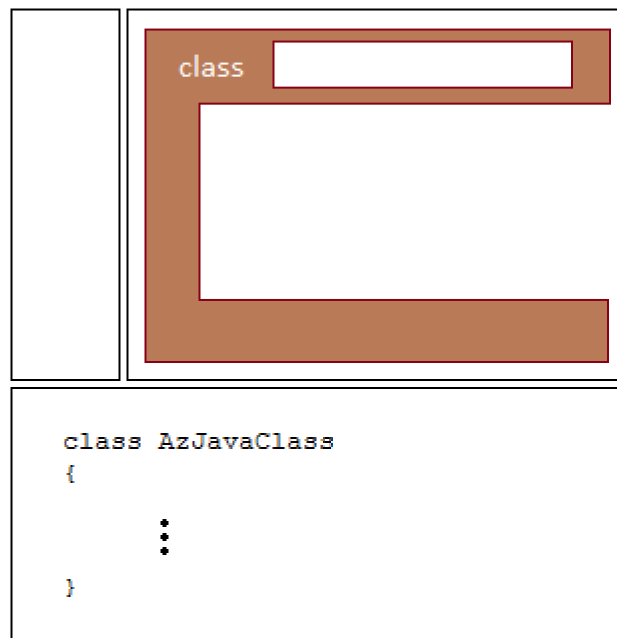


Figure 4.2: A snapshot of the Editor Suite with the nugget designed to represent a Java class.

2) The enabler selects the *class* nugget, as indicated by the diagonal pattern shown in Figure 4.3, and uses the Context Menu Specifier to add entries to context menu. Figure 4.3 shows the contextual action entries of the *class* nugget specified in English. The contextual properties of the nugget can be added by switching to the *properties* tab in the Context Menu Specifier. The enabler can specify the entries in a different language by

choosing the language from the *type in* dropdown box and even translate the entries to different languages by selecting the *Translate to Languages* link.

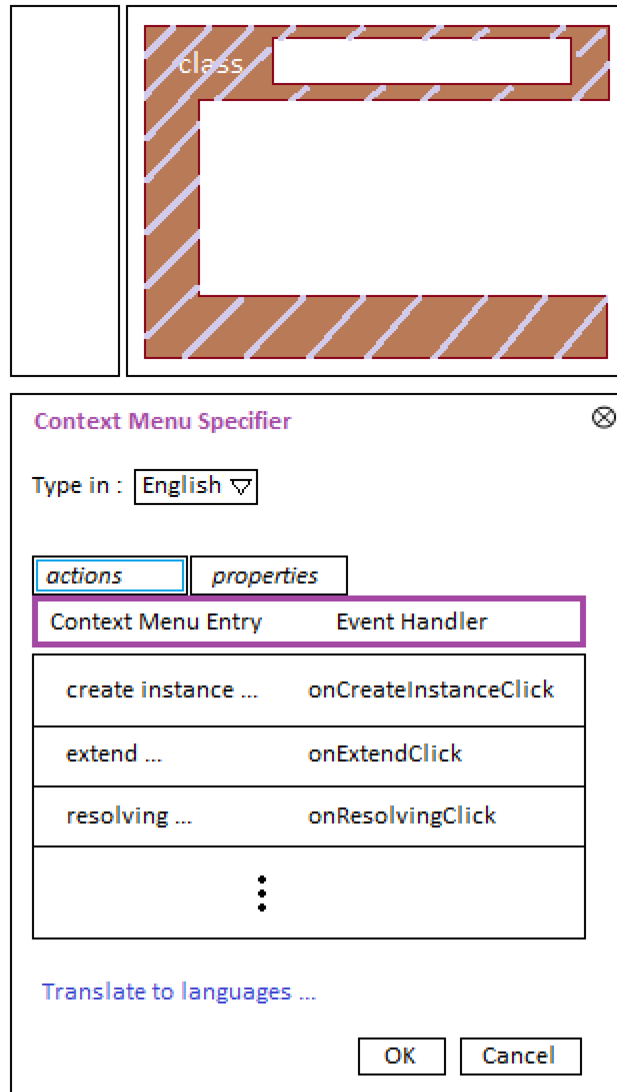


Figure 4.3: Specifying context menu entries using the Context Menu Specifier.

