

```

1: package fernuni.propra.main;
2:
3:
4: public class ParameterSet {
5:     private String runParameter;
6:     private String inputFile;
7:     private Integer timeLimit;
8:     private static String[] validRunParameters = {"s","sd","v","vd", "d"};
9:     private static final String formInputFileParameter = " if=\"pathToFile
\n";
10:     private static final String formTimeLimitParameter = "l=timeLimit , wh
ere timeLimit is a positive Integer number";
11:     private static final String formRunParameter = "r=parameter";
12:
13:
14:     public ParameterSet() {
15:
16:     }
17:
18:     void setRunParameter(String runParameter) throws ParameterSetException
{
19:         if (this.runParameter != null) {
20:             throw new ParameterSetException("Run parameter is alre
ady set. Please provide only one run parameter specification.");
21:         } else if (!isValidRunParameter(runParameter)) {
22:             String message = "Run parameter is not valid. Please p
rovide a valid run parameter in the form " + formRunParameter+ ", where parameter is o
ne of: ";
23:             for (String validParameter : validRunParameters) {
24:                 message = message + validParameter;
25:                 message = message + " ";
26:             }
27:             throw new ParameterSetException(message);
28:         } else {
29:             this.runParameter = runParameter;
30:         }
31:     }
32:
33:     void setInputFile(String inputFile) throws ParameterSetException {
34:         if (this.inputFile != null) {
35:             throw new ParameterSetException("Path to input file is
already set. Please provide only one input file specification.");
36:         } /*else if (!inputFile.startsWith("\n") || !inputFile.endsWit
h("\n")) {
37:             throw new ParameterSetException("The path to the input
file you entered is: " + inputFile +System.getProperty("line.separator") + " The inpu
t file parameter needs to start and end with \" . \"
38:                 + "Please supply the path to the input
file in the form: if=\"pathToFile\n\"");
39:         } */
40:         else {
41:             this.inputFile = inputFile.replace("\n", "");
42:
43:         }
44:     }
45:
46:     void setTimeLimit(int timeLimit) throws ParameterSetException {
47:         if (this.timeLimit != null) {
48:             throw new ParameterSetException("Time limit is already
set. Please provide only one time limit specification.");
49:         } else {
50:             this.timeLimit = timeLimit;
51:         }
52:     }
53:
54:     boolean isValidParameterSet() throws ParameterSetException {
55:         if(runParameter == null) {
56:             String message = "No run parameter provided. Please pr
ovide a valid run parameter in the form " + formRunParameter+ ", where parameter is on
e of: ";
57:             for (String validParameter : validRunParameters) {
58:                 message = message + validParameter;
59:                 message = message + " ";
60:             }
61:             throw new ParameterSetException(message);
62:         }
63:         if (runParameter.equals("v") || runParameter.equals("vd") || r
unParameter.equals("d")) {
64:             if (inputFile != null) {
65:                 return true;
66:             } else {
67:                 throw new ParameterSetException("No path to th
e input file is specified. Please provide the path to the input file in the form:" + f
ormInputFileParameter);
68:             }
69:         } else if (runParameter.equals("s") || runParameter.equals("sd
")) {
70:             if (inputFile == null) {
71:                 throw new ParameterSetException("No path to th
e input file is specified. Please provide the path to the input file in the form:" + f
ormInputFileParameter);
72:             } else if (timeLimit == null) {
73:                 throw new ParameterSetException("No time limit
is specified. Please provide a time limit in the form:" + formTimeLimitParameter);
74:             } else {
75:                 return true;
76:             }
77:         } else {
78:             String message = "Run parameter is not valid. Please p
rovide a valid run parameter in the form " + formRunParameter+ ", where parameter is o
ne of: ";
79:             for (String validParameter : validRunParameters) {
80:                 message = message + validParameter;
81:                 message = message + " ";
82:             }
83:             throw new ParameterSetException(message);
84:         }
85:
86:     }
87:
88:     private boolean isValidRunParameter(String runParameter) {
89:         boolean isValid = false;
90:         for (String validParameter : validRunParameters) {
91:             if (validParameter.equals(runParameter)) {
92:                 isValid = true;
93:                 break;
94:             }
95:         }
96:         return isValid;
97:     }
98:
99:     String getRunParameter() {
100:         return this.runParameter;
101:     }
102:
103:     String getInputFile() {
104:         return this.inputFile;
105:     }

```

```
106:
107:     int getTimeLimit() {
108:         return this.timeLimit;
109:     }
110:
111:     /*String[] getValidRunParameters() { // TODO
112:         String[] outStrings = new String[validRunParameters.length];
113:         for (int i = 0; i < validRunParameters.length; i++) {
114:             outStrings[i] = validRunParameters[i];
115:         }
116:         return outStrings;
117:     } */
118:
119: }
```

```
1: package fernuni.propra.main;
2:
3: public class ParameterSetException extends Exception {
4:
5:     public ParameterSetException() {
6:         // TODO Auto-generated constructor stub
7:     }
8:
9:     public ParameterSetException(String message) {
10:        super(message);
11:        // TODO Auto-generated constructor stub
12:    }
13:
14:    public ParameterSetException(Throwable cause) {
15:        super(cause);
16:        // TODO Auto-generated constructor stub
17:    }
18:
19: }
```



```

1: package fernuni.propra.main;
2:
3: import java.util.Iterator;
4: import java.util.StringTokenizer;
5:
6: import fernuni.propra.algorithm.UserSolveAAS;
7: import fernuni.propra.algorithm.UserSolveAASException;
8: import fernuni.propra.algorithm.UserValidateAAS;
9: import fernuni.propra.algorithm.UserValidateAASException;
10: import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
11: import fernuni.propra.file_processing.UserReadInputWriteOutputException;
12: import fernuni.propra.internal_data_model.IRoom;
13: import fernuni.propra.internal_data_model.Lamp;
14: import fernuni.propra.internal_data_model.Point;
15: import fernuni.propra.user_interface.UserDisplayAAS;
16:
17: /**
18:  * Haupteinstiegspunkt der Anwendung.
19:  *
20:  * In der Main-Komponente müssen unter anderem die Eingabeparameter verarbeitet werden.</br>
21:  *
22:  * Für den Ablaufparameter "r" wird folgende Festlegung getroffen:
23:  * <ul>
24:  * <li>"s" (solve): durch die XML-Datei beschriebene Probleminstanz wird eine Lösung ermittelt. Die Positionen der Lampen werden in der angegebenen XML-Datei gespeichert. Wenn in der XML-Datei bereits eine Lösung enthalten ist, so ist diese zu überschreiben.</li>
25:  * <li>"sd" (solve & display): wie "s", nur dass der Raum sowie die ermittelte n Positionen der Lampen zusätzlich in der grafischen Oberfläche angezeigt werden.</li>
26:  * <li>"v" (validate): durch diese Option wird geprüft, ob der in der angegebenen XML-Datei enthaltene Raum durch die ebenso dort angegebenen Lampen vollständig ausgeleuchtet ist. Das Ergebnis der Prüfung sowie die Anzahl und Positionen der Lampen werden ausgegeben. Falls die angegebene XML-Datei keinen zulässigen Raum enthält, wird eine Fehlermeldung ausgegeben. Die Ausgabe erfolgt in der Kommandozeile.</li>
27:  * <li>"vd" (validate & display): wie "v", nur dass der Raum und die Lampen nach der Validierung zusätzlich in der grafischen Oberfläche angezeigt werden.</li>
28:  * <li>"d" (display): der in der XML-Datei enthaltene Raum und die Lampen werden in der grafischen Oberfläche angezeigt. Falls die angegebene XML-Datei keinen zulässigen Raum enthält, wird eine Fehlermeldung auf der Kommandozeile ausgegeben.</li>
29:  * </ul>
30:  * Der Eingabeparameter "if" ist ein String, der den Pfad der Eingabedatei beinhaltet.</br>
31:  *
32:  * Der Parameter für ein Zeitlimit "l" ist eine positive natürliche Zahl, welche die maximale Rechenzeit in Sekunden angibt.
33:  */
34: public class Main {
35:
36:     /**
37:      * Haupteinstiegsfunktion
38:      */
39:     private static final String generalHelpMessage = "Java -jar ProPra.jar r=runParameter if=\"pathToFile\" l=timeLimit \n \n "
40:         + "The runParameter specified by r =runParameter is mandatory and must be one of s, sd, v, vd or d .\n "
41:         + "The input file parameter is also mandatory. pathToFile specifies the full path to a valid input file. The \" before and after pathToFile are mandatory\n"
42:         + "The time limit parameter is optional. For runParameter = s or runParameter = sd. This parameter specifies how long the solution algorithm searches for an optimal lamp layout. Time limit must be a positive integer number.";

```

```

43:
44:
45:
46:     public static void main(String[] args) {
47:
48:         for (String arg : args) { // Debug
49:             System.out.println(arg);
50:         }
51:
52:         ParameterSet parameterSet = new ParameterSet();
53:         try {
54:             for (String paramString : args) {
55:                 StringTokenizer st = new StringTokenizer(paramString, "=");
56:                 if(st.countTokens()==2) {
57:                     String parameterKey = st.nextToken().trim();
58:                     String parameterValue = st.nextToken().trim();
59:                     switch (parameterKey) {
60:                         case "r":
61:                             parameterSet.setRunParameter(parameterValue);
62:                             break;
63:                         case "if":
64:                             parameterSet.setInputFile(parameterValue);
65:                             break;
66:                         case "l":
67:                             parameterSet.setTimeLimit(Integer.parseInt(parameterValue));
68:                             break;
69:                         default:
70:                             throw new GeneralException();
71:                     }
72:                 } else {
73:                     throw new GeneralException();
74:                 }
75:             }
76:         }
77:
78:         if (parameterSet.isValidParameterSet()) {
79:             IRoom room;
80:             UserReadInputWriteOutputAAS userReadWriteAAS;
81:             UserDisplayAAS userDisplayAAS;
82:             UserValidateAAS userValidateAAS;
83:             UserSolveAAS userSolveAAS;
84:             int numberOfLampsInSolution;
85:
86:             printMessageToConsole("Starting computation ..");
87:
88:             switch (parameterSet.getRunParameter()) {
89:                 case "s":
90:                     userReadWriteAAS = new UserReadInputWriteOutputAAS(parameterSet.getInputFile());
91:                     room = userReadWriteAAS.readInput();
92:                     userSolveAAS = new UserSolveAAS();
93:                     numberOfLampsInSolution = userSolveAAS.solve(room, parameterSet.getTimeLimit());
94:                     userReadWriteAAS.writeOutput(room);
95:

```

```

96:                printMessageToConsole("Computation fin
ished ...");
97:                break;
98:                case "sd":
99:                    userReadWriteAAS = new UserReadInputWr
iteOutputAAS(parameterSet.getInputFile());
100:                    room = userReadWriteAAS.readInput();
101:                    userSolveAAS = new UserSolveAAS();
102:                    numberOfLampsInSolution = userSolveAAS
.solve(room, parameterSet.getTimeLimit());
103:                    userReadWriteAAS.writeOutput(room);
104:                    userDisplayAAS = new UserDisplayAAS();
105:                    userDisplayAAS.display(room);
106:                    printMessageToConsole("Computation fin
ished ...");
107:                    printMessageToConsole("Number of lamps
necessary:" + numberOfLampsInSolution); // TODO
108:                    printMessageToConsole(userSolveAAS.get
RuntimeInformation().toString());
109:                    break;
110:                    case "v":
111:                        userReadWriteAAS = new UserReadInputWr
iteOutputAAS(parameterSet.getInputFile());
112:                        room = userReadWriteAAS.readInput();
113:                        userValidateAAS = new UserValidateAAS(
);
114:                        userValidateAAS.validate(room);
115:                        printMessageToConsole(userValidateAAS.
getResultString());
116:                        break;
117:                        case "vd":
118:                            userReadWriteAAS = new UserReadInputWr
iteOutputAAS(parameterSet.getInputFile());
119:                            room = userReadWriteAAS.readInput();
120:                            userValidateAAS = new UserValidateAAS(
);
121:                            userValidateAAS.validate(room);
122:                            //System.out.println(userValidateAAS.g
etResultString());
123:                            printMessageToConsole(userValidateAAS.
getResultString());
124:                            userDisplayAAS = new UserDisplayAAS();
125:                            userDisplayAAS.display(room);
126:                            break;
127:                            case "d":
128:                                userReadWriteAAS = new UserReadInputWr
iteOutputAAS(parameterSet.getInputFile());
129:                                room = userReadWriteAAS.readInput();
130:                                userDisplayAAS = new UserDisplayAAS();
131:                                userDisplayAAS.display(room);
132:                                break;
133:                                default:
134:                                    }
135:                                }
136:                            } else {
137:                                throw new GeneralException();
138:                            }
139:                        }
140:                    }
141:                }
142:            } catch (ParameterSetException e) {
143:                printMessageToConsole(e.getMessage());
144:                System.exit(0);
145:            }
146:        } catch (NumberFormatException nfe) {
147:            printMessageToConsole("The timeLimit parameter specifi
ed by l=timeLimit is not an integer number");
148:        } catch (GeneralException ge) {
149:            printMessageToConsole(generalHelpMessage);
150:            System.exit(0);
151:        } catch (UserReadInputWriteOutputException e) {
152:            printMessageToConsole(e.getMessage());
153:            System.exit(0);
154:        } catch (UserValidateAASException e) {
155:            printMessageToConsole(e.getMessage());
156:            System.exit(0);
157:        } catch (UserSolveAASException e) {
158:            printMessageToConsole(e.getMessage());
159:            System.exit(0);
160:        }
161:    }
162:
163:
164:    private static void printMessageToConsole(String message) {
165:        System.out.println(message);
166:    }
167: }

```

```
1: package fernuni.propra.main;
2:
3: public class GeneralException extends Exception {
4:
5:     public GeneralException() {
6:         // TODO Auto-generated constructor stub
7:     }
8:
9:     public GeneralException(String message) {
10:         super(message);
11:         // TODO Auto-generated constructor stub
12:     }
13:
14:     public GeneralException(Throwable cause) {
15:         super(cause);
16:         // TODO Auto-generated constructor stub
17:     }
18:
19:
20:
21: }
```



```
1: package fernuni.propra.file_processing;
2:
3: import fernuni.propra.internal_data_model.IRoom;
4:
5: /**
6:  * A provider of an algorithm that provides persistence to an {@link IRoom} in
stance.
7:  * @author alex
8:  *
9:  */
10: public interface IPersistence {
11:     /**
12:      * Used to read an {@link IRoom} that is present a certain location.
13:      * @param location : The location at which the {@link IRoom} is stored
.
14:      * @return : The {@link IRoom}
15:      * @throws PersistenceException : thrown if an unexpected error occur
ed during the reading process.
16:      */
17:     IRoom readInput(String location) throws PersistenceException;
18:
19:     /**
20:      * Used to persistently store the {@link IRoom} at a certain location.
21:      * @param room : The {@link IRoom}
22:      * @param location : The location at which the {@link IRoom} is stored
.
23:      * @throws PersistenceException : thrown if an unexpected error occur
ed during the writing process.
24:      */
25:     void writeOutput(IRoom room, String location) throws PersistenceExcept
ion;
26:
27: }
```



```

1: package fernuni.propra.file_processing;
2:
3: import fernuni.propra.internal_data_model.IRoom;
4: import fernuni.propra.internal_data_model.Lamp;
5: import fernuni.propra.internal_data_model.LineSegment;
6: import fernuni.propra.internal_data_model.Point;
7: import fernuni.propra.internal_data_model.Room;
8:
9: import java.io.File;
10:
11: import java.io.FileNotFoundException;
12: import java.io.FileOutputStream;
13: import java.io.FileReader;
14: import java.io.IOException;
15: import java.io.InputStream;
16: import java.io.InputStreamReader;
17: import java.io.StringReader;
18: import java.util.ArrayList;
19: import java.util.Iterator;
20: import java.util.LinkedList;
21: import java.util.List;
22:
23:
24: import org.jdom2.Document;
25: import org.jdom2.Element;
26: import org.jdom2.JDOMException;
27: import org.jdom2.input.SAXBuilder;
28: import org.jdom2.input.sax.XMLReaders;
29: import org.jdom2.output.Format;
30: import org.jdom2.output.XMLOutputter;
31:
32: /**
33:  * A specific provider of persistence for an {@link IRoom} that stores/and reads the {@link IRoom}
34:  * from an xml-file that adheres to the Document Type Definition (DTD) specified in Listing 1 of [1], i.e.
35:  *
36:  * 01 <?xml version="1.0" encoding="UTF-8"?>
37:  * 02 <!ELEMENT Raum (ID, ecken, lampen?)>
38:  * 03 <!ELEMENT ID (#PCDATA)>
39:  * 04 <!ELEMENT ecken (Ecke*)>
40:  * 05 <!ELEMENT lampen (Lampe*)>
41:  * 06 <!ELEMENT Ecke (x, y)>
42:  * 07 <!ELEMENT Lampe (x, y)>
43:  * 08 <!ELEMENT x (#PCDATA)>
44:  * 09 <!ELEMENT y (#PCDATA)>
45:  * <p>
46:  * {@link FilePersistence} makes use of
47:  * the JDOM2-library (see http://www.jdom.org/)
48:  *
49:  * @author alex
50:  *
51:  * [1] Aufgabenstellung Programmierpraktikum SS 2020
52:  */
53: class FilePersistence implements IPersistence {
54:     private static final String DTDFileName = "DataModel.dtd";
55:
56:     //all lamps are turned on initially
57:     @Override
58:     public IRoom readInput(String xmlFilePath) throws PersistenceException
59: {
60:
61:         Document document = null;
62:
63:         InputStreamReader isr = null;
64:         Room outRoom = null;
65:         try {
66:             File xmlFile = new File(xmlFilePath);
67:             checkFileAvailability(xmlFile);
68:
69:             isr = new FileReader(xmlFile);
70:             StringBuilder sb = insertDTDForValidation(isr);
71:
72:             //parse xml
73:             SAXBuilder builder = new SAXBuilder(XMLReaders.DTDVALIDATING);
74:             document = builder.build(new StringReader(sb.toString()));
75:
76:             Element roomNode = document.getRootElement();
77:
78:             String ID = roomNode.getChildText("ID");
79:
80:             Element cornersNode = roomNode.getChild("ecken");
81:             if (cornersNode == null) { // if no corners are provided (which is valid according to DTD) an exception is thrown since no computation can be done
82:                 throw new PersistenceException("No corners provided. Cannot compute anything. Please provide an input file with a valid number of Ecken.");
83:             }
84:             List<Element> cornerNodes = cornersNode.getChildren("Ecke");
85:
86:             LinkedList<Point> corners = new LinkedList<Point>();
87:             List<LineSegment> walls = new ArrayList<LineSegment>();
88:
89:             //loop over all corners
90:             for (Element cornerNode : cornerNodes) {
91:                 Point tmpPoint = new Point(Double.parseDouble(cornerNode.getChildText("x")), Double.parseDouble(cornerNode.getChildText("y")));
92:                 // add wall
93:                 if (!corners.isEmpty()) {
94:                     LineSegment newWall = new LineSegment(corners.getLast(), tmpPoint);
95:                     testAndAddWallToWalls(newWall, walls);
96:                 }
97:                 // add corner
98:                 corners.add(tmpPoint);
99:             }
100:             // add last wall
101:             LineSegment newWall = new LineSegment(corners.getLast(), corners.getFirst());
102:             testAndAddWallToWalls(newWall, walls);
103:
104:             //add lamps
105:             List<Lamp> lamps = getLamps(roomNode, walls);
106:
107:             outRoom = new Room(ID, lamps, corners);
108:
109:         } catch (JDOMException e) {
110:             throw new PersistenceException(e);
111:         } catch (NumberFormatException e) {
112:             throw new PersistenceException(e);
113:         } catch (IOException e) {
114:             throw new PersistenceException(e);
115:         } finally {
116:             if (isr != null) {
117:                 try {

```

```

115:         isr.close();
116:     } catch (IOException e) {
117:         throw new PersistenceException(e);
118:     }
119: }
120: }
121: return outRoom;
122: }
123:
124:
125: @Override
126: public void writeOutput(IRoom room, String xmlFile) throws Persistence
Exception {
127:     FileOutputStream fos = null;
128:     try {
129:         fos = new FileOutputStream(xmlFile);
130:
131:         //build xml structure conforming with DTD definition
132:         Document outDocument = new Document();
133:
134:         //root node
135:         Element roomNode = new Element("Raum");
136:         outDocument.addContent(roomNode);
137:
138:         // write ID
139:         Element ID = new Element("ID");
140:         ID.addContent(room.getID());
141:         roomNode.addContent(ID);
142:
143:         // write corners
144:         Element cornersNode = new Element("ecken");
145:         Iterator<Point> cornersOfRoomIterator = room.getCorner
s();
146:         while (cornersOfRoomIterator.hasNext()) {
147:             Point corner = cornersOfRoomIterator.next();
148:             Element cornerNode = new Element("Ecke");
149:             // write x,y
150:             Element xNode = new Element("x");
151:             Element yNode = new Element("y");
152:             xNode.addContent(String.valueOf(corner.getX()));
153:             yNode.addContent(String.valueOf(corner.getY()));
154:             cornerNode.addContent(xNode);
155:             cornerNode.addContent(yNode);
156:             cornersNode.addContent(cornerNode);
157:         }
158:         roomNode.addContent(cornersNode);
159:
160:         Iterator<Lamp> lampIterator = room.getLamps();
161:         if (lampIterator.hasNext()) {
162:             Element lampsNode = new Element("lamphen");
163:             while (lampIterator.hasNext()) {
164:                 Lamp lamp = lampIterator.next();
165:                 if (lamp.getOn()) { // lamps are only
166:                     // are not only candidates but part of the best solution.
167:                     Element lampNode = new Element
("Lampe");
168:                     Element xNode = new Element("x");
169:                     Element yNode = new Element("y");
170:                     f(lamp.getX());
171:                     f(lamp.getY());
172:                     lampNode.addContent(xNode);
173:                     lampNode.addContent(yNode);
174:                     lampsNode.addContent(lampNode);
175:                 }
176:             }
177:             roomNode.addContent(lampsNode);
178:         }
179:
180:         XMLOutputter xmlOutputter = new XMLOutputter(Format.ge
tPrettyFormat());
181:         xmlOutputter.getFormat().setEncoding("UTF-8");
182:         try {
183:             // actually write output
184:             xmlOutputter.output(outDocument, fos);
185:         } catch (IOException e) {
186:             throw new PersistenceException(e);
187:         }
188:     } catch (IOException ioe) {
189:         throw new PersistenceException(ioe);
190:     } finally { // clean up
191:         if (fos != null) {
192:             try {
193:                 fos.close();
194:             } catch (IOException e) {
195:                 throw new PersistenceException(e);
196:             }
197:         }
198:     }
199: }
200:
201: /**
202:  * Returns a {@link List} of {@link Lamp}s that have been specified in
203:  * the xml file. Uses JDOM-2.
204:  * @param roomNode : The "Raum" xml-Node of this valid inputfile
205:  * @param walls
206:  * @return
207:  * @throws IOException
208:  */
209: private List<Lamp> getLamps(Element roomNode, List<LineSegment> walls)
throws IOException {
210:     Element lampsNode = roomNode.getChild("lamphen");
211:     List<Lamp> lamps = new LinkedList<Lamp>();
212:     if (lampsNode != null) { // contains lamps
213:         List<Element> lampNodes = lampsNode.getChildren("Lampe");
214:         for (Element lampNode: lampNodes) {
215:             Lamp tmpLamp = new Lamp(Double.parseDouble(lam
pNode.getChildText("x")), Double.parseDouble(lampNode.getChildText("y")));
216:             if (tmpLamp.isInsidePolygon(walls)) { // the l
amp is actually positioned inside the room
217:                 lamps.add(tmpLamp);
218:             } else {
219:                 throw new IOException("Not all lamps a
re actually inside the room. Please provide a valid room layout");
220:             }
221:         }
222:     }

```

```

223:         }
224:         return lamps;
225:     }
226:
227:     /**
228:      * Allows for format checking of the file content that is provided by
229:      * an {@link InputStreamReader}. Inserts the DTD specification
230:      * after the first ">" (ASCII dez = 62) has been read -> works for xml
231:      * files. Otherwise the specification is appended to the end of the
232:      * file, which will produce nothing meaning full.
233:      * @param isr : The {@link InputStreamReader} obtained from an xml fil
234:
235:      * @return A {@link StringBuilder} that has the DTD-specification adde
236:      * d to its xml-header.
237:      * @throws IOException : If read from the supplied {@link InputStreaM
238:      * R
239:      * eader} fails.
240:      */
241:     private StringBuilder insertDTDForValidation(InputStreamReader isr) th
242:     rows IOException {
243:         StringBuilder sb = new StringBuilder();
244:
245:         int c = -1;
246:         while((c = isr.read()) != -1) {
247:             sb.append((char) c);
248:             if (c == 62) {
249:                 break;
250:             }
251:         }
252:
253:         sb.append(System.getProperty("line.separator"));
254:         sb.append(readDTDFile());
255:         sb.append(System.getProperty("line.separator"));
256:         while((c = isr.read()) != -1) {
257:             sb.append((char) c);
258:         }
259:         return sb;
260:     }
261:
262:     private static void checkFileAvailability(File xmlFile) throws FileNot
263:     FoundException {
264:         if (!xmlFile.exists()) {
265:             throw new FileNotFoundException("File not found at \""
266: + xmlFile + "\". Enter a valid file path.");
267:         }
268:         if(!xmlFile.isFile()) {
269:             throw new FileNotFoundException("Path does not point t
270: o a file. Enter a valid file path.");
271:         }
272:     }
273:
274:     static void testAndAddWallToWalls(LineSegment newWall, List<LineSegmen
275: t> walls) throws PersistenceException {
276:         //checks intersections and perpendicularity
277:         if (walls.isEmpty()) {
278:             walls.add(newWall);
279:             return;
280:         }
281:         if (!newWall.penetratesLineSegments(walls)) {
282:             if (newWall.perpendicular(walls.get(walls.size()-1)))
283:                 walls.add(newWall);
284:             } else {
285:                 throw new PersistenceException("Sucessive wall
286: s are not perpendicular. Please provide a valid room layout!");
287:             }
288:         } else {
289:             throw new PersistenceException("Walls intersect. Pleas
290: e provide a valid room layout!");
291:         }
292:     }
293:
294:     private String readDTDFile() throws IOException{
295:         InputStreamReader isr = null;
296:
297:         StringBuilder sb = new StringBuilder();
298:         sb.append("<!DOCTYPE Raum [");
299:
300:         try {
301:             InputStream inputStream = getClass().getResourceAsStre
302: am(System.getProperty("file.separator") + DTDFilename);
303:             isr = new InputStreamReader(inputStream);
304:             boolean firstTagRead = false;
305:             int c = -1;
306:             while((c=isr.read())!=-1) {
307:                 if(firstTagRead) {
308:                     sb.append((char) c);
309:                 }
310:                 else {
311:                     if(c==62) {
312:                         firstTagRead = true;
313:                     }
314:                 }
315:             }
316:         } catch (IOException e) {
317:             throw new IOException(e);
318:         } finally {
319:             if (isr != null) {
320:                 try {
321:                     isr.close();
322:                 } catch (IOException e) {
323:                     throw new IOException(e);
324:                 }
325:             }
326:         }
327:         sb.append(System.getProperty("line.separator"));
328:         sb.append("]>");
329:         return sb.toString();
330:     }

```


./ProPra2020_workspace/File_Processing_Component/src/fernuni/propra/file_processing/UserReadInputWriteOutputException.java

```
1: package fernuni.propra.file_processing;
2:
3: public class UserReadInputWriteOutputException extends Exception {
4:     public UserReadInputWriteOutputException() {
5:         super();
6:         // TODO Auto-generated constructor stub
7:     }
8:
9:     public UserReadInputWriteOutputException(String message, Throwable cau
se) {
10:         super(message, cause);
11:         // TODO Auto-generated constructor stub
12:     }
13:
14:     public UserReadInputWriteOutputException(String message) {
15:         super(message);
16:         // TODO Auto-generated constructor stub
17:     }
18:
19:     public UserReadInputWriteOutputException(Throwable cause) {
20:         super(cause);
21:         // TODO Auto-generated constructor stub
22:     }
23:
24: }
```



```
1: package fernuni.propra.file_processing;
2:
3: public class PersistenceException extends Exception {
4:
5:     public PersistenceException() {
6:         super();
7:         // TODO Auto-generated constructor stub
8:     }
9:
10:    public PersistenceException(String message, Throwable cause) {
11:        super(message, cause);
12:        // TODO Auto-generated constructor stub
13:    }
14:
15:    public PersistenceException(String message) {
16:        super(message);
17:        // TODO Auto-generated constructor stub
18:    }
19:
20:    public PersistenceException(Throwable cause) {
21:        super(cause);
22:        // TODO Auto-generated constructor stub
23:    }
24:
25: }
```



```

1: package fernuni.propra.file_processing;
2:
3: import fernuni.propra.internal_data_model.IRoom;
4:
5: public class UserReadInputWriteOutputAAS {
6:     private final String location;
7:     private IPersistence persistence;
8:
9:     public UserReadInputWriteOutputAAS(String location) {
10:         this.location = location;
11:         this.persistence = new FilePersistence();
12:     }
13:
14:     public IRoom readInput() throws UserReadInputWriteOutputException {
15:         try {
16:             return persistence.readInput(location);
17:         } catch (PersistenceException e) {
18:             throw new UserReadInputWriteOutputException(e);
19:         }
20:     }
21:
22:     public void writeOutput(IRoom room) throws UserReadInputWriteOutputExc
ption {
23:         try {
24:             persistence.writeOutput(room, location);
25:         } catch (PersistenceException e) {
26:             throw new UserReadInputWriteOutputException(e);
27:         }
28:     }
29:
30: }

