Deverywhere: Develop Software Everywhere - A Template-Based Developing Abstraction

Alex Tilkin

November 2015

MASTER THESIS

Weston School of Computer Science

Supervisor 1: Prof. Shmuel Tyszberowicz

Supervisor 2: Dr. Yishai Feldman (IBM Research)

Preface

This work is a Master's thesis in MTA (Academic College of Tel-Aviv Yafo) as part of the study program M.Sc. in Computer Science. It was carried out during 2014-2015. The academic supervisor of this work is Prof. Shmuel Tyszberowicz. The idea for this thesis yielded by Dr. Yishai Feldman (IBM Research) who is also a supervisor in this work. This research is in cooperation with Ari Gam who is an M.Sc. student in Computer Science at Tel-Aviv University. The readers of this document are assumed to have knowledge in programming languages, system engineering, and software development tools.

Tel-Aviv, 1-11-2015

(Your signature)

Alex Tilkin

Acknowledgment

I would like to thank Prof. Shmuel Tyszberowicz, Dr. Yishai Feldman, and Ari Gam for their great help during the research.

Abstract

In this work we developed an idea of a new approach to programming, where the programmer may use mobile devices for programming in addition to PCs and laptops. In this research we related to two major problems: how to present the code on mobile devices, and how the programmer interacts with the mobile device while s/he programs. Previous works have failed to address this issue because they concentrated on the dictation of the code word by word instead of describing what transpires when using natural language. Our approach is to use natural language for interaction between the mobile device and the programmer. During the research we studied programming environments, performed experiments, built concepts for compact representation of the code, and constructed a prototype in which one can dictate code in natural language and it will understand what is meant. This research is important for the development of programming environments, it may present a solid foundation for future work in this domain both for students and for researchers. The aim of this thesis is to investigate the domain of programming environments, and to develop a concept of a new generation of programming environments. The idea is to use mobile devices such as tablets and smart phones as development tools, in addition to PCs and laptops that are already used today and to integrate all of them into one comprehensive solution.

In order to perform such a study, we had to investigate and learn different programming environments as well as the current state of programming. Secondly, we had to understand what the obstacles in developing a programming environment on mobile platforms are, and to provide a solution to every problem. We found that in order to allow programmers to use mobile platforms as development environments, we have to use voice and touch gestures. In order to learn how programmers use speech as a gesture while programming, we performed a set of experiments. Those experiments gave us a sense of the optimum interaction which can be achieved between a computer and a programmer.

Afterwards, we started to investigate the most dominant and common features that are used today in programming environments. Based on those features, we designed concepts of compact representation of the code so it could fit according to the sizes of mobile devices. Based on the concepts of compact representation of the code, we decided to add the possibility of config-

uring the representation of the code so the programmer could read to code in the way it makes most sense to him or her.

We investigated the fields of natural language processing and programming languages and built two modules. The first one is a speech to text engine which translates spoken commands to textual commands. The second one is a module that accepts textual commands, validates and resolves them. Those two modules are significant to this study because they are keys of success in this study. They will be used in future work.

This work has been published in an academic article entitled "Deverywhere: Develop Software Everywhere".

Based on the research and its results, we are confident that it is possible to develop a full-scale programming environment that will be used on mobile devices, PCs and laptops. In addition, the programmer will be able to seamlessly switch devices in order to proceed with his or her coding.

Contents

Chapter 1

Introduction

In the early days of computing, programmers had to work in offices. Personal computers allowed programmers to work at home as well. Laptops further expanded the working environment, and we often see people programming in coffee shops, terminals, trains, and airplanes. With ubiquitous mobile devices becoming increasingly popular, there is an opportunity to allow programmers to work in even more restrictive environments. While such small devices are unlikely to become the preferred working environment, they can be useful in circumstances where urgent action is required and other equipment is unavailable.

This scenario presents two major obstacles: first, the lack of a convenient keyboard; and second, the small screen space, which limits the amount of code that can be shown simultaneously. Some have advocated the creation of new programming languages for mobile platforms, but the cost of adopting a new language, with its related tools and infrastructure, seems to be too great for the benefit of occasionally programming on a mobile device. This applies to the development of mobile and non-mobile applications alike; professional programmers who develop mobile applications still prefer to use large screens and physical keyboards. Instead, we focus on easy ways to use existing languages, such as Java, on mobile devices. Our proposed solution, called **Deverywhere**, addresses both challenges, by using templates to make voice and touch input very effective for programming, and for showing much more code in a limited space. Templates, used in context, allow voice input for creating, editing, and navigation; and allow a compact representation of programs that makes maximum use of the given screen space. Both uses require a high degree of configuration, since programmers have different preferences re-

garding the way they want to voice and see programs. The underlying representation is always the original language, so that each programmer can see a tailored view while seamlessly collaborating on the same code with others.

These ideas are also relevant to programming on laptop and desktop systems, for people with disabilities such as repetitive-stress injuries (RSI) that limit keyboard usage, and partial vision loss, which requires the use of very large fonts. For some programmers, no screen is large enough, and so we expect that these programmers will use the compact representation of code even on large displays.

1.1   Background

In-order to create such an IDE we need to investigate several areas. The first one is, what are the features that we need in such a developing environment? The second one is, how do programmers tend to describe the code that they want to insert? And the third one is, how can we create a new representation of the code without harming the understanding of it? We need to investigate all those areas in order to design a new environment for developing with a new approach that uses voice and touch gestures.

Several studies have been published that have related to this problem. Some of the studies deal more with research and some deal essentially with application. Susan L. Graham and Andrew Begel from Berkeley University worked on a project named *SPEED*. In their study they developed an add-on that integrates into Eclipse and allows the programmer to insert lines of Java code using speech. Sihan Li, Tao Xie (North Carolina State University) and Nikolai Tillman (Microsoft Research) worked on TouchDevelop. This project provides a simple and clear environment which allows the developer to develop application directly on mobile devices using touch gesture. Dennis Strein and Hans Kratz developed appfour, an IDE that runs on mobile devices. The user can compile applications directly on the mobile device and run them.

None of the works that are described above provides a comfortable solution for IDE on mobile devices that is fully integrable with stationary computers. None of the works addresses the issue of lack of keyboards and small screen. Our work addresses this issue by providing a comfortable IDE which uses voice, touch and compact representation of the code.

Dictation systems exist today, but their use for programming is limited. Lacking any domain knowledge, they require most of the program to be dictated letter by letter, which is impractical. By building an understanding of program syntax and some semantics into the dictation tool, it is possible to make this process much more efficient.

1.2 Problem Formulation

Programming on mobile devices presents two major obstacles: the lack of a physical keyboard, and the small screen space, which limits the amount of code that can be shown simultaneously. This work addresses both challenges, and offers a method to enable programming on mobile and other devices with limited input and output capabilities. The method uses templates to make voice and touch input very effective for programming, and showing much more code in a limited space. These ideas are also relevant to programming on laptop and desktop systems, for people with disabilities such as repetitive-stress injuries (RSI) that limit keyboard usage, and partial vision loss, which requires the use of very large fonts.

In this work we concentrate on several targets: design of a new representation of the code so it will fit on mobile screens and will be readable as well; Creation of a set of templates that will allow the programmer to program by dictating the code; and allow the user to configure the representation of the code.

1.3 Literature Survey

In the following list we present the main books and articles that relate to problems that are related to our area of research:

- It is shown in Andrew and Susan [2, Chap. 2] that programmers ran into problems of orally expressing their thoughts when they had to dictate a program. This information is very important to us and consequently we designed our experiments based on it.

- It is shown in Andrew and Susan [2, Chap. 3] how spoken Java is processed. They developed several tools for analyzing the semantics and syntax of spoken Java. We are interested in studying the Harmonia tool for our research as well as integrating it in our system.

- Graham [6] provide all the information about how to use the Harmonia tool and how to integrate it in programming tools. It is useful information for our future work.

- It is shown in Feldman [4, Programming By Voice and Touch] the basic ideas and concepts of our work. Basically, this paper is the starting point for this research.

1.4 Objectives

The main objectives are the following:

1. Design the representation of the code in compact mode.

2. Design a concept for configuring language features. It needs to be comfortable and intuitive for the user.

3. Perform a series of experiments with different volunteers. The purposes of those experiments are to understand how programmers pronounce the code that they want to insert. What are the most negligible actions that programmers take?

4. Based on the experiments, define a set of templates that will be used as a tool to identify programmers' commands and transform them into lines of code.

5. Search and investigate existing programming features. The features that we look for are those which are related to the Java and generally to programming. All those features have the potential to be integrated into the system.

6. Build a prototype that proves that developing code on mobile devices is possible.

7. Provide a solid foundation for future works based on this research.

1.5 Limitations

Our study has several limitations which are summarized in the following list:

- This project requires a lot of coding work. Since I'm the only programmer, the implementation will be very limited.

- The analysis part is very long and difficult. It might require a lot of time of the research.

- Not many studies have been done which relate to the field "Developing on Mobile Devices" or "Developing using Voice". Therefore, we don't have many resources that we can depend upon, or from which we can learn. Most the work will have to do on our own.

- To implement this system a knowledge of NLP is required. We might need to study this field in order to use tools from it.

1.6 Approach

In this section I provide the scientific approach for each objective (the objectives and the approaches are correlated by the numbers of the times in the list):

1. We will study the most common and major domains of programming features that exist in Java. Afterward we will collect enough programming features covering enough domains and will study other languages to determine how those features are represented there. We will search in related works for new ideas for representations and will have discussions about new those ideas. After all those processes are completed we will design a new compact representation.

2. We need to study the most common programming features and collect them into categories. We need to design methods to configure programming features. Thereafter we will have before us the programming features of both groups which will allow us to determine how the of members from the second group are related to elements from the first group.

3. We will build a series of experiments which will represent a different style of programming. For example, we will utilize object oriented, and algorithmic programming styles. There will be two programmers, the typist and the speaker. The speaker will request from the typist to create a program that is already written in Java. The speaker will dictate the program and the typist will type it. After all experiments are accomplished we will compare the source and the reference (dictated program). Furthermore, we will analyze the differences between the desired and the actual results.

4. We will extract all commands that have been dictated during the experiments. We will group them into sets, for each set we will create a template that represents this group.

5. Schedule a series of team meetings. During every meeting the members of the group will suggest programming features that can be integrated into the system. Every feature will be discussed. All features that have been chosen will be saved in an archive.

6. I will build a text file that contains a set of commands. Every command is a simulation of a dictation that the programmer pronounced. I will build a set of rules that know how to handle the commands.

1.7   Contribution

The main contribution is to allow programmers to develop from everywhere, this will cause increase developers productivity. We will design a convenient way of programs dictation and a new design of compact representation. This project will contribute a flexible IDE for Pcs and a combustible development environment. Additional and very important contribution is that project will help people with disabilities.

1.8   Structure of the Report

- Experiments

- Supported Features

- Compact Representation

- Configuration of Representation

- Programming in Natural Language

- Prototype

- Summary

- Publications

Chapter 2

Experiments

2.1 Introduction

This chapter provides information about experiments that have been performed. The main goal of those experiments is to understand how we pronounce the code that we want to insert. The secondary goal is to create a repository of commands that will be grouped into categories. Based on the repository we will create templates that will help to analyze the pronounced commands.

Every experiment consisted of two active participants and two passive participants (passive participants are listeners). One of the active participants was the speaker and the other one was the typist. In every experiment the typist gave to the speaker a programming task where he needed to implement a program.

The speaker had to dictate a program and the typist had to type exactly what the speaker dictated. The speaker had to dictate lines of code in such way that the typist could understand what he means, but without excessive detail. For example, if the speaker had to dictate the code in Figure 2.1. He would dictate it as follows: *"For each element in elements call to to string"*.

The typist needed to follow the dictations of the speaker and to type the code to the text editor (all four participants could see the screen). The typist typed the code in Java. Every one of the participants could participate and provide suggestions for pronouncing the commands. The typist could delete, edit and navigate in the code with no limitations. There were no time constraints and no limitations on the amount of lines. All experiments were recorded.

After all experiments were performed, we analyzed them and extracted only the relevant

lines that represent commands. For each experiment we created a table that contains two columns. The left column contains the commands that were dictated and the right column represents the code that was typed.

Remark:The commands that were inserted into the tables are filtered from irrelevant vowels. For example, the commands *create class ummm look up* (where *ummm* is the vowel) has been converted to *create class look up*

```
foreach(Element element in elements){
   element.toString();
}
```

Figure 2.1: A simple for each loop where every item in elements activates it's toString method

2.2 Experiments

2.2.1 Experiment No.1

- Date: 28/Apr/2014.

- Speaker: Alex Tilkin.

- Typist: Ari Gam.

- Description: Implement a small program that contains an interface called *Lookup*. This interface has one method called *find*. It returns *Object* and receives *String*. a class called *SimpleLookup* that implements *Lookup*. It has two private members: *Names* that is an array of *String*s, and *Values* that is an array of *Object*s. The implemented method *find* iterates over all elements in *Names* and compares every one of them with *Name*. If it finds such element it returns the matched element. In addition a method called *processValues* that receives: *String[] names*, and *Lookup table*. The program was presented to the speaker during all of the experiment.

Table 2.1 represents the order of the commands that were dictated (top to bottom). Figure 2.2 represents the code that was presented to the speaker. Figure 2.3 represents the results of the dictation.

```java
interface Lookup {
   Object find(String name);
}


void processValues(String[] names, Lookup table) {
   for (int i = 0; i != names.length; i++) {
      Object value = table.find(names[i]);
      if (value != null)
         processValue(names[i], value);
   }
}


class SimpleLookup implements Lookup {
   private String[] Names;
   private Object[] Values;

   public Object find(String name) {
      for (int i = 0; i < Names.length; i++) {
         if (Names[i].equals(name))
            return Values[i];
      }

      return null;
   }
}
```

Figure 2.2: The original Java code that was presented to the speaker during experiment No. 1

```
interface Lookup{
   Object find(String name){
   }
}


void processValues(String[] names, Lookup table){
   for(int i = 0; i < names.Length(); i++){
      Object value = table.find(names[i]);
        if(value != null){
           processValues(names[i], value);
        }
      }
   }


   class SimpleLookup implements Lookup{
      private Strings[] names;
      private Object[] values;

      public Object find(String name){
         for (int i = 0; i < name.Length(); i++){
            if (names[i].equals(name)){
               return values[i];
            }
         }

         return null;
      }
   }
```

Figure 2.3: The result of dictating the code in Figure 2.2

The speaker said	The typer typed
Create class LookUp	+Class LookUp
Create a method processValues that returns void and accepts array of strings names and lookupTable	+processValues(names, table)
Create a loop from zero to the length of names	for 0 ≤ i <names.length
Create value type of object accepts table.find, accepts names at i's index	value ← table.find(names[i])
If value different from null then	value ≠ null ?
call to processValue that accepts name at i's index and value	processValue(name[i], value)
We done with processValues	
Create a class SimpleLookUp implements LookUp	+Class SimpleLookUP : LookUp
Delete the last row	
Create array of strings call it names and make it private	-[] names
Create values type of array of object and make it private	-[] values
Create a method that returns an object call it find accepts name type of string and make it public	+find(name)
Create a loop from zero to the length of names	for 0 ≤ i <names.length
If names at i's index period equals accept name then	names[i].equals(name) ?
Return values at i's index	↩ values[i]
Exit the for loop	
Return null	null

Table 2.1: This table presents the major commands that have been dictated during experiment No.1

2.2.2   Experiment No.2

- Date: 28/Apr/2014.

- Speaker: Alex Tilkin.

- Typist: Ari Gam.

- Description: Implement a method called getInterpolatedValue. It receives two integers and returns double. The method needs to return the interpolated value based on certain conditions. The program was presented to the speaker during all the experiment.

Table 2.2 represents the order of the commands that were dictated (top to bottom). Figure 2.5 represents the code that was presented to the speaker. Figure 2.5 represents the results of the dictation.

```java
public final double getInterpolatedValue(double x, double y){
   if(useBicubic){
      return getBicubicInterpolatedPixel(x, y, this);
   }
   if(x < 0.0 || x >= width-1.0 || y < 0.0 || y >= height-1.0){
      if(x < -1.0 || x >= width || y < -1.0 || y >= height){
         return 0.0;
      }
      else{
         return getInterpolatedEdgeValue(x, y);
      }
   }
   int xBase = (int)x;
   int yBase = (int)y;
   double xFraction = x - xBase;
   if(xFraction < 0.0){
      xFraction = 0.0;
   }
   double lowerLeft = getPixelValue(xBase, yBase);
   double lowerRight = getPixelValue(xBase + 1, yBase);
   double upperAverage = upperLeft + xFraction * (upperRight - upperLeft);
}
```

Figure 2.4: The code that was presented to the speaker during experiment No.2

```java
public final double getInterpolatedValue(double x, double y){

   if(useBicubic){

      return getBicubicInterpolatedPixel(x, y, this);

      }

      if(x < 0.0 || x >= width-1.0 || y < 0.0 || y >= height-1.0){

         if(x < -1.0 || x >= width || y < -1.0 || y >= height){

         return 0.0;

      }

      else{

         return getInterpolatedEdgeValue(x, y);

      }

   }

   int xBase = (int)x;

   int yBase = (int)y;

   double xFraction = x - xBase;

   if(xFraction < 0.0){

      xFraction = 0.0;

   }

   double lowerLeft = getPixelValue(xBase, yBase);

   double lowerRight = getPixelValue(xBase + 1, yBase);

   double upperAverage = upperLeft + xFraction * (upperRight - upperLeft);

}
```

Figure 2.5: The result of dictating the code in Figure 2.3

The speaker said	The typer typed
Create method getInterpolatedValue that accepts arguments x and y	+getInterpolatedValue(x, y)
if useBicubic	useByCubiq?
Change y to i, change q to c, Change capital C to small c	useBicubic?
return a call to getInterpulatedPixel that accepts arguments x, y and this	↩ getBicubicInterpulatedPixel(x, y, this)
We are done with the if	
if x is less than zero dot zero or x is greater or equal to width minus one dot zero or y is less than zero dot zero or y is greater or equal to height minus one dot zero then	x < 0.0 || x ≥ width - 1.0 || y < 0.0 || y ≥ height - 1.0 ?
if x is less than minus one dot zero or x is greater or equal to width or y is less than minus one dot zero or y is greater or equal to height then return zero dot zero	x < -1.0 || x ≥ width || y < -1.0 || y ≥ height ? ↩ 0.0
else return a call to getInterpulatedEdgeValue that accepts parameters x and y	: ↩ getInterpulatedEdgeValue(x, y)
We are done with the outer if	
Assign x to xBase	xBase ← x
Assign y to yBase	yBase ← y
Subtract xBase from x and assign it to xFraction	xFraction ↩ x - xBase
If xFraction is less than zero period zero then assign zero period zero to xFraction	xFraction < 0.0 ? xFraction ← 0.0
We are done with the if	

Table 2.2: This table presents the major commands that were dictated during experiment No.2

The speaker said	The typist typed
Assign the returned value from getPixelValue that accepts parameters xBase and yBase to lowerLeft	lowerLeft ← getPixelValue(xBase, yBase)
Assign the returned value from getPixelValue that accepts first parameter xBase plus one and second parameter yBase to lowerRight	lowerRight ← getPixelValue(xBase + 1, yBase)
Assign to upperAverage the calculation of upperLeft plus xFraction times open parenthesis upperRight minus upperLeft close parenthesis	upperAverage = upperLeft + xFraction * (upperRight - upperLeft)
return the calculation of lowerAverage plus yFraction times open parenthesis upperAverage minus lowerAverage	lowerAverage + yFraction * (upperAverage - lowerAverage)

Table 2.3: Processing Table 2.2. This table presents the major commands that were dictated during experiment No.2

2.2.3 Experiment No.3

Part A

- Date: 12/May/2014.

