PatternsEditor

Release Report

Rocking Machines

Zonios Christos:2194

Poulos Vasileios:2805

Pasoi Sofia:2798

VERSIONS HISTORY

Date	Version	Description	Author
19/04/2018	1	Implementation of User Stories 1,2,3 and 4.	Rocking Machines

Introduction

This document provides information concerning the firs release of the project.

1.1 Purpose

A software development pattern defines a general reusable solution to a commonly occurring software development problem within a particular context. Patterns constitute a significant asset of the software engineering community. Amongst the very first approaches we have the GoF design patterns catalog that concerns best OO development practices. Then, there are also regular conferences (e.g. PLoP, EuroPLoP) that take place for more than 20 years and whose main topic is the identification of new patterns and pattern languages (the term pattern language is typically used to refer to a set of related patterns). Patterns are formally specified in terms of pattern templates. So far, several pattern templates have been proposed in the literature.

The main goal of this project is to develop a PatternsEditor, an application that makes pattern writting easier, especially for young inexperienced pattern writers. At a glance, PatternsEditor shall allow a patterns writer to prepare a new pattern based on well known templates change the structure of an existing pattern by switching between these templates, and generate actual pattern documents in well known formats (simple text, Latex), and so on.

1.2 Document Structure

The rest of this document is structured as follows. Section 2 specifies the acceptance tests that have been employed for this release of the project. Section 3 specifies the main design concepts for this release of the project.

2 Acceptance Tests

Acceptance testing is a test conducted to determine if the requirements of a specification or contract are met. For the user stories included in this releases specify below corresponding tests using a typical tabular form.

2.1 Tests for User Story **<1>**

Test ID	Acceptance Test [US1]
Class	PatternLanguage
Test Class	shouldMakePLObject()
Test Method	PatternLanguage(String name)and generateDefaultTitle()
Description	Test the construction of a PatternLanguage object with and without a given name, against an expected PatternLanguage object. Input 1: User does not give name Input2: User gives a name Expected Output: Default name Expected Output: The name that was given

2.2 Tests for User Story **<2>**

Test ID	Acceptance Test [US2]
Class	TemplateFactory`() (in class TemplateFactory)`
Test Class	AcceptanceTests
Test Method	shouldMakeTemplates()
Description	Test the createTemplate() method of TemplateFactory for each different template to see if the contents of the created patterns follow the expected template structure. Input: The name of template's type(ex. Gang Of 4) Expected Output: Creation of the template's type that user chooses

Test ID	Acceptance Test [US2]
Class	`add() (in class PatternComposite)`
Test Class	AcceptanceTests

Test Method	Add()
Description	Test the add() method of a PatternLanguage object to see if the object contents change expectedly. Specifically checks if object was added in componentList. Input: composite.add(component) Expected Output: Component is added to the ComponentsList

2.3 Tests for User Story **<3>**

Test ID	Acceptance Test [US3]
Class	`remove(in class PatternComposite)`
Test Class	AcceptanceTests
Test Method	`remove()`
Description	Tests the remove() method of a PatternLanguage object to see if the object contents change expectedly. Specifically checks if item was removed from componentList. Input: composite.add(component) Expected Output: Component is removed from the ComponentsList

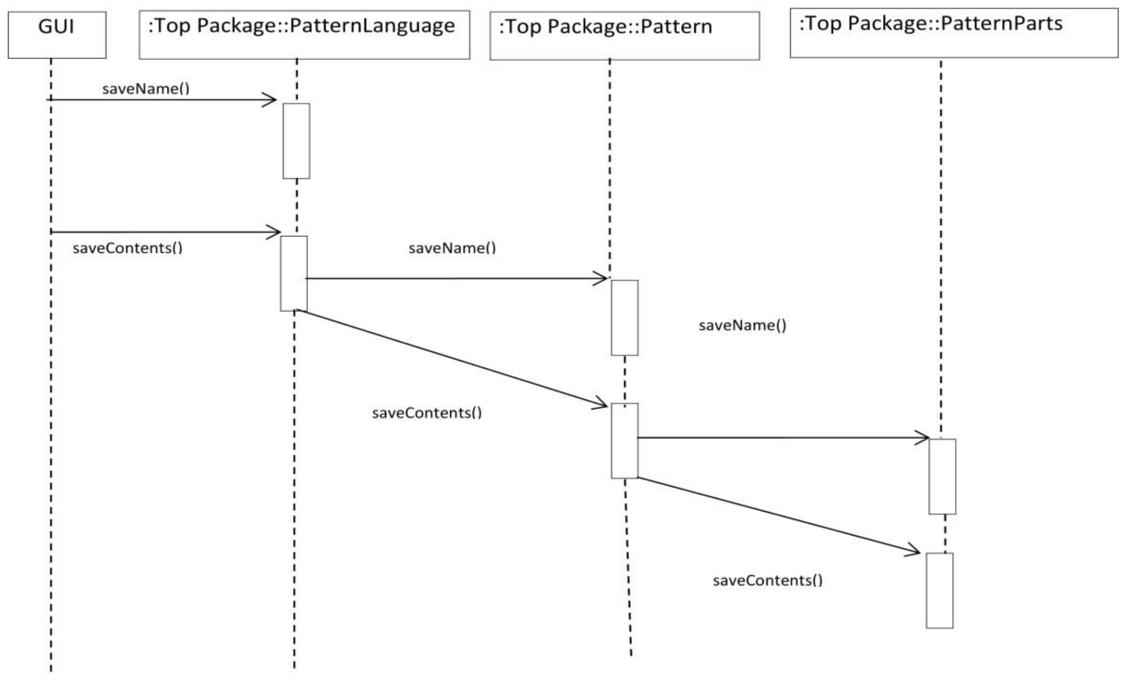
2.4 Tests for User Story **<4>**

Test ID	Acceptance Test [US4]
Class	setContents(in class PatternPart)
Test Class	AcceptanceTests
Test Method	`shouldSetContents()`
Description	Test the setContents() method of a Pattern object to see if the respective contents change expectedly. Input: Pattern's(object) name and contents Expected Output: The name and the contents that were choosen