```



```

1: package fernuni.propra.internal_data_model;
2:
3: import java.util.Iterator;
4: import java.util.LinkedList;
5: import java.util.List;
6:
7:
8:
9: public class Room implements IRoom {
10:     private List<Lamp> lamps = new LinkedList<Lamp>();
11:     private final LinkedList<Point> corners;
12:     private boolean counterClockWise;
13:     private double minX, maxX, minY, maxY;
14:     private List<Wall> walls = new LinkedList<Wall>();
15:     private String ID;
16:
17:     public double getMinX() {
18:         return minX;
19:     }
20:
21:     public double getMaxX() {
22:         return maxX;
23:     }
24:
25:     public double getMinY() {
26:         return minY;
27:     }
28:
29:     public double getMaxY() {
30:         return maxY;
31:     }
32:
33:     public Room(String ID, List<Lamp> lamps, LinkedList<Point> corners) {
34:         if (lamps != null) {
35:             this.lamps = lamps;
36:         }
37:         this.corners = corners;
38:         this.ID = ID;
39:         computeDimensionAndOrientation();
40:     }
41:
42:
43:     @Override
44:     public Iterator<Lamp> getLamps() {
45:         return lamps.iterator();
46:     }
47:
48:     @Override
49:     public Iterator<Point> getCornerRadius() {
50:         if (counterClockWise) {
51:             return corners.iterator();
52:         } else {
53:             return corners.descendingIterator();
54:         }
55:     }
56:
57:     @Override
58:     public void addLamp(Lamp lamp) {
59:         lamps.add(lamp);
60:     }
61:
62:
63:     @Override
64:     public int getNumberOfLamps() {
65:         return lamps.size();
66:     }
67:
68:     private void computeDimensionAndOrientation() {
69:         if (corners.isEmpty()) {
70:             throw new IllegalArgumentException("Room does not have
71:             any corners!");
72:         } else {
73:             minX = corners.get(0).getX(); maxX = minX;
74:             minY = corners.get(0).getY(); maxY = minY;
75:
76:             Point mostBottomMostRightPoint = null;
77:             for (Point corner : corners) {
78:                 if (mostBottomMostRightPoint != null) {
79:                     if (corner.getY() <= mostBottomMostRightPoint.
80:                         getY()) {
81:                         if (corner.getX() > mostBottomMostRightP
82:                             .getX()) {
83:                             mostBottomMostRightPoint = cor
84:                                 ner;
85:                         } else {
86:                             mostBottomMostRightPoint = corner;
87:                         }
88:                     }
89:                     if (corner.getX() < minX) {
90:                         minX = corner.getX();
91:                     } else if (corner.getX() > maxX) {
92:                         maxX = corner.getX();
93:                     }
94:                     if (corner.getY() < minY) {
95:                         minY = corner.getY();
96:                     } else if (corner.getY() > maxY) {
97:                         maxY = corner.getY();
98:                     }
99:                 }
100:             }
101:             this.counterClockWise = isCounterClockWise(mostBottomMostRight
102:                 Point);
103:         }
104:     }
105:
106:     @Override
107:     public Iterator<Wall> getWalls() {
108:         if (walls.isEmpty()) {
109:             computeWalls();
110:         }
111:         return walls.iterator();
112:     }
113:
114:     private void computeWalls() {
115:         Point firstCorner = null;
116:         Point previousCorner = null;
117:
118:         Iterator<Point> cornersIterator = getCornerRadius();
119:         int tag = 0;
120:
121:         while (cornersIterator.hasNext()) {
122:             Point corner = cornersIterator.next();
123:             if (firstCorner == null) {
124:                 firstCorner = corner;
125:             } else {

```

```

124:                Wall newWall = new Wall(previousCorner, corner
, tag);
125:                walls.add(newWall);
126:            }
127:            previousCorner = corner;
128:            tag++;
129:        }
130:        Wall newWall = new Wall(previousCorner, firstCorner, tag);

131:        walls.add(newWall);
132:    }
133:
134:    @Override
135:    public String getID() {
136:        return this.ID;
137:    }
138:
139:
140:    private boolean isCounterClockWise(Point mostBottomMostRightPoint) {
141:        // https://stackoverflow.com/questions/1165647/how-to-determin
e-if-a-list-of-polygon-points-are-in-clockwise-order/1180256#1180256
142:        int indexOfBMRMP = corners.indexOf(mostBottomMostRightPoint);
143:        Point previous;
144:        Point next;
145:        if (indexOfBMRMP == 0) {
146:            previous = corners.get(corners.size()-1); //TODO this
is faster with ArrayList?
147:            next = corners.get(indexOfBMRMP+1);
148:        } else if (indexOfBMRMP == (corners.size()-1)) {
149:            previous = corners.get(indexOfBMRMP-1);
150:            next = corners.get(0);
151:        } else {
152:            previous = corners.get(indexOfBMRMP-1);
153:            next = corners.get(indexOfBMRMP+1);
154:        }
155:
156:        double dx1 = mostBottomMostRightPoint.getX()-previous.getX();
157:        double dx2 = next.getX() - mostBottomMostRightPoint.getX();
158:
159:        double dy1 = mostBottomMostRightPoint.getY()-previous.getY();
160:        double dy2 = next.getY() - mostBottomMostRightPoint.getY();
161:
162:        double crossProduct = dx1*dy2 - dx2*dy1;
163:        return crossProduct > 0;
164:    }
165:
166:    @Override
167:    public void replaceLamps(List<Lamp> lamps) {
168:        if (lamps != null) {
169:            this.lamps = lamps;
170:        }
171:    }
172:
173:    @Override
174:    public String printLampPositions() {
175:        int n = 1;
176:        String lineSeparator = System.getProperty("line.separator");
177:        StringBuilder sb = new StringBuilder();
178:        sb.append("The room contains ");
179:        sb.append(String.valueOf(this.lamps.size()));
180:        String singPl = lamps.size() == 1 ? " lamp." : " lamps.";
181:        sb.append(singPl);
182:        sb.append(lineSeparator);
183:        sb.append("The lamps are located at:");
184:
185:        sb.append(lineSeparator);
186:        for (Lamp lamp : lamps) {
187:            sb.append("Lamp ");
188:            sb.append(String.valueOf(n));
189:            sb.append(" located at x=");
190:            sb.append(String.valueOf(lamp.getX()));
191:            sb.append(" y=");
192:            sb.append(String.valueOf(lamp.getY()));
193:            sb.append(" The lamp is ");
194:            String onOff = lamp.getOn() ? "turned on." : "turned o
ff.";
195:            sb.append(onOff);
196:            sb.append(lineSeparator);
197:            n++;
198:        }
199:        String outString = sb.toString();
200:        return outString;
201:    }
202: }

```

./ProPra2020_workspace/File_Processing_Component/src/fernuni/propra/internal_data_model/LineSegmentException.java

Fri Jun

```
1: package fernuni.propra.internal_data_model;
2:
3: public class LineSegmentException extends Exception {
4:
5:     public LineSegmentException() {
6:         super();
7:         // TODO Auto-generated constructor stub
8:     }
9:
10:    public LineSegmentException(String message) {
11:        super(message);
12:        // TODO Auto-generated constructor stub
13:    }
14:
15:    public LineSegmentException(Throwable cause) {
16:        super(cause);
17:        // TODO Auto-generated constructor stub
18:    }
19:
20: }
```



```
1: package fernuni.propra.internal_data_model;
2:
3: import java.util.Iterator;
4: import java.util.List;
5:
6: public interface IRoom{
7:     Iterator<Lamp> getLamps();
8:     int getNumberOfLamps();
9:     Iterator<Point> getCorners();
10:    void addLamp(Lamp lamp);
11:    Iterator<Wall> getWalls();
12:    double getMinX();
13:    double getMaxX();
14:    double getMinY();
15:    double getMaxY();
16:    String getID();
17:    void replaceLamps(List<Lamp> lamps);
18:    String printLampPositions();
19:
20: }
21:
```



```

1: package fernuni.propra.internal_data_model;
2:
3: import java.util.ArrayList;
4: import java.util.List;
5:
6: public class Point {
7:     private final double x;
8:     private final double y;
9:     private final static double TOL = 0.0001;
10:    private final static int PRECISION = 1000;
11:    public final static double INF = 100000000;
12:
13:    public Point(double x, double y) {
14:        this.x = x;
15:        this.y = y;
16:    }
17:
18:    public double getX() {
19:        return x;
20:    }
21:
22:    public double getY() {
23:        return y;
24:    }
25:
26:    public boolean isEqual(Point other) {
27:        if (other == null) return false;
28:        return (Math.round(getX() * PRECISION) == Math.round(o
ther.getX() * PRECISION))
29:            && (Math.round(getY() * PRECISION) ==
Math.round(other.getY() * PRECISION));
30:        //return (Math.abs(getX()-other.getX()) + Math.abs(getY()-oth
er.getY())) < TOL;
31:    }
32:
33:    public boolean isOnLineSegment(Point p1, Point p2) {
34:        if (!p1.sameX(p2) && !p1.sameY(p2)) throw new IllegalArgumentException
xception("Input is not
35:        a horizontal or vertical line!");
36:        boolean xAgrees = this.sameX(p1)
37:            && this.sameX(p2);
38:        boolean yAgrees = Point.agrees(getY(), p1.getY())
39:            && Point.agrees(getY(), p2.getY());
40:        boolean xInRange = isInRange(getX(), p1.getX(), p2.getX());
41:        boolean yInRange = isInRange(getY(), p1.getY(), p2.getY());
42:        return (xAgrees && yInRange) || (yAgrees && xInRange);
43:    }
44:
45:    public boolean isOnLineSegment (LineSegment lineSegment) {
46:        return isOnLineSegment (lineSegment.getP1(), lineSegment.getP2(
));
47:    }
48:
49:    boolean sameX(Point other) {
50:        return Point.agrees(this.x, other.x);
51:    }
52:
53:    boolean sameY(Point other) {
54:        return Point.agrees(this.y, other.y);
55:    }
56:
57:    boolean largerX(Point other) {
58:        return Point.isLarger(this.x, other.x);
59:    }
60:
61:    boolean largerY(Point other) {
62:        return Point.isLarger(this.y, other.y);
63:    }
64:
65:    public boolean isInsidePolygon(List<LineSegment> lineSegments) {
66:        ArrayList<LineSegment> arrayLinesSegments = new ArrayList<Line
Segment>(lineSegments);
67:
68:        // pre lineSegment must be a valid polygonal
69:        LineSegment testLineSegXP = new LineSegment(this, new Point(IN
F, getY()));
70:        LineSegment testLineSegYP = new LineSegment(this, new Point(get
X(), INF));
71:
72:        int intersectionCountXP = 0;
73:        int intersectionCountYP = 0;
74:        for (LineSegment lineSegment : lineSegments) {
75:            try {
76:                testLineSegXP.intersectionWithLineSegment(line
Segment);
77:                if (isOnLineSegment(lineSegment.getP1(), lineS
egment.getP2())) {
78:                    return true; // if point is on wall ->
79:                } else {
80:                    intersectionCountXP++;
81:                }
82:                /*if (lineSegment.getP1().isOnLineSegment(test
LineSeg.getP1(), testLineSeg.getP2()) ||
83:                    lineSegment.getP2().isOnLineSegment(te
stLineSeg.getP1(), testLineSeg.getP2())) {
84:                    intersectedLineSegmentHasEndPointOnTes
tLineSegCount ++;
85:                } */
86:            } catch (LineSegmentException e) {
87:            }
88:
89:            try {
90:                testLineSegYP.intersectionWithLineSegment(line
Segment);
91:                if (isOnLineSegment(lineSegment.getP1(), lineS
egment.getP2())) {
92:                    return true; // if point is on wall ->
93:                } else {
94:                    intersectionCountYP++;
95:                }
96:                /*if (lineSegment.getP1().isOnLineSegment(test
LineSeg.getP1(), testLineSeg.getP2()) ||
97:                    lineSegment.getP2().isOnLineSegment(te
stLineSeg.getP1(), testLineSeg.getP2())) {
98:                    intersectedLineSegmentHasEndPointOnTes
tLineSegCount ++;
99:                } */
100:            } catch (LineSegmentException e) {
101:            }
102:
103:
104:        }
105:
106:        if ((intersectionCountXP % 2) != 0 || (intersectionCountYP % 2
) != 0) { // if number of intersections is odd -> point is in polygonal
107:            return true;

```

```

108:         } else {
109:             return false;
110:         }
111:     }
112:
113:     public boolean isInXRange(double xLow, double xHigh) {
114:         if (xLow>xHigh) throw new IllegalArgumentException("xLow > xHi
gh");
115:         return isInRange(getX(), xLow, xHigh);
116:     }
117:
118:     public boolean isInYRange(double yLow, double yHigh) {
119:         if(yLow > yHigh) throw new IllegalArgumentException("yLow > yH
igh");
120:         return isInRange(getY(), yLow, yHigh);
121:     }
122:
123:     public boolean isInsideRectangle(Point p1, Point p3) {
124:         return isInXRange(p1.getX(), p3.getX()) && isInYRange(p1.getY(
), p3.getY());
125:     }
126:
127:     private static boolean agrees(double x, double x1) {
128:         return Math.abs(x-x1)<TOL;
129:     }
130:
131:     private static boolean isLarger(double x, double x1) {
132:         return x-x1 > TOL;
133:     }
134:
135:     private static boolean isInRange(double x, double x1, double x2) {
136:         return (Math.min(x1, x2) < x + TOL) && (Math.max(x1, x2) > x -
TOL);
137:     }
138:
139:     @Override
140:     public boolean equals(Object o) {
141:         if (o == this) return true;
142:         if (!(o instanceof Point)) {
143:             return false;
144:         }
145:
146:         Point point = (Point) o;
147:
148:         return isEqual(point);
149:     }
150:
151:     @Override
152:     public int hashCode() {
153:         int result = 17;
154:         result = 31 * result + (int) Math.round(x * PRECISION);
155:         result = 31 * result + (int) Math.round(y * PRECISION);
156:         return result;
157:     }
158:
159:
160:
161:
162:
163: }

```

```

1: package fernuni.propra.internal_data_model;
2:
3: import java.util.List;
4:
5: public class LineSegment {
6:     private final static double TOL = 0.0001;
7:     private final Point p1;
8:     private final Point p2;
9:
10:
11:     public LineSegment(Point p1, Point p2) {
12:         // pre p1 != 0, p2 != 0
13:         this.p1 = p1;
14:         this.p2 = p2;
15:     }
16:
17:     public Point getP1() {
18:         return p1;
19:     }
20:
21:     public Point getP2() {
22:         return p2;
23:     }
24:
25:     public boolean isHorizontal() {
26:         return p1.sameY(p2) && !p1.isEqual(p2);
27:     }
28:
29:     public boolean isVertical() {
30:         return p1.sameX(p2) && !p1.isEqual(p2);
31:     }
32:
33:     public boolean overlapsXrange(double xLow, double xHigh) {
34:         if (xLow > xHigh) throw new IllegalArgumentException("xLow > xHi
gh! Insert a valid range!");
35:         // pre xLow < xHigh
36:         boolean p1IsInRange = p1.isInXRange(xLow, xHigh);
37:         boolean p2IsInRange = p2.isInXRange(xLow, xHigh);
38:         boolean p1SmallerP2Greater = (new Point(xLow, p1.getY()).large
rX(getP1())) &&
39:             (getP2().largerX(new Point(xHigh, p2.getY())))
40:         ;
41:         boolean p2SmallerP1Greater = (new Point(xLow, p2.getY()).large
rX(getP2())) &&
42:             (getP1().largerX(new Point(xHigh, p1.getY())))
43:         ;
44:         return p1IsInRange || p2IsInRange || p1SmallerP2Greater || p2
SmallerP1Greater;
45:     }
46:
47:     public boolean overlapsYrange(double yLow, double yHigh) {
48:         if (yLow > yHigh) throw new IllegalArgumentException("yLow > yHi
gh! Insert a valid range!");
49:         // pre yLow < yHigh
50:         boolean p1IsInRange = p1.isInYRange(yLow, yHigh);
51:         boolean p2IsInRange = p2.isInYRange(yLow, yHigh);
52:         boolean p1SmallerP2Greater = (new Point(p1.getX(), yLow).larger
Y(getP1())) &&
53:             (getP2().largerY(new Point(p2.getX(), yHigh)))
54:         ;
55:         boolean p2SmallerP1Greater = (new Point(p2.getY(), yLow).large
rY(getP2())) &&
56:             (getP1().largerY(new Point(p1.getX(), yHigh)));

```

```

55:
56:         return p1IsInRange || p2IsInRange || p1SmallerP2Greater || p2
SmallerP1Greater;
57:     }
58:
59:
60:     public boolean perpendicular(Point p1, Point p2) {
61:         double dx = this.p2.getX() - this.p1.getX();
62:         double dy = this.p2.getY() - this.p1.getY();
63:
64:         double dxOther = p2.getX() - p1.getX();
65:         double dyOther = p2.getY() - p1.getY();
66:
67:         double scalarProduct = dx * dxOther + dy * dyOther;
68:
69:         return Math.abs(scalarProduct) < TOL ;
70:     }
71:
72:     public boolean perpendicular(LineSegment other) {
73:         return perpendicular(other.getP1(), other.getP2());
74:     }
75:
76:     public boolean penetratesLineSegments(List<LineSegment> lineSegments)
{
77:         if (lineSegments.isEmpty()) return false;
78:         boolean penetrates = false;
79:         for (int j = 0; j < lineSegments.size(); j++) {
80:             LineSegment tmpLineSegment = lineSegments.get(j);
81:             //if (getP1().isOnLineSegment(tmpLineSegment) || getP2
().isOnLineSegment(tmpLineSegment)) {
82:                 // return false;
83:                 //}
84:                 try {
85:                     Point intersectionPoint = this.intersectionWit
hLineSegment(tmpLineSegment);
86:                     if (!intersectionPoint.isEqual(this.getP1()) &
&
87:                         !intersectionPoint.isEqual(thi
s.getP2())) {
88:                         penetrates = true;
89:                         break;
90:                     }
91:                 } catch (LineSegmentException e) {
92:                     }
93:             }
94:         }
95:         return penetrates;
96:     }
97:
98:     public Point intersectionWithLineSegment(LineSegment other) throws Lin
eSegmentException {
99:         if (perpendicular(other)) {
100:             double x;
101:             double y;
102:             if (isHorizontal()) {
103:                 y = getP1().getY();
104:                 x = other.getP1().getX();
105:             } else {
106:                 y = other.getP1().getY();
107:                 x = getP1().getX();
108:             }
109:             Point outPoint = new Point(x, y);
110:             if (outPoint.isOnLineSegment(other) && outPoint.isOnLi

```

```
neSegment(this)) {
    111:                                return outPoint;
    112:                                } else {
    113:                                throw new LineSegmentException("Line Segments
do not intersect!");
    114:                                }
    115:                                } else {
    116:                                throw new LineSegmentException("Lines are not perpendi
cular cannot return (unique) intersection point!");
    117:                                }
    118:
    119:                                }
    120:
    121:                                public boolean isEqual(LineSegment other) {
    122:                                return getP1().isEqual(other.getP1()) && getP2().isEqual(other
.getP2());
    123:                                }
    124:
    125: }
```

```
1: package fernuni.propra.internal_data_model;
2:
3:
4: public class Wall extends LineSegment{
5:     private int tag;
6:
7:
8:     public Wall(Point p1, Point p2, int tag) {
9:         super(p1, p2);
10:        this.tag = tag;
11:    }
12:
13:    public Wall(LineSegment lineSegment, int tag) {
14:        super(lineSegment.getP1(), lineSegment.getP2());
15:        this.tag = tag;
16:    }
17:
18:    public boolean isNorthWall() {
19:        return isHorizontal() && getP1().largerX(getP2());
20:    }
21:
22:    public boolean isWestWall() {
23:        return isVertical() && getP1().largerY(getP2());
24:    }
25:
26:    public boolean isSouthWall() {
27:        return isHorizontal() && getP2().largerX(getP1());
28:    }
29:
30:    public boolean isEastWall() {
31:        return isVertical() && getP2().largerY(getP1());
32:    }
33:
34:    public int getTag() {
35:        return tag;
36:    }
37:
38:
39:
40: }
```



```

1: package fernuni.propra.internal_data_model;
2:
3: import java.util.HashSet;
4: import java.util.Iterator;
5:
6: public class Lamp extends Point{
7:     private volatile boolean on;
8:     HashSet<Integer> tagsOfCoveredRectangles = new HashSet<Integer>();
9:
10:    public Lamp(double x, double y) {
11:        super(x, y);
12:    }
13:
14:    public Lamp(double x, double y, int tag) {
15:        super(x, y);
16:        tagsOfCoveredRectangles.add(tag);
17:    }
18:
19:    public Lamp(double x, double y, HashSet<Integer> tags) {
20:        super(x,y);
21:        if (tags != null) {
22:            this.tagsOfCoveredRectangles = tags;
23:        }
24:    }
25:
26:    public void addTag(Integer tag) {
27:        tagsOfCoveredRectangles.add(tag);
28:    }
29:
30:    public HashSet<Integer> getCopyOfTags() {
31:        HashSet<Integer> outTags = new HashSet<Integer>();
32:        for (Integer tag : tagsOfCoveredRectangles) {
33:            outTags.add(tag);
34:        }
35:        return outTags;
36:    }
37:
38:    public Iterator<Integer> iteratorTag() {
39:        return tagsOfCoveredRectangles.iterator();
40:    }
41:
42:    public void turnOn() {
43:        on = true;
44:    }
45:
46:    public void turnOff() {
47:        on = false;
48:    }
49:
50:    public boolean getOn() {
51:        return on;
52:    }
53:
54:    public Lamp deepCopy() {
55:        Lamp outLamp = new Lamp(this.getX(),this.getY());
56:        for(Integer tag : this.tagsOfCoveredRectangles) {
57:            outLamp.addTag(tag);
58:        }
59:        if (this.on) {
60:            outLamp.turnOn();
61:        }
62:        return outLamp;
63:    }
64:
65:
66: }
```



```

1: package fernuni.propra.user_interface;
2:
3: import java.awt.BasicStroke;
4: import java.awt.Color;
5: import java.awt.Dimension;
6: import java.awt.Graphics;
7: import java.awt.Graphics2D;
8: import java.awt.RenderingHints;
9: import java.awt.geom.AffineTransform;
10:
11:
12: import javax.swing.JPanel;
13:
14: import fernuni.propra.internal_data_model.IRoom;
15:
16: public abstract class RoomPanelAbstract extends JPanel{
17:     private IRoom room;
18:     private double scale;
19:
20:     public RoomPanelAbstract(IRoom room) {
21:         this.room = room;
22:         setBackground(Color.WHITE);
23:         setPreferredSize(new Dimension(1024, 768));
24:     }
25:
26:     String getRoomID() {
27:         return room.getID();
28:     }
29:
30:     @Override
31:     protected void paintComponent(Graphics g) {
32:         super.paintComponent(g);
33:
34:         Graphics2D g2D= (Graphics2D) g;
35:         g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING, Renderin
gHints.VALUE_ANTIALIAS_ON);
36:
37:         transformToRoomCoordinates(g2D);
38:         drawRoom(g2D);
39:
40:         scaleBackToScreenCoordinates(g2D);
41:         drawLamps(g2D);
42:         drawRectangles(g2D); // TODO nur Test
43:
44:         g2D.scale(1, -1);
45:         g2D.translate(0, -getHeight());
46:
47:     }
48:
49:     protected abstract void drawRectangles(Graphics2D g2D);
50:
51:     protected abstract void drawLamps(Graphics2D g2D) ;
52:
53:     protected abstract void drawRoom(Graphics2D g2D) ;
54:
55:     private void scaleBackToScreenCoordinates(Graphics2D g2D) {
56:         AffineTransform myTransform;
57:         myTransform = AffineTransform.getScaleInstance(1/scale, 1/scale);
58:
59:         g2D.transform(myTransform);
60:     }
61:
62:     private void transformToRoomCoordinates(Graphics2D g2D) {
63:         double sy = (0.9 * getHeight()) / (room.getMaxY() - room.getMinY(
));
64:
65:         scale = Math.min(sx, sy);
66:         double centerOffset = 0.5*Math.min(0.1*getWidth(), 0.1*getHeigh
t());
67:         AffineTransform myTransform = AffineTransform.getScaleInstance
(scale, scale);
68:         g2D.transform(myTransform);
69:         myTransform = AffineTransform.getTranslateInstance(-room.getMi
nX() + centerOffset/scale, -room.getMinY() + centerOffset/scale);
70:         g2D.transform(myTransform);
71:     }
72:
73:     protected IRoom getRoom() {
74:         return this.room;
75:     }
76:
77:     protected double getScale() {
78:         return this.scale;
79:     }
80:
81: }

```



```
1: package fernuni.propra.user_interface;
2: import java.awt.BorderLayout;
3:
4: import javax.swing.JFrame;
5:
6: public class RoomFrame extends JFrame{
7:     RoomPanelAbstract roomPanel;
8:
9:     public RoomFrame(RoomPanelAbstract roomPanel) {
10:         this.roomPanel = roomPanel;
11:         setTitle(this.roomPanel.getRoomID());
12:         add(roomPanel, BorderLayout.CENTER);
13:         pack();
14:         setDefaultCloseOperation(DISPOSE_ON_CLOSE);
15:         setLocationRelativeTo(null);
16:         setVisible(true);
17:     }
18:
19:
20: }
```



```

1: package fernuni.propra.user_interface;
2: import java.awt.BasicStroke;
3: import java.awt.Color;
4: import java.awt.Dimension;
5: import java.awt.Graphics;
6: import java.awt.Graphics2D;
7: import java.awt.Polygon;
8: import java.awt.Rectangle;
9: import java.awt.RenderingHints;
10: import java.awt.geom.AffineTransform;
11: import java.util.ArrayList;
12: import java.util.Iterator;
13: import java.util.List;
14:
15: import javax.swing.JPanel;
16: import fernuni.propra.internal_data_model.IRoom;
17: import fernuni.propra.internal_data_model.Lamp;
18: import fernuni.propra.internal_data_model.LineSegment;
19: import fernuni.propra.internal_data_model.Point;
20: import fernuni.propra.internal_data_model.Wall;
21:
22: public class RoomPanel extends JPanelAbstract {
23:     private static final int PIXEL_LAMP_DIAMETER = 10; // in pixels
24:     private List<PlotRectangle> rectangles = new ArrayList<PlotRectangle>();
25:
26:     public RoomPanel(IRoom room) {
27:         super(room);
28:     }
29:
30:     private static class PlotRectangle { //TODO only for testing
31:         private double x,y,width,height;
32:         private Color color;
33:         private String name;
34:
35:         public PlotRectangle(String name, Color color, double x, double y, double width, double height) {
36:             this.x = x;
37:             this.y = y;
38:             this.width = width;
39:             this.height = height;
40:             this.name = name;
41:             this.color = color;
42:         }
43:     }
44:
45:     public void addRectangle(String name, Color color, double x, double y, double width, double height) { //TODO only for testing
46:         rectangles.add(new PlotRectangle(name, color, x, y, width, height));
47:     }
48:
49:     public void removeLastRectangle() { // TODO for Debug
50:         if (rectangles.size() > 0) {
51:             rectangles.remove(rectangles.size() - 1);
52:         }
53:     }
54:
55:
56:     @Override
57:     protected void drawLamps(Graphics2D g2D) {
58:         double scale = getScale();
59:         IRoom room = getRoom();
60:         Iterator<Lamp> lampIterator = room.getLamps();
61:         while (lampIterator.hasNext()) {
62:             Lamp lamp = lampIterator.next();
63:             Color lampColor = lamp.getOn() ? Color.YELLOW : Color.DARK_GRAY;
64:             g2D.setColor(lampColor);
65:             g2D.fillOval((int) (lamp.getX() * scale) - (int) Math.round(PIXEL_LAMP_DIAMETER/2.0),
66:                 (int) (lamp.getY() * scale) - (int) Math.round(PIXEL_LAMP_DIAMETER/2.0), PIXEL_LAMP_DIAMETER, PIXEL_LAMP_DIAMETER);
67:             g2D.setStroke(new BasicStroke(2));
68:             g2D.setColor(Color.BLACK);
69:             g2D.drawOval((int) (lamp.getX() * scale) - (int) Math.round(PIXEL_LAMP_DIAMETER/2.0),
70:                 (int) (lamp.getY() * scale) - (int) Math.round(PIXEL_LAMP_DIAMETER/2.0), PIXEL_LAMP_DIAMETER, PIXEL_LAMP_DIAMETER);
71:         }
72:     }
73:
74:
75:     @Override
76:     protected void drawRoom(Graphics2D g2D) {
77:         double scale = getScale();
78:         IRoom room = getRoom();
79:         g2D.setStroke(new BasicStroke(2/((float) scale)));
80:         Polygon p = new Polygon();
81:
82:         Iterator<Point> cornerIterator = room.getCornerRadius();
83:         while (cornerIterator.hasNext()) {
84:             Point corner = cornerIterator.next();
85:             p.addPoint((int) (corner.getX() * scale), (int) (corner.getY() * scale));
86:         }
87:
88:         g2D.setColor(Color.ORANGE);
89:         g2D.fillPolygon(p);
90:
91:         g2D.setColor(Color.BLACK);
92:         Iterator<Wall> wallIterator = room.getWalls();
93:         while (wallIterator.hasNext()) {
94:             LineSegment wall = wallIterator.next();
95:             g2D.drawLine((int) wall.getP1().getX() * scale, (int) wall.getP1().getY() * scale, (int) wall.getP2().getX() * scale, (int) wall.getP2().getY() * scale);
96:         }
97:
98:     }
99:
100:
101:     @Override
102:     protected void drawRectangles(Graphics2D g2D) { //TODO nur Test
103:         double scale = getScale();
104:         g2D.setStroke(new BasicStroke(2));
105:         for (PlotRectangle rectangle: rectangles) {
106:             g2D.setColor(rectangle.color);
107:             g2D.drawRect((int) (rectangle.x * scale), (int) (rectangle.y * scale), (int) (rectangle.width * scale), (int) (rectangle.height * scale));
108:             g2D.drawString(rectangle.name, (int) ((rectangle.x + rectangle.width/2) * scale), (int) ((rectangle.y + rectangle.height/2) * scale));
109:         }
110:     }