The enabler finalizes the entries by hitting the OK button to insert empty event handlers corresponding to each contextual entry into the class `AzJavaClass` as shown in Figure 4.4. The enabler defines these event handlers to respond to the users' selection of a contextual entry. Figure 4.5 shows a similar approach followed by the enabler to specify the contextual entries of the Rules context menu of the *class* nugget. These entries would be mapped to their corresponding rules at a later step.

```
class AzJavaClass
{
      ⋮

    onCreateInstanceClick
    {
          //method body
    }

    onExtendClick
    {
          //method body
    }

    onResolvingClick
    {
          //method body
    }

      ⋮
}
```

Figure 4.4: The code editor displaying the added event handlers.

The enabler proceeds to specify the entries of the Constructs context menu using the Constructs Context Menu specifier as shown in Figure 4.6. Each entry is similarly associated with an event handler, however, a default behavior can optionally be assigned by specifying the name of the class to be instantiated with an insertion of its corresponding nugget in the programming editor upon the selection of an entry by the user. Note how the class entry is assigned a default behavior to instantiate `AzJavaClass` with an insertion of the *class* nugget in the programming editor when the user selects it from the Constructs context menu.

The enabler continues to define nuggets and their representative classes for the rest of the entries in Figure by repeating steps 1 and 2.

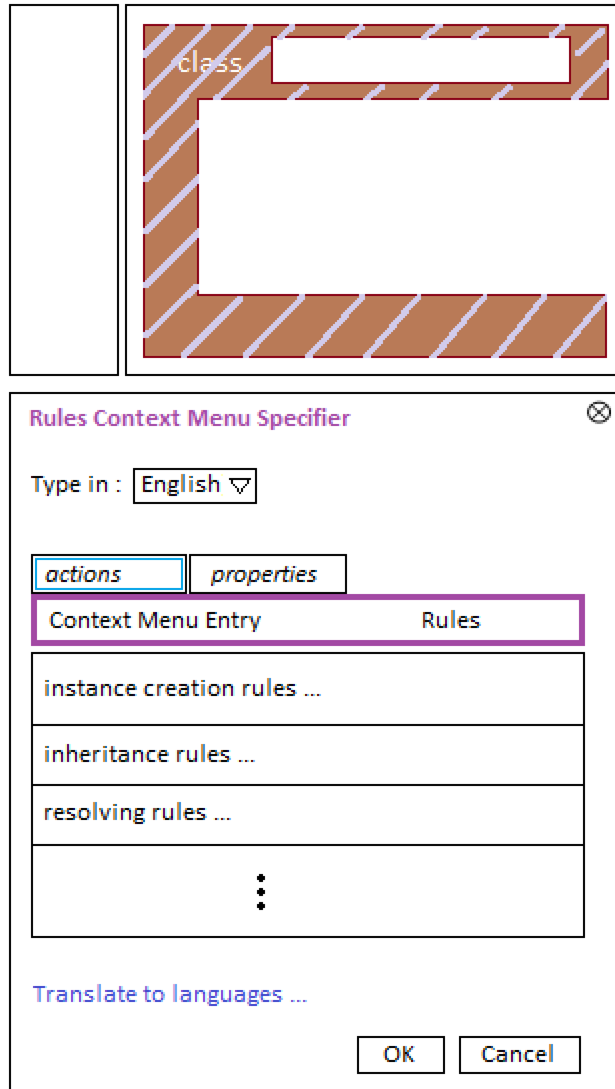


Figure 4.5: Specifying entries of the Rules context menu using the Rules Context Menu Specifier.