- Speaker: Ari Gam.

- Typist: Alex Tilkin.

- Description: Implement the Bubble Sort algorithem . The speaker was asked to implement the Bubble Sort algorithm without any assistance. The algorithm had to be implemented in Java. No source code presented to the speaker.

Table 2.4 represents the order of the commands that were dictated (top to bottom). Figure 2.6 represents the result of the dictation by the speaker in part A.

```java
class BubbleSort{

  public void do(){

    for(int i = 0; i < data.length - 1; i++){

      for(int j = 0; j < i; j++){

        if(data[i] > data[j]){

          int temp = data[j];

          data[j] = data[i];

          data[i] = temp;

        }

      }

    }

  }


  private int[] data;


  public BubbleSort(int[] init){

    data = new int[init.length];

    for(int i = 0; i < init.length; i++){

      data[i] = init[i];

    }

  }

}
```

Figure 2.6: The result of the diction of the Bubble Sort algorithm

Part B

- Date: 28/Apr/2014.

- Speaker: Ari Gam.

- Typist: Alex Tilkin.

- Description: After the speaker completed the implementation of the Bubble Sort algorithm he was asked to improve its time complexity by adding an additional condition. No source code was presented to the speaker.

Table 2.4 represents the order of the commands that were dictated (top to bottom). Figure 2.7 represents the result of the dictation by the speaker in part B.

```java
class BubbleSort{
  public void do(){

    for(int i = 0; i < data.length - 1; i++){
      boolean done = true;
      for(int j = 0; j < i; j++){
        if(data[i] > data[j]){
          int temp = data[j];
          data[j] = data[i];
          data[i] = temp;
          done = false;
        }
      }
      if(done){
        break;
      }
    }
  }

  private int[] data;

  public BubbleSort(int[] init){
    data = new int[init.length];
    for(int i = 0; i < init.length; i++){
      data[i] = init[i];
    }
  }
}
```

Figure 2.7: The result after the additional condition was added to the Bubble Sort algorithm

The speaker said	The typist typed
Create class bubble sort	+Class BubbleSort
Public void do with no arguments	+Do
Create array of ints call it data and make it private	-[] data
Create constructor that receives an array of ints and name it init	+BubbleSort([] init)
Copy init to data	data = init.clone
Go to Do method	
Create a loop from zero to the length of data minus one	for 0 ≤ i <data.length - 1
Create an inner loop from zero to i	for 0 ≤ j <i
If the i element of data bigger than the j element of data then switch between them	data[i] >data[j] ? temp ← data[i] data[i] ← data[j] data[j] ← temp
Here starts part B	
Go to the beginning of Do	
Create a boolean variable done initialized to false	done ← false
Add to the exit condition of outer loop not done	for 0 ≤ i <data.length - 1 & done
undo	for 0 ≤ i <data.length - 1
Move the statement boolean done initialized to false to the first line of the outer loop	
Change the value from false to true	done ← true
Go to the end of the if	
Initialize done with false	done ← false

Table 2.4: This table presents the major commands that were dictated during the experiment No.3

2.2.4 Experiment No.4

- Date: 19/May/2014.

- Speaker: Alex Tilkin.

- Typist: Yishai Feldman.

- Description: A program that simulates TV controller was presented to the speaker. The program contains the following interfaces: *Command*, and *ElectronicDevice* and the following classes: *TurnOff, TurnOn, VolumeUp, VolumeDown, TV*, and *DeviceButton*. the classes *TurnOff, TurnOn, VolumeUp* and *VolumeDown* implements Command. The class *TV* implements *ElectronicDevice*. The whole application is designed based on the Command design pattern. During the whole experiment the code was presented to the speaker.

Figure 2.8 represents the *Command* interface. Figure 2.9 represents the *VolumeDown* class that implements the *Command* interface. Figure 2.10 represents the *VolumeUp* class that implements the *Command* interface. Figure 2.11 represents the *TurnOn* class that implements the *Command* interface. Figure 2.12 represents the *TurnOff* class that implements the *Command* interface. Figure 2.13 represents the *ElectronicDevice* interface. Figure 2.14 represents the *TV* class that implements the *ElectronicDevice* interface. Figure 2.15 is the represents the *Device-Button* class. This class contains the *main* method.

The following tables represent the order of the major commands that were dictated (top to bottom) during experiment No.4: Table 2.5, Table 2.6, Table 2.7, Table 2.8.

Remark:In this experiment we present each class only once and not source and dictation result. This is because the source and the dictation results are identical.

```
public interface Command {
    void execute();
}
```

Figure 2.8: The *Command* interface

```java
public class VolumeDown implements Command {

    private TV tv;

    public VolumeDown(TV tv) {

        this.tv = tv;

    }

    @Override

    public void execute() {

        tv.volumeDown();

    }

}
```

Figure 2.9: The *VolumeDown* class that implements the *Command* interface

```java
public class VolumeUp implements Command {

    private TV tv;

    public VolumeUp(TV tv) {

        this.tv = tv;

    }

    @Override

    public void execute() {

        tv.volumeUp();

    }

}
```

Figure 2.10: The *VolumeUp* class that implements the *Command* interface

```java
public class TurnOn implements Command {

   private ElectronicDevice electronicDevice;


   public TurnOn(ElectronicDevice electronicDevice) {

      this.electronicDevice = electronicDevice;


   }


   @Override

   public void execute() {

      electronicDevice.on();

   }

}
```

Figure 2.11: The *TurnOn* class that implements the *Command* interface

```java
public class TurnOff implements Command {

   private ElectronicDevice electronicDevice;

   public TurnOff(ElectronicDevice electronicDevice) {

      this.electronicDevice = electronicDevice;

   }


   @Override

   public void execute() {

      electronicDevice.off();

   }

}
```

Figure 2.12: The *TurnOff* class that implements the *Command* interface

```java
public interface ElectronicDevice {

    void on();


    void off();


    void volumeUp();


    void volumeDown();
}
```

Figure 2.13: The *ElectronicDevice* interface

```java
public class TV implements ElectronicDevice {

    private int volume;

    @Override
    public void on() {
        System.out.println("The TV is on");
    }

    @Override
    public void off() {
        System.out.println("The TV is off");
    }

    @Override
    public void volumeUp() {
        volume++;
        System.out.println("The volume is now " + volume);
    }

    @Override
    public void volumeDown() {
        volume--;
        System.out.println("The volume is now " + volume);
    }
}
```

Figure 2.14: The *TV* class that implements the *ElectronicDevice* interface

```java
public class DeviceButton {

    private Command command;

    public DeviceButton(Command command) {
        this.command = command;
    }

    public void press() {
        command.execute();
    }

    public static void main(String[] args) {
        ElectronicDevice tv = new TV();
        Command turnOffCommand = new TurnOff(tv);
        Command turnOnCommand = new TurnOn(tv);
        DeviceButton deviceButtonOn = new DeviceButton(turnOnCommand);
        DeviceButton deviceButtonOff = new DeviceButton(turnOffCommand);
        deviceButtonOff.press();
        deviceButtonOn.press();
    }
}
```

Figure 2.15: The *DeviceButton* class that contains the *main* method

The speaker said	The typer typed
Create interface ElectronicDevice	+interface ElectronicDevice
Create method on	+on
Create method off	+off
Create method volumeUp	+volumeUp
Create method volumeDown	+volumeDown
Create class TvRemoteControl	+Class TvRemoteControl
Without RemoteControl	+Class Tv
That implements ElectronicDevice	+Class : ElectronicDevice
Go to on	
Print the TV is on	Print "The TV is on"
Print the TV is off	
Print the TV is off	Print "The TV is off"
Create local field volume	-volume
Go to volumeUp	
Do volume plus plus	volume++
Print the volume is now and concatenate volume	Print "The volume is now" + volume
Add space after now	"The volume is now " + volume
Go to volumeDown	
Do volume minus minus	volume--
Print the volume is now space concatenate volume	Print "The volume is now " + volume
Create interface Command	+interface Command
Create method execute	+execute
Create class TurnTVOn implements Command	+class TurnTvOn : Command

Table 2.5: This table presents the major commands that have been dictated during experiment No.4

The speaker said	The typer typed
Create constructor that accepts TV	+TurnTvOn(tv)
Assign tv to field tv	this.tv ← tv
Go to execute	
Create class VolumeUp implements Command	+class volumeUp : Command
Create constructor that accepts TV	+volumeUp(tv)
Assign tv to field tv	this.tv ← tv
Go to execute	
TV period volumeUp	tv.volumeUp
Create class VolumeDown implements Command	+Class VolumeDown : Command
Create constructor that accepts TV	+volumeDown(tv)
Assign tv to field tv	this.tv ← tv
Go to execute	
TV period volumeDown	tv.volumeDown
Create class DeviceButton	+Class DeviceButton
Create constructor that accepts command	+deviceButton(command)
Assign command to field command	this.command ← command
Create method press	+press
Command period execute	command.execute
The programmer detected mistakes	
Rename TurnTVOn to TurnOn	+Class TurnOn : Command
Change constructor's parameter type to ElectronicDevice	+turnOn(electronicDevice)
Change the type of the field tv to ElectronicDevice	+ElectronicDevice tv
Rename tv to ElectronicDevice	+ElectronicDevice electronicDevice
Rename TurnTvOff to TurnOff	+TurnOff : Command

Table 2.6: Processing Table 2.5. This table presents the major commands that were dictated during experiment No.4

The speaker said	The typer typed
Change constructor's parameter type to ElectronicDevice	+turnOff(electronicDevice)
Change the type of the field tv to ElectronicDevice	+ElectronicDevice tv
Rename tv to ElectronicDevice	+ElectronicDevice electronicDevice
Go to volumeUp	
Change constructor's parameter type to ElectronicDevice	+volumeUp(electronicDevice)
Change the type of the field tv to ElectronicDevice	+ElectronicDevice tv
Rename tv to ElectronicDevice	+ElectronicDevice electronicDevice
Go to volumeDown	
Change constructor's parameter type to ElectronicDevice	+volumeDown(electronicDevice)
Change the type of the field tv to ElectronicDevice	+ElectronicDevice tv
Rename tv to ElectronicDevice	+ElectronicDevice electronicDevice
Create main	+main([]args)
Create ElectronicDevice type of TV	+main([]args)
Create Command type of TurnOff that receives tv and name it turnOffCommand	turnOffCommand ← TurnOff(tv)
Create Command turnOnCommand type of TurnOn and intialize it with TV	turnOnCommand ← TurnOn(tv)
Create DeviceButton that accepts turnOnCommand and assign it to deviceButtonOn	deviceButtonOn ← DeviceButton(turnOnCommand)

Table 2.7: Processing Table 2.6. This table presents the major commands that were dictated during experiment No.4

The speaker said	The typer typed
Create new DeviceButton, name it deviecButtonOff and initialize it with turnOffCommand	deviceButtoff ← DeviceButton(turnOffCommand)
deviecButtonOff period press	deviceButtoff.press
deviecButtonOn period press	deviceButton.press
Save	
Run	

Table 2.8: Processing Table 2.7. This table presents the major commands that were dictated during experiment No.4

2.2.5 Experiment No.5

- Date: 03/07/2014.

- Speaker: Perry Shalom.

- Typist: Alex Tilkin.

- Description: In this experiment the typist was asked to implement a program that builds a car. The program had to be designed based on the Builder design pattern. The program had one main class *Car* It had to contain three private fields type of String: *wheels, engine,* and *body*. A private constructor that accepts all three parameters that initialize the fields. If one of the parameter is a null or empty string, the constructor should return and not initialize any one of the fields. The *Car* class had to contain a private static class *CarBuilder*, it had to contain three private fields type of String: *wheels, engine,* and *body*. It had to contain a method named *buildCar* that checks if all three fields are initialized and return a new instance of *Car* class. If one of the fields is not initialized then the method will return null. A main method needs to build a car by using the *CarBuilder* class. No source code was presented to the speaker.

Figure 2.16 represents the dictation result of experiment No.5. Figure 2.17 represents the com-

pact representation of Figure 2.16. Table 2.9 and Table 2.10 represents the major commands that were taken during experiments No.5.

```java
public class Car{
    private String _wheels;
    private String _engine;
    private String _body;

    private Car(String wheels, String engine, String body){
        if(body == null || engine == null || wheels == null){
            return;
        }
        _wheels = wheels;
        _engine = engine;
        _body = body;
    }

    public static class CarBuilder{
        String Body;
        String Wheels;
        String Engine;
        public Car BuildCar(){
            if(Body != null && Wheels != null && Engine != null){
                return new Car(Wheels, Engine, Body);
            }
        return null;
        }
    }

    public static void main(String[] args){
        Car.CarBuilder carBuilder = new CarBuilder();
        carBuilder.Engine = "honda";
        carBuilder.Wheels = "4";
        carBuilder.Body = "private";
        Car car;
        car = carBuilder.BuildCar();
    }
}
```

```
+Class Car
    -string _wheels
    -string _engine
    -string _body


    -Car(body, engine, wheels)
      _body ← body
      _engine ← engine
      _wheels ← wheels


    +static Class CarBuilder
        +string _body
        +string _wheels
        +string _engine


        +Car BuildCar()
          body ≠ null ∧ engine ≠ null ∧ wheels ≠ null ?
            ↩ new Car(body, engine, wheels)
          ↩ null


+static Main(args[])
    carBuilder ← new CarBuilder()
    carBuilder.engine ← "honda"
    carBuilder.wheels ← "4"
    carBuilder.body ← "private"
    car ← CarBuilder.BuildCar
```

Figure 2.17: The result of experiment No.5 in compact representation

The speaker said	The typer typed
Create interface ElectronicDevice	+interface ElectronicDevice
Create class car	+Class Car
Create wheels type of string and make it private	-string _wheels
Create engine type of string and make it private	-string _engine
Create body type of string make it private	-string _body
Create static inner class CarBuilder	+Class CarBuilder
Create method CreateCar that returns Car	+Car *CreateCar*
Change CreateCar to BuildCar	+Car BuildCar
Create body type of string make it public	+string body
Create wheels type of string and make it public	+string wheels
Create engine type of string and make it public	+string engine
Go to BuildCar	
If wheels and body and engine are not empty strings then	wheels $\neq$ null $\wedge$ body $\neq$ null $\wedge$ engine $\neq$ null ?
Return to Car	
Create a constructor that receives its three fields and initializes them	+Car(body, engine, wheels) _body $\leftarrow$ body _engine $\leftarrow$ engine _wheels $\leftarrow$ wheels
go to the the beginning of the constructor	
If body equals null or engine equals null or wheels equals null then return	body = null $\vee$ wheels = null $\vee$ engine = null ? $\hookleftarrow$
Make the constructor of Car private	-Car
Go to the If of BuildCar in CarBuilder	
return a new instance of car with the fields body, engine and wheels	$\hookleftarrow$ new Car(body, engine, wheels)
Exit the If,	

Table 2.9: This table presents the major commands that were dictated during experiment No.5

The speaker said	The typist typed
return null	↩ null
Create Main inside Car	*main*
change the method to static	+Car *BuildCar*
Undo	+Car BuildCar
Go to the beginning of Main	
Create an instance of CarBuilder	carBuilder ← new CarBuilder
Initialize engine of carBuilder with honda	carBuilder.engine ← "honda"
Initialize wheels of carBuilder the string four	carBuilder.wheels ← "4"
Initialize body of carBuilder the string private	carBuilder.body ← "private"
Create identifier type of car	car
Call to BuildCar of CarBuilder and put the returned value into car	car ← CarBuilder.BuildCar

Table 2.10: Proceeding Table 2.9. This table presents the major commands that were dictated during experiment No.5

2.3   Commands Repository

The following lists present commands that were recorded during the experiments (see 2.2). It is important to note that the commands were extracted from the recordings manually and filtered to make the process of organization more easy. For example, random vowels, long pauses and "off the record discussions" were removed.

The commands were organized by categories. Each category represents a common action that was repeated during the Experiments. Under the categories you will find sub-categories that make the category more specific in relation to the language feature to which they relate.

The phrases are grouped so they will make a pattern. Below some of the groups you may find a template that represents the group. We could not find a pattern for every group, but for those we could it helped us to build the CFG language for the BNF parser. Based on those patterns we

create a set of BNF rules which is discussed in subsection 6.3.3 and implemented in section 7.3. By using the BNF rules the system will resolve the textual commands that the user dictates and transforms them to data that the system will be able to handle.