```

`./ProPra2020_workspace/User_Interface_Component/src/fernuni/propra/user_interface/RoomPanel.java`

Tue Jun 02 17:06:49 2020

```
114:         }  
115:  
116:  
117: }
```


./ProPra2020_workspace/User_Interface_Component/src/fernuni/propra/user_interface/UserDisplayAAS.java

Wed Apr 15 13:49:09

```
1: package fernuni.propra.user_interface;
2: import fernuni.propra.internal_data_model.IRoom;
3:
4: public class UserDisplayAAS {
5:     public void display(IRoom room) {
6:         RoomPanelAbstract roomPanel = new RoomPanel(room);
7:         RoomFrame roomFrame = new RoomFrame(roomPanel);
8:     }
9:
10: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.List;
4:
5: import fernuni.propra.algorithm.runtime_information.IRuntimePositionOptimizer;
6: import fernuni.propra.internal_data_model.IRoom;
7: import fernuni.propra.internal_data_model.Lamp;
8: import fernuni.propra.internal_data_model.Point;
9:
10: /**
11:  * A provider of an algorithm that finds a minimum set (and number) of tagged
12:  * {@link Lamp}s that
13:  *   illuminates an {@link IRoom} instance.
14:  *   <p>
15:  *   Implementing classes: {@link PositionOptimizer}
16:  *   @author alex
17:  *   </p>
18:  *   public interface IPositionOptimizer {
19:  *       /** A method that initiates the computation of an optimal (i.e. a set
20:  *       with a minimum number of lamps
21:  *       *   that are turned on) set of lamps, represented by a
22:  *       *   {@link List}<{@link Lamp}>.
23:  *       *   <p>
24:  *       *   The set of {@link Lamp}s that should be minimized is supplied as a
25:  *       *   {@link List} of
26:  *       *   tagged {@link Lamp}s. The tags of each {@link Lamp} uniquely denote
27:  *       *   the portion an {@link IRoom} that is
28:  *       *   illuminated by each {@link Lamp}. The union of all such tags must b
29:  *       *   e equivalent to all portions of the {@link IRoom}
30:  *       *   i.e. if the union of all tags associated with illuminated {@link L
31:  *       *   amp}s is equal to the set of all tags, the {@link IRoom}
32:  *       *   must be illuminated.
33:  *       *   <p>
34:  *       *   Detailed runtime information can be stored to an {@link IRuntimePos
35:  *       *   itionOptimizer} instance
36:  *       *   <p>
37:  *       *   The computation may be interrupted, by interrupting the executing t
38:  *       *   hread. Implementations need to guarantee
39:  *       *   that an {@link InterruptedException} is thrown as fast as possible
40:  *       *   in this case.
41:  *       *   <p>
42:  *       *   @param taggedCandidates : <{@link List}<{@link Lamp}>> . The set of
43:  *       *   {@link Lamp}s that should be minimized is supplied as a {@link List} of
44:  *       *   tagged {@link Lamp}s. The tags of each {@link Lamp} uniquely denot
45:  *       *   e the portion an {@link IRoom} that is
46:  *       *   illuminated by each {@link Lamp}. The union of all such tags must b
47:  *       *   e equivalent to all portions of the {@link IRoom}
48:  *       *   i.e. if the union of all tags associated with illuminated {@link L
49:  *       *   amp}s is equal to the set of all tags, the {@link IRoom}
50:  *       *   must be illuminated.
51:  *       *   @param runTimeInformation : Detailed runtime information can be sto
52:  *       *   red to an {@link IRuntimePositionOptimizer} instance
53:  *       *   @return A {@link List}<{@link Lamp}> that represents the best solu
54:  *       *   tion (i.e. a set with a minimum number of {@link Lamp}s
55:  *       *   that are turned on) after the computation has finished.
56:  *       *   @throws InterruptedException
57:  *       *   </p>
58:  *       *   List<Lamp> optimizePositions( List<Lamp> taggedCandidates,
59:  *       *       IRuntimePositionOptimizer runTimeInformation) throws I
60:  *       *       nterruptedException;
61:  *       *   </p>
62:  *       *   A method that allows to get the currently available best solution (

```

```

i.e. a set with a minimum number of {@link Lamp}s
49:     *   that are turned on) as a {@link List}<{@link Lamp}>, where
50:     *   a minimum number of {@link Lamp}s is turned on that still illuminat
51:     *   es the {@link IRoom} represented by the tags of the Lamps.
52:     *   <p>
53:     *   Should only be called after {@link optimizePositions}. Otherwise no
54:     *   solution is available and 0 will be returned.
55:     *   <p>
56:     *   @return A list of {@link Lamp}s that represents the currently avail
57:     *   able best solution.
58:     *   </p>
59:     *   List<Lamp> getCurrentBestSolution();
60:     *   </p>
61:     *   Returns the number of turned on {@link Lamp}s in the currently avai
62:     *   lable best solution.
63:     *   <p>
64:     *   Should only be called after {@link optimizePositions}. Otherwise no
65:     *   solution is available and 0 will be returned.
66:     *   @return The number of turned on {@link Lamp}s in the currently avai
67:     *   lable best solution.
68:     *   </p>
69:     *   int getNumberOfOnLampsBestSolution();
70:     *   </p>

```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.Comparator;
4: import java.util.Iterator;
5: import java.util.LinkedList;
6: import java.util.List;
7:
8: import fernuni.propra.internal_data_model.Wall;
9:
10: /**
11:  * An abstract class that represent a generic {@link Wall} container for a cer
tain type of {@link Wall}s,
12:  * where the type of the {@link WallContainerAbstract} must be specified by im
plementing subclasses.
13:  * <p>
14:  * {@link Wall}s in the container are sorted according to total order specifie
d by a {@link Comparator}<{@link Wall}>,
15:  * where the ordering must be specified by implementing subclasses. The total
ordering is done
16:  * by mapping {@link Wall}s on double numbers and
17:  * using the total ordering defined by the ordering of real numbers.
18:  * <p>
19:  * {@link Wall}s can be added if they are of the correct type.
20:  * <p>
21:  * The {@link WallContainerAbstract} can also return the nearest ( in the sens
e of the specified ordering), valid wall, where
22:  * validity of a {@link Wall} must be specified by implementing subclasses.
23:  * <p>
24:  * Implements the template pattern, where templates are given for the two core
functionalities, i.e. adding walls
25:  * and obtaining a nearest wall in some sense. Subclasses must fill in the bla
nks by implementing the abstract methods.
26:  * <p>
27:  * Implementing classes : {@link WallContainerEast}, {@link WallContainerNorth
}, {@link WallContainerSouth}, {@link WallContainerWest}
28:  * <p>
29:  * @author alex
30:  *
31:  */
32: public abstract class WallContainerAbstract implements Iterable<Wall> {
33:
34:     protected List<Wall> walls; // the walls in this container
35:
36:     /**
37:      * Constructor
38:      */
39:     public WallContainerAbstract() {
40:         walls = new LinkedList<Wall>();
41:     }
42:
43:     /**
44:      * A method that allows to add a {@link Wall} of the correct type to t
he container.
45:      *
46:      * @param wall : the {@link Wall} to be added
47:      * @throws WallContainerException : thrown if wall is not of the corre
ct type.
48:      */
49:     public void add(Wall wall) throws WallContainerException{
50:         if (!isCorrectWallType(wall)) throw new WallContainerException
("Wall does not "
51:         + "have the correct orientation for this wall
container!");
52:         walls.add(wall);

```

```

53:         walls.sort(getComparator());
54:     }
55:
56:     /**
57:      * A method that allows to search for the next {@link Wall} (in the se
nse of the total ordering defined
58:      * by the implementing subclass). The search can be further specified
by defining a range of doubles,
59:      * whose meaning also needs to be specified by the implementing subcla
sses and the clients.
60:      * @param low : lower limit of the range
61:      * @param high : upper limit of the range
62:      * @param limit : limit to be used to find the next wall according to
the ordering.
63:      * @return the next {@link Wall}
64:      * @throws WallContainerException : if range is not set correctly
65:      */
66:     public Wall getNearestWall(double low, double high, double limit) thro
ws WallContainerException{
67:         if (low > high) throw new WallContainerException(); // not a v
alid range
68:         Iterator<Wall> iterator = walls.iterator(); // walls are order
ed
69:         Wall nextWall;
70:         while(iterator.hasNext()) {
71:             nextWall = iterator.next();
72:             if (isValidWall(nextWall, limit, low, high)) { // wall
fits the range and is also the next wall according to the ordering
73:                 return nextWall;
74:             }
75:         }
76:         return null;
77:     }
78:
79:     /**
80:      * Provides access to all the walls in the container by returning an i
terator.
81:      */
82:     public Iterator<Wall> iterator() { // TODO return only a copy?
83:         return walls.iterator();
84:     }
85:
86:     /**
87:      * Computes whether a {@link Wall} has is indeed a subsequent - relati
ve to the limit - {@link Wall} with respect to to the
88:      * total ordering of this container.
89:      * @param wall
90:      * @param limit : limit that characterizes the wall and is used to dec
ide if wall is indeed a subsequent {@link Wall} with respect to the
91:      * total ordering of {@link Wall}s in the container.
92:      * @param low : lower end of the range that is used to further specify
the search for the next {@link Wall}
93:      * @param high : higher end of the range that is used to further speci
fy the search for the next {@link Wall}
94:      * @return a boolean that shows whether the {@link Wall} is a valid {@
link Wall} that matches the semantics of
95:      * {@link getNearestWall}
96:      */
97:     protected abstract boolean isValidWall(Wall wall, double limit, double
low, double high);
98:
99:     /**
100:      *
101:      * @return A {@link Comparator}<{@link Wall}> that specifies the total

```

./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/WallContainerAbstract.java

Fri Jun 26 15:55:16 20

```
ordering that is used in this container.
102:      */
103:      protected abstract Comparator<Wall> getComparator();
104:
105:      /**
106:       *
107:       * @param wall : the {@link Wall} to be checked.
108:       * @return a boolean that indicates whether wall has the suited type f
or this container.s
109:       */
110:      protected abstract boolean isCorrectWallType(Wall wall);
111: }
```

```
1: package fernuni.propra.algorithm;
2: /**
3:  * This exception is thrown if something went wrong within the methods of a {@@
link WallContainerAbstract}.
4:  * <p>
5:  * @author alex
6:  *
7:  */
8: public class WallContainerException extends Exception {
9:
10:     public WallContainerException(String message) {
11:         super(message);
12:     }
13:
14:     public WallContainerException(Throwable cause) {
15:         super(cause);
16:     }
17:
18:     public WallContainerException() {
19:         super();
20:     }
21:
22: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.HashSet;
4: import java.util.Iterator;
5:
6: import fernuni.propra.algorithm.runtime_information.IRuntimeIlluminationTester
;
7: import fernuni.propra.internal_data_model.IRoom;
8: import fernuni.propra.internal_data_model.Lamp;
9:
10: /**
11:  * A provider of an algorithm that tests if a room is illuminated by a number
of lamps.
12:  * <p>
13:  * Implementing classes: {@link IIlluminationTester}
14:  * <p>
15:  * @author alex
16:  *
17:  */
18: public interface IIlluminationTester {
19:     /**
20:      * Tests whether an instance of {@link IRoom} is illuminated, by the {
{@link Lamp}s that are part of that {@link IRoom}.
21:      * @param room : The {@link IRoom} instance to be checked (must contain
information about the {@link Lamp}s)
22:      * @param runtimeInfo : A data structure of type {@link IRuntimeIllumi
nationTester} that can store runtime information.
23:      * @return A boolean that represents whether room is illuminated (true
) or not (false).
24:      * @throws IlluminationTesterException
25:      */
26:     boolean testIfRoomIsIlluminated(IRoom room, IRuntimeIlluminationTester
runtimeInfo) throws IlluminationTesterException;
27:
28:     /**
29:      * Tests whether an a room is illuminated.
30:      * @param taggedLampsIterator : An Iterator for a set of {@link Lamp}s
that are tagged. Each tag represents a portion
31:      * of the room that is illuminated by that {@link Lamp}.
32:      * @param allTags : The tags that represent all portions of the room.
The union of all tagged portions of the room
33:      *
retrieves the room.
34:      * @param runtimeInfo : A data structure of type {@link IRuntimeIllumi
nationTester} that can store runtime information.
35:      * @return
36:      */
37:     boolean testIfRoomIsIlluminated(Iterator<Lamp> taggedLampsIterator, Ha
shSet<Integer> allTags, IRuntimeIlluminationTester runtimeInfo);
38:
39:     /**
40:      * Tests whether an a room is illuminated.
41:      * @param illuminatedTags : A set of tags that represents illuminated
portions of the room. Each tag represents
42:      *
a portion of t
he room. The union of all tagged portions of the room retrieves the room.
43:      * @param allTags : The tags that represent all portions of the room.
The union of all tagged portions of the room
44:      *
retrieves the room.
45:      * @param runtimeInfo : A data structure of type {@link IRuntimeIllumi
nationTester} that can store runtime information.
46:      * @return
47:      */
48:     boolean testIfRoomIsIlluminated(HashSet<Integer> illuminatedTags, Hash
Set<Integer> allTags, IRuntimeIlluminationTester runtimeInfo);

```

49: }


```

1: package fernuni.propra.algorithm;
2:
3: import java.util.ArrayList;
4: import java.util.HashSet;
5: import java.util.Iterator;
6: import java.util.LinkedList;
7: import java.util.List;
8:
9: import fernuni.propra.algorithm.runtime_information.IRuntimeCandidateSearcher;
10: import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
11: import fernuni.propra.algorithm.util.Rectangle;
12: import fernuni.propra.algorithm.util.RectangleWithTag;
13: import fernuni.propra.internal_data_model.IRoom;
14: import fernuni.propra.internal_data_model.Lamp;
15: import fernuni.propra.internal_data_model.Point;
16:
17: /**
18:  *
19:  * A specific provider of an algorithm that can compute a {@link List} of pote
ntial {@link Lamp} positions
20:  * for an instance of {@link IRoom}.
21:  * <p>
22:  * The algorithm works as follows:
23:  * <p>
24:  * 1.) The original partial rectangles (instances of {@link RectangleWithTag})
of the room are constructed for {@link IRoom} according the method described in [1].
25:  * This is delegated to {@link OriginalPartialRectanglesFinder}. The set o
f potential lamp positions is initialized as the returned set.
26:  * <p>
27:  * 2.) All pairs of original partial rectangles are intersected. If an overla
p is found,
28:  * the resulting rectangle is added to the set of potential lamp
positions and the tags of both original
29:  * rectangles are added to the tags of the new rectangle
30:  * <p>
31:  * 3.) The set of potential lamp positions is reduced by only keeping those l
amp positions whose tags are not a subset of
32:  * the tags of other rectangles in the set
33:  * <p>
34:  * 4.) Steps 2.) and 3.) are repeated until the set does not change any more
35:  * <p>
36:  * 5.) The potential lamp positions are the centers of the remaining tagged r
ectangles
37:  * <p>
38:  * 6.) {@link Lamp} objects are created at these {@link Point}s and the lamps are
tagged with the tags of
39:  * the corresponding tagged {@link RectangleWithTag}, i.e. the tags of al
l original partial rectangles
40:  * of the room that contain the {@link Lamp} are saved to the {@link Lamp}s ta
gs.
41:  * <p>
42:  * 7.) A {@link List} of all such {@link Lamp}s is returned.
43:  *
44:  * <p>
45:  * Implemented interfaces and super classes: {@link ICandidateSearcher}
46:  *
47:  * <p>
48:  * <p>
49:  * [1]: Aufgabenstellung zum Grundpraktikum Programmierung im Sommersemester 2
020
50:  *
51:  *
52:  *
53:  * @author alex
54:  *
55:  */
56: public class CandidateSearcher implements ICandidateSearcher{
57:
58:
59:     public CandidateSearcher() {
60:     }
61:
62:     @Override
63:     public List<Lamp> searchCandidates(IRoom room, IRuntimeCandidateSearch
er runtimeCandidateSearcher) throws CandidateSearcherException, InterruptedException {
64:         List<Lamp> centersOfReducedRectangles = null; // the potential
lamp positions
65:         try {
66:             // find original partial rectangles
67:             runtimeCandidateSearcher.startTimeOriginalPartialRecta
nglesFind(); // store runtime for construction of original partial rectangles
68:             ArrayList<RectangleWithTag> originalRectangles =
AbstractAlgorithmFactory.getAlgorithmF
actory().createOriginalPartialRectanglesFinder().
69:                 findOriginalPartialRectangles(room, ru
ntimeCandidateSearcher);
70:             runtimeCandidateSearcher.stopTimeOriginalPartialRectan
glesFind();
71:             // reduce rectangles: result is non overlapping set of
rectangles. Each rectangle contains all tags of
72:             // original rectangles that it overlaps
73:             List<RectangleWithTag> reducedRectangles = reduceRecta
ngles(originalRectangles);
74:
75:             // create lamp objects at each potential position
76:             centersOfReducedRectangles = new LinkedList<Lamp>();
77:             for (RectangleWithTag rectangle : reducedRectangles) {
78:                 Point point = rectangle.getCenter();
79:                 Lamp lamp = new Lamp(point.getX(), point.getY(
));
80:                 //Iterator<Integer> tagsOfRectangleIterator =
rectangle.getTagIterator();
81:                 Iterator<Integer> tagsOfRectangleIterator = re
ctangle.getCopyOfTags().iterator();
82:                 while(tagsOfRectangleIterator.hasNext()) {
83:                     lamp.addTag(tagsOfRectangleIterator.ne
xt());
84:                 }
85:                 centersOfReducedRectangles.add(lamp);
86:             }
87:
88:         } catch (OriginalPartialRectanglesFinderException e) {
89:             throw new CandidateSearcherException(e); // chain exce
ptions
90:         } catch (RuntimeExceptionLamps rte) {
91:             throw new CandidateSearcherException(rte);
92:         }
93:
94:         return centersOfReducedRectangles;
95:     }
96:
97: }
98:
99:
100: /**
101:  * Reduces an original set of tagged partial rectangles to a set of no
n-overlapping partial rectangles that contain all tags of
102:  * all original partial rectangles that intersect the final rectangle.
103:  * <p>

```

```

104:      * The algorithm works as follows:
105:      * @param originalRectanglesTagged : The original rectangles
106:      * @return A set of reduced rectangles as described above.
107:      * @throws InterruptedException
108:      */
109:      ArrayList<RectangleWithTag> reduceRectangles(ArrayList<RectangleWithTa
g> originalRectanglesTagged) throws InterruptedException{
110:
111:
112:          ArrayList<RectangleWithTag> reducedRectanglesLastIteration = o
riginalRectanglesTagged;
113:          ArrayList<RectangleWithTag> reducedRectanglesCurrentIteration
= null;
114:
115:          boolean reductionOccured; // the set of rectangles has been fu
rther reduced in the current iteration
116:          do { // as long as set of rectangles can still be reduced
117:
118:              reductionOccured = false;
119:              ArrayList<RectangleWithTag> intersectedRectangleWithTags = new
ArrayList<RectangleWithTag>(); // set of rectangles with tag that can be constructed
from intersections of last iteration rectangles
120:              for (int i = 0; i < reducedRectanglesLastIteration.size(); i++
) {
121:                  RectangleWithTag rectangleWithTagI = reducedRectangles
LastIteration.get(i);
122:                  boolean intersectFoundI = false;
123:                  for (int j = i+1; j< reducedRectanglesLastIteration.si
ze(); j++) {
124:                      RectangleWithTag rectangleWithTagJ = reducedRe
ctanglesLastIteration.get(j);
125:                      Rectangle overlappingRectangle = rectangleWith
TagI.overlap(rectangleWithTagJ);
126:
127:                      if(overlappingRectangle != null) { // intersec
tion detected
128:                          intersectFoundI = true;
129:
130:                          // determine tags of overlap
131:                          HashSet<Integer> tagsOfOverlap= new Ha
shSet<Integer>();
132:                          tagsOfOverlap.addAll(rectangleWithTagI
.getCopyOfTags());
133:                          tagsOfOverlap.addAll(rectangleWithTagJ
.getCopyOfTags());
134:
135:                          // determine all rectangles that also
contain center of overlapping rectangle and add that to the tags of the overlap
136:                          for (int k = j+1; k < reducedRectangle
sLastIteration.size(); k++) {
137:                              RectangleWithTag rectangleWith
TagK = reducedRectanglesLastIteration.get(k);
138:                              if(overlappingRectangle.getCen
ter().isInsideRectangle(rectangleWithTagK.getP1(), rectangleWithTagK.getP3()) ) {
139:                                  tagsOfOverlap.addAll(r
ectangleWithTagK.getCopyOfTags());
140:                              }
141:                          }
142:
143:                          // add to all new rectangles
144:                          RectangleWithTag overlappingRectangleW
ithTag = new RectangleWithTag(overlappingRectangle, tagsOfOverlap);
145:                          intersectedRectangleWithTags.add(overl
appingRectangleWithTag);
146:
147:                      }
148:                      // still need to keep rectangleWithTagI if no intersec
tion with other rectangles is found
149:                      if (!intersectFoundI) {
150:                          intersectedRectangleWithTags.add(rectangleWith
TagI);
151:                      }
152:                  }
153:
154:                  // determine those rectangles whose tags are not contained in
another rectangle's tags
155:                  reducedRectanglesCurrentIteration = new ArrayList<RectangleWit
hTag>();
156:                  boolean[] isMinRectangle = new boolean[intersectedRectangleWit
hTags.size()]; // isMinRectangle[i] = true -> tags of rec_i are not contained in anoth
er rectangle
157:                  for (int i = 0; i<isMinRectangle.length; i++) {
158:                      isMinRectangle[i] = true;;
159:                  }
160:
161:                  for (int i = 0; i<intersectedRectangleWithTags.size(); i++) {
162:                      RectangleWithTag rectangleWithTagI = intersectedRectan
gleWithTags.get(i);
163:
164:
165:                      for (int j = i+1; j < intersectedRectangleWithTags.siz
e(); j++) {
166:                          RectangleWithTag rectangleWithTagJ = intersect
edRectangleWithTags.get(j);
167:                          boolean iSubsetOfJ = rectangleWithTagJ.getCopy
OfTags().containsAll(rectangleWithTagI.getCopyOfTags());
168:                          boolean jSubsetOfI = rectangleWithTagI.getCopy
OfTags().containsAll(rectangleWithTagJ.getCopyOfTags());
169:                          if (iSubsetOfJ && jSubsetOfI) { // equal
170:                              isMinRectangle[i] = false;
171:                          } else if (iSubsetOfJ) {
172:                              isMinRectangle[i] = false;
173:                          } else if (jSubsetOfI) {
174:                              isMinRectangle[j] = false;
175:                          }
176:                      }
177:
178:                  }
179:
180:                  // keep only those rectangles whose tags are not contained in
another rectangle's tags for next iteration
181:                  for (int i = 0; i<isMinRectangle.length; i++) {
182:                      if (isMinRectangle[i]) {
183:                          reducedRectanglesCurrentIteration.add(intersec
tedRectangleWithTags.get(i));
184:                      } else {
185:                          reductionOccured = true; // overlap detected
186:                      }
187:                  }
188:
189:                  // overwrite for next iteration
190:                  reducedRectanglesLastIteration = reducedRectanglesCurrentItera
tion;
191:
192:              } while(reductionOccured);
193:
194:              return reducedRectanglesCurrentIteration;
195:          }

```

`./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/CandidateSearcher.java`

Sat Jun 27 20:58:47 2020

```
196:
197:
198: }
```



```
1: package fernuni.propra.algorithm;
2:
3: /**
4:  * Thrown if something went wrong within the Solve use case (optimized {@link
Lamp} positions are found for
5:  * an {@link IRoom})
6:  * @author alex
7:  *
8:  */
9: public class UserSolveAASEException extends Exception {
10:
11:     public UserSolveAASEException() {
12:     }
13:
14:     public UserSolveAASEException(String message) {
15:         super(message);
16:     }
17:
18:     public UserSolveAASEException(Throwable cause) {
19:         super(cause);
20:     }
21:
22: }
```



```
1: package fernuni.propra.algorithm;
2:
3: /**
4:  * An exception that is thrown if test that checks whether a room is illuminat
ed or not fails due
5:  * to some unexpected error.
6:  * <p>
7:  * @author alex
8:  *
9:  */
10: public class IlluminationTesterException extends Exception {
11:     public IlluminationTesterException() {
12:         // TODO Auto-generated constructor stub
13:     }
14:
15:     public IlluminationTesterException(String message) {
16:         super(message);
17:         // TODO Auto-generated constructor stub
18:     }
19:
20:     public IlluminationTesterException(Throwable cause) {
21:         super(cause);
22:         // TODO Auto-generated constructor stub
23:     }
24:
25:
26: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.Comparator;
4: import java.util.Iterator;
5:
6: import fernuni.propra.internal_data_model.Wall;
7:
8: /**
9:  * A specific container that stores north walls. Those {@link Wall}s can be sp
ecified by
10:  * two {@link Point}s in a horizontal-vertical coordinate system. The {@link W
all}s
11:  * in this container are ordered in ascending order with respect to the vertic
al component
12:  * (y-component) of their {@link Point}s.
13:  * <p>
14:  * The total ordering requested by {@link WallContainerAbstract} is such that
walls
15:  * <p>
16:  * Extended classes and implemented interfaces: {@link WallContainerAbstract}.
17:  * <p>
18:  * @author alex
19:  *
20:  */
21: public class WallContainerNorth extends WallContainerAbstract{
22:
23:     @Override
24:     protected boolean isValidWall(Wall wall, double limit, double low, dou
ble high) {
25:         return wall.overlapsXrange(low, high) && wall.getP1().getY()>
=limit;
26:     }
27:
28:     @Override
29:     protected Comparator<Wall> getComparator() {
30:         return new Comparator<Wall>() { // TODO: dont sort complete li
st -> find correct position and insert there
31:             @Override
32:             public int compare(Wall o1, Wall o2) {
33:                 if (o1.getP1().getY() < o2.getP1().getY()) {
34:                     return -1;
35:                 } else if (o1.getP1().getY()>o2.getP1().getY()
) {
36:                     return 1;
37:                 }
38:                 return 0;
39:             }
40:         };
41:     }
42:
43:     @Override
44:     protected boolean isCorrectWallType(Wall wall) {
45:         return wall.isNorthWall();
46:     }
47: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.ArrayList;
4: import java.util.HashSet;
5: import java.util.Iterator;
6:
7: import fernuni.propra.algorithm.runtime_information.IRuntimeOriginalPartialRectanglesFinder;
8: import fernuni.propra.algorithm.util.RectangleWithTag;
9: import fernuni.propra.internal_data_model.IRoom;
10: import fernuni.propra.internal_data_model.Point;
11: import fernuni.propra.internal_data_model.Wall;
12:
13: public class OriginalPartialRectanglesFinder_old implements IOriginalPartialRectanglesFinder {
14:
15:     private static double findWallTOL = 0.001;
16:     private HashSet<Integer> allTags = new HashSet<Integer>();
17:     private WallContainerEast wallContainerEast = new WallContainerEast();
18:
19:     private WallContainerNorth wallContainerNorth = new WallContainerNorth();
20:
21:     private WallContainerWest wallContainerWest = new WallContainerWest();
22:     private WallContainerSouth wallContainerSouth = new WallContainerSouth();
23:
24:     private ArrayList<RectangleWithTag> originalRectangles = new ArrayList<RectangleWithTag>();
25:
26:     public OriginalPartialRectanglesFinder_old() {
27:         // TODO Auto-generated constructor stub
28:     }
29:
30:     //public static OriginalPartialRectanglesFinder getOriginalPartialRectanglesFinder() {
31:     //    if (singleton == null) {
32:     //        singleton = new OriginalPartialRectanglesFinder();
33:     //    }
34:     //    return singleton;
35:     //}
36:
37:     @Override
38:     public ArrayList<RectangleWithTag> findOriginalPartialRectangles(IRoom room, IRuntimeOriginalPartialRectanglesFinder rti) throws OriginalPartialRectanglesFinderException {
39:         try {
40:             sortWallsToContainers(room);
41:             constructOriginalPartialRectangles();
42:         } catch (WallContainerException | OriginalPartialRectanglesFinderException e) {
43:             throw new OriginalPartialRectanglesFinderException(e);
44:         }
45:
46:         return originalRectangles;
47:     }
48:
49:     @Override
50:     public HashSet<Integer> getAllTags() { // TODO findOriginalPartialRectangles needs to be called first
51:         return allTags;
52:     }
53:
54:     void sortWallsToContainers(IRoom room) throws WallContainerException, OriginalPartialRectanglesFinderException {
55:         Iterator<Wall> wallIterator = room.getWalls();
56:         while (wallIterator.hasNext()) {
57:             Wall nextWall = wallIterator.next();
58:             if (nextWall.isEastWall()) {
59:                 wallContainerEast.add(nextWall);
60:             } else if (nextWall.isNorthWall()) {
61:                 wallContainerNorth.add(nextWall);
62:             } else if (nextWall.isWestWall()) {
63:                 wallContainerWest.add(nextWall);
64:             } else if (nextWall.isSouthWall()) {
65:                 wallContainerSouth.add(nextWall);
66:             } else {
67:                 throw new OriginalPartialRectanglesFinderException("Wall orientation cannot be determined! Wall might not be horizontal or vertical");
68:             }
69:         }
70:
71:         void constructOriginalPartialRectangles() throws WallContainerException {
72:             int rectangleNo = 0;
73:
74:             for (Wall northWall : wallContainerNorth) {
75:                 double yNorth = northWall.getP1().getY();
76:                 double westXLimit = northWall.getP2().getX();
77:                 double eastXLimit = northWall.getP1().getX();
78:
79:                 Wall nextWestWall = wallContainerWest.getNearestWall(yNorth - findWallTOL, yNorth - findWallTOL, westXLimit);
80:
81:                 Wall nextEastWall = wallContainerEast.getNearestWall(yNorth - findWallTOL, yNorth - findWallTOL, eastXLimit);
82:
83:                 double xWest = nextWestWall.getP1().getX();
84:                 double xEast = nextEastWall.getP1().getX();
85:
86:                 Wall nextSouthWall = wallContainerSouth.getNearestWall(xWest+findWallTOL, xEast-findWallTOL, yNorth);
87:                 double ySouth = nextSouthWall.getP1().getY();
88:
89:                 rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
90:             }
91:
92:             for (Wall eastWall : wallContainerEast) {
93:                 double xEast = eastWall.getP1().getX();
94:                 double southYLimit = eastWall.getP1().getY();
95:                 double northYLimit = eastWall.getP2().getY();
96:
97:                 Wall nextSouthWall = wallContainerSouth.getNearestWall(xEast-findWallTOL, xEast - findWallTOL, southYLimit);
98:                 Wall nextNorthWall = wallContainerNorth.getNearestWall(xEast-findWallTOL, xEast - findWallTOL, northYLimit);
99:
100:                 double ySouth = nextSouthWall.getP1().getY();
101:                 double yNorth = nextNorthWall.getP1().getY();
102:
103:                 Wall nextWestWall = wallContainerWest.getNearestWall(ySouth+findWallTOL, yNorth-findWallTOL, xEast);
104:                 double xWest = nextWestWall.getP1().getX();

```