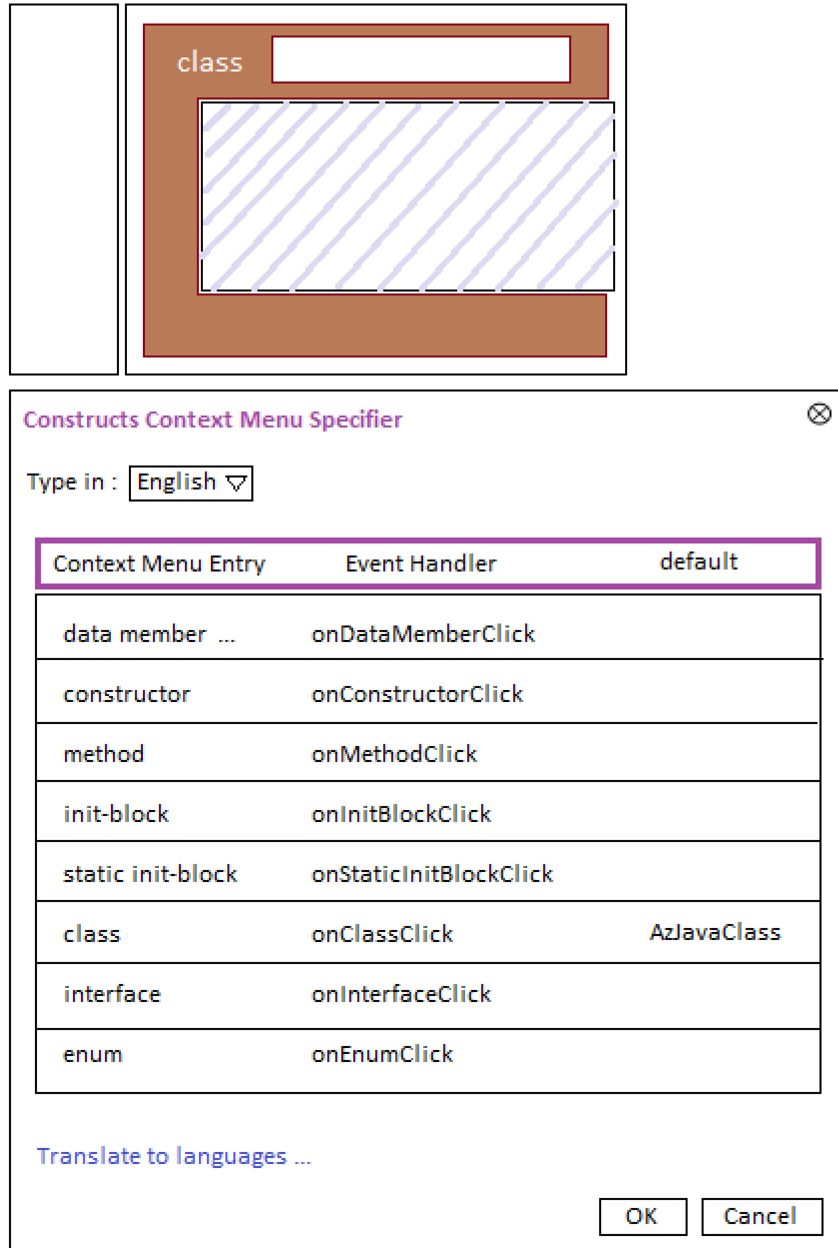


Figure 4.6: Specifying the entries for the Constructs context menu using the Context Menu Specifier.

3) The enabler uses the Rules Specifier to specify the rules and link them to their corresponding entries from the Rules context menu. Each rule is further associated with Instantiators and Contextualizers, which are methods defined in the class of the nugget that instantiate and contextualize the rule respectively. As shown in Figure 4.7, the enabler specifies the label of the nugget and the entry from its Rules context menu to link the

rules to. The figure also shows one of the rules specified for resolving a class mapped to its Instantiator method `fetchJavaHome` and its Contextualizer method `getJavaHome`, which are defined by the enabler in `AzJavaClass`.