3 Design

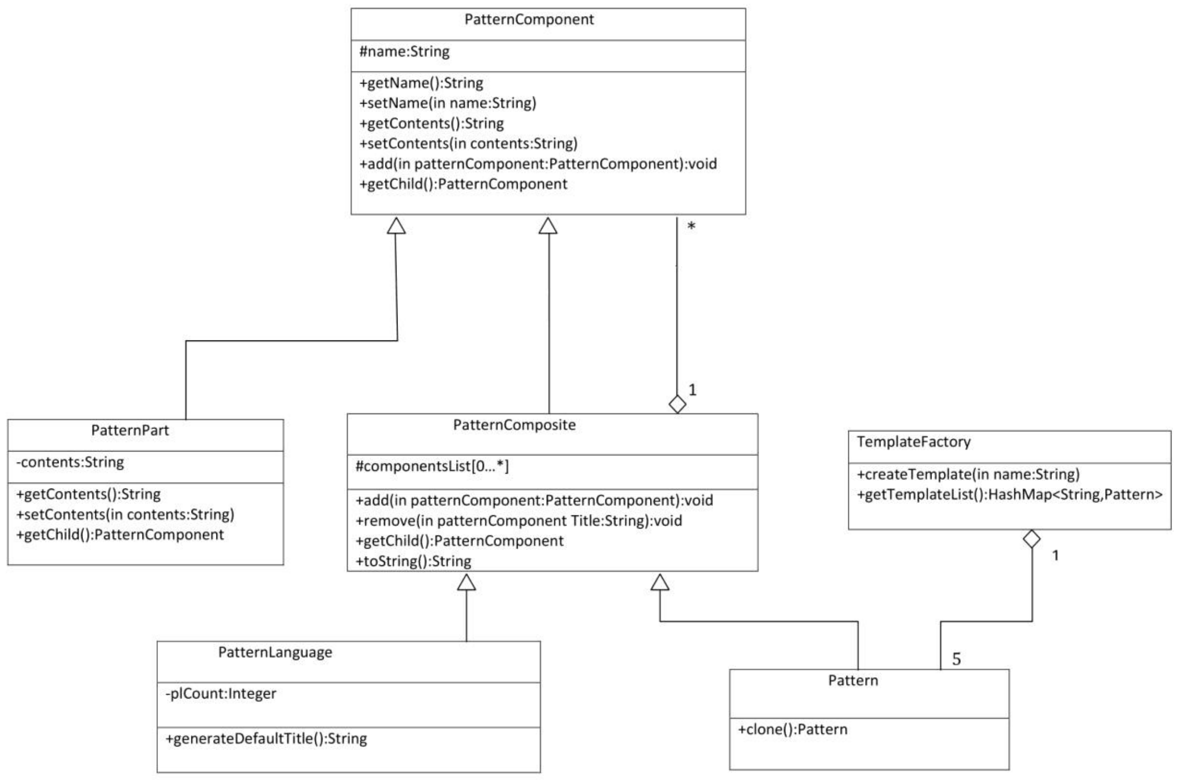
3.1 Architecture

UML Package Diagram



3.2 Design

UML Class Diagram


PatternComponent
#name:String
+getName():String
+setName(in name:String)
+getContents():String
+setContents(in contents:String)
+add(in patternComponent:PatternComponent):void
+getChild():PatternComponent
*
1
PatternPart
-contents:String
+getContents():String
+setContents(in contents:String)
+getChild():PatternComponent
PatternComposite
#componentsList[0...*]
+add(in patternComponent:PatternComponent):void
+remove(in patternComponent Title:String):void
+getChild():PatternComponent
+toString():String
TemplateFactory
+createTemplate(in name:String)
+getTemplateList():HashMap<String,Pattern>
1
5
PatternLanguage
-plCount:Integer
+generateDefaultTitle():String
Pattern
+clone():Pattern

CRC cards

Class Name: PatternComponent	
Responsibilities: ▪ Knows the name of a pattern or the name f a pattern language.	**Collaborations:** ▪ PatternComposite

Class Name: PatternPart	
Responsibilities: ▪	**Collaborations:** ▪ PatternComponent ▪ PatternComposite

Class Name: PatternComposite

Responsibilities:	Collaborations:
▪ Creates an ArrayList<PatternComponent>. ▪ Adds a PatternComponet object in the ArrayList<PatternComponent>. ▪ Removes a PatternComponet object of the ArrayList<PatternComponent>.	▪ PatternComponent

Class Name: TemplateFactory	
Responsibilities:	**Collaborations:**
▪ Creates prototypes of 5 different types of template. ▪ Adds the prototypes in a HashMap.	▪ Pattern.

Class Name: Pattern	
Responsibilities:	**Collaborations:**
▪ Creates an exact copy of a template.	▪ PatternComponent ▪ PatternComposite ▪ TemplateFctory

