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.

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 differ-

ent 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 [5] 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 Feldman [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.

```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

```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){
            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

2.3   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.

```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 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.

```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

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 ≤ 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 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 *Device-Button* 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

```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 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 ≠ null ∧ body ≠ null ∧ engine ≠ null ?
Return to Car	
Create a constructor that receives its three fields and initializes them	+Car(body, engine, wheels) _body ← body _engine ← engine _wheels ← 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 ∨ wheels = null ∨ engine = null ? ↩
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	↩ 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

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 like: navigation, refactoring and any other feature that requires from the user to point where he want to take the action. Below is a list of several features that provides example where is it needed.

- Refactoring - When a user asks to refactor a certain object he say something lie "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 a high level commands and generate code automatically. Due to the complexity to implement 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 a 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 can not be resolved. Change visibility for types that are accessed but not visible. Rename to a similar type for references to types that can not be resolved. Add import statement for types that can not 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 is not matching 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 can not 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 can not be resolved. Rename to a similar method for references to methods that can not 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 can not be resolved. Rename to a variable with similar name for references that can not 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 generalize 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 response not only with textual output but also with suggestions and recommendations for his

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

3.2.13 Undo, Redo

- Priority: 2

Every step that have 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 pronounce 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 it's 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 pronounce 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 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. The technique is to keep 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 not both 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 information and can be omitted. The proposal is to emit the lines that contains 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 providing the scope of code that the user wants to see it's source code.

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 [2].

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 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. Inter-procedural Flow has several applications, 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

Prototype

4.1 Introduction

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

This prototype presents an ability of programing using speech and represent 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 used is discussed and described.

4.2 Speech to Text

This is a module which responsible for the translation from speech to text. The engine that is used for the STT process is Google Speech V2 server. It is shown in Mey [6] how to interact with this server.

4.2.1 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 need to be used is `https://www.google.com/speech-api/v2/recognize`.

Request

The request that is need to 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_c2OseW
```

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 it's 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
    }
```

4.2.2   Speech to Text Library

This library 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/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 [7] where to get the source code from and how to use it's API.

Appendix A

Additional Products

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 (a):

```
+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↓
```

Fig. 1 (b):

```
+LinkedList
  +makeEmpty
    head ← tail ← null
    size ← 0
  +remove(pos)
    assert 0 ≤ pos < size
    pos ≡ 0
      result ← head.data
      head ← head.next
      size ≡ 1 ◊ tail ← null
    else
      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
```

Fig. 1 (c):

```
+LinkedList
  +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
    size↑
  +addFirst(obj)
    assert 0 ≤ pos < size
    size ≡ 0 ?        add(obj)
    else
      newNode ← new Node(obj, head)
      head ← newNode
      size↑
  + toString
    Join data with spaces from head by next
```
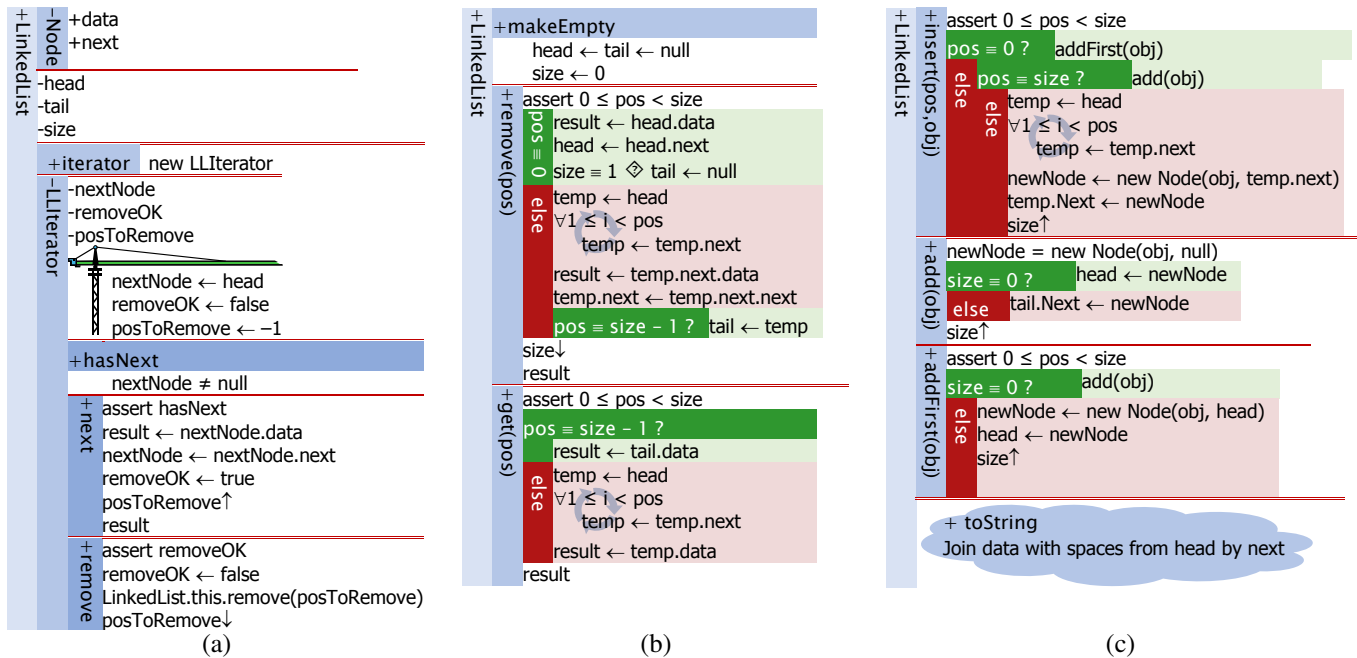
Fig. 1: The `LinkedList` class, as would be shown by Deverywhere.

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 $0 \le x < s$ 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

Appendix C

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.

C.1 Introduction

C.1.1 More Details

Bibliography

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

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

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

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

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

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

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

[8] 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.