UNIVERSITY OF BIRMINGHAM

School of Computer Science

College of Engineering and Physical Sciences

MSc. Project

Jumping Machine Implemented On Computer

Submitted in conformity with the requirements
for the degree of MSc. Computer Science
School of Computer Science
University of Birmingham

Lixuan Dai, MSc.
Student ID: 1926198
Supervisor: Dr Paul Levy

September 2019

Declaration

The material contained within this report has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this report has been conducted by the author unless indicated otherwise.

Signed .....

Lixuan Dai

"You have to learn the rules of the game.

And then you have to play better than anyone else"

Albert Einstein

MSc. Project

Jumping Machine Implemented On Computer

Lixuan DaiDegree Postnomials

Contents

1 Abstract **1**

2 Acknowledgements **2**

3 Introduction **3**

3.1 Background . . . 3

3.2 Goals . . . 3

3.3 Structure of the paper . . . 3

4 Research **4**

5 Project specification **5**

5.1 Jumping machine . . . 5

5.2 Graphical syntax . . . 7

5.3 Execution principle . . . 8

5.4 Requirement specification . . . 9

6 Design **11**

7 Implementation **15**

7.1 Implementation of code builder . . . 15

7.2 Implementation of code executer . . . 17

7.3 Some features of the software . . . 20

7.4 testing . . . 22

8 Evaluation **27**

9 Conclusion **29**

9.1 Limitation . . . 29

9.2 Further work . . . 30

References **32**

10 Appendix A: Submission **33**

10.1 Git repository . . . 33

10.2 File structure . . . 33

10.3 How to run the software . . . 33

11 Appendix B: 1st generation prototype **34**

12 Appendix C: A running-on-paper example **36**

Table of Abbreviations

CBN	call-by-name
CBPV	call-by-push-value
CBV	call-by-value
GUI	Graphical User Interface
JVM	Java virtual machine
pm	Pattern match

1 Abstract

A Java-based software is built to implement a kind of abstract machine called jumping machine, which can build code diagram using graphical syntax and simulate the execution procedure of program. The key feature distinguishing jump machine from other abstract machines is that jumping machine can embody the jumping procedure in program execution. Therefore, it can be used as a pedagogical tool to help new programmers to understand the program execution procedure.

2 Acknowledgements

First and foremost, I would like to show my deepest gratitude to my project supervisor, Dr. Paul Levy for his patience, wisdom and kind help. He is a respectable and responsible scholar who gave me a lot of valuable suggestion. Without his enlightening instruction and impressive kindness, I could not have finished the project.

Besides, I shall extend my thanks to my friends for their encouragement and support.

Last but not least, I would like to thank my girl, Jiayu Min for staying with me,

3 Introduction

3.1 Background

When a program is executed, there are two kinds of jumping existed in this procedure.

The program will jump into a function when the function is called, and it can jump back to the original code when the called function returns. A graphical abstract machine defined in (Levy 2005) can embody the jumping process vividly and specifically. This machine can be used as a teaching tool in computer science to help students to understand the program execution and arise their interest on programming language.

3.2 Goals

(Levy 2005) shows the how easy to run a program on paper using jumping semantics. In this project, we try to develop a software to implement the jumping machine. Therefore, it needs to achieve two basic goals:

a. Build a code diagram.
b. Execute the code using jumping semantics.

3.3 Structure of the paper

In this section, we have already introduced the basic information about the project, including the research background and the objectives we need achieve. In section 2, we review some related literature of programming language and abstract machine, and make a comparison with some abstract machine. In section 3, we illustrate the definition of the jumping machine and the graph syntax, we also list the detailed requirement specification of the software. In section 4, we give the design of the software. In section 5, we explain how we implement the software. In section 6, we evaluate the process and the product. In section 7, we conclude the work we have done, and also point out the limitations and what we can do in future of this software. In appendix A, we give the access to the software as well as the file structure. In appendix B, we provides the first generation prototype of the software. In appendix C, we give an example of implementing the jumping machine on paper.

4 Research

To implement the jumping machine, first of all we need to know the definition of abstract machine. According to (Diehl et al. 2000), abstract machines are machines because they permit step-by-step execution of program, but comparing to the real hardware machine, they omit many details and specific implementations. Abstract machine is an important concept in theoretical computer science, it is widely used in various fields, especially for computational inference and algorithm complexity analysis.

A well-known abstract machine is Turing machine, which was designed by (Turing 1936). In this paper, Turing gave the definition of automatic machine.Automatic machine is a machine that does not rely on the external operator, instead all of its motions are completely determined by the configuration. The Turing automatic machine is an ideal device which consists of a head and an infinite tape, and it also has a simple instruction set, including reading or writing the content on tape and moving tape. According to Turing's theory, this kind of machine can be used to compute any computable sequence.

Another example is Java virtual machine (JVM), which is the corestone of Java platform. JVM is a stack-based machine. As a virtual machine, JVM does not assume any particular implementation technology, host hardware, or host operating system (Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley 2015). The Java language shields the platform-specific information using the Java virtual machine, so that the Java language compiler can generate the target code (bytecode) running on the Java virtual machine and run it on various platforms without modification. When the Java virtual machine executes bytecode, it interprets the bytecode as machine instruction execution on a specific platform. This is why Java can "compile once, run everywhere" (n.d.).

In addition to the abstract machines we have listed above, there are a variety of abstract machines that can be classified according to their semantics, language characteristics, data structures, and implementation functions.

Due to time and space limitations, we have not done a very in-depth study of abstract machines. But this field has a very broad research direction for researchers.

5 Project specification

5.1 Jumping machine

The language of the jumping machine is call-by-push-value (CBPV), which has been introduced in paper (Levy 1999). CBPV is a new typed paradigm based on Filinski's variant of Moggi's computational λ-Calculus, which contains both features of call-by-name (CBN) and call-by-value (CBV) calculi, and both of them can be translated into CBPV. The key features of CBPV can be concluded as that CBPV has two typed constructors for `thunks` and `producer`, and `values` and `computations` is regarded as two disjoint classes in CBPV. Besides, the traditional operational semantics: higher-order definitional interpreter and CK-machine(Felleisen & Friedman 1986) is also involved in our jumping machine.

Like the CK-machine, our jumping machine also has a stack, which can store frames, values and tags. The context in the stack may change when executing the code.

In this paper, we only discuss the instructions involved in the jumping machine, which will be listed as following:

- **print** - The print instruction is a single instruction which will neither change the content of the stack nor lead to a jump.

- **let be**- The instruction let V be x. M means we would bind the value of V on x then evaluate M.

- **push into stack** - The instruction V can be regarded as operation push V in jumping machine.

- **pop from stack** - The instruction x can be regarded as operation pop x in jumping machine. If the stack is empty, the program will terminate.

- **pattern match** - The pattern match is implemented in the jumping machine as the form pm as, which can be used to handle conditional branches and value matching. There are two types of pattern match, and they can be presented in term syntax as:

  – 1. $\texttt{pm}\ (i, V)\ \texttt{as}\ \{\ldots, (i, \texttt{x}).M_i, \ldots\}$

  It can be used to handle conditional branches, we can find the correct branch to execute by comparing the value of i, then we would bind the value V on x and evaluate M.

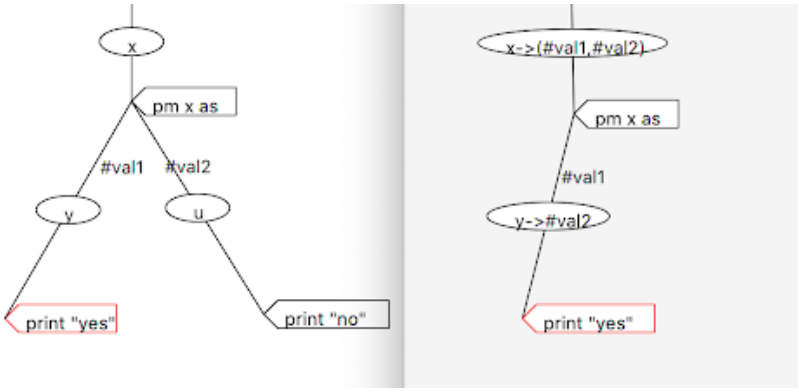

Figure 5.1: *The graphical syntax of pattern match 1*

– 2. $\texttt{pm}\ (V, V')\ \texttt{as}\ (\texttt{x}, \texttt{y}).M$ It would divided the set (V, V) into two parts and then bound to x and y respectively.