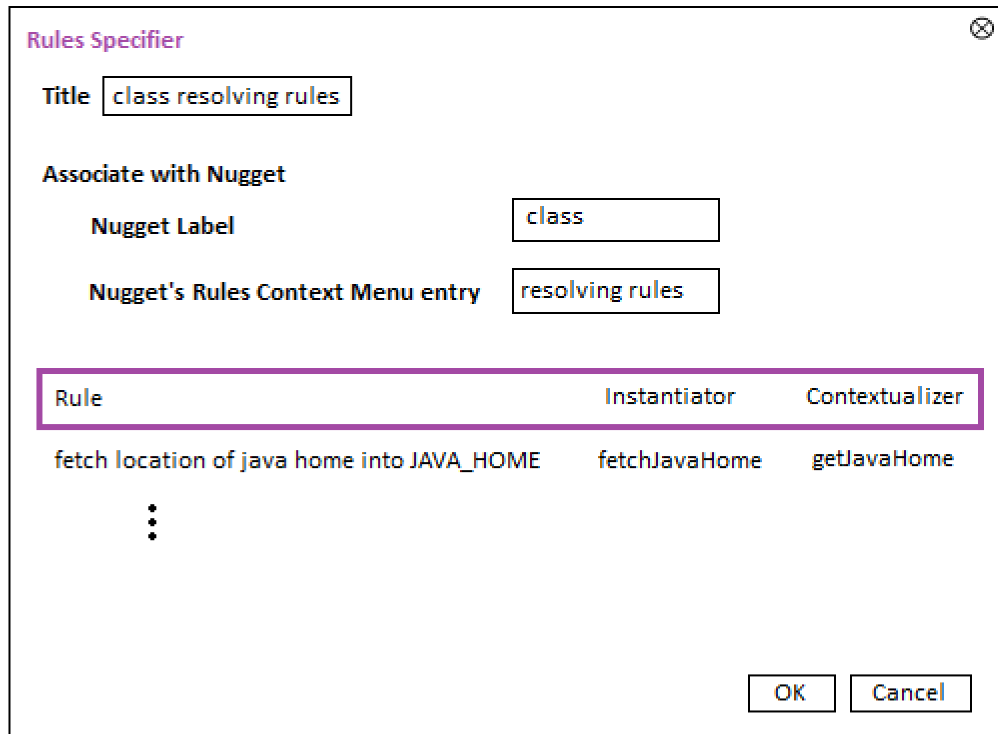


Figure 4.7: Specifying class resolving rules and their Instantiators and Contextualizers.

The framework instantiates the rules based on the execution state or the return values of their Instantiator methods. For example, the rule specified in Figure 4.7 is instantiated by the framework in Figure 3.40 in green indicating a return value from the method `fetchJavaHome` that caused the framework to interpret the rule as applicable and non-violated.

The framework executes the Contextualizer method associated with a rule when the user requests that it be contextualized. For example, the framework executes the method `get-JavaHome` when the user contextualizes the first rule in Figure 3.40 to provide the contextualized information in an inset.

The implementation of these methods traverses the instance structure of the program or accesses the configurations of the system to instantiate and contextualize the semantic rules. Enablers are responsible for the correctness of the implementation of these methods to convey accurate semantic rules to programmers.

4) The enabler continues to define `AzJavaClass` and the representative classes of other nuggets that can be inserted in a Java class.

5) The enabler uses the Nugget Panel Configurator as shown in Figure 4.8 to add the *class* nugget to the Nuggets Panel without associating it with any category. The enabler then hits the PLAY button to launch the programming interface as shown in Figure 4.9.

