Integrating Visualization Software into Learning Objects

Jens Peter Träff

Kongens Lyngby 2011
IMM-B.Sc.-2011-10

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

In this project we will develop a framework for integrating visualization into learning objects, such that animations and explanatory text can be shown simultaneously.

We will in this project focus on Jeliot, a system that visualizes and animates Java programs. Our primary goal is to find out if it is feasible to do this kind of integration.

First we will describe what is understood by a learning object, why a framework would be beneficial, which features are crucial and how we will achieve them.

Then we will present the model of the program, making up the core of the framework. This is the part that handles all manipulation of the actual learning object. We will dicuss various choices made to create the best possible program.

We will then proceed to present and discuss the user interaction. Which will be how he/she will experience a learning object designed for this system.

A complete user guide can be found in the appendix.

Finally we will discuss how much work goes into creating learning objects for this framework, how the framework will be distributed and how we can improve the framework in the future. This will include minor improvements, and integration of different concepts.

Resumé

I dette projekt vil vi udvikle et system for at integrere visualiseringer i læringsobjekter, således at animationer og forklarende tekst kan blive vist samtidigt. Vi vil her fokusere på JELIOT, et system der visualisere og animere JAVA programmer. Vores primære mål er at finde ud af om det kan betale sig at lave denne form for integration.
Først vil vi beskrive hvad der forstås med et læringsobjekt, hvorfor sådan et system ville være gavnligt, hvilke egenskaber der er grundlæggende for systemet og hvordan vi vil opnå dem. Derefter vil vi præsentere modellen i vores program, der er kernen i systemet. Det er denne del der håndtere alt manipulation med det aktuelle læringsobjekt. Vi vil diskutere diverse valg der er blevet truffet for at skabe det bedst mulige program. Vi vil da gå videre til at præsentere og diskutere brugerens interaktion. Hvilket vil være hvordan han vil opleve et læringsobjekt designet for dette system. En komplet brugervejledning kan blive fundet i appendikset.
Til sidst vil vi diskutere hvor meget arbejde der skal lægges i at skabe læringsobjekter til dette system, hvordan systemet vil blive viderebragt og hvordan vi kan forbedre systemet i fremtiden. Deriblandt mindre forbedringer og integrering af andre koncepter.

Preface

This thesis was prepared at DTU Informatics, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the B.Sc. degree in engineering. The project has been done from 1/2 2011 - 27/6 2011 and is worth 15 ECTS points.
Part of this project was done in the United States at the University of Connecticut.

The thesis deals with integration of visualization software and textual explanation into Learning Objects. The main focus is to investigate if such a framework can indeed by created and to make it more feasible for instructors to use visualization software in learning objects. In this thesis we will develop such a framework integrated with the Jeliot Animation Software. The project was done under supervision from Professor Jørgen Villadsen, Technical University of Denmark.
Co-Supervisor Professor Mordechai Ben-Ari, Weizmann Institue of Science, Israel.
Author of JELIOT, Niko Myller, has kindly modified his program to comply with the Visualization interface.

Lyngby, June 2011
Jens Peter Träff

Acknowledgements

I will like to thank my advisor Jørgen Villadsen, who has kindly provided support on how to meet the requirements of DTU and set me up with this great group of people who work with Jeliot and Learning Objects.

I will direct a speciel thanks to my Co-advisor Professor Mordechai Ben-Ari, who has been a great support in developing this framework.

Author of Jeliot, Nico Myller has been very kind to adapt Jeliot to the Visualization interface. Without that, this project would not have been possible.

Finally a thanks to my friend John Larsen, who has provided user feedback and continuously questioned the report.

Contents

6

Chapter 1

Introduction

In this paper I will present a framework for developing learning objects integrated with visualization software. I will focus on working with Jeliot, a Java animation software developed by Nico Myller[1]. Before starting, I'll like to define what is understood by a Learning object

Learning Objects

"Learning objects can be used for quick instruction and/or incorporation into an online education curriculum. For the purposes of this site we define learning objects as digital, re-usable pieces of content that can be used to accomplish a learning objective. That means that a learning object could be a text document, a movie, a mp3, a picture or maybe even a website. The key is to describe why something is a learning object and in what context a person might learn something from it. " definition found at[2].

For our purpose we will consider a learning object to consist of text and some Java code. Since Jeliot is used to learn introductory programming, The learning objects created in this framework focuses on this aspect. Although we will

[1] Eastern University of Finland

[2] http://www.learning-objects.net/index.php

design it such that it is as generic as possible.
Learning Object will at times be appreviated by LO.

1.1 What can be achieved by a framework

Learning objects today are mostly a loose collection of information from different sources, i.e text, stepwise explanations and some sort of animation.
This can be relatively confusing for students trying to learn new information. As a consequence, LO's are hardly the selfcontained, easy-to-use, modules that students can use as a complimentary source of knowledge/learning.
It has been argued in [5] that visualization software enhances learning of a complicated topic like a new programming language. Learning Objects using visualizations have been created to take advantage of this.
They, however suffer from lack of integration, i.e. text and animations are not shown in the same window nor are they necessarily *consistent*. Furthermore instructors have to spend a considerable amount of time creating the learning objects with JELIOT as discussed in [4]:

By developing a framework for design of LOs integrated with visualization software, we can achieve some of the traits mentioned in the Learning Object section and mend some of the issues raised in previous paragraph.

- We can display text and animation simultaneously in the same window. Removing the need to cycle through various windows open at the same time.
- We can make it possible to display text and animation interleaved, thus making it *consistent*.
- We can add stepwise descriptions. Linking hard-to-understand parts to explanatory text, thus further add learning.
- Integrated LOs created by this framework will make it possible for students to learn at their own pace, playing around with the text, animations and explanations. Thus reinforce the learning from lectures.
- The amount of work the instructor has to put into a LO can be reduced to just the actual information the students need. This should make using LOs attractive to more instructors, and should greatly increase the use by those already using the concept.

Before defining key features of such a framework, we will take a look at the visualization software.

1.2 Jeliot and visualization software

Jeliot is a animation system of programs in Java. It takes a program in Java and *automatically* generates a detailed animation of the execution of the program.
It animates each step in the code and keeps track of variables, methods and earlier calls to methods.
This is excellent for assisting novice programmers in learning the concepts of Java. It contains methods for controlling the animation. The user have the ability to start, stop, restart and stepwise progress the animation. He can load earlier code into the program too.
Those functions describe what can reasonably be expected as user control of a visualization/animation software.

For more on Jeliot see [1] and [4]. Jeliot has an online website at [2].

1.2.1 Visualization interface

Visualization software commonly provides the user with some basic functionality. This can be the ability to:

- play the animation
- stop the animation
- go stepwise forward in the animation
- restart the animation
- load something to be visualized

This suggests that we can come up with an interface Visualization that specifies those features. Thus if an animation software implements the interface, we should be able to work with it in our design of integrated learning objects. the Visualization interface has been designed by prof. Mordechai Ben-Ari.

1.3 Plan for the report

In the next chapter we will define keyfeatures, that we will strive to implement. Following that we will deal with the design and implementation of the model part of the framework, Which contains the main methods for manipulating the LO.
Chapter 4 describes the development of the GUI and the whole program. This part describes the control mechanisms and tests the functionality of the whole framework.
Chapter 5 contains a short evaluation and some suggestions for future work, that can be done on this framework. It is followed by a conclusion.
In Appendix A the userguide for the framework can be found.
In Appendix B the actual source code can be found.
Both the whole framework and the main program in the framework will be named LOjel. It should be clear from the context which one are ment.

Before we move on to the requirements of the framework, we will define a few terms:

Animation Step:

The animation in JELIOT occur in steps. Each animation step refer to one step in JELIOT. When a step is said to be animating, it means that the actual animation of the given step is being conducted.

Animation:

When talking about the animation, we refer to the animation of the whole LO. That is of all the JAVA-code contained in the LO.
Throughout the report, animation might be used as a reference to the text of the LO interleaved with the animation. It should be clear from the context what is meant. When the animation is said to be running, it refers to the whole animation being automatically progressed step by step.

Step Description:

We distinguish between an animation step in JELIOT and the stepwise descriptions prepared by the instructor. We will call each description prepared by an instructor a step description.

Chapter 2

Overall requirements

In this chapter we will analyse the overall requirements for a framework for creating and displaying learning objects integrated with visualization software. There are two kind of users, the students and the instructor. The students will use the learning objects and are thus the main users of the program. The instructor are the creator of the LO.
We will first look at it from the student's point of view and then from the instructors side. Finally we will present the overall design goals.

2.1 Features of this framework

In this section we take a look at desirable key features in our framework:

2.1.1 Student point of view

From the students point of view, it would be important that it is easy to use, is beneficial for learning and stable. Some features that can help accomplish this is:

The ability to control the animation, that is, play, stop, step, reset, step back
In this way the student gets absolute control of the pace of the animation and can go stepwise forward at the tricky part of the subject.

Good synchronization between text and animation
This ensures that the student get the information in a simple way, and that the stepwise explanation and animation together can help explain difficult concepts in the LO.

Easily accessible explanatory text and a stepwise description
Preferably in the same window, this removes the need to cycle through different windows and should give the student more focus on the actual content.

Easy to navigate in
In order to focus maximally on the learning of a new subject, it has to be easy to operate. It should be easy to access a tutorial and help menues as well.

2.1.2 Instructor's point of view

Overall product should be instructive and beneficial for student learning

Design process should be simple
This will help convince teachers/instructors that LO's is an asset worth using in teaching. It enables the instructor to focus on the actual content that he want the student to learn.

Ability to link comments to specific steps of the animation
Enables the instructor to customize which steps he wants to add speciel information for.

Good customizability regarding explanation and checks
Allows the instructor to emphasize the aspects he wants to focus on, and at the same time gives the opportunity to do a simple evaluation of student understanding.

Should be easy to distribute to students
Otherwise to much time will spend on learning to master LOjel instead of the actual content, which might lead to students/instructors choosing not to spend time on LOs.

Be able to check the final LO pretty easy
Should ease the development process, as the instructor is quickly able to determine if a certain element accomplishes his idea.

2.2 Overall design goals

To address the features we have described in the previous section, we will rely on the following components:

- Use a classic Model-View-Control approach
- Use a system based on the JAVA class JFileChooser for opening the contents of a LO
- Integrate JELIOT/the animation software such that size etc. can be manipulated
- Try to utilize design that emphasizes learning and assists in the learning process
- Reduce the instructors workload to writing a few files and some synchronization measures

By using a filechooser loading systems, we make it easy to choose between various examples and allows us to keep one copy of our framework opened, and access the different LOs from there.
Loading the animation software into a component, allows us to resize and customize the display to fit the student/instructor.
By making the animation software comply with Visualization interface, we are able to control the pace of the animations and interleave it with text.
We will try to make some simple measures that should help reduce the cognitive load. Such as making text and animation visible in the same window at the same time, so minimal amount of scrolling will be needed.
By using a file based approach to construct the LOs, the instructor only has to focus on writing those files and can easily make changes in his LO. Since the instructor is the only one who knows how many animation steps each of his explanatory steps corresponds to, he will have to specify this.

In the next chapter we will move on to the actual design of the framework.

Chapter 3

The Model

In this chapter we will take a look at the program that will form our framework.

Overview of the framework

The framework consists of a program, the animation software, (in our case Jeliot), and the files that provide the actual content to the LOs.
The program, which we call LOjel, follows a model-view-control approach. We will in the following describe the model.

The model

The model consists of the Model class, the Visualization interface and a class, JeliotLOVisualization that makes Jeliot comply with the requirements from the Visualization interface.
The Model class is responsible for performing the actual computations and executing the various methods when called from the control part.

3.1 Design

The main problems we need to handle in the model are the following:

- Load text for explanations, checks and answers
- Synchronize stepwise explanation with animation as specified by thefile
- Advance the state of the LO by one step
- Allow the LO to progress automatically, Play
- Open a screen for file selection and choose an LO
- Rewind state back to start
- Be able to load a new LO, when one is already loaded
- Keep the methods generic whenever possible

Load text

The text should be loaded by using standard open functions. We will design the functions such that LOs will require 4 different file format, and such that the LO can be opened by clicking on any file with the LOs name before the extension. We will use 4 files to increase customizeability and to make it easy to craft each file. An overview of the fileformats can be found in table 3.1

Filecontent	**Extension**	**Format**
JAVA code	`java`	The JAVA program to be animated in JELIOT.
Explanatory text	`exp`	The background explanation text as it should appear.
Question	`chk`	The text of the question, followed by a blank line, followed by the word `answer` and answer on a new line.
Step descriptions	`stp`	The first line contains the total number of steps; for each step, the following format is used: `step` [step no.]: [number of animation steps][text]

Table 3.1: LO file formats

Synchronization and display of text and stepwise explanations

First we have to decide on the internal representation of the active Learning Object. We use a single counter to keep track of what step is currently animated. The value of this counter will denote the *state* of the current LO. When we design methods for manipulating the state of the LO, we do it by updating the state, display corresponding text, call appropriate action in the visualization software.
Text from previous steps, and the explanatory text should still be visible.
This approach ensures that the stepwise explanations and animations will appear when and as specified by the instructor.