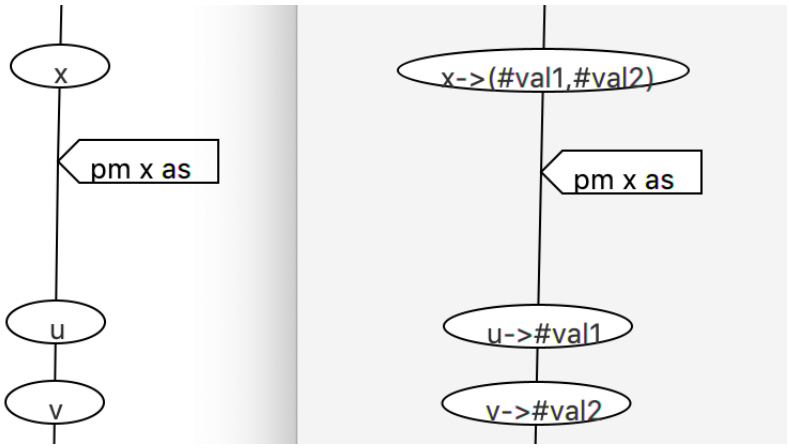

Figure 5.2: *The graphical syntax of pattern match 2*

Figure 5.1 and 5.2 also gives the graphical syntax form to present pattern match, and the definition of graphical syntax is given later.

- **to** - To execute the instruction M to x. N, we evaluate M first, and replace x by the return value of M, then evaluate N. The form [.] to x. N is called frame in jumping machine. Generally, we push the frame into the stack when executing it.

- **force** - To execute the instruction force thunk M, the program jumps to the position of the thunk point and then evaluate M.

- **return** - To execute the instruction return V, we will check the status of stack, if it is empty, the program terminates, and the final return value is V. Otherwise we pop the frame [.] to x. N from the stack, then make a bind between x and V, then evaluate N.

5.2 Graphical syntax

The graphical syntax is given in (Levy 2005), which can be used in jumping machine to write the program. It will be illustrated in this section, and examples of graphical syntax are also given as following in Figure 5.3.

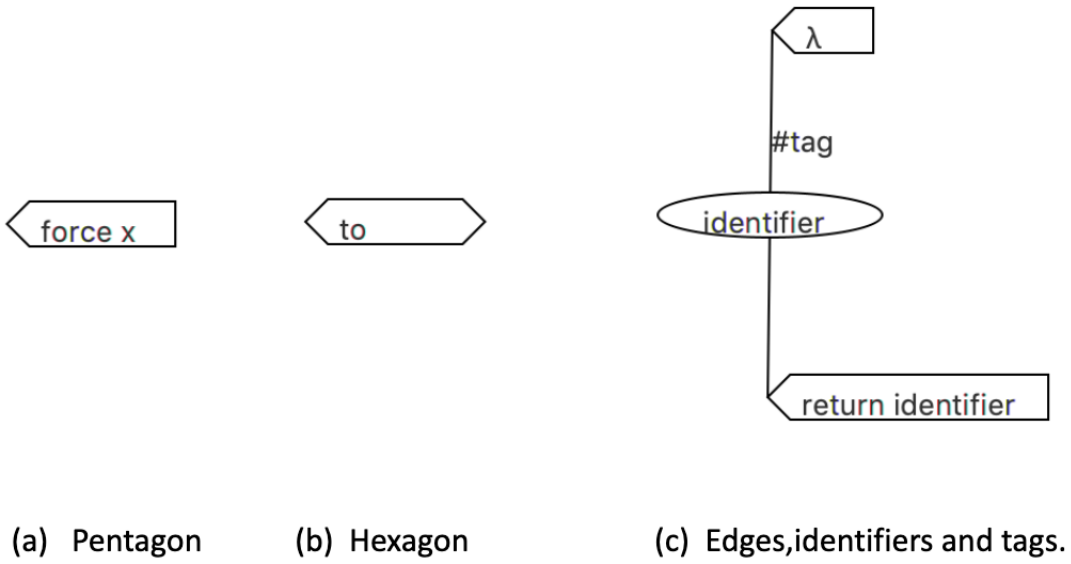

Figure 5.3: *The examples of graphical syntax*

- **Polygon** - Polygons are used to represent instructions. There are two kinds of. polygon in the syntax, the pentagon and the hexagon. In the program diagram, each instruction will be enclosed in the polygon.

  - **Pentagon** - Pentagon presents almost instructions except the sequencing instruction to.

  - **Hexagon** - Hexagon only presents the instruction to, which is also regarded as a. frame in jumping machine. When we execute the instruction, we would push the frame to the stack

    .

- **Point** - There are two kinds of point defined here, the normal link-point is the leftmost vertex of the polygon. Besides, there are two jump-points. The thunk is regarded as a point, when we execute the force instruction, we will jump to the corresponding thunk point. And the frame point is the rightmost vertex of a hexagon. When we return a value, we pop the frame from the stack and jump to corresponding frame point.

- **Edge** - The edge is used to connect instruction polygons. To execute the program, the program counter can find next instruction according to instructions and edges.

  Besides, the bindings and tags are placed on edges.

- **Tag** - Tags are placed on the edges. For a polygon, there may be more than one. edges on its link-point, which is called conditional branches. We can distinguish these edges by adding tags, and we would choose exact one edge according to the tags and instructions when executing the code.

- **Ellipse** - The ellipse in this diagram is used to present the identifier.

5.3 Execution principle

The jumping machine also follows the Von Neumann architecture. Therefore, the execution cycle can be regarded as the fetch-decode-execute cycle.

Initially we have a code tree, when executing, we will build another tree in graphical syntax called trace tree.

And the execution process is described as following:

- **Fetch** - We copy the instruction polygon that pointed by the program counter from the code to trace.

- **Decode** - We decode the instruction obtained in fetch cycle. In this phase, all points are replaced by pt i, where i indicates the position of the point in trace. Besides, we need look up the branch of the trace and replace the identifier by its binding value.

- **Execute** - We execute the instruction and draw an edge. The edge usually starts on the link point of the obtained polygon unless a jump occurs. In this case, we should find the corresponding jump-point and draw an edge from this point. Meanwhile, the program counter points to the next instruction polygon.

We repeat the above operations until the program terminates.

The trace tree is similar as the code tree, but it is more complicated which shows the binding relationships and jump process. The code tree does not change during execution while the trace tree grows in the process.

Every polygon, edge, ellipse and point in trace has a teacher in code since they are all copied from code tree.

Note: Every polygon in trace has and only has exact one teacher in code diagram. However, polygon in code may be the teacher of two or more polygons, because it may be copied more than once (executed many times). Besides, some instruction polygons in code diagram may be never executed. In this condition, it cannot be the teacher. These rules are also for edges, ellipses and points.

5.4 Requirement specification

In this section, we give the requirement specification of the software.

To make it clear, we can divide it into several parts according to the goals of project. Since the jumping machine uses graphical syntax to present the code and trace, it is necessary that the software should be able to draw shapes and lines. Table 3.1 gives the requirement specification on code program building.

5.1.1	The program must allow the user to build the code diagram.
5.1.2	The program must allow the user to draw polygons on canvas.
5.1.3	The program must allow the user to add instruction text in polygons.
5.1.4	The program could ensure that the instruction should be fully enclosed by the polygon.
5.1.5	The program must allow the user to draw the edges to connect polygons.
5.1.6	The program must allow the user to add tag and identifier on edges.
5.1.7	The program should allow the user to adjust the position of polygons.
5.1.8	The program should allow the user to delete polygons and edges.

Table 5.1: *The requirement specification on building code tree*

Table 5.2 gives the requirement specification on executing the code diagram.