Class Name: PatternLanguage	
Responsibilities:	**Collaborations:**
▪ Sets a default name in the created pattern language, in case the user does not give one.	▪ PatternComponent ▪ PatternComposite

```
package datamodel;

/*
 *      PatternComponent class that defines the basic methods of the composite
 *      structure, along with default TRIVIAL IMPLEMENTATIONS.
 *          |    think this means: define methods, some may do nothing and that's
 *          |    because we need child classes to inherit them.
 *          |
 *          |    "Trivial" typically refers to an implementation that demonstrates
 *          |    the relevant functionality and no more.
 */

public class PatternComponent implements Cloneable
{
        protected String name;

        public PatternComponent (String name)
    {
        this.name = name;
    }

        /**
     * @return  components name
     */
        public String getName() {
                return this.name;
        }

    public void setName(String name) {
        this.name = name;
    }

    public String getContents() {return null;}

    public void setContents(String contents) {}

    //public void setContents(String contents, int x){}

    //TODO: Should write this.name to file (V2)
    //public void saveName(){}

    //TODO: Should write contents to file (V2)
    //public void saveContents() {}

        public void add(PatternComponent component){}

        //public void remove(){}

    public PatternComponent getChild(){
        return new PatternComponent("");
    }
}


package datamodel;

public class PatternPart extends PatternComponent

{
    private String contents;

    /**
     * Constructor.
     *
     * @param name - name of the pattern part
     * @param contents - contents/description of the pattern part
     */
    public PatternPart(String name, String contents) {
        super(name);
        this.contents = contents;
    }
```

```java
    /**
     * Constructor.
     *
     * @param name - name of the pattern part
     */
    public PatternPart(String name) {
                super(name);
        }

    /** Return contents, passed to the constructor.  */
    @Override
    public String getContents() {
        return contents;
    }

    /** Set the contents field. */
    @Override
    public void setContents(String contents) {
        this.contents = contents;
    }

    //TODO: Write contents to output file (V2)
    /*
    public void saveContents(){
        return;
    }
    */

    /** Return string representation of the pattern part. */
    @Override
        public String toString() {
        return this.name + ":\n" + this.contents;
    }
}


package datamodel;

import java.util.ArrayList;

public abstract class PatternComposite extends PatternComponent
{

    private Integer count;      // so no two patters are created with the same name

    /**
     * ArrayList of components.
     */
    protected ArrayList<PatternComponent> componentsList;

    /**
     * Calling parent constructor, initializing array.
     *
     * @param name Name of the composite
     */
    public PatternComposite(String name)
    {
        super(name);
        this.count = 0;
        this.componentsList = new ArrayList<>();
    }

    /**
     * Adds a PatternComponent item in ArrayList
     *
     * @param component new component added to the list
     */
    @Override
    public void add(PatternComponent component)
    {
        for (PatternComponent i: this.componentsList) {
            if (i.getName().equals(component.getName())) {
                this.count++;
                component.setName(component.getName() + "-" + Integer.toString(this.count));
            }
        }
        this.componentsList.add(component);
    }

    /**
     * Removes a PatternComponent item from the ArrayList
     *     if component List contains the item requested, remove it.
     *
     * @param patternComponentTitle pattern components name
     */
    public void remove(String patternComponentTitle)
    {
        this.componentsList.removeIf((PatternComponent p) -> p.getName().equals(patternComponentTitle));

    }

    public ArrayList<PatternComponent> getComponentsList() {
        return componentsList;
    }

    //TODO: will be implemented in Release 2.0
    //public abstract void decorateComponents(DecoratorAbstractFactory decoratorFactory);

    @Override
    public PatternComponent getChild() {
        return super.getChild();
    }

    @Override
    public String toString() {
        String list="";
        for(int i=0;i<componentsList.size();i++){
            list=componentsList.get(i).toString();
        }
        return list;
    }
}


package datamodel;

import java.util.HashMap;

public class TemplateFactory
{
    /* Map holding Pattern prototypes and mapping them to their names */
    private HashMap<String, Pattern> templatesList;

    /**
     * Constructor
     *
     * Initializes patterns by creating their names and their individual parts,
     * then uses them as prototypes for the templates list, mapping each pattern to its name
     */
    public TemplateFactory()
    {
        /* Initialize map of templates */
        templatesList = new HashMap<>();

        /* Create the prototype Patterns and add to them each part/section */
        Pattern micro = new Pattern("Micro-Pattern");
        micro.add(new PatternPart("Name", "What shall this pattern be called by practitioners?"));
        micro.add(new PatternPart("Template", "Which template is followed for the pattern specification ?"));
        micro.add(new PatternPart("Problem", "What is motivating us to apply this pattern? "));
        micro.add(new PatternPart("Solution", " How do we solve the problem?"));

        Pattern inductive = new Pattern( "Inductive Mini-Pattern");
        inductive.add(new PatternPart("Name", "What shall this pattern be called by practitioners?"));
        inductive.add(new PatternPart("Template", "Which template is followed for the pattern specification ? "));
        inductive.add(new PatternPart("Context", " What are the assumed environment or a priori assumptions for applying
this pattern?"));
        inductive.add(new PatternPart("Forces", "What are the different design motivations that must be balanced? ?"));
        inductive.add(new PatternPart("Solution", "How do we solve the problem? "));

        Pattern deductive = new Pattern( "Deductive Mini-Pattern");
        deductive.add(new PatternPart("Name", "What shall this pattern be called by practitioners?"));
        deductive.add(new PatternPart("Template", "Which template is followed for the pattern specification ? "));