2.3.1 Administration commands

- SAVE

- RUN

2.3.2 I/O

- Print The TV is on

- Print The TV is off

PRINT string

- Print The volume is now concatenate volume

PRINT [string CONCATENATE languageFeature]+

2.3.3 Navigation

Method

- Go to on

- Go to off

- Go to volumeUp

- Go to volumeDown

- Go to execute

- Go to main

- Go to the beginning of main

- Go to Do method

- Go to the beginning of Do

- Go to BuildCar

GO TO methodName

Loop

- Go to the for loop

- Go to the while loop

- Go to the end of the for loop

- Go to the end of the while loop

- Go to the beginning of do while loop

GO TO LOCATION LOOP TYPE

Control Block

- Go to the end of the if

- Go to the If of BuildCar in CarBuilder

Constructor

- Go to the the beginning of the constructor

GO TO location CONSTRUCTOR

2.3.4 Exit

If

- We are done with the if

- We are done with the outer if

WE ARE DONE WITH THE controlBlock

- Exit the if

EXIT THE controlBlock

- Go outside the If

GO OUTSIDE THE conditionBlcok

Method

- We are done with processValues

WE ARE DONE WITH methodName

- Exit processValues

EXIT methodName

Loop

- Exit the for

- Exit the while

EXIT THE loop

- We are done with the for

- We are done with the while

WE ARE DONE WITH THE loop

Class

- Exit car

EXIT className

2.3.5 Expression

Mathematical Expression

- LowerAverage plus yFraction times open parenthesis upperAverage minus lowerAverage

- UpperLeft plus xFraction times open parenthesis upperRight minus upperLeft close parenthesis

- Do volume++

- volume--

Assignment Expression

- Copy init to data

- Done initialized to false

- Initialize engine of carBuilder with the string "honda"

- Initialize wheels of carBuilder the string "4"

- Initialize body of carBuilder the string "4"

- Put the returned value into car

- xBase is assigned x

- yBase is assigned y

- xFraction is assigned x minus xBase

- LowerLeft is assigned a call to getPixelValue that accepts parameters xBase and yBase

- LowerRight is assigned a call to getPixelValue that accepts first parameter xBase plus 1 and second parameter yBase

- UpperAverage is assigned upperLeft plus xFraction times open parenthesis upperRight minus upperLeft close parenthesis

- Assign tv to the field tv

- Assign to field command command

- Switch between the "i" and "i+1" elements in data

2.3.6 Collections

Add

- Add new rectangle to shapes

- Add new circle to shapes

- Add new triangle to shapes that accepts 5, 4 and 3

ADD NEW object to collection OPTION

2.3.7 Conditions

- If useBicubic

- If x is less than zero dot zero or x is greater or equal to width minus one dot zero or y is less than zero dot zero or y is greater or equal to height minus one dot zero then

- If x is less than minus one dot zero or x is greater or equal to width or y is less than minus one dot zero or y is greater or equal to height then return zero dot zero

- If xFraction is less than zero dot zero then xFraction is assigned zero dot zero

- If value different from null then

- If names at i's index dot equals accept name then return values at i's index

- If the i element of data bigger than the j element of data then switch between the "i" and "i+1" elements in data

- If wheels and body and engine are not empty strings then

- If body = null or engine = null or wheels = null return

IF condition THEN expressions

2.3.8 Return

- Return a call to getInterpulatedPixel that accepts arguments x, y and this

- Return a call to getInterpulatedEdgeValue that accepts parameters x and y

- Return lowerAverage plus yFraction times open parenthesis upperAverage minus lowerAverage

- Return values at i's index

- Return null

- Return zero dot zero

- return a new instance of car with the fields body, engine and wheels

- ** Return to Car and create a constructor that received its three fields and initializes them

- Return a+b+c

RETURN expression

2.3.9 Call

Method

- Call to getInterpulatedPixel that accepts arguments x, y and this

- Call to getInterpulatedEdgeValue that accepts parameters x and y

- Call processValue that accepts name at i's index and value

- Call deviceButtonOff dot press

- Call deviceButtonOn dot press

- Call to BuildCar of CarBuilder and put the returned value into car

- TV dot on

- TV dot off

- tv dot volumeUp

- tv dot volumeDown

CALL [to]* methodName [that "that accepts"]+ [parameters arguments]+ [argumentName and]+ identifier dot methodName

2.3.10 Delete

- Delete the last row

DELETE [the]* languageFeature

2.3.11 Change/Modify

- Change y to i, change q to c, Change capital C to small c

- Undo

Class

- (Create class command) Create class TvRemoteControl; (Modifying the command) Without RemoteControl

- Rename TurnTVOn to TurnOn

Methods

- Rename TurnTvOff to TurnOff

- Change getPerimeter to return a+b+c

- Change CreateCar to BuildCar

- Change the method to static

Strings

- Add space after now

Constructors

- Change the constructor parameter type to ElectronicDevice

- Change the constructor of Car to private

Identifiers

- Change type of tv to ElectronicDevice

- Rename tv to electronicDevice

- Turn frameWidth and frameHeight to parameters

- Move the statement boolean done initialized to false to the first line of the outer loop

- Change the value from false to true

Loops

- Add to the exit condition of outer loop not done

2.3.12   Inheritance/Implementation

Inheritance

- Inherits from CarBase

- Inherits from Animal

INHERITS [from]* className

Implemintation

- Implements ElectronicDevice

- Implements command

- Implements LookUp

2.3.13   Create

Method

- Create method on

- Create method off

- Create method volumeUp

- Create method volumeDown

- Create method execute

- Create method press

CREATE method methodName

Class

- Create a class LookUp

- Create a class SimpleLookUp implements LookUp

- Create class TvRemoteControl that implements ElectronicDevice

- Create class TurnTVOn implements command

- Create class TurnTvOff implements command

- Create class VolumeUp implements command

- Create class VolumeDown implements command

- Create class DeviceButton

- Create class car

- Create static inner class CarBuilder

- Add a new subclass of ShapeTriangle

- Class bubblesort

Loop

- Create a loop from zero to the length of names

- Create a loop from zero to the length of names in intervals of one

- Create a loop from zero to the length of data minus one

- Create an inner loop from zero to i

Interfaces

- Create interface ElectronicDevice

- Create interface Command

Constructors

- Create constructor that accepts TV

- Create constructor that accepts Command

- Create a constructor that accepts as arguments 3 values of shapes: a, b, c

- Create constructor that receives and array of ints and name it init

- Create a constructor that received its three fields and initializes them

Identifiers

- Create names type of array of strings

- Create array of string call it names and make it private

- Create values type of array of object and make it private

- Create ElectronicDevice of type TV

- Create Command of type TurnOff that accepts tv. Name it turnOffCommand

- Create Command turnOnCommand of type TurnOn. Initialize it with tv

- Create DeviceButton that accepts turnOnCommand. Assign it to deviceButtonOn

- Create new DeviceButton. Name it deviecButtonOff. Initialize it with turnOffCommand

- Create array of ints call it data and make it private

- Create a boolean variable done initialized to false

- Create wheels type of string and make it private

- Create engine type of string and make it private

- Create body type of string make it private

- Create body type of string make it public

- Create wheels type of string and make it public

- Create engine type of string and make it public

- Create value type of object accepts table.find, accepts names at i's index

- Create local field volume

- Create an instance of CarBuilder

- Create identifier from the type of car

Main

- Create main

- Create Main inside Car

Chapter 3

Supported Features

3.1   Introduction

This chapter discusses the research that has been done to collect information about existing technologies for the Java language. The purpose of this research is to collect information about potential technologies that can be integrated into the Deverywhere system and to increase its functionality. All features have been discussed by the research group whether they have a potential to be integrated or not. The features that are presented in this chapter are only those which have been chosen to be integrated. Note that this research is flexible and the list of features might be changed. This part is important for the research because our architecture and prototype are designed in such way so all the mentioned features can be integrated into it.

Every feature that is discussed in this chapter is followed by a numerical value which represents the priority of the feature. The range of the numbers is 1-4 where 1 is the highest priority and 4 is the lowest priority. The definition of priority for our purposes is how important that feature is to this research.

3.2   List of Features

3.2.1   Programming by Voice (writing)

- Priority: 1

Allow the user to program using his or her voice.

- In case the system stumbles on a case of ambiguity it will present options to the user from which he could choose the most appropriate solution.

- The system should distinguish between when the user dictates to it or speaks to someone (this is a very complicated feature to implement so it might be postponed until later works).

3.2.2   Navigation by Voice

- Priority: 2

The user could navigate in the code using his or her voice.

3.2.3   Editing by Voice

- Priority: 3

The user could edit the code by using his or her voice.

3.2.4   Compact View Mode

- Priority: 1

Provide an easy method for understanding, comfortable and compact representation for code. Allow the use of emoticons and other graphical symbols in order to represent language features such as: classes, methods, and variables.

3.2.5   Refactoring

Enable the user to perform refactoring operations on the code with voice commands. Because it is complicated to support all refactoring features, we decided to choose several features that will be supported (prioritized list where the first one has the highest priority).

- Rename Element (Variable, Rename, Field etc.) - Changing the name into a new one that better reveals its purpose

- Constructor Using Fields - Create constructor by selecting a couple of fields and create a constructor that receives those fields as a formal variables that initialize the local fields

- Surround with "try-catch" - Surround a chunk of code with "try-catch" statement

- Move Element (Method or Field) - move to a more appropriate Class or source file

- Push Down - Move fields from derived class to the base class

- Pull Up - Move fields from base class to derived class

- Self-Encapsulate Field - force code to access the field with getter and setter methods (should be hidden) (described in details in the section "Getters and Setters Identification")

- Change Method Signature

3.2.6 Object Identification

- Priority: 2 (relates to 3.2.2)

This is a core feature that has to be implemented in-order to allow other features such as: navigation, refactoring and any other feature that requires from the user to point to where he wants to take the action. Below is a list of several features that provides an example where is it needed.

- Refactoring - When a user asks to refactor a certain object he says something like "change car to truck". In order to change this identifier, first the system needs to identify it and then start the process of renaming it.

- Navigation in code - In order to allow navigation in code the system needs to identify the object and its location in order to navigate correctly. For example, one may say "go to a method drive of car" and the cursor will go to the first line of the method "Drive" in the class "Car".

- Code Selection - user may select pieces of code by telling the system start and end points. For example, "Select the code from line sixteen to line twenty four".

3.2.7   Temporal Abstraction

- Priority: 4

This feature allows the user to speak in high level commands and generate code automatically. Due to the complexity of implementing the whole feature, we chose to concentrate on the section "Sequences as Conventional Interfaces": Allows the user to speak in loops terminology and presents the code in mathematical sequences representation.

- The user will speak in loops terminology and create Java loops.

- Transform Java loops to mathematical sequences.

- Perform well-known algorithms on collections, for example. "Perform Quick Sort on the collection cars by the field year".

3.2.8   Details on Touch

- Priority: 2

Allow the user to inspect an element in (e.g. a variable) and peek into hidden details (e.g. its type) without losing orientation.

3.2.9   Changing the View Mode

- Priority: 4

Allow a user to change the view mode from compact to explicit and vise versa (explicit means to show the types and the accessibility of each object).

3.2.10   Fish Eye

- Priority: 3

Improve user orientation by displaying a large part of the program on the screen; less-relevant lines will be displayed in small font.

3.2.11 Quick Fix

- Priority: 2

This is the same feature as Eclipse has.

- Package Declaration - (need to run in the background). Add missing package declaration or correct package declaration. Move compilation unit to package that corresponds to the package declaration.

- Imports - (need to run in the background). Remove unused, unresolvable or non-visible import. Invoke 'Organize imports' on problems in imports.

- Types - Create new class, interface, enum, annotation or type variable for references to types that cannot be resolved. Change visibility for types that are accessed but not visible. Rename to a similar type for references to types that cannot be resolved. Add import statement for types that cannot be resolved but exist in the project. Add explicit import statement for ambiguous type references (to import-on-demands for the same type). If the type name does not match with the compilation unit name, either rename the type or rename the compilation unit. Remove unused private types. Add missing type annotation attributes.

- Constructors - Create new constructor for references to constructors that cannot be resolved (this, super or new class creation). Reorder, add or remove arguments for constructor references that mismatch parameters. Change method with constructor name to constructor (remove return type). Change visibility for constructors that are accessed but not visible. Remove unused private constructor. Create constructor when super call of the implicit default constructor is undefined, not visible or throws an exception. If type contains unimplemented methods, change type modifier to 'abstract' or add the method to implement.

- Methods - Create new method for references to methods that cannot be resolved. Rename to a similar method for references to methods that cannot be resolved. Reorder or remove arguments for method references that mismatch parameters. Correct access (visibility,

static) of referenced methods. Remove unused private methods. Correct return type for methods that have a missing return type or where the return type does not match the return statement. Add return statement if missing. For non-abstract methods with no body, change to 'abstract' or add body. For an abstract method in a non-abstract type, remove abstract modifier of the method or make type abstract. For an abstract/native method with body, remove the abstract or native modifier or remove body. Change method access to 'static' if method is invoked inside a constructor invocation (super, this). Change method access to default access to avoid emulated method access. Add 'synchronized' modifier. Override hashCode(). Open the 'Generate hashCode() and equals()' wizard.

- Fields and variables - Correct access (visibility, static) of referenced fields. Create new fields, parameters, local variables or constants for references to variables that cannot be resolved. Rename to a variable with similar name for references that cannot be resolved. Remove unused private fields. Correct non-static access of static fields. Add 'final' modifier to local variables accessed in outer types. Change field access to default access to avoid emulated method access. Change local variable type to fix a type mismatch. Initialize a variable that has not been initialized. Create getter and setters for invisible or unused fields. Create loop variable to correct an incomplete enhanced 'for' loop by adding the type of the loop variable.

- Exception Handling - Remove unneeded catch block. Remove unneeded exceptions from a multi-catch clause. Handle uncaught exception by surrounding with try/catch or adding catch block to a surrounding try block. Handle uncaught exceptions by surrounding with try/multi-catch or adding exceptions to existing catch clause (1.7 or higher).Handle uncaught exception by adding a throw declaration to the parent method or by generalizing an existing throw declaration

3.2.12   Dictation User Experience and Error Correction

- Priority: 1

This feature provides rich user experience for dictation. When the user dictates, the system will respond not only with textual output but also with suggestions and recommendations for his or

her work. For example, the user said "create for loop", an ambiguity might happen. The system needs to present to the user relevant options and let him decide what he means. In addition, just as in Eclipse, the system will mark compilation error in real time.

3.2.13   Undo, Redo

- Priority: 2

Every step that has been taken during the development process can be reverted.

3.2.14   Templates and Concise commands

- Priority: 1

Allow the user to generate code without explicitly pronouncing what needs to be written in the code, e.g., one can say "Create main inside Car" and the program will generate main method inside the class Car. Another example can be, while the cursor is inside the Car class, the user can say "Create a constructor" and the program will generate a constructor with no parameters (no parameters because the user didn't say that he wants parameters inside the constructor).

3.2.15   Save the Program as Regular Source Code

- Priority: 1

Open existing source code file.

3.2.16   Support multiple source files for analyzing

- Priority: 3

Refactoring and displaying definitions.

3.2.17   Search

- Priority: 3

Search and display results.

3.2.18   Source control integration

- Priority: 4

Allow source control programs such as GitHub to integrate the system and control your code.

3.2.19   Stand Alone System

- Priority: 4

The system runs on the mobile device, Internet connectivity is needed only if downloading/uploading source code.

3.2.20   Multi Platform

- Priority: 4

The system runs on major mobile operating systems.

3.2.21   License

- Priority: 2

Open source.

3.2.22   Show Time Complexity of Methods

- Priority: 4

Near every method show its time complexity. For example, +print(object) *o(n)*

3.2.23   Command Variability

- Priority: 3

Sometimes we use different words that have the same meaning (i.e. we say "create a function" when we actually mean "create a method"). In order to provide a convenient environment for programming one can hold a thesaurus (e.g. Wordnet) that will include relations between similar words. This thesaurus can be modified (add, remove, edit), users should be able to define their favorite ways of talking. This includes the choice of words to describe templates, features, and locations. (This may circumvent the need for dictionaries and improve the effectiveness of the process) Both of these assume a fixed set of templates. Ideally, we would also have the following, perhaps for "super users": Create new templates. This should be as flexible as possible.

3.2.24   Programming Languages Support

- Priority: 4

One may transform the compact code to any language that the transformer will support (e.g. Java, C#). Relevant for variable name conventions, libraries and for explicit mode.

3.2.25   Recommendations System

- Priority: 4

While we work on our program the apprentice will recommend modifications that will improve the code (i.e. instead of writing nested code the apprentice will recommend to write an "if" statement with negative logic and a "return" or "continue" command).

- The recommendations will be presented as a list, the programmer will choose the option by clicking on it or pronouncing the option number.

3.2.26   Duplication Handling

- Priority: 2

Once we generate an identifier that already exists the application will handle this and rename the identifier so it will be unique in its scope.

3.2.27 String Construction by Voice

- Priority: 2

String manipulation (e.g. concatenate strings), dictation of characters (e.g. white spaces in hard-coded strings). Simplest that can work.

3.2.28 Real Estate

- Priority: 4

We assume that this technology can be implemented not only on mobile devices but also on even more futuristic devices, e.g., Google Glass.

3.2.29 Auto Identifier Names Generation

- Priority: 1

The application will generate field names based on class name. Note that in compact representation the system might not show class name (i.e. Car car will be displayed as car).

3.2.30 Extension Methods

- Priority: 3

A method is added to an object after the original object was compiled. The modified object is often a class, a prototype or a type. Extension methods are permitted by some object-oriented programming languages. There is no syntactic difference between calling an extension method and calling a method declared in the type definition, e.g., one can say "obj.to___", and a list of methods such as: "toFirstUpper", "toString" appears.

3.2.31 Multiple Views

- Priority: 2

This feature allows multiple views in a single language, but without modifying the source code. Davis and Kiczales' registration-based abstractions enables programmers to switch between different views of their program at the press of a button. It is a convenient technique that allows the programmer to view the code in different representations so he will understand the code better and faster. Reference: "How Programming Languages Will Co-evolve with Software Engineering: A Bright Decade Ahead".

3.2.32 Breadcrumbs (Presence in Classes)

- Priority: 3

While we code we create classes and inside them inner classes, methods, properties etc. Sometimes, we might lose our presence due to over encapsulations. In order to solve this issue we suggest a technique of keeping the header definition of the outer object on top of the screen while we scroll down in the inner object, e.g.,

```
Class Human
   Class Brain
```

So, one can always understand the presence in the code in terms of encapsulation.

3.2.33 Getters and Setters Identification

- Priority: 2

Getter/ Setter technique is a very useful technique but one can't avoid writing explicitly "Set...", "Get...". We assume that the system needs to identify the pattern of whether it is a getter or a setter or neither of them, and only print the suffix of the method (ignore the explicit prefix), e.g.,

```
public SetName(string name)
{_name = name;}
```

Transformed to,

```
person.name(name)
```

One can only call the method Name, and based on the template of the call the systems will understand if the user wants to call a setter or a getter. Moreover, instead of writing person.name(name) one can write person.Name ← name

3.2.34   Omit Declaration Lines

- Priority: 1

Line that contains only object declaration is not a necessary bit of information and can be omitted. The proposal is to emit the lines that contain only object declaration without any binding, e.g., the "Object obj" can be omitted.

3.2.35   Operator Overloading

- Priority: 2

Allow using simple operators instead of using Java libraries for special types, e.g., instead of writing BigDecimalExtension.operator_plus(x,y) we will write e1+e2 (e1 and e2 are type of big decimal).

3.2.36   Lambda expression

- Priority: 3

Lambda expressions are a new and important feature included in Java SE 8. They provide a clear and concise way to represent one method interface using an expression. Lambda expressions also improve the Collection libraries making it easier to iterate through, filter, and extract data from a Collection. In addition, new concurrency features improve performance in multi-core environments.

3.2.37   Type Inference

- Priority: 3

You rarely need to write down type signatures anymore, Types Deduction - The application will deduce the type of the identifier based on the right side of the statement.

3.2.38 Source Code on Demand

- Priority: 2

One can see the source code by requesting the system to provide the original full source of the code which the user wants to see.

3.2.39 Collaboration

- Priority: 4

Collaboration with colleagues.

3.2.40 Native representation

- Priority: 3

Allow mathematical expressions in their native representation, as described in Eisenberg [3].

3.2.41 Inter-procedural Flow

- Priority: 3

A technique that helps the user to understand the flow of the program easily. It presents code blocks connected one to another based on the flow of the program. This technique helps the user to understand the flow of a program in a much better way. Inter-procedural Flow is based on the Control Flow Graph (CFG) representation technique, using graph notation, of all paths that might be traversed through a program during its execution. Inter-procedural Flow has several application;we present one that fits our system the most:

- Code Bubbles - This is an application that helps the users to navigate and investigate their code in a novel approach. Link: http://www.andrewbragdon.com/codebubbles_site. asp. Additional link which provides a more detailed description is http://cs.brown. edu/~spr/codebubbles/.

3.2.42 Annotations

- Priority: 2

Allow the user to dictate annotations and display annotations in compact form. Allow using tools that are based on annotations:

- JML

- Doxygen

Chapter 4

Compact Representation

4.1   Introduction

Programming languages weren't designed to be presented on small screens, and therefore when you try to present programs on small screens it is difficult to read them. Programming languages are textual, and so require a keyboard In order to edit programs. Standard on-screen keyboards are inconvenient for texting, let alone for programming. In addition, they require one third of the screen which makes the small screen even smaller. Our approach is to use conventional languages such as Java and C++, but allow each programmer to have a tailored compact view that fits a small screen. We believe that we should support programmers better in doing what they already know how to do instead of requiring them to learn a new language and tools only for the purpose of sometimes developing code on mobile phones. Deverywhere is not a new language; it is a way for each programmer to see the code in the way that makes the most sense to him or her. Our solution for small screens is a compact representation of the code, which means focusing on the important information necessary to understand the code.