5.2.1	The program must be able to execute the code diagram built by the user.
5.2.2	The program must generate the trace diagram while executing the code.
5.2.3	The program should allow the user to adjust the position of the trace.
5.2.4	The program could be able to show the newest polygon or line added to the trace.
5.2.5	For an object in the trace, the program could be able to find its teacher in the code.
5.2.6	The program should be able to display the content in the stack.

Table 5.2: *The requirement specification on executing code tree*

Besides, to perfect the software, Table 5.3 gives the requirement specification for some extra functions that would be expected by users.

When building a complicated code diagram, we may expect to save it to file, as well as the loading function, so that we can re-use the diagram in future. Besides, when executing the code, sometimes we may expect to trackback to the last status (the undo function), which can be quite useful for analyzing the program executing procedure.

5.3.1	The program must be able to save the code diagram in file.
5.3.2	The program should allow user to choose the file to save the diagram.
5.3.3	If the file does not exist, the program could create such file and save it.
5.3.4	The program must be able to read the saved diagram file and re-draw it.
5.3.5	The program must be able to run the re-drew diagram.
5.3.6	The program should allow user to choose the file and open it.
5.3.7	If the file does not exist or it does not save the diagram, the program should report an error.
5.3.8	The program should implement the undo function when executing the code diagram.

Table 5.3: *The requirement specification of extra functions*

6 Design

We use Java as the language to develop the software, and JavaFX to complete the GUI and the drawing functions.

According to the goals and specifications, we can divide the software into two modules: the code builder and the code executer. It is clear that the former is used to build the code tree while the latter is used to execute the code.

To implement the jumping machine, the software should not only be able to draw the graphical syntax, it should also be able to store the information along with these shapes. Therefore, we define some new objects to present the polygons, edges, points and identifiers, respectively, and these objects and data structures of the software are elaborated in detail in this section.

```
public class Block {

    private int id;
    private int fatherid;
    private String text;
    private String type;
    private Shape shape;
    private Label label;
    private Label idLabel;
    ArrayList<Point> linkpoints;


    public Block(String type, String text){
        this.type=type;
        this.text=text;
        this.linkpoints=new ArrayList<>();
        this.idLabel=new Label();
    }
```

Figure 6.1: *The data structure of object Block*

Figure 6.1 shows the field variables and constructor of the object Block, which is inherited by two subclasses codeBlock and traceBlock. Presenting the instruction polygons in code and trace, respectively. The Block stores the key information of the polygon, including the type of polygon (pentagon or hexagon), instructions, points on the polygon and so on.

The type of polygon is stored as a String, Hgon represents hexagon while Pgon represents pentagon. The instruction is stored as a JavaFX object label so that it can be displayed on the screen, and we can get the content of the instruction by using the method getText(). The points including both link-point and jump-point are stored in the array list points. Usually, the link-point is stored on the index 0 of the list, and we can also get the position of the polygon since it is the leftmost vertex of the polygon.

For the traceBlock, it stores the ID of its teacher as well as a bind table. The former can help it to find its teacher in the code tree, while the latter can make the binding operation more efficient.

```
public class Point {

    private int pointID; //Point id;
    private double x;    //axis of the point
    private double y;
    Circle pt;    //The javaFX object displayed on Canvas

    private int traceID; //To record the polygon ID that the point is placed
    ArrayList<Link>links; //The links started or end on the point are stored in the array list

    public Point(double x, double y, int traceID) {
        this.x = x;
        this.y = y;
        this.traceID = traceID;
        this.links=new ArrayList<>();
        this.pt=new Circle(x,y, radius: 3.0);
    }
```

Figure 6.2: *The data structure of object Point*

Figure 6.2 shows the field variables and constructor of the object Point. Point presents link-point and jump-point in the jumping machine. It stores the axis of the point and the ID of the polygon it belongs to. Besides, the links start or end at the point are stored in the array list links.

To find the teacher of the point on the trace, we first find the teacher of the polygon it belongs to, and the teacher point has the same index in the list points.

```
public class Link {


    private Line line; //The JavaFX object displayed on the Canvas.

    private int traceID1; //start point trace id
    private int traceID2; //end point trace id
    //private boolean settled;
    private Point p1;  //The start point
    private Point p2;  //The end point
    private Label tag; //The tag on the edge.
    private Link correspondLink; // For the link in the trace diagram, record its teacher.

    ArrayList<Binding> bindings; //Store all the identifiers on the edge.

    public Link(Line line, Point p1, Point p2){
        this.line=line;
        this.p1=p1;
        this.p2=p2;
        this.traceID1=p1.getTraceID();
        this.traceID2=p2.getTraceID();
        //this.settled=true;
        this.tag=new Label();
        this.bindings=new ArrayList<>();
    }
```

Figure 6.3: *The data structure of object Link*

Figure 6.3 shows the field variables and constructor of the object Link. Link presents the edge in jumping machine. Its position on the canvas can be got according to the start point and end point, and the polygons connected by the link can be get as well. Besides, the tag of the edge is stored as

a label so that it can be displayed on canvas directly. The identifiers are stored in a array list as a new object binding.

```
public class Binding {


    private Ellipse ellipse;
    private Label bindKey;

    public Binding(Ellipse ellipse,Label label){
        this.ellipse=ellipse;
        this.bindKey=label;
    }
```

Figure 6.4: *The data structure of object Binding*

Figure 6.4 shows the field variables and constructor of the object Binding, which presents the identifier on the edge. It contains a label to store the identifier and a JavaFX object Ellipse.

In the trace diagram, the label of the binding would display both the identifier and its binding value.

Figure 6.5 is the class diagram of the software, which is helpful for analyzing the software architecture. Because of the complexity of the software, the given class diagram is not the complete one, which only lists the main variables and functions In each class. And the classes are introduced in the section.