        deductive.add(new PatternPart("Problem", "  What is motivating us to apply this pattern? "));
        deductive.add(new PatternPart("Solution", ": How do we solve the problem? ?"));
        deductive.add(new PatternPart("Benefits", " What are the potential positive outcomes of applying this pattern?
"));
        deductive.add(new PatternPart("Consequences", "What are potential shortcomings and consequences of applying this
pattern? \n"));

        Pattern gof = new Pattern( "Gang-Of-Four Pattern");
        gof.add(new PatternPart("Name", "What is the pattern called? "));
        gof.add(new PatternPart("Template", " Which template is followed for the pattern specification ?  "));
        gof.add(new PatternPart("Pattern Classification", " Is the pattern creational, structural, or behavioral? "));
        gof.add(new PatternPart("Intent", "What problem does this pattern solve?"));
        gof.add(new PatternPart("Also Known As", " What are other names for this pattern? "));
        gof.add(new PatternPart("Motivation", "What is an example scenario for applying this pattern? "));
        gof.add(new PatternPart("Applicability", " When does this pattern apply? "));
        gof.add(new PatternPart("Structure", "Which are the classes of the objects in this pattern?"));
        gof.add(new PatternPart("Participants", " What are the objects that participate in this pattern? "));
        gof.add(new PatternPart("Collaborations", " How do these objects interoperate? "));
        gof.add(new PatternPart("Consequences", " What are the trade-offs of using this pattern? "));
        gof.add(new PatternPart("Implementation", "Which techniques or issues arise in applying this pattern? "));
        gof.add(new PatternPart("Sample Code", "What is an example of the pattern in source code?"));
        gof.add(new PatternPart("Known Uses", " What are some examples of real systems using this pattern? "));
        gof.add(new PatternPart("Related Patterns", " What other patterns from this pattern collection are related to
this pattern?"));

        Pattern sop = new Pattern( "System Of Patterns Template");
        sop.add(new PatternPart("Name", "What is the pattern called? "));
        sop.add(new PatternPart("Template", " Which template is followed for the pattern specification ? "));
        sop.add(new PatternPart("Also Known As", " What are other names for this pattern? "));
        sop.add(new PatternPart("Example", "What is an example of the need for this pattern? "));
        sop.add(new PatternPart("Context", " When does this pattern apply? "));
        sop.add(new PatternPart("Problem", "What is the problem solved by this pattern?"));
        sop.add(new PatternPart("Solution", "What is the underlying principal underlying this pattern? "));
        sop.add(new PatternPart("Structure", "What objects are involved and related?"));
        sop.add(new PatternPart("Dynamics", " How do these objects collaborate? "));
        sop.add(new PatternPart("Implementation", "What are some guidelines for implementing this pattern? "));
        sop.add(new PatternPart("Example Resolved", "Show how the previous example is resolved using the pattern"));
        sop.add(new PatternPart("Variants", "What are important variations of this pattern? "));
        sop.add(new PatternPart("Known Uses", "What are real-world systems using this pattern? "));
        sop.add(new PatternPart("Consequences", " What are the benefits and liabilities of using this pattern? "));

        /* Add the prototypes to the list */
        templatesList.put(micro.getName(), micro);
        templatesList.put(inductive.getName(), inductive);
        templatesList.put(deductive.getName(), deductive);
        templatesList.put(gof.getName(), gof);
        templatesList.put(sop.getName(), sop);

    }

    /**
     * Clones a pattern prototype (a template) and returns the copy
     *
     * @param templateName name of the template
     * @return Pattern based on template prototype
     */
    public Pattern createTemplate(String templateName)
    {
        return templatesList.get(templateName).clone();
    }

    /**
     * Getter for the map of templates
     *
     * @return the template list
     */
    public HashMap<String, Pattern> getTemplatesList() {
        return templatesList;
    }


}



package datamodel;

public class Pattern extends PatternComposite implements Cloneable
{

    /**
     * Calling parent constructor.
     * <p>
     *      TODO: A pattern is a list of PatternPart objects with specific titles (which are dictated by the template)!
     *      TODO: and descriptions. You can find those in one of the provided pdf docs
     * </p>
     *
     * @param name Name of the composite
     */
    public Pattern(String name) {

        super(name);
    }

    /**
     * Creates a clone of a PatternComponent object (deep copy).
     * <p>
     *      Note :  There is a secondary implementation overriding clone() method of Object class
     *              I think it is better to use copy constructors or factory methods
     *              instead of overriding clone().
     * </p>
     *
     * @return PatternComponent clone.
     */
    @Override
    public Pattern clone()
    {
        Pattern newPattern = new Pattern(this.getName());
        for (PatternComponent p : this.componentsList){
            newPattern.componentsList.add(new PatternPart(p.getName(),p.getContents()));
        }
        return newPattern;
    }

    //TODO: (V2)
    //@Override
    //public void decorateComponents(DecoratorAbstractFactory decoratorFactory){}
}


package datamodel;

public class PatternLanguage extends PatternComposite
{
    private static Integer plCount = 0;

    public PatternLanguage()
    {
        super("Default");
    }
    /**
     * Overloaded constructor with default parameters
     *
     * @param name Pattern Language's name
     */
    public PatternLanguage(String name)
    {
        super(name);
        if (name == null || name.equals("") || name.isEmpty())
        {
            name = this.generateDefaultTitle(); // Should generate using a counter (e.g. Pattern Language #1, Pattern
Language #2, ...)
        }
        this.name = name;
    }

    /**
     * Generates a default pattern language title, based on a global counter of pattern languages
     * @return default title for the new pattern language
     */
    public String generateDefaultTitle()
    {
        plCount++;
        String title = "Pattern Language #" + Integer.toString(plCount);
        return title;
    }



    //TODO: Release 2.0
    //public void decorateComponents()
}
```