```

107:
108:         rectangleNo = addOriginalPartialRectangle(rectangleNo,
yNorth, xWest, xEast, ySouth);
109:     }
110:
111:
112:         for (Wall westWall: wallContainerWest) {
113:             double xWest = westWall.getP1().getX();
114:             double southYLimit = westWall.getP2().getY();
115:             double northYLimit = westWall.getP1().getY();
116:
117:             Wall nextSouthWall = wallContainerSouth.getNearestWall
(xWest + findWallTOL, xWest + findWallTOL, southYLimit);
118:             Wall nextNorthWall = wallContainerNorth.getNearestWall
(xWest + findWallTOL, xWest + findWallTOL, northYLimit);
119:
120:             double ySouth = nextSouthWall.getP1().getY();
121:             double yNorth = nextNorthWall.getP1().getY();
122:
123:             Wall nextEastWall = wallContainerEast.getNearestWall(y
South+findWallTOL, yNorth-findWallTOL, xWest);
124:             double xEast = nextEastWall.getP1().getX();
125:
126:             rectangleNo = addOriginalPartialRectangle(rectangleNo,
yNorth, xWest, xEast, ySouth);
127:         }
128:
129:         for (Wall southWall: wallContainerSouth) {
130:             double ySouth = southWall.getP1().getY();
131:             double eastXLimit = southWall.getP2().getX();
132:             double westXLimit = southWall.getP1().getX();
133:
134:             Wall nextEastWall = wallContainerEast.getNearestWall(y
South + findWallTOL, ySouth + findWallTOL, eastXLimit);
135:             Wall nextWestWall = wallContainerWest.getNearestWall(y
South + findWallTOL, ySouth + findWallTOL, westXLimit);
136:
137:             double xEast = nextEastWall.getP1().getX();
138:             double xWest = nextWestWall.getP1().getX();
139:
140:             Wall nextNorthWall = wallContainerNorth.getNearestWall
(xWest+findWallTOL, xEast-findWallTOL, ySouth);
141:             double yNorth = nextNorthWall.getP1().getY();
142:
143:             rectangleNo = addOriginalPartialRectangle(rectangleNo,
yNorth, xWest, xEast, ySouth);
144:         }
145:     }
146:
147:     private int addOriginalPartialRectangle(int rectangleNo, double yNorth
, double xWest, double xEast, double ySouth) {
148:         Point southWestCorner = new Point(xWest,ySouth);
149:         Point northEastCorner = new Point(xEast,yNorth);
150:         int tag = rectangleNo++;
151:         RectangleWithTag partialRectangle = new RectangleWithTag(south
WestCorner, northEastCorner, tag);
152:         allTags.add(tag);
153:         originalRectangles.add(partialRectangle);
154:         return rectangleNo;
155:     }
156:
157:     public Iterator<RectangleWithTag> iteratorOriginalRectangles() {
158:         return originalRectangles.iterator();
159:     }

```

```

160:
161:     // TODO for tests
162:     Iterator<Wall> eastIterator() {
163:         return wallContainerEast.iterator();
164:     }
165:
166:     Iterator<Wall> northIterator() {
167:         return wallContainerNorth.iterator();
168:     }
169:
170:     Iterator<Wall> westIterator() {
171:         return wallContainerWest.iterator();
172:     }
173:
174:     Iterator<Wall> southIterator() {
175:         return wallContainerSouth.iterator();
176:     }
177:
178:
179:
180:
181: }

```

./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/IAusleuchtung.java

Sun Jun 07 09:51:03 2020

```
1: package fernuni.propra.algorithm;
2:
3: /**
4:  * Defines an interface that provides the functionality of the program to other
5:  * applications as an API (Application Programming Interface).
6:  * <p>
7:  * Implementing classes: {@link Ausleuchtung}
8:  * <p>
9:  * @author alex
10:  */
11: public interface IAusleuchtung {
12:
13:     public abstract boolean validateSolution(String xmlFile);
14:
15:     public abstract int solve(String xmlFile, int timeLimit);
16:
17: }
```


./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/OriginalPartialRectanglesFinderException.java

Mon

```
1: package fernuni.propra.algorithm;
2:
3: /**
4:  * An exception that is thrown if the computation of original partial rectangles of an {@link IRoom} fails due
5:  * to some unexpected error.
6:  * @author alex
7:  *
8:  */
9: public class OriginalPartialRectanglesFinderException extends Exception {
10:
11:     public OriginalPartialRectanglesFinderException() {
12:
13:     }
14:
15:     public OriginalPartialRectanglesFinderException(String message) {
16:         super(message);
17:
18:     }
19:
20:     public OriginalPartialRectanglesFinderException(Throwable cause) {
21:         super(cause);
22:     }
23:
24:
25: }
```



```

1: package fernuni.propra.algorithm.util;
2:
3: import fernuni.propra.internal_data_model.Point;
4:
5: /**
6:  * A rectangle that does provides functionality to support the algorithm.
7:  * <p>
8:  * The rectangle is specified by providing the corner {@link Point}s,
9:  * where in an horizontal-vertical/x-y coordinate system, the points are
10:  * always arranged as
11:  * <p>
12:  * P1: bottom left
13:  * P2: bottom right
14:  * P3: top right
15:  * P4: top left
16:  * <p>
17:  * The functions {@link equals} and {@link hashCode} are overwritten so that
18:  * {@link Rectangle}s are considered equal if their corner points are each equ
al. This also
19:  * supports usage in a {@link HashSet}.
20:  * <p>
21:  * @author alex
22:  *
23:  */
24: public class Rectangle {
25:     private Point p1,p2,p3,p4; // corner points
26:
27:     public Rectangle(Point p1, Point p3) {
28:         if(!isValidRectangle(p1, p3)) {
29:             throw new IllegalArgumentException("Rectangle not init
ialized correctly");
30:         }
31:         this.p1 = p1;
32:         this.p2 = new Point(p3.getX(),p1.getY());
33:         this.p3 = p3;
34:         this.p4 = new Point(p1.getX(), p3.getY());
35:     }
36:
37:     /**
38:      * @return bottom left {@link Point}
39:      */
40:     public Point getP1() {
41:         return new Point(p1.getX(), p1.getY());
42:     }
43:
44:     /**
45:      * @return bottom right {@link Point}
46:      */
47:     public Point getP2() {
48:         return new Point(p2.getX(), p2.getY());
49:     }
50:
51:     /**
52:      * @return top right {@link Point}
53:      */
54:     public Point getP3() {
55:         return new Point(p3.getX(), p3.getY());
56:     }
57:
58:     /**
59:      * @return top left {@link Point}
60:      */
61:     public Point getP4() {
62:         return new Point(p4.getX(), p4.getY());

```

```

63:     }
64:
65:     /**
66:      * Computes the overlap of two {@link Rectangle} which is also a {@lin
k Rectangle}.
67:      * The overlap is determined by finding the coordinates of P1, i.e. (x
1, y1), and
68:      * P3, i.e. (x3,y3) as
69:      * <p>
70:      * x1 = max(this.p1.x, other.p1.x),
71:      * y1 = max(this.p1.y, other.p1.y),
72:      * x3 = min(this.p3.x, other.p3.y),
73:      * y3 = min(this.p3.y, pther.p3.y)
74:      * <p>
75:      * The resulting rectangle which is defined by P1, P3 is then checked
for orientation
76:      * and for validity.
77:      * <p>
78:      * If the resulting rectangle is valid it is returned. If not then no
overlap exists
79:      * and null is returned.
80:      * <p>
81:      *
82:      * @param other : a {@link Rectangle} that is compared with the callin
g {@link Rectangle}
83:      * @return The {@link Rectangle} that represents the overlap between t
he calling {@link Rectangle} and the parameter
84:      * other. If the rectangles don't overlap, then null is
returned.
85:      */
86:     public Rectangle overlap(Rectangle other) {
87:         Point p1 = new Point(Math.max(this.p1.getX(), other.p1.getX())
,
88:             Math.max(this.p1.getY(), other.p1.getY()));
89:         Point p3 = new Point(Math.min(this.p3.getX(), other.p3.getX())
,
90:             Math.min(this.p3.getY(), other.p3.getY()));
91:         if(isValidRectangle(p1, p3)) {
92:             Rectangle outRectangle = new Rectangle(p1, p3);
93:             if(outRectangle.isCounterClockWise()) {
94:                 return outRectangle;
95:             } else {
96:                 return null;
97:             }
98:         } else {
99:             return null;
100:         }
101:     }
102:
103:     /**
104:      *
105:      * @return the center {@link Point} of the {@link Rectangle}.
106:      */
107:     public Point getCenter() {
108:         double width = p2.getX() - p1.getX();
109:         double height = p3.getY() - p1.getY();
110:         return new Point(p1.getX()+width/2.0, p1.getY()+height/2.0);
111:     }
112:
113:     /**
114:      *
115:      * Checks if {@link Point}s have counter clock wise orientation
116:      * by evaluating of the cross product P2P3 x P2P1
117:      * <p>

```

```

118:      * @return A boolean that shows whether the {@link Rectangle} as count
er clock wise orientation.
119:      */
120:      boolean isCounterClockWise() {
121:          double dx1 = 0.0;
122:          double dx2 = p1.getX()-p2.getX();
123:          double dy1 = p3.getY() - p2.getY();
124:          double dy2 = 0.0;
125:
126:          return dx1 * dy2 - dx2 * dy1 > 0;
127:      }
128:
129:      @Override
130:      public boolean equals(Object o) {
131:          if (o == this) return true;
132:          if (!(o instanceof Rectangle)) {
133:              return false;
134:          }
135:          Rectangle r = (Rectangle) o;
136:          return getP1().isEqual(r.getP1()) && getP2().isEqual(r.getP2())
)
137:              && getP3().isEqual(r.getP3()) && getP4().isEqu
al(r.getP4()) ;
138:      }
139:
140:      @Override
141:      public int hashCode() {
142:          int result = 17;
143:          result = 31 * result + p1.hashCode();
144:          result = 31 * result + p3.hashCode();
145:          return result;
146:      }
147:
148:      /**
149:       * Checks whether the {@link Rectangle} is a valid {@link Rectangle} i
n the sense
150:       * that P1 might be the bottom left point and P3 the top right point.
151:       * @param p1 : P1 of the rectangle
152:       * @param p3 : P3 of the rectangle
153:       * @return A boolean that shows whether the {@link Rectangle} is a val
id rectangle with nonzero volume
154:       */
155:      private static boolean isValidRectangle(Point p1, Point p3) {
156:          boolean isValidRectangle = p1.getX()< p3.getX() && p3.getY()>
p1.getY();
157:          return isValidRectangle;
158:      }
159:
160: }

```

```

1: package fernuni.propra.algorithm.util;
2:
3: import java.util.Collection;
4: import java.util.HashSet;
5:
6: import fernuni.propra.internal_data_model.Point;
7:
8: /**
9:  * A rectangle that can also be tagged, i.e. have a set of integers that repre
sent the tags.
10:  * <p>
11:  * The {@link RectangleWithTag} is designed to represent an original partial r
ectangle of an {@link IRoom}
12:  * instance.
13:  * The tags typically represent the portions of the {@link IRoom} that are ill
uminated if the associated
14:  * {@link RectangleWithTag} is illuminated. This means that the union of the
15:  * tags of all {@link RectangleWithTag}s of an {@link IRoom} should be equal to al
l tags, i.e. all portions of the
16:  * {@link IRoom}. The tags are stored internally as a {@link HashSet}<{@link
Integer}>.
17:  * <p>
18:  * <p>
19:  * Extended classes: {@link Rectangle}
20:  * <p>
21:  * @author alex
22:  *
23:  */
24: public class RectangleWithTag extends Rectangle{
25:
26:     private HashSet<Integer> tags = new HashSet<Integer>(); // the tags of
the Rectangle
27:
28:     /**
29:      * Constructor
30:      * @param p1 : left bottom {@link Point}
31:      * @param p3 : top right {@link Point}
32:      * @param initialTags : a {@link Collection} of initial tags.
33:      */
34:     public RectangleWithTag(Point p1, Point p3, Collection<Integer> initia
lTags) {
35:         super(p1,p3);
36:         if (initialTags != null) {
37:             tags.addAll(initialTags);
38:         }
39:     }
40:
41:     /**
42:      * Constructor
43:      * @param rectangle : A {@link Rectangle} that is used to create the n
ew {@link RectangleWithTag}
44:      * @param initialTags : a {@link Collection} of initial tags.
45:      */
46:     public RectangleWithTag(Rectangle rectangle, Collection<Integer> initi
alTags) {
47:         this(rectangle.getP1(), rectangle.getP3(), initialTags);
48:     }
49:
50:     /**
51:      * Constructor
52:      * @param p1 : left bottom {@link Point}
53:      * @param p3 : top right {@link Point}
54:      * @param initialTags : a single initial tag.
55:      */
56:     public RectangleWithTag(Point p1, Point p3, Integer initialTag) {
57:         super(p1,p3);
58:         if (initialTag != null) {
59:             tags.add(initialTag);
60:         }
61:     }
62:
63:     /**
64:      * Constructor
65:      * @param rectangle : A {@link Rectangle} that is used to create the n
ew {@link RectangleWithTag}
66:      * @param initialTag : a single initial tag.
67:      */
68:     public RectangleWithTag(Rectangle rectangle, Integer initialTag) {
69:         this(rectangle.getP1(),rectangle.getP3(), initialTag);
70:     }
71:
72:     /**
73:      * Tests whether the tags of this {@link RectangleWithTag} contain a c
ertain tag
74:      * @param tag : the integer tag that is to be checked
75:      * @return : a boolean that represents whether this {@link RectangleWi
thTag} contains the tag
76:      */
77:     public boolean containsTag(Integer tag) {
78:         return tags.contains(tag);
79:     }
80:
81:     /**
82:      * Adds a certain tag to the tags of this {@link Rectangle}
83:      * @param tag : the integer tag to be added
84:      */
85:     public void addTag(Integer tag) {
86:         tags.add(tag);
87:     }
88:
89:     /**
90:      * Returns a copy of the {@link RectangleWithTag}s tags.
91:      * @return : A {@link HashSet} with all copies of the tags of this {@l
ink RectangleWithTag}.
92:      */
93:     public HashSet<Integer> getCopyOfTags() {
94:         HashSet<Integer> outTags = new HashSet<Integer>();
95:         for (Integer tag : tags) {
96:             outTags.add(tag.intValue()); // boxing + unboxing TODO
97:         }
98:         return outTags;
99:     }
100:
101:     /**
102:      *
103:      * @return sum of all tags for hashCode
104:      */
105:     private int getSumOfTags() {
106:         int result = 0;
107:         for (Integer tag: tags) {
108:             result = result + tag;
109:         }
110:         return result;
111:     }
112:
113:     @Override
114:     public boolean equals(Object o) {
115:         if (o == this) return true;

```

```
116:            if(!(o instanceof RectangleWithTag)) {
117:                return false;
118:            }
119:            RectangleWithTag r = (RectangleWithTag) o;
120:            return getP1().isEqual(r.getP1()) && getP2().isEqual(r.getP2())
)
121:                && getP3().isEqual(r.getP3()) && getP4().isEqual
al(r.getP4()) && getSumOfTags() == r.getSumOfTags() ;
122:        }
123:
124:        @Override
125:        public int hashCode() {
126:            int result = 17;
127:            result = 31 * result + getP1().hashCode();
128:            result = 31 * result + getP3().hashCode();
129:            result = 31 * result + getSumOfTags();
130:            return result;
131:        }
132:
133: }
```

```

1: package fernuni.propra.algorithm;
2:
3: import java.util.Comparator;
4: import java.util.Iterator;
5:
6: import fernuni.propra.internal_data_model.Wall;
7:
8: /**
9:  * A specific container that stores west walls. Those {@link Wall}s can be spe
cified by
10:  * two {@link Point}s in a horizontal-vertical coordinate system. The {@link W
all}s
11:  * in this container are ordered in descending order with respect to the horiz
ontal component
12:  * (x-component) of their {@link Point}s.
13:  * <p>
14:  * The total ordering requested by {@link WallContainerAbstract} is such that
walls
15:  * <p>
16:  * Extended classes and implemented interfaces: {@link WallContainerAbstract}.
17:  * <p>
18:  * @author alex
19:  *
20:  */
21: public class WallContainerWest extends WallContainerAbstract{
22:
23:     @Override
24:     protected boolean isValidWall(Wall wall, double limit, double low, dou
ble high) {
25:         boolean isValidWall = wall.overlapsYrange(low, high) && wall.
getP1().getX()<=limit;
26:         return isValidWall;
27:     }
28:
29:
30:     @Override
31:     protected Comparator<Wall> getComparator() {
32:         return new Comparator<Wall>() {
33:             @Override
34:             public int compare(Wall o1, Wall o2) {
35:                 if (o1.getP1().getX() > o2.getP1().getX()) {
36:                     return -1;
37:                 } else if (o1.getP1().getX()<o2.getP1().getX()
) {
38:                     return 1;
39:                 }
40:                 return 0;
41:             }
42:         };
43:     }
44:
45:
46:     @Override
47:     protected boolean isCorrectWallType(Wall wall) {
48:         return wall.isWestWall();
49:     }
50:
51: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
4: import fernuni.propra.file_processing.UserReadInputWriteOutputException;
5: import fernuni.propra.internal_data_model.IRoom;
6:
7: /**
8:  * Diese Klasse wird als API (Application Programming Interface) verwendet. D
as
9:  * bedeutet, dass diese Klasse als Bibliothek für andere Applikationen verwen
det
10:  * werden kann.
11:  *
12:  * Bitte achten Sie darauf, am bereits implementierten Rahmen (Klassenname,
13:  * Package, Methodensignaturen) KEINE Veränderungen vorzunehmen.
14:  * Selbstverständlich können und müssen Sie innerhalb der Methoden Ände
rungen
15:  * vornehmen
16:  */
17: public class Ausleuchtung implements IAusleuchtung {
18:
19:     /**
20:      * Ermittelt die eingegebene Lösung auf Korrektheit
21:      * @param xmlFile Dokument mit der Lösung, die validiert werden soll.
22:      * @return true, falls die eingelesene Lösung korrekt ist
23:      */
24:     @Override
25:     public boolean validateSolution(String xmlFile) {
26:         UserReadInputWriteOutputAAS userReadWriteAAS = new UserReadInp
utWriteOutputAAS(xmlFile);
27:
28:         try {
29:             IRoom room = userReadWriteAAS.readInput();
30:             UserValidateAAS userValidateAAS = new UserValidateAAS(
);
31:
32:             boolean isIlluminated = userValidateAAS.validate(room)
;
33:
34:             return isIlluminated;
35:         } catch (UserReadInputWriteOutputException e) {
36:             // TODO Fehlermeldung auf Konsole ausgeben?
37:             //e.printStackTrace();
38:             return false;
39:         } catch (UserValidateAASException e) {
40:             // TODO Fehlermeldung auf Konsole ausgeben?
41:             //e.printStackTrace();
42:             return false;
43:         }
44:
45:     }
46:
47:     /**
48:      * Ermittelt eine Lösung zu den eingegebenen Daten
49:      * @param xmlFile Dokument, das die zu lösende Problem Instanz enthält
50:      */
51:     @Override
52:     public int solve(String xmlFile, int timeLimit) {
53:         UserReadInputWriteOutputAAS userReadWriteAAS = new UserReadInp
utWriteOutputAAS(xmlFile);
54:
55:         try {
56:             IRoom room = userReadWriteAAS.readInput();
57:
58:             UserSolveAAS userSolveAAS = new UserSolveAAS();
59:             int numberOfLampsBestSolution = userSolveAAS.solve(roo
m, timeLimit);
60:
61:             return numberOfLampsBestSolution;
62:         } catch (UserReadInputWriteOutputException e) {
63:             // TODO Fehlermeldung auf Konsole ausgeben?
64:             //e.printStackTrace();
65:             return 0;
66:         } catch (UserSolveAASException e) {
67:             // TODO Fehlermeldung auf Konsole ausgeben?
68:             //e.printStackTrace();
69:             return 0;
70:         }
71:
72:     }

```



```
1: package fernuni.propra.algorithm;
2:
3: /**
4:  * Thrown if something went wrong with the validation of an {@link IRoom} instance.
5:  * @author alex
6:  *
7:  */
8: public class UserValidateAASEException extends Exception {
9:
10:     public UserValidateAASEException() {
11:     }
12:
13:     public UserValidateAASEException(String message) {
14:         super(message);
15:     }
16:
17:     public UserValidateAASEException(Throwable cause) {
18:         super(cause);
19:     }
20:
21: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.ArrayList;
4: import java.util.HashSet;
5: import java.util.Iterator;
6: import java.util.LinkedList;
7: import java.util.List;
8:
9: import fernuni.propra.algorithm.runtime_information.IRuntimePositionOptimizer;
10: import fernuni.propra.internal_data_model.IRoom;
11: import fernuni.propra.internal_data_model.Lamp;
12:
13: public class PositionOptimizer_old implements IPositionOptimizer{
14:     private static List<Lamp> currentBestSolution;
15:     private static int numberIlluminatedLampsBestSolution;
16:     private static IIlluminationTester illuminationTester = AbstractAlgori
    thmFactory.getAlgorithmFactory().createIlluminationTester();
17:
18:     public PositionOptimizer_old() {
19:     }
20:
21:     @Override
22:     public List<Lamp> optimizePositions( List<Lamp> taggedCandidates, IRun
    timePositionOptimizer runTimeInformation) throws InterruptedException{
23:
24:         // all lamps are on -> illuminated
25:         currentBestSolution = taggedCandidates;
26:         numberIlluminatedLampsBestSolution = taggedCandidates.size();
27:
28:         HashSet<Integer> allTags = new HashSet<Integer>();
29:         for (Lamp lamp : taggedCandidates) {
30:             lamp.turnOff(); // make sure all lamps are turned off
31:             Iterator<Integer> tagIterator = lamp.iteratorTag();
32:             while(tagIterator.hasNext()) {
33:                 allTags.add(tagIterator.next());
34:             }
35:         }
36:
37:         ArrayList<Lamp> lamps = deepCopyLamps(taggedCandidates);
38:
39:         //HashSet<Integer> illuminated = new HashSet<Integer>();
40:
41:         searchSolution(lamps,0, allTags, 0,runTimeInformation);
42:
43:         return currentBestSolution;
44:
45:     }
46:
47:     private void searchSolution(ArrayList<Lamp> lamps, int idx,
48:         HashSet<Integer> allTags, int numberLampsOn, IRuntimeP
    ositionOptimizer runTimeInformation) throws InterruptedException {
49:
50:         if (Thread.currentThread().isInterrupted()) {
51:             throw new InterruptedException();
52:         }
53:
54:         if(illuminationTester.testIfRoomIsIlluminated(lamps.iterator()
    , allTags, runTimeInformation)) { // valid solution found
55:             if (numberLampsOn<=numberIlluminatedLampsBestSolution)
56:
57:
58:
59:         }
60:
61:         System.out.println("Solution found with " + nu
    mberLampsOn + " lamps turned on.");
62:         currentBestSolution = deepCopyLamps(lamps);
63:         numberIlluminatedLampsBestSolution = numberLam
    psOn;
64:
65:         } else { // not a valid solution
66:             if (idx < lamps.size()) {
67:                 if(numberLampsOn<numberIlluminatedLampsBestSol
    ution) {
68:                     Lamp lamp = lamps.get(idx);
69:                     lamp.turnOn();
70:                     searchSolution(deepCopyLamps(lamps), i
    dx+1, allTags, numberLampsOn+1, runTimeInformation);
71:                     lamp.turnOff();
72:                     searchSolution(deepCopyLamps(lamps), i
    dx+1, allTags, numberLampsOn, runTimeInformation);
73:                 }
74:             }
75:         }
76:
77:     }
78:
79: }
80:
81: private static ArrayList<Lamp> deepCopyLamps(List<Lamp> lamps) {
82:     ArrayList<Lamp> outLamps = new ArrayList<Lamp>(lamps.size());
83:     Iterator<Lamp> lampsIterator = lamps.iterator();
84:     while(lampsIterator.hasNext()) {
85:         Lamp lamp = lampsIterator.next();
86:         outLamps.add(lamp.deepCopy());
87:     }
88:     return outLamps;
89: }
90:
91: private static HashSet<Integer> deepCopyHashSet(HashSet<Integer> hashS
    et) {
92:     HashSet<Integer> outHashSet = new HashSet<Integer>();
93:     for (Integer integer : hashSet) {
94:         Integer outInteger = (int) integer;
95:         outHashSet.add(outInteger);
96:     }
97:     return outHashSet;
98: }
99:
100: @Override
101: public List<Lamp> getCurrentBestSolution() {
102:     if (currentBestSolution == null) {
103:         return null;
104:     }
105:     List<Lamp> outLamps = new LinkedList<Lamp>();
106:     Iterator<Lamp> lampIterator = currentBestSolution.iterator();
107:     while(lampIterator.hasNext()) {
108:         outLamps.add(lampIterator.next().deepCopy());
109:     }
110:     return outLamps;
111: }
112:
113: }
114:
115: @Override
116: public int getNumberOfOnLampsBestSolution() {
117:     return numberIlluminatedLampsBestSolution;

```

./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/PositionOptimizer_old.java

Mon Jun 15 20:02:12 20

```
118:         }  
119:  
120:  
121:  
122:  
123: }
```

```
1: package fernuni.propra.algorithm;
2:
3: /**
4:  * Thrown if something went wrong within the validation algorithm, i.e. the check
5:  * whether an {@link IRoom} is illuminated by its {@link Lamp}s.
6:  * <p>
7:  * @author alex
8:  *
9:  */
10: public class ValidateKException extends Exception {
11:
12:     public ValidateKException() {
13:     }
14:
15:     public ValidateKException(String message) {
16:         super(message);
17:     }
18:
19:     public ValidateKException(Throwable cause) {
20:         super(cause);
21:     }
22:
23: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.Comparator;
4: import java.util.Iterator;
5:
6: import fernuni.propra.internal_data_model.Wall;
7:
8: /**
9:  * A specific container that stores south walls. Those {@link Wall}s can be sp
ecified by
10:  * two {@link Point}s in a horizontal-vertical coordinate system. The {@link W
all}s
11:  * in this container are ordered in descending order with respect to the verti
cal component
12:  * (y-component) of their {@link Point}s.
13:  * <p>
14:  * The total ordering requested by {@link WallContainerAbstract} is such that
walls
15:  * <p>
16:  * Extended classes and implemented interfaces: {@link WallContainerAbstract}.
17:  * <p>
18:  * @author alex
19:  *
20:  */
21: public class WallContainerSouth extends WallContainerAbstract {
22:
23:     @Override
24:     protected boolean isValidWall(Wall wall, double limit, double low, dou
ble high) {
25:         return wall.overlapsXrange(low, high) && wall.getP1().getY() <
=limit;
26:     }
27:
28:     @Override
29:     protected Comparator<Wall> getComparator() {
30:         return new Comparator<Wall>() { // TODO: dont sort complete li
st -> find correct position and insert there
31:             @Override
32:             public int compare(Wall o1, Wall o2) {
33:                 if (o1.getP1().getY() > o2.getP1().getY()) {
34:                     return -1;
35:                 } else if (o1.getP1().getY() < o2.getP1().getY()
) {
36:                     return 1;
37:                 }
38:                 return 0;
39:             }
40:         };
41:     }
42:
43:     @Override
44:     protected boolean isCorrectWallType(Wall wall) {
45:         return wall.isSouthWall();
46:     }
47: }

```