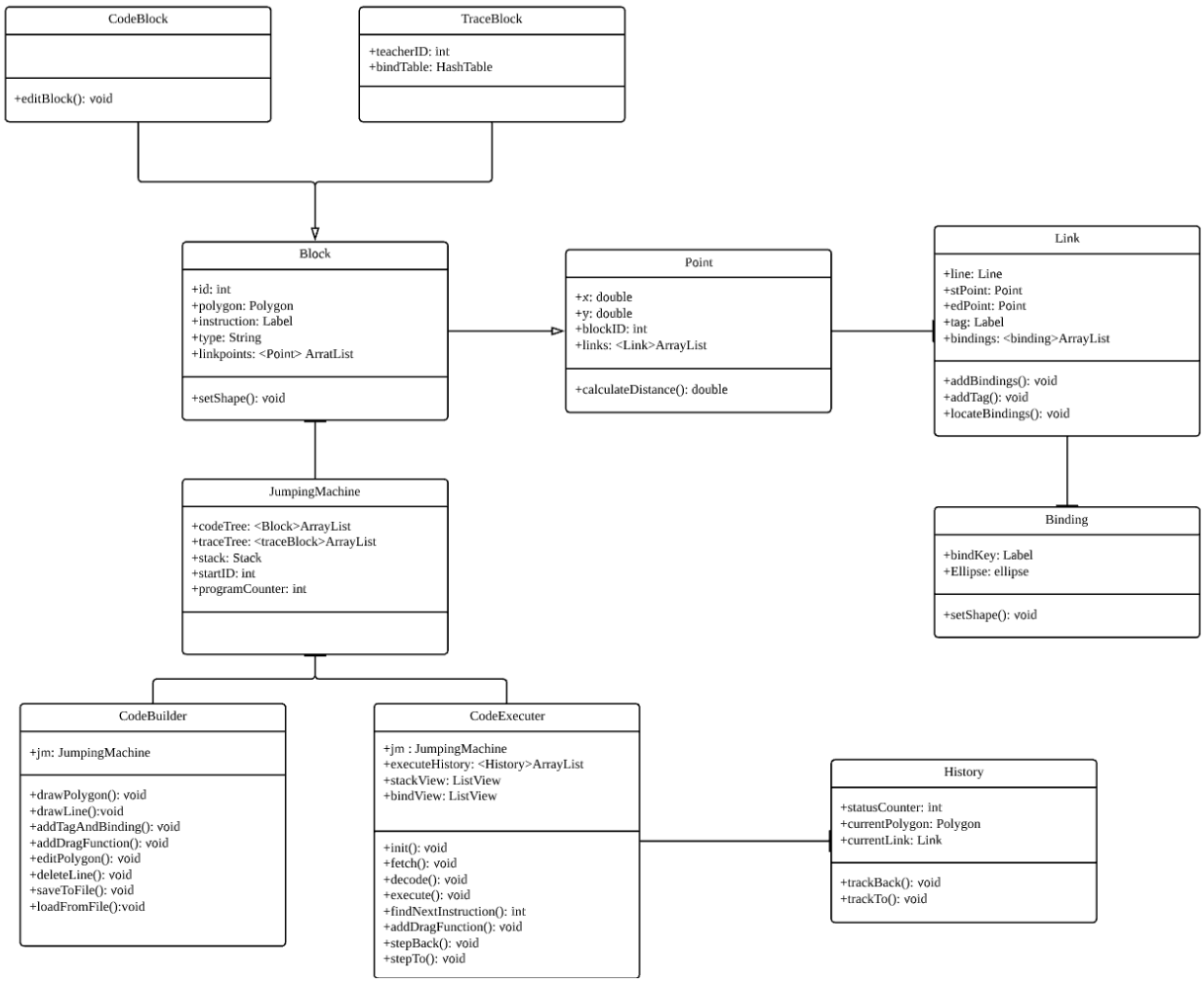

Figure 6.5: *The class diagram of the software*

The class JumpingMachine defines a new object which presents the jumping machine. It owns two array lists codeTree and traceTree, which stores the code diagram and trace diagram, respectively. Besides, it also owns a stack that the frames, tags and identifiers can be stored in it as the type of String.

The class codeBuilder and codeExecuter are the FXML controllers. The former defines the operations in code builder including addBlock (draw the instruction polygons on canvas), addLink, addTag and so on. while the latter defines the operation of executing the code.

The class History defines a new object History which is used to implement the undo function. When we execute the code diagram, the software would generate a new history which stores the information of the current status of trace diagram, and it would be stored in the arraylist executingHistory in the class code executer.

7 Implementation

7.1 Implementation of code builder

Figure 7.1 shows the GUI of the code builder module. It consists of four components, which are introduced as following:

Figure 7.1: *The GUI of code builder*

1 **The instruction menu**

All the instructions involved in the jumping machine is listed in this menu. We can draw the instruction polygon by clicking the corresponding button.

2 **The instruction text field**

The text field is for the user to input the instructions. The sample text in the text field would change when we select an instruction in menu 1). For instance, when we click the button force, the text in the text field would be force x. Though we usually need to change the value or the name of identifiers, it in some extent can help the user to understand the structure of the instruction.

3 **The Operation Menu**

The menu defines some important operations in the software. The Save and Open button is used to save the code diagram to a file and re-draw the code diagram saved in file. When click the Clear button, the canvas would be reset. When click the Run button, the software would

open a new window called code executer to run the code diagram.

4 **Canvas**

The code diagram is built on the canvas.

The basic operations on code builder are also listed as following:

Draw the instruction polygon

To draw the polygon, we will first select the instruction type on the left-top menu and then input the correct instructions. After that, we need press the Enter key, the program will generate the instruction polygon on the canvas, then we can drag it to the right place.

Draw a line to connect two polygons

To draw the edge, we need to determine its start point and end point. In jumping machine, all the edge should start and end on the link-point or jump-point. In this software, it is achieved by double-click the canvas. When we double click the canvas, the program will automatically find and select the closest link-point (or jump-point). If there are two selected point, the program will draw a line between them and reset them as non-selected.

Note: When we double click the canvas and find the closest point, the distance between them will be calculated. If it is more than a threshold value (e.g., 30.0), this double click would be treated as a mistake and cancelled.

Delete a line

The deletion of a line can be simply achieved by right clicking on this line, then it would be deleted along with its tag and identifiers on it.

Edit the instruction on polygon

If we need to modify the instructions on the polygon, we could right click on the polygon, then we can input the correct instruction on the pop-up text field and press Enter, the text on the polygon would be fixed and the size of the polygon would be re-calculated according to the length of the new instruction.

Add tag or identifiers on the edge

We could double click the line, then input the content we want to add on the pop-up text field and press Enter. The content would be regarded as a tag if it starts with , otherwise an identifier. To distinguish, the identifier would be enclosed by ellipse.

Note: According to the jumping semantics, there are at most two identifiers on one line. Therefore, if the line has already two identifiers and we still try to add an identifier, it would be treated as a mistake and cancelled.

Save the code diagram to a file

The software can save the code diagram to a file by recording the information of all the polygons, points, edges, tags and identifiers, including the text, the position and so forth.

To save the diagram, we should click the Save button and the then input the file path and file name in the pop-up text field. If the file does not exist, it would be created. Otherwise, the file would be overwritten by the software.

Open a code diagram file

When we open a code diagram file, the software can read the data and then re-draw the code tree.

We can open the file by clicking the Open button, and then input the file path and file name in the pop-up text field. If the file does not exist, or it is not a code diagram file, the software would pop-up a message box to report an error.

Open the code executer

We can run the code executer by clicking the Run button.

7.2 Implementation of code executer

Figure 7.2 shows the GUI of code executer, like the code builder, it can also be divided into several components.

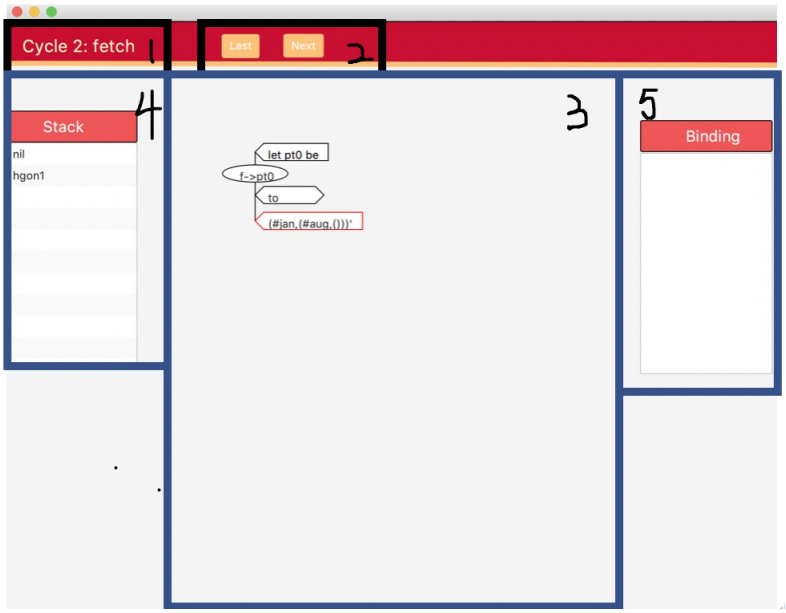

Figure 7.2: *The GUI of code executer*

1 The label shows the status of execution cycle the program is in.

2 There are two buttons for executing the next instruction or track back to last instruction.

3 The software would output the trace diagram on the canvas.

4 The list view is used to display the content of the stack. The stack can store frames, tags and identifiers as the type of String. Since the frame is presented by hexagon in graphical syntax, when we want to push the frame into stack, we push the String hgon i, where i indicates the position of the hexagon in diagram.

5 When we click a polygon, the list view would display all binding values on this branch.

The program execution principle has been introduced in section 3.

The code executer owns an object jumpMachine called jm which stores the code diagram and trace diagram, an arraylist executingHistory to store the status of trace diagram in each execution cycle, a program counter to store the ID of the block we need to execute next in the code diagram and a status counter.

When the code executer is executed, we obtained the jm and startID from code builder. We set the program counter to the value of startID and the status counter to -1. The execution process in code executer is like the step debug in IDE that we can either step to the next status or go back to the last status.