Forward one step

This is done by updating the program state, displaying the new text and then calling the animation software to animate the next step. The text should be displayed before the anim. software is called to ensure the text is viewable when the animation is occuring.

Forward automatically

This function should be done by utilising the one-step-forward function, the program sits in a loop and keep advancing the state of the LO. After each step, we will pause a short time to give the animation, time to finish. This continues until either the animation is halted or the animation is completed.
Choosing not to make use of the play-method guarenteed by the Visualization interface provides more flexibility, as we can control the pace, stop the animation after each completed step and add various functionality between steps.

LO opening

To make this user friendly, a filechooser dialogue should open on demand, and then the user only has to click on one file, with the base filename of the learning object of interest. Then the new LO should load into the program.

Rewind state

This should be handled by resetting all animation parameters, and rewinding the state to 0. Then displaying the text corresponding to the state.

Loading a new LO

This will be done by resetting all parameters used in the current LO, and then executing the load methods.

3.2 Implementation

The model consists of the Visualization interface, JeliotLOVisualization, a class that adapts the visualization software to the interface and the Model class, a class specified in this program. First we will show an overview of the model and then we will proceed by describing the key methods in Model below (except for the constructor and initializeVisualization in JeliotLOVisualization, all methods are accessed via methods in the Model class):

Model	
Type:	**Name of field:**
String	description
String[]	stepsDescription
int[]	JeliotSteps
int	currentstep
int	actualJeliotStep
boolean	run
LO_frame	LOframe
String	baseFileName
Return type:	**Name of method**
void	load_Text(
void	load_step_explanation
void	open
void	loadLO
void	forward_animation_one_step
void	play_animation
void	update_labels
void	stop_animation
void	totalRewind
void	resetAll
void	resetAnimation

Visualization	
no fields	
void	load
JComponent	initializeVisualization
void	runFromStart
void	step
void	reset
void	stop

JeliotLOVisualization	
no fields	
JComponent	initializeVisualization
void	load
void	reset
void	step

Table 3.2: Overview of the model

`initializeVisualization`

This method is responsible for loading the animation software into a JComponent.

`open`

Creates a JFileChooser, and extracts the basename of the file (name of file withouth extensions) selected by the user.

`loadLO`

Opens the selected Learning Object. This is done by applying the appropriate file extensions to the previously extracted basename and then making use of the two load methods specified below.

`load_Text`

This function loads the content of the .exp file, which contain the explanatory text provided by the instructor. The text is stored in a single String variable.

`load_step_explanation`

Loads the content of the .stp file, which contain the stepwise explanation prepared by the instructor. It links step descriptions to animation steps, this enables us to jump around in the program state, and still maintain synchronization.
We use the two arrays jeliotSteps and stepsDescription to keep track of those factors. The size of the arrays are equal to the number of step descriptions provided by the instructor, every entry in the arrays corresponds to a step description. I.e. stepsDescription[0] holds the first step description and jeliotSteps[0] holds how many animation steps this step description should be displayed for.
When loading the information we read the file line by line. This requires the instructor to start every new step description by a new line starting with ”step” and a ”:” everything after the colon will be displayed. The instructor can write anything he wants after ”step” and before ”:”, like ”step 1:”. The way we load allows the instructor to use more than one line for each step description.

`forward_animation_one_step`

This function is responsible for performing the actions that will allow the animation and text to move one step forward. This is done by utilizing the programs step counter, currentstep, by simply incrementing it. To advance the animation the step method from the visualisation interface is used.

`play_animation`

This function is responsible for starting the simulation and make it run on its own. The program has a boolean field, run, determining whether play is active or not. The method sits in a loop, executing the forward_animation_one_step method and then waiting a brief amount of time to give the animation step time to finish before moving on to the next step. In this version a 3 second wait is used. It sits in the loop until either the animation is stopped, or the whole Java program has been animated.

`stop_animation`

This method halts the running of the animation when called.
this is done by setting `run` to false.

`restartAnimation`

This function is responsible for rewinding the animation, such that it is ready to start from the beginning.
We achieve this by resetting the programs animation parameters (including `run`), and calling the reset method from the visualization interface.

`resetAll`

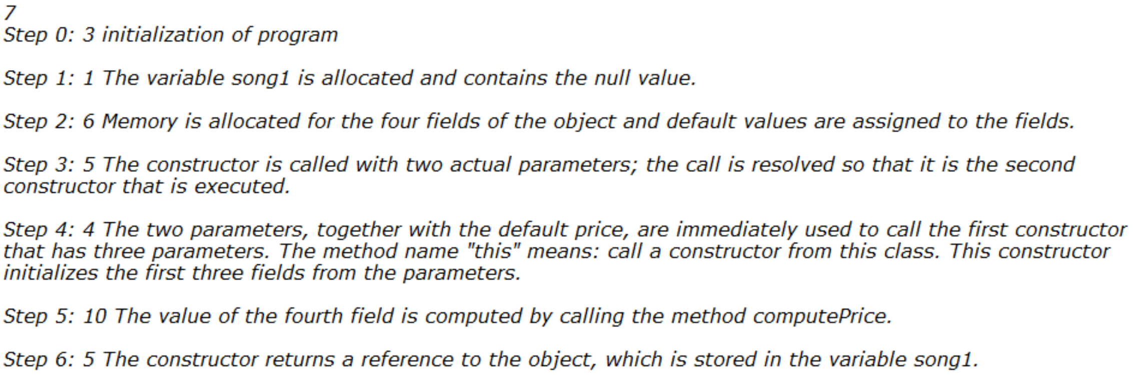
This method resets the whole program, so that a new LO can be loaded into the display.
This is achieved by emptying all the arrays, disallocating them, and then resetting all the necessary parameters.

`update_labels`

Updates the current step the program is in, `run`, and then calls for the view-part to show the corresponding text.

3.3 Tests

In this section we have tests of the primary methods in the model. When testing the play/step methods we primarily test the text updates, the animations will be tested in the GUI section.
The tests are carried out using the Constructor LO example. Where nothing is mentioned the tests went as expected.
Figure 3.1 shows the constructor.stp file that defines the step descriptions to be displayed, and how many steps corresponds to each step description:



7
Step 0: 3 initialization of program

Step 1: 1 The variable song1 is allocated and contains the null value.

Step 2: 6 Memory is allocated for the four fields of the object and default values are assigned to the fields.

Step 3: 5 The constructor is called with two actual parameters; the call is resolved so that it is the second constructor that is executed.

Step 4: 4 The two parameters, together with the default price, are immediately used to call the first constructor that has three parameters. The method name "this" means: call a constructor from this class. This constructor initializes the first three fields from the parameters.

Step 5: 10 The value of the fourth field is computed by calling the method computePrice.

Step 6: 5 The constructor returns a reference to the object, which is stored in the variable song1.

Figure 3.1: Screenshot of the constructor44.stp file

First we, in a table, list the chosen test cases and their properties, then in a second table we list input and output for those test cases.

Test of the Load step method

We will in this method test how bad format will influence the LO when loaded.

load_step_explanation		
Case	Properties	Explanation
A	.stp file follows the correct format	the arrays should be loaded as expectet
B	one entry lacks number of animation steps	What happens if one crucial information is missing
C	one entry lacks ':'	check what happens with a small typo
D	what happens if "step" is not placed on a new line	testing consequence of typo

load_step_explanation		
Case	Input	Output
A	the file in 3.1	arrays are loaded as expected
B	the file in 3.1 but line two has been altered such that '3' has been removed	failure to load LO
C	the file in 3.1 but line two has been altered such that ':' has been removed	failure to load LO
D	the file in 3.1 but step 2 has been moved such that it starts at the same line step 1 ends.	step 2 is loaded as a part of step 1's description, and not as an individual step

Test of **forward_animation_one_step** method

These tests focuses on the text part and the model manipulations done. Tests of the actual animation and synchronization is deferred to the GUI section.

forward_animation_one_step		
Case	Property	Explanation
A	forward_animation_one_step called and new step description reached	Tests if it updates and displays appropriate description
B	forward_animation_one_step called and no new step description reached	Tests if it updates and displays appropriate description
C	forward_animation_one_step called when animation is finished	After finishing further advancement should retain the already displayed information
D	forward_animation_one_step called when play_animation is on	Tests if the running of the animation continues after we manually move one step forward
E	forward_animation_one_step called when no LO is loaded	Tests what happens if no LO is present

forward_animation_one_step		
Case	Input	Output
A	forward_animation_one_step at currentstep 3	currentstep set to 4 and the corresponding text is displayed
B	forward_animation_one_step at currentstep 2	currentstep set to 3 and the corresponding text is displayed
C	forward_animation_one_step at currentstep 34	currentstep set to 35 all the already presented text remains
D	forward_animation_one_step at currentstep 4 while run is true	currentstep set to 5, corresponding text is shown, and the animation stops there
E	forward_animation_one_step at currentstep 0 while no LO loaded	currentstep set to 1, nothing else happens

Test of **play_animation** method

In this method we test whether the animation runs automatically once started, and if it is responsive to other controllers.

play_animation		
Case	Property	Explanation
A	play_animation called and animation just started	Tests if it updates and displays appropriate description
B	play_animation called and animation finished	Tests if it proceeds after animation is finished
C	play_animation called when play_animation has already been called	Tests what happens if Play is called successively
D	play_animation called after a stop_animation has been called	Tests if it resumes as it supposed to after a break
E	play_animation called when no LO is loaded	Tests what happens if no LO is present

play_animation		
Case	Input	Output
A	play_animation at currentstep 0	currentstep set to 1 and the corresponding text is displayed, currenstep set to 2...
B	play_animation at currentstep 34	nothing happens, already displayed text remains
C	play_animation when play_animation has been called previously	ignores the last play_animation call
D	play_animation after stop_animation	animation resumes from currenstep
E	play_animation at currentstep 0 while no LO loaded	currentstep set to 1, nothing else happens

Test of **stop_animation** method

stop_animation		
Case	Property	Explanation
A	Animation running	tests if it can stop the animation
B	Animation not running	tests if it influences the run variable when it is not supposed to

stop_animation		
Case	Input	Output
A	stop_animation when run is true	run is false
B	stop_animation when run is false	run is false

Test of the **restartAnimation** method

restartAnimation		
Case	Property	Explanation
A	Animation just started	tests if works just after initialization
B	Animation has run for a while	Tests if there are any residues due to the animation having run
C	Animation finished	Tests if parameters is influenced by the animation having finished

restartAnimation		
Case	Input	Output
A	restartAnimation called at currentstep 0	parameters in restartAnimation is reset to initial values
B	restartAnimation called at currentstep 6	parameters in restartAnimation is reset to initial values
C	restartAnimation called at currentstep 34	parameters in restartAnimation is reset to initial values

Test of **resetAll** method

resetAll		
Case	Property	Explanation
A	Animation has run	tests how reset works if the animations have run
B	Animation has finished	Tests if finishing the animation leaves residues that is not cleared up
C	LO just loaded	Tests the case where a new LO has just been loaded

resetAll		
Case	Input	Output
A	resetAll called at currentstep 6	parameters in resetAll is reset to initial values and arrays are emptied
B	resetAll called at currentstep 34	parameters in resetAll is reset to initial values and arrays are emptied
C	resetAll called after LO has just been loaded	parameters in resetAll is reset to initial values and arrays are emptied

3.4 Discussion of the Model

In the design we have chosen an approach, that once a LO is opened, moving around in the animation is very simple.

Using an array to hold the step descriptions for each step, makes jumping around in the animation very easy. This was done to ensure we would be able to implement both forward, backward and restart functions.

The JeliotSteps array holds the cumulative animation steps corresponding to each step description. This makes it easy to display the correct information at all steps. At the same time the last entry holds the total number of steps in the animation, which is used as a check multiple places.

During the load, those two arrays are filled. Now when using the LO, we only need to update what step we are at, and then call the same display method.

Now each method for controlling the animation, consists of updating the current step, calling the display method, and then a call to the animation software. This gives a very simple design, that is easy to change, and adapt to different preferences.

The tradeoff is a display method that has to do a few calculations. When implementing the play_animation method we chose not to use Jeliot's own play method, and instead define our own using the forward_animation_one_step method and the run boolean. By doing it this way we are able to control the pace of the animation, we can easily stop it and we can synchronize it with text.