Acceptance Tests

```
package datamodel;

import java.util.ArrayList;

import static org.junit.jupiter.api.Assertions.assertEquals;

class AcceptanceTests{
    /*The rest of the acceptance tests are in PatternCompositeTest*/

    /**
     * Acceptance Test [US1]
     */
    @org.junit.jupiter.api.Test
    void shouldMakePLObject()
    {
        /*Test with no given name*/
        PatternLanguage new_pl = new PatternLanguage();
        assertEquals("Default",new_pl.getName());


        /*Test with given name*/
        PatternLanguage named_pl = new PatternLanguage("TestName");
        assertEquals("TestName",named_pl.getName());
    }

    /**
     * Acceptance Test [US2]
     */
    @org.junit.jupiter.api.Test
    void shouldMakeTemplates()
    {
        /*Initialize template Factory*/
        TemplateFactory tf = new TemplateFactory();

        /*Micro-Pattern Test*/
        ArrayList<PatternComponent> microList = new ArrayList<>();
        microList.add(new PatternPart("Name", "What shall this pattern be called by practitioners?"));
        microList.add(new PatternPart("Template", "Which template is followed for the pattern specification ?"));
        microList.add(new PatternPart("Problem", "What is motivating us to apply this pattern? "));
        microList.add(new PatternPart("Solution", " How do we solve the problem?"));

        Pattern micro = tf.createTemplate("Micro-Pattern");
        assertEquals("Micro-Pattern",micro.getName());

        for (int i = 0; i < micro.componentsList.size(); i++)
        {
            assertEquals(microList.get(i).toString(), micro.componentsList.get(i).toString());
        }

        /*Inductive Mini-Pattern Test*/
        ArrayList<PatternComponent> inductiveList = new ArrayList<>();
        inductiveList.add(new PatternPart("Name", "What shall this pattern be called by practitioners?"));
        inductiveList.add(new PatternPart("Template", "Which template is followed for the pattern specification ? "));
        inductiveList.add(new PatternPart("Context", " What are the assumed environment or a priori assumptions for
applying this pattern?"));
        inductiveList.add(new PatternPart("Forces", "What are the different design motivations that must be balanced?
?"));
        inductiveList.add(new PatternPart("Solution", "How do we solve the problem? "));

        Pattern inductive = tf.createTemplate("Inductive Mini-Pattern");
        assertEquals("Inductive Mini-Pattern",inductive.getName());

        for (int i = 0; i < inductive.componentsList.size(); i++)
        {
            assertEquals(inductiveList.get(i).toString(), inductive.componentsList.get(i).toString());
        }

        /*Deductive Mini-Pattern Test*/
        ArrayList<PatternComponent> deductiveList = new ArrayList<>();
        deductiveList.add(new PatternPart("Name", "What shall this pattern be called by practitioners?"));
        deductiveList.add(new PatternPart("Template", "Which template is followed for the pattern specification ? "));
        deductiveList.add(new PatternPart("Problem", "  What is motivating us to apply this pattern? "));
        deductiveList.add(new PatternPart("Solution", ": How do we solve the problem? ?"));
        deductiveList.add(new PatternPart("Benefits", " What are the potential positive outcomes of applying this
pattern?  "));
        deductiveList.add(new PatternPart("Consequences", "What are potential shortcomings and consequences of applying
this pattern? \n"));

        Pattern deductive = tf.createTemplate("Deductive Mini-Pattern");
        assertEquals("Deductive Mini-Pattern",deductive.getName());

        for (int i = 0; i < deductive.componentsList.size(); i++)
        {
            assertEquals(deductiveList.get(i).toString(), deductive.componentsList.get(i).toString());
        }

        /*Gang-Of-Four Pattern Test*/
        ArrayList<PatternComponent> gofList = new ArrayList<>();
        gofList.add(new PatternPart("Name", "What is the pattern called? "));
        gofList.add(new PatternPart("Template", " Which template is followed for the pattern specification ?  "));
        gofList.add(new PatternPart("Pattern Classification", " Is the pattern creational, structural, or behavioral?
"));
        gofList.add(new PatternPart("Intent", "What problem does this pattern solve?"));
        gofList.add(new PatternPart("Also Known As", " What are other names for this pattern? "));
        gofList.add(new PatternPart("Motivation", "What is an example scenario for applying this pattern? "));
        gofList.add(new PatternPart("Applicability", " When does this pattern apply? "));
        gofList.add(new PatternPart("Structure", "Which are the classes of the objects in this pattern?"));
        gofList.add(new PatternPart("Participants", " What are the objects that participate in this pattern? "));
        gofList.add(new PatternPart("Collaborations", " How do these objects interoperate? "));
        gofList.add(new PatternPart("Consequences", " What are the trade-offs of using this pattern? "));
        gofList.add(new PatternPart("Implementation", "Which techniques or issues arise in applying this pattern? "));
        gofList.add(new PatternPart("Sample Code", "What is an example of the pattern in source code?"));
        gofList.add(new PatternPart("Known Uses", " What are some examples of real systems using this pattern? "));
        gofList.add(new PatternPart("Related Patterns", " What other patterns from this pattern collection are related to
this pattern?"));

        Pattern gof = tf.createTemplate("Gang-Of-Four Pattern");
        assertEquals("Gang-Of-Four Pattern",gof.getName());

        for (int i = 0; i < gof.componentsList.size(); i++)
        {
            assertEquals(gofList.get(i).toString(), gof.componentsList.get(i).toString());
        }

        /*System Of Patterns Template Test*/
        ArrayList<PatternComponent> sopList = new ArrayList<>();
        sopList.add(new PatternPart("Name", "What is the pattern called? "));
        sopList.add(new PatternPart("Template", " Which template is followed for the pattern specification ? "));
        sopList.add(new PatternPart("Also Known As", " What are other names for this pattern? "));
        sopList.add(new PatternPart("Example", "What is an example of the need for this pattern? "));
        sopList.add(new PatternPart("Context", " When does this pattern apply? "));
        sopList.add(new PatternPart("Problem", "What is the problem solved by this pattern?"));
        sopList.add(new PatternPart("Solution", "What is the underlying principal underlying this pattern? "));
        sopList.add(new PatternPart("Structure", "What objects are involved and related?"));
        sopList.add(new PatternPart("Dynamics", " How do these objects collaborate? "));
        sopList.add(new PatternPart("Implementation", "What are some guidelines for implementing this pattern? "));
        sopList.add(new PatternPart("Example Resolved", "Show how the previous example is resolved using the pattern"));
        sopList.add(new PatternPart("Variants", "What are important variations of this pattern? "));
        sopList.add(new PatternPart("Known Uses", "What are real-world systems using this pattern? "));
        sopList.add(new PatternPart("Consequences", " What are the benefits and liabilities of using this pattern? "));

        Pattern sop = tf.createTemplate("System Of Patterns Template");
        assertEquals("System Of Patterns Template",sop.getName());

        for (int i = 0; i < sop.componentsList.size(); i++)
        {
            assertEquals(sopList.get(i).toString(), sop.componentsList.get(i).toString());
        }
    }

    /**
     * Acceptance Test [US2]
     * Check if item was added in componentList.
     */
    @org.junit.jupiter.api.Test
    void add()
    {
        PatternLanguage composite = new PatternLanguage("TestComposite");
        PatternComponent component = new PatternComponent("Component");
        composite.add(component);
        assertEquals(composite.getComponentsList().get(0).getName(),"Component");
    }

    /**
     * Acceptance Test [US3]
     * Check if item was removed from componentList.
     */
    @org.junit.jupiter.api.Test
    void remove()
    {
        PatternLanguage composite = new PatternLanguage("TestComposite");
        PatternComponent component = new PatternComponent("Component");
        composite.add(component);
        composite.remove("Component");
        assertEquals(true,composite.getComponentsList().isEmpty());

    }


    /**
     * Acceptance Test [US4]
     */
    @org.junit.jupiter.api.Test
    void shouldSetContents()
    {
        Pattern new_pattern = new Pattern("Pattern");
        new_pattern.add(new PatternPart("qwer","asdf"));
        new_pattern.componentsList.get(0).setContents("qwer");
        assertEquals("qwer",new_pattern.componentsList.get(0).getContents());
    }

}

Tests for bugs

package datamodel;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class TemplateFactoryTest {

    @Test
    void shouldMakeTemplates() {

        TemplateFactory tf = new TemplateFactory();
        assertNotEquals(tf.getTemplatesList().isEmpty(), 0);
    }

    @Test
    void shouldMakeCopy() {

        TemplateFactory tf = new TemplateFactory();
        PatternComponent newPattern = tf.createTemplate("Micro-Pattern");
        PatternComponent oldPattern = tf.getTemplatesList().get("Micro-Pattern");

        newPattern.setName("asdf");
        oldPattern.setName("qwer");

        assertNotEquals(oldPattern.getName(), newPattern.getName());

    }
    @Test
    void shouldMakeDeepCopy() {
        TemplateFactory tf = new TemplateFactory();
        Pattern newPattern = tf.createTemplate("Micro-Pattern");

        PatternPart newPart = (PatternPart) newPattern.getComponentsList().get(0);
        PatternPart oldPart = (PatternPart) tf.getTemplatesList().get("Micro-Pattern").getComponentsList().get(0);

        newPart.setName("Lorem ipsum");
        oldPart.setName("dolor sit amet");

        assertNotEquals(newPart.getName(), oldPart.getName());
    }

}

package datamodel;

import org.junit.jupiter.api.Test;

import java.util.ArrayList;

import static org.junit.jupiter.api.Assertions.*;

class PatternTest{

    /**
     * Check if clone makes a deep copy
     * Make sure that cloning makes a new object.
     * If original components name changes, copy's name should stay
     * the same
     */
    @Test
    void shouldClone() {

        Pattern oldPattern = new Pattern("newPattern");
        PatternPart part = new PatternPart("Lorem ipsum");
        oldPattern.add(part);

        /* Test that clone works correctly */
        Pattern newPattern = oldPattern.clone();
        assertEquals(oldPattern.getComponentsList().get(0).getName(), newPattern.getComponentsList().get(0).getName());
        assertNotEquals(oldPattern.getComponentsList().get(0), newPattern.getComponentsList().get(0));


    }

    @Test
    void shouldMakeDeepCopy() {

        Pattern oldPattern = new Pattern("newPattern");
        PatternPart part = new PatternPart("Lorem ipsum");
        oldPattern.add(part);

        Pattern newPattern = oldPattern.clone();

        /* Test that clone has different components after changing */
        newPattern.getComponentsList().get(0).setName("dolor sit");
        assertNotEquals(oldPattern.getComponentsList().get(0), newPattern.getComponentsList().get(0));

        /* Test that clone has same size components before adding */
        assertEquals(oldPattern.getComponentsList().size(), newPattern.getComponentsList().size());

        /* Test that clone has different size after adding components */
        newPattern.add(new PatternPart("amet"));
        assertNotEquals(oldPattern.getComponentsList().size(), newPattern.getComponentsList().size());
    }
}
```