If we step to next status by clicking the Next button, the value of status counter would plus one. Then the software would check if this status has been executed before (If the value of status counter

is smaller than the size of executingHistory). If yes, the code executer can set the trace diagram according to the data stored in the executingHistory.get(statusCounter). Otherwise, the code executer would run the program following the fetch-decode-execute cycle, meanwhile it would create a new object History to store the status of trace diagram, which would be added into the arraylist executingHistory.

If we step back to last status by clicking the Last button. The status must have been stored in executingHistory, so the trace diagram can be set according to the data stored in the executingHistory.get(statusCounter), then the value of the status counter would minus one to point the current status history.

It is true that the trace diagram keeps growing in the execution process. So the object History would store the new generated polygons, lines, tags and identifiers as well as the change in stack on the trace diagram. When the program steps to the last status, we can simply set the visible property of those shapes to false. And when we steps to this status again, we can just reset the visible property to true , which is much more efficient than re-executing the code.

Besides, we give the introduction of other operations and some features in code executer as following:

- **Operation 1. Adjust the trace tree**

  Like the code builder, we can adjust the trace by drag the polygons to proper position to make it looks good. A difference in the code executer is that when we drag a polygon in the trace, the subtree of this node would be moved as well.

- **Operation 2. Click the polygon on the trace**

  From the given Figure 7.3 we can see that when we click the polygon on the trace, this polygon and its teacher would be highlighted in the diagram. Meanwhile, the binding table would list all the binding values on this branch. This function is quite helpful in code analysis.

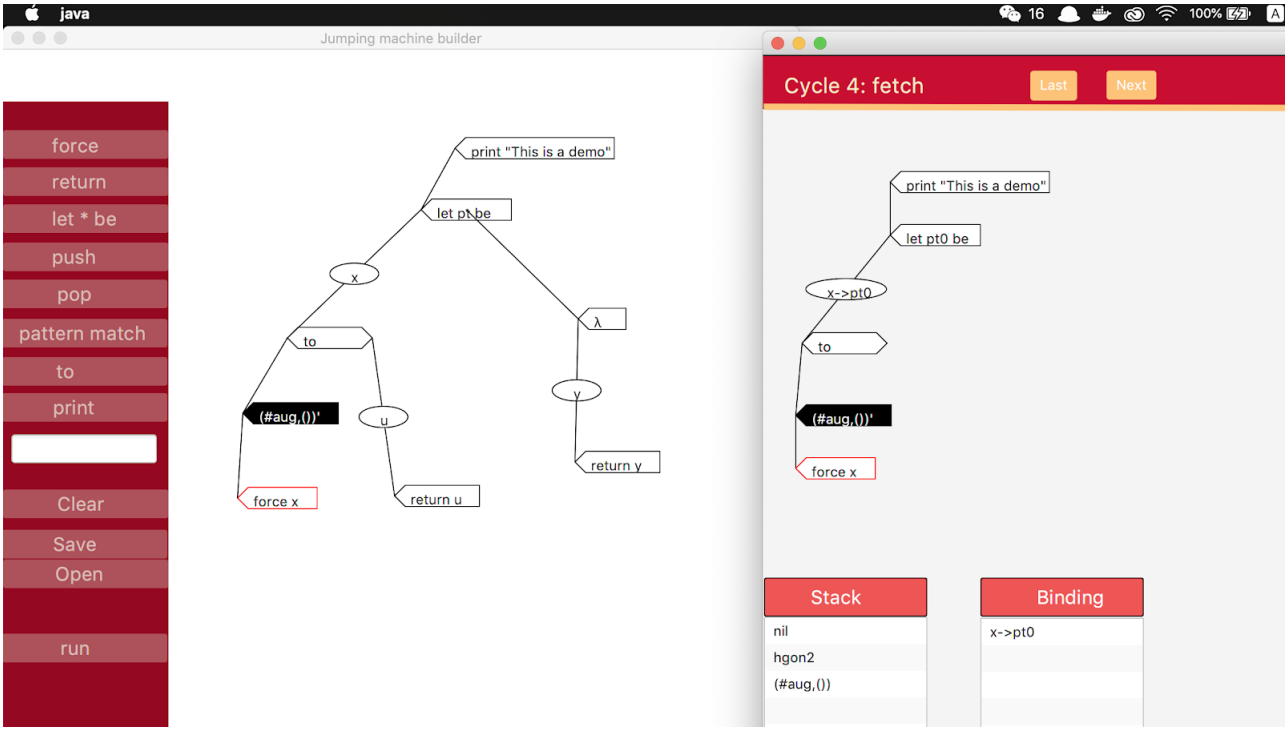

Figure 7.3: *The result of clicking polygon*

7.3 Some features of the software

In section 5.1 and 5.2, we introduced that how we implement the code builder and code executer to fulfill the project goals, and in this section, we would introduce some problems we meet during the process and how do we solve them.

- **Automatically calculate the polygon width**

  In the first prototype, the size of the polygon is fixed, the program set its width as 100.0. However, it cause a problem that the length of the instruction label may exceed the width of polygon, so it cannot be enclosed by the polygon. To solve the problem, we determined to reset the width of polygon when the instruction label is initialized or edited. The formula to calculate the width can be written as:

  W=lx+bias

  Where W is the width of polygon, x is the number of characters in the instruction, and l and bias are two constant numbers which we set as 0.5 and 40.0 in here.

  Besides, this algorithm is also applied for ellipse, we reset the x-radius of ellipse using the similar formula when the identifier is added to an edge or some value is bound on it.

- **Automatically check the start polygon**

  To execute the code diagram, we have to know which instruction should be executed first. The

polygon that contains this instruction is called start polygon.
In the software, the start polygon will be found automatically when we build the code diagram. Actually it is difficult to achieve because there is no difference between the start polygon and the others.

At first we tried to set the first polygon that the user put on canvas as the start polygon, but it is not perfect. One problem is that if we want to expand the code, or change the initial code structure, we have to rebuild the whole diagram.

The code diagram is built as the tree structure. For a standard tree, the root must be on the top of the tree. So we assume the programer would build the code trace in the standard structure,then the problem now becomes how to find the topmost polygon, which is much easier.

Every time when we add a new polygon or we adjust the position of polygons, the software would find the link-point of each polygon, and compare the value of Y of the point with each other to find the topmost polygon.

- **Highlight the current polygon and edge**
  In fetch cycle, we copy a polygon from code diagram to trace diagram. To make it clear, the software set its stroke in red. If this polygon is not the start polygon, we should reset the stroke of the edge from last polygon to black.

  In decode cycle, we decode the instruction and the stroke of the polygon is still in red.

  In execute cycle, the software would executed the instruction on polygon, and then it would draw a new edge which is in red, meanwhile the stroke of polygon would be reset to black.

  As a consequence, we can easily find the current position on code diagram and trace diagram, which can help us to analyze the code.

- **Combined value**
  In the jumping machine, a value can be a tag, an identifier or a combination of two values. To present the combine value, we separate the two values with commas and enclose them in parentheses. Examples of combined value representation is given as following Table.

This software needs to have the ability to handle combined values. According to the jumping semantics, we may push a combined value into stack or return a combined value, in these cases, we need to replace all identifiers to its binding value. Besides, we may use pattern match instruction to decompose a combined vale. In this case, we need to decompose the combined value into two and bound them to identifiers respectively.

Though a combine value can only be combined by two values, these two values may be also combined values, which makes the problem much complicated.

We implemented an algorithm to handle combined value which is based on recursion.

7.4 testing

This section gives the method of testing the software.

It is time-consuming for such a complex software to test each methods separately by JUnit test or GUI test. Instead, we firstly focus on checking the correctness of the algorithm that transforming code diagram into trace diagram, since this is the most important functions of the project, we must ensure it works well.

Firstly, we designed and built 20 code diagrams using code builder, then they are saved in file named from test1 to test20. Then we opened these files and executed on code executer.

During this process, we can also test the functions of drawing polygons and lines, file saving and loading and code executing. The software successfully completed all the tasks, and the trace diagrams it generated are same as those we get by running code diagram on paper. And the test table is given as Table 7.1.