The main problem with this approach is that we have no way of knowing when Jeliot has finished animating a step, and hence we have to impose a wait in the method, to give it time to finish. With the methods we have available via the Visualization interface, the wait is going to be rather arbitrary, as we have to go with the highest encountered time, to ensure we never get out of synch. with the text. This however may lead to unnecessary waits between animation steps.

However had we chosen to use the built-in play method, it would have been very hard to interleave the text with the animations.

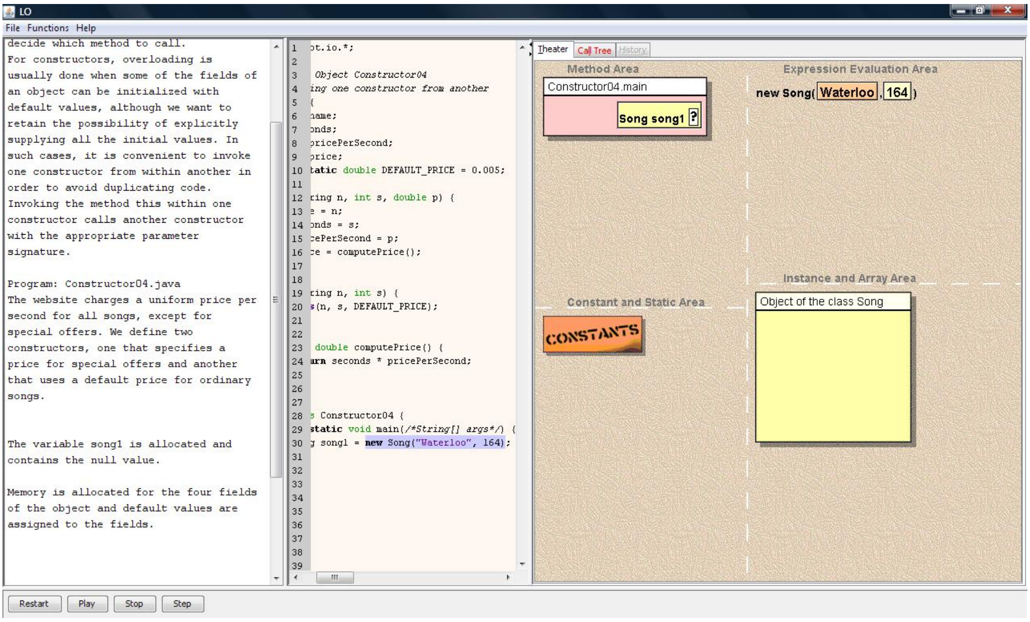
Chapter 4

The GUI

In this chapter we will present the GUI. We have tried to implement the User Interface to accomodate the requirements established in the section 1.3 and 1.4. The UI is the primary face of LOjel and is the environment the user will experience learning objects in.

Description of the GUI

Before going into details we will show a view of the running LOjel GUI. It is shown in figure4.1. The screenshot shows the *text pane* on the left, holding both explanatory text and stepwise descriptions. the *animation pane* on right, is split into an animation part and a part showing the Java code being executed. The buttons located in the bottom left are used for controlling the animation.



decide which method to call.
For constructors, overloading is usually done when some of the fields of an object can be initialized with default values, although we want to retain the possibility of explicitly supplying all the initial values. In such cases, it is convenient to invoke one constructor from within another in order to avoid duplicating code. Invoking the method this within one constructor calls another constructor with the appropriate parameter signature.

Program: Constructor04.java
The website charges a uniform price per second for all songs, except for special offers. We define two constructors, one that specifies a price for special offers and another that uses a default price for ordinary songs.

The variable song1 is allocated and contains the null value.

Memory is allocated for the four fields of the object and default values are assigned to the fields.

```
ɔt.io.*;

 Object Constructor04
ing one constructor from another
{
ɔame;
ɔnds;
ɔricePerSecond;
ɔrice;
tatic double DEFAULT_PRICE = 0.005;

ring n, int s, double p) {
ɛ = n;
ɔnds = s;
ɔePerSecond = p;
ɔe = computePrice();


ring n, int s) {
s(n, s, DEFAULT_PRICE);


 double computePrice() {
urn seconds * pricePerSecond;



s Constructor04 {
static void main(/*String[] args*/) {
g song1 = new Song("Waterloo", 164);
```

Figure 4.1: Screenshot of LOjel GUI running

4.1 Design of GUI

In this section we look at the main problems we need to address in the GUI:

- Create a main screen that holds the components of the LO
- Provide the user with means to control the animation
- Display the text, such that it corresponds to the current animation
- Provide means for easy access to filehandling, functions and help menues
- Make the size of the screen and panels customizeable

The Main screen

The main screen is what the user will see. It has to present the components of the LO in such a way that both text, code and animations are visible. Buttons

for controlling the animation should also be visible.
A side by side layout should be used.

Animation control

The user should be able to control the animation, by use of self-explanatory buttons. To make it easier for the user to cycle through the animation, accelerator keys will be implemented. The buttons should be easily accessible.

Text display

The text is displayed in a text pane at the left side of the screen. It should be able to show both explanatory text and stepwise explanations. It must be able to handle jumps in the animation. Finally it should ensure that the newest step description is about halfway up in the screen, so the user better can keep his focus on both animations and text.

Menues

Should be selfexplanatory, logicly ordered and provide the necessary functions.

Scalability

It should be possible to scale the panes in the screen, so that the animation pane can be made larger to accomodate a complicated animation. The screen should also be resizable.

4.2 Implementation

First shown is a diagram over the structure of LOjel, that provides an overview of the classes used in the GUI
Only methods that are actually used are mentioned and get/set methods are left out:

Model	
Type:	**Name of field:**
String	description
String[]	stepsDescription
int[]	JeliotSteps
int	currentstep
int	actualJeliotStep
boolean	run
LO_frame	LOframe
String	baseFileName
Return type:	**Name of method**
void	load_Text
void	load_step_explanation
void	open
void	loadLO
void	forward_animation_one_step
void	play_animation
void	update_labels
void	stop_animation
void	totalRewind
void	resetAll
void	resetAnimation

LO_frame	
Model	model
Text_Panel	textPanel
Status_Panel	statusP
Jeliot_Panel	jeliotP
Visualization	viz
JSplitPane	north
JSplitPane	south
void	initFrame
void	actionPerformed
void	keyPressed
void	mousePressed

Text_Panel	
Font	plainFont
LO_Frame	LOframe
JTextPane	jtp
JScrollPane	jsp
JTextArea	disp
void	setupLabels
void	displayExplanation
void	DisplaySteps
void	removeStepText
void	setJTextPaneFont

Status_panel

Jeliot_Panel

CheckDialogue	
JPanel	mainpanel
JTextPane	helpt
JTextField	answer
JPanel	south
JPanel	east
JTextPane	displayAnswer
LO_Frame	LOframe
String	correctAnswer
void	setupText
void	actionPerformed

JeliotLOVisualization	
JComponent	initializeVisualization
void	load
void	reset
void	step

Visualization	
void	load
JComponent	initializeVisualization
void	runFromStart
void	step
void	reset
void	stop

Figure 4.2: Overview of all classes in LOjel

The main component of the GUI will be LOframe, which is the frame holding the rest of the components, it is the principal listener. following our model-view-control approach, It is part of the control package. Text_Panel, Status_Panel, Jeliot_Panel, CheckDialogue are all in the view package. The model part has been described in the previous chapter and resides in the model package.

LO_frame:
LO_frame consists of a nested splitpane holding 3 panes. a text pane, an animation software pane and a statuspane. Using a nested splitpane allows the internal panes to be resized relative to each other
It features a menu-bar providing the user with various options. File has one selection Load for loading an LO, Functions has the selection check my knowledge which presents the student with a question that is part of the LO. Help has the standard selections of About and Help which are standard
It is attached as a listener to all panes, buttons and menu items.
The view is made up of 3 panes, a text pane, an animation pane and a statuspane. The following three classes make up the 3 panes:

Text_Panel:
The text panel makes up the text pane and is responsible for displaying the explanatory text and the stepwise descriptions according to the currentstep of the program. It uses a scrollpane, when displaying a new step description, it automatically scrolls the pane down, so the latest shown description is around the middle of the pane.
All previous step descriptions and the explanatory text is still displayed.
The class contain methods for doing those tasks.

Jeliot_Panel:
This panel makes up the animation pane and contains the animation software, its only task is to display the animations. The panel utilizes a scrollpane.

Status_Panel:
This makes up the statuspane which contains the 4 buttons used to control the animation. Each button has a listener attached to it.

CheckDialogue:
Creates a new frame, where the question text, prepared by the instructor, is displayed. An editable textfield is provided for getting the answer. A JLabel display the answer to the question when prompted by the user.
A simple text loading method is used

Visualization:
This interface provides the methods allowed to control the visualization-software from outside. The interface was provided by prof. Mordechai Ben-Ari

JeliotLOVisualization:
This class is provided by author of JELIOT, Nico Myller to implement the Visualization interface. Unfortunately at present, it does not implement every method in the interface, why only some methods can be used. As stated in the previous chapter we decided not to use the play and stop methods.

Controlling the animation

LO_frame works as the main controller class. Below is a short description of how it control the functions.

```
ActionPerformed
```

- When a button is clicked this method is called. The event string is analysed and the approprate actions in Model are executed .
  When calling Play, a new thread is created and calls the play_animation method in Model. This is done to prevent Jeliot from freezing the GUI while animating.
- When a menu item is selected, this method calls the appropriate actions in Model. The items are Load, Check my Knowledge, Help and About.

```
KeyPressed
```

LO_frame implements key listener and is attached to all components. This allows the user to use accelerator keys to control the animation. Whenever a key is pressed keyPressed is called, and the method takes the appropriate action.
The following keystrokes are used:

Button	Function	Keyboard Shortcut
Step	calls forward_animation_one_step	Space.
Play	calls play_animation	Enter
Stop	calls stop_animation	Esc
Restart	calls restartAnimation	Backspace

```
mousePressed
```

LOframe implements the mouse listener. This is primarily because a keylistener component has to be in focus, for it to generate key-events. We use the mouse listener to ensure that no matter which component (except the animation screen) the mouse is pressed in, a mousePressed event is generated and this method

transfer focus to Status_Panel which has a keylistener attached.
mouseEvents generated when pressing the mouse in the animation panel are apparently consumed by JELIOT.

4.3 Test

We have chosen to test the following areas of the GUI:

- Opening of a LO via filechooser
- Test of menu items
- Resizing of components
- Controlling the animation, including accelerators and synchronization

Before testing each functionality, we present the goal we want to achieve when executing it.
We first list test cases and their properties, then we list input and expected output for each test case:

Opening of LO via Filechooser

When testing for this we have the following goals we want to achieve: first that the text and animations are loaded (which is tested in the model section), and secondly the GUI displays the explanation in the *textpanel*, not scrolled down, the code is shown and the "curtains" are drawn back in the *animation pane*
Testcases are shown in 4.3 and 4.3

Open functionality		
Case	Properties	Explanation
A	Files exists, formats are fine and are in right directory	tests if the function open when everything is as its supposed to be
B	Files exists, formats are fine, but in wrong directory	Tests the programs reaktion to missing a misplaced file
C	format of .stp file is wrong	Tests what happens if the stp file is wrongly formattet
D	format of .chk file is wrong	Tests what happens when .chk file is out of order
E	.exp file is missing	Tests what happens if the explanation file is missing
F	.chk file is missing	Tests what happens if the check, question and answer file is missing
G	.stp file is missing	Tests what happens if the stepwise explanation file is missing
H	.java file is missing	Tests what happens if the java file is missing
I	animation of current LO has started	Tests what happens if the existing LO is already running

Open functionality		
Case	Input	Output
A	Files are the one from the constructorLO, all placed right	output as stated in goal.
B	Files are the ones from the constructorLO but placed at the toplevel directory	nothing happens, the current LO continues to run
C	the constructor.stp file has been altered so a total number of steps are missing	nothing happens
D	the constructor.chk file has been altered so ”answer” is no longer there	new LO is opened but the check dialogue doesn’t work
E	the constructor.exp file is deleted	nothing happens
F	the constructor.stp file is deleted	nothing happens
G	the constructor.chk file is deleted	same as in D
H	the constructor.java file is deleted	nothing happens
I	same as A but a current LO is already running	output as stated in goal

Test of Menu Items

When testing for this we have the following goals we want to achieve:
Every menu item should perform the desired actions and open up the various dialogues.
We perform the tests simply by clicking the menu items from within the GUI, below is the test results:

- Load correctly opens the filechooser and calls the appropriate methods
- Check my Knowledge correctly opens the check dialogue and behaves as wanted
- About opens the about screen and loads the text from about.html
- Help opens the help screen and loads the text from help.html

Test of resizing

Goal:
When resizing the various components, the internal drawings and proportions are the same.
This test was done by opening the program and varying the size of the 3 main panels and the main frame.
We observe that the goals are met.