```

1: package fernuni.propra.algorithm;
2:
3:
4: import java.awt.event.ActionEvent;
5: import java.awt.event.ActionListener;
6:
7: import javax.swing.Timer;
8:
9: import fernuni.propra.algorithm.runtime_information.IRuntimeInformation;
10: import fernuni.propra.algorithm.runtime_information.IRuntimeReader;
11: import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
12: import fernuni.propra.algorithm.runtime_information.RuntimeInformation;
13: import fernuni.propra.internal_data_model.IRoom;
14:
15: /**
16:  * Use case that provides access to the solution algorithm, which allows to co
17:  * mpute the optimal {@link Lamp}
18:  * positions for a given {@link IRoom} instance and a given time limit which h
19:  * as to be specified as an integer number
20:  * representing the seconds a solution is allowed to take.
21:  *
22:  * @author alex
23:  */
24: public class UserSolveAAS {
25:     IRuntimeInformation runTimeInformation = new RuntimeInformation();
26:
27:     /**
28:      * The interface to the solution algorithm. A separate thread is start
29:      * ed to handle the algorithm that is
30:      * interrupted after the time limit has been reached.
31:      *
32:      * <p>
33:      * The solving is delegated to an instance of {@link SolveK} that con
34:      * trols the execution of the algorithm
35:      * and makes the results available.
36:      *
37:      * <p>
38:      * @param room : {@link IRoom} instance for which the optimal {@link L
39:      * amp} positions have to be found.
40:      * @param time : The time limit in seconds as an integer number. Negat
41:      * ive numbers are treated as infinite
42:      * time limits.
43:      * @return The number of {@link Lamp}s that are turned on in the best
44:      * solution.
45:      * @throws UserSolveAASException
46:      */
47:     public int solve(IRoom room, int time) throws UserSolveAASException {
48:         SolveK solveControl = new SolveK(room, runTimeInformation);
49:         try {
50:             runTimeInformation.startTime();
51:
52:             if (time > 0) { // if time argument is smaller than ze
53: ro time limit is ignored
54:                 Timer timer = new Timer(time * 1000, new Actio
55: nListener() {
56:
57:                 @Override
58:                 public void actionPerformed(ActionEvent
59: t e) {
60:
61:                     solveControl.interrupt();
62:
63:                 }
64:             });
65:             timer.start();
66:
67:         } catch (InterruptedException ie) {
68:             throw new UserSolveAASException(ie);
69:         } catch (RuntimeExceptionLamps rte) {
70:             throw new UserSolveAASException(rte);
71:         } finally {
72:             solveControl.interrupt(); // stop solveControl thread
73:         }
74:     }
75:
76:     /**
77:      * Provides access to runtime information.
78:      * @return A data structure of type {@link IRuntimeReader} that allows
79:      * to obtain run time information.
80:      */
81:     public IRuntimeReader getRuntimeInformation() {
82:         return runTimeInformation;
83:     }
84:
85: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import fernuni.propra.algorithm.runtime_information.IRuntimeInformation;
4: import fernuni.propra.algorithm.runtime_information.IRuntimeReader;
5: import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
6: import fernuni.propra.algorithm.runtime_information.RuntimeInformation;
7: import fernuni.propra.internal_data_model.IRoom;
8: /**
9:  * Use case that provides access to the validation algorithm for an {@link IRoom} instance.
10:  * <p>
11:  * The test for illumination is delegated to an instance of {@link ValidateK} that controls the
12:  * execution of the algorithm and returns the result.
13:  * <p>
14:  *
15:  * @author alex
16:  *
17:  */
18: public class UserValidateAAS {
19:     private ValidateK validateK = new ValidateK();
20:     private String resultString; // the result to be displayed.
21:     IRuntimeInformation runtimeInfo = new RuntimeInformation();
22:
23:     /**
24:      * Provides the user with access to the validation algorithm
25:      * @param room
26:      * @return
27:      * @throws UserValidateAASException
28:      */
29:     public boolean validate(IRoom room) throws UserValidateAASException{
30:         try {
31:             runtimeInfo.startTime();
32:             boolean isIlluminated = validateK.validate(room, runtimeInfo);
33:             runtimeInfo.stopTime();
34:             resultString = computeResultString(room, isIlluminated);
35:
36:             return isIlluminated;
37:         } catch (ValidateKException e) {
38:             throw new UserValidateAASException(e);
39:         } catch (RuntimeExceptionLamps e) {
40:             throw new UserValidateAASException(e);
41:         }
42:
43:     /**
44:      * Computes a result string that can be displayed to the user.
45:      * @param room : {@link IRoom} instance that has to be checked.
46:      * @param isIlluminated : a boolean that represents whether the room is illuminated or not
47:      * @return
48:      */
49:     private static String computeResultString(IRoom room, Boolean isIlluminated) {
50:         String lineSeparator = System.getProperty("line.separator");
51:         StringBuilder sb = new StringBuilder("The room ");
52:         sb.append(room.getID());
53:         String illuminatedOrNot = isIlluminated ? " is illuminated. " : " is NOT illuminated. ";
54:         sb.append(illuminatedOrNot);
55:         sb.append(lineSeparator);
56:         sb.append(room.printLampPositions());
57:         String outString = sb.toString();
58:
59:         return outString;
60:     }
61:
62:     /**
63:      * Can be used to get the result of the algorithm once it is available due to a prior call to
64:      * validate of the same instance.
65:      * @return A string that represents the result of the test.
66:      * @throws UserValidateAASException : e.g. if validate has not been called prior to this.
67:      */
68:     public String getResultString() throws UserValidateAASException{
69:         if (resultString != null) {
70:             return resultString;
71:         } else {
72:             throw new UserValidateAASException("No result available. Call validate first.");
73:         }
74:     }
75:
76:     /**
77:      * Get runtime information.
78:      * @return a data structure of type {@link IRuntimeReader} that can be used to obtain runtime information.
79:      */
80:     public IRuntimeReader getRuntimeInformation() {
81:         return runtimeInfo;
82:     }
83:
84: }

```


./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/CandidateSearcherException.java

Sat Jun 06 11:43:

```
1: package fernuni.propra.algorithm;
2:
3: /**
4:  * An exception that is thrown if the search of potential candidates for lamp
positions fails due
5:  * to some unexpected error.
6:  * @author alex
7:  *
8:  */
9: public class CandidateSearcherException extends Exception {
10:
11:     public CandidateSearcherException() {
12:
13:     }
14:
15:     public CandidateSearcherException(String message) {
16:         super(message);
17:
18:     }
19:
20:     public CandidateSearcherException(Throwable cause) {
21:         super(cause);
22:
23:     }
24:
25: }
```



```

1: package fernuni.propra.algorithm.runtime_information;
2: /**
3:  * An interface that extends the extended interfaces so that implementing clas
ses can declare
4:  * that they are capable of storing overall runtime information.
5:  * <p>
6:  * Implementing classes: {@link RuntimeInformation}
7:  * <p>
8:  * Extended interfaces: {@link IRuntimeCandidateSearcher}, {@link IRuntimePosi
tionOptimizer},
9:  *                               {@link IRuntimeIlluminationTester}, {@l
ink IRuntimeReader}
10:  * <p>
11:  * @author alex
12:  *
13:  */
14: public interface IRuntimeInformation extends IRuntimeCandidateSearcher,
15:         IRuntimePositionOptimizer, IRuntimeIlluminationTester, IRuntimeReader{
16:     /**
17:      * Start the clock for the overall computation.
18:      * @throws RuntimeExceptionLamps : if not handled correctly
19:      */
20:     void startTime() throws RuntimeExceptionLamps;
21:
22:     /**
23:      * Stop the clock for the overall computation.
24:      * @throws RuntimeExceptionLamps
25:      */
26:     void stopTime() throws RuntimeExceptionLamps;
27: }

```


./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/runtime_information/IRuntimeCandidateSearcher.java

```
1: package fernuni.propra.algorithm.runtime_information;
2:
3: /**
4:  * An interface that extends the extended interfaces so that implementing classes can declare
5:  * that they are capable of storing runtime information for the part of the algorithm that deals
6:  * with finding candidates for {@link Lamp} positions.
7:  * <p>
8:  * Implementing classes: {@link RuntimeInformation}
9:  * <p>
10:  * Extended interfaces: {@link IRuntimeOriginalPartialRectanglesFinder}
11:  * <p>
12:  * @author alex
13:  *
14:  */
15: public interface IRuntimeCandidateSearcher extends IRuntimeOriginalPartialRectanglesFinder{
16:     /**
17:      * Start the clock for the part of the algorithm that deals
18:      * with finding candidates for {@link Lamp} positions.
19:      * @throws RuntimeExceptionLamps : if not handled correctly
20:      */
21:     void startTimeCandidateSearch() throws RuntimeExceptionLamps;
22:
23:     /**
24:      * Stop the clock for the part of the algorithm that deals
25:      * with finding candidates for {@link Lamp} positions.
26:      * @throws RuntimeExceptionLamps
27:      */
28:     void stopTimeCandidateSearch() throws RuntimeExceptionLamps;
29: }
```


./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/runtime_information/IRuntimeOriginalPartialRectanglesFinder

```
1: package fernuni.propra.algorithm.runtime_information;
2:
3: /**
4:  * An interface that extends the extended interfaces so that implementing classes
5:  * can declare
6:  * that they are capable of storing runtime information for the part of the algorithm
7:  * that deals
8:  * with finding the original partial rectangles of the {@link IRoom}.
9:  * <p>
10:  * Implementing classes: {@link RuntimeInformation}
11:  * <p>
12:  * @author alex
13:  */
14: public interface IRuntimeOriginalPartialRectanglesFinder {
15:     /**
16:      * Start the clock for the part of the algorithm that deals
17:      * with finding the original partial rectangles of the {@link IRoom}..
18:      * @throws RuntimeExceptionLamps : if not handled correctly
19:      */
20:     void startTimeOriginalPartialRectanglesFind() throws RuntimeExceptionLamps;
21:
22:     /**
23:      * Stop the clock for the part of the algorithm that deals
24:      * with finding the original partial rectangles of the {@link IRoom}.
25:      * @throws RuntimeExceptionLamps : if not handled correctly
26:      */
27:     void stopTimeOriginalPartialRectanglesFind() throws RuntimeExceptionLamps;
28: }
```


./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/runtime_information/IRuntimeReader.java

Fri May 0

```
1: package fernuni.propra.algorithm.runtime_information;
2:
3: public interface IRuntimeReader {
4:     long getElapsedTimeCandidateSearch() throws RuntimeExceptionLamps;
5:     long getElapsedTimeOptimizePositions() throws RuntimeExceptionLamps;
6:     long getElapsedTimeOriginalPartialRectanglesFind() throws RuntimeExcep
tionLamps;
7:     long getElapsedTime() throws RuntimeExceptionLamps;
8:     long getElapsedTimeIlluminationTest() throws RuntimeExceptionLamps;
9: }
```



```

1: package fernuni.propra.algorithm.runtime_information;
2:
3: import java.util.concurrent.TimeUnit;
4:
5: public class RuntimeInformation implements IRuntimeInformation, IRuntimeReader
{
6:     private volatile long startTime = -1;
7:     private volatile long stopTime = -1;
8:
9:     private volatile long candidateSearchStartTime = -1;
10:    private volatile long candidateSearchStopTime = -1;
11:
12:    private volatile long originalPartialRectanglesFindStartTime = -1;
13:    private volatile long originalPartialRectanglesFindStopTime = -1;
14:
15:    private volatile long optimizePositionsStartTime = -1;
16:    private volatile long optimizePositionsStopTime = -1;
17:
18:    private volatile long illuminationTestStartTime = -1;
19:    private volatile long illuminationTestStopTime = -1;
20:
21:
22:    @Override
23:    public void startTimeCandidateSearch() throws RuntimeExceptionLamps {
24:        if (candidateSearchStartTime != -1 && candidateSearchStopTime
!= -1) {
25:            throw new RuntimeExceptionLamps();
26:        }
27:        candidateSearchStartTime = System.nanoTime();
28:    }
29:
30:
31:    @Override
32:    public void stopTimeCandidateSearch() throws RuntimeExceptionLamps {
33:        if (candidateSearchStartTime == -1 || candidateSearchStopTime
!= -1) {
34:            throw new RuntimeExceptionLamps();
35:        }
36:        candidateSearchStopTime = System.nanoTime();
37:    }
38:
39:
40:    @Override
41:    public long getElapsedTimeCandidateSearch() throws RuntimeExceptionLam
ps {
42:        if (candidateSearchStartTime == -1 && candidateSearchStopTime
== -1) {
43:            throw new RuntimeExceptionLamps();
44:        }
45:        return candidateSearchStopTime-candidateSearchStartTime;
46:    }
47:
48:
49:
50:    @Override
51:    public void startTimeOriginalPartialRectanglesFind() throws RuntimeExc
eptionLamps {
52:        if (originalPartialRectanglesFindStartTime != -1 && originalPa
rtialRectanglesFindStopTime != -1) {
53:            throw new RuntimeExceptionLamps();
54:        }
55:        originalPartialRectanglesFindStartTime = System.nanoTime();
56:    }
57:
58:
59:    @Override
60:    public void stopTimeOriginalPartialRectanglesFind() throws RuntimeExce
ptionLamps {
61:        if (originalPartialRectanglesFindStartTime == -1 || originalPa
rtialRectanglesFindStopTime != -1) {
62:            throw new RuntimeExceptionLamps();
63:        }
64:        originalPartialRectanglesFindStopTime = System.nanoTime();
65:    }
66:
67:
68:    @Override
69:    public long getElapsedTimeOriginalPartialRectanglesFind() throws Runti
meExceptionLamps {
70:        if (originalPartialRectanglesFindStartTime == -1 && originalPa
rtialRectanglesFindStopTime == -1) {
71:            throw new RuntimeExceptionLamps();
72:        }
73:        return originalPartialRectanglesFindStopTime-originalPartialRe
ctanglesFindStartTime;
74:    }
75:
76:
77:    @Override
78:    public void startTimeOptimizePositions() throws RuntimeExceptionLamps
{
79:        if (optimizePositionsStartTime != -1 && optimizePositionsStopT
ime != -1) {
80:            throw new RuntimeExceptionLamps();
81:        }
82:        optimizePositionsStartTime = System.nanoTime();
83:    }
84:
85:
86:    @Override
87:    public void stopTimeOptimizePositions() throws RuntimeExceptionLamps {
88:        if (optimizePositionsStartTime == -1 || optimizePositionsStopT
ime != -1) {
89:            throw new RuntimeExceptionLamps();
90:        }
91:        optimizePositionsStopTime = System.nanoTime();
92:    }
93:
94:
95:    @Override
96:    public long getElapsedTimeOptimizePositions() throws RuntimeExceptionL
amps {
97:        if (optimizePositionsStartTime == -1 && optimizePositionsStopT
ime == -1) {
98:            throw new RuntimeExceptionLamps();
99:        }
100:        return optimizePositionsStopTime-optimizePositionsStartTime;
101:    }
102:
103:
104:    @Override
105:    public void resetTimeOptimizePositions() {
106:        optimizePositionsStartTime = -1;
107:        optimizePositionsStopTime = -1;
108:    }
109:
110:    @Override
111:    public void startTimeIlluminationTest() throws RuntimeExceptionLamps {

```

```

112:         if (illuminationTestStartTime != -1 && illuminationTestStopTim
e != -1) {
113:             throw new RuntimeExceptionLamps();
114:         }
115:         illuminationTestStartTime = System.nanoTime();
116:
117:     }
118:
119:     @Override
120:     public void stopTimeIlluminationTest() throws RuntimeExceptionLamps {
121:         if (illuminationTestStartTime == -1 || illuminationTestStopTim
e != -1) {
122:             throw new RuntimeExceptionLamps();
123:         }
124:         illuminationTestStopTime = System.nanoTime();
125:
126:     }
127:
128:     @Override
129:     public long getElapsedTimeIlluminationTest() throws RuntimeExceptionLa
mps {
130:         if (illuminationTestStartTime == -1 && illuminationTestStopTim
e == -1) {
131:             throw new RuntimeExceptionLamps();
132:         }
133:         return illuminationTestStopTime-illuminationTestStopTime;
134:     }
135:
136:
137:     @Override
138:     public void startTime() throws RuntimeExceptionLamps {
139:         if (startTime != -1 && stopTime != -1) {
140:             throw new RuntimeExceptionLamps();
141:         }
142:         startTime = System.nanoTime();
143:
144:     }
145:
146:     @Override
147:     public void stopTime() throws RuntimeExceptionLamps {
148:         if (startTime == -1 || stopTime != -1) {
149:             throw new RuntimeExceptionLamps();
150:         }
151:         stopTime = System.nanoTime();
152:
153:     }
154:
155:     @Override
156:     public long getElapsedTime() throws RuntimeExceptionLamps {
157:         if (startTime == -1 && stopTime == -1) {
158:             throw new RuntimeExceptionLamps();
159:         }
160:         return stopTime-startTime;
161:     }
162:
163:
164:
165:     @Override
166:     public String toString() {
167:         String lineSeparator = System.getProperty("line.separator");
168:         StringBuilder sb = new StringBuilder("Runtime Information");
169:         sb.append(lineSeparator);
170:         sb.append("Total runtime: ");
171:         try {
172:             sb.append((double) Math.round((double) getElapsedTime(
) / 1_000_000_000 * 100)/100);
173:             sb.append(" s.");
174:         } catch (RuntimeExceptionLamps e) {
175:             sb.append("not available");
176:         }
177:         sb.append(lineSeparator);
178:         sb.append("thereof ");
179:         sb.append(lineSeparator);
180:
181:         sb.append("searching for lamp position candidates: ");
182:         try {
183:             sb.append((double) Math.round((double) getElapsedTimeC
andidateSearch() / 1_000_000_000 * 100)/100);
184:             sb.append(" s.");
185:         } catch (RuntimeExceptionLamps e) {
186:             sb.append("not available");
187:         }
188:         sb.append(lineSeparator);
189:
190:
191:         sb.append("optimizing lamp positions: ");
192:         try {
193:             sb.append((double) Math.round((double) getElapsedTimeO
ptimizePositions() / 1_000_000_000 * 100)/100);
194:             sb.append(" s.");
195:         } catch (RuntimeExceptionLamps e) {
196:             sb.append("not available");
197:         }
198:         sb.append(lineSeparator);
199:
200:         sb.append("testing if room is illuminated: ");
201:         try {
202:             sb.append((double) Math.round((double) getElapsedTimeI
lluminationTest() / 1_000_000_000 * 100)/100);
203:             sb.append(" s.");
204:         } catch (RuntimeExceptionLamps e) {
205:             sb.append("not available");
206:         }
207:         sb.append(lineSeparator);
208:
209:         sb.append("constructing original partial rectangles: ");
210:         try {
211:             sb.append((double) Math.round((double) getElapsedTimeO
riginalPartialRectanglesFind() / 1_000_000_000 * 100)/100);
212:             sb.append(" s.");
213:         } catch (RuntimeExceptionLamps e) {
214:             sb.append("not available");
215:         }
216:
217:         String outString = sb.toString();
218:         return outString;
219:     }
220:
221: }

```

./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/runtime_information/IRuntimeIlluminationTester.java

```
1: package fernuni.propra.algorithm.runtime_information;
2: /**
3:  * An interface that extends the extended interfaces so that implementing classes
4:  * can declare
5:  * that they are capable of storing runtime information for the part of the algorithm
6:  * that deals
7:  * with checking whether an {@link IRoom} is illuminated.
8:  * <p>
9:  * Implementing classes: {@link RuntimeInformation}
10:  * <p>
11:  * Extended interfaces: {@link IRuntimeOriginalPartialRectanglesFinder}
12:  * <p>
13:  * @author alex
14:  */
15: public interface IRuntimeIlluminationTester extends IRuntimeOriginalPartialRectanglesFinder{
16:     /**
17:      * Start the clock for the part of the algorithm that deals
18:      * with checking whether an {@link IRoom} is illuminated.
19:      * @throws RuntimeExceptionLamps : if not handled correctly
20:      */
21:     void startTimeIlluminationTest() throws RuntimeExceptionLamps;
22:     /**
23:      * Stop the clock for the part of the algorithm that deals
24:      * with checking whether an {@link IRoom} is illuminated.
25:      * @throws RuntimeExceptionLamps
26:      */
27:     void stopTimeIlluminationTest() throws RuntimeExceptionLamps;
28: }
```


./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/runtime_information/RuntimeExceptionLamps.java

```
1: package fernuni.propra.algorithm.runtime_information;
2:
3: /**
4:  * Thrown if a data structure for the storing of runtime information, e.g. {@link
5:  * RuntimeInformation}, is used incorrectly.
6:  * E.g. if methods are called in a wrong order (time is stopped before a clock
7:  * is started.
8:  * @author alex
9:  */
10: public class RuntimeExceptionLamps extends Exception {
11:     public RuntimeExceptionLamps() {
12:         // TODO Auto-generated constructor stub
13:     }
14:
15:     public RuntimeExceptionLamps(String message) {
16:         super(message);
17:         // TODO Auto-generated constructor stub
18:     }
19:
20:     public RuntimeExceptionLamps(Throwable cause) {
21:         super(cause);
22:         // TODO Auto-generated constructor stub
23:     }
24:
25:
26: }
```


./ProPra2020_workspace/Algorithm_Component/src/fernuni/propra/algorithm/runtime_information/IRuntimePositionOptimizer.java

```
1: package fernuni.propra.algorithm.runtime_information;
2:
3: public interface IRuntimePositionOptimizer extends IRuntimeIlluminationTester
{
4:     void startTimeOptimizePositions() throws RuntimeExceptionLamps;
5:     void stopTimeOptimizePositions() throws RuntimeExceptionLamps;
6:     void resetTimeOptimizePositions();
7:
8: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.Iterator;
4:
5: import fernuni.propra.algorithm.runtime_information.IRuntimeIlluminationTester
;
6: import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
7: import fernuni.propra.internal_data_model.IRoom;
8: import fernuni.propra.internal_data_model.Lamp;
9:
10: /**
11:  * A controll class that controls the program flow for the validation of an {@
link IRoom} instance,
12:  * i.e. for the test whether that {@link IRoom} is illuminated by its associat
ed {@link Lamp}s or not.
13:  * <p>
14:  * Delegates the algorithm to an instance of {@link IilluminationTester}, whic
h is obtained from a call to
15:  * the {@link AbstractAlgorithmFactory}.
16:  * <p>
17:  * Furthermore a data structure is given to {@link ValidateK} in order to stor
e runtime information to that
18:  * data structure.
19:  * <p>
20:  * @author alex
21:  *
22:  */
23: public class ValidateK {
24:     /**
25:      * Checks whether an {@link IRoom} instance is illuminated or not by d
elegating to {@link IilluminationTester}.
26:      * @param room : the {@link IRoom} to be checked
27:      * @param runtimeInfo : {@link IRoom} the data structure to which {@li
nk ValidateK} will write runtime information.
28:      * @return
29:      * @throws ValidateKException
30:      */
31:     boolean validate(IRoom room, IRuntimeIlluminationTester runtimeInfo) t
hrows ValidateKException {
32:         try {
33:             // turn all lamps on
34:             Iterator<Lamp> lampIterator = room.getLamps();
35:             while(lampIterator.hasNext()) {
36:                 lampIterator.next().turnOn();
37:             }
38:             runtimeInfo.startTimeIlluminationTest();
39:             boolean isIlluminated = AbstractAlgorithmFactory.getAl
gorithmFactory().createIlluminatiationTester().testIfRoomIsIlluminated(room, runtimeInfo
);
40:             runtimeInfo.stopTimeIlluminationTest();
41:             return isIlluminated;
42:         } catch (IlluminationTesterException e) {
43:             throw new ValidateKException(e);
44:         } catch( RuntimeExceptionLamps rte) {
45:             throw new ValidateKException(rte);
46:         }
47:     }
48:
49:
50:
51: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.ArrayList;
4: import java.util.HashSet;
5: import java.util.Iterator;
6:
7: import fernuni.propra.algorithm.runtime_information.IRuntimeOriginalPartialRec
tanglesFinder;
8: import fernuni.propra.algorithm.util.Rectangle;
9: import fernuni.propra.algorithm.util.RectangleWithTag;
10: import fernuni.propra.internal_data_model.IRoom;
11: import fernuni.propra.internal_data_model.LineSegment;
12: import fernuni.propra.internal_data_model.Point;
13: import fernuni.propra.internal_data_model.Wall;
14: /**
15:  * Provides an algorithm to computes all original partial rectangles of an {@li
nk IRoom} instance and to tag
16:  * these rectangles with tags that correspond to a portion of the {@link IRoom
} that is illuminated if the associated
17:  * partial rectangle is illuminated.
18:  * <p>
19:  * The tags/portions correspond to {@link Wall}s of the {@link IRoom} that are
20:  * illuminated if the associated rectangle is illuminated.
21:  * <p>
22:  * Implemented interfaces : {@link IOriginalPartialRectanglesFinder}
23:  * <p>
24:  *
25:  * @author alex
26:  *
27:  */
28: public class OriginalPartialRectanglesFinder implements IOriginalPartialRectan
glesFinder{
29:
30:     private static double findWallTOL = 0.001; // necessary in order to co
rrectly determine original partial rectangles TODO set it to size of room?
31:     private HashSet<Integer> allTags = new HashSet<Integer>(); // all port
ions of the room
32:     private WallContainerEast wallContainerEast = new WallContainerEast()
;
33:     private WallContainerNorth wallContainerNorth = new WallContainerNorth
();
34:     private WallContainerWest wallContainerWest = new WallContainerWest();
35:     private WallContainerSouth wallContainerSouth = new WallContainerSouth
();
36:     private HashSet<Rectangle> originalRectangles = new HashSet<Rectangle>
(); // all original partial rectangles of the room
37:     private ArrayList<RectangleWithTag> originalRectanglesTagged = new Arr
ayList<RectangleWithTag>(); // all tagged tagged original partial rectangles
38:
39:     @Override
40:     public ArrayList<RectangleWithTag> findOriginalPartialRectangles(IRoom
room, IRuntimeOriginalPartialRectanglesFinder rti) throws OriginalPartialRectanglesFi
nderException {
41:         try {
42:             sortWallsToContainers(room);
43:             constructOriginalPartialRectangles();
44:         } catch (WallContainerException | OriginalPartialRectanglesFin
derException e) {
45:             throw new OriginalPartialRectanglesFinderException(e);
46:         }
47:
48:         return originalRectanglesTagged;
49:     }
50:
51:     @Override
52:     public HashSet<Integer> getAllTags() {
53:         return allTags;
54:     }
55: }
56:
57: /**
58:  * Sorts all walls of the {@link IRoom} instance to suited containers
depending on their orientation
59:  * <p>
60:  * Package access is granted for testing purposes.
61:  * <p>
62:  * @param room : The {@link IRoom} for which the {@link Wall}s need to
be sorted
63:  * @throws OriginalPartialRectanglesFinderException : thrown if wall o
rientation of a wall of the {@link IRoom} cannot be
64:  * determined
65:  */
66: void sortWallsToContainers(IRoom room) throws OriginalPartialRectangl
esFinderException {
67:     Iterator<Wall> wallIterator = room.getWalls();
68:     try {
69:         while(wallIterator.hasNext()) {
70:             Wall nextWall = wallIterator.next();
71:             if (nextWall.isEastWall()) {
72:                 wallContainerEast.add(nextWall);
73:             } else if (nextWall.isNorthWall()) {
74:                 wallContainerNorth.add(nextWall);
75:             } else if (nextWall.isWestWall()) {
76:                 wallContainerWest.add(nextWall);
77:             } else if (nextWall.isSouthWall()) {
78:                 wallContainerSouth.add(nextWall);
79:             } else {
80:                 throw new OriginalPartialRectanglesFin
derException("Wall orientation cannot be determined! Wall might not be horizontal or v
ertical");
81:             }
82:         }
83:     } catch (WallContainerException wce) {
84:         throw new OriginalPartialRectanglesFinderException(wce
.getMessage());
85:     }
86:
87: }
88:
89: /**
90:  * Constructs the original partial rectangles for an {@link IRoom} and
tags the rectangles with
91:  * the {@link Wall} indices that correspond to {@link Wall}s that are
illuminated if the rectangle
92:  * is illuminated.
93:  * <p>
94:  * Package access for testing purposes.
95:  * <p>
96:  * {@link sortWallsToContainers} needs to be called first.
97:  * <p>
98:  * @throws WallContainerException : {@link Wall} handling does not wor
k.
99:  */
100: void constructOriginalPartialRectangles() throws WallContainerExceptio
n {
101:
102:     // construct original partial rectangles for each north wall

```