Test case	**Actions**	**Expected result**	**Actual result**
Draw an instruction polygon on Canvas.	Click the corresponding instruction button and then input the instruction and press "Enter".	The instruction will be added on canvas along with the polygon.	The instruction was added on canvas along with the polygon.
Draw an edge to connect two polygons.	Double click the canvas to set start point, then repeat this operation to set end point.	A line will be added on canvas.	A line was added on canvas.
Delete an edge	Right click the edge.	The edge as well as its tag and identifiers will be removed from canvas.	The edge as well as its tag and identifiers were removed from canvas.
Add tags on edge.	Double click an edge and input the tag.	The tag will be placed on the edge and displayed on canvas.	The tag was placed on the edge and displayed on canvas.
Add an identifier on edge.	Double click an edge and input the identifier.	The identifier enclosed by an ellipse is placed on the edge and displayed on canvas.	The identifier enclosed by an ellipse is placed on the edge and displayed on canvas.
Add two identifiers on one edge.	Double click an edge and input the identifier, and then repeat the operation.	The two identifiers enclosed by ellipses are placed on the edge and displayed on canvas.	The two identifiers enclosed by ellipses are placed on the edge and displayed on canvas.
Add three or more identifiers on one edge	Double click an edge and input the identifier for three or more times.	The first two identifiers enclosed by ellipses are placed on the edge and displayed on canvas.	The first two identifiers enclosed by ellipses are placed on the edge and displayed on canvas.
Move a polygon.	Drag the polygon and move it.	The positions of the polygon as well as the points on it are changed.	The positions of the polygon as well as the points on it were changed.

Test case	**Actions**	**Expected result**	**Actual result**
Modify the instruction on a polygon.	Right click the polygon and input the new instruction.	The instruction on the polygon is modified and the size of the polygon is re-calculated.	The instruction on the polygon was modified and the size of the polygon was re-calculated.
Save the code diagram into a existed file.	Click the Save button and input the file name.	The file would be overwritten by the software.	The file was overwritten by the software.
Save the code diagram into a file that does not exist.	Click the Save button and input the file name.	The software would create a new file and store the data of trace diagram.	The software created a new file and stored the data of trace diagram.
Open a existed trace diagram file.	Click the Open button and input the file name.	The software would read the data in the file and re-draw the trace diagram on canvas.	The software read the data in the file and re-draw the trace diagram on canvas.
Try to open a file that does not exist.	Click the Open button and input the file name.	The software would pop-up a message box.	The software pop-up a message box.
Reset the canvas.	Click the Clear button.	All the elements on canvas would be removed.	All the elements on canvas were removed.
Open the code executer.	Click the Run button.	The code executer would be opened in a new window.	The code executer was opened in a new window.

Table 7.1: *The Test cases*

The table lists all the test cases and the test methods, as well as the expected result and actual result. According to the table, we can see that the software had passed all test cases, which can be considered as a basis of evaluation.

Figure 7.4 and 7.5 shows an example of code diagram and trace diagram obtained by the software, and the original diagram is built on paper given by [1] as an exercise. In Appendix C, we also give the detailed running-on-paper process of the diagram.