Test of animation control and synchronization

Goal:
For each controller: when selected (clicked or by a keystroke) appropriate actions should be executed. The text and animation should appear in as specified by the instructor.

A series of test for each component is run, and afterwards we show that using the accelerator keys the same functionality can be achieved. We will have done the same tests for the accelerator keys as for the buttons, but we will refrain from stating them in this report.

Step Forward

the goal of this function is to display the text corresponding to this step and then show the animation.

Step forward		
Case	Properties	Explanation
A	Step button is clicked once currentstep is 3	tests if the new text is displayed and then the current animation is done
B	Step button is clicked twice, but allowing the animation time to finish	Tests if the step method can be repeated
C	Step button is clicked twice in succession	Tests if the step method can handle two succesive calls, without loosing sync.
D	Step button is clicked after the animation is finished	Tests if the program can handle LO's after the animation is finished
E	Step button is clicked when play is active	Tests if step can be used, while the animation is being run
F	Step button is clicked when no LO is loaded	Tests if step can be used, while there is no LO

Step forward		
Case	Input	Output
A	Step button is clicked where currentstep is 3	output as stated in goal.
B	Step button is clicked twice, but second only after first animation has finished, currentstep is 3	same as A, followed by the text and animation of ensuing step
C	Step button is clicked twice, currentstep is 3	same as B
D	Step button is clicked where currentstep is 34	nothing visible happens, but currenstep is advanced to 35
E	Play is called followed by Step	The animation is moved one step forward, and the running of the animations are stopped
F	Step is called with no LO loaded	nothing happens

In test case C, there was a slight mismatch between text and animation, as the animation wasn't allowed time to finish.

Play

The goal of this function is to automatically advance the animation in a synchronized fashion, call Step method, and then give it time to finish.

Play		
Case	Properties	Explanation
A	Play button is clicked just after LO has been loaded	Tests if it fulfills the goal from a standard starting point
B	Play button is clicked, after step forward	Tests if it is able to continue playing from any given currentstep
C	Play button is clicked after a Stop	Tests if Play is able to resume playing after having been stopped
D	Play button is clicked after a Restart	Tests if Play is able to start playing after having been reset
E	Play button is clicked when animation has finished	Tests if it executes weird behaviour after the animation has finished
F	Play button is clicked successively	Tests if it can handle successive hits
G	Play button is clicked after a different LO has been loaded	Tests if Play can handle switched LO's

Play		
Case	Input	Output
A	a LO is loaded Play button is clicked	output as stated in goal.
B	Step button is clicked followed by Play	output as stated in goal
C	Stop button is clicked, followed by Play	output as stated in goal
D	Restart button is clicked, followed by Play	output as stated in goal
E	Play button is clicked at currentstep 34	nothing happens
F	Play button is clicked twice	output is as stated goal
G	a LO is loaded, advanced one step, then a new is opened and Play button is clicked	output as stated goal

Stop

The goal of this method is to stop the animation from running.

Stop		
Case	Properties	Explanation
A	Stop button is clicked while a step is being animated	tests if it allows the current animation to finish and then stops running
B	Stop button is clicked while waiting for next step to be animated	Tests if it prevents a new step from being drawn
C	Stop button is clicked twice	Tests if it can handle succesive calls

Stop		
Case	Input	Output
A	Stop button is clicked after play has been started and an animation is being done	running halts when animation is finished
B	Stop button is clicked after play has been started and an animation has been done	same as A, no new step is allowed to start
C	Stop button is clicked while no LO has been loaded	nothing happens

Restart

The Goal of this method is to bring the animation back to the initial point. We tested the parameters in the previous chapter, so the primary focus is if the animation gets reset.

Restart		
Case	Properties	Explanation
A	Restart button is clicked just after a LO has been loaded	Tests if it works when a LO is just loaded
B	Restart button is clicked when animation is running	Tests if it stops the animation and restarts regardless of being in the middle of a stepanimation
C	Restart button is clicked when animation is finished	Tests if can rewind the animation when it is fully finished
D	Restart button is clicked successively	Tests if it can handle successive hits

Restart		
Case	Input	Output
A	Restart button is clicked after a LO has been loaded	the curtains are drawn back again, nothing else happens
B	Restart button is clicked after a step has been called	animation is immediately reset and the curtains drawn back again
C	Restart button is clicked when the animation is finished	animation is restartet and the curtains are drawn back again
D	Restart button is clicked twice	animation is restartet and the curtains are drawn back again each time it is clicked

Testing the accelerator keys

Goal: Each key should result in the appropriate action taken, and be able to do it regardless of the component currently in focus, except when the focus is in the animation pane.
We first test if the keystrokes are linked to the right actions, and then we test under what circumstances they work.

Accelerator keys		
Case	Properties	Explanation
A	Space is hit while the focus is in statuspanel	Test if Space works
B	Enter is hit while the focus is in statuspanel	Test if Enter works
C	Esc is hit while the focus is in statuspanel	Test if Esc works
C	Backspace is hit while the focus is in statuspanel	Test if Backspace works
E	Space is hit while the focus is not in the mainframe	Test if it works while focus is outside the program
F	Space is hit while the focus is in the textpanel	Test if it works while focus is in the textpanel
G	Space is hit while the focus is in the JeliotPanel	Test if it works while focus is in the Jeliotpanel
H	Space is hit after a button has been clicked	Test if it can handle a button being clicked first
I	Space is hit after a menu item has been selected	Test if it can work after the menu items have been selected

Accelerator keys		
Case	Input	Output
A	StatusPanel is set in focus and Space is hit	step forward method is executed
B	StatusPanel is set in focus and Enter is hit	Play method is executed
C	StatusPanel is set in focus and Enter followed by Esc is hit	Play is started but stopped after Esc is clicked
D	StatusPanel is set in focus and Backspace is hit	Restart is executed
E	Another window is set in focus and Space is hit	nothing happens
F	TextPanel is set in focus and Space is hit	step forward method is executed
G	JeliotPanel is set in focus and Space is hit	nothing happens
H	Step button is clicked and Space is hit	step forward method is executed
I	Check my Knowledge is selected and Space is hit	step forward method is executed

4.4 Discussion of the program

Most of the features described in the introduction chapter have been implemented and shown to work. However the ability to go one step back in the animation has not been fully implemented. The actual call to the animation software has been left out
The Visualization interface does not define methods that will allow us to smoothly implement it and therefore we have left it out. With the current methods, we would have had to reanimate the previous n-2 steps again, which would be infeasible.

For the accelerator keys to work, a panel with a KeyListener attached must be in focus. We have implemented a MouseListener to ensure that each mouse click transfers focus to Status_Panel. There is one situation where it does not work; when the animation pane is in focus all events are consumed by the visualization software, rendering our listeners useless
To ensure the accelerator keys can be used, it is thus preferable that the user clicks on either the textitstatuspane or the *textpane*. This is cumbersome and clearly an element worth improving.

There is an issue with the buttons and accelerator keys in the case where Jeliot asks for user input. The input has to be finished by an 'enter' stroke, which results in the transfer of focus to the 1. button which is the restart button. If the user is not aware of this, and just hits another accelerator key, the whole animation will restart.
This issue is hard to fix, since the method causing the focus transfer happens in the animation software, which we have no access to. However we can minimize the effect by switching the location of the restart and the more 'harmless' stop button.

As discussed in the previous chapter the play method has a predefined wait between each animation step. This causes situations with unnecessary wait time, and where the animation is not finished when advancing to next step. This is preferably remedied by implementing a sort of notification from the animation software whenever it is done animating a step.
It would be obvious to put it in Visualization.

The lack of notification bring some other issues with it. One is the issue of the user rapidly clicking on the buttons. As we saw in the test section. Restart and

Stop is no problem, Step initially works fine, but after mulitple hits the text and animation gets out of sync. Play has the problem that in a very rare case, we might end up having two threads running play_animation at the same time, which might lead to synchronization trouble.
To solve those issues we could deactivate the buttons when clicked, but we would have to do it for an unknown period of time. Which might create unnecessary long waits in controlling the animation. The best solution would be to work with the authors of the animation software, to implement the notification system.

We chose to implement an evaluation dialogue, where the instructor have the opportunity to prepare questions for the student about concepts in the LO.
Since this program is primarily a demonstration of possibilities, the current implementation only features a single question and answer, but it can easily be changed to pose arbitrarily many questions.
Since the check questions are not critical to the LO, we have made it possible to run the LO even though the .chk file is missing or in disorder

To make the overall user experience more smooth, we wanted to make the newest step description appear approximately at center of the *text pane*. To achieve this we automatically scroll the text pane down, whenever new text is added (and if it is needed) and append some whitespace characters. The number of whitespace characters can be argued about, some people prefer the newest text at the top of the screen and some prefer as much previous text in the same vision as possible. The number chosen seeks to accomodate both views.

The in-program help and about files are written as html files. This makes it easy to display the same text both on the web and in the LOjel.

One of the key features was to make it easy for instructors to write new learning objects We have reduced the workload significantly by reducing his/hers work to only write 3 files (with a fourth optional). The only file requiring some work besides crafting the actual content of the LO, is the stepwise explanation file.
However since only the instructor knows exactly how many animation steps he want each of his step descriptions to cover, he would have to specify this anyway. It might have been made easier by a conversion guide between type of animation and animation steps, i.e. an assignment operation corresponds to 2 animation-steps.

4.5 Conclusion on the GUI

We have constructed a working program that fulfills almost every requirement specified in the introduction. There are some small issues but nothing that hampers the overall functionality. Use of the program and possible improvements will be discussed in the next chapter.

Chapter 5

Evaluation and future work

In this chapter we will briefly look at how this framework is used and how we can improve it.

5.1 The Complete Framework

The final framework consists of the LOjel program, the Jeliot program distributed as a .jar file, a doc directory containg help and about screens, an example directory containing all files used for constructing LO's. This is the directory where all new LO's should be placed. Finally a .bat file, LOjel.bat, is provided as the driver file. Once clicked it will start LOjel up.
The framework is distributed in a zip file, that can be unpacked and run directly using the bat file.

5.2 Use and feasibility of LOjel

The original purpose of this paper, was to investigate if it was feasible to construct a framework for integrating visualization software with text into learning

objects.
We showed in the last chapter that the essential features and most of the ones we wanted, for such a framework, was met in LOjel.
Full instructions on how to use LOjel can be found in appendix A.
We tested how long it took to adapt a learning object to this framework, that is an already existing LO where all text and java code exist in advance but as seperate entities.
The tested LO was "constructor 4.4" created by prof. Mordechai Ben-Ari, and consists of some explanatory text, a piece of JAVA code, some keypoints regarding the code and a few evaluation questions. It took at most 10 minutes to adapt it to LOjel, which should be in the range of a "feasible" amount of time spent.
Afterwards we created a small LO, focusing on the While-loop construct. We used a piece of code, Average.java, originally included in the JELIOT distribution. We then created explanatory text, stepwise description and a single evaluation question.
The whole creation process took approximately 30 minutes. Although the LO was not done using refined pedagogical means and can be made much better, it serves as an example that using LOjel can provide good LO's without spending insurmountable amounts of time.
The previous two examples indicates that it is indeed feasible to integrate visualization software and text into learning objects using a framework like LOjel.

5.3 Possible Improvements

There are a number of possible improvements to the framework, some focuses on improving the current functionalities while others focus on major added functionality. Some of the suggestions have been touched upon in the discussion in chapter 2.

Minor Improvements and issue fixes

Incorporating a notification method in the visualization interface. This should enable us to synchronize text and animation better.
Expanding the Visualization with a one-step-rewind method. This will provide a more flexible control system for the animation. With this method the student is able to go 1 or 2 steps back and replay does step. With the current version, they have to restart and then simulate all previous n-1 step. Which is cumbersome and not beneficial for learning.

A label in the *status pane* indicating whether the system is currently animating or not, would prevent "good-intended" users from clicking a button multiple times, because they was unsure if it was activated.
Another minor improvement would be the question and answer dialogue, that could be improved both by handling more question/answers and by changing the format of the dialogue, i.e. multiple choice could be used. This would greatly enhance the feedback the student could get from the learning object.

Major Improvements

Development GUI
To assist the development process of an LO, we could create an instructor-mode for the framework, where the text editor was editable.
In this state it should be possible to control the animation, and then at any step in the animation, the instructor should be able to add text and save the mapping to a file. We would then ease the creation of the .stp file and would thus reduce the overall work needed by the instructor.