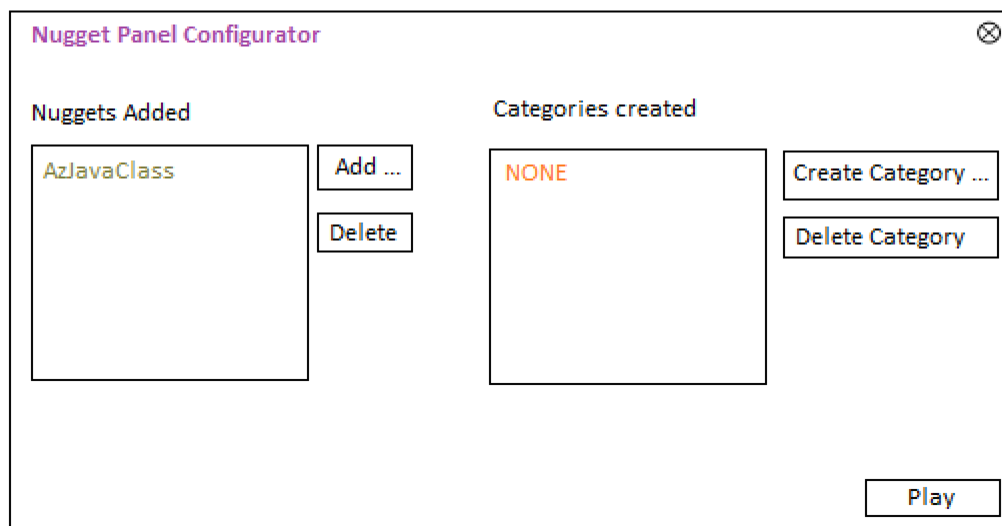


Figure 4.8: Adding *class* nugget to Nuggets Panel using the Nugget Panel Configurator.

6) The enabler verifies the entire process by dragging and dropping the *class* nugget onto the programming editor and interacting with it as shown in Figure 4.10.

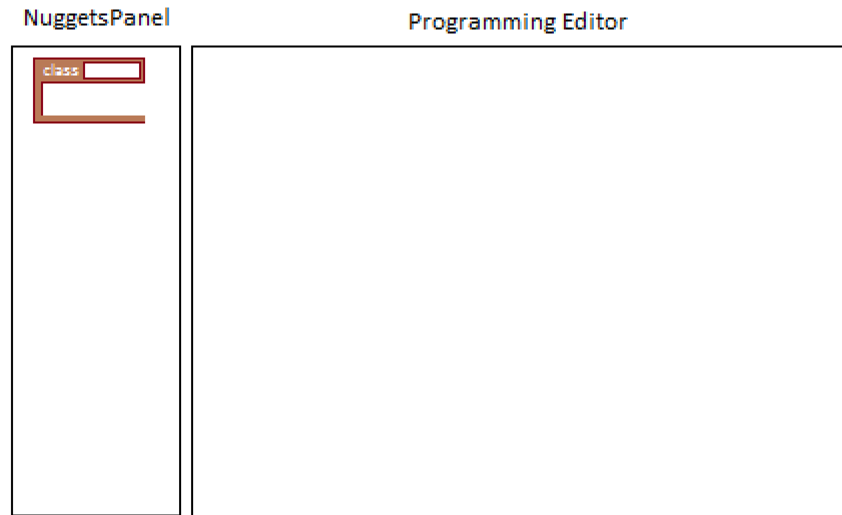


Figure 4.9: The Programming Interface displaying the *class* nugget in the nuggets panel.