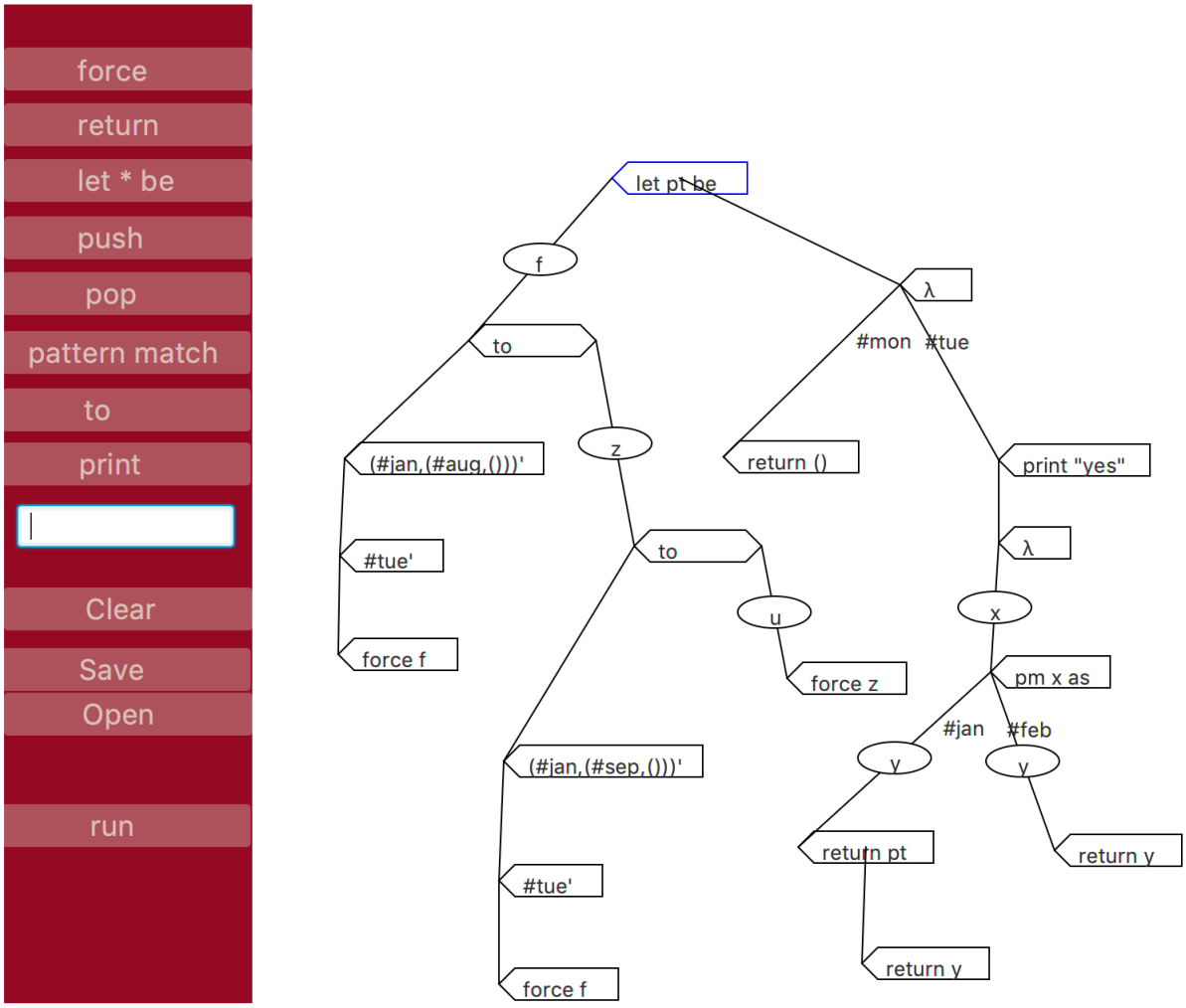

Figure 7.4: *The code tree built by the software*

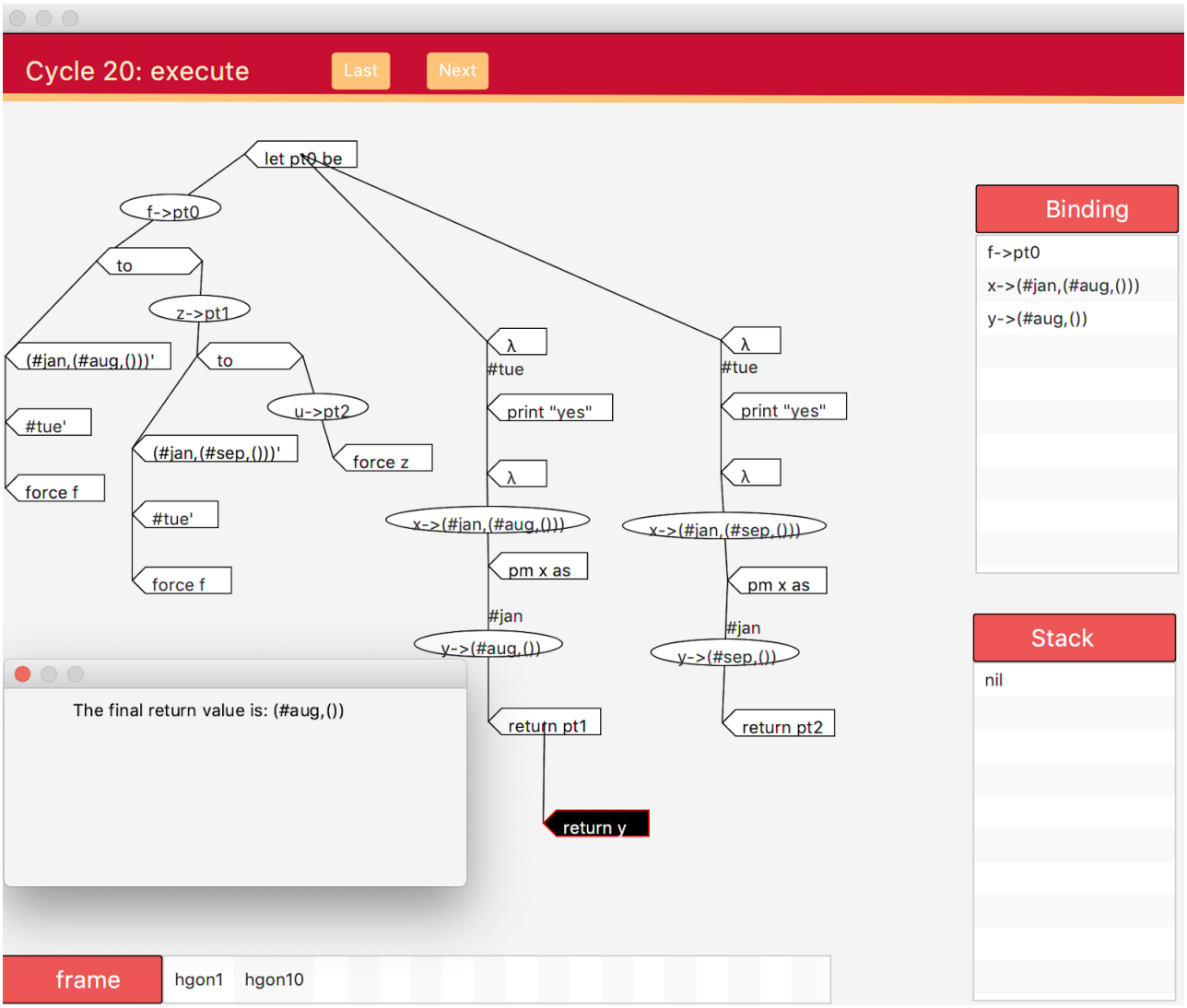

Figure 7.5: *The trace tree generated by the software*

8 Evaluation

This section gives the criteria to evaluate the software as well as the result of evaluation.

We can evaluate the software from four aspects listed as following, and for each aspect, we rate the performance of the software in 5 levels, representing for outstanding, good, satisfactory, poor and unsatisfactory. Finally, we can get a total score of the software as an evaluation result.

Target completion **Outstanding (5/5)**

The software has an excellent performance on target completion, since it successfully complete all goals and requirements. Though we has only 20 pre-built code diagrams, it can at least prove that there is no serious error in the logic of the algorithm. In future, we will re-test it by building more code diagrams.

Useability **Satisfactory (3/5)**

In terms of the usability, we consider that whether the software is easy to understand and whether it is easy to operate.

For the first question, the answer is No since the software does not give enough instructions on how to use it. There are many implicit operations in the software, e.g. In code builder we can draw a line by double-clicking the screen twice.

But on the other hand, it simplifies the operations of the software. For a proficient user, he or she can save considerable time because of this design. Besides, for the purpose of simplifying operations, all the buttons for Confirm is replaced by the operation of pressing Enter.

We give the first generation prototype in Appendix B for comparison.

GUI **Satisfactory (3/5)**

Overall the GUI of this software is satisfactory, though it can be improved in many aspects.

We tried to make the GUI simple and clear, so that the user can easily find the key information displayed on the screen.

However, because of the time limitation, we do not have enough time to adjust it and make it looks good. We may find many inconsistencies in the GUI. For example, the theme color of code executer is slightly different from that in code builder, and the message box just simply displays the text,

regardless of layout, aesthetics, etc.

Software security and stability **Poor (2/5)**

For the software security, we did not take measures on this field. Because we think it is unnecessary to spend much time and resource in improving the performance on security of a pedagogical software.

For the software stability, we use the Java exception machanism to avoid the program crash. When we take unsafe operations like file input and output, the software would throw an exception if it cannot be normally executed.

However, for the code builder, there is no error detection. The software would not give a feedback to the user when they build a incorrect diagram that cannot be executed or input wrong instructions. The mistake may be found only on when the diagram is executed by the code executer, and it may lead to unexpected termination of the execution.

9 Conclusion

The project aims to implement a jumping abstract machine on computer. A brief introduction of related topic is given in this paper, along with the requirement specification, the software architecture ,the class diagram and the implementation details. Besides, the criteria is given for evaluating the performance of the software. Overall the project has successfully achieved all the goals and almost requirements listed in the specification, though there are still many things that can be improved. Therefore, we conclude the limitations of the software and the further work which are given as following:

9.1 Limitation

- **Operability**

  In the code execution phase, one problem is that when the code executer generated a polygon on canvas, it may cover the diagram. So the users have to adjust its position by themselves, which is unfriendly to the users.

- **Useability**

  The usage of the software is not easy to understand. One problem is that there is few instructions given to guide the user. Besides, the user who are unfamiliar with the graphical syntax may also feel confused in using the software.

  We consider adding a guidance module on the software, which may be helpful for new users to understand the jumping semantics and be familiar with this software.

- **GUI**

  The GUI of this software still has much room for improvement, such as the defects mentioned in the evaluation section.

- **Stability**

  There are no error detection for users when they build the code diagram, the error may be not found until the execution cycle which may cause the unexpected termination.

- **Code specification**

  In the design phase of the project, we are not well planning the development process and code

specification. Although the software has finally achieved its desired goals, it still has a lot of deficiencies at the code level that need to be optimized.

First of all, there are few comments in the code, which makes it difficult to be read and understood. Another problem is the software architecture, which is too simple and not well-organized. We defines too many methods in the `CodeBuilder` and `CodeExecuter`, which makes it hard for maintenance. Besides, there are some methods,especially those for drawing on canvas is repeatly defined in the two classes.

9.2 Further work

For the limitations listed above, we already have a plan for most of them, but some we still cannot come up with a solution yet, e.g. it is really challenging to make the code executer draw the polygon in a proper place automatically without manual operation.

And in this section, the foremost tasks we need to do in future have been given as following:

- We consider re-designing the GUI to make it beautiful and have a clear structure. A well-designed GUI can also promote the work efficiency of users.

- We consider implementing the error detection mechanism on code builder, the software would monitor the input of user and judge if it is legal. If it is illegal, the software would set the disable property of "Run" button as true to forbid the users execute the code.

- We consider making the code much more standized by taking following measures:

  - We consider adding comments for the code, as it is challenging for programmers to understand code without comment.

  - We consider to reconstruct the architecture of the software because it was unorganized. We will define a new interface `GUIController` implemented by codeBuilder and codeExecuter, which defines the common GUI interaction methods. Besides, we will also define the helper methods in different manager classes according to their functions, the new software architecture is given in Figure 9.1.