This chapter discusses how the code can be presented in compact representation. It covers most of the major domains of programming idioms. Every topic that is discussed in this chapter is accompanied with an explanation on how it will be configurable by the user and examples of this are provided by default.

4.2 Operators

This section discusses how atomic operators will be represented. Every symbol may be modified by the user to any character or icon that s/he wants.

4.2.1 Basic Operators

Table 4.1 provides a set of basic atomic operators that are used in almost every line of code.

Operator	Symbol
Plus	+
Minus	-
Multiplication	*
Division	/
Modulo	%
Increment	++
Decrement	- -
Null	⊥
Shift	≪; ≫; ⋙
Relational	<; >; ≤; ≥
Equality	=; ≠
Bitwise AND	&
Bitwise exclusive OR	∧
Bitwise inclusive OR	|
Logical AND	∧
Logical OR	∨
Ternary	?; :
Assignment	←—; +←—; -←—; *←—; /←—; %←—; &←—; ∧←—; ∨←—; ≪←—; ≫←—; ⋙←—

Table 4.1: This table presents operators and symbols that represent them in compact representation. The symbols are examples for how symbols may be presented. The user may modify to any representation that s/he wants. This idea is discussed in chapter 5. Note: ; is used as a separator between operator symbols.

4.3 Statement Terminators

A statement terminator is used to demarcate the end of an individual statement. For example, a statement terminator in Java is ';' (semicolon) similarly to C, C++ and other programming languages. This character doesn't give any information except where the statement ends. The

programmer will be able to modify the symbol, s/he may decide if it will be displayed or how it will look, e.g., result ⟵ nextNode.data, this is a line of code that is missing a statement terminator symbol. It is also possible to modify the style of the terminators or to use an icon.

4.4 Mathematical Expressions

Mathematical expressions are very common in programming. Sometimes they can be very long and complex. Therefore, we would like to represent those expressions in a more compact and native way. We can reduce the length of expressions by changing mathematical expressions representation to native mathematical representation, e.g., $(a-b)/(c-d) \longleftarrow \frac{a-b}{c-d}$.

4.5 Boolean expressions

This section discusses compact representation of boolean expressions.

4.5.1 Range

In-order to check if an integer is in a range the programmer needsto write $0 <= i \&\& i < 10$. We propose to write this expression in a natural representation, it will save space and will be easier for reading. For example, $0 \le i < 10$.

4.6 Scopes

This section discusses ideas of how the programmer may modify the representation of a scope.

4.6.1 Scope Brackets

The programmer may omit the brackets, then the indentation of the code will denote the scope of the code. Figure 4.1 presents a small example of code without scope brackets.

```
age = 10 ?
adult ⟵ false
isChild ⟵ true
```

Figure 4.1: This is an example of a control block that checks if the age is smaller than 10 then the false is assigned to the identifier *adult* and true is assigned to the identifier *isChild*.

4.6.2 Frame

Figure 4.1 shows that both lines of code are related to the condition because they have the same indentation. In addition it is possible to frame the scope. It may help the programmer to understand scopes better. 4.2 presents how frame can be used to represent scopes.

Figure 4.2: This is the same code that is presented in Figure 4.1 but a frame is used to represent a scope.

4.6.3 Indentation

It is also possible to reduce the number of spaces.

Figure 4.3: This is the same code that presented in Figure 4.2 but with a shorter indentation.

The programmer may modify frame's border color, background color, border width, and border style.

4.6.4 Line Break

Line breaks have an influence on the length and the width of the code. Therefore, we would like to provide several different representations for line breaking.

New Line

Every block of code starts in the same line with the condition key word. Figure 4.4 presents an example of this style.

```
if [condition]
then [code_1]
else [code_2]
[code_3]
```

Figure 4.4: This is a style where every block of code starts in the same line with the condition key word.

2 Columns

In this representation the *then* and the *else* are below the if expression but placed side by side. Figure 4.5 presents an example of this style.

```
if [condition]
then [code_1] else [code_2]
[code_1]       [code_2]
[code_2]
```

Figure 4.5: This is a style where the *then* and the *else* are below the if expression but placed side by side.

4.6.5 Fill

This configuration is almost identical to section 4.6.4 but here if one of the code blocks is longer than the other it will fill the whole line. Figure 4.6 presents an example of this style.

```
if [condition]
then [code_1] else [code_2]
[code_1]       [code_2]
[        code_2             ]
[        code_2             ]
```

Figure 4.6: This is a style where if one of the code blocks is longer than the other it will fill the whole line.

4.7 Accessibilities

The programmer may change the accessibility of classes, and attributes. For example, they can be presented in the default way, e.g., *public, private, protected, internal* or they can be presented with symbols as show in Table 4.2. Also, it is possible to color the symbols: $+, -, \sim, \pm$. Moreover, the programmer may ignore symbols and just color the identifier, e.g., head will stand for private, iterator will stand for public. It is also possible to use icons instead.

Operator	Symbol
Public	+
Private	-
Protected	$\pm$
Internal	~

Table 4.2: This table represents possible representations for accessibilities.

4.8 Implementation and Inheritance

Instead of using long terms such as: *Implements*, and *inherits*, the programmer may change the text to a more compact representation, e.g., ":" from C++. It is also possible to use text or icon.

4.9 Types

4.9.1 Style

The programmer may modify the style of types, e.g., change int to *int*.

4.9.2 Text Value

The programmer may modify the textual value of types, e.g., change int to integer.

4.9.3 Omit Types

In-order to save space the programmer may hide types, access level, modifier and show only identifiers. Since types are hidden the programmer may not understand what the type of the identifier is. In order to avoid ambiguity we suggest a technique by which when an object is created the name of the identifier will be the same as the type (but the first letter will be in lowercase), e.g., private Person _person will be presented person (the Red color denotes that this is a private field and the *italic* denotes that it is static).

4.10 Fields

The programmer may configure so that every field that is created will has a specific prefix or postfix, e.g., if the user configured the prefix to be "_" and asked to create the field *car* the system will write "_car". The programmer may configure that all fields will have a certain style, e.g., _car.

4.11 Methods

This section discusses compact representation of methods.

4.11.1 Omit Returned Type

Returned types is important for compilation but also it provides information to the programmer what type is returned from methods. This information may be omitted because it does not need to be presented. Figure 4.7 shows an example of a method in which returned type is hidden.

```
foo(int a)
return a
```

Figure 4.7: The returned type (in this case it is int) is omitted.

4.11.2 Omit Types of Formal Parameters

In addition the programmer may omit types of formal parameters of methods. **??** shows the same method that is presented in Figure 4.7 but without the type of its formal parameter.

```
foo(a)
return a
```

Figure 4.8: The type of the formal parameter is omitted.

4.11.3 Omit Parentheses

If a method doesn't have formal parameters it is possible to omit its parentheses. **??** shows a method that doesn't have formal parameters hence its parentheses are omitted.

```
foo
    → "Hello, World!"
```

Figure 4.9: This is method whose parentheses are omitted. The icon of the mobile phone denotes that the text "Hello, world!" will be printed to the console.

4.11.4 Constructors

Constructors are used in every class that is created. There is no need to name the constructors in the name of the classes, we can use symbols or words that will represent it. This section presents ideas that might be used as a representation for constructors.

Different Name
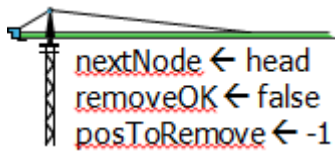
Figure 4.10 presents an example of how a constructor may be represented with a different name other than the name of the class. The programmer may choose any text that make sense to him or her.

```
constructor
name ← "name"
```

Figure 4.10: This is an example of a constructor that uses the word "constructor" as its symbol.

Icon

Instead of using words, the programmer may use an icon that will represent constructors. Figure 4.11 presents an example where an icon of a crane is used to represent a constructor.

Figure 4.11: This is an example where the programmer chooses to use an icon of a crane to represent constructors.

4.12 Control Blocks

Condition statements are very common in programming. It is important to allow programmers to modify their appearance. This section discusses compact representation of control blocks.

4.12.1   If Statements

Figure 4.12 presents an example how "if" statements may be presented. We use the short version of if statement. It is also allowed to use the short version without the else part(":").

```
[Condition] ?
[Code]
:
[Code]
```

Figure 4.12:

4.12.2   Switch

Similar to "if" statements, the programmer may change the representation of the switch statements. In Figure 4.13 shows an example of switch that is presented in compact representation. Note: The word default may be modified as well.

```
name ?
"Danny":
[code]
"Ben":
[code]
"Alex":
[code]
default:
[code]
```

Figure 4.13: This is an example of a switch statement that uses the "?" character instead of "switch". In addition, breaks are omitted.

4.13   Temporal Abstraction

Temporal abstraction is a technique where the programmer instead of writing loops in imperative programming concept, writes the code in functional programming concept. For example, Figure 4.14 shows an example of code that checks if a collection contains a certain element.

```
for(int index = 0; index < collection.length; index++){
if(collection[index] == element)
return true;
}
return false;
```

Figure 4.14: This is an example of a for loop that iterates over a collection and looks for an element by comparing instances.

One can see that the code in Figure 4.14 has an index that runs from zero to the length of the collection and checks each element to determine if it is the element for which it is looking. If the element has been identified then True is returned. Figure 4.16 presents the same code but in temporal abstraction.

```
return collection.exists(element);
```

Figure 4.15: This is an example of the loop in Figure 4.14 in temporal abstraction representation.

The method "exists" in Figure 4.15 accepts an element and returns *True* if it exists in the collection, or *False* if it doesn't. One may notice that the first version is much longer than the second. This is a very important advantage. This example is simple, while Figure 4.16 presents a more complicated example of temporal abstraction.

```
shapes.stream()
.filter(s -> s.getColor() == BLUE)
.forEach(s -> s.setColor(RED));
```

Figure 4.16: This is an example of code that is written in temporal abstraction technique. The elements whose color is blue are selected and their color is changed to red.

The following list contains methods that will be supported by this system. The signature of those methods is such that accepts two arguments, but we should not be confused since those are "extension methods":

- drop(java.lang.Iterable<T> iterable, int count) - Returns a view on this iterable that provides all elements except the first count entries.

- elementsEqual(java.lang.Iterable<?> iterable, java.lang.Iterable<?> other) - Determines whether two iterables contain equal elements in the same order.

- exists(java.lang.Iterable<T> iterable, Functions.Function1<? super T,java.lang.Boolean> predicate) - Returns true if one or more elements in iterable satisfy the predicate.

- filter(java.lang.Iterable<?> unfiltered, java.lang.Class<T> type) - Returns all instances of class type in unfiltered.

- filter(java.lang.Iterable<T> unfiltered, Functions.Function1<? super T,java.lang.Boolean> predicate) - Returns the elements of unfiltered that satisfy a predicate.

- filterNull(java.lang.Iterable<T> unfiltered) - Returns a new iterable filtering any null references.

- findFirst(java.lang.Iterable<T> iterable, Functions.Function1<? super T,java.lang.Boolean> predicate) - Finds the first element in the given iterable that fulfills the predicate.

- findLast(java.lang.Iterable<T> iterable, Functions.Function1<? super T,java.lang.Boolean> predicate) - Finds the last element in the given iterable that fulfills the predicate.

- flatten(java.lang.Iterable<? extends java.lang.Iterable<? extends T» inputs) - Combines multiple iterables into a single iterable.

- fold(java.lang.Iterable<T> iterable, R seed, Functions.Function2<? super R,? super T,? extends R> function) - Applies the combinator function to all elements of the iterable in turn and uses seed as the start value.

- forall(java.lang.Iterable<T> iterable, Functions.Function1<? super T,java.lang.Boolean> predicate) - Returns true if every element in iterable satisfies the predicate.

- forEach(java.lang.Iterable<T> iterable, Procedures.Procedure1<? super T> procedure) - Applies procedure for each element of the given iterable.

- forEach(java.lang.Iterable<T> iterable, Procedures.Procedure2<? super T,? super java.lang.Integer> procedure) - Applies procedure for each element of the given iterable

- head(java.lang.Iterable<T> iterable) - Returns the first element in the given iterable or null if empty.

- isEmpty(java.lang.Iterable<?> iterable) - Determines if the given iterable contains no elements.

- isNullOrEmpty(java.lang.Iterable<?> iterable) - Determines if the given iterable is null or contains no elements.

- join(java.lang.Iterable<?> iterable) - Returns the concatenated string representation of the elements in the given iterable.

- join(java.lang.Iterable<?> iterable, java.lang.CharSequence separator) - Returns the concatenated string representation of the elements in the given iterable.

- join(java.lang.Iterable<T> iterable, java.lang.CharSequence before, java.lang.CharSequence separator, java.lang.CharSequence after,Functions.Function1<? super T,? extends java.lang.CharSequence function) - Returns the concatenated string representation of the elements in the given iterable.

- join(java.lang.Iterable<T> iterable, java.lang.CharSequence separator, Functions.Function1<? super T,? extends java.lang.CharSequence> function) - Returns the concatenated string representation of the elements in the given iterable.

- last(java.lang.Iterable<T> iterable) - Returns the last element in the given iterable or null if empty.

- map(java.lang.Iterable<T> original, Functions.Function1<? super T,? extends R> transformation) - Returns an iterable that performs the given transformation for each element of original when requested.

- operator_plus(java.lang.Iterable<? extends T> a, java.lang.Iterable<? extends T> b) - Concatenates two iterables into a single iterable.

- reduce(java.lang.Iterable<? extends T> iterable, Functions.Function2<? super T,? super T,? extends T> function) - Applies the combinator function to all elements of the iterable in turn.

- size(java.lang.Iterable<?> iterable) - Returns the number of elements in iterable.

- sort(java.lang.Iterable<T> iterable) - Creates a sorted list that contains the items of the given iterable.

- sort(java.lang.Iterable<T> iterable, java.util.Comparator<? super T> comparator) - Creates a sorted list that contains the items of the given iterable.

- sortBy(java.lang.Iterable<T> iterable, Functions.Function1<? super T,C> key) - Creates a sorted list that contains the items of the given iterable.

- tail(java.lang.Iterable<T> iterable) - Returns a view on this iterable that contains all the elements except the first.

- take(java.lang.Iterable<T> iterable, int count) - Returns a view on this iterable that provides at most the first count entries.

- toInvertedMap(java.lang.Iterable<? extends K> keys, Functions.Function1<? super K,V> computeValues) - Returns a map for which the Map.values() are computed by the given function, and each key is an element in the given keys.

- toList(java.lang.Iterable<T> iterable) - Returns a list that contains all the entries of the given iterable in the same order.

- toMap(java.lang.Iterable<? extends V> values, Functions.Function1<? super V,K> computeKeys) - Returns a map for which the Map.values() are the given elements in the given order, and each key is the product of invoking a supplied function computeKeys on its corresponding value.

- toSet(java.lang.Iterable<T> iterable) - Returns a set that contains all the unique entries of the given iterable in the order of their appearance.

4.14   Loops

This section discusses compact representation of loops. We suggest several representations for loops, some of them may look like the for loop.

4.14.1   Temporal Abstraction

Temporal Abstraction is a very useful technique when it comes to loops representation. section 4.13 describes how Temporal Abstraction works and how does it used in our system. Below you can find six modes of representation that supported, every mode provided with description and an example of how the program will look like in this mode. The example that is used is a program that calculates sum of square roots of all positive elements of a collection. Figure 4.17 shows how code looks like with no temporal abstraction.

```
int sum = 0;
for (int i : input) {
if (i > 0)
sum += sqrt(i)
}
```

Figure 4.17: This is an example of how the program looks like if Temporal Abstraction is disabled.

Figure 4.18 is an example from Java 8 java.util.stream where a sequence of methods called one-by-one and the returned value of the first method activates the next method. We call this type of representation **Program**.

```
input.stream()
.filter(i → i > 0)
.mapToInt(i → sqrt(i))
.sum()
```

Figure 4.18: This is an example of how Temporal Abstraction may be used. In this code all the elements that their value is greater than 0 are selected. Each element is inserted into a map where it's value is the key and the value is it's square. Finally all value are summed. The result of this code is sum of squares of all elements that their values are greater then 0.

Another representation is a lexical representation. The code is represented by a phrase that explains what happens in the line of code. We call this representation **Paragraph of Text**.

```
Sum the square roots of the positive integers of input
```

Figure 4.19: This is an example of how line of code may be represented by a phrase. For example, this phrase explains what happens in Figure 4.18.

Another idea of representing Temporal Abstraction code is by a technique we call **Bullets**. Figure 4.20 shows an example of the code in Figure 4.18 where every action that is taken on the

stream presented in a different bullet. For example, the filtering condition named *Filter* and the criteria are positive, which means take only the positive elements.

```
- Enumerate: input
- Filter:    positive?
- Map:       sqrt
- Accumulate: +, 0
```

Figure 4.20: This is an example of the **Bullets** technique. The original code is presented in Figure 4.18. **Enumerate** is the stream that we want to process. **Filter** is the passing criteria. **Map** is the data structure in which each element will be inserted and **Accumulate** is the action that will be taken on all the elements in the data structure.

Signal Flow technique is similar to signal-processing concepts. The different parts of the plan are represented as stages, and the computation is represented as a signal that flows through the stages. Figure 4.21 shows an example of the code in Figure 4.18 in **Signal Flow** representation.

Enumerate → Filter → Map → Accumulate

input   positive? sqrt   +, 0

Figure 4.21: This is an example of the **Signal Flow** technique. The original code is presented in Figure 4.18. The first stage is **Enumerate** that enumerates the input; the second stage is **Filter** that filters the input based on the criteria the third stage is **Map** that maps the filtered elements; and the last stage is **Accumulate** that accumulates the elements.

The last representation technique is **Spreadsheet**, this is a simple way to represent what the code does by a small example. The idea is yielded from the concept that some people prefer to see an example in order to understand what the code does instead of reading it. Figure 4.22 shows an example of how the code in Figure 4.18 may be presented with the **Spreadsheet** technique.

Positive input:	1	7	0	-29	42
sqrt (int):	1	3			6
0-based Sum:	10				

Figure 4.22: This is an example of the **Spreadsheet** represents the code in Figure 4.18. The first line of is a set of values that may be in a data structure. The second line is the result of the sqrt function. Note that there are no values below the values 0 and -29, this is because they don't fullfill the criteria (only positive elements). The third row is the result of summing the elements that are left. In that case they are: 1, 3, and 6. When summing them we will get 10.

4.14.2   Imperative Representation

This section discusses imperative representations for loops. Loops are very common, hence we would like to provide several types of compact representation for them. We call this type of representation Imperative. In computer science terminologyies, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state. We decided to call this compact representation imperative because it reminds the imperative programming languages such as: C, C++, and Java.

Loop Icon

The first representation that we suggest is called **Loop Icon**. The idea is to replace the key words that represent loops with ↻. By replacing the keywords we save space on one hand but still it is clear to the reader that this is a loop. Figure 4.23 shows an example of a loop without the *for* keyword, Instead ↻ is placed.

```
↻(int i = 0; i < n; i++)
system.out.println(i)
```

Figure 4.23: This is an example where the *for* keyword was replaced with ↻.