GUI

```
package gui;
```

```java
import datamodel.Pattern;
import datamodel.PatternLanguage;
import datamodel.TemplateFactory;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.stage.Stage;

public class Main extends Application {

    private static Stage window;                            // The primaryStage
    private static Scene start;                             // Opening scene
    private static Scene plView;                            // Pattern Language view
    private static Scene patternView;                       // Pattern Edit view
    private static TemplateFactory templateFactory;         // TemplateFactory object used for initialization
    private static PatternLanguage pl;                      // The Pattern Language we're working on
    private static Pattern currentPattern;                  // The Pattern we're currently editing


    /**
     * Runs the application, creating a window with the Start scene and running initialize()
     * @param primaryStage the root Stage object
     */
    @Override
    public void start(Stage primaryStage) throws Exception {
        window = primaryStage;

        FXMLLoader loader = new FXMLLoader(getClass().getResource("start.fxml"));
        Parent root = loader.load();

        window.getIcons().add(new Image("/assets/img/icon.png"));
        window.setTitle("Rocking Machines - Patterns Editor");
        start = new Scene(root, 800, 600);
        window.setScene(start);
        window.show();

        this.initialize();

    }

    public static void main(String[] args) {
        launch(args);
    }

    /** Creates a new TemplateFactory object */
    private void initialize() {
        Main.templateFactory = new TemplateFactory();
    }

    /*************************
     * Getters and Setters   *
     *************************/

    /**
     *
     * @return window
     */
    public static Stage getWindow() {
        return window;
    }

    public static Scene getStart() {
        return start;
    }

    public static Scene getPlView() {
        return plView;
    }
    public static void setPlView(Scene plViewScene) {
        Main.plView = plViewScene;
    }

    public static Scene getPatternView() {
        return patternView;
    }

    public static void setPatternView(Scene patternView) {
        Main.patternView = patternView;
    }

    public static TemplateFactory getTemplateFactory() {
        return templateFactory;
    }

    public static PatternLanguage getPl() {
        return pl;
    }

    public static void setPl(PatternLanguage pl) {
        Main.pl = pl;
    }

    public static Pattern getCurrentPattern() {
        return currentPattern;
    }

    public static void setCurrentPattern(Pattern currentPattern) {
        Main.currentPattern = currentPattern;
    }
}
```

```java
package gui;
import datamodel.PatternLanguage;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Node;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.scene.control.TextField;
import javafx.stage.Modality;
import javafx.stage.Stage;

import java.util.Optional;


public class MainViewController {


    @FXML private TextField titleInput;


    /**
     * Spawns a modal in order to title the new pattern language
     * @param event The button click
     * @throws Exception if not able to show the dialog
     */
    @FXML
    void handleCreatePLTitle(ActionEvent event) throws Exception {
        Stage dialog = new Stage();
        dialog.initModality(Modality.APPLICATION_MODAL);
        Stage window = (Stage) ((Node)event.getSource()).getScene().getWindow();
        dialog.initOwner(window);

        Parent modal = FXMLLoader.load(getClass().getResource("createPLTitle.fxml"));

        Scene dialogScene = new Scene(modal, 600, 400);
        dialog.setTitle("Rocking Machines - Patterns Editor");
        dialog.setScene(dialogScene);
        dialog.show();
    }

    /**
     * Closes the modal used to title the new pattern language, on pressing the Cancel button
     * @param event the click on the Cancel button
     */
    @FXML
    void handleCancelCreatePL(ActionEvent event) {
        Stage currentStage = (Stage) ((Node)event.getSource()).getScene().getWindow();
        currentStage.close();
    }

    /**
     * Switches to the view Pattern Language scene
     * @param window  the window this function was called from
     * @throws Exception when not able to create and show the new scene
     */
    protected void viewNewPL(Stage window) throws Exception {

        /* Load the new scene into a variable */
        FXMLLoader loader = new FXMLLoader(getClass().getResource("plView.fxml"));

        Parent patternLanguageView = loader.load();
        Scene plView = new Scene(patternLanguageView, 800, 600);

        PLViewController c = loader.getController();
        plView.setUserData(c);
        Main.setPlView(plView);

        c.renderPLView(window);

    }

    /**
     * Handles the Create button click when creating a new pattern language
     * @param event the click on the Create button
     * @throws Exception on failure of viewNewPL, called within
     */
    @FXML
    void handleCreatePL(ActionEvent event) throws Exception {

        /* Get the current window into a variable */
        Stage window = (Stage) ((Node)event.getSource()).getScene().getWindow();

        /* Read the new pattern language title from the text field provided
         * to the user, and create a new PatternLanguage object
         */
        String title = this.titleInput.getText();
        if (title == null || title.equals("") || title.isEmpty()) {
            Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
            alert.setTitle("Confirmation Dialog");
            alert.setHeaderText("You did not enter a name for the new pattern language.");
            alert.setContentText("Are you sure you want to use a default title?");

            Optional<ButtonType> result = alert.showAndWait();

            if (result.isPresent() && result.get() == ButtonType.OK){
                alert.close();
                PatternLanguage newPL = new PatternLanguage(title);
                Main.setPl(newPL);
                this.viewNewPL(window);
            }
            else {
                alert.close();
            }
        }
        else {
            PatternLanguage newPL = new PatternLanguage(title);
            Main.setPl(newPL);
            this.viewNewPL(window);
        }


    }

}


package gui;

import datamodel.PatternComponent;
import javafx.fxml.FXML;

import javafx.event.ActionEvent;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.scene.Node;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.text.Font;
import javafx.stage.Stage;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Optional;

public class PatternViewController {

    /* The container in which we add the Pattern Fields */
    @FXML private Pane pane;

    @FXML private TextField patternTitleInput;

    /* Maps the name of the pattern part to the TextField corresponding to it */
    @FXML private HashMap<String, TextField> names = new HashMap<>();

    /* Maps the name of the pattern part to the TextArea corresponding to its contents */
    @FXML private HashMap<String, TextArea> contents = new HashMap<>();

    /**
     * Handles the click on the Save button, updating the name of the pattern, as well as
     * the name and content of each pattern part corresponding to the pattern currently being edited
     * @param event the click on the save button
     * @throws Exception on failure to find the pattern in the pattern language
     */
    public void handleSavePattern(ActionEvent event) throws Exception {

        String newName = this.patternTitleInput.getText();

        /*
         * Update pattern name according to its corresponding user input field
         * Search for the pattern in the pattern language and tag it's index
         * Then check if the new name already exists in the pattern language
         */
        int index = -1;
        ArrayList<PatternComponent> list = Main.getPl().getComponentsList();
        for (int i=0; i<list.size(); i++) {
            if (list.get(i).getName().equals(Main.getCurrentPattern().getName()))
                index = i;
        }
        if (index == -1)
            throw new Exception("Error: Pattern not found in the pattern language. Please report this incident.");
        PatternComponent pat = list.get(index);

        /* Find out if the new pattern name already exists */
        Boolean flag = true;
        for (PatternComponent i: Main.getPl().getComponentsList()) {

            if (i.getName().equals(newName)) {
                if (i.equals(list.get(index)))
                    continue;
                flag = false;
                break;
            }

        }

        /* Handle the new pattern name */
        if (flag) {
            pat.setName(newName);


            /* Hold each pattern part/section into an ArrayList */
            ArrayList<PatternComponent> partsList = Main.getCurrentPattern().getComponentsList();
            /* Iterate through the parts/sections and update their fields by using the user input */