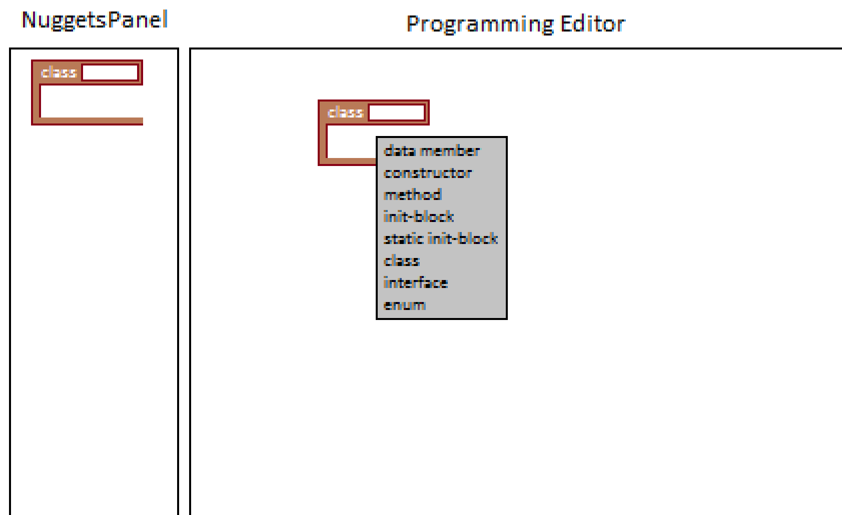


Figure 4.10: Dragging and dropping the *class* nugget onto the Programming Editor and interacting with it to bring up its Constructs context menu.

4.3 Summary

In summary, the framework offered by Az-Nuggets is a collection of APIs and tools that offer the following benefits to the enabler.

- Provides a default organization of nuggets in the programming editor

- Automatically instantiates nuggets when they are dragged and dropped or inserted in the programming editor
- Maintains a mapping between the nuggets in the programming editor and their instances in a way that any interaction with a nugget in the programming editor is communicated to its corresponding instance.
- Organizes the instances in a way that reflects the nugget structure in the programming editor.
- Factors out common concepts into generic nuggets that can be customized to implement language-specific behaviors if necessary.

Chapter 5

Conclusion

With Az-Nuggets, programmers can assemble programs with nuggets that are associated with contextual information that enables programming by concept and the exploration of the syntactic and semantic details of a language. Programmers can view the rules associated with a contextual property or an action of a nugget. These rules are instantiated to confirm their applicability in a particular context of a program, and can be contextualized to further convey why the rules apply. With these features, the user can explore language-related features and semantics from within the IDE without having to refer to programming language manuals or search the web.

Az-Nuggets is also an Application Framework that provides enablers with a collection of APIs and a set of tools to design for a language. These tools can be used to design nuggets, associate contextual information, specify and tag rules to contextual entries and provide nuggets to programmers. The instantiation and maintenance of instances of the classes representing nuggets are handled by the framework when the user drags and drops or inserts these nuggets into the programming editor. The user's interactions with nuggets in the programming editor are communicated to the corresponding nugget instances by the framework by sending appropriate messages. The framework lets enablers focus on writing code for activating nuggets by offering these functionalities.

5.1 Future Work

Az-Nuggets will be implemented with the proposed features and be user-tested to evaluate its performance and usability. It would be interesting to investigate the application of these how well these features apply to declarative languages like logical and functional

programming languages including the languages that support one or more programming paradigms [24].

Another interesting feature that might be offered with Az-Nuggets is Video commenting, giving users the ability to add video comments to the code in a way similar to adding textual comments. We make the following claims about video comments that would need to be verified by conducting user studies:

- Video comments are more engaging for users to make as they interact with the application directly and comment on it rather than describing it abstractly in text. It involves recording direct actions in actual configurations rather than providing the details in text.

- They capture additional helpful details that could otherwise be ignored in textual comments. For example, a video comment on a web application captures details about the tools and techniques used by the programmer to work with the web application without the programmer having to explicitly state them in the comment.

- Video comments are faster to record than equivalent textual comments providing the same level of detail.

- Programmers could revisit their thought processes and design decisions less effortlessly with video comments than their textual counterparts.

- Programmers could point at subject and comment on it and its relation with other elements in the domain by bringing them into one visual context.

- Video comments, in comparison with textual comments, keep the code clean as no textual comments are needed in between the lines of code. The amount of information that could be delivered in a video comment if translated to text accounts for lengthy textual comments that make code hard to read.

- The effect of any aspect of a program or a snippet of code on its output could be recorded effectively.

