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

Alex Tilkin

November 2015

MASTER DISSERTATION

Wotson 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 idea for this thesis yielded by Dr. Yishai Feldman (IBM Research). This research is in cooperation with Ari Gam who is M.Sc. student in Computer Science from Tel-Aviv University. The readers of this document assumed to have knowledge in Java language, system engineering, and software development tools.

Trondheim, 2012-12-16

(Your signature)

Alex Tilkin

Acknowledgment

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

A.T.

Summary and Conclusions

Here you give a summary of your your work and your results. This is like a management summary and should be written in a clear and easy language, without many difficult terms and without abbreviations. Everything you present here must be treated in more detail in the main report. You should not give any references to the report in the summary – just explain what you have done and what you have found out. The Summary and Conclusions should be no more than two pages.

You may assume that you have got three minutes to present to the Rector of NTNU what you have done and what you have found out as part of your thesis. (He is an intelligent person, but does not know much about your field of expertise.)

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 and Java Script, 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 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.

1.1 Background

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

Several works have been that related to this problem. Part of works deal more with research and part deal more with application. Susan L. Graham and Andrew Begel from Berkeley university Worked on a project named *SPEED*. In their work they developed an add-on that integrated 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 simple and clear environment which allows the developer to develop application right 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 right on the the mobile device and run them.

None of the works that described above provides a comfortable solution for IDE on mobile devices that fully integrable with stationary computers. None of the works addresses the issue of lack of keyboards and small screen. Our work address 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 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.

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, 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. In this work we concentrate on several targets: design a new representation of the code so it will fit on mobile screens and will be readable as well; Create a set of templates that will allow the programmer to program by dictating the code; 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 treat problems that are similar to what we are studying:

- It is shown in Andrew and Susan [1, Chap. 2] that programmers ran into problems of expressing their thought invoice when they had to dictate a program. This information is very important to us. We designed our experiments based on it

- It is shown in Andrew and Susan [1, 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 and integrate it in our system

- Graham [2] provide all the information about how to use the Harmonia tool and how to integrate in programming tools. It is useful information for for future work
- It is shown in Yishai [3, Programming By Voice and Touch] the basic ideas and concepts of our work. Basically this paper is the start-up point of 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 who are related to the Java and generally to programming. All those features have a potential to be integrated into the system
6. Build a prototype that proves that developing on mobile devices is possible
7. Provide a solid foundation for future works based on this research

The objectives shall be written as *fundamental objectives* telling what to do and not *means objectives* telling how to do it.

All objectives shall be stated such that we, after having read the thesis, can see whether or not you have met the objective. “To become familiar with ...” is therefore not a suitable objective.

1.5 Limitations

Our study have several limitations. The following is a list that represents limitations that we have in this study:

- This project requires a lot of codding work. Since I'm the only programmer the implementation will be very limited
- The analysis part it very long is difficult. Might require a lot of time of the research
- Not many studies have been done who relate to the field "Developing on Mobile Devices" or "Developing using Voice" are quite rare. Therefore, we don't have a many resources that we can learn from. Most the work we'll have to do on our own
- To implement this system a knowledge in 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. After we will manage to collect enough programming features (cover enough domains) we will study other languages how those features represented there. We will search in related works for new ideas for representations and will have discusses about new ideas. After all those process will be 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. After we will have both groups, we will find the relation between members from the second group with elements from the first group.
3. We will build a series of experiments. Every experiment will represent different style of programming, for example, object oriented, algorithmic. There will be two programmers,

one will be the typist and the other will be the speaker. The speaker will ask from the typer to create a program that is already written in Java. The speaker will dictate the program and the speaker will type. After all experiments will be accomplished we will compare the source the and the reference (dictated program). And will analyze the difference between the desired results and the actual results.

4. We will extract all commands that have been pronounced 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. 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 contained of a set of commands. Every command is a simulation of a dictation that the programmer pronounced. I will build a limited set of rules that know how to handle the commands and appropriate line will be written on the screen. The command will be in compact mode.

1.7 Structure of the Report

Remark: Write here the structure of the document

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 grouped into categories. Based on the repository will create templates that will help to analyze the pronounced commands.

Every experiment contained 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 typer. In every experiment the typer gave to the speaker a programming task where he need to implement a program.

The speaker had to dictate a program and the typer had to type exactly what the speaker dictated. The speaker had to dictate lines of code in such way so the typer could understand what does he means but not too detailed. For example, if the speaker had to dictate the code in Figure 2.1. He would dictate it like this, *"For each element in elements call to to string"*.

The typer need to follow dictations of the speaker and to type the code to the text editor (all four participants could see the screen). The typer typed the code in Java. Every one of the participants could participate and provide suggestion for pronouncing the commands. The typer could delete, edit and navigate in the code with no limitations. No time constrains and no limitations on the amount of lines. All experiments have been recorded.

After all experiments have been performed we analyzed them and extracted only the relevant lines that represent commands. For each experiment we created a table that contained of two columns. The left column contains the commands that have been dictated and the right column represents the code that has been typed.