Loop Watermark

This technique is the most compact representation that we came up with that related to loops representation. We use the background to place the loop symbol, Figure 4.24 shows an example of a loop that uses the **Loop Watermark** representation.


```
toString
        result ← ""
        temp ← head
        i ∈ [1, pos)
                result += temp.data + " "
                temp ← temp.next

    ⌐ result
```
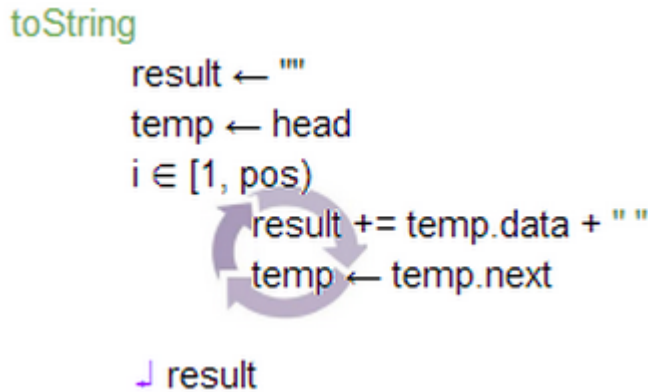
Figure 4.24: This is a public method named *toString*, in middle of the method there is a loop that runs from 1 to *pos*, for each iteration it concatenates the value of *temp.data* with space and concatenates it with result. The technique that is used in the line $i \in [1, pos)]$ is described in section 4.14.2.

Loop Range Representation

The idea for Loop Range technique was taken from mathematics. For example, $i \in [1, pos)$ ↻ represents that i is between 1 and *pos* (not included). It is also very convenient to use this form to iterate over objects in data structures, e.g., a loop over a list of cars may look like this, $item \in cars$. The programmer may iterate only on part of the list. For example, in order to iterate over the first 10 items we will write $item \in cars[0, 10]$ ↻.

Another way to present loop range is to use the representation that is discussed in subsection 4.5.1. For example, for $0 \leq i < 10$. This is a loop the iterates from i to 10.

Loop variables

This technique suggests to present the loop variable and on the background a symbol that shows that this is a loop. For example, Figure 4.23 will be presented like this Figure 4.25.

Figure 4.25: This is an example how a loop like is shown in Figure 4.23 can be presented in Loop Variable mode.

Loop Complexity

Sometimes when a programmer is programming and uses others code he might be interested in the complexity of the code that he uses. Therefore we suggest a representation technique where the programmer may see the complexity instead of the code itself. the code that is shown in Figure 4.23 may be presented as shown in Figure 4.26.

Figure 4.26: This is an example of **Loop Complexity** representation. The complexity of the code in Figure 4.23 is *O(n)*, therefore it is represented with *O(n)* and a circulation loops that denotes that this is a loop.

Loop Signature

This technique allows the programmer to see only the signature of the loop without the code inside it. For example, the loop

```
for (int i = begin; i < n; i++)
system.out.println(i)
```

Will be presented as follows:

```
for (int i = begin; i < n; i++)
```

4.15 System Methods

Our motivation is to provide a compact representation to all system methods. System method is a method that has been provided by the developers of the language, e.g., System.arraycopy(). Currently it is quite difficult to pinpoint what are going to be the functions that the system will support but the motivation is to denote that this is part of our scope of the work. Gradually we'll add more and more functionality to this section.

Print to Screen (System.out)

Instead of writing long lines, e.g., System.out.println("Hello, World!"), one may show only println "Hello, World!" or use an icon like this: .

Memory Copy

Java provides methods that allow copying memory from one data structure to another, e.g., System.arraycopy();. This method accepts the source, the target, the initial index of the source, the initial index of the target, and how many items to copy. We suggest the following representation a[i : ...]←b[i : j], which provides more compact view of the code, more readable and understandable code (note: the ... in the *a* vector means that the program will use as much as memory as it needs). In this example the system copies all elements from index i to index j from vector b to vector a starting from index i.

Strings Utilities

The class StringUtils is a very useful class when it comes to performing operations on strings, e.g., check if string is empty or null; remove substrings, replace substrings, and split a string. The problem with commands is they are very long, e.g., StringUtils.isEmpty(name) (name is a string). Instead we would like to give the option to write this line as follows: name.isEmpty.

4.16   Layouts

Deverywhere is a system that aims not only for mobile devices but also for personal computers. Compact representation can also be useful to make larger screens more clear and efficient. This section discusses ideas of how code may be presented on large screens.

4.16.1   Tiles

If the user would like to review several methods at the same time he may choose this representation to see them in a grid. In Figure 4.27 9 methods are displayed simultaneously on the tablet/computer screen. Each method's body is displayed in compact form. Figure 4.28 presents a more clear view of how the methods may be organized.

Figure 4.27: This is an example of Tiles.

method 1	method 2	method 3
[compact body]	[compact body]	[compact body]
method 4	method 5	method 6
[compact body]	[compact body]	[compact body]
method 7	method 8	method 9
[compact body]	[compact body]	[compact body]

Figure 4.28: This is a more clear view of the methods in **??**. The methods are organized in a transparent table.

4.16.2   Breadcrumbs

While we write code we nest methods inside classes; classes inside classes; control blocks inside methods; control blocks inside control blocks etc. Sometimes, we might lose our orientation un the code due to over nesting. In-order to solve this issue we suggest a technique where the nesting hierarchy will be presented to the user at the top of the screen, e.g., we have a class *Human*, inside this class we have a class *Brain* and inside this class we have a method *think*. Figure 4.29 presents how code may be presented to the user if currently he is located in the method *think* (note: there might be code between the three first rows).

```
+class Human
+class Brain
+think
[code]
```

Figure 4.29: This is an example of a method (*think*) inside the class *Brain* inside the class *Human*. In order not to lose the orientation inside the code, the titles of encapsulating classes are kept and the code between them is folded. Therefore, while the programmer writes code inside the method *think* he knows that he is inside the classes *Brain* and *Human*.

4.16.3 Inter-procedural Flow

A technique that helps the programmer to understand the flow of the program easily. It presents code blocks connected one to another based on the flow of the program. This technique helps the programmer to understand the flow a program in a much better way. Inter-procedural Flow is based on Control Flow Graph (CFG) representation technique, using graph notation, of all paths that might be traversed through a program during its execution.

Code Bubbles

Developers spend significant time reading and navigating code fragments spread across multiple locations. The file-based nature of contemporary IDEs makes it prohibitively difficult to create and maintain a simultaneous view of such fragments. We propose a novel user interface metaphor for code understanding and maintenance based on collections of lightweight, editable fragments called bubbles, which form concurrently visible working sets.

The essential goal of this feature is to make it easier for developers to see many fragments of code (or other information) at once without having to navigate back and forth. Each of these fragments is shown in a bubble.

The following is an example of how will look like compact representation with Code Bubbles. Figure 4.30 is a class which named *Person* who has two private fields: name - the name of the person; partOfTheDay - which indicates which part of the day it is (morning, noon, afternoon, midnight, night etc.). It has two methods: greetings - receives a name of a person and greets him; getPartOfTheDay - returns the part of the day.

```
+class Person
      -name
      -partOfTheDay

      +greetings(name)
            lPartOfTheDay <- getPartOfTheDay
          → lPartOfTheDay + " " + name + " have a nice day"

      +getPartOfTheDay
            partOfTheDay
```

Figure 4.30: This is an example of Code Bubbles.

4.17   Example

This is an example of a compact representation of a program that implements a linked list. The original program was written in Java and is much longer than the one that presented below. We took part of the original program and converted its representation from Java representation to a compact representation. Note that we could choose different representation. The example that provided below is a combination of representation preferences.

The name of the class is *LinkedList,* you can see it on the light blue stripe on the right side. In addition you can see that this is a public class because it has a plus near its name. It has an inner class named *Node* an it is private. You can see it at the top of the code, we know that it is private because it has a minus near its name. The *Node* class has two public attributes: data and next. *LLIterator* is a private inner class that has thee private members: *nextNode, removeOK,* and *posToRemove.* below you may see an icon of a crane that represents a constructor. Below you may find a public method named *remove,* this method accepts pos. The method has a condition (if pos equals to 0).If It is equal it will perform the three lines on the green background. If it is not equal then the code on the red background will work. Note the three circular arrows. The lines of code that are over those circular arrows are a loop. This program is an example how code may be presented in compact representation and still be easy to understand.

This is a concept that we are aiming to reach. Note that this is just one example of compact representation. One may configure the representation of the code so it will fit his or her

preferences.

+LinkedList

-Node
+data
+next

-head
-tail
-size

+iterator   new LLIterator

-LLIterator
-nextNode
-removeOK
-posToRemove

nextNode ← head
removeOK ← false
posToRemove ← -1

+hasNext
nextNode ≠ null

+next
assert hasNext
result ← nextNode.data
nextNode ← nextNode.next
removeOK ← true
posToRemove++
result

+remove
assert removeOK
removeOK ← false
LinkedList.this.remove(posToRemove)
posToRemove--

+makeEmpty
head ← tail ← null
size ← 0

+remove(pos)
assert 0 ≤ pos < size

Pos=0
result ← head.data
head ← head.next
size = 1 ? tail ← NULL

Pos ≠ 0
temp ← head
1 ≤ i < pos
    temp ← temp.next
result ← temp.next.data
temp.next ← temp.next.next
pos = size - 1 ?   tail ← temp

size--
result

+get(pos)
assert 0 ≤ pos < size

Pos = size - 1 ?
result ← tail.data

else
temp ← head
1 ≤ i < pos
    temp ← temp.next
result ← temp.data

result

+insert(pos,obj)
assert 0 ≤ pos < size

Pos = 0 ?
addFirst(obj)

else
Pos = size ?
add(obj)

else
temp ← head
1 ≤ i < pos
    temp ← temp.next
newNode ← new Node(obj, temp.next)
temp.Next ← newNode
size++

+add(obj)
newNode = new Node(obj, NULL)

size = 0 ?   head ← newNode

else   Tail.Next ← newNode

result

+addFirst(obj)
assert 0 ≤ pos < size

size = 0 ?   add(obj)

else
newNode ← new Node(obj, head)
head ← newNode
size++

+ toString
Join data with space from head by next

Chapter 5

Configuration of Representation

5.1 Introduction

One of the goals of this project is to allow the programmer to configure the representation of the code on different devices. PCs are a much more convenient environment for programming than laptops and specifically more convenient than tablets and mobile devices. We would like to allow this ability because every programmer prefers to see the code in different representations. As discussed in chapter 4 we suggest different styles of compact representation to programming idioms. We assume that it would be very convenient for the programmer if s/he could configure different representations for different environments.

This chapter discusses how the representation of programming idioms may be configured. This is a basic concept that might guide those who will proceed with this research and will decide to focus on the configuration of the representation of the code. We believe that different language features have different configuration features in common. For example, Access Levels may be presented as an icon, text, stylized text, e.g., *private*, and may be omitted. Operators may be presented as an icon, text, or stylized text, e.g., **+**. It is clear that both operators and accessibilities may be presented by text, stylized text, and as an icon. This conclusion may help those who will design the configuration of the compact representation of operators and the accessibilities (for example: private, public).

The motivation is that the developers who will develop a configuration system for a programming feature will be aware of all the other programming features that have configuration

features in common. It will optimize the configuration system and will create a more intuitive experience for the user.

It is important to note that this is a generic idea that might be expanded by others. Everyone may add language features, a configuration feature or add configuration features to language features.

5.2   Terminology

This section provides explanations about terms that appear in this section.

- Configuration Feature - A feature that helps the user to set a configuration for a programming feature.

- Language Feature - A feature in a programming language,. for example, loops.

5.3   Configuration scope

5.3.1   Configuration Main Settings

We assume that there are three main modes of configuration of representation: Desktop (PCs and laptops) - This configuration provides the most detailed environment that the user needs, since it has a big screen; Tablet - This environment is smaller than the previous one, but the screen is big enough to show details. Therefore, it will show details but many fewer; and Mobile Phones - This is the smallest environment for developing, therefore, it will provide the most compact representation.

All three configurations that are described above come with default settings that meet the requirements of the environment. The user may customize the representations so they will fit his or her needs.

5.3.2   Features and Configurations

In order to understand how each programming feature may be represented, we reviewed chapter 3 and chapter 4. Those chapters helped us to divide programming features into groups,

e.g., operators. After we managed to group the programming features, we created groups of configuration features. Each configuration feature represents a technique of how a programming feature may be represented. Table 5.1 presents all configuration features that we grouped and Table 5.2 presents all language features and the configuration features that may be used to present them.

In order to understand how every programming feature may be represented you need to use both Table 5.1 and Table 5.2. For example, let us take the first row in Table 5.2. The language feature is Operators, it represents all the operators in Java. The referenced configuration features IDs are: STYLE, ICON and TEXT, which means it may be represented in three ways.

ID	Configuration Feature
TEMPORAL	Temporal Abstraction - Off/ Program/ Paragraph of Text/ Bullets /Signal flow/ Spreadsheet
PLAN	Cliche - Replace code with high-level description
STYLE	Style - foreground color; background color; size; font; enable/disable bold; enable/disable italics, and enable/disable underlined
ICON	Icon - an icon that represents programming feature
TEXT	Text - a text that represents programming feature
OMIT	Omit - Remove syntax features in-order to preserve space
ORDER	Order - Modify the order of elements, e.g., put the "if" before or after the condition
INDENTATION	Indentation Depth - Selecting how many spaces, tabs, pixels or percentage (%) of the line length the indentation will take
FRAME	Frame - Use frame as the boundaries of the scope. Also modify frame style. Also use symbols like "{" that encapsulates the code
NATIVE	Native Form - Convert mathematical expressions into native representation style
LAYOUT	New Line Break - Select where text is located; non-textual spatial arrangement of blocks; breadcrumbs
RANGE	Range Representation - Select how range will be presented, e.g., $i \in [1, pos)$ will be used for loops
WATERMARK	Watermark - Set background image
COMPLEXITY	Complexity - Show complexity If it can be computed using Static Analysis
IO	I/O - Replace code with just input and output
CONTRIBUTORS	Contributors - Replace code with contributor's name

Table 5.1: This table has two columns: ID and Configuration Feature. Configuration Feature (right column) is a technique how a programming feature may be configured. Every configuration feature has a short explanation. The left column is the given ID for every configuration feature. For example, the ID TEMPORAL represents the Temporal Abstraction configuration feature.

Language Feature	Configuration Features IDs
Operators	STYLE, ICON, TEXT
Conditions	STYLE, TEXT, ICON, RANGE
Accessibilities	STYLE, ICON, TEXT, OMIT
Implementation and Inheritance	STYLE, ICON, TEXT
Methods, Constructors	STYLE, ICON, TEXT, FRAME, CONTRIBUTORS, PLAN, OMIT, LAYOUT
Classes	LAYOUT, STYLE, TEXT, OMIT
OMIT	Omit - Remove syntax features in-order to preserve space
Fields	TEXT, STYLE
Control Blocks	STYLE, ICON, TEXT, ORDER, INDENTATION , FRAME, LAYOUT
Statement Terminators	STYLE, ICON, TEXT, OMIT
Types	STYLE, TEXT, OMIT
Delimiters, Operators, Separators	STYLE, TEXT, ICON, OMIT
Keywords	STYLE, ICON, TEXT, OMIT
Scope	FRAME, INDENTATION, TEXT, STYLE, CONTRIBUTORS, PLAN, LAYOUT
Expressions	RANGE, CONTRIBUTORS, NATIVE
Loops	STYLE, ICON, TEXT, OMIT, TEMPORAL, COMPLEXITY, IO, CONTRIBUTORS, PLAN, LAYOUT
Comments	OMIT

Table 5.2: This table has two columns: Language Feature and Configuration Features IDs. Language feature column presents all language features that we managed to group until today. Configuration Features IDs are a set of configuration features that may represent a language feature. For example, Operators may be represented with STYLE, ICON, and TEXT.

Chapter 6

Programming in Natural Language

6.1 Introduction

Programming languages are textual, and so require a keyboard in order to edit programs. Standard on-screen keyboards are inconvenient for texting, let alone for programming. In addition, they take almost one-third of the screen which makes the small screen even smaller. In order to avoid using the inconvenient on-screen keyboard or an external device, we suggest to program by voice. Voice dictation is based on a common set of templates; these templates are individually customizable so that each developer can use the idioms that are most convenient for him or her.

We describe the idea of programming in natural language by writing requirements in order to bridge the gap between the code that the programmer wants to write and a code that is written on the screen. We claim that natural language can serve as the main tool for programming. We do not claim that it is the only tool, the programmer may use other gestures such as touch, on-screen keyboard, or external devices when needed

6.2 Configuration

We suggest that every programmer will dictate in the most convenient way for him or her. Every one of us has a different way of describing the things that s/he wants to say. it is the same with code dictation.

For example, take the Java loop in Figure 6.1. We would dictate it character by character: "for, open parenthesis, int, space" etc. This is inconvenient and cumbersome. Instead, we would like to forget about the syntax and just describe what is needed. One way to say it is: "for i from zero to n". But another programmer may prefer to say "repeat n times" to describe the same loop.

We don't just dictate code from top to bottom; we also need to edit existing code and navigate to specific places in the code. Navigation depends on the context of what is shown on the screen and on the surrounding code. For example, if the screen contains only one loop, I can say "Go to the loop". Or I could say "Rename the index of the first loop to j" to navigate to the first loop on the screen and change its index variable. This demonstrates that templates used for dictation have named parts, and I can refer to these parts when I issue editing or navigation commands.

```java
for(int i = 0; i < n; i++)
```

Figure 6.1: A simple for loop

6.3 Natural Language Processing

In order to process dictation, we suggest a two-phase process: converting speech to text, and understanding the context of the text. We use a speech to text engine for the speech conversion and context free grammar to understand the context.

6.3.1 Speech To Text Engines

We found several speech-to-text engines that we can use for our solution: Nuance Dragon Nuance [9] is a software developer kit (SDK) used by developers and integrators to add speech recognition capabilities into in-house and commercial applications or workflow applications. This toolkit, which enables everything from free-text dictation to command and control functionality, can be deployed as part of a server or client-based solution; Kaldi Povey et al. [11] is a toolkit for speech recognition written in C++ and licensed under the Apache License v2.0. Kaldi is intended for use by speech recognition researchers. Google Speech Server V2 Mey [7] is an online free speech-to-text engine which runs on a server, used primarily by researchers.

We decided to use Google Speech Server V2 because it is free and easy to use. There are many more speech-to-text engines, but we decided to concentrate on the most well-known so Google Speech is the one that we choose to use.

6.3.2 Context Free Grammar

A context-free grammar (CFG) is a tuple $G = (T, N, S, R)$, where T is the finite set of terminals of the language, N is the set of non-terminals that represent phrases in a sentence, $S \in N$ is the start variable used to represent a full sentence in the language. R is the set of production rules of the form $N \rightarrow (N \bigcup T)*$.

Extended Backus Naur Form

Extended Backus Naur Form is a notation for formally describing syntax for how to write entities in language. The meta-language is based on a suggestion by Niklaus Wirth (Wirth, 1977) that is based on the Backus-Naur Form and that contains the most common extensions, i.e:

- Terminal symbols of the language are quoted so that any character, including one used in Extended BNF, can be defined as a terminal symbol of the language being defined.

- [and] indicate optional symbols.

- { and } indicate repetition.

- Each rule has an explicit final character so that there is never any ambiguity about where a rule ends.

- Brackets group items together. It is an obvious convenience to use (and) in their ordinary mathematical sense.