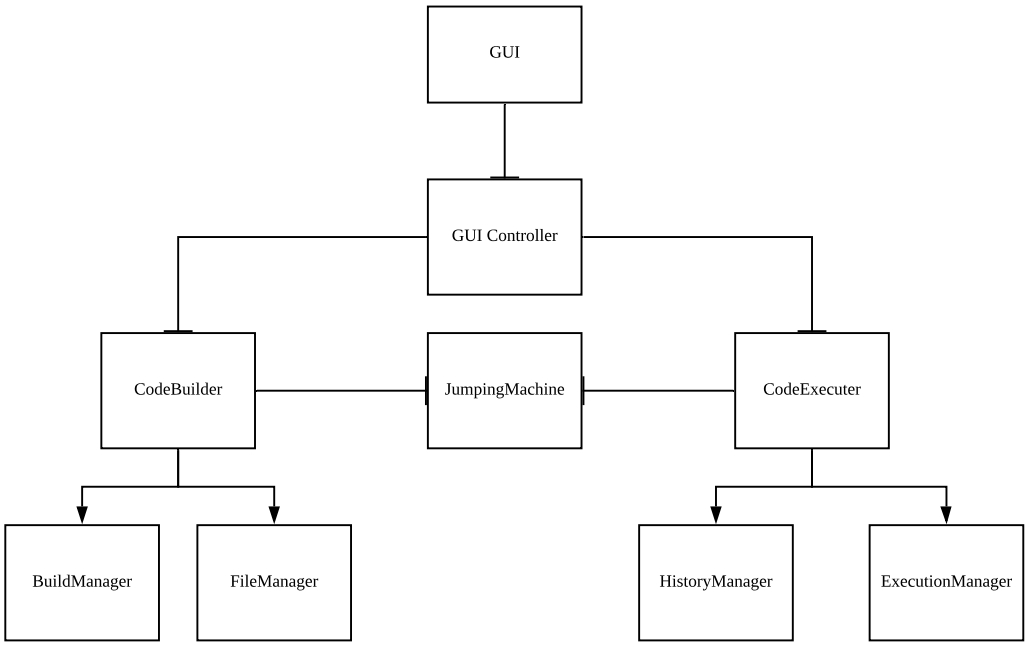

Figure 9.1: *The ideal architecture of the software*

For the new architecture, the data we need, including the code diagram and trace diagram is stored in `JumpingMachine`. And all the methods is defined in the corresponding managers: For the code builder, `FileManager` controls the file saving and loading, while `BuildManager` controls the building processes.

For the code executer, when running a program, it the state has been executed before, the `HistoryManager` would read the saved History and reset the data, otherwise, the `ExecutionManager` would execute it.

Finally, for myself, I would like to spend some time on relative research. I gain some new knowledge about this field which seems interesting. Learn more about abstract machines and give me a deeper understanding of this project and also the programming language.

References

(n.d.), Available:
`https://www.zhihu.com/topic/19566470/hot`. Online; accessed 04 September 2019.

Diehl, S., Hartel, P. & Sestoft, P. (2000), 'Abstract machines for programming language implementation', *Future Generation Computer Systems* **16**, 739–751.

Felleisen, M. & Friedman, D. (1986), 'Control operators, the secd-machine,and the -calculus', *Formal Description of Prog* .

Levy, P. B. (1999), 'Call-by-push-value: A subsuming paradigm (extended abstract)', *LNCS* **1581**, 228–242.

Levy, P. B. (2005), 'Jumping semantics for call-by-push-value', *University of Birmingham technical report CSR-06-10* pp. 27–40.

Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley (2015), 'Java Virtual Machine Specification', Available:
`https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-1.html`. Online; accessed 04 September 2019.

Turing, A. M. (1936), 'ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM'.

10 Appendix A: Submission

10.1 Git repository

All code of this project can be found at following git repository:

Github: https://github.com/Phob1a/Jumping-machine

Gitlab: https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2018/lxd901

10.2 File structure

```
/
├── docs
│   ├── report -- contains a digital copy of this report, and the LaTeX source
│   └── proposal -- contains a copy of the Project Proposal
├── src -- contains the Java code source for this project                    .
│   └── GUI -- contains the FXML file for GUI implementation
├── test -- A folder contains all of the test code diagram files
└── README.md -- contains instructions on using the code, which will be completed
                soon
```

10.3 How to run the software

After downloading and compiling the code of the project, We can start the software by running Main.java.

11 Appendix B: 1st generation prototype

In the appendix, we give the first generation prototype of the software. It is clear that the design of the first generation prototype is immature.

Though the prototype can fulfill the basic goals of the project, the GUI looks bad, which is given as Figure B.1 and B.2. In the design phase, we just tried to make the software works and considered less on the beauty of GUI.

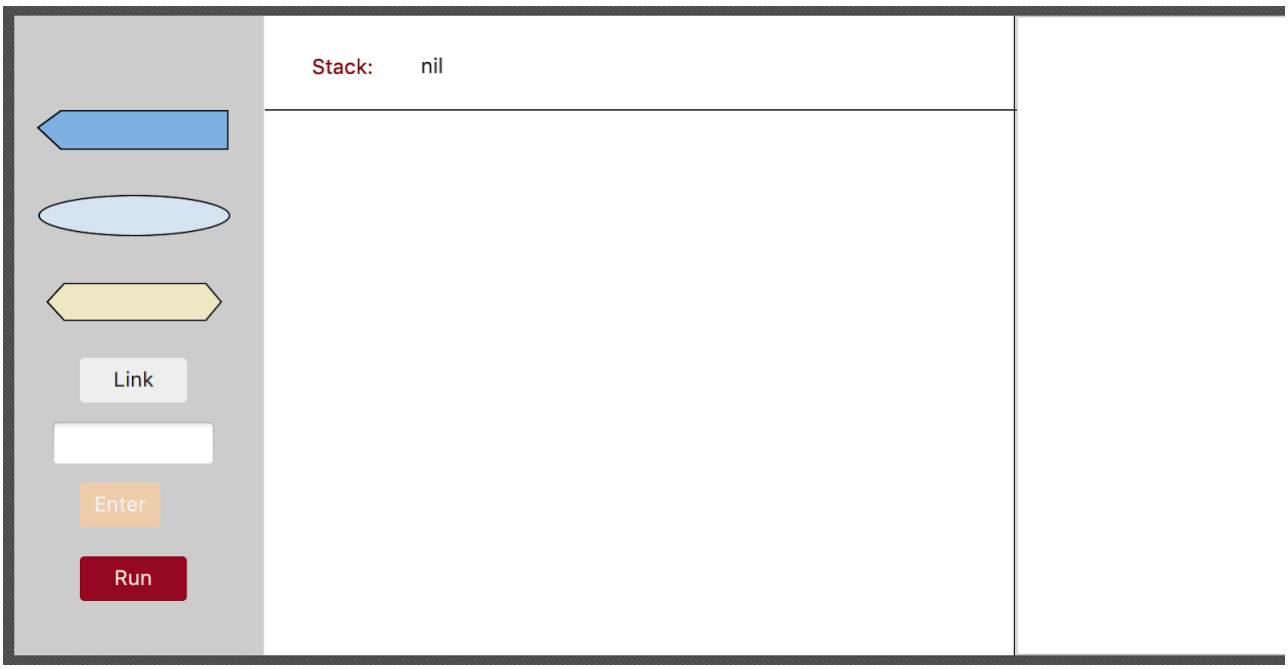

Figure B.1: *The GUI of code builder in 1st generation prototype*

Figure B.2: *The GUI of code executer in 1st generation prototype*

There are a number of buttons on the GUI, and almost operation relies on them. One of the most serious problem is that it is unfriendly for users because of the complex operations. For instance, if we want to draw an instruction polygon on canvas, firstly we need to click the corresponding shapes and then click the text field to input the instructions. After that we have to click the Enter button, and finally click the canvas to draw the polygon. It involves too many mouse operations in a single task.

Another problem is that it allows the user to draw ellipse directly. However, the ellipse is used to present identifiers, which can only be placed on edges. It increased the difficulty on building phase, and it may also confuse someone who are proficient on the graphical syntax.

It is fixed in the final version of the software. In the software, we can only build polygons directly, and the identifier can only be added on edges.

Besides, there are also many little drawbacks, for example, the content of the stack on the screen is unclear and easy to be ignored.

Compared with the prototype, there are a lot of improvement as we gained much experience on building the prototype.

12 Appendix C: A running-on-paper example

We have written down the trace diagram on paper , which shows the step-by-step execution process in detail. From this example, we can also know how easy to run a program on paper using jumping semantics, as well as how it embodies the jumping process.
Finally, we got the same result as the one generated by software.