            for (PatternComponent part : partsList) {
                part.setContents(contents.get(part.getName()).getText());
                part.setName(names.get(part.getName()).getText());
            }

            /* Get the current window into a variable */
            Stage window = Main.getWindow();

            /* Go back to Pattern Language View scene */
            PLViewController c = (PLViewController) Main.getPatternView().getUserData();
            c.renderPLView(window);
        }
        else {
            /* Show error dialog if pattern name already exists */
            Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
            alert.setTitle("Confirmation Dialog");
            alert.setHeaderText("Name not saved: There is already a pattern named \"" + newName + "\"");
            alert.setContentText("Would you like to save the rest of the changes?");

            Optional<ButtonType> result = alert.showAndWait();
            if (result.isPresent() && result.get() == ButtonType.OK) {
                /* Hold each pattern part/section into an ArrayList */
                ArrayList<PatternComponent> partsList = Main.getCurrentPattern().getComponentsList();
                /* Iterate through the parts/sections and update their fields by using the user input */
                for (PatternComponent part : partsList) {
                    part.setContents(contents.get(part.getName()).getText());
                    part.setName(names.get(part.getName()).getText());
                }
            }
            else {
                alert.close();

            }
            PLViewController c = (PLViewController) Main.getPatternView().getUserData();
            c.handleEditPattern(event);
        }

    }

    /**
     * Handles the click on the Cancel button, leaving the Pattern Edit scene
     * @param event the click on the cancel button
     */
    public void handleCancelPattern(ActionEvent event) {
        /* Get the current window into a variable */
        Stage window = Main.getWindow();

        /* Go back to the Pattern Language View scene */
        PLViewController c = (PLViewController) Main.getPatternView().getUserData();
        c.renderPLView(window);
    }

    /**
     * Populates the Pane container with a GridPane containing, in each cell, a TextField and a TextArea,
     *  corresponding to the name and contents of the pattern part/section respectively.
     *
     * The fields are filled with the current names and contents so the user can see and change what
     * they see fit.
     */
    public void populatePatternParts() {
        /* Holds the individual parts/sections of the pattern we're currently editing */
        ArrayList<PatternComponent> partsList = Main.getCurrentPattern().getComponentsList();

        /* Dictate the column number of the GridPane */
        int size = partsList.size();
        int numCols = 2;
        int gpCols;
        if (size/numCols == 0) {
            gpCols = size;
        }
        else {
            gpCols = numCols;
        }

        /* Initialize row and column index to zero */
        int row = 0;
        int col = 0;

        /* Create the GridPanes which will hold the pattern rename and the parts/sections of the pattern */
        GridPane gp = new GridPane();

        /* Clear instance HashMaps, we need to start fresh */
        this.names.clear();
        this.contents.clear();

        /*
         * Pattern name input field
         */
        String patternTitle = Main.getCurrentPattern().getName();

        VBox renameVbox = new VBox();
        Label rename = new Label("Rename Pattern: ");
        rename.setPadding(new Insets(10));
        rename.setFont(Font.font("DejaVu Sans Mono", 18));

        TextField renameInput = new TextField(patternTitle);
        renameInput.setPadding(new Insets(10));
        this.patternTitleInput = renameInput;

        GridPane.setHalignment(rename, HPos.CENTER);
        GridPane.setHalignment(renameInput, HPos.CENTER);

        renameVbox.getChildren().clear();
        renameVbox.getChildren().addAll(rename, renameInput);

        gp.add(renameVbox, 0, 0, 2, 1);

        row++;


        /* Iterate through the list of pattern parts/sections */
        for (PatternComponent part: partsList) {

            if (col >= gpCols) {
                col = 0;
                row++;
            }

            /* Hold the pattern part name for mapping the user input to the respective fields */
            String title = part.getName();

            /* Create the VBox holding the input fields */
            VBox vbox = new VBox();
            TextField name = new TextField(title);
            TextArea contents = new TextArea(part.getContents());
            contents.setFont(Font.font("DejaVu Sans Mono", 12));

            /* Format textArea size so it's nice and big */
            contents.setWrapText(true);
            contents.setMaxSize(300, 100);
            int temp = (contents.getText().length() / contents.getPrefColumnCount()) + 1;
            contents.setPrefRowCount(
                    temp>4 ? temp+1 : 4);

            HBox.setHgrow(name, Priority.ALWAYS);

            /* Map the input fields to the name(title) of the pattern part/section */
            this.names.put(title, name);
            this.contents.put(title, contents);


            vbox.getChildren().clear();
            vbox.getChildren().add(name);
            vbox.getChildren().add(contents);
            vbox.setSpacing(5);

            gp.add(vbox, col, row);
            GridPane.setHalignment(vbox, HPos.CENTER);

            col++;
        }
        gp.setHgap(20);
        gp.setVgap(20);

        gp.setMinSize(Region.USE_COMPUTED_SIZE, Region.USE_COMPUTED_SIZE);
        gp.setMaxSize(Region.USE_COMPUTED_SIZE, Region.USE_COMPUTED_SIZE);
        gp.setPrefSize(Region.USE_COMPUTED_SIZE, Region.USE_COMPUTED_SIZE);

        pane.getChildren().clear(); //remove previous GridPane
        pane.getChildren().add(gp); // add the GridPane


    }





}


package gui;

import datamodel.Pattern;
import datamodel.PatternComponent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Node;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ButtonType;
import javafx.scene.control.Control;
import javafx.scene.layout.*;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.event.ActionEvent;

import java.util.ArrayList;
import java.util.Optional;

import static java.lang.Integer.MAX_VALUE;

public class PLViewController {

    @FXML private Text plTitle;
    @FXML private Pane patternContainer;
    private String selectedPatternId = null;
```

```java
/**
 * Sets the Text field as the title of the Pattern Language
 * @param title the name of the Pattern Language
 */
@FXML
private void setTitle(String title) {
    plTitle.setText(title);
}

/**
 * Switch to template selection scene
 * @param event the button click
 * @throws Exception on failure to load the fxml file
 */
@FXML
void handleAddPattern(ActionEvent event) throws Exception {
    /* Get the current window into a variable */
    Stage window = Main.getWindow();

    FXMLLoader loader = new FXMLLoader(getClass().getResource("templateView.fxml"));
    Parent templateViewRoot = loader.load();
    TemplateViewController c = loader.getController();
    c.populateTemplates();

    Scene templateView = new Scene(templateViewRoot, 800, 600);

    window.close();
    window.setTitle("Rocking Machines - Patterns Editor");
    window.setScene(templateView);
    window.show();
}

/**
 * Returns to the starting scene, so we can change the pattern language
 * @param event the button click
 */
@FXML
public void handleChangePL(ActionEvent event) {

    /* Get the current window into a variable */
    Stage window = Main.getWindow();

    window.close();

    window.setScene(Main.getStart());
    window.show();
}


/** Populates the Pane container with a GridPane holding a button for each pattern in the pattern language */
private void populatePatterns() {

    /* ArrayList holding the patterns in the pattern language */
    ArrayList<PatternComponent> patternsList = Main.getPl().getComponentsList();

    /* Dictate the number of columns there should be in the GridPane */
    int size = patternsList.size();
    int numCols = 3;
    int gpCols;
    if (size/numCols == 0) {
        gpCols = size;
    }
    else {
        gpCols = numCols;
    }

    /* Initialize row and column index to zero */
    int row = 0;
    int col = 0;
    /* Create the GridPane which will hold the Buttons (patterns) */
    GridPane gp = new GridPane();


    /* Iterate through the list of patterns in the pattern language */
    for (PatternComponent pattern: patternsList) {

        if (col >= gpCols) {
            col = 0;
            row++;
        }

        /* Pattern name is used as the Button name & id */
        String name = pattern.getName();
        Button btn = new Button(name);                      // Create the Button
        btn.setId(name);                                    // Set button id to its title
        btn.setOnAction((e) -> this.handlePickPattern(e));  // Set button handler to handlePickTemplate

        /* Set the buttons to be the same size */