```

103:         for( Wall northWall : wallContainerNorth) {
104:             double yNorth = northWall.getP1().getY();
105:
106:             // find west wall
107:             double westXLimit = northWall.getP2().getX();
108:             Wall nextWestWall = wallContainerWest.getNearestWall(y
North - findWallTOL,
109:                 yNorth - findWallTOL, westXLimit);
110:             // find east wall
111:             double eastXLimit = northWall.getP1().getX();
112:             Wall nextEastWall = wallContainerEast.getNearestWall(y
North - findWallTOL, yNorth - findWallTOL, eastXLimit);
113:
114:             // find south wall
115:             double xWest = nextWestWall.getP1().getX();
116:             double xEast = nextEastWall.getP1().getX();
117:             Wall nextSouthWall = wallContainerSouth.getNearestWall
(xWest+findWallTOL, xEast-findWallTOL, yNorth);
118:             double ySouth = nextSouthWall.getP1().getY();
119:
120:             // add this original partial rectangle
121:             addOriginalPartialRectangle(yNorth, xWest, xEast, ySou
th);
122:         }
123:
124:         // construct original partial rectangles for each east wall
125:         for (Wall eastWall: wallContainerEast) {
126:
127:             // find north and south wall
128:             double xEast = eastWall.getP1().getX();
129:             double southYLimit = eastWall.getP1().getY();
130:             double northYLimit = eastWall.getP2().getY();
131:
132:             Wall nextSouthWall = wallContainerSouth.getNearestWall
(xEast- findWallTOL, xEast - findWallTOL, southYLimit);
133:             Wall nextNorthWall = wallContainerNorth.getNearestWall
(xEast- findWallTOL, xEast - findWallTOL, northYLimit);
134:
135:             // find west wall
136:             double ySouth = nextSouthWall.getP1().getY();
137:             double yNorth = nextNorthWall.getP1().getY();
138:
139:             Wall nextWestWall = wallContainerWest.getNearestWall(y
South+findWallTOL, yNorth-findWallTOL, xEast);
140:             double xWest = nextWestWall.getP1().getX();
141:
142:             // add this original partial rectangle
143:             addOriginalPartialRectangle(yNorth, xWest, xEast, ySou
th);
144:         }
145:
146:         // construct original partial rectangles for each west wall
147:         for (Wall westWall: wallContainerWest) {
148:
149:             // find north and south wall
150:             double xWest = westWall.getP1().getX();
151:             double southYLimit = westWall.getP2().getY();
152:             double northYLimit = westWall.getP1().getY();
153:
154:             Wall nextSouthWall = wallContainerSouth.getNearestWall
(xWest + findWallTOL, xWest + findWallTOL, southYLimit);
155:             Wall nextNorthWall = wallContainerNorth.getNearestWall
(xWest + findWallTOL, xWest + findWallTOL, northYLimit);
156:
157:             // find east wall
158:             double ySouth = nextSouthWall.getP1().getY();
159:             double yNorth = nextNorthWall.getP1().getY();
160:
161:             Wall nextEastWall = wallContainerEast.getNearestWall(y
South+findWallTOL, yNorth-findWallTOL, xWest);
162:             double xEast = nextEastWall.getP1().getX();
163:
164:             // add this original partial rectangle
165:             addOriginalPartialRectangle(yNorth, xWest, xEast, ySou
th);
166:         }
167:
168:         // construct original partial rectangles for each south wall
169:         for (Wall southWall: wallContainerSouth) {
170:
171:             // find east and west walls
172:             double ySouth = southWall.getP1().getY();
173:             double eastXLimit = southWall.getP2().getX();
174:             double westXLimit = southWall.getP1().getX();
175:
176:             Wall nextEastWall = wallContainerEast.getNearestWall(y
South + findWallTOL, ySouth + findWallTOL, eastXLimit);
177:             Wall nextWestWall = wallContainerWest.getNearestWall(y
South + findWallTOL, ySouth + findWallTOL, westXLimit);
178:
179:             double xEast = nextEastWall.getP1().getX();
180:             double xWest = nextWestWall.getP1().getX();
181:
182:             // find north wall
183:             Wall nextNorthWall = wallContainerNorth.getNearestWall
(xWest+findWallTOL, xEast-findWallTOL, ySouth);
184:             double yNorth = nextNorthWall.getP1().getY();
185:
186:             // add this original partial rectangle
187:             addOriginalPartialRectangle(yNorth, xWest, xEast, ySou
th);
188:         }
189:     }
190:     /**
191:      * Finds tags for original rectangles and adds it to global original r
ectangles if this rectangle does
192:      * not already exist
193:      * @param yNorth
194:      * @param xWest
195:      * @param xEast
196:      * @param ySouth
197:      */
198:     private void addOriginalPartialRectangle(double yNorth, double xWest,
double xEast, double ySouth) {
199:         Point southWestCorner = new Point(xWest,ySouth);
200:         Point northEastCorner = new Point(xEast,yNorth);
201:         Rectangle partialRectangle = new Rectangle(southWestCorner, no
rthEastCorner);
202:
203:
204:         if (!originalRectangles.contains(partialRectangle)) { // same
rectangle does not already exist -> add
205:             // determine all tags
206:             HashSet<Integer> allWallsCoveredByRectangleBoundary =
findTagsOfAllCoveredWalls(partialRectangle);
207:             // add to rectangles
208:             originalRectangles.add(partialRectangle);
209:             // add to partial rectangles

```

```

210:                originalRectanglesTagged.add(new RectangleWithTag(part
ialRectangle, allWallsCoveredByRectangleBoundary));
211:                // include all tags of this new rectangle to the tags
of the room
212:                allTags.addAll(allWallsCoveredByRectangleBoundary);
213:            }
214:        }
215:
216:    /**
217:     * Finds the tags of all covered walls of this original partialRectang
le
218:     * @param partialRectangle : the original partial rectangle to be chec
ked
219:     * @return a set of tags that corresponds to the tags of the {@link Wa
ll}s illuminated by that rectangle
220:     */
221:    private HashSet<Integer> findTagsOfAllCoveredWalls(Rectangle partialR
ectangle) {
222:        HashSet<Integer> allWallsCoveredByRectangleBoundary = new Hash
Set<Integer>();
223:
224:        //check east walls
225:        LineSegment eastWall = new LineSegment(partialRectangle.getP2(
), partialRectangle.getP3());
226:        for (Wall wall : wallContainerEast) {
227:            if (wall.getP1().isOnLineSegment(eastWall) && wall.ge
tP2().isOnLineSegment(eastWall)) { // wall of current rectangles covers a wall of of th
e room -> if the rectangle is illuminated then this wall is also illuminated
228:                allWallsCoveredByRectangleBoundary.add(wall.ge
tTag());
229:                //allTags.add(wall.getTag());
230:            }
231:        }
232:
233:        //check north walls
234:        LineSegment northWall = new LineSegment(partialRectangle.getP3
()), partialRectangle.getP4());
235:        for (Wall wall : wallContainerNorth) {
236:            if (wall.getP1().isOnLineSegment(northWall) && wall.ge
tP2().isOnLineSegment(northWall)) { // wall of current rectangles covers a wall of of th
e room -> if the rectangle is illuminated then this wall is also illuminated
237:                allWallsCoveredByRectangleBoundary.add(wall.ge
tTag());
238:                //allTags.add(wall.getTag());
239:            }
240:        }
241:
242:        //check west walls
243:        LineSegment westWall = new LineSegment(partialRectangle.getP4(
), partialRectangle.getP1());
244:        for (Wall wall : wallContainerWest) {
245:            if (wall.getP1().isOnLineSegment(westWall) && wall.ge
tP2().isOnLineSegment(westWall)) { // wall of current rectangles covers a wall of of th
e room -> if the rectangle is illuminated then this wall is also illuminated
246:                allWallsCoveredByRectangleBoundary.add(wall.ge
tTag());
247:                //allTags.add(wall.getTag());
248:            }
249:        }
250:
251:        //check south walls
252:        LineSegment southWall = new LineSegment(partialRectangle.getP1
()), partialRectangle.getP2());
253:        for (Wall wall : wallContainerSouth) {

```

```

254:            if (wall.getP1().isOnLineSegment(southWall) && wall.ge
tP2().isOnLineSegment(southWall)) { // wall of current rectangles covers a wall of of
the room -> if the rectangle is illuminated then this wall is also illuminated
255:                allWallsCoveredByRectangleBoundary.add(wall.ge
tTag());
256:                //allTags.add(wall.getTag());
257:            }
258:        }
259:
260:        return allWallsCoveredByRectangleBoundary;
261:    }
262:
263:    // for testing
264:    public Iterator<RectangleWithTag> iteratorOriginalRectangles() {
265:        return originalRectanglesTagged.iterator();
266:    }
267:
268:    // for testing
269:    Iterator<Wall> eastIterator() {
270:        return wallContainerEast.iterator();
271:    }
272:
273:    Iterator<Wall> northIterator() {
274:        return wallContainerNorth.iterator();
275:    }
276:
277:    Iterator<Wall> westIterator() {
278:        return wallContainerWest.iterator();
279:    }
280:
281:    Iterator<Wall> southIterator() {
282:        return wallContainerSouth.iterator();
283:    }
284:
285:
286:
287:
288: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.ArrayList;
4: import java.util.HashSet;
5: import java.util.Iterator;
6:
7: import fernuni.propra.algorithm.runtime_information.IRuntimeOriginalPartialRectanglesFinder;
8: import fernuni.propra.algorithm.util.Rectangle;
9: import fernuni.propra.algorithm.util.RectangleWithTag;
10: import fernuni.propra.internal_data_model.IRoom;
11: import fernuni.propra.internal_data_model.Point;
12: import fernuni.propra.internal_data_model.Wall;
13:
14: /**
15:  * tags all original partial rectangles with consecutive numbers
16:  * @author alex
17:  *
18:  */
19: public class OriginalPartialRectanglesFinder2 implements IOriginalPartialRectanglesFinder {
20:
21:     private static double findWallTOL = 0.001;
22:     private HashSet<Integer> allTags = new HashSet<Integer>();
23:     private WallContainerEast wallContainerEast = new WallContainerEast();
24:
25:     private WallContainerNorth wallContainerNorth = new WallContainerNorth();
26:     private WallContainerWest wallContainerWest = new WallContainerWest();
27:     private WallContainerSouth wallContainerSouth = new WallContainerSouth();
28:     private HashSet<Rectangle> originalRectangles = new HashSet<Rectangle>();
29:     private ArrayList<RectangleWithTag> originalRectanglesTagged = new ArrayList<RectangleWithTag>();
30:
31:     @Override
32:     public ArrayList<RectangleWithTag> findOriginalPartialRectangles(IRoom room, IRuntimeOriginalPartialRectanglesFinder rti) throws OriginalPartialRectanglesFinderException {
33:         try {
34:             sortWallsToContainers(room);
35:             constructOriginalPartialRectangles();
36:         } catch (WallContainerException | OriginalPartialRectanglesFinderException e) {
37:             throw new OriginalPartialRectanglesFinderException(e);
38:         }
39:         return originalRectanglesTagged;
40:     }
41:
42:     @Override
43:     public HashSet<Integer> getAllTags() {
44:         return allTags;
45:     }
46:
47:     void sortWallsToContainers(IRoom room) throws WallContainerException, OriginalPartialRectanglesFinderException {
48:         Iterator<Wall> wallIterator = room.getWalls();
49:         while (wallIterator.hasNext()) {
50:             Wall nextWall = wallIterator.next();
51:             if (nextWall.isEastWall()) {
52:                 wallContainerEast.add(nextWall);
53:
54:             } else if (nextWall.isNorthWall()) {
55:                 wallContainerNorth.add(nextWall);
56:             } else if (nextWall.isWestWall()) {
57:                 wallContainerWest.add(nextWall);
58:             } else if (nextWall.isSouthWall()) {
59:                 wallContainerSouth.add(nextWall);
60:             } else {
61:                 throw new OriginalPartialRectanglesFinderException("Wall orientation cannot be determined! Wall might not be horizontal or vertical");
62:             }
63:         }
64:     }
65:
66:     void constructOriginalPartialRectangles() throws WallContainerException {
67:         int rectangleNo = 0;
68:
69:         for (Wall northWall : wallContainerNorth) {
70:             double yNorth = northWall.getP1().getY();
71:             double westXLimit = northWall.getP2().getX();
72:             double eastXLimit = northWall.getP1().getX();
73:
74:             Wall nextWestWall = wallContainerWest.getNearestWall(yNorth - findWallTOL,
75:                 yNorth - findWallTOL, westXLimit);
76:
77:             Wall nextEastWall = wallContainerEast.getNearestWall(yNorth - findWallTOL, yNorth - findWallTOL, eastXLimit);
78:
79:             double xWest = nextWestWall.getP1().getX();
80:             double xEast = nextEastWall.getP1().getX();
81:
82:             Wall nextSouthWall = wallContainerSouth.getNearestWall(xWest+findWallTOL, xEast-findWallTOL, yNorth);
83:             double ySouth = nextSouthWall.getP1().getY();
84:
85:             rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
86:         }
87:
88:         for (Wall eastWall : wallContainerEast) {
89:             double xEast = eastWall.getP1().getX();
90:             double southYLimit = eastWall.getP1().getY();
91:             double northYLimit = eastWall.getP2().getY();
92:
93:             Wall nextSouthWall = wallContainerSouth.getNearestWall(xEast-findWallTOL, xEast - findWallTOL, southYLimit);
94:             Wall nextNorthWall = wallContainerNorth.getNearestWall(xEast-findWallTOL, xEast - findWallTOL, northYLimit);
95:
96:             double ySouth = nextSouthWall.getP1().getY();
97:             double yNorth = nextNorthWall.getP1().getY();
98:
99:             Wall nextWestWall = wallContainerWest.getNearestWall(ySouth+findWallTOL, yNorth-findWallTOL, xEast);
100:             double xWest = nextWestWall.getP1().getX();
101:
102:             rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
103:         }
104:     }

```

```

107:
108:         for (Wall westWall: wallContainerWest) {
109:             double xWest = westWall.getP1().getX();
110:             double southYLimit = westWall.getP2().getY();
111:             double northYLimit = westWall.getP1().getY();
112:
113:             Wall nextSouthWall = wallContainerSouth.getNearestWall
(xWest + findWallTOL, xWest + findWallTOL, southYLimit);
114:             Wall nextNorthWall = wallContainerNorth.getNearestWall
(xWest + findWallTOL, xWest + findWallTOL, northYLimit);
115:
116:             double ySouth = nextSouthWall.getP1().getY();
117:             double yNorth = nextNorthWall.getP1().getY();
118:
119:             Wall nextEastWall = wallContainerEast.getNearestWall(y
South+findWallTOL, yNorth-findWallTOL, xWest);
120:             double xEast = nextEastWall.getP1().getX();
121:
122:             rectangleNo = addOriginalPartialRectangle(rectangleNo,
yNorth, xWest, xEast, ySouth);
123:         }
124:
125:         for (Wall southWall: wallContainerSouth) {
126:             double ySouth = southWall.getP1().getY();
127:             double eastXLimit = southWall.getP2().getX();
128:             double westXLimit = southWall.getP1().getX();
129:
130:             Wall nextEastWall = wallContainerEast.getNearestWall(y
South + findWallTOL, ySouth + findWallTOL, eastXLimit);
131:             Wall nextWestWall = wallContainerWest.getNearestWall(y
South + findWallTOL, ySouth + findWallTOL, westXLimit);
132:
133:             double xEast = nextEastWall.getP1().getX();
134:             double xWest = nextWestWall.getP1().getX();
135:
136:             Wall nextNorthWall = wallContainerNorth.getNearestWall
(xWest+findWallTOL, xEast-findWallTOL, ySouth);
137:             double yNorth = nextNorthWall.getP1().getY();
138:
139:             rectangleNo = addOriginalPartialRectangle(rectangleNo,
yNorth, xWest, xEast, ySouth);
140:         }
141:     }
142:
143:     private int addOriginalPartialRectangle(int rectangleNo, double yNorth,
double xWest, double xEast, double ySouth) {
144:         Point southWestCorner = new Point(xWest, ySouth);
145:         Point northEastCorner = new Point(xEast, yNorth);
146:         Rectangle partialRectangle = new Rectangle(southWestCorner, no
rthEastCorner);
147:         if (!originalRectangles.contains(partialRectangle)) { // same
rectangle does not already exist -> add
148:             int tag = rectangleNo++;
149:             originalRectangles.add(partialRectangle);
150:             originalRectanglesTagged.add(new RectangleWithTag(part
ialRectangle, tag));
151:             allTags.add(tag);
152:         }
153:         return rectangleNo;
154:     }
155:
156:     public Iterator<RectangleWithTag> iteratorOriginalRectangles() {
157:         return originalRectanglesTagged.iterator();
158:     }

```

```

159:
160:     // TODO for tests
161:     Iterator<Wall> eastIterator() {
162:         return wallContainerEast.iterator();
163:     }
164:
165:     Iterator<Wall> northIterator() {
166:         return wallContainerNorth.iterator();
167:     }
168:
169:     Iterator<Wall> westIterator() {
170:         return wallContainerWest.iterator();
171:     }
172:
173:     Iterator<Wall> southIterator() {
174:         return wallContainerSouth.iterator();
175:     }
176:
177:
178:
179:
180: }

```



```
1: package fernuni.propra.algorithm;
2:
3: /**
4:  * Thrown if an unexpected error occurred during solution.
5:  * <p>
6:  * @author alex
7:  *
8:  */
9: public class SolveKException extends Exception {
10:
11:     public SolveKException() {
12:         super();
13:     }
14:
15:     public SolveKException(String message) {
16:         super(message);
17:     }
18:
19:     public SolveKException(Throwable cause) {
20:         super(cause);
21:     }
22:
23: }
```



```

1: package fernuni.propra.algorithm;
2:
3: /**
4:  * An abstract factory that configures the solution algorithm by instantiating
specific instances of the
5:  * interfaces {@link ICandidateSearcher}, {@link IPositionOptimizer} and {@lin
k IilluminationTester} that are
6:  * to be used within the algorithm.
7:  * <p>
8:  * The {@link AlgorithmFactory1} implements the "abstract factory" (concrete f
actory) and "singleton" design patterns.
9:  * <p>
10:  *
11:  * Implemented interfaces and super classes: {@link AbstractAlgorithmFactory}
12:  *
13:  * @author alex
14:  *
15:  */
16: public class AlgorithmFactory1 extends AbstractAlgorithmFactory{
17:     private static AlgorithmFactory1 singleton;
18:
19:     private AlgorithmFactory1() {};
20:
21:     static AlgorithmFactory1 getAlgorithmFactory1() {
22:         if (singleton == null) {
23:             singleton = new AlgorithmFactory1();
24:         }
25:         return singleton;
26:     }
27:
28:     @Override
29:     public ICandidateSearcher createCandidateSearcher() {
30:         return new CandidateSearcher();
31:     }
32:
33:     @Override
34:     public IPositionOptimizer createPositionOptimizer() {
35:         return new PositionOptimizer();
36:     }
37:
38:     @Override
39:     public IilluminationTester createIlluminationTester() {
40:         return new IlluminationTester();
41:     }
42:
43:     @Override
44:     public IOriginalPartialRectanglesFinder createOriginalPartialRectangle
sFinder() {
45:         return new OriginalPartialRectanglesFinder();
46:     }
47:
48: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.List;
4:
5: import fernuni.propra.algorithm.runtime_information.IRuntimeInformation;
6: import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
7: import fernuni.propra.internal_data_model.IRoom;
8: import fernuni.propra.internal_data_model.Lamp;
9:
10: // computes best solution in given time limit and replaces lamps with found be
st solution in room
11:
12: /**
13:  * A control class that controls the program flow for the solution use case {@
link UserSolveAAS}. It computes the
14:  * configuration of {@link Lamp}s that has a minimum number of illuminated {@l
ink Lamp}s and still illuminates
15:  * a specific {@link IRoom} instance.
16:  * <p>
17:  * {@link SolveK} forwards certain tasks to instances of other classes.
18:  * <p>
19:  * The general algorithm works as follows:
20:  * <p>
21:  * 1.) A number of possible {@link Lamp} candidates is computed
22:  * by forwarding to an instance of {@link ICandidateSearcher} that is obta
ined from
23:  * the {@link AbstractAlgorithmFactory} singleton.
24:  * <p>
25:  * 2.) The candidates are then provided to an instance of {@link IPositionOpti
mizer} that is also
26:  * obtained from the {@link AbstractAlgorithmFactory} singleton. The {@lin
k IPositionOptimizer}
27:  * finds an optimal configuration of lamp positions.
28:  * <p>
29:  * 3.) The currently available best solution replaces the lamps of the provide
d {@link IRoom} instance
30:  * and the number of illuminated {@link Lamp}s in that best solution is st
ored.
31:  * <p>
32:  * 4.) The number of illuminated {@link Lamps} in best solution can be obtaine
d.
33:  * <p>
34:  * {@link SolveK} extends {@link Thread} in order to make it possible to stop t
he computation externally (e.g. after
35:  * a time limit has passed) by calling the interrupt() method of an instance o
f {@link SolveK}. This sets the
36:  * interruptedFlag which is checked at several stages in the solve method. If
the flag is set, an {@link InterruptedException}
37:  * is thrown, which leads to a stop of the algorithm. If a valid solution is a
vailable, this solution is stored as noted in 4.)
38:  * <p>
39:  * The computation must be started by calling the inherited start() method.
40:  * <p>
41:  * {@link SolveK} also supports the wait-notify mechanism for synchronizing, b
y allowing to let clients test whether the solution is already
42:  * available and waiting until the computation is done. So the algorithm provi
ded can be started externally, and subsequently a client can wait until
43:  * the computation is done or interrupt it whenever it wants. This means that
a solution might not be available at all (i.e. if a client
44:  * prematurely interrupts {@link SolveK}). Therefore, clients should check the
return value of
45:  * {@link testIfComputationIsFinished} for an exception. Only if this return v
alue is null, no exception has
46:  * occurred and a valid solution has been stored to the {@link IRoom} instance
);
47:  * <p>
48:  *
49:  *
50:  * <p>
51:  * Implemented superclasses : {@link Thread}
52:  * @author alex
53:  *
54:  */
55: public class SolveK extends Thread{
56:     private IRuntimeInformation runTimeInformation;
57:     private IRoom room;
58:     private ICandidateSearcher candidateSearcher;
59:     private IPositionOptimizer positionOptimizer;
60:     private boolean computationFinished;
61:     private volatile List<Lamp> bestSolution;
62:     private volatile int numberLampsOnBestSolution;
63:     private volatile SolveKException exception = null; // to be communicat
ed to main thread
64:
65:     /**
66:      * Constructor for {@link SolveK}
67:      * @param room : An {@link IRoom} instance for which an optimal {@link
Lamp} configuration has
68:      * to be found. The {@link Lamp} configur
ation is stored to this {@link IRoom} instance
69:      * @param runTimeInformation : An instance of {@link IRuntimeInformati
on} to which detailed runtime information can be stored.
70:      */
71:     public SolveK(IRoom room, IRuntimeInformation runTimeInformation) {
72:         this.room = room;
73:         this.runTimeInformation = runTimeInformation;
74:         this.candidateSearcher = AbstractAlgorithmFactory.getAlgorithm
Factory().createCandidateSearcher();
75:         this.positionOptimizer = AbstractAlgorithmFactory.getAlgorithm
Factory().createPositionOptimizer();
76:     }
77:
78:     /**
79:      * Provides the steps 1.) and 2.) of the algorithm specified in the do
cumentation of {@link SolveK}
80:      * @param runTimeInformation : An instance of {@link IRuntimeInformati
on} that can be used to save runtime information
81:      * @throws SolveKException : thrown if an unexpected error is thrown i
n the solution procedure
82:      * @throws InterruptedException : thrown if the solution procedure is
interrupted. This is done intentionally to allow
83:      * interruption by clients and is catche
d in run().
84:      */
85:     private void solve(IRuntimeInformation runTimeInformation) throws Solv
eKException, InterruptedException{
86:         List<Lamp> candidates;
87:         try {
88:             // 1.) search lamp candidates
89:             runTimeInformation.startTimeCandidateSearch();
90:             try {
91:                 candidates = candidateSearcher.searchCandidate
s(room,
92:                     runTimeInformation);
93:             } catch(InterruptedException ie) {
94:                 runTimeInformation.stopTimeCandidateSearch();
95:                 throw new InterruptedException(ie.getMessage());

```

```

96:         }
97:         runTimeInformation.stopTimeCandidateSearch();
98:         System.out.println("Number of candidates found: " + ca
ndidates.size());
99:
100:        // 2.) find optimal configuration of lamps
101:        runTimeInformation.startTimeOptimizePositions();
102:        try {
103:            positionOptimizer.optimizePositions(
104:                candidates, runTimeInformation
);
105:        } catch (InterruptedException ie) {
106:            runTimeInformation.stopTimeOptimizePositions()
;
107:            throw new InterruptedException(ie.getMessage()
);
108:        }
109:        runTimeInformation.stopTimeOptimizePositions();
110:
111:        } catch (CandidateSearcherException e) {
112:            throw new SolveKException(e); // something went wrong
113:        } catch (RuntimeExceptionLamps rte) {
114:            throw new SolveKException(rte); // something went wron
g
115:        }
116:    }
117:
118:    @Override
119:    public void run() {
120:        try {
121:            solve(runTimeInformation);
122:        } catch (SolveKException e) {
123:            this.exception = e;
124:        } catch (InterruptedException ie) {
125:        }
126:
127:        //write output and set best solution
128:        bestSolution = positionOptimizer.getCurrentBestSolution();
129:        numberLampsOnBestSolution = positionOptimizer.getNumberOfOnLam
psBestSolution();
130:        if(bestSolution != null) { // null if interrupted or exception
at candidate searcher -> no solution available
131:            // optimal configuration is supplied to room
132:            room.replaceLamps(bestSolution);
133:        } else {
134:            exception = new SolveKException("Not enough time to co
mpute a solution!"); // exceptions to be passed to client
135:        }
136:        setComputationFinished(true);
137:    }
138:
139:    /**
140:     * Sets switch and also notifies those threads waiting for a result in
the waiting queue
141:     * @param computationFinished
142:     */
143:    private synchronized void setComputationFinished(boolean computati
nished) {
144:        this.computationFinished = computationFinished;
145:        notifyAll();
146:    }
147:
148:    /**
149:     * Allows clients to join the waiting queue and wait for a result
150:     * @return {@link SolveKException} that shows if a solution is availab
le (return value is null) or not
151:     * @throws InterruptedException
152:     */
153:    public synchronized SolveKException testIfComputationFinished() throws
InterruptedException{
154:        while(!computationFinished) {
155:            wait();
156:        }
157:        return exception;
158:    }
159:
160:    /**
161:     * Clients can obtain the number of lamps that are turned on in the be
st solution.
162:     * @return Number of lamps that are turned on in best solution.
163:     */
164:    public synchronized int getNumberOfOnLampsBestSolution() {
165:        return numberLampsOnBestSolution;
166:    }
167:
168:
169: }

```

```

1: package fernuni.propra.algorithm;
2:
3: import java.util.HashSet;
4: import java.util.Iterator;
5: import java.util.List;
6:
7: import fernuni.propra.algorithm.runtime_information.IRuntimeIlluminationTester
;
8: import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
9: import fernuni.propra.algorithm.util.RectangleWithTag;
10: import fernuni.propra.internal_data_model.IRoom;
11: import fernuni.propra.internal_data_model.Lamp;
12:
13: /**
14:  * A specific provider of an algorithm that can check whether a room is illumi
nated by a set of {@link Lamp}s
15:  * or not.
16:  * <p>
17:  * Several methods are provided for this purpose.
18:  * <p>
19:  * The algorithm for {@link testIfRoomIsIlluminated}({@link IRoom} room, {@lin
k IRuntimeIlluminationTester} runtimeInfo) works as follows:
20:  * <p>
21:  * 1.) Find all original partial rectangles by forwarding to {@link OriginalPa
rtialRectanglesFinder} and determine the tags of all "parts" that consitute the room
22:  * e.g. all walls.
23:  * <p>
24:  * 2.) Iterate over all {@link Lamp}s in room and compute the set of illuminat
ed rectangles by checking if an illuminated
25:  * {@link Lamp} is inside a rectangle and (if yes) adding the tags of that re
ctangle to the set of illuminated rectangles.
26:  * <p>
27:  * 3.) Check if the set of tags of illuminated rectangles contains all tags of
the room.
28:  * <p>
29:  * <p>
30:  * The algorithm for {@link testIfRoomIsIlluminated}({@link Iterator}<{@link Lam
p}> taggedLampsIterator, {@link HashSet}<{@link Integer}> allTags, {@link IRuntimeIllu
minationTester} runtimeInfo) works as follows:
31:  * <p>
32:  * 1.) Iterate over tagged lamps and construct a set of tags of illuminated or
iginal rectangles
33:  * <p>
34:  * 2.) Check if the set of tags of illuminated rectangles contains all tags of
original rectangles
35:  * <p>
36:  * <p>
37:  * The algorithm for {@link testIfRoomIsIlluminated}({@link HashSet}<{@link Inte
ger}> illuminatedTags, {@link HashSet}<{@link Integer}> allTags, {@link IRuntimeIllumi
nationTester} runtimeInfo) works as follows:
38:  * <p>
39:  * 1.) Check if the set of tags of illuminated rectangles (illuminatedTags) co
ntains all tags of original rectangles (allTags)
40:  * <p>
41:  * <p>
42:  * Implemented interfaces and super classes: {@link ICandidateSearcher}
43:  * @author alex
44:  *
45:  */
46: public class IlluminationTester implements IilluminationTester{
47:
48:     public IlluminationTester() {}
49:
50:
51:     @Override
52:     public boolean testIfRoomIsIlluminated(IRoom room, IRuntimeIlluminatio
nTester runtimeInfo) throws IlluminationTesterException {
53:         try {
54:             // find original rectangles
55:             IOriginalPartialRectanglesFinder originalRectanglesFin
der = AbstractAlgorithmFactory.getAlgorithmFactory().createOriginalPartialRectanglesFin
der();
56:             runtimeInfo.startTimeOriginalPartialRectanglesFind();
57:             List<RectangleWithTag> rectanglesWithTag = originalRec
tanglesFinder.findOriginalPartialRectangles(room, runtimeInfo);
58:             runtimeInfo.stopTimeOriginalPartialRectanglesFind();
59:
60:             // store all tags
61:             HashSet<Integer> allTags = originalRectanglesFinder.ge
tAllTags();
62:
63:             // compute set of tags of illuminated lamps
64:             HashSet<Integer> tagsOfAllIlluminatedLamps = new HashS
et<Integer>();
65:             Iterator<Lamp> lampIterator = room.getLamps();
66:             while(lampIterator.hasNext()) {
67:                 Lamp lamp = lampIterator.next();
68:                 if(lamp.getOn()) {
69:                     for(RectangleWithTag rec : rectanglesW
ithTag) {
70:                         if(lamp.isInsideRectangle(rec.
getP1(), rec.getP3())) {
71:                             Iterator<Integer> tagI
terator = rec.getCopyOfTags().iterator();
72:                             //Iterator<Integer> ta
gIterator = rec.getTagIterator();
73:                             while(tagIterator.hasN
ext()) {
74:                                 tagsOfAllIllum
inatedLamps.add(tagIterator.next());
75:                             }
76:                         }
77:                     }
78:                 }
79:             }
80:             // check if the set of tags of illuminated rectangles
contains the tags of all rectangles
81:             if(tagsOfAllIlluminatedLamps.containsAll(allTags)) {
82:                 return true;
83:             } else {
84:                 return false;
85:             }
86:
87:         } catch (OriginalPartialRectanglesFinderException e) {
88:             throw new IlluminationTesterException(e);
89:         } catch (RuntimeExceptionLamps rte) {
90:             throw new IlluminationTesterException(rte);
91:         }
92:     }
93:
94:     @Override
95:     public boolean testIfRoomIsIlluminated(Iterator<Lamp> taggedLampsItera
tor, HashSet<Integer> allTags, IRuntimeIlluminationTester runtimeInfo) {
96:         return illuminatedLampsCoverAllTags(taggedLampsIterator, allTa
gs);
97:     }
98:
99:     private static boolean illuminatedLampsCoverAllTags(Iterator<Lamp> tag

```