6.3.3 Dictation Parser

We use EBNF to create a set of rules that will validate commands that the programmer dictates. This set of rules can be extended easily by every user to understand EBNF. We use IEEE

IEC EBNF [1] as our main reference to learn how to use EBNF. In [8, pages 51-86] Michal Gordon and David Harel claim that formal structured natural language requirements can serve as the means and the end to programming the behavior of reactive system using fully executable languages such as live sequence charts (LSC). We adopt this approach and use it to create a set of rules to process user dictation.

Grammar Construction

We now show how our grammar translates controlled natural language. Since we allow a CFG, we can increment the grammar with additional rules that allow various ways of generating similar constructs. We can thus increase the set of accepted specifications by augmenting the grammar. However, ambiguity may grow as the grammar grows which would require the user to explicitly disambiguate his or her intentions in too many cases for the process to be friendly. We shall describe how basic commands such as: creation, navigation, and modification are parsed. The *Dictation Parser* tries to build a parse tree for every dictation that it receives. A valid dictation is one for which the *Diction Parser* succeeded to create a parse tree. A number of advanced ideas are not yet supported in the current implementation. Nevertheless, the current grammar allows implementing fully executable systems, and has been tested, among other examples, on the *LinkList* Example, and on the *Command Pattern* Example, see 6.3.6.

The set of rules is divided into four main layers in the Parser layer: *Top, Categories, Forms of Expressions*, and *Common*. The *Top* layer contains rules that have access to all categories. The *Categories* layer contains sub-sections that is divided into different categories. For example, the *Creation* sub-section contains all rules that deal with creation. The *Common* layer knows nothing about any category, it contains all the necessary information for all actions. The *Forms of Expressions* layer contains all all possible forms of expressing any command. For example, there are several options to pronounce that you want to return something from a method, you may say: "that returns", "returns", "return" and more. The *Top* layer and the *Categories* layer have access to the *Common* layer and to *Forms of Expressions* layers. The *Common* layer has an access to the *Forms of expressions* layer. Figure 6.4 shows how rules are divided into layers and sub-categories. Note that this division has no effect on the action of rules. Its function is only to understand the set of rules better.

We now shall explain every sub-category inside the *Categories* layer. *Creation* contains all rules that parse any creation dictation. For example, "Create a new class". *Navigation* contains all rules that parse any navigation dictation. For example, "Go to class person". *Modification* contains all rules that parse any modification dictation.

We now discuss two dictations that have the same meaning but are expressed differently. It is important to Deverywhere to allow programmers to dictate code in their preferred way. Let's assume that the programmer created a class *Car* and now he wants to create an inner class *Wheel*. One way to say it is *"create a static public class named wheel inside class car"*. Figure 6.2 presents the parse tree that is generated for this dictation. This dictation will lead to creating a new static class called *Wheel* inside a class that is called *Car*.
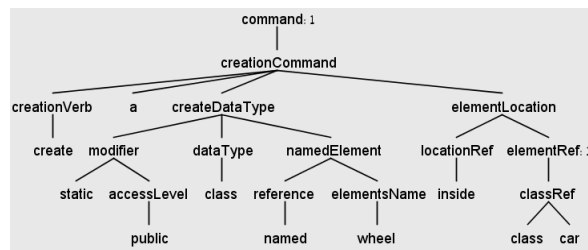
Figure 6.2: The parse tree for the sentence "create a static public class named wheel inside class car".

Another way to say it is *"create an inner class wheel"*. In this case, the class that will be created won't be static and the cursor of the IDE shall be placed inside class *Car*, otherwise the class *Wheel* won't be created inside class *Car*. Figure 6.3 presents the parse tree that is generated for this dictation. This dictation will lead to creating a new class called *Wheel* inside a class that is called *Car*.
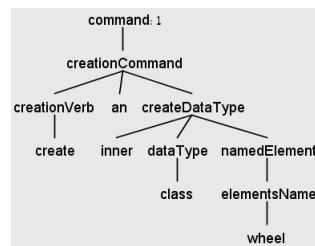
Figure 6.3: The parse tree for the sentence "create an inner class wheel".
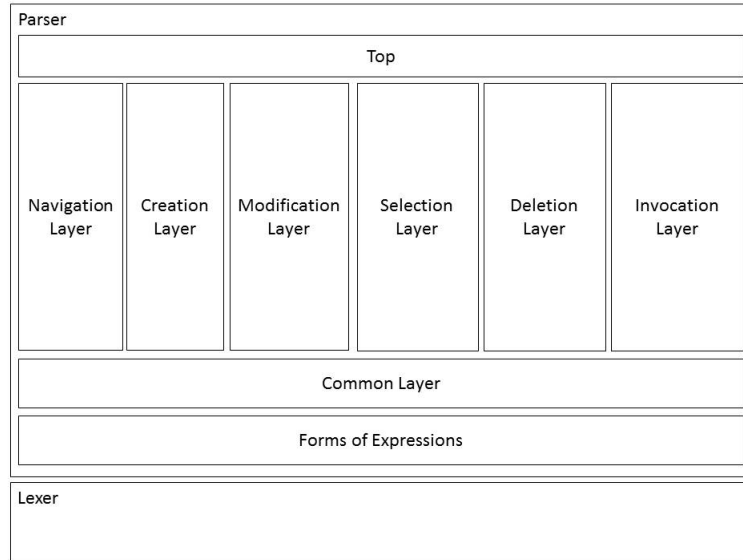
Figure 6.4: This is the architecture of the Dictation Parser rules

6.3.4 Parser

This section describes the parser of the Dictation Parser module. All layers are divided into tables; every table groups the commands of the layer.

Top Layer

Table 6.1 represents the commands of the Top layer.

Rule	Description
command: creationCommand | navigationCommand | selectionCommand | modificationCommand | deletionCommand | invocationCommand;	This is the main command

Table 6.1: This table represents the commands of the Top layer.

Creation Layer

Table 6.2 and Table 6.3 represents the commands of the Creation Layer. This layer handles all creation commands that the user dictates.

Rule	Description
creationCommand: creationVerb? (AN | A)? (createField | createMethod | createConstructor | createDataType | createBlock | createLoop) elementLocation?;	The main command of the Creation Layer
creationVerb: CREATE | NEW | OPEN;	Creation verb
createField: fieldModifier? fieldRef;	Creates a field
createMethod: modifier? (METHOD | FUNCTION) namedElement ((THAT_ACCEPTS | WITH) parametersList)? (returnsVars Element)?;	Creates a method
createConstructor: modifier? CONSTRUCTOR ((THAT_ACCEPTS | WITH) parametersList)?;	Creates a constructor
createDataType: modifier? (INNER)? dataType namedElement ((implementsVars | extendsVars) Element)?;	Creates a data type
createBlock: BLOCK | createBlockStatement;	Creates a block

Table 6.2: This table represents the commands of the Creation layer.

Rule	Description
createBlockStatement: localVariableDeclaration	statement;	Creates a block of stattements
createLoop: createForEachLoop	createWhileLoop	createDoWhileLoop	createForLoop;	Creates a loop
createForEachLoop: FOR_EACH Element IN Element command?;	Creates a foreach loop
createWhileLoop: WHILE expression DO? command?;	Creates a while loop
createDoWhileLoop: DO command WHILE expression;	Creates a do while loop
createForLoop: FOR Element (FROM (number | elementsElement))? (TO (number | elementsElement))? command?;	Creates a for loop

Table 6.3: This table represents the commands of the Creation layer.

Navigation Layer

Table 6.4 represents the commands of the Navigation Layer. This layer handles all navigation commands that the user dictates.

Rule	Description
navigationCommand: navigationVerb (dataType | FIELD | METHOD)? Element | exitCommand;	Creates navigation command
navigationVerb: GO_TO | WE_ARE_DONE_WITH;	Navigation verb
exitCommand: (EXIT | QUIT) (elementsElement | (dataType | FIELD | METHOD)? Element);	Creates exit command
exit: WE_ARE_DONE_WITH | EXIT;	Variations of exit

Table 6.4: This table represents the commands of the Navigation layer.

Modification Layer

Table 6.5 represents the commands of the Modification Layer. This layer handles all modification commands that the user dictates.

Rule	Description
modificationCommand: modifyAccessLevel;	Creates modification command
modifyAccessLevel: modificationVerb accessLevel;	Modifies the access level. For example, changes from private to public
modificationVerb: MAKE_IT | CHANGE_IT;	Modification verb

Table 6.5: This table represents the commands of the Modification layer.

Selection Layer

Table 6.6 represents the commands of the Selection Layer. This layer handles all selection commands that the user dictates.

Rule	Description
selectionCommand: (NUMBER | OPTION)? number;	Creates selection command

Table 6.6: This table represents the commands of the Selection layer.

Deletion Layer

Table 6.7 represents the commands of the Deletion Layer. This layer handles all deletion commands that the user dictates.

Rule	Description
deletionCommand: (DELETE | REMOVE) (line | elementsElement);	Creates deletion command

Table 6.7: This table represents the commands of the Deletion layer.

Invocation Layer

Table 6.8 represents the commands of the Invocation Layer. This layer handles all invocation commands that the user dictates.

Rule	Description
invocationCommand: CALL? elementsElement;	Creates invoke command

Table 6.8: This table represents the commands of the Invocation layer.

Common Layer

Table 6.9 and Table 6.10 represent the commands of the Invocation Layer. This layer handles all invocation commands that the user dictates.

Rule	Description
fieldModifier: (FINAL | CONST)? modifier TRANSIENT? VOLATILE?;	Adds different modifications to a field. For example, final static public.
variableModifier: (FINAL | CONST) | STATIC;	Adds final or static to a field.
modifier: ABSTRACT? STATIC? accessLevel;	Adds abstract, static and access level.
accessLevel: PRIVATE | PUBLIC | PROTECTED;	Adds access level.
localVariableDeclaration: variableModifier* elementsName OF_TYPE Element;	Declares a local variable.
statement: expression | RETURN expression? | TRY CATCH? FINALLY? | THROW expression | SWITCH expression? | BREAK | CONTINUE | caseVars elementsElement;	Defines a statement.
expression: primary | expression (plusVars plusVars | minusVars minusVars) | expression (equalsVars | isDifferentVars | lessThanEqualsVars | greaterThanEqualVars | greaterThanVars | lessThanVars | IS_NOT | IS) expression | (plusVars | minusVars | plusVars plusVars | minusVars minusVars) expression | IF expression ((AND | OR) expression)* THEN command (ELSE command)? | expression (equalsVars | isDifferentVars | lessThanEqualsVars | greaterThanEqualVars | greaterThanVars | lessThanVars | IS_NOT | IS) expression | NEW (expression | elementRef) | ASSIGN expression TO expression;	Defines an expression.

Table 6.9: This table represents the commands of the Common layer.

Rule	Description
primary: OPEN_PARENTHESES expression? | elementsElement | number;	Defines the most primitive expression.
elementsElement: (elementRef OF)? elementRef;	Reference to an element of an element. For example, method of a class.
elementLocation: locationRef (elementRef | line);	Reference to element location.
fieldRef:,FIELD (elementsName? OF_TYPE Element | OF_TYPE Element namedElement | elementsName);	Reference to a field.
elementRef: classRef | fieldRef | enumRef | interfaceRef | unspecifiedRef;	Reference to an element.
classRef: CLASS Element;	Reference to a class.
namedElement: reference? elementsName;	Reference to an element.
elementsName: (Element AND)* Element;	Defines a set of elements.
enumRef: ENUM Element;	Reference to an enum.
interfaceRef: INTERFACE Element;	Reference to an interface.
unspecifiedRef: Element;	Reference to an unspecified element.
reference: NAMED | CALLED;	Different variations of references.
locationRef: INSIDE | IN | AFTER | BEFORE | ABOVE | BELOW;	Defines variations of locations.
parametersList: (parameter AND)* parameter;	Defines a list of parameters.
parameter: elementsElement (OF_TYPE Element)?;	Defines a parameter,
dataType: CLASS | ENUM | INTERFACE;	Defines data types.
line: LINE NUMBER? number;	Defines a line. For example, line number 6.
number: Number | ZERO | ONE | TWO | THREE | FOUR | FIVE | SIX | SEVEN | EIGHT | NINE;	Defins numbers.

Table 6.10: This table is the continuation of Table 6.9.

Forms of Expressions

Table 6.11 and Table 6.12 represents the commands of the forms of expressions.

Rule	Description
returnsVars: THAT_RETURNS | RETURNS | RETURN;	Defines different variations to pronounce "returns".
implementsVars: IMPLEMENTS | IMPLEMENT | THAT_IMPLEMENTS;	Defines different variations to pronounce "implements".
extendsVars: EXTENDS | EXTEND | THAT_EXTENDS;	Defines different variations to pronounce "extends".
plusVars: PLUS | MATH_PLUS;	Defines different variations to pronounce "plus".
minusVars: MINUS | MATH_MINUS;	Defines different variations to pronounce "minus".
equalsVars: IS_EQUAL_TO | EQUAL_TO | EQUALS_TO | EQUALS | IS_EQUALS;	Defines different variations to pronounce "equals".
isDifferentVars: IS_DIFFERENT_FROM | DIFFERENT_FROM;	Defines different variations to pronounce "is different".
lessThanVars: LESS_THAN | LESS_THAN_MATH | IS_LESS_THAN;	Defines different variations to pronounce "less than".
lessThanEqualsVars: LESS_THAN_EQUAL | LESS_THAN_EQUAL_MATH | LESS_THAN_EQUAL_MATH_SPACE;	Defines different variations to pronounce "less than equals".

Table 6.11: This table represents the commands of the forms of expressions of different commands.

Rule	Description
greaterThanVars: GREATER_THAN | GREATER_THAN_MATH | IS_GREATER_THAN;	Defines different variations to pronounce "greater than".
greaterThanEqualVars: GREATER_THAN_EQUAL | GREATER_THAN_EQUAL_MATH | GREATER_THAN_EQUAL_MATH_SPACE;	Defines different variations to pronounce "greater than equal".
forEachVars: FOR_EACH | FOR_EACH_SPACE;	Defines different variations to pronounce "for each".
caseVars : CASE | IN_CASE;	Defines different variations to pronounce "case".

Table 6.12: This table is the continuation of Table 6.11.

6.3.5 Lexer

This section describes the lexer of the Dictation Parser module. The lexer is built so it only has simple regular expressions and definitions.

- METHOD: 'method';

- FUNCTION: 'function';

- CONSTRUCTOR: 'constructor';

- FIELD: 'field';

- BLOCK: 'block';

- INSIDE: 'inside';

- IN: 'in';

- AFTER: 'after';

- BEFORE: 'before';

- ABOVE: 'above';

- BELOW: 'below';

- INNER: 'inner';

- OF_TYPE: 'of type';

- NEW: 'new';

- NAMED: 'named';

- CALLED: 'called';

- LINE: 'line';

- NUMBER: 'number';

- OPTION: 'option';

- AN: 'an';

- A: 'a';

- THAT_ACCEPTS: 'that accepts';

- WITH: 'with';

- AND: 'and';

- OR: 'or';

- TO: 'to';

- FROM: 'from';

- GO_TO: 'go to';

- EXIT: 'exit';

- QUIT: 'quit';

- WE_ARE_DONE_WITH: 'we are done with';

- MAKE_IT: 'make it';

- CHANGE_IT: 'change it to';

- CREATE: 'create';

- OPEN: 'open';

- OPEN_PARENTHESES: 'open parentheses';

- CALL: 'call';

- OF: 'of';

- PERIOD: 'period';

- PERIOD_CHAR: '.';

- DELETE: 'delete';

- REMOVE: 'remove';

- ASSIGN: 'assign';

- PLUS: 'plus';

- MATH_PLUS: '+';

- MINUS: 'minus';

- MATH_MINUS: '-';

- IS: 'is';

- IS_NOT: 'is not';

- IS_EQUAL_TO: 'is equal to';

- EQUAL_TO: 'equal to';

- EQUALS_TO: 'equals to';

- EQUALS: 'equals';

- IS_EQUALS: 'is equals';

- IS_DIFFERENT_FROM: 'is different from';

- DIFFERENT_FROM: 'different from';

- LESS_THAN: 'less than';

- LESS_THAN_MATH: '<';

- IS_LESS_THAN: 'is less than';

- IS_LESS_THAN_EQUAL: 'is less than equal';

- IS_LESS_THAN_EQUALS: 'is less than equals';

- LESS_THAN_EQUAL: 'less than equal';

- LESS_THAN_EQUALS: 'less than equals';

- LESS_THAN_EQUAL_MATH: '<=';

- LESS_THAN_EQUAL_MATH_SPACE: '< =';

- GREATER_THAN: 'greater than';

- GREATER_THAN_MATH: '>';

- IS_GREATER_THAN: 'is greater than';

- GREATER_THAN_EQUAL: 'greater than equal';

- GREATER_THAN_EQUAL_MATH: '>=';

- GREATER_THAN_EQUAL_MATH_SPACE: '> =';

- IF: 'if';

- THEN: 'then';

- ABSTRACT: 'abstract';

- ASSERT: 'assert';

- CATCH: 'catch';

- CLASS: 'class';

- CONST: 'const';

- DO: 'do';

- ELSE: 'else';

- ENUM: 'enum';

- EXTENDS: 'extends';

- EXTEND: 'extend';

- THAT_EXTENDS: 'that extends';

- FINAL: 'final';

- FOR: 'for';

- IMPLEMENTS: 'implements';

- THAT_IMPLEMENTS: 'that implements';

- IMPLEMENT: 'implement';

- INTERFACE: 'interface';

- PRIVATE: 'private';

- PROTECTED: 'protected';

- PUBLIC: 'public';

- STATIC: 'static';

- SUPER: 'super';

- THROW: 'throw';

- THROWS: 'throws';

- TRANSIENT: 'transient';

- TRY: 'try';

- VOID: 'void';

- VOLATILE: 'volatile';

- WHILE: 'while';

- FOR_EACH: 'foreach';

- FOR_EACH_SPACE: 'for each';

- THAT_RETURNS: 'that returns';

- RETURNS: 'returns';

- RETURN: 'return';

- SWITCH: 'switch';

- SYNCHRONIZED: 'synchronized';

- STRICTFP: 'strictfp';

- NATIVE: 'native';

- PACKAGE: 'package';

- IMPORT: 'import';

- INSTANCEOF: 'instanceof';

- FINALLY: 'finally';

- CONTINUE: 'continue';

- DEFAULT: 'default';

- BREAK: 'break';

- CASE: 'case';

- IN_CASE: 'in case';

- ZERO: 'zero';

- ONE: 'one';

- TWO: 'two';

- THREE: 'three';

- FOUR: 'four';

- FIVE: 'five';

- SIX: 'six';

- SEVEN: 'seven';

- EIGHT: 'eight';

- NINE: 'nine';

- WS: [\t \r \n \u000C]+ -> skip;

- Number: [0-9]+;

- Element: [a-zA-Z0-9]+;

```
for(int i = 0; i < n; i++)
```

Figure 6.5: A simple for loop

6.3.6   Grammar Testing

To test the grammar, we provide two programs and their transcripts. The first example is an implementation of a linked list and the second is an example of the Command design pattern.