        btn.setMaxWidth(MAX_VALUE);
        btn.setPrefSize(Region.USE_COMPUTED_SIZE, Region.USE_COMPUTED_SIZE);
        HBox.setHgrow(btn, Priority.ALWAYS);
        btn.setAlignment(Pos.CENTER);
        btn.setPadding(new Insets(10));

        gp.add(btn, col, row);
        GridPane.setHalignment(btn, HPos.CENTER);
        col++;
    }
    gp.setVgap(20);
    gp.setHgap(20);

    patternContainer.getChildren().clear(); //remove previous GridPane
    patternContainer.getChildren().add(gp); // add the GridPane
}

/**
 * Handles the click on a button representing a pattern, setting an instance variable
 * @param event the button click
 */
private void handlePickPattern(ActionEvent event) {
    Control src = (Control)event.getSource();
    this.selectedPatternId = src.getId();
}

/**
 * Handles the click on the Delete button, tries to delete the pattern from the pattern language
 * Shows an error if no pattern is picked
 * Shows a warning before deleting
 * @param event the button click
 */
public void handleDeletePattern(ActionEvent event) {
    /* Show error dialog if no pattern is picked for removal */
    if (this.selectedPatternId == null || this.selectedPatternId.isEmpty()) {

        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Error Dialog");
        alert.setHeaderText("You did not select a pattern.");
        alert.setContentText("Please select a pattern to remove.");
        alert.showAndWait();
    }
    /* Show a warning dialog before deleting the pattern */
    else {
        Alert alert = new Alert(Alert.AlertType.WARNING);
        alert.setTitle("Warning Dialog");
        alert.setHeaderText("Are you sure you want to delete " + this.selectedPatternId + "?");
        alert.setContentText("All patterns named " + this.selectedPatternId + " will be deleted!");

        Optional<ButtonType> result = alert.showAndWait();
        if (result.isPresent() && result.get() == ButtonType.OK) {
            Main.getPl().remove(this.selectedPatternId);
            this.selectedPatternId = null;
            this.renderPLView((Stage) ((Node)event.getSource()).getScene().getWindow());
        }
        else {
            alert.close();

        }

    }
}

/**
 * Handles the click on the Edit button, tries to enter the PatternView scene
 * @param event the button click
 * @throws Exception on failure to load the FXML file or failure to find the pattern in the Pattern Language
 */
public void handleEditPattern(ActionEvent event) throws Exception {

    /* Show an error dialog if no pattern was chosen for editing */
    if (this.selectedPatternId == null || this.selectedPatternId.isEmpty()) {

        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Information Dialog");
        alert.setHeaderText("You did not select a pattern.");
        alert.setContentText("Please select a pattern to edit.");
        alert.showAndWait();
    }
    else {
        /* hold the pattern the user wants to edit in local variable and set it in the static variable
         * if the pattern is not found (unexpected behaviour), throw an exception
         */
        Boolean flag = true;
        for (PatternComponent i: Main.getPl().getComponentsList()) {
            if (i.getName().equals(this.selectedPatternId)) {
                Main.setCurrentPattern((Pattern) i);
                flag = false;
            }
        }
        if (flag) {
            throw new Exception("Could not find pattern. Unexpected behaviour. Please report this issue.");
        }

        /* Get the current window into a variable */
        Stage window = Main.getWindow();

        FXMLLoader loader = new FXMLLoader(getClass().getResource("patternView.fxml"));
        Parent patternViewRoot = loader.load();
        PatternViewController c = loader.getController();
        c.populatePatternParts();


        Scene patternView = new Scene(patternViewRoot, 800, 600);
        patternView.setUserData(this);
        Main.setPatternView(patternView);


        window.close();
        window.setTitle("Rocking Machines - Patterns Editor");
        window.setScene(patternView);
        window.show();
    }
}

/**
 * Renders the Pattern Language scene
 * @param window the window we want to change into the PLView scene
 */
public void renderPLView(Stage window) {

    this.setTitle(Main.getPl().getName());
    this.populatePatterns();

    this.selectedPatternId = null;
```

```java
        /* Close pop-up window and change the window variable to the primaryStage */
        window.close();
        window = Main.getWindow();

        /* Render the new scene into primaryStage */
        window.setScene(Main.getPlView());
        window.show();
    }

}


package gui;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.geometry.Insets;
import javafx.scene.Node;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ButtonType;
import javafx.scene.control.Control;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.scene.layout.Region;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.util.*;

import static java.lang.Integer.MAX_VALUE;

public class TemplateViewController {

    @FXML private VBox templateContainer;
    private String templateId = null;

    /**
     * Populate the template selection scene with buttons corresponding to pattern templates
     */
    @FXML
    public void populateTemplates() {

        List<Button> buttonList = new ArrayList<>();              // Collection to hold created Button objects

        Set templatesSet = Main.getTemplateFactory().getTemplatesList().keySet(); // Get all names so we can put them on
buttons
        for (Object name: templatesSet) {

            String title = name.toString();
            Button btn = new Button(title);                     // Create the Button
            btn.setId(title);                                   // Set button id to its title
            btn.setOnAction((e) -> this.handlePickTemplate(e)); // Set button handler to handlePickTemplate

            /* Make buttons the same size */
            btn.setMaxWidth(MAX_VALUE);
            HBox.setHgrow(btn, Priority.ALWAYS);
            btn.setPadding(new Insets(10));

            /* Finally, add the button to the list */
            buttonList.add(btn);
        }


        /* Add buttons to gui */
        templateContainer.setSpacing(20);
        templateContainer.getChildren().clear(); //remove all Buttons that are currently in the container
        templateContainer.getChildren().addAll(buttonList); // add new Buttons from the list

    }

    /**
```

```
     * Returns to the new Pattern Language scene view on clicking Cancel
     * @param event the button click
     * @throws Exception on failure to load fxml file
     */
    @FXML
    public void handleCancel(ActionEvent event) throws Exception {
        Stage currentStage = (Stage) ((Node)event.getSource()).getScene().getWindow();
        PLViewController c = new PLViewController();
        c.renderPLView(currentStage);
    }

    /**
     * Handles the click on a Button representing a pattern, and sets an instance variable to that pattern
     * @param event the button click
     */
    @FXML
    private void handlePickTemplate(ActionEvent event) {
        Control src = (Control)event.getSource();
        this.templateId = src.getId();

    }

    /**
     * Handles the click on the create Button, calling notifyDefault() when no template was picked,
     *  and switchToPatternView() otherwise
     * @param event the click on the Create button
     */
    @FXML
    public void handleCreate(ActionEvent event) {
        /* Get the current window into a variable */
        Stage window = (Stage) ((Node)event.getSource()).getScene().getWindow();

        if (this.templateId == null || this.templateId.isEmpty() || this.templateId == "null") {
            this.notifyDefault(window);
        }
        else {
            this.switchToPatternView(window);
        }
    }

    /**
     * Changes to the Pattern Language view
     * @param window the current window
     */
    private void switchToPatternView(Stage window) {

        Main.getPl().add(Main.getTemplateFactory().createTemplate(templateId));
        PLViewController c = (PLViewController) Main.getPlView().getUserData();
        c.renderPLView(window);
    }

    /**
     * Spawns a confirmation dialog, asking the user if they want to continue using the default template
     * @param window the current window
     */
    private void notifyDefault(Stage window) {
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setTitle("Confirmation Dialog");
        alert.setHeaderText("You did not select a template for the new pattern.");
        alert.setContentText("Are you sure you want to use the default template \"Micro-Pattern\"?");

        Optional<ButtonType> result = alert.showAndWait();

        /* Set the template to the default one and call switchToPatternView() if user clicks on OK */
        if (result.isPresent() && result.get() == ButtonType.OK){
            alert.close();
            this.templateId = "Micro-Pattern";
            switchToPatternView(window);
        }
        else {
            alert.close();
        }
    }
}
```