```

gedLampsIterator, HashSet<Integer> allTags) {
100:         // compute set of tags of illuminated rectangles
101:         HashSet<Integer> tagsOfAllIlluminatedLamps = new HashSet<Integer>();
102:         while(taggedLampsIterator.hasNext()) {
103:             Lamp lamp = taggedLampsIterator.next();
104:             if (lamp.getOn()) {
105:                 Iterator<Integer> tagIterator = lamp.iteratorTags();
106:                 while(tagIterator.hasNext()) {
107:                     tagsOfAllIlluminatedLamps.add(tagIterator.next());
108:                 }
109:             }
110:         }
111:         if (tagsOfAllIlluminatedLamps.containsAll(allTags)) {
112:             return true;
113:         } else {
114:             return false;
115:         }
116:     }
117:
118:     @Override
119:     public boolean testIfRoomIsIlluminated(HashSet<Integer> illuminatedTags,
120:     HashSet<Integer> allTags,
121:         IRuntimeIlluminationTester runtimeInfo) {
122:         return illuminatedTags.containsAll(allTags);
123:     }

```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.Comparator;
4: import java.util.Iterator;
5:
6: import fernuni.propra.internal_data_model.Wall;
7:
8: /**
9:  * A specific container that stores east walls. Those {@link Wall}s can be spe
cified by
10:  * two {@link Point}s in a horizontal-vertical coordinate system. The {@link W
all}s
11:  * in this container are ordered in ascending order with respect to the horizo
ntal component
12:  * (x-component) of their {@link Point}s.
13:  * <p>
14:  * The total ordering requested by {@link WallContainerAbstract} is such that
walls
15:  * <p>
16:  * Extended classes and implemented interfaces: {@link WallContainerAbstract}.
17:  * <p>
18:  * @author alex
19:  *
20:  */
21: public class WallContainerEast extends WallContainerAbstract {
22:
23:     @Override
24:     protected boolean isValidWall(Wall wall, double limit, double low, dou
ble high) {
25:         return wall.overlapsYrange(low, high) && wall.getP1().getX()>
=limit;
26:     }
27:
28:
29:     @Override
30:     protected Comparator<Wall> getComparator() {
31:         return new Comparator<Wall>() {
32:             @Override
33:             public int compare(Wall o1, Wall o2) {
34:                 if (o1.getP1().getX() < o2.getP1().getX()) {
35:                     return -1;
36:                 } else if (o1.getP1().getX()>o2.getP1().getX()
) {
37:                     return 1;
38:                 }
39:                 return 0;
40:             }
41:         };
42:     }
43:
44:
45:     @Override
46:     protected boolean isCorrectWallType(Wall wall) {
47:         return wall.isEastWall();
48:     }
49: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.List;
4:
5: import fernuni.propra.algorithm.runtime_information.IRuntimeCandidateSearcher;
6: import fernuni.propra.internal_data_model.IRoom;
7: import fernuni.propra.internal_data_model.Lamp;
8: import fernuni.propra.internal_data_model.Point;
9:
10: /**
11:  *
12:  * A provider of an algorithm that can compute a {@link List} of potential {@link Lamp} positions
13:  * for an instance of {@link IRoom}.
14:  *
15:  * <p>
16:  * Implementing classes: {@link CandidateSearcher}
17:  * <p>
18:  *
19:  * @author alex
20:  *
21:  *
22:  *
23:  */
24: public interface ICandidateSearcher {
25:
26:     /**
27:      * A method that provides the functionality of {@link ICandidateSearcher} to callers. It returns a {@link List} of
28:      * {@link Lamp}s that are potential lamp positions at which lamps might be placed to illuminate the room.
29:      * @param room : an instance of {@link IRoom} for which the lamp positions are to be determined.
30:      * @param runtimeCandidateSearcher : an instance of {@link IRuntimeCandidateSearcher} to which runtime information can be saved
31:      * @return a {@link List} of
32:      * {@link Lamp}s that contains potential lamp positions for the provided {@link IRoom}
33:      * @throws CandidateSearcherException : thrown if an error occurs during execution
34:      * @throws InterruptedException : thrown if the executing thread is interrupted, e.g. to stop after a certain time
35:      */
36:     List<Lamp> searchCandidates(IRoom room, IRuntimeCandidateSearcher runtimeCandidateSearcher) throws CandidateSearcherException, InterruptedException;
37:
38: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import java.util.ArrayList;
4: import java.util.HashSet;
5: import java.util.Iterator;
6: import java.util.LinkedList;
7: import java.util.List;
8:
9: import fernuni.propra.algorithm.runtime_information.IRuntimePositionOptimizer;
10: import fernuni.propra.internal_data_model.IRoom;
11: import fernuni.propra.internal_data_model.Lamp;
12:
13: /**
14:  * A specific provider of an algorithm that finds a minimum set (and number) o
f tagged {@link Lamp}s that
15:  * illuminates an {@link IRoom} instance.
16:  * <p>
17:  * The algorithm works as follows:
18:  * <p>
19:  * 1.) Global (to this class) fields are introduced
20:  * that store the currently best solution ({@link List}<{@link Lamp}>) and the
number of {@link Lamp}s
21:  * that are turned on in the currently best solution
22:  * <p>
23:  * 2.) All {@link Lamp}s in the supplied list are turned on (i.e. the {@link I
Room} represented by the portions
24:  * represented by the tags of the {@link Lamp}s is illuminated. The number
of illuminated {@link Lamp}s is
25:  * consequently set to the size of the originally supplied set of lamps.
26:  * <p>
27:  * 3.) An index idx that can be used to navigate the set of {@link Lamp}s is i
ntroduced (idx = 0 initially)
28:  * and the ideal configuration of {@link Lamp}s is computed by recursively
calling the
29:  * method searchSolution in the manner of a backtracking algorithm. The me
thod follows the following pseudo-code
30:  * <p>
31:  * PROCEDURE searchSolution (lamps, idx) {
32:  * <p>
33:  * if ( check if all portions of room are illuminated) { // if not all oth
er branches cannot illuminate the room either
34:  * <p>
35:  * if(number of turned on lamps < number of on lamps in best solution
) {
36:  * <p>
37:  * best solution = lamps
38:  * <p>
39:  * number of lamps in best solution = number of turned on lamps }
40:  * <p>
41:  * if (idx < size of lamps) {
42:  * <p>
43:  * searchSolution(lamps, idx+1)
44:  * <p>
45:  * turn off lamp[idx]
46:  * <p>
47:  * searchSolution(lamps, idx+1)
48:  * <p>
49:  * }
50:  * <p>
51:  * }
52:  * <p>
53:  * }
54:  * <p>
55:  * 4.) The computation can be interrupted by interrupting the executing threa

```

```

d. The computation will
56:  * stop immediately with an {@link InterruptedException}.
57:  * <p>
58:  * 5.) The currently available best solution can now be obtained.
59:  *
60:  * <p>
61:  * Implemented interfaces and super classes: {@link IPositionOptimizer}
62:  *
63:  * @author alex
64:  *
65:  */
66: public class PositionOptimizer implements IPositionOptimizer{
67:     private static List<Lamp> currentBestSolution;
68:     private static int numberIlluminatedLampsBestSolution;
69:     private static IIlluminationTester illuminationTester = AbstractAlgori
thmFactory.getAlgorithmFactory().createIlluminationTester();
70:
71:     public PositionOptimizer() {
72:     }
73:
74:     @Override
75:     public List<Lamp> optimizePositions(List<Lamp> taggedCandidates, IRunt
imePositionOptimizer runTimeInformation) throws InterruptedException
76:     {
77:
78:         // all lamps are on -> illuminated
79:         currentBestSolution = taggedCandidates;
80:         numberIlluminatedLampsBestSolution = taggedCandidates.size();
81:
82:         HashSet<Integer> allTags = new HashSet<Integer>();
83:         for (Lamp lamp : taggedCandidates) {
84:             lamp.turnOn(); // make sure all lamps are turned on
85:             Iterator<Integer> tagIterator = lamp.iteratorTag();
86:             while(tagIterator.hasNext()) {
87:                 allTags.add(tagIterator.next());
88:             }
89:         }
90:
91:         ArrayList<Lamp> lamps = deepCopyLamps(taggedCandidates);
92:         searchSolution(lamps, 0, allTags, numberIlluminatedLampsBestSol
ution, runTimeInformation);
93:
94:         return currentBestSolution;
95:
96:     }
97:
98:     /**
99:     * Implements the backtracking algorithm as indicated in the commentar
y on the {@link PositionOptimizer} class.
100:     * @param lamps : the set of lamps for which an optimal configuration
is to be found
101:     * @param idx : index of current iteration
102:     * @param allTags : a set of all portions of the room that need to be
illuminated
103:     * @param numberLampsOn : number of lamps that are turned on in the pr
vided lamps argument
104:     * @param runTimeInformation : a data structure that can be used to st
ore runtime information
105:     * @throws InterruptedException
106:     */
107:     private void searchSolution(ArrayList<Lamp> lamps, int idx,

```

```

111:         HashSet<Integer> allTags, int numberLampsOn, IRuntimeP
ositionOptimizer runTimeInformation) throws InterruptedException{
112:             if(Thread.currentThread().isInterrupted()) {
113:                 throw new InterruptedException("Computation interrupte
d.");
114:             }
115:
116:             if(illuminationTester.testIfRoomIsIlluminated(lamps.iterator()
, allTags, runTimeInformation)) { /* valid solution found, else case
117:                 does not need to be investigated since it all lamps wi
th an idx larger than the supplied idx are already turned on*/
118:                 if (numberLampsOn<numberIlluminatedLampsBestSolution)
{ // new best solution found
119:                     System.out.println("Solution found with " + nu
mberLampsOn + " lamps turned on.");
120:                     currentBestSolution = deepCopyLamps(lamps);
121:                     numberIlluminatedLampsBestSolution = numberLam
psOn;
122:                 }
123:                 if (idx < lamps.size()) { // there are further configu
rations that can be investigated in this branch
124:                     Lamp lamp = lamps.get(idx);
125:                     //branch 1
126:                     // lamp does not need to be turned on since it
has been initialized as turned on
127:                     searchSolution(deepCopyLamps(lamps), idx+1, al
lTags, numberLampsOn, runTimeInformation);
128:
129:                     //branch2
130:                     lamp.turnOff();
131:                     searchSolution(deepCopyLamps(lamps), idx+1, al
lTags, numberLampsOn-1, runTimeInformation);
132:
133:                 }
134:
135:                 } else { // not a valid solution, with all lamps > idx turned
on
136:
137:                 }
138:             }
139:
140:             /**
141:              * Provides functionality to deep copy an ArrayList of {@link Lamp}s
142:              * @param lamps
143:              * @return a deep copy of the provided list of {@link Lamp}s
144:              */
145:             private static ArrayList<Lamp> deepCopyLamps(List<Lamp> lamps) {
146:                 ArrayList<Lamp> outLamps = new ArrayList<Lamp>(lamps.size());
147:                 Iterator<Lamp> lampsIterator = lamps.iterator();
148:                 while(lampsIterator.hasNext()) {
149:                     Lamp lamp = lampsIterator.next();
150:                     outLamps.add(lamp.deepCopy());
151:                 }
152:                 return outLamps;
153:             }
154:
155:
156:             private static HashSet<Integer> deepCopyHashSet(HashSet<Integer> hashS
et) {
157:                 HashSet<Integer> outHashSet = new HashSet<Integer>();
158:                 for (Integer integer : hashSet) {
159:                     Integer outInteger = (int) integer;
160:                     outHashSet.add(outInteger);
161:                 }
162:
163:                 return outHashSet;
164:             }
165:
166:             @Override
167:             public List<Lamp> getCurrentBestSolution() {
168:                 if (currentBestSolution == null) {
169:                     return null;
170:                 }
171:                 List<Lamp> outLamps = new LinkedList<Lamp>();
172:                 Iterator<Lamp> lampIterator = currentBestSolution.iterator();
173:                 while(lampIterator.hasNext()) {
174:                     outLamps.add(lampIterator.next().deepCopy());
175:                 }
176:                 return outLamps;
177:             }
178:
179:             @Override
180:             public int getNumberOfOnLampsBestSolution() {
181:                 return numberIlluminatedLampsBestSolution;
182:             }
183:
184:
185:
186:
187: }

```

```

1: package fernuni.propra.algorithm;
2:
3: import java.util.ArrayList;
4: import java.util.HashSet;
5:
6: import fernuni.propra.algorithm.runtime_information.IRuntimeOriginalPartialRectanglesFinder;
7: import fernuni.propra.algorithm.util.RectangleWithTag;
8: import fernuni.propra.internal_data_model.IRoom;
9: /**
10:  * A provider of an algorithm that can find the original partial rectangles for an {@link IRoom} instance.
11:  * <p>
12:  * Implementing classes: {@link OriginalPartialRectanglesFinder}
13:  * <p>
14:  * @author alex
15:  *
16:  */
17: public interface IOriginalPartialRectanglesFinder {
18:     /**
19:      * The original partial rectangles of an {@link IRoom} are tagged with {@link Integer}s that
20:      * denote "parts" (e.g. walls) of the room that are illuminated if the associated partial rectangle is illuminated.
21:      * If all tags are illuminated then the room is illuminated.
22:      * This method returns a set that contains all tags of all original partial rectangles of the {@link IRoom}.
23:      * <p>
24:      * The {@link findOriginalPartialRectangles} method needs to be called first in order for {@link getAllTags}() to work.
25:      * @return : {@link HashSet}<{@link Integer}> a set of all tags of the original partial rectangles of the {@link IRoom} parameter
26:      * of the previously called {@link findOriginalPartialRectangles} method.
27:      */
28:     HashSet<Integer> getAllTags();
29:     /**
30:      * Returns all original partial rectangles of an {@link IRoom} parameter and saves runtime information
31:      * to the instance of {@link IRuntimeOriginalPartialRectanglesFinder}. All rectangles are tagged with Integers that denote
32:      * the parts of the room (e.g. walls) that are illuminated if the rectangle is illuminated. If all tags are illuminated then it
33:      * must follow that the room is illuminated. Identical rectangles are only stored once.
34:      * @param room : the {@link IRoom} instance for which the original partial rectangles are to be determined
35:      * @param rt : a data structure of type {@link IRuntimeOriginalPartialRectanglesFinder} that can be used to store runtime information.
36:      * @return a list of tagged original partial rectangles of the room.
37:      * @throws OriginalPartialRectanglesFinderException
38:      */
39:     ArrayList<RectangleWithTag> findOriginalPartialRectangles(IRoom room, IRuntimeOriginalPartialRectanglesFinder rt) throws OriginalPartialRectanglesFinderException;
40:
41: }

```



```

1: package fernuni.propra.algorithm;
2:
3:
4: /**
5:  * Specifies an interface for the construction of parts of an algorithm define
d by consistent
6:  * instances of {@link ICandidateSearcher}, {@link IPositionOptimizer} and {@l
ink IilluminationTester}.
7:  * <p>
8:  * Implements the abstract factory design pattern. Subclasses, i.e. "concrete
factories "
9:  * must implement this interface.
10:  * <p>
11:  *
12:  * Extending classes: {@link AlgorithmFactory1}
13:  *
14:  *
15:  * @author alex
16:  *
17:  */
18: public abstract class AbstractAlgorithmFactory {
19:
20:     /**
21:      * Provides an instance of a "concrete factory", that can deliver cons
istent {@link ICandidateSearcher},
22:      * {@link IPositionOptimizer} and {@link IilluminationTester} objects.
23:      * @return An instance of a "concrete factory".
24:      */
25:     public static AbstractAlgorithmFactory getAlgorithmFactory() {
26:         return AlgorithmFactory1.getAlgorithmFactory1();
27:     }
28:
29:     /**
30:      * Delivers an instance of {@link ICandidateSearcher} that works with
the algorithms defined by the "concrete factory".
31:      * @return A consistent instance of {@link ICandidateSearcher}.
32:      */
33:     public abstract ICandidateSearcher createCandidateSearcher();
34:
35:     /**
36:      * Delivers an instance of {@link IPositionOptimizer} that works with
the algorithms defined by the "concrete factory".
37:      * @return A consistent instance of {@link IPositionOptimizer}.
38:      */
39:     public abstract IPositionOptimizer createPositionOptimizer();
40:
41:     /**
42:      * Delivers an instance of {@link IilluminationTester} that works with
the algorithms defined by the "concrete factory".
43:      * @return A consistent instance of {@link IilluminationTester}
44:      */
45:     public abstract IilluminationTester createIlluminationTester();
46:
47:     /**
48:      * Delivers an instance of {@link IOriginalPartialRectanglesFinder} th
at works with the algorithms defined by the inheriting "concrete factor"
49:      * @return
50:      */
51:     public abstract IOriginalPartialRectanglesFinder createOriginalPartial
RectanglesFinder();
52:
53: }

```



```

1: package Algorithm_Component;
2:
3: import fernuni.propra.algorithm.*;
4:
5: import org.junit.Test;
6: import static org.junit.Assert.*;
7:
8: /*
9:  * Informationen über das Unit-Testen mit Hilfe von JUnit finden Sie unter ht
tp://www.vogella.com/tutorials/JUnit/article.html.
10:  * In dem dort hinterlegten Dokument sind alle notwendigen Hilfsmittel erl ut
ert.
11:  *
12:  * Designen Sie Ihre Unit-Tests nach dem Arrange-Act-Assert-Prinzip
13:  */
14:
15: public class API_Test_Validation {
16:
17:     @Test
18:     public void validateFileHasToBeValid() {
19:         // Arrange
20:         IAusleuchtung api = new Ausleuchtung();
21:         // Act
22:         boolean solutionValid = api.validateSolution("");
23:         // Assert
24:         assertTrue("Ohne Angabe einer Datei wurde eine zul ssige L s
ung gefunden.", !solutionValid);
25:     }
26:
27:     @Test
28:     public void validateTruePositive() {
29:         // Arrange
30:         IAusleuchtung api = new Ausleuchtung();
31:         // Act
32:         boolean solutionValid = api.validateSolution("instances/valida
tionInstances/Selbsttest_20a_solved.xml");
33:         // Assert
34:         assertTrue("Eine zul ssige L sung wurde als nicht zul ssig
gewertet.", solutionValid);
35:     }
36:
37:     @Test
38:     public void validateTrueNegative() {
39:         // Arrange
40:         IAusleuchtung api = new Ausleuchtung();
41:         // Act
42:         boolean solutionValid = api.validateSolution("instances/valida
tionInstances/Selbsttest_20a_incomplete.xml");
43:         // Assert
44:         assertTrue("Eine unzul ssige L sung wurde als zul ssig gewe
rtet.", !solutionValid);
45:     }
46:
47: }

```



```

1: package fernuni.propra.file_processing;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.ArrayList;
6: import java.util.LinkedList;
7: import java.util.List;
8:
9: import org.junit.Before;
10: import org.junit.Test;
11:
12: import fernuni.propra.file_processing.FilePersistence;
13: import fernuni.propra.file_processing.PersistenceException;
14: import fernuni.propra.internal_data_model.IRoom;
15: import fernuni.propra.internal_data_model.LineSegment;
16: import fernuni.propra.internal_data_model.Point;
17: import fernuni.propra.internal_data_model.Room;
18:
19: public class FilePersistenceTest {
20:     Point p1,p2,p3,p4,p5;
21:     LineSegment l1,l2,l3,l4,l5;
22:     List<LineSegment> lineSegments;
23:     IRoom room;
24:     LinkedList<Point> corners;
25:
26:
27:     @Before
28:     public void setUp() {
29:         p1 = new Point (0,0);
30:         p2 = new Point (1,0);
31:         p3 = new Point (1,1);
32:         p4 = new Point (0,1);
33:         l1 = new LineSegment (p1, p2);
34:         l2 = new LineSegment (p2, p3);
35:         l3 = new LineSegment (p3,p4);
36:         l4 = new LineSegment (p4,p1);
37:         l5 = new LineSegment (p1, p3);
38:         lineSegments = new ArrayList<LineSegment>();
39:         lineSegments.add(l1);lineSegments.add(l2); lineSegments.add(l3
); lineSegments.add(l4);
40:
41:         corners= new LinkedList<Point>();
42:         corners.add(p1); corners.add(p2); corners.add(p3); corners.add
(p4);
43:
44:         room = new Room("test", null, corners);
45:     }
46:
47:
48:     @Test
49:     public void testTestWallAndAddToWalls() {
50:         //Arrange
51:         LineSegment lccw1 = new LineSegment (p1,p4);
52:         LineSegment lccw2 = new LineSegment (p4, p3);
53:         LineSegment lccw3 = new LineSegment (p3,p2);
54:         LineSegment lccw4 = new LineSegment (p2, p1);
55:
56:         ArrayList<LineSegment> walls1 = new ArrayList<LineSegment>();
57:         ArrayList<LineSegment> walls2 = new ArrayList<LineSegment>();
58:         walls2.add(lccw1);
59:         ArrayList<LineSegment> walls3 = new ArrayList<LineSegment>();
60:         walls3.add(lccw1); walls3.add(lccw2);
61:         ArrayList<LineSegment> walls4 = new ArrayList<LineSegment>();
62:         walls4.add(lccw1); walls4.add(lccw2); walls4.add(lccw3);
63:
64:
65:         ArrayList<LineSegment> walls5 = new ArrayList<LineSegment>();
66:         ArrayList<LineSegment> walls6 = new ArrayList<LineSegment>();
67:         walls6.add(l1);
68:         ArrayList<LineSegment> walls7 = new ArrayList<LineSegment>();
69:         walls7.add(l1); walls7.add(l2);
70:         ArrayList<LineSegment> walls8 = new ArrayList<LineSegment>();
71:         walls8.add(l1); walls8.add(l2); walls8.add(l3);
72:
73:         //Act, Assert
74:         try {
75:             FilePersistence.testAndAddWallToWalls(lccw1, walls1);
76:         } catch (PersistenceException e) {
77:             fail(e.getMessage());
78:         }
79:         try {
80:             FilePersistence.testAndAddWallToWalls(lccw2, walls2);
81:         } catch (PersistenceException e) {
82:             fail(e.getMessage());
83:         }
84:         try {
85:             FilePersistence.testAndAddWallToWalls(lccw3, walls3);
86:         } catch (PersistenceException e) {
87:             fail(e.getMessage());
88:         }
89:         try {
90:             FilePersistence.testAndAddWallToWalls(lccw4, walls4);
91:         } catch (PersistenceException e) {
92:             fail(e.getMessage());
93:         }
94:
95:         try {
96:             FilePersistence.testAndAddWallToWalls(l1, walls5);
97:         } catch (PersistenceException e) {
98:             fail(e.getMessage());
99:         }
100:         try {
101:             FilePersistence.testAndAddWallToWalls(l2, walls6);
102:         } catch (PersistenceException e) {
103:             fail(e.getMessage());
104:         }
105:         try {
106:             FilePersistence.testAndAddWallToWalls(l3, walls7);
107:         } catch (PersistenceException e) {
108:             fail(e.getMessage());
109:         }
110:         try {
111:             FilePersistence.testAndAddWallToWalls(l4, walls8);
112:         } catch (PersistenceException e) {
113:             fail(e.getMessage());
114:         }
115:
116:         //Assert
117:         assertEquals(walls1.get(0), lccw1);
118:         assertEquals(walls2.get(1), lccw2);
119:         assertEquals(walls3.get(2), lccw3);
120:         assertEquals(walls4.get(3), lccw4);
121:         assertEquals(walls5.get(0), l1);
122:         assertEquals(walls6.get(1), l2);
123:         assertEquals(walls7.get(2), l3);
124:         assertEquals(walls8.get(3), l4);
125:
126:     }

```

```

127:
128:
129:     @Test
130:     public void testReadInput() {
131:         //Arrange
132:         String[] xmlPathesOK = {"instances/validationInstances/Selbstt
est_clockwise.xml",
133:                                "instances/validationInstances/Selbsttest_coun
terClockwise.xml",
134:                                "instances/validationInstances/Selbsttest_100a
_incomplete.xml",
135:                                "instances/validationInstances/Selbsttest_100a
_incomplete.xml",
136:                                "instances/validationInstances/Selbsttest_100a
_solved.xml",
137:                                "instances/validationInstances/Selbsttest_100a
.xml",
138:                                "instances/validationInstances/Selbsttest_100b
.xml",
139:                                "instances/validationInstances/Selbsttest_20a_
incomplete.xml",
140:                                "instances/validationInstances/Selbsttest_20a_
solved.xml",
141:                                "instances/validationInstances/Selbsttest_20a.
xml",
142:                                "instances/validationInstances/Selbsttest_20b.
xml",
143:                                "instances/validationInstances/Selbsttest_20c.
xml"
144:                                };
145:
146:         String[] xmlPathesNOK = {"instances/validationInstances/Selbst
test_clockwiseNOK.xml",
147:                                "instances/validationInstances/Selbsttest_coun
terClockwiseNOK.xml"
148:                                };
149:
150:
151:         FilePersistence persistence = new FilePersistence();
152:
153:         //Act, Assert
154:         for (String xmlFile: xmlPathesOK) {
155:             IRoom room = null;
156:             try {
157:                 room = persistence.readInput(xmlFile);
158:             } catch (PersistenceException e) {
159:                 fail(e.getMessage());
160:             }
161:         }
162:
163:         for (String xmlFile: xmlPathesNOK) {
164:             IRoom room = null;
165:             try {
166:                 room = persistence.readInput(xmlFile);
167:                 fail("This xml file is not OK!" + xmlFile);
168:             } catch (PersistenceException e) {
169:
170:             }
171:         }
172:
173:     }
174:
175:     @Test
176:     public void testWriteOutput() {
177:
178:         //Arrange
179:         IPersistence persistence = new FilePersistence();
180:
181:         //Act
182:         try {
183:             persistence.writeOutput(room, "/Users/alex/Desktop/test
");
184:         } catch (PersistenceException e) {
185:             fail(e.getMessage());
186:         }
187:
188:         /*@Test
189:         public void testIsCounterClockWise() {
190:             Point p1 = new Point(0,0);
191:             Point p2 = new Point(1,0);
192:             Point p3 = new Point(1,1);
193:             Point p4 = new Point(0,1);
194:             List<Point> counterClockWise = new ArrayList<Point>();
195:             List<Point> clockWise = new ArrayList<Point>();
196:             counterClockWise.add(p1); counterClockWise.add(p2); counterClo
ckWise.add(p3); counterClockWise.add(p4);
197:             clockWise.add(p4); clockWise.add(p3); clockWise.add(p2); clock
Wise.add(p1);
198:             assertTrue(!FilePersistence.isCounterClockWise(clockWise, p2))
;
199:             assertTrue(FilePersistence.isCounterClockWise(counterClockWise
, p2));
200:         }*/
201:
202:
203:
204: }

```

./ProPra2020_workspace/Test_Component/src/fernuni/propra/file_processing/FileProcessingTests.java