Finally, the application framework of Az-Nuggets will be implemented to provide more flexible tools and APIs to minimize the effort required from enablers to provide a language in Az-Nuggets. This will attract more enablers to contribute support for various languages in Az-Nuggets.

Bibliography

[1] B. David and K. Michael, *Objects First with Java: A Practical Introduction Using BlueJ*, 5th ed. Boston: Pearson, 2012, pp. xvi.

[2] B. Karen and M. Resnick, “New frameworks for studying and assessing the development of computational thinking,” in AERA, 2012.

[3] C. Stephen et al., “Teaching Objects-first In Introductory Computer Science,” In Proc. SIGCSE 2003, Reno, Nevada, USA, 2003.

[4] D. Jim, *The Java Developer's Guide to Eclipse,* 2nd ed. Boston: Eclipse, 2005, pp. 1.

[5] D. Wanda et al., "Mediated Transfer: Alice 3 to Java," In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, ACM, 2012.

[6] G. Roger et al., “Transforming Source Code Examples into Programming Tutorials,” Presented at CCGI 2011. The Sixth International Multi-Conference on Computing in the Global Information Technology, Luxembourg City, 2011.

[7] H. Poul and K. Michael, "Greenfoot: Combining object visualisation with interaction," In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ACM, 2004.

[8] K. Michael, *Introduction to Programming with Greenfoot Object-Oriented Programming in Java™ with Games and Simulations*, 1st ed. New Jersey: Prentice Hall, 2010, pp. 14.

[9] K. Michael, "The Greenfoot programming environment," in *ACM Transactions on Computing Education (TOCE)*, 2010.

[10] K. Michael, "Using BlueJ to introduce programming," in *Reflections on the Teaching of Programming*, Springer Berlin Heidelberg, 2008, pp. 98-115.

[11] K. Michael and John Rosenberg, "Guidelines for teaching object orientation with Java," in *ACM SIGCSE Bulletin*. Vol. 33. No. 3. ACM, 2001.

[12] L. Yu Bin and D. Xinfa, "Research on the IDE of Visual Programming Language," Advanced Materials Research, 219-220, 2011, pp. 140.

[13] L. Liz et al., *App Inventor: Create Your Own Android Apps*, 1st ed. California: O'Reilly, 2011, pp. xxii.

[14] M. John and M. Resnick, "The scratch programming language and environment," in *ACM Transactions on Computing Education (TOCE)*, 2010.

[15] M. John and L. Burd, "Scratch: a sneak preview," in *Creating, Connecting and Collaborating through Computing, 2004. Proceedings. Second International Conference on*. IEEE, 2004.

[16] M. Rausch, "AgentSheets–Programming above C-Level," In *Computer Graphik Topics* 10, Vielen Bereichen, 1998, pp. 10-12.

[17] P. Jeffrey, "Alice: easy to use interactive 3D graphics," in *Proceedings of the 10th annual ACM symposium on User interface software and technology*, ACM, 1997.

[18] R. Alexander, "Making Programming more Conversational," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC '11, IEEE Computer Society*, Los Alamitos, CA, 2011, pp 18-22.

[19] R. Alexander, "Demonstration of Conversational Programming in Action," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC '11, IEEE Computer Society*, Los Alamitos, CA, 2011, pp 18-22.

[20] R. Alexander and I. Andri, "Behavior Processors: Layers between End-Users and Java Virtual Machines," in *Proceedings of the 1997 IEEE Symposium of Visual Languages, Computer Society,* Capri, Italy, 1997, pp. 402-409.

[21] R. Alexander, "Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments," Ph.D. dissertation, Dept. of Comp. Sci., University of Colorado at Boulder, 1993.

[22] T. Lewis, *Object-Oriented Application Frameworks*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 1995.

[23] T. Jason, *Google App Inventor for Android*, U.S.A: Wiley, 2011.

[24] V. Roy and H. Seif, *Concepts, Techniques, and Models of Computer Programming*, U.S.A: The MIT Press, 2004.