LinkedList Example

This is an implementation of the data structure linked list. Figure 6.6-Figure 6.12 presents the program that implements Linked List. Figure 6.3.6 lists the dictations that generate this program.

```java
public class LinkedList implements Iterable {

   private Node head;

   private Node tail;

   private int size;


   public Iterator iterator(){

      return new LLIterator();

   }


   private class Node{

      public Object data;

      public Object next;


      public Node(Object data, Object next){

         this.next = next;

         this.data = data;

      }


      public Object getNext(){

         return next;

      }


      public Object getData(){

         return data;

      }


      public void setNext(Object next){

         this.next = next;

      }

   }
```

Figure 6.6: Implementation of Linked List in Java part 1

```java
private class LLIterator implements Iterator{

   private Node nextNode;

   private boolean removeOK;

   private int posToRemove;


   private LLIterator(){

      nextNode = head;

      removeOK = false;

      posToRemove = -1;

   }


   public boolean hasNext(){

      return nextNode != null;

   }


   public Object next(){

      assert hasNext();


      Object result = nextNode.getData();

      nextNode = nextNode.getNext();


      removeOK = true;

      posToRemove++;


      return result;

   }
```

Figure 6.7: Implementation of Linked List in Java part 2

```java
    public void remove(){

        assert removeOK;

        removeOK = false;

        LinkedList.this.remove(posToRemove);

        posToRemove--;

    }

}


public void makeEmpty(){

    head = tail = null;

    size = 0;

}


public Object remove(int pos){

    assert pos >= 0 && pos < size;

    Object result;

    if( pos == 0 ){

        result = head.getData();


        head = head.getNext();

        if( size == 1 )

            tail = null;

    }

    else{

        Node temp = head;

        for(int i = 1; i < pos; i++)

            temp = temp.getNext();


        result = temp.getNext().getData();

        temp.setNext( temp.getNext().getNext() );

        if( pos == size - 1)

            tail = temp;

    }

    size--;
```

```java
    public void remove(){

        assert removeOK;

        removeOK = false;

        LinkedList.this.remove(posToRemove);

        posToRemove--;

    }

}


public void makeEmpty(){

   head = tail = null;

   size = 0;

}
```

Figure 6.9: Implementation of Linked List in Java part 4

```java
public Object remove(int pos){

   assert pos >= 0 && pos < size;

   Object result;

   if( pos == 0 ){

      result = head.getData();

      head = head.getNext();

      if( size == 1 )

         tail = null;

   }

   else{

      Node temp = head;

      for(int i = 1; i < pos; i++)

         temp = temp.getNext();


      result = temp.getNext().getData();

      temp.setNext( temp.getNext().getNext() );

      if( pos == size - 1)

         tail = temp;

   }

   size--;

   return result;

}
```

Figure 6.10: Implementation of Linked List in Java part 5

```java
public Object get(int pos){

   assert pos >= 0 && pos < size;

   Object result;

   if( pos == size - 1 )

      result = tail.getData();

   else{

      Node temp = head;

      for(int i = 0; i < pos; i++)

         temp = temp.getNext();


      result = temp.getData();

   }


   return result;

}


public void insert(int pos, Object obj){

   assert pos >= 0 && pos <= size;

   if(pos == 0)

      addFirst(obj);


   else if( pos == size )

      add(obj);

   else{

      Node temp = head;

      for(int i = 1; i < pos; i++)

         temp = temp.getNext();


      Node newNode = new Node(obj, temp.getNext());

      temp.setNext( newNode );

      size++;

   }

}
```

Figure 6.11: Implementation of Linked List in Java part 6

```java
    public void add(Object obj){
       Node newNode = new Node(obj, null);
       if( size == 0 )
          head = newNode;
       else
          tail.setNext(newNode);


       tail = newNode;
       size++;
    }


    public void addFirst(Object obj){
       if(size == 0)
          add(obj);
       else{
          Node newNode = new Node(obj, head);
          head = newNode;
          size++;
       }
    }


    public String toString(){
       String result = "";
       Node temp = head;
       for(int i = 0; i < size; i++){
          result += temp.getData() + " ";
          temp = temp.getNext();
       }


       return result;
    }

 }
```

Figure 6.12: Implementation of Linked List in Java part 7

This is the list of the dictations that generates the program.

- Create class LinkedList that implements Iterable

- Create private field head of type Node

- Create private field tail of type Node

- Create private field size of type int

- Create method iterator that returns Iterator

- Return new LLIterator

- Exit method iterator

- Create inner class Node

- Create field data of type Object

- Make it public

- Create public field next of type Object

- Create constructor that accepts data of type Object and next of type Node

- Assign next to this.next

- Assign data to this.data

- Create method getNext that returns Object

- Return next

- Create method getData that returns Object

- Return data

- Create method setNext that accepts next of type object

- Assign next to this.next

- Exit class Node

- Create inner class LLIterator that implement Iterator

- Create a constructor

- Assign head to field nextNode

- Assign false to field removeOK

- Assign -1 to field posToRemove

- Ceate method hasNext that returns boolean

- Return is nextNode different from null

- Ceate method next that return Object

- Assert hasNext

- Create result of type object and assign nextNode.getData to it

- Call nextNode.getNext and assign it to nextNode

- Assign true to removeOK

- Increase posToRemove

- Return result

- Create method remove

- Assert removeOK

- Assign false to removeOK

- Call LinkedList.this.remove that accepts posToRemove

- Decrease posToRemove

- Exit LLIterator

- Crete method makeEmpty

- Assign null to tail and head

- Assign zero to size

- Create method remove that accept pos of type int and returns Object

- Assert is pos more than 0 and pos less than size

- Create result of type Object

- If pos equal to 0 then

- Call head.getData and assign it to result

- Call head.getNext and assign it to head

- If size equal 1 then

- Assign null to tail

- Exit the if

- Else

- Create temp of type Node and assign head to it

- For i from 1 to pos

- Call temp.getNext and assign it to temp

- Exit the loop

- Call temp.getNext.getData and assign it to result

- Call temp.setNext that accepts temp.getNext.getNext

- If pos equal size minus 1 then

- Assign temp to tail

- Exit else

- Decrease size

- return result

- Create method get that accepts pos of type int and returns Object

- Assert is pos more or equal to 0 and less than size

- Create result of type Object

- If pos equal size minus 1 then

- Call tail.getData and assign it to result

- Else

- Create temp of type Node and assign head to it

- For i from 0 to pos

- Call temp.getNext and assign it to temp

- Exit for

- Call temp.getData and assign it to result

- Exit else

- Return result

- Create method insert that accepts pos of type int and obj of type Object

- Assert is pos more or equal to 0 and less than size

- If pos equal to 0 then

- Call addFirst that accepts obj

- Else if pos equal size then

- Call add that accepts obj

- Else

- Assign head to temp of type Node

- For i from 1 to pos

- Call temp.getNext and assign it to temp

- Exit the for loop

- Create new Node that accepts obj and temp.getNext and assign it to newNode of type Node

- Call temp.setNext that accept newNode

- Increase size

- Exit method insert

- Create method add that accepts obj of type Object

- Create new Node that accepts obj and null and assign it to newNode of type Node

- If size equal 0 then

- Assign newNode to head

- Else

- Call tail.setNext that accepts newNode

- Exit

- Assign newNode to tail

- Increase size

- Create method addFirst that accepts obj of type Object

- If size equal 0 then

- Call add that accepts obj

- Else

- Create new Node that accepts obj and head and assign it to newNode of type Node

- Assign newNode to head

- Increase size

- Create method toString that returns String

- Create result of type String and initialize it with an empty string

- Assign head to temp of type Node

- For i from 0 to size

- Result plus equal temp.getData plus space

- Call temp.getNext and assign it to temp

- Exit the loop

- Return result

Car Builder Example

This is an implementation of the Builder design pattern. We created a class of type Car that has properties and contains an inner class of type CarBuilder that has the functionality to create an instance of Car. In addition, it has main. Figure 6.13-Figure 6.14 presents the program that implements Linked List. Figure 6.3.6 lists the dictations that generate this program.

```java
public class Car{
   private String _wheels;
   private String _engine;
   private String _body;

   private Car(String wheels, String engine, String body){
      if(body == null || engine == null || wheels == null)
         return;

      _wheels = wheels;
      _engine = engine;
      _body = body;
   }

   public static class CarBuilder{
      String Body;
      String Wheels;
      String Engine;

      public Car BuildCar(){
         if(Body != null && Wheels != null && Engine != null){
            return new Car(Wheels, Engine, Body);
         }

         return null;
      }
   }
}
```

Figure 6.13: Implementation of the design pattern Builder that creates a class of type Car part 1

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

    Car.CarBuilder carBuilder = new CarBuilder();

    carBuilder.Engine = "honda";

    carBuilder.Wheels = "4";

    carBuilder.Body = "private";

    Car car;

    car = carBuilder.BuildCar();

    }

}
```

Figure 6.14: Implementation of the design pattern Builder that creates a class of type Car part 2

This is the list of the dictations that generate the program.

- Create class car

- Create field wheel of type string

- Create field engine of type string

- Create field body of type string

- Create private constructor that accept wheels of type string engine of type string and body of type string

- If body equals null or engine equals null or wheels equals null then return

- Assign wheels to field wheels

- Assign engine to field engine

- Assign body to field body

- Create inner class CarBuilder

- Create public field body of type string

- Create public field wheels of type string

- Create public field engine of type string

- Create method buildCar that returns Car

- If body different from null or wheels different from null or engine different from null then return new car that accepts wheels engine and body

- Return null

- Create main inside car

- Create new CarBuilder and assign it to carBuilder of type carBuilder

- Assign honda as string to carBuilder.engine

- Assign 4 as string to carBuilder.engine

- Assign private as string to carBuilder.engine

- Create car of type car

- Call carBuilder.BuildCar and assign it to car

6.4   Implementation

To implement the *Dictation Parser*, we reviewed different applications such as: BNFC, BNF Parser Generator, BNF for Java, Antlr 4. We decided to use Antlr 4 because it can be integrated well in our solution, it is a popular and well-known tool and it is well known to other members of the team. The implementation of the *Dictation Parser* is explained in details in 7.3.

Chapter 7

Prototype

7.1 Introduction

This chapter discusses the prototype that provides a proof of concept for this research. It includes every module of the prototype discussed and described in this chapter. All architecture designs and specifications are described in detail. Note that not all modules and ideas that are discussed here are implemented, this detail will be mentioned for every module.

This prototype presents an ability of programming using speech and represents code in a compact way. It is written in the Java language and uses different third party services and applications. Every service and application that is used is discussed and described.

7.2 Speech to Text

This is a module which is responsible for the translation from speech to text. The engine that is used for the STT process is the Google Speech V2 server. How to interact with this server is shown in Mey [7] .

7.2.1 Google Speech V2 Server

Google Speech V2 server provides STT service with no charge for everyone who willing to use their service.

Remark:In order to use this service you have to be a chromium user (https://www.chromium.org/) because Google's server requires developer's key.

Host

In order to access this service the host that must be used is https://www.google.com/speech-api/v2/recognize.

Request

The request that must be sent to Google's server is:

```
curl -X POST \
--data-binary @'test.flac' \
--header 'Content-Type: audio/x-flac; rate=44100;' \
'https://www.google.com/speech-api/v2/recognize?output=json&lang=en-us&key=
AIzaSyDT41KV3j_c2OseWNNt4xv79MD9sj9p2j4'
```

Explanation about the curl:

data-binary is a path on your local machine to audio file that you want to translate.

header is the information about encoding of the audio. In this example is flac with a bit rate of 44,100.

uri is the host of the server with concatenation of the output format, language, and developer key. In this example we use json (output=json), English (lang=en-us), and our developer key is AIzaSyDT41KV3j_c2OseWNNt4xv79MD9sj9p2j4.

Response

The response format is JSON. When Google is 100% confident in its translation, it will return the following object:

```
{
    "result":
```

```
[
{
    "alternative":
    [
    {
       "transcript":"good morning Google how are you feeling today"
    }
    ],
    "final":true
}
],
"result_index":0
}
```

When it's doubtful, it adds a confidence parameter for you. It also seems to add multiple transcripts for some reason.

```
{
"result":
[
{
  "alternative":
  [
    {
       "transcript":"this is a test",
       "confidence":0.97321892
     },
     {
       "transcript":"this is a test for"
     }
    ],
    "final":true
}
```

```
        ],
        "result_index":0
    }
```

7.2.2   Speech to Text Library

This library is used to bring speech recognition to Processing applications (processing is a programming language, development environment, and online community. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology). Using WebSocket, Google Chrome and Processing, you can get unlimited speech recognition results in your Sketch. As of May 2014, you need a developer API key to use this library. The new API has a limit of 50 requests per day.

This library is an open source with dependency in Processing, hence we had to make modifications so it will fit our needs. Several modifications have been made to this open source:

- The dependency in Processing has been removed so the application could run as a simple Java project.

- A new response parser has been developed because the one that was in use didn't know how to parse the new format of Google Speech V2.

- Auto Speech Recording had memory issues.

- Various redesign modifications.

It is shown in Schulz [12] where to get the source code from and how to use its API.

7.2.3   Testing the Module

In order to be sure that we can use this module in our system we performed several tests with several testers. We used the Linked List program that is presented in section 4.17 to perform the main test. Table 7.1 and Table 7.2 are the result of the dictation of the Linked List program. Every command was dictated separately, each row contains three columns: Expected Result, is the context that was dictated, Actual Result, is the context that was returned, and Match Score,

represents how both contexts are matched. Since programming is not a natural language then we expect mismatches. For example, the user may dictate "create field has next of type boolean". He means that he wants to create a field named hasNext which is of type boolean. The module will return "create field has next of type boolean". We do not expect the speech to text engine to return hasNext, therefore, this conclusion should be considered in the Dictation Parser. A similar issue is shortened words. For example, if the user dictates "create field pos" then the system may return "create field pose". pos is different from pose, since pos is not a legal word in the language. We expect this mismatch and assume that this will be handled by the user.

We calculated the average matching and the result was 97.5%. This result is within the accectable limitations for our system. The response time is dependent on the length of the context and how busy the server is. The response time is [0.5, 5] seconds. The response time is within our acceptable range. If we want to upgrade the performance of our system we may consider to upgrading this module. For example, we may exchange it with IBM Watson.

Expected Result	Actual Result	Match Score (%)
create class linked list	create class linked list	100
create inner class node	create inner class node	100
create field data of type object	create field data of type object	100
make it public	make it public	100
create public field next of type object	create public field next of type object	100
we are done with class node	we are done with class node	100
create field tail of type node	create field tail of type node	100
create an int field size	create an int field size	100
create method iterator	create method iterator	100
return new llinterrater	return new ll interrater	100
number 2	number 2	100
exit method	exit method	100
create inner class LLinterrater	create inner class LLL interrater	90
create a constructor	create a constructor	100
assign head to field next node	assign head to field next node	100
assign false to field removeok	assign false to field remove ok	100
assign minus 1 to field postoremove	assign minus 12 field post to remove	60
number 2	number 2	100
finish constructor	finish constructor	100
create method hasnext	create method hasnext	100
return nextnode is not equal to null	return next node is not equal to null	100
we are done with class LLinterrater	we are done with class LLL interrater	90

Table 7.1: This table represents the context that was dictated and the context that was returned. For each pair there is a matching score [0, 100]%.

Expected Result	Actual Result	Match Score (%)
create method get that accepts pos of type int	create method get that accepts pose of type int	100
assert pos is between 0 and size	assert pose is between 0 and size	100
if pos equals size -1	ethos equals size -1	80
then assign tail. Data to result	then assign tail. Data to result	100
otherwise	otherwise	100
temp is asigned head	temp is a signed head	100
create a loop from one to pos	create a loop from one to pose	100
assign them. next to temp	sign them. Next to temp	100
set the result to temp. data	set result to temp. Data	100
return result	return result	100

Table 7.2: This table is the continuation of Table 7.1.

7.3 BNF Parser

Antlr 4

Antlr stands for Another Tool for Language Recognition. The tool is able to generate compiler or interpreter for any computer language. In addition to obvious use, e.g. the need to parse a real 'big' programming language such as Java, PHP or SQL, it can help with smaller, more common tasks.

It is useful any time you need to evaluate expressions unknown at compile-time or to parse non-trivial user input or files in a weird format. Of course, it is possible to create a custom handmade parser for any of these tasks. However, it usually takes much more time and effort. A little knowledge of a good parser generator may turn these time-consuming tasks into easy and fast exercises.

ANTLR seems to be popular in open source word. Among others, it is used by Apache Camel, Apache Lucene, Apache Hadoop, Groovy and Hibernate. They all needed parser for a custom

language. For example, Hibernate uses ANTLR to parse its query language HQL.

ANTLR is code generator. It takes so called grammar file as input and generates two classes: lexer and parser.

Lexer runs first and splits input into pieces called tokens. Each token represents a more or less meaningful piece of input. The stream of tokes is passed to parser which does all necessary work. It is the parser which builds an abstract syntax tree, interprets the code or translates it into some other form.

Grammar file contains everything ANTLR needs to generate correct lexer and parser Whether it should generate java or python classes, whether parser generates abstract syntax tree, assembler code or directly interprets code and so on.

Most importantly, grammar file describes how to split input into tokens and how to build tree from tokens. In other words, grammar file contains lexer rules and parser rules.

Each lexer rule describes one token: TokenName: regular expression; Parser rules are more complicated. The most basic version is similar as in lexer rule: ParserRuleName: regular expression;

They may contain modifiers that specify special transformations on input, root and childs in result abstract syntax tree or actions to be performed whenever the rule is used. Almost all work is usually done inside parser rules.

In order to learn how to use Antlr 4, I used The Definitive ANTLR 4 [10] book, written by Terence Parr who is the man behind Antlr.

Chapter 8

Summary

In the introduction, I expressed the hope that the work in this thesis could be a "first step" towards programming on mobile devices. In this final chapter, I will conclude by describing the progress made towards this goal in terms of my development of the prototype, the experiments that we carried out, the concepts that have been developed and additional products that were developed during the research. I will also suggest some future research directions that could provide the next steps along the path to a practical and widely applicable inference system.

During the research we explored the domain of software development tools and programming environments. We developed a concept enabling the programmer to use mobile devices as programming environments in addition to PCs and laptops. We encountered a number of questions that we had to answer before starting to develop a concept and a prototype for such an idea. What language are we going to use? How will the code be represented on small screens? How will the programmer interact with the mobile device?

After we found solutions to all those questions we decided that we need to do a set of experiments to understand how a programmer interacts with a programming environment when s/he needs to use his or her voice. Following all the experiments we extracted the most significant information in order to understand how programmers tend to describe the code that they want to be written. The information that we extracted is essential for this study because it allowed us to determine the best direction concerning how to proceed with the study.

After we completed the experiments, we decided that we need to explore the existing programming environments and features in different programming languages, especially Java. We

collected a huge set of programming features, which we will need in future work in order to expand the usability of the application that will be built. Based on the features that we collected, we developed a series of concepts of how code may be represented on a small screen (mobile phones and tablets). Those concepts may be used in future work to complete the prototype and to prove that development on mobile devices using voice and touch is possible. We had the idea of allowing every programmer to configure the representation of the code that makes the most sense to him or her. Meaning, every programmer would be able to configure the representation of the code that would be easier for understanding. This is a very advanced idea and not easy for implementation, but we managed to develop a fundamental concept of how it may work.

As mentioned before, we found that in order to allow the programmer to code on mobile devices we have to develop an idea of programming by voice. Therefore, we researched related works, available systems and tools and developed two modules: a speech to text engine, based on Google Speech Server, and a context free grammar parser and lexer. We used Antlr4 for this. Those two modules will be used in future works.

During the research process we gave a three hour lecture in a research seminar of Advanced Software Tools in The Blavatnik School of Computer Science in Tel-Aviv University. Also, we published a paper in the MobileSoft Conference which is part of the ICSE conference. The conference, where we gave a presentation, took place in Florence in May 2015. For future work we plan to use the modules that have been developed up to today and to develop an IDE for mobile devices. In addition, the Parser and Lexer are designed and written in such a way so it will be very easy to expand their usability.

All projects are committed to GitHub and they are public, which means every programmer is able to connect to that repository and contribute as he or she wishes.

Appendix A

Publications

This appendix contains all additional products that this research yielded.

A.1   MobileSoft 2015 Conference

Our research group submitted a paper to a mobile research conference and it has been accepted. Details of the conference and the paper provided in the Following list.

- Name: MobileSoft 2015

- Conference Website: http://mobilesoftconf.org/2015/

- Location: Italy, Florance

- Dates: May 16-17, 2015

- Paper's Title: Deverywhere: Develop Software Everywhere

- Authors: Yishai A. Feldman, Ari Gam, Alex Tilkin, and Shmuel Tyszberowicz.

- Affiliations: IBM Research, Tel Aviv University, The Academic College Tel Aviv Yaffo.

This conference was the 2nd ACM International Conference on Mobile Software Engineering and Systems Sponsored by ACM SIGSOFT.

Deverywhere: Develop Software Everywhere

Yishai A. Feldman,* Ari Gam,† Alex Tilkin,‡ and Shmuel Tyszberowicz‡

*IBM Research – Haifa, Israel; Email: yishai@il.ibm.com
†Blavatnik School of Computer Science, Tel Aviv University, Israel; Email: theprap@gmail.com
‡School of Computer Science, The Academic College Tel Aviv Yaffo, Israel; Emails: alextilk@gmail.com, tyshbe@tau.ac.il

Abstract—**Professional programmers use desktop or laptop computers as a preference. However, they sometimes need to continue their work on the go, when they may only have access to mobile devices. Thus, mobile devices can be important but not exclusive development platforms. Therefore, it is necessary to support programming in conventional languages on mobile devices, such as phones and tablets.**

Programming on mobile devices presents two major obstacles: the lack of a physical keyboard, and the small screen space, which limits the amount of code that can be shown simultaneously. This paper addresses both challenges, and offers a method to enable programming on mobile and other devices with limited input and output capabilities, by using templates to make voice and touch input very effective for programming, and showing much more code in a limited space. These ideas are also relevant to programming on laptop and desktop systems, for people with disabilities such as repetitive-stress injuries (RSI) that limit keyboard usage, and partial vision loss, which requires the use of very large fonts.

I. INTRODUCTION

In the early days of computing, programmers had to work in offices. Personal computers allowed programmers to work at home as well. Laptops further expanded the working environment, and we often see people programming in coffee shops, terminals, trains, and airplanes. With ubiquitous mobile devices becoming increasingly popular, there is an opportunity to allow programmers to work in even more environments. While such small devices are unlikely to become the preferred working medium, they can be useful in circumstances where urgent action is required and other equipment is unavailable.

This scenario presents two major obstacles: first, the lack of a convenient keyboard; and second, the small screen space, which limits the amount of code that can be shown simultaneously. Some have advocated the creation of new programming languages for mobile platforms,[1] but the cost of adopting a new language, with its related tools and infrastructure, seems to be too great for the benefit of occasionally programming on a mobile device. This applies to the development of mobile and non-mobile applications alike; professional programmers who develop mobile applications still prefer to use large screens and physical keyboards. Instead, we focus on easy ways to use existing languages, such as Java and JavaScript, on mobile devices. Our proposed solution, called *Deverywhere*, addresses both challenges, by using templates to make voice and touch input very effective for programming, and to show much more

---

[1] See, for example, the "Theme and goals" section of the PROMOTO 2014 Workshop, http://research.microsoft.com/en-us/events/promoto2014.

code in a limited space. Templates, used in context, allow voice input for program creation, editing, and navigation; and allow a compact representation of programs that makes maximum use of the given screen space. Both uses require a high degree of configuration, since programmers have different preferences regarding the way they want to voice and see programs. The underlying representation is always the original language, so that each programmer can see a tailored view while seamlessly collaborating on the same code with others.

These ideas are also relevant to programming on laptop and desktop systems, for people with disabilities such as repetitive-stress injuries (RSI) that limit keyboard usage, and partial vision loss, which requires the use of very large fonts. For some programmers, no screen is large enough, and so we expect that these programmers will use the compact representation of code even on large displays.

II. PROGRAMMING BY VOICE AND TOUCH

Dictation systems exist today, but their use for programming is extremely limited. Lacking any domain knowledge, they require most of the program to be dictated letter by letter, which is impractical. By building an understanding of program syntax and some semantics into the dictation tool, it is possible to make this process much more efficient. For example, the spoken words "for i from zero to n" can be interpreted as the Java idiom