Tue Apr 14 13:40:47 202

```
1: package fernuni.propra.file_processing;
2:
3: import org.junit.runner.RunWith;
4: import org.junit.runners.Suite;
5: import org.junit.runners.Suite.SuiteClasses;
6:
7: import fernuni.propra.internal_data_model.LineSegmentTest;
8: import fernuni.propra.internal_data_model.LineSegmentTestParameterized;
9: import fernuni.propra.internal_data_model.PointTest;
10:
11: @RunWith(Suite.class)
12: @SuiteClasses({ FilePersistenceTest.class, LineSegmentTest.class, LineSegmentT
estParameterized.class, PointTest.class })
13: public class FileProcessingTests {
14:
15: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.ArrayList;
6: import java.util.Iterator;
7: import java.util.LinkedList;
8: import java.util.List;
9:
10: import org.junit.Before;
11: import org.junit.Test;
12:
13: import fernuni.propra.internal_data_model.IRoom;
14: import fernuni.propra.internal_data_model.Lamp;
15: import fernuni.propra.internal_data_model.Point;
16: import fernuni.propra.internal_data_model.Room;
17: import fernuni.propra.internal_data_model.Wall;
18:
19: public class PositionOptimizerTest {
20:     private IRoom mockRoom, room, room2, roomStar, roomHufeisen;
21:     private Point p1, p2, p3, p4, p5, p6, p7, p8;
22:     private Point pc1, pc2, pc3, pc4, pc5, pc6, pc7, pc8, pc9, pc10, pc11, p
c12;
23:     private Point p31, p32, p33, p34, p35, p36, p37, p38;
24:     private Wall w1, w2, w3, w4;
25:     private LinkedList<Point> corners, corners2;
26:
27:     @Before
28:     public void setup() {
29:         p1 = new Point(0,0);
30:         p2 = new Point(1,0);
31:         p3 = new Point (1,1);
32:         p4 = new Point(0,1);
33:
34:         p5 = new Point(0.5, 1.0);
35:         p6 = new Point(0.5, 0.5);
36:         p7 = new Point(0, 0.5);
37:
38:
39:
40:
41:         corners= new LinkedList<Point>();
42:         corners.add(p1); corners.add(p2); corners.add(p3); corners.add
(p4);
43:
44:         corners2= new LinkedList<Point>();
45:         corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2
.add(p5);
46:         corners2.add(p6); corners2.add(p7);
47:
48:         room = new Room("test", null, corners);
49:         room2 = new Room("test", null, corners2);
50:
51:
52:
53:
54:         pc1 = new Point(1,-1);
55:         pc2 = new Point(2,-1);
56:         pc3 = new Point(2,1);
57:         pc4 = new Point(1,1);
58:         pc5 = new Point(1,2);
59:         pc6 = new Point(-1,2);
60:         pc7 = new Point(-1,1);
61:         pc8 = new Point(-2,1);
62:         pc9 = new Point(-2,-1);
63:         pc10 = new Point (-1,-1);
64:         pc11 = new Point (-1,-2);
65:         pc12 = new Point(1,-2);
66:         LinkedList<Point> cornersStar = new LinkedList<Point>();
67:         cornersStar.add(pc1); cornersStar.add(pc2); cornersStar.add(pc3)
; cornersStar.add(pc4); cornersStar.add(pc5);
68:         cornersStar.add(pc6); cornersStar.add(pc7); cornersStar.add(pc8)
; cornersStar.add(pc9); cornersStar.add(pc10);
69:         cornersStar.add(pc11); cornersStar.add(pc12);
70:
71:         roomStar = new Room("star", null, cornersStar);
72:
73:
74:         p31 = new Point(-2,0);
75:         p32 = new Point(2,0);
76:         p33 = new Point(2,2);
77:         p34 = new Point(1,2);
78:         p35 = new Point(1,1);
79:         p36 = new Point(-1,1);
80:         p37 = new Point(-1,2);
81:         p38 = new Point(-2,2);
82:         LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
83:         cornersHufeisen.add(p31); cornersHufeisen.add(p32); cornersHufeis
en.add(p33); cornersHufeisen.add(p34); cornersHufeisen.add(p35);
84:         cornersHufeisen.add(p36); cornersHufeisen.add(p37); cornersHufeis
en.add(p38);
85:         roomHufeisen = new Room("hufeisen", null, cornersHufeisen);
86:
87:     }
88:
89:     @Test
90:     public void testOptimizePositions() {
91:         //Arrange
92:         IPositionOptimizer positionOptimizer = AbstractAlgorithmFactor
y.getAlgorithmFactory().createPositionOptimizer();
93:         IPositionOptimizer positionOptimizer2 = AbstractAlgorithmFacto
ry.getAlgorithmFactory().createPositionOptimizer();
94:         ICandidateSearcher candidateSearcher = AbstractAlgorithmFactor
y.getAlgorithmFactory().createCandidateSearcher();
95:
96:         List<Lamp> taggedCandidates = null;
97:         try {
98:             taggedCandidates = candidateSearcher.searchCandidates (
room, null);
99:
100:             Lamp lamp = new Lamp(0.0,0.0);
101:             lamp.addTag(1);
102:             taggedCandidates.add(lamp);
103:         } catch (CandidateSearcherException | InterruptedException e)
{
104:             fail("Candidates Searcher failed!");
105:         }
106:
107:         List<Lamp> taggedCandidates2 = new LinkedList<Lamp>();
108:         Lamp lamp1 = new Lamp(0,0);
109:         lamp1.addTag(0);
110:         Lamp lamp2 = new Lamp(0,0);
111:         lamp2.addTag(1);
112:         Lamp lamp3 = new Lamp(0,0);
113:         lamp3.addTag(2);
114:         Lamp lamp4 = new Lamp(0,0);
115:         lamp4.addTag(3);
116:         Lamp lamp5 = new Lamp(0,0);

```

```

117:         lamp5.addTag(1); lamp5.addTag(2);
118:         Lamp lamp6 = new Lamp(0,0);
119:         lamp6.addTag(2); lamp6.addTag(3);
120:         Lamp lamp7 = new Lamp(0,0);
121:         lamp7.addTag(3); lamp7.addTag(0);
122:
123:         Lamp lamp8 = new Lamp(0,0);
124:         lamp8.addTag(1); lamp8.addTag(2); lamp8.addTag(3);
125:         Lamp lamp9 = new Lamp(0,0);
126:         lamp9.addTag(2); lamp9.addTag(3); lamp9.addTag(0);
127:         Lamp lamp10 = new Lamp(0,0);
128:         lamp10.addTag(3); lamp10.addTag(0); lamp10.addTag(1);
129:
130:         Lamp lamp11 = new Lamp(0,0);
131:         lamp11.addTag(0); lamp11.addTag(1); lamp11.addTag(2); lamp11.a
ddTag(3);
132:
133:         taggedCandidates2.add(lamp1); taggedCandidates2.add(lamp2); ta
ggedCandidates2.add(lamp3); taggedCandidates2.add(lamp4);
134:         taggedCandidates2.add(lamp5); taggedCandidates2.add(lamp6); ta
ggedCandidates2.add(lamp7); taggedCandidates2.add(lamp8);
135:         taggedCandidates2.add(lamp9); taggedCandidates2.add(lamp10); t
aggedCandidates2.add(lamp11);
136:
137:         //Act
138:         List<Lamp> optimizedLamps = new LinkedList<Lamp>();
139:         try {
140:             optimizedLamps = positionOptimizer.optimizePositions(
taggedCandidates, null);
141:         } catch (InterruptedException e) {
142:             // TODO Auto-generated catch block
143:             e.printStackTrace();
144:         }
145:         List<Lamp> optimizedLamps2 = new LinkedList<Lamp>();
146:         try {
147:             optimizedLamps2 = positionOptimizer2.optimizePositions
(taggedCandidates2, null);
148:         } catch (InterruptedException e) {
149:             // TODO Auto-generated catch block
150:             e.printStackTrace();
151:         }
152:
153:         //Assert
154:         for (int i = 0; i< optimizedLamps2.size()-1; i++) {
155:             assertFalse(optimizedLamps2.get(i).getOn());
156:         }
157:         assertTrue(optimizedLamps2.get(optimizedLamps2.size()-1).getOn
());
158:
159:         assertTrue(optimizedLamps.get(0).getOn());
160:         assertFalse(optimizedLamps.get(1).getOn());
161:     }
162:
163: }

```

```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import org.junit.Before;
6: import org.junit.Test;
7:
8: import fernuni.propra.internal_data_model.Point;
9: import fernuni.propra.internal_data_model.Wall;
10:
11: public class WallContainerEastTest {
12:     Point p1,p2,p3,p4;
13:     Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;
14:
15:     @Before
16:     public void setUp() {
17:         //Arrange
18:         p1 = new Point(0,0);
19:         p2 = new Point(1,0);
20:         p3 = new Point(1,1);
21:         p4 = new Point(0,1);
22:         w1 = new Wall(p1,p2,0);
23:         w2 = new Wall(p2,p3,0);
24:         w3 = new Wall(p3,p4,0);
25:         w4 = new Wall(p4,p1,0);
26:
27:         w5 = new Wall(p2,p1,0);
28:         w6 = new Wall(p3, p2,0);
29:         w7 = new Wall(p4, p3,0);
30:         w8 = new Wall(p1,p4,0);
31:
32:         w9 = new Wall(p1,p1,0);
33:     }
34:
35:
36:     @Test
37:     public void testAdd() {
38:         //Arrange
39:         WallContainerEast wallContainerEast = new WallContainerEast();
40:
41:         //Act
42:         boolean test1 = false;
43:         try {
44:             wallContainerEast.add(w1);
45:             fail("WallContainerException expected");
46:         } catch (WallContainerException ex) {
47:             test1 = true;
48:         }
49:
50:         boolean test2 = false;
51:         try {
52:             wallContainerEast.add(w2);
53:             test2 = true;
54:         } catch (WallContainerException ex) {
55:             fail();
56:         }
57:
58:         //Assert
59:         assertTrue(test1);
60:         assertTrue(test2);
61:     }
62:
63:     @Test
64:     public void testGetNearestEastWall() {
65:
66:         //Arrange
67:         WallContainerEast wallContainerEast = new WallContainerEast();
68:
69:         try {
70:             wallContainerEast.add(w2);
71:         } catch (WallContainerException e) {
72:             // TODO Auto-generated catch block
73:             e.printStackTrace();
74:         }
75:
76:         try {
77:             wallContainerEast.add(w8);
78:         } catch (WallContainerException e) {
79:             // TODO Auto-generated catch block
80:             e.printStackTrace();
81:         }
82:
83:         //Act
84:         Wall w10 = null;
85:         try {
86:             w10 = wallContainerEast.getNearestWall(-1, 1, 0.5);
87:         } catch (WallContainerException e) {
88:             fail(e.getMessage());
89:         }
90:
91:         Wall w11 = null;
92:         try {
93:             w11 = wallContainerEast.getNearestWall(-1, 1, 0.0);
94:         } catch (WallContainerException e) {
95:             fail(e.getMessage());
96:         }
97:
98:         Wall w12 = null;
99:         try {
100:             w12 = wallContainerEast.getNearestWall(-1, 1, -0.001);
101:         } catch (WallContainerException e) {
102:             fail(e.getMessage());
103:         }
104:
105:         Wall w13 = null;
106:         try {
107:             w13 = wallContainerEast.getNearestWall(-1, -0.5, -0.00
108: 1);
109:         } catch (WallContainerException e) {
110:             fail(e.getMessage());
111:         }
112:
113:         //Assert
114:         assertTrue(w10.getP1().isEqual(w2.getP1()) && w10.getP2().isEq
115: ual(w2.getP2()));
116:         assertFalse(w11.getP1().isEqual(w2.getP1()) && w11.getP2().isE
117: qual(w2.getP2()));
118:         assertTrue(w12.getP1().isEqual(w8.getP1()) && w12.getP2().isEq
119: ual(w8.getP2()));
120:         assertTrue(w13 == null);

```



```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import org.junit.Before;
6: import org.junit.Test;
7:
8: import fernuni.propra.internal_data_model.Point;
9: import fernuni.propra.internal_data_model.Wall;
10:
11: public class WallContainerWestTest {
12:     Point p1,p2,p3,p4;
13:     Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;
14:
15:     @Before
16:     public void setUp() {
17:         //Arrange
18:         p1 = new Point(0,0);
19:         p2 = new Point(1,0);
20:         p3 = new Point(1,1);
21:         p4 = new Point(0,1);
22:         w1 = new Wall(p1,p2,0);
23:         w2 = new Wall(p2,p3,0);
24:         w3 = new Wall(p3,p4,0);
25:         w4 = new Wall(p4,p1,0);
26:
27:         w5 = new Wall(p2,p1,0);
28:         w6 = new Wall(p3, p2,0);
29:         w7 = new Wall(p4, p3,0);
30:         w8 = new Wall(p1,p4,0);
31:
32:         w9 = new Wall(p1,p1,0);
33:     }
34:
35:
36:     @Test
37:     public void testAdd() {
38:         //Arrange
39:         WallContainerWest wallContainerWest = new WallContainerWest();
40:
41:         //Act
42:         boolean test1 = false;
43:         try {
44:             wallContainerWest.add(w1);
45:             fail("WallContainerException expected");
46:         } catch (WallContainerException ex) {
47:             test1 = true;
48:         }
49:
50:         boolean test2 = false;
51:         try {
52:             wallContainerWest.add(w4);
53:             test2 = true;
54:         } catch (WallContainerException ex) {
55:             fail();
56:         }
57:
58:         //Assert
59:         assertTrue(test1);
60:         assertTrue(test2);
61:     }
62:
63:     @Test
64:     public void testGetNearestWestWall() {
65:
66:         //Arrange
67:         WallContainerWest wallContainerWest = new WallContainerWest();
68:
69:         try {
70:             wallContainerWest.add(w4);
71:         } catch (WallContainerException e) {
72:             // TODO Auto-generated catch block
73:             e.printStackTrace();
74:         }
75:
76:         try {
77:             wallContainerWest.add(w6);
78:         } catch (WallContainerException e) {
79:             // TODO Auto-generated catch block
80:             e.printStackTrace();
81:         }
82:
83:         //Act
84:         Wall w10 = null;
85:         try {
86:             w10 = wallContainerWest.getNearestWall(-1, 1, 0.5);
87:         } catch (WallContainerException e) {
88:             fail(e.getMessage());
89:         }
90:         boolean test1 = w10.getP1().isEqual(w4.getP1()) && w10.getP2().isEqual(w4.getP2());
91:
92:         Wall w11 = null;
93:         try {
94:             w11 = wallContainerWest.getNearestWall(-1, 1, 1.0);
95:         } catch (WallContainerException e) {
96:             fail(e.getMessage());
97:         }
98:         boolean test2 = w11.getP1().isEqual(w4.getP1()) && w11.getP2().isEqual(w4.getP2());
99:
100:         Wall w12 = null;
101:         try {
102:             w12 = wallContainerWest.getNearestWall(-1, 1, 1.001);
103:         } catch (WallContainerException e) {
104:             fail(e.getMessage());
105:         }
106:         boolean test3 = w12.getP1().isEqual(w6.getP1()) && w12.getP2().isEqual(w6.getP2());
107:
108:         Wall w13 = null;
109:         try {
110:             w13 = wallContainerWest.getNearestWall(-1, -0.5, -0.001);
111:         } catch (WallContainerException e) {
112:             fail(e.getMessage());
113:         }
114:         boolean test4 = w13 == null;
115:
116:         //Assert
117:         assertTrue(test1);
118:         assertTrue(!test2);
119:         assertTrue(test3);
120:         assertTrue(test4);
121:
122:     }
123:
124: }

```



```

1: package fernuni.propra.algorithm.util;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.ArrayList;
6: import java.util.HashSet;
7: import java.util.List;
8:
9: import org.junit.Before;
10: import org.junit.Test;
11:
12: import fernuni.propra.internal_data_model.Point;
13:
14: public class RectangleWithTagTest {
15:     private Point p1,p2,p3,p4;
16:     private RectangleWithTag rec1;
17:
18:     @Before
19:     public void setUp() throws Exception {
20:
21:         //Arrange
22:         p1 = new Point(0,0);
23:         p2 = new Point(1,0);
24:         p3 = new Point(1,1);
25:         p4 = new Point(0,1);
26:         List<Integer> initTags = new ArrayList<Integer>();
27:         initTags.add(1);
28:         rec1 = new RectangleWithTag(p1, p3, initTags);
29:     }
30:
31:     @Test
32:     public void testContainsTag() {
33:         //Act
34:         boolean test1 = rec1.containsTag(1);
35:         boolean test2 = !rec1.containsTag(2);
36:
37:         //Assert
38:         assertTrue(test1);
39:         assertTrue(test2);
40:     }
41:
42:     @Test
43:     public void testAddTag() {
44:         fail("Not yet implemented");
45:     }
46:
47:     @Test
48:     public void testHashSet() {
49:         //Arrange
50:         List<Integer> initTags2 = new ArrayList<Integer>(); initTags2.
add(2);
51:         RectangleWithTag newRectangleWithTag = new RectangleWithTag(p1
, p3, initTags2 );
52:         HashSet<RectangleWithTag> rectanglesWithTags = new HashSet<Rec
tangleWithTag>();
53:         rectanglesWithTags.add(rec1);
54:
55:
56:         //Act
57:         boolean test1 = rectanglesWithTags.contains(newRectangleWithTa
g);
58:         boolean test2 = rec1.equals(newRectangleWithTag);
59:
60:         //Assert
61:         assertFalse(test1);
62:         assertFalse(test2);
63:     }
64:
65: }

```



```

1: package fernuni.propra.algorithm.util;
2:
3: import static org.junit.Assert.*;
4:
5: import org.junit.Before;
6: import org.junit.Test;
7:
8: import fernuni.propra.internal_data_model.Point;
9:
10: public class RectangleTest {
11:     Point p1,p2,p3,p4;
12:     Rectangle rec;
13:
14:     @Before
15:     public void setUp() {
16:         //Arrange
17:         p1 = new Point(0,0);
18:         p2 = new Point(1,0);
19:         p3 = new Point(1,1);
20:         p4 = new Point(0,1);
21:         rec = new Rectangle(p1, p3);
22:     }
23:
24:     @Test
25:     public void testOverlap() {
26:         //Arrange
27:         Point p5 = new Point(0.5,0.5);
28:         Point p6 = new Point(1.5,0.5);
29:         Point p7 = new Point(1.5,1.5);
30:         Point p8 = new Point(0.5,1.5);
31:         Point p9 = new Point(0.5,2.0);
32:         Point p10 = new Point(-1, -1);
33:
34:         Rectangle rec2 = new Rectangle(p5, p7);
35:         Rectangle rec3 = new Rectangle(p5, p3);
36:         Rectangle rec5 = new Rectangle(p3, p7);
37:         Rectangle rec8 = new Rectangle(p10, p9);
38:         Rectangle rec9 = new Rectangle(p1, new Point(0.5, 1.0));
39:         Rectangle rec11 = new Rectangle(p2, new Point(2*p2.getX(), 1.0
));
40:
41:         //Act
42:         Rectangle rec4 = rec.overlap(rec2);
43:         Rectangle rec6 = rec.overlap(rec5);
44:         Rectangle rec7 = rec.overlap(rec);
45:         Rectangle rec10 = rec.overlap(rec8);
46:         Rectangle rec12 = rec.overlap(rec11);
47:
48:
49:         assertTrue(rec3.equals(rec4));
50:         assertTrue(rec6 == null);
51:         assertTrue(rec7.equals(rec));
52:         assertTrue(rec9.equals(rec10));
53:         assertTrue(rec12 == null);
54:
55:     }
56:
57:     @Test
58:     public void testGetCenter() {
59:
60:         //Act
61:         Point result = rec.getCenter();
62:
63:         //Assert
64:         assertTrue(result.isEqual(new Point(0.5, 0.5)));
65:
66:     }
67:
68:     @Test
69:     public void testIsEqual() {
70:         //Arrange
71:         Point p5 = new Point(2,0);
72:         Point p6 = new Point(2,2);
73:         Rectangle rec2 = new Rectangle(p1, p6);
74:
75:         //Act
76:         boolean test1 = rec.equals(rec);
77:         boolean test2 = rec.equals(rec2);
78:
79:         //Assert
80:         assertTrue(test1);
81:         assertFalse(test2);
82:
83:     }
84:
85:     @Test
86:     public void testIsCounterClockWise() {
87:         //Arrange
88:         Rectangle rec2 = new Rectangle(p1, p1);
89:
90:         //Act
91:         boolean test1 = rec2.isCounterClockWise();
92:         boolean test2 = rec.isCounterClockWise();
93:
94:         //Assert
95:         assertFalse(test1);
96:         assertTrue(test2);
97:     }
98:
99: }

```



```
1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import org.junit.Before;
6: import org.junit.Test;
7:
8: public class RectangleWithTagTest {
9:
10:     @Before
11:     public void setUp() throws Exception {
12:     }
13:
14:     @Test
15:     public void testContainsTag() {
16:         fail("Not yet implemented");
17:     }
18:
19:     @Test
20:     public void testAddTag() {
21:         fail("Not yet implemented");
22:     }
23:
24: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.Iterator;
6: import java.util.LinkedList;
7: import java.util.List;
8:
9: import org.junit.Before;
10: import org.junit.Test;
11:
12: import fernuni.propra.internal_data_model.IRoom;
13: import fernuni.propra.internal_data_model.Lamp;
14: import fernuni.propra.internal_data_model.Point;
15: import fernuni.propra.internal_data_model.Room;
16: import fernuni.propra.internal_data_model.Wall;
17:
18: public class UserSolveAASTest {
19:
20:     private IRoom room, room2, roomStar, roomHufeisen;
21:     private Point p1, p2, p3, p4, p5, p6, p7, p8;
22:     private Point pc1, pc2, pc3, pc4, pc5, pc6, pc7, pc8, pc9, pc10, pc11, p
c12;
23:     private Point p31, p32, p33, p34, p35, p36, p37, p38;
24:     private Wall w1, w2, w3, w4;
25:     private LinkedList<Point> corners, corners2;
26:
27:     @Before
28:     public void setup() {
29:         p1 = new Point(0,0);
30:         p2 = new Point(1,0);
31:         p3 = new Point(1,1);
32:         p4 = new Point(0,1);
33:
34:         p5 = new Point(0.5, 1.0);
35:         p6 = new Point(0.5, 0.5);
36:         p7 = new Point(0, 0.5);
37:
38:
39:
40:         corners= new LinkedList<Point>();
41:         corners.add(p1); corners.add(p2); corners.add(p3); corners.add
(p4);
42:
43:         corners2= new LinkedList<Point>();
44:         corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2
.add(p5);
45:         corners2.add(p6); corners2.add(p7);
46:
47:         room = new Room("test", null, corners);
48:         room2 = new Room("test", null, corners2);
49:
50:
51:         pc1 = new Point(1,-1);
52:         pc2 = new Point(2,-1);
53:         pc3 = new Point(2,1);
54:         pc4 = new Point(1,1);
55:         pc5 = new Point(1,2);
56:         pc6 = new Point(-1,2);
57:         pc7 = new Point(-1,1);
58:         pc8 = new Point(-2,1);
59:         pc9 = new Point(-2,-1);
60:         pc10 = new Point(-1,-1);
61:         pc11 = new Point(-1,-2);
62:
63:         pc12 = new Point(1,-2);
64:         LinkedList<Point> cornersStar = new LinkedList<Point>();
65:         cornersStar.add(pc1); cornersStar.add(pc2); cornersStar.add(pc3)
; cornersStar.add(pc4); cornersStar.add(pc5);
66:         cornersStar.add(pc6); cornersStar.add(pc7); cornersStar.add(pc8)
; cornersStar.add(pc9); cornersStar.add(pc10);
67:         cornersStar.add(pc11); cornersStar.add(pc12);
68:
69:         roomStar = new Room("star", null, cornersStar);
70:
71:         p31 = new Point(-2,0);
72:         p32 = new Point(2,0);
73:         p33 = new Point(2,2);
74:         p34 = new Point(1,2);
75:         p35 = new Point(1,1);
76:         p36 = new Point(-1,1);
77:         p37 = new Point(-1,2);
78:         p38 = new Point(-2,2);
79:         LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
80:         cornersHufeisen.add(p31); cornersHufeisen.add(p32); cornersHufe
isen.add(p33); cornersHufeisen.add(p34); cornersHufeisen.add(p35);
81:         cornersHufeisen.add(p36); cornersHufeisen.add(p37); cornersHufe
isen.add(p38);
82:         roomHufeisen = new Room("hufeisen", null, cornersHufeisen);
83:
84:     }
85:
86:     @Test
87:     public void testSolve() {
88:         //Arrange
89:         UserSolveAAS userSolve = new UserSolveAAS();
90:
91:         //Act
92:         try {
93:             userSolve.solve(room, 100);
94:         } catch (UserSolveAASException e) {
95:             fail();
96:         }
97:
98:         //Assert
99:         assertTrue(room.getLamps().hasNext());
100:     }
101:
102: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import org.junit.Before;
6: import org.junit.Test;
7:
8: import fernuni.propra.internal_data_model.Point;
9: import fernuni.propra.internal_data_model.Wall;
10:
11: public class WallContainerSouthTest {
12:     Point p1,p2,p3,p4;
13:     Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;
14:
15:     @Before
16:     public void setUp() {
17:         //Arrange
18:         p1 = new Point(0,0);
19:         p2 = new Point(1,0);
20:         p3 = new Point(1,1);
21:         p4 = new Point(0,1);
22:         w1 = new Wall(p1,p2,0);
23:         w2 = new Wall(p2,p3,0);
24:         w3 = new Wall(p3,p4,0);
25:         w4 = new Wall(p4,p1,0);
26:
27:         w5 = new Wall(p2,p1,0);
28:         w6 = new Wall(p3, p2,0);
29:         w7 = new Wall(p4, p3,0);
30:         w8 = new Wall(p1,p4,0);
31:
32:         w9 = new Wall(p1,p1,0);
33:     }
34:
35:
36:     @Test
37:     public void testAdd() {
38:         //Arrange
39:         WallContainerSouth wallContainerSouth = new WallContainerSouth
40:     );
41:
42:         //Act
43:         boolean test1 = false;
44:         try {
45:             wallContainerSouth.add(w2);
46:             fail("WallContainerException expected");
47:         } catch (WallContainerException ex) {
48:             test1 = true;
49:         }
50:
51:         boolean test2 = false;
52:         try {
53:             wallContainerSouth.add(w1);
54:             test2 = true;
55:         } catch (WallContainerException ex) {
56:             fail();
57:         }
58:
59:         //Assert
60:         assertTrue(test1);
61:         assertTrue(test2);
62:     }
63:
64:     @Test
65:     public void testGetNearestWestWall() {
66:         //Arrange
67:         WallContainerSouth wallContainerSouth = new WallContainerSouth
68:     );
69:
70:         try {
71:             wallContainerSouth.add(w1);
72:         } catch (WallContainerException e) {
73:             // TODO Auto-generated catch block
74:             e.printStackTrace();
75:         }
76:
77:         try {
78:             wallContainerSouth.add(w7);
79:         } catch (WallContainerException e) {
80:             // TODO Auto-generated catch block
81:             e.printStackTrace();
82:         }
83:
84:         //Act
85:         Wall w10 = null;
86:         try {
87:             w10 = wallContainerSouth.getNearestWall(-1, 1, 0.5);
88:         } catch (WallContainerException e) {
89:             fail(e.getMessage());
90:         }
91:
92:         boolean test1 = w10.getP1().isEqual(w1.getP1()) && w10.getP2()
93:         .isEqual(w1.getP2());
94:
95:         Wall w11 = null;
96:         try {
97:             w11 = wallContainerSouth.getNearestWall(-1, 1, 1.0);
98:         } catch (WallContainerException e) {
99:             fail(e.getMessage());
100:         }
101:
102:         boolean test2 = w11.getP1().isEqual(w7.getP1()) && w11.getP2()
103:         .isEqual(w7.getP2());
104:
105:         Wall w12 = null;
106:         try {
107:             w12 = wallContainerSouth.getNearestWall(-1, 1, 1.001);
108:         } catch (WallContainerException e) {
109:             fail(e.getMessage());
110:         }
111:
112:         boolean test3 = w12.getP1().isEqual(w7.getP1()) && w12.getP2()
113:         .isEqual(w7.getP2());
114:
115:         Wall w13 = null;
116:         try {
117:             w13 = wallContainerSouth.getNearestWall(-1, -0.5, -0.0
118:         01);
119:         } catch (WallContainerException e) {
120:             fail(e.getMessage());
121:         }
122:
123:         boolean test4 = w13 == null;
124:
125:         //Assert
126:         assertTrue(test1);
127:         assertTrue(test2);
128:         assertTrue(test3);
129:         assertTrue(test4);
130:     }
131:
132: }

```

`./ProPra2020_workspace/Test_Component/src/fernuni/propra/algorithm/WallContainerSouthTest.java`

Wed Jun 10 12:05:13 2020

```
123:  
124: }
```



```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import java.awt.Color;
6: import java.util.ArrayList;
7: import java.util.HashSet;
8: import java.util.Iterator;
9: import java.util.LinkedList;
10: import java.util.List;
11:
12: import org.junit.Before;
13: import org.junit.BeforeClass;
14: import org.junit.Test;
15:
16: import fernuni.propra.algorithm.runtime_information.RuntimeInformation;
17: import fernuni.propra.algorithm.util.Rectangle;
18: import fernuni.propra.algorithm.util.RectangleWithTag;
19: import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
20: import fernuni.propra.file_processing.UserReadInputWriteOutputException;
21: import fernuni.propra.internal_data_model.IRoom;
22: import fernuni.propra.internal_data_model.Lamp;
23: import fernuni.propra.internal_data_model.Point;
24: import fernuni.propra.internal_data_model.Room;
25: import fernuni.propra.internal_data_model.Wall;
26: import fernuni.propra.user