Development of intelligent question answer module
In the current version the evaluation is done by preprepared questions from the instructor. By introducing a module for automatic question generation, we would be able to evaluate and produce suggestions based on his needs. Furthermore this will reduce the workload for the instructors.
JELIOTis currently providing an option that generates questions about the outcome of a given code line [3]. This prompts the student to be interactive and think about the code, before it is executed and thereby enhances learning.
If this feature could be made available through the Visualization interface, learning objects in LOjel would benefit greatly.

Chapter 6

Conclusion

We have in this project shown that it is possible to construct a framework that integrates visualization software, in our particular case Jeliot, and explanatory text into a learning object using a common interface. We have argued that it is easy to use by students, and requires little effort in preparation by the instructors.
We have thus proved the feasibility of integrating visualization and explanatory text into learning objects through a common interface.

Appendix A

Userguide to LOjel

LOJEL—Learning Objects with JELIOT
User's Guide

Version 1.0

Jens Peter Träff
Technical University of Denmark
DK-2800 Lyngby, Denmark

June 24, 2011

1 Introduction

LOJEL is a framework for creating and displaying *learning objects (LOs)* based upon JELIOT. JELIOT is a system animation of program in JAVA. It takes a program in JAVA and *automatically* generates a detailed animation of the execution of the program. LOJEL is designed to facilitate the creation of LOs by integrating textual material with the JELIOT animations. Since the animations are generated automatically by JELIOT, the effort needed to create LOs is much less that would be required using generic multimedia software such as Flash.

For more on JELIOT see:

- A. Moreno, N. Myller, E. Sutinen, M. Ben-Ari. Visualizing programs with Jeliot 3. *Conference on Advanced Visual Interfaces*, Gallipoli, Italy, 2004, 373–376.
- M. Ben-Ari, R. Bednarik, R. Ben-Bassat Levy, G. Ebel, A. Moreno, N. Myller, E. Sutinen. A decade of research and development on program animation: The Jeliot experience. *Journal of Visual Languages and Computing*, 2011 (in press).
  Available online at `http://dx.doi.org/10.1016/j.jvlc.2011.04.004`.

The JELIOT website is: `http://cs.joensuu.fi/jeliot/`.

Section 2 describes how to install and run LOJEL. Section 3 explains how to work the the LOJEL interface. Section 4 shows how an instructor creates learning objects for LOJEL. Section 5 documents the software package.

2 Installation and execution

LOJEL requires JAVA JRE 1.5 or above.

The program is distributed in a zip file: `lojel-n-n.zip`. Download the zip file and open it into a clean directory. The zip-file contains the following directorys: `src` for the source files, `bin` for the executable files, `doc` for the documentation and `examples` for the example LOs. Additional files are `help.html` and `about.html` that are used for the help and about screens.

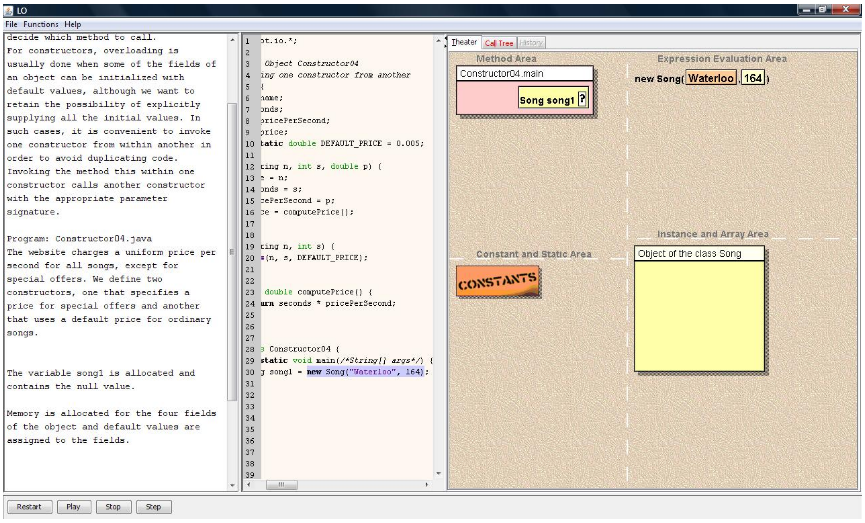
To run from the installation directory, use the following command:

```
java -cp bin;jeliot.jar control.Driver
```

This command is contained in the file `LOjel.bat`.

3 Graphical user interface

A screenshot of the LOJEL GUI is shown below in figure 1. In addition to a menu bar, there are three panes in the frame: the *text pane* on the left, the *animation pane* on the right, and the *status pane* on the bottom.



decide which method to call.
For constructors, overloading is usually done when some of the fields of an object can be initialized with default values, although we want to retain the possibility of explicitly supplying all the initial values. In such cases, it is convenient to invoke one constructor from within another in order to avoid duplicating code. Invoking the method this within one constructor calls another constructor with the appropriate parameter signature.

Program: Constructor04.java
The website charges a uniform price per second for all songs, except for special offers. We define two constructors, one that specifies a price for special offers and another that uses a default price for ordinary songs.

The variable song1 is allocated and contains the null value.

Memory is allocated for the four fields of the object and default values are assigned to the fields.

```
ɔt.io.*;

 Object Constructor04
ing one constructor from another
{
name;
onds;
pricePerSecond;
price;
tatic double DEFAULT_PRICE = 0.005;

ring n, int s, double p) {
e = n;
onds = s;
cePerSecond = p;
ce = computePrice();


ring n, int s) {
s(n, s, DEFAULT_PRICE);


 double computePrice() {
urn seconds * pricePerSecond;



s Constructor04 {
static void main(/*String[] args*/) {
g song1 = new Song("Waterloo", 164);
```

Figure 1: Screenshot of LOJEL

3.1 Menus

The File menu has one selection Load for loading an LO. The Function menu has the selection Check my knowlegde which presents the student with a question that is part of the LO. The Help and About screens are standard.

3.2 Text pane

The text pane consists of a scrollable non-edible text area. It is here that the explanatory text and the stepwise explanations are shown. Whenever the description of a new step is displayed, the pane is scrolled such that the new text is around the middle of the screen.

3.3 Animation pane

The animation pane contains the display of JELIOT: the source code on the left and the animation on the right. Here is an explanation of the elements of this display.

When editing a program in JELIOT, animation pane is covered with a blue curtain. When you move to the animation state, the curtain slides open and reveals a light brown background. When you start the animation, the frame is divided into four separate areas with dashed white lines. The areas in left-right, top-bottom order are: the *Method Area*, the *Expression Evaluation Area*, the *Constant Area*, and the *Instance and Array Area*.

The Method Area displays the stack activation frames for all the methods that are currently being processed. Activation frames are displayed as boxes that hold variables inside. Return values are animated with a larger box holding the value inside. For variables of primitive or `String` type, the value is displayed adjacent to the name; references are shown as arrow to *Instance and Array Area*. A `null` is denoted by the electrical ground symbol.

The expression evaluation area animates the evaluation of expressions. Information on the results of evaluating expressions are also shown here, as well as the dialog boxes for user input.

Whenever any literals (of any type) are needed by the code, they are brought to the animation from the *Constants box* in the Constant Area.

Finally, the Instance and Array Area holds dynamically allocated objects, such as instances of classes and arrays.

3.4 Status pane

The LO is controlled by four buttons in the status pane (Table 1).

Button	**Function**	**Shortcut**
Step	Run one step of the animation and display corresponding text.	Space
Play	Run Step until Stop is clicked or the program ends.	Enter
Stop	Stop a Run.	Esc
Restart	Start at the beginning of the LO	Backspace

Table 1: Animation toolbar commands.

Important:

- By *step* is meant a step of the LO, which may consist of more than one step of JELIOT.
- The Play command is executing by calling the Step command and then waiting 3 seconds before proceeding with next step.

4 How to prepare a learning object

An LO for use with LOJEL consists of four files with the same name and with the extensions shown in Table 2. They must all be in the same directory. Note that only the JAVA files is actually required.

Filecontent	Extension	Format
Java code	`java`	The Java program to be animated in Jeliot.
Explanatory text	`exp`	Text.
Question	`chk`	The text of the question, followed by a blank line, followed by the word `answer` and answer on a new line.
Step descriptions	`stp`	The first line contains the total number of steps; for each step, the following format is used: `step` [step no.]: [number of animation steps] [text] example of a step file can be seen in figure: 2

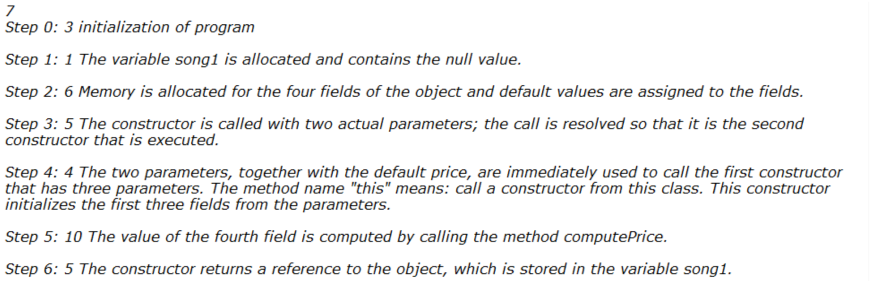
Table 2: LO file formats

The Java file has to follow certain conventions; see the Jeliot documentation.

The explanatory text provides the information that introduces the student to the concept of the LO. It is displayed in the text pane when the LO is loaded.

The question provides the instructor with the opportunity to have LOJel ask the student a question and check the answer.

The most important file is the Step file that coordinates the step-by-step explanation with steps of the animation. A *step of the LO* can require Jeliot to execute several of its own steps. Therefore, the instructor must provide a list of lines, one for each step of the LO, which specify the number of steps of the animation and the text that is displayed at the same time. The instructor will have to experiment to find the optimal number of Jeliot steps for one LOJel step.



7
Step 0: 3 initialization of program

Step 1: 1 The variable song1 is allocated and contains the null value.

Step 2: 6 Memory is allocated for the four fields of the object and default values are assigned to the fields.

Step 3: 5 The constructor is called with two actual parameters; the call is resolved so that it is the second constructor that is executed.

Step 4: 4 The two parameters, together with the default price, are immediately used to call the first constructor that has three parameters. The method name "this" means: call a constructor from this class. This constructor initializes the first three fields from the parameters.

Step 5: 10 The value of the fourth field is computed by calling the method computePrice.

Step 6: 5 The constructor returns a reference to the object, which is stored in the variable song1.

Figure 2: sample step file

5 Software documentation

This program is built according to classic Model-View-Control design. The main controller class is the LO_frame, which instantiates the GUI and listens to all its components. It contains the methods actionPerformed, mouseClicked and keyPressed which handle all the events generated by the menu items, buttons and the accelerator keys. It calls the appropriate methods for handling the events.

The Model class is the main class in the model-part. It contain fields for storing and synchronizing text and animation. It contains the following prime methods:

- load_text loads the explanatory text from the .exp file.
- load_step_explanation loads the stepwise description and links it to the appropriate animation steps. While loading the information, the method fills out the arrays jeliotSteps and stepsDescription
- open opens the filechooser and extracts the basefilename from the file. This is used in the loadLO method, which in turn calls the various load methods with the basefilename and the appropriate extensions.
- forward_animation_one_step increments the current step, displays the new text and then calls the animation software for it to advance the simulation by one step.
- play_animation successively calls the forward method until either the stop method is called or the animation finishes. After each call the method waits for 3000 ms. to give the animation time to finish.
- stop_animaiton stops the animation by setting a flag.
- resetAnimation resets all the animation parameters and displays the text corresponding to the 0'th step.
- totalRewind Rewinds the animation by first calling the stop_animation method and then calling resetAnimation.
- resetALL empties all arrays and resets all parameters to prepare for a loading of a new LO.

The interface Visualization specifies the interface between the animation software and LOjel: the methods that the animation software has to implement.

JeliotLOVisualization is the class that adapts JELIOT to fit the requirements of the Visualization interface.

The GUI consists of a number of classes specifying different components:

Text_Panel constructs the left text pane on the left side; it consists of a JTextArea placed in a JScrollPane. displaySteps shows the stepwise explanation appropriate for the current animation step and scrolls the pane down to keep the new text in focus. displayExplanation displays the explanatory text.

Jeliot_Panel is the panel holding the animation software, in this case JELIOT. Status_Panel is the bottom panel, it holds the animation control buttons. InfoWindow allows us to use html documents to specify the help and about screens. checkDialogue generates the check dialogue, using the text of the `chk` file; it provides a question for the student and can evaluate his answer. Driver is the class that starts the whole program.