```
for (int i = 0; i < n; i++) body
```

The current insertion point would be left at the body of the loop. Furthermore, Deverywhere will know that this place in the template is called "the body," so that a further instruction to "edit the body of the for loop" will return to that point. Touch can alternatively be used for the same purpose.

Other locations can also be associated with templates. For example, the template above can define "the loop index" as referring to the variable i, so that the developer can later say "rename the loop index to j." The general form of a Java `for` statement can define the "initialization," "test," and "update" locations, referring to the three parts inside the parentheses. A conditional template may have three locations, referring to the condition, the consequent, and alternative.

Similarly, an "iterate" template can be defined to generalize the use of iterators that are not accessible through the `Iterable` interface. For example, suppose that `tree` is an element of class `Tree<Element>`, which does not implement `Iterable` but provides an iterator through a

method `inOrderIterator()`. The utterance "iterate on tree in order" will create the following code:

```
Iterator<Element> iterator =
    tree.inOrderIterator();
while (iterator.hasNext()) {
    Element element = iter.next();
    □
}
```

The box represents the current insertion point. In this example, the types of the iterator and the element have been inserted automatically, based on the type of `tree`. Their names have been chosen heuristically, but they can be changed by the developer; they can later be referred to as "the iterator" and "the element," respectively.

Such templates can be created for language constructs as well as for other types of patterns, such as application frameworks. For instance, we can dictate "event key is shift enter" for the following common Dojo expression that checks the details of a keyboard event:

```
event.keyCode == dojo.keys.ENTER
&& event.shiftKey
```

As part of the template-based input method, developers will be able to specify the kind of syntactic element they are about to enter, such as class, method, variable, or constant. A suitable template will be applied; for example, a constant in Java will automatically be defined as `public static final`. Similarly, a "main method" will open with the already supplied header `public static void main(String[] args)`. The same words ("main method," "constant X," etc.) can be used to navigate to the appropriate element. In addition, it should be possible to refer to elements according to their position in the text shown on the screen; for example, "first for loop," "inner if statement," "loop on i." These templates should be recognized regardless of how the code was entered. This implies that Deverywhere should recognize templates from the original language text, without relying on any external annotations.

A convenient way to add templates should be provided, as the number of possible templates is unlimited, and may even be programmer-specific. Each template is associated with utterances to create it, with named locations, and with its compact representation. The utterances form a grammar, which need not be completely unambiguous, since the development environment can offer a choice between alternatives. This, however, should be avoided as much as possible.

The same utterance and compact representation can be associated with more than one code template, in order to support multiple programming languages. One of the significant advantages of this template-based system is its ability to treat multiple languages in a similar way (to the extent that the languages provide similar mechanisms, of course).

Context is crucial to understanding. For example, names in a program are usually limited to a relatively small set, which depends on the current scope; a number of methods can be used to select the correct one efficiently. One way is to start naming a variable (or class name, method name, etc.) either by spelling or, if it is composed of known words, by sounding them. Once the choice becomes small enough to show it on the screen, completion can be made by sounding the number of the correct choice. Another way is to assign short nicknames to variables and other named elements, then refer to them by their nicknames. The initial definition of a variable cannot be made in this way, as the space of choices is unlimited. If the name is a known word or a series of known words, they can be dictated and Deverywhere can join them in the way appropriate for the programming language and the type of the element; for example, using CamelCase with appropriate capitalization in Java.

Refactoring and other source transformations, as in Eclipse, are a must for Deverywhere, with the appropriate modification of the relevant wizards to work with voice entry. In addition, other capabilities would be useful. For instance, suppose the developer wants to use the result of a method call that is part of an expression in another context. In Eclipse, the developer would have to mark the method call expression, then apply the Extract Local Variable refactoring, then move the generated variable definition upwards if necessary, and finally use it. Instead, in the voice-programming system, the developer will be able to say "use the result of the second call to substring." This will create the variable at the correct position, and insert a use of the variable at the current insertion point.

Statically-typed languages such as Java are often very verbose, especially in the specification of types. However, type inference methods exist for these languages, and current IDEs use them to identify errors and fix them automatically. The voice-programming system will not require (yet allow) the specification of types at any point, and will try to infer as much information as possible. In the example above, "for i from zero to n," it is clear that `i` is an `int`, and this need not be mentioned at all. When type inference fails, information about the variable may be limited; for example, the system will not be able to suggest methods for that variable. Therefore, the developer should always have a verbal command that adds a type definition to the variable at the current insertion point. This will add the missing type at the point the variable is defined, but will not change the current view or the current insertion point, so that the developer can continue choosing the method to call without further distractions.

III. PROGRAMMING ON SMALL DISPLAYS

While mobile displays are getting larger, they are still significantly smaller than the displays developers are used to. Even a 7" phone/tablet provides much less screen space for programming than a typical 21" desktop display. The amount of code that developers can see simultaneously has a great effect on their productivity. Special techniques are therefore required in order for programming on small displays to be effective. The templates that are used for voice entry can also be used for display.

There are many ways to show information visually in a compact way; for example, using special symbols, colors,

Fig. 1: The `LinkedList` class, as would be shown by Deverywhere.

fonts, backgrounds, borders, and even watermarks. Some information, such as noise words (`then`, `else`, `end`, etc.), types, throws declarations, access levels, and package prefixes, can even be omitted altogether (with an option of showing them selectively, perhaps using a touch gesture).

The example in Figure 1 shows many of these techniques; it may be extreme for some tastes, but, as mentioned above, the way program features are shown in the compact representation must be individually configurable, for the preferences of each developer and also based on screen size. This particular presentation of the program should be thought of as an example of how Deverywhere can be customized rather than as the definitive output format. The figure shows the code separated into three parts, each of which fits the size of a reasonable smartphone.

This code of example is taken from the University of Texas CS307 course of 2011 (https://www.cs.utexas.edu/~scottm/cs307/javacode/codeSamples/LinkedList.java), and implements a linked list. It has not been modified in any way except for the presentation, and the intent is for the code kept in the source-control repository to be the same as the original.

This example demonstrates many possible features of compact representations of programs. Names of enclosing classes and methods appear on the bar that also serves for the indentation, unless the bodies are very short. Many language elements are omitted (comments, types, keywords such as `class` and `return`, empty pairs of parentheses, and symbols such as statement terminators and braces); others have been replaced by short notation (+ and − for `public` and `private`, the equivalence symbol for ==, arrows for ++ and −−, and special

notation for constructors, conditionals, and loops). Background colors are used to show scopes (shades of blue for classes and methods, green for the `then` and red for the `else` part of conditionals); the scope of a constructor is denoted by the extent of the crane symbol. Note that class and method names appear at the start of every relevant screen, not just at the beginning of their scopes.

Templates have been used in several places to make code more concise (and perhaps also more readable). Getters and setters have been compressed to look like field reads and writes, as is done in several languages (but not in Java). The common mathematical idiom $0 \leq x < s$ is used instead of the cumbersome notation that uses a conjunction of two inequalities. The same notation is used here for loop bounds, in the (very common) case of a unit increment.

Conditionals are shown here using the same block structure as classes and methods, except for the colors used to denote the different branches. One exception, demonstrating a somewhat more conventional way of representing a conditional, appears inside the `remove` method in part (b). This notation is similar to the three-way conditional expression of Java, but uses a different symbol to denote that it is a statement rather than an expression.

Loops are shown with a circular watermark that denotes the scope, and a special symbol ($\forall$) to indicate the loop control. This is perhaps the most unusual notation we show in this paper; while it may be chosen by few developers, we use it to show the variety of notation that can be used for a compact representation of programs.

The last method, `toString()`, is different from the other parts of this example. In some cases, it is useful to show some

documentation (perhaps, but not necessarily, the Javadoc) instead of the method body. This offers a very concise view that shows the intent without the implementation; this is often very useful.

Other possibilities, not shown in this example, include automatically inlining a value that is only used once provided the resulting expression is not too large; this could have been done (twice) for `newNode` in part (c). Existing Eclipse templates, such as expanding `sysout` to `System.out.println`, can be used in reverse to compact programs that use the expansions of such templates. This can be extended even further; for example, given an analysis that converts imperative idioms to functional ones, such as LambdaFicator [6], the view can show the functional form without changing the underlying program.

As can be seen from this example, the representation is sometimes ambiguous. This seems to us to be acceptable, since the meaning will in most cases be obvious from the context. In any case, the developer will always be able to ask to see more details (perhaps by touching locations for which more information is desired).

Program slicing, and especially Fine Slicing [1], is a very effective technique for showing a small part of the code that is relevant to a particular purpose. When browsing code, slicing can be used to prevent the need for a lot of scrolling. The program view would normally consist of a single method, possibly with some context given as a breadcrumbs view. In some cases, it is useful to show enclosing conditionals and loops, but without intervening details; this is a kind of poor-man's slicing that is more easily implemented.

IV. ROADMAP

In order to map the requirements and possibilities of development on limited platforms, we performed several dictation experiments where the speaker tried to dictate a program as naturally as possible, and the writer attempted to understand as literally as possible. In addition, we studied previous work on mobile programming environments [5], [7], compact representations [3], [8], [9] and dictation [2], [4], and abstractions used by various programming languages.

Previous work on mobile programming environments [5], [7] has focused on new languages that are more natural for the mobile environment. In contrast, we believe that such new languages will have very limited use, mostly for small applications. Professional development will continue to be done in more established languages, and a mobile solution for professional programmers should support these languages without requiring changes in the underlying technology.

We found many interesting ideas in Intentional Software [9] and registration-based abstractions [3], [8], which have discussed multiple views of the same underlying program. VoiceCode [4] and Spoken Java [2] focus on voice entry of programs. Deverywhere combines voice input with a compact presentation using the same set of templates.

The results of our study are several lists and relationships. One list contains features that a mobile programming environment can support; in addition to those discussed above, the list includes items such as the automatic application of quick fixes, renaming conflicting elements, extension methods [10], and two-dimensional expressions (as in mathematics). A second list contains features for compact representation; a third lists features for voice input; and a fourth lists configuration modes, such as the use of typographic styles, layouts, frames, and watermarks. We also created a matrix relating input and output options with the various ways each of these can be shown; this will be the basis for the developer-specific customization.

V. CONCLUSION

Deverywhere relies on templates that have information about their structure that supports flexible voice input and various types of compact representations. Customizability is crucial to support different personal styles as well as different sizes and capabilities of programming environments.

Once fully implemented, Deverywhere has the potential to make programming on devices with limited user interfaces, such as mobile phones and tablets, much more convenient than they are now. This will allow developers to work more comfortably in environments in which programming was very difficult.

These techniques can also provide enormous help to developers who have various kinds of disabilities that prevent them from using existing interfaces on laptop and desktop environments effectively. In fact, we conjecture that many developers would enjoy having the benefits of Deverywhere as additions to their normal working environments.

REFERENCES

[1] A. Abadi, R. Ettinger, and Y. A. Feldman. Fine slicing: Theory and applications for computation extraction. In *Proc. 15th Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 471–485, Mar. 2012.

[2] A. Begel and S. L. Graham. Spoken programs. In *IEEE Symp. Visual Languages and Human-Centric Computing*, pages 99–106, Sept 2005.

[3] S. Davis and G. Kiczales. Registration-based language abstractions. In *Proc. ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 754–773, 2010.

[4] A. Désilets, D. C. Fox, and S. Norton. VoiceCode: An innovative speech interface for programming-by-voice. In *Extended Abstracts on Human Factors in Computing Systems (CHI)*, pages 239–242, 2006.

[5] G. Essl. Mobile phones as programming platforms. In *Programming Methods for Mobile and Pervasive Systems (PMMPS)*, 2010.

[6] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. LambdaFicator: from imperative to functional programming through automated refactoring. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1287–1290, 2013.

[7] S. Li, T. Xie, and N. Tillmann. A comprehensive field study of end-user programming on mobile devices. In *IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC)*, pages 43–50, September 2013.

[8] J.-J. Nunez and G. Kiczales. Understanding registration-based abstractions: A quantitative user study. In *Proc. IEEE Int'l Conf. Program Comprehension (ICPC)*, pages 93–102, 2012.

[9] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *Proc. 21st Annual ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 451–464, 2006.

[10] Xtend – modernized java. http://eclipse.org/xtend.

Appendix B

Acronyms

NLP Natural Language Processing

ASR Automatic Speech Recognition

STT Speech to Text

JSON (JavaScript Object Notation) is a lightweight data-interchange format

API Application Programming Interface

CFG Context Free Grammar

EBNF Extended Backus Naur Form

IDE Integrated Development Environment

Bibliography

[1] Iso/iec 14977 : 1996(e), extended backus naur form (ebnf) a standard syntactic metalanguage.

[2] Andrew, B. and Susan, L. G. (2011). *Spoken Programs.* PhD thesis, Computer Science Division, EECS University of California, Berkeley.

[3] Eisenberg, A. D. (2008). *Presentation Techniques for more Expressive Programs.* PhD thesis, The University Of British Columbia.

[4] Feldman, Y. A. (2013). Template-based development on mobile platforms. Technical report, IBM Research.

[5] Feldman, Y. A., Tilkin, A., Gam, A., and Tyszberowicz, S. (2015). Deverywhere: Develop software everywhere.

[6] Graham, S. L. (2015). Harmonia research project. http://harmonia.cs.berkeley.edu/harmonia/index.html.

[7] Mey, G. D. (2015). Google speech api v2. https://github.com/gillesdemey/google-speech-v2/blob/master/README.md.

[8] Michal, G. and David, H. (2009). Programming in natural language. *10th Int. Conf. on Computational Linguistics and Intelligent Text Processing (CICLing'09)*, pages 456–467.

[9] Nuance, D. (2015). Nuance dragon. http://www.nuance.com/for-developers/dragon/index.htm.

[10] Parr, T. (2012). *The Definitive ANTLR 4 Reference.* The Pragmatic Programmers, LLC.

[11] Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O., Goel, N., Hannemann, M., Motlicek, P., Qian, Y., Schwarz, P., Silovsky, J., Stemmer, G., and Vesely, K. (2011). *The Kaldi Speech Recognition Toolkit*. IEEE Signal Processing Society. IEEE Catalog No.: CFP11SRW-USB.

[12] Schulz, F. (2015). Speech to text library for java/processing. `http://stt.getflourish.com/archive/`.

[13] Sihan, L., Tao, X., and Tillmann, N. (2013). A comprehensive field study of end-user programming on mobile devices. Technical report, North Carolina State University.