Remark:The commands that have been inserted into the tables are filtered from irrelevant vowles. 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 foreach loop where every item in elements activates it's toString method

2.2 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 presented to the speaker during all the experiment.

Table 2.1 represents the order of the commands that have been 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.

```
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 $0 \leq i <$names.length
Create value type of object accepts table.find, accepts names at i's index	value $\leftarrow$ table.find(names[i])
If value different from null then	value $\neq$ 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 $0 \leq i <$names.length
If names at i's index period equals accept name then	names[i].equals(name) ?
Return values at i's index	$\hookleftarrow$ 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

2.3 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 presented to the speaker during all the experiment.

Table 2.2 represents the order of the commands that have been 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.

```
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

```
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	$\hookleftarrow$ 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	$x < 0.0 \Vert x \geq \text{width} - 1.0 \Vert y < 0.0 \Vert y \geq \text{height} - 1.0$?
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	$x < -1.0 \Vert x \geq \text{width} \Vert y < -1.0 \Vert y \geq \text{height}$? $\hookleftarrow$ 0.0
else return a call to getInterpulatedEdgeValue that accepts parameters x and y	: $\hookleftarrow$ getInterpulatedEdgeValue(x, y)
We are done with the outer if	
Assign x to xBase	xBase $\leftarrow$ x
Assign y to yBase	yBase $\leftarrow$ y
Subtract xBase from x and assign it to xFraction	xFraction $\hookleftarrow$ x - xBase
If xFraction is less than zero period zero then assign zero period zero to xFraction	xFraction < 0.0 ? xFraction $\leftarrow$ 0.0
We are done with the if	

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

The speaker said	The typer 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 have been dictated during experiment No.2

2.4 Experiment No.3

2.4.1 Part A

- Date: 12/May/2014.
- Speaker: Ari Gam.
- Typist: Alex Tilkin.
- Description: Implement the Bubble Sort algorithm . The speaker 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 have been dictated (top to bottom). Figure 2.6 represents the result of the dictation by the speaker in part A.

```
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

2.4.2 Part B

- Date: 28/Apr/2014.
- Speaker: Ari Gam.
- Typist: Alex Tilkin.

- Description: After the speaker has completed the implementation of the Bubble Sort algorithm he has been asked to improve it's time complexity by adding additional condition. No source code presented to the speaker.

Table 2.4 represents the order of the commands that have been 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 has been added to the Bubble Sort algorithm

The speaker said	The typer 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 $0 \le i < data.length - 1$
Create an inner loop from zero to i	for $0 \le j < i$
If the i element of data bigger than the j element of data then switch between them	$data[i] > data[j]$? $temp \leftarrow data[i]$ $data[i] \leftarrow data[j]$ $data[j] \leftarrow temp$
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 $0 \le i < data.length - 1$ & done
undo	for $0 \le i < data.length - 1$
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 have been dictated during experiment No.3

2.5 Experiment No.4

- Date: 19/May/2014.
- Speaker: Alex Tilkin.
- Typist: Yishai Feldman.
- Description: A program that simulates TV controller has been 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 *DeviceButton* class. This class contains the *main* method.

The following tables represents the order of the major commands that have been 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

```
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 have been 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 have been 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 have been dictated during experiment No.4

2.6 Experiment No.5

- Date: 03/07/2014.
- Speaker: Perry Shalom.
- Typist: Alex Tilkin.
- Description: In this experiment the typist has been 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 null or empty string the constructor should return and not initialize anyone 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 presented to the speaker.

Figure 2.16 represents the dictation result of experiment No.5. Figure 2.17 represents the compact representation of Figure 2.16. Table 2.9 and Table 2.10 represents the major commands that have been 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 have been dictated during experiment No.5

The speaker said	The typer 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 have been dictated during experiment No.5

Chapter 3

Supported Features

3.1 Introduction

This chapter discusses about 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 in to Deverywhere system and to increase it's functionality. All features have been discussed by the research group whether they have a potential to be integrated or not. The features that 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 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 meaning of priority is how important that feature to this research.

3.2 List of Features

3.2.1 Programming by Voice (writing)

- Priority: 1

Allow the user to program using his voice.

- In case the system stumbles a case of ambiguity it will present options to the user which he could choose the most appropriate solution.
- The system should distinguish between when the user dictates to it or speaking to someone (this is very complicated feature to implement so it might be postponed to later works).

3.2.2 Navigation by Voice

- Priority: 2

The user could navigate in the code using his voice.

3.2.3 Editing by Voice

- Priority: 3

The user could edit the code by using his voice.

3.2.4 Compact View Mode

- Priority: 1

Provide an easy 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. Due to it is complicated to support all refactoring features we decided to choose several 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 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

Appendix A

Acronyms

NLP Natural Language Processing

ASR Automatic Speech Recognition

RAMS Reliability, availability, maintainability, and safety

Appendix B

Additional Information

This is an example of an Appendix. You can write an Appendix in the same way as a chapter, with sections, subsections, and so on.

B.1 Introduction

B.1.1 More Details

Include more appendices as required.

Bibliography

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

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

[3] Yishai, F. (2013). Template-based development on mobile platforms. Technical report, IBM Research.