6 Known Issues

- The accelerator keys don't work when JELIOT panel is in focus.
- A method for determining if the animation software is currently running has not yet been implemented. Therefore, an arbitrary 3000 ms wait has been used.

Appendix B

Source code for LOjel

```
package model;

import java.awt.Rectangle;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Date;
import java.util.Scanner;

import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

import control.LO_frame;



/**The Model class is the internal representation of ↩
    the program and holds almost all methods used for↩
     manipulating the simulations and
 * the learning object in generel.
 * @author Jens Peter Träff
 *
 */


public class Model {

   public String description="";
   public String[] stepsDescription; //estimate how many steps are required later
   public int[] jeliotSteps; //holds the cumulative number of animsteps
   public int currentStep;
   private int actualJeliotStep; //represents the actual step explanation currently being displayed
   private boolean run=false;
   private LO_frame LOframe;
   private String baseFileName="";
   //private jeliotobject jeliot;


   public Model(LO_frame LOframe){
      this.LOframe= LOframe;
   }
   public int getActualJeliotStep() {
      return actualJeliotStep;
   }



   /**Load the explanatory text into the string variable "description"
    * @param filename
    * @throws FileNotFoundException
    */
   public void load_Text(String filename) throws FileNotFoundException{

      Scanner input= new Scanner(new File(filename));
      while (input.hasNextLine()){
         description += input.nextLine();
         description += "\n";
      }
   }

   /**Load the stepwise explanation, each step is ↩
       loaded into the stepsdescription
    * array, and the corresponding animation steps ↩
        are placed in the JeliotStep array
    * @param filename
    * @throws FileNotFoundException
    */
   public void load_step_explanation(String filename)↩
       throws FileNotFoundException{
      Scanner input= new Scanner(new File(filename));
      if(input.hasNextInt()){
         int totalsteps=input.nextInt();
         stepsDescription= new String[totalsteps];
         jeliotSteps = new int[totalsteps];
      }
      int currentStepReading=-1;
      while (input.hasNextLine()){
         String nextLine=input.nextLine();
         if(nextLine.toLowerCase().contains("step")){↩
            //this part can be done more smoothly, ↩
            investigate when possible
            currentStepReading++;

            String s = nextLine.substring(nextLine.↩
               indexOf(":")+1); //skips the intro ↩
               part
            Scanner read = new Scanner(s);
            if(read.hasNextInt()){ //reads number of ↩
               animation steps associated with this ↩
               explanation step
               if (currentStepReading==0){
                  jeliotSteps[currentStepReading]=↩
                     read.nextInt();
               }else{
               jeliotSteps[currentStepReading]=↩
                  jeliotSteps[currentStepReading-1]+↩
                  read.nextInt();
               }
               }

            read.close();
```

```
//          char x=nextLine.charAt(nextLine.indexOf("↩
   step")+4+4);
//          System.out.println(Character.↩
   getNumericValue(x));
            stepsDescription[currentStepReading]= s.↩
               substring(3);
            //System.out.println(currentStepReading);
         }
         else if (currentStepReading==-1){
            System.out.println("illegal␣input");
               //alternative to skip a line?
         }
         else{
         // System.out.println("jeg er her");
            stepsDescription[currentStepReading]+="\n↩
               " + nextLine;
         }
      }

// for (int i=0; i<jeliotSteps.length; i++){
//    System.out.println("jeliot step "+ i + " " +↩
   jeliotSteps[i]);
// }
   //System.out.println("indhold af nr 1 " + steps[0]↩
       + " færdig");
   //TODO works quite well initially,still needs to ↩
      figure out how to gorge the length of the ↩
      Jeliot animation,
   //and thus determine length of array...
}

/**Open the fileChooser for loading a new Learning ↩
   Object
 * @throws IOException
 */
public void open() throws IOException{
      JFrame parent = new JFrame();
      String selectedFileName;
      JFileChooser FC = new JFileChooser("./examples"↩
         );
      int returnVal = FC.showOpenDialog(parent);
      if(returnVal == JFileChooser.APPROVE_OPTION) {
         selectedFileName =  FC.getSelectedFile().↩
            getName();
         baseFileName= selectedFileName.substring(0, ↩
            selectedFileName.indexOf('.'));
         baseFileName="./examples/"+baseFileName;
         loadLO(baseFileName);
      }
   }



public String getBaseFileName() {
   return baseFileName;
}



/**When a file is chosen, the learning object using ↩
   this file is loaded and
 * initialized
 * @param filename
 * @throws IOException
 */
public void loadLO(String filename) throws ↩
   IOException{
   //if program has previously been initialized, ↩
      reset all
   if(jeliotSteps!=null){
      resetAll();
   }
   load_step_explanation(filename+".stp");
   load_Text(filename+".exp");
   LOframe.getViz().load(filename+".java");
   LOframe.getTextPanel().displayExplanation();
// LOframe.getTextPanel().setVisible(true);
   //find a way to give jeliot time to load the code
   }

/**Move the animation one step forward
 * @throws Exception
 */
public void forward_animation_one_step() throws ↩
   Exception{
   update_labels(1);
   LOframe.getViz().step(1);

   //move_jeliot_animation_one_forward
   //jeliot.step(1);
   //TODO calls update_labels (1) and then moves ↩
      animation forward.(calls jeliot)
}

/**Move the animation one step backwards
 * not used in the current implementation
 * @throws Exception
 */
public void rewind_animation_one_step() throws ↩
   Exception{
   update_labels(-1);
   System.out.println("heree");
   LOframe.getViz().step(-2);
   //jeliot.run(-1);
}

/**Update the labels, to reflect the actual step ↩
   currently animated.
 * and call methods to update the text shown in the ↩
    textpanel
 * @param stepsMoved
 */
public void update_labels(int stepsMoved){
   currentStep+=stepsMoved;
   LOframe.getTextPanel().displaySteps();
// LOframe.getJeliotP().repaint();
   //TODO update the labels affected by a step change↩
       in the animation,
}


/**only used for testing purposes
 *
 */
public void Display_text(){
   System.out.println(description);
}

/**Automatically advances the simulation by calling ↩
   the step forward function
 * waiting a short amount of time, and then advancing↩
     again. This continues until
 * either the animation is finished or the Stop-method is called
 * @throws Exception
 */
public void play_animation() throws Exception{
   if(run==true){
      return;
   }
   run=true;
   while(run && currentStep<jeliotSteps[jeliotSteps.length-1]){ //less than number of total steps

   forward_animation_one_step();
   Thread.sleep(3000); //give jeliot time to finish the animations
}

}

/**Stops the animation
 *
 */
public void stop_animation(){
   run=false;
}

/**Rewinds the animation and stops it if it was
 * playing
 */
public void restartAnimation() {
stop_animation();
resetAnimation();

}

/**Resets the animation, and all animation parameters
 *
 */
private void resetAnimation() {
currentStep=0;
actualJeliotStep=0;
try {
   LOframe.getViz().reset();
} catch (Exception e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
}
LOframe.getTextPanel().removeStepText();
update_labels(0);
}


/**Reset all parameters
 * making the program ready for displaying a new Learning Object
 *
 */
public void resetAll(){
   description="";
   for(int i=0; i<stepsDescription.length; i++){
      stepsDescription[i]="";
   }
   for(int i=0; i<jeliotSteps.length; i++){
      jeliotSteps[i]=0;
   }
// for(int i=0; i<LOframe.getTextPanel().getEndMarkers().length; i++){
//     LOframe.getTextPanel().getEndMarkers()[i]=0;
// }
   LOframe.getTextPanel().getDisp().setText("");
   currentStep=0;
   actualJeliotStep=0;
   //TODO write method to reset all parameters when called
}



public void setActualJeliotStep(int actualJeliotStep){
   this.actualJeliotStep = actualJeliotStep;
}

}
```

```
/*

  Universal Java interface to a visualization
  Copyright 2010 by Moti Ben-Ari under GNU GPL

  This interface is intended to enable pedagogical software
  (such as learning objects, learning management systems,
  interactive learning environments) to control visualizations
  written in Java which will implement the interface.

  The details of the parameters, etc., are to be specified separately
  for each visualization implementing the interface.

*/
package model;
public interface Visualization {

  // Initialize the visualization, possibly with arguments
  // The visualization is to be displayed in the supplied frame
  // Alternatively, the visualization could supply the JFrame
  public abstract void initialize(
    javax.swing.JFrame frame, String args[]) throws Exception;
  public abstract javax.swing.JFrame initialize(
    String args[]) throws Exception;

  public abstract javax.swing.JComponent initializeVisualization(
          String args[]) throws Exception;

  // Load a file such as a program or algorithm to visualize
  public abstract void load(String fileName) throws java.io.IOException;

  // Get/Set internal options
  public abstract String[] getOptions();
  public abstract void     setOptions(String args[]);

  // Run from start or run something, step, reset the visualization
  public abstract void runFromStart()   throws Exception;
  public abstract void run(String what) throws Exception;
  public abstract void step(int steps)  throws Exception;
  public abstract void reset()          throws Exception;
  public abstract void stop()          throws Exception;

  // Query the visualization and return information
  //   such as the value of a variable
  // For an object, its toString would be returned
  public abstract int    getIntValue   (String name);
  public abstract double getDoubleValue(String name);
  public abstract String getStringValue(String name);
  public abstract String getObjectValue(String name);
}
```

```
package model;

import java.io.File;
import java.io.IOException;

import javax.swing.JComponent;
import javax.swing.JFrame;

import jeliot.Jeliot;
import jeliot.gui.LoadJeliot;



/** This class adaps jeliot to visualization interface
 * @author Niko Myller
 *
 */
public class JeliotLOVisualization extends Jeliot implements Visualization {

   public JeliotLOVisualization() {
      super("jeliot.io.*");
   }

   public JFrame initialize(String[] args) throws Exception {
      LoadJeliot.simpleStart(this);
      handleArgs(args);
      return gui.getFrame();
   }

   public JComponent initializeVisualization(String[] args) throws Exception {
      LoadJeliot.simpleStart(this);
      handleArgs(args);
      return gui.getTopSplitPane();
   }

   public void load(String fileName) throws IOException {
      final File programFile = new File(fileName);
      if (programFile.exists()) {
         setProgram(programFile);
         try {
            reset();
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
   }

   public void reset() throws Exception {
      cleanUp();
      compile(null);
   }

   public void runFromStart() throws Exception {
      playAnimation();
   }

   public void step(int steps) throws Exception {
      int count = 0;
      // if the animation has ended or stopped to wait for input
      // then the steps should end as well even ↩
         though there are steps left
      //System.out.println("" + (count < steps) + ":"↩
          + !isFreezed() + ":" + !isFinished());
      while (count < steps && !isFreezed() && !↩
         isFinished()) {
         if (playStepAnimation()) {
            count++;
         }
         Thread.sleep(100);
      }
   }

   public void stop() throws Exception {
      pauseAnimation();
   }

   public void run(String what) throws Exception {
      // TODO Auto-generated method stub
   }

   public void setOptions(String[] args) {
      // TODO Auto-generated method stub
   }

   public double getDoubleValue(String name) {
      // TODO Auto-generated method stub
      return 0;
   }

   public int getIntValue(String name) {
      // TODO Auto-generated method stub
      return 0;
   }

   public String getObjectValue(String name) {
      // TODO Auto-generated method stub
      return null;
   }

   public String[] getOptions() {
      // TODO Auto-generated method stub
      return null;
   }

   public String getStringValue(String name) {
      // TODO Auto-generated method stub
      return null;
   }

   public void initialize(JFrame frame, String[] args↩
      ) throws Exception {
      // TODO Auto-generated method stub
   }
}
```

```
package control;

import java.awt.Rectangle;
import java.io.FileNotFoundException;
import java.io.IOException;

import javax.swing.UIManager;

import view.WelcomeDialog;

/**Responsible for starting the program.
 * @author Jens Peter Träff
 *
 */
public class Driver {
   public static void main(String[] args) throws ↩
      Exception{

      {
         // first we set the LookAndFeel to the ↩
            operation system's default.
         try
         {
            UIManager.setLookAndFeel(UIManager.↩
               getSystemLookAndFeelClassName());
         } catch (Exception e)
         {
         }


         // Make a frame and show it
         LO_frame frame = new LO_frame();
```

```
            //frame.setVisible(true);

            Thread.sleep(3000);

            //frame.getViz().runFromStart();

            //frame.getViz().step(20);
        }


    }
}
```

```
package control;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLayeredPane;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.Timer;


import view.CheckDialog;
import view.InfoWindow;
import view.Java_source_code;
import view.Jeliot_Panel;
import view.Status_Panel;
import view.Text_Panel;
import view.WelcomeDialog;

import model.JeliotLOVisualization;
import model.Model;
import model.Visualization;

/** The LO_frame act as the root of the GUI, and listens to all the panels, buttons
 * and menus. It places a textpanel in the left part of a nested splitpane, a jeliotpanel in
 * the right, and a statuspanel in the south panel
 * @author Jens Peter Träff
 *
 */
public class LO_frame extends JFrame implements ActionListener, KeyListener, MouseListener {


   private Model model;
   private Text_Panel textPanel;
   private JSplitPane center= new JSplitPane();
   private JSplitPane south = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
   private Status_Panel statusP;
   private Jeliot_Panel jeliotP;
   private Visualization viz = new JeliotLOVisualization();

   /**Constructs the GUI and sets up all the panels
    * add itself as listener to all member panels.
    * @throws Exception
    */
   public LO_frame() throws Exception{
      model=new Model(this);

//    model.load_Text("C:/test2.txt");
//    model.Display_text();
//    model.load_step_explanation("C:/test3.txt");
      initFrame();
      //Visualization viz = new JeliotLOVisualization();
      String[] Config=new String[0];
      JComponent vis=viz.initializeVisualization(Config);
      //JSplitPane jeliot= new JSplitPane();


      //vis.setSize(new Dimension((int) (this.getWidth()), (int) (this.getHeight())));
      jeliotP=new Jeliot_Panel(new Dimension(this.getWidth(), (int) (this.getHeight()*0.9)), model, this);
      statusP=new Status_Panel(new Dimension(this.getWidth(), (int) (this.getHeight()*0.05)), model, this);
      textPanel= new Text_Panel(this);

      center.setLeftComponent(textPanel);
      center.setRightComponent(jeliotP);
      setLayout(new BorderLayout());
      //center.setLayout(new GridLayout(1,2));

      //set up the menubar
      JMenuBar menub = new JMenuBar();
      setJMenuBar(menub);
      JMenu menuf = new JMenu("File");
      JMenu menue = new JMenu("Functions");
      JMenu menuh = new JMenu("Help");
   // JMenu sub = new JMenu("Open Demo");
      menub.add(menuf);
      menub.add(menue);
      menub.add(menuh);

      JMenuItem knowledge = new JMenuItem("Check my knowledge");

      JMenuItem tutorial = new JMenuItem("About");
      JMenuItem help = new JMenuItem("Help");
      JMenuItem open = new JMenuItem("Load");
      menue.add(knowledge);
      menuh.add(tutorial);
      menuh.add(help);
      menuf.add(open);
      open.addActionListener(this);
      help.addActionListener(this);
      tutorial.addActionListener(this);
      knowledge.addActionListener(this);



      //JComponent visComp = viz.↩
         initializeVisualization(Config.DEFAULT_ARGS↩
         );
      //add visComp to your GUI
            //viz.load("C:/Average.java");  //↩
               programFile.getCanonicalPath()
      JScrollPane jscroll = new JScrollPane(vis);
      //add the panel to the frame
      jeliotP.setBackground(Color.BLUE);
      jeliotP.setLayout(new BorderLayout());
      jeliotP.add(jscroll, BorderLayout.CENTER);
      jeliotP.addKeyListener(this);

//    jeliotP.add(vis);


      //center.add(jCode);
//    center.add(textPanel);
//    center.setRightComponent(jeliotP);
      //ensure the dividerLocation is set right ↩
         always
      int location1 = (int) (this.getWidth()*0.5);
      center.setDividerLocation(location1);
   // center.setPreferredSize(new Dimension(this.↩
      getWidth(), this.getHeight()));
      int location = (int) (this.getHeight()*0.98) ;
      south.setDividerLocation( location);
      textPanel.addKeyListener(this);
      south.setRightComponent(statusP);
      south.setLeftComponent(center);

      //south.getLeftComponent().setPreferredSize(new↩
          Dimension(this.getWidth(), (int) (this.↩
         getHeight()*0.05)));
```

```
//      add(textPanel, BorderLayout.WEST);
        add(south, BorderLayout.CENTER);
        //add(statusP, BorderLayout.SOUTH);
        center.addKeyListener(this);
        south.addKeyListener(this);
//      Timer time = new Timer (100, this);
//      time.setActionCommand("time");
//      time.start();
        //textPanel.displaySteps();
        textPanel.addMouseListener(this);
        statusP.addMouseListener(this);
        addMouseListener(this);
        pack();
        //set the frame size to maximized
        setExtendedState(Frame.MAXIMIZED_BOTH);
        this.setVisible(true);

    }


    /**Initializes the frame, specifies size, location↩
        and default close operation
     *
     */
    private void initFrame()
    {
        setTitle("LO");
        setSize(new Dimension(700, 700));
        setLocation(100, 100);
        setDefaultCloseOperation(EXIT_ON_CLOSE);


        //pack();
    }


    public Visualization getViz() {
        return viz;
    }




public Jeliot_Panel getJeliotP() {
      return jeliotP;
   }

public Model getModel(){
   return this.model;
}



@Override
public void actionPerformed(ActionEvent e) {
   if(e.getActionCommand().equals("Step")){
         try {
            System.out.println("here");
            statusP.requestFocusInWindow();
            model.stop_animation();
            model.forward_animation_one_step();
            statusP.requestFocusInWindow();
         } catch (Exception e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
         }
   }else if(e.getActionCommand().equals("Play")){
      Thread c=new Thread(new Runnable() {
           public void run() {//we have to run a different thread to utilize this
              try {
            statusP.requestFocusInWindow();
               model.play_animation();

         } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
         }
           }
         });
      c.start();



   } else if(e.getActionCommand().equals("Stop")){
      System.out.println("in␣stop");
      model.stop_animation();
      statusP.requestFocusInWindow();
   }

   else if (e.getActionCommand().equals("Restart")){
      try {
         model.restartAnimation();
         statusP.requestFocusInWindow();
         //model.rewind_animation_one_step();
      } catch (Exception e1) {
         // TODO Auto-generated catch block
         e1.printStackTrace();
      }
      }
      else if (e.getActionCommand().equals("About")){
         WelcomeDialog wel;
         InfoWindow w;
//       try {
            w=new InfoWindow("./doc/about.html", ".",↩
                null, "about");
            w.reload();
            w.setVisible(true);
////           wel = new WelcomeDialog(this);
////           wel.showIt();
//       } catch (FileNotFoundException e1) {
//
//          e1.printStackTrace();
//       }
      }
      else if (e.getActionCommand().equals("Help")){
         WelcomeDialog wel;
         InfoWindow w;
//       try {
            w=new InfoWindow("help.html", "./doc", ↩
               null, "help");
            w.reload();
            w.setVisible(true);

      }

      else if (e.getActionCommand().equals("Check␣my␣↩
         knowledge")){
         CheckDialog check;
         try {
            check = new CheckDialog(new JFrame(), ↩
               this);
```

```
        check.showIt();
      } catch (FileNotFoundException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
      }
    } else if (e.getActionCommand().equals("Load"))↩
      {
      try {
        model.open();
      } catch (IOException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
      }
    }

// //below is just for fun
// else if(e.getActionCommand().equals("time")){
//    System.out.println("jeppe");
//    if(this.jeliotP.count<10){
//    this.jeliotP.count++;
//    }else this.jeliotP.count=0;

  }


    //textPanel.displaySteps();

@Override
public void keyPressed(KeyEvent e) {

  if(e.getKeyChar()=='␣'){
    try {
      model.forward_animation_one_step();
    } catch (Exception e1) {
      // TODO Auto-generated catch block
      e1.printStackTrace();
    }
  }else if (e.getKeyCode()==KeyEvent.VK_BACK_SPACE){
    model.restartAnimation();
  } else if (e.getKeyCode()==KeyEvent.VK_ENTER){
    model.stop_animation();
    Thread c=new Thread(new Runnable() {
       public void run() {//we have to run a ↩
         different thread to utilize this
```

```
              try {
               model.play_animation();
         } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
         }
           }
         });
      c.start();
   } else if (e.getKeyCode()==KeyEvent.VK_ESCAPE){
      model.stop_animation();
   }

}


@Override
public void keyReleased(KeyEvent arg0) {

}


@Override
public void keyTyped(KeyEvent arg0) {
   // TODO Auto-generated method stub

}






public Text_Panel getTextPanel() {
   return textPanel;
}


@Override
public void mouseClicked(MouseEvent e) {
   // TODO Auto-generated method stub
   System.out.println("joo");
}

```

```java

@Override
public void mouseEntered(MouseEvent e) {
   // TODO Auto-generated method stub

}


@Override
public void mouseExited(MouseEvent e) {
   // TODO Auto-generated method stub

}


@Override
public void mousePressed(MouseEvent e) {
   System.out.println(e.getComponent().hasFocus());
   if(e.getComponent().equals(statusP)){
   statusP.requestFocusInWindow();
   }

}


@Override
public void mouseReleased(MouseEvent e) {
   // TODO Auto-generated method stub

}

}
```

```java
package view;

import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.Rectangle;

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSpinner;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JTextPane;
import javax.swing.ScrollPaneConstants;
import javax.swing.SwingUtilities;
import javax.swing.text.BadLocationException;
import javax.swing.text.MutableAttributeSet;
import javax.swing.text.Position;
import javax.swing.text.StyleConstants;
import javax.swing.text.StyledDocument;

import model.Model;

import control.LO_frame;

public class Text_Panel extends JPanel{

   private static final Font plainFont = new Font(↩
      Font.MONOSPACED, Font.PLAIN, 14);
// private static final Font highlightFont = new Font↩
    ("Serif", Font.ITALIC, 18);
   private LO_frame LOframe;
   private int endMarker;
// private StyledDocument doc;
   private JTextPane jtp;
   public JTextPane getJtp() {
      return jtp;
   }
   private JTextArea disp;
   private JScrollPane jsp;

   public Text_Panel(LO_frame LOframe) {

      setLayout(new BorderLayout());
      this.LOframe = LOframe;
      setPreferredSize(new Dimension((int) (LOframe.↩
         getWidth()*0.5), LOframe.getHeight()));
      setBounds(0, 0, (int) (LOframe.getWidth()*0.2),↩
          LOframe.getHeight());
      setupLabels();
   }

   /**constructs the text area and scrollpane
    *
    */
   public void setupLabels()
   {
      jtp=new JTextPane();
      disp=new JTextArea();
      jtp.setPreferredSize(new Dimension(this.↩
         getWidth(), this.getHeight()));
      //jtp.setBounds(0, 0, 200, this.getHeight());
      //doc = jtp.getStyledDocument();//find out what↩
          styled document does
      jtp.setEditable(false);
      jsp=new JScrollPane(jtp);
      jsp.setBounds(0, 0, this.getWidth(), this.↩
         getHeight());
      //jtp.scrollRectToVisible()
      add(jsp, BorderLayout.CENTER);
      jtp.addMouseListener(LOframe);
      jtp.addKeyListener(LOframe);

//    addStylesToDocument(doc);

}

   public int getEndMarkers() {
      return endMarker;
   }

   /**Displays the explanatory text
    *
    */
   public void displayExplanation(){
      disp.append(LOframe.getModel().description +"\n↩
         ");
      jtp.setFont(plainFont);
      jtp.setText(disp.getText());
//    endMarkers=new int[LOframe.getModel().↩
   jeliotSteps[LOframe.getModel().jeliotSteps.length↩
   -1]];
//    int endOfPlainText=0;
//    try {
//       doc.remove(0, doc.getLength());
//       doc.insertString( 1,LOframe.getModel().description +"\n", null);
//       endOfPlainText=doc.getEndPosition().getOffset();
//       doc.insertString(endOfPlainText, "Now follows the Animation and explanation \n", null); //text for the instructor to specify?
//       //setJTextPaneFont(jtp, plainFont,Color.black,  endOfPlainText,doc.getEndPosition().getOffset(), false);
//    } catch (BadLocationException e) {
//       // TODO Auto-generated catch block
//       e.printStackTrace();
//    }
//    int startOfStepText=doc.getEndPosition().getOffset(); //marks the end of the plain text, and where the explanations are supposed to start, might be done more smartly to allow interchangeability...
//    endMarkers[0]=endOfPlainText;
//    endMarkers[1]=startOfStepText;
//    //setJTextPaneFont(jtp,plainFont,Color.black, 0, endOfPlainText, false);
   }


   /**This method displays the step description at the current step.
    *
    *
    */
   public void displaySteps() {
      Model local=LOframe.getModel();
      if(local.currentStep==1 || local.currentStep>local.jeliotSteps[local.getActualJeliotStep()]){
         if(local.currentStep!=1){
            local.setActualJeliotStep(local.getActualJeliotStep()+1);
         }
            disp.append(local.stepsDescription[local.getActualJeliotStep()] + "\n");
            jtp.setText(disp.getText());
            String breakk="";
            //ensure that the newest text is 2/3 up the page
            for(int c=0; c<10; c++){
               breakk+="\n";
            }
            jtp.setText(disp.getText()+breakk);
      }

      int last = jtp.getText().length();
//    try {
//       System.out.println(jtp);

      System.out.println(jtp.getSize());
//    jtp.scrollRectToVisible(jtp.modelToView(last));
      jtp.scrollRectToVisible(new Rectangle(0, last, 2,2));
      jtp.setCaretPosition(last);
//    catch (javax.swing.text.BadLocationException e)
//    {System.err.println("Error setting caret position when writing\n" +
//             "\n");}

   }

/**Removes the old text from the textpane
 * used when restarting animation.
 */
public void removeStepText(){
   jtp.setText("");
   disp.setText("");
   displayExplanation();
}


/**This method changes the font of text between offset and length. currently not used
 * @param jtp
 * @param font
 * @param c
 * @param offset
 * @param length
 * @param highlight
 */
```

```
public static void setJTextPaneFont(JTextPane jtp, ↩
   Font font, Color c, int offset, int length, ↩
   boolean highlight) {
    // Start with the current input attributes for ↩
       the JTextPane. This
    // should ensure that we do not wipe out any ↩
       existing attributes
    // (such as alignment or other paragraph ↩
       attributes) currently
    // set on the text area.
    MutableAttributeSet attrs = jtp.↩
       getInputAttributes();

    // Set the font family, size, and style, based on↩
        properties of
    // the Font object. Note that JTextPane supports ↩
       a number of
    // character attributes beyond those supported by↩
        the Font class.
    // For example, underline, strike-through, super-↩
        and sub-script.
    StyleConstants.setFontFamily(attrs, font.↩
       getFamily());
    StyleConstants.setFontSize(attrs, font.getSize())↩
       ;
    StyleConstants.setItalic(attrs, (font.getStyle() ↩
       & Font.ITALIC) != 0);
    StyleConstants.setBold(attrs, (font.getStyle() & ↩
       Font.BOLD) != 0);

    // Set the font color
    StyleConstants.setForeground(attrs, c);

    //set the background color (highlighting)
 /*    if(highlight){
      StyleConstants.setBackground(attrs, Color.↩
         YELLOW);}
    else{
      StyleConstants.setBackground(attrs, Color.WHITE↩
         );
    }
   */

    // Retrieve the pane's document object
```

```java
    StyledDocument doc = jtp.getStyledDocument();

    // Replace the style for the entire document. We exceed the length
    // of the document by 1 so that text entered at the end of the
    // document uses the attributes.
    doc.setCharacterAttributes(offset,length , attrs, false);
}

public JTextArea getDisp() {
   return disp;
   // TODO Auto-generated method stub

}


}
```

```java
package view;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;

import javax.swing.JButton;
import javax.swing.JPanel;

import control.LO_frame;

import model.Model;

/**This panel holds the buttons for controlling the animation
 * and is where future information related to animation control should be displayed to
 * the user.
 * @author Jens Peter Träff
 *
 */
public class Status_Panel extends JPanel {


   public Status_Panel(Dimension dimension, Model model, LO_frame LO_frame) {
      setPreferredSize(dimension);
      setLayout(new BorderLayout());
      JPanel x = new JPanel();
      JButton forward= new JButton("Step");
      JButton rewind= new JButton("Restart");
      JButton play = new JButton("Play");
      JButton stop = new JButton("Stop");
      x.add(rewind);
      x.add(play);
      x.add(stop);
      x.add(forward);
      this.add(x, BorderLayout.WEST);
      stop.addActionListener(LO_frame);
      rewind.addActionListener(LO_frame);
      play.addActionListener(LO_frame);
      forward.addActionListener(LO_frame);
      x.addKeyListener(LO_frame);
      this.addKeyListener(LO_frame);

   }

}
```

```
package view;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;

import javax.swing.JLabel;
import javax.swing.JPanel;



import model.Model;
import control.LO_frame;

public class Jeliot_Panel extends JPanel {

    public Jeliot_Panel(Dimension dimension, Model model, LO_frame LO_frame) {
        setPreferredSize(dimension);
```

```
      setBackground(Color.white);
      this.addMouseListener(LO_frame);
      //model.getJeliot().initialize(this);
   }
  }
```

```
/*
 * Created on 28.10.2004
 *
 * To change the template for this generated file go ↩
    to
 * Window - Preferences - Java - Code Generation - ↩
    Code and Comments
 */
package view;

import java.awt.Image;
import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.util.ResourceBundle;

import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.event.HyperlinkEvent;
import javax.swing.event.HyperlinkListener;

import jeliot.util.DebugUtil;
import jeliot.util.ResourceBundles;

/**
 * When creating a subclass of infoWindow you should ↩
    create a public
 * constructor that populates the udir and fileName ↩
    fields and calls
 * reload() method.
 *
 * @author Niko Myller
 */
public class InfoWindow extends JFrame implements ↩
   HyperlinkListener {

    /**
```

```
     * The resource bundle for gui package.
     */
    static protected ResourceBundle messageBundle = ↩
       ResourceBundles
            .getGuiMessageResourceBundle();

    /**
     * The pane where helping information will be ↩
        shown.
     */
    protected JEditorPane editorPane = new ↩
       JEditorPane();

    /**
     * The pane that handles the scrolling of the ↩
        editor pane showing the content.
     */
    protected JScrollPane jsp;

    /**
     * User directory where Jeliot was loaded.
     */
    protected String udir;

    /**
     * File name where the content should be read.
     */
    protected String fileName;

    /**
     * constructs the HelpWindow by creating a JFrame↩
        .
     * Sets inside the JFrame JScrollPane with ↩
        JEditorPane editorPane.
     * Sets the size of the JFrame as 400 x 600
     *
     * @param fileName file where the content is ↩
        loaded
     * @param udir directory of the current ↩
        invocation
     * @param icon Icon to be shown in the upper ↩
        right corner of the window.
     * @param title title of the JFrame
     */
    public InfoWindow(String fileName, String udir, Image icon, String title) {
        super(title);

        this.fileName = fileName;
        this.udir = udir;

        editorPane.setEditable(false);
        editorPane.addHyperlinkListener(this);

        jsp = new JScrollPane(editorPane);
        jsp.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
        getContentPane().add(jsp);

        setIconImage(icon);

        reload();
        setSize(600, 600);

    }

    /**
     *
     */
    public void reload() {
        try {
            File f = new File(udir, fileName);
            showURL(f.toURI().toURL());
        } catch (Exception e) {
            if (DebugUtil.DEBUGGING) {
                e.printStackTrace();
            }
        }
    }

    /**
     * Shows the given url in the editor pane.
     *
     * @param   url The document in the url will be showed in JEditorPane editorPane.
     */
    public boolean showURL(URL url) {
        try {
            editorPane.setPage(url);
        } catch (IOException e) {
            try {
                editorPane.setPage(Thread.↩
                   currentThread()
                        .getContextClassLoader().↩
                           getResource(fileName));
            } catch (IOException e1) {
                try {
                    editorPane.setPage(this.getClass↩
                       ().getClassLoader()
                            .getResource(fileName));
                } catch (IOException e2) {
                    try {
                        editorPane.setPage(Thread.↩
                           currentThread()
                                .↩
                                   getContextClassLoader↩
                                   ().getResource(↩
                                   fileName.↩
                                   substring(↩
                                   fileName.↩
                                   lastIndexOf("/") ↩
                                   + 1)));
                    } catch (IOException e3) {
                        try {
                            editorPane.setPage(this.↩
                               getClass().↩
                               getClassLoader()
                                    .getResource(↩
                                       fileName.↩
                                       substring(↩
                                       fileName.↩
                                       lastIndexOf("↩
                                       /") + 1)));
                        } catch (IOException e4) {

                            System.err.println(↩
                               messageBundle
                                    .getString("bad.↩
                                       URL")
                                    + "␣" + url);
                            if (DebugUtil.DEBUGGING) ↩
                               {
                                e1.printStackTrace();
                            }
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    }

    /* (non-Javadoc)
     * @see javax.swing.event.HyperlinkListener#hyperlinkUpdate(javax.swing.event.HyperlinkEvent)
     */
    public void hyperlinkUpdate(HyperlinkEvent e) {
        if (e.getEventType().toString().equals(
                HyperlinkEvent.EventType.ACTIVATED.toString())) {
            showURL(e.getURL());
        }
    }
}
```

```
package view;




import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JTextPane;
import javax.swing.text.BadLocationException;
import javax.swing.text.MutableAttributeSet;
import javax.swing.text.StyleConstants;
import javax.swing.text.StyledDocument;

import control.LO_frame;


/**This class constructs and sets up the Check ↩
   dialogue used for evaluation question
 *  Jens Peter Träff S082971
 *
 *
 */


public class CheckDialog extends JDialog implements ↩
   ActionListener{
   private JPanel mainPanel = new JPanel();
   private JTextPane helpt;
   private JTextField answer;
   private JPanel south;
   private JTextPane displayAnswer;
   private JPanel east;
   private String correctAnswer="";
   private LO_frame LOframe;
   /**
    * Constructs a new frame with the help dialog in ↩
       it
    * @param Frame
    * @throws FileNotFoundException
    */
   public CheckDialog(Frame frame, LO_frame LOframe) ↩
      throws FileNotFoundException
   {
      super(frame,"Check␣dialog",true);
      this.LOframe=LOframe;
      this.setLocation(100,100);
      this.getContentPane().add(mainPanel);
      helpt = new JTextPane();
      helpt.setBounds(100, 100, 300, 300);
      JScrollPane js=new JScrollPane(helpt);
      setupText();
      js.setPreferredSize(new Dimension(300, 300));
      mainPanel.setLayout(new BorderLayout());
      mainPanel.add(js, BorderLayout.CENTER);
      answer= new JTextField(20);
      answer.setEditable(true);

      displayAnswer= new JTextPane();
      displayAnswer.setPreferredSize(new Dimension(200, 100));
      JLabel information = new JLabel("evaluation of you answer:");

      //answer.setSize(100, 20);
      south=new JPanel();
      east= new JPanel();
      south.setLayout(new FlowLayout());
      east.setLayout(new BorderLayout());
      south.add(answer);
      east.setPreferredSize(new Dimension(200, 300));
      east.add(displayAnswer, BorderLayout.SOUTH);
      east.add(information, BorderLayout.CENTER);
      JButton jb=new JButton("check answer");
      south.add(jb);
      jb.addActionListener(this);
      this.add(south, BorderLayout.SOUTH);
      this.add(east, BorderLayout.EAST);
      this.pack();
   }

   public void showIt()
   {
      setVisible(true);
   }


   /**
    * The text itself
    * It loads the text from the basefilename in model and .chk extension.
    * @throws FileNotFoundException
    */

   private void setupText() throws ↩
      FileNotFoundException {
      Scanner input= new Scanner(new File(LOframe.↩
         getModel().getBaseFileName()+".chk"));
      String description="";
      boolean question=true;
      while (input.hasNextLine() && question){
         String q=input.nextLine();
         if(!q.equals("answer")){
            description += q;
            description += "\n";
         } else {question=false;}


      }
      //Scanner input1= new Scanner(new File(LOframe.↩
         getModel().getBaseFileName()+".ans"));
      correctAnswer="";
      while (input.hasNextLine()){
         correctAnswer += input.nextLine();
      }
      StyledDocument doc = helpt.getStyledDocument();
      MutableAttributeSet attrs = helpt.↩
         getInputAttributes();
      StyleConstants.setFontFamily(attrs, "italic");
       StyleConstants.setFontSize(attrs, 16);
      // doc.setCharacterAttributes(0, 300, attrs, ↩
         false);
      try {
         doc.insertString(0,description,
               attrs)


         ;
      } catch (BadLocationException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }

   @Override
   public void actionPerformed(ActionEvent e) {
      if(e.getActionCommand().equals("check␣answer"))↩
         {
         String userAnswer=answer.getText();
         if(userAnswer.equals(correctAnswer)){
            displayAnswer.setText("the␣answer␣is␣true↩
               ,␣you␣are␣ready␣to␣move␣on");
         }else {
            displayAnswer.setText("your␣answer␣is␣not↩
               ␣quite␣right,␣try␣taking␣another␣look↩
               ␣at␣the␣animations");
            }
         }
      }


}
```

Bibliography

[1]

[2]

[3]

[4] Mordechai Ben-Ari, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. A decade of research and development on program animation: The jeliot experience. 2011.

[5] Ronit Ben-Bassat Levy and Mordechai Ben-Ari. A survey of research on the jeliot program animation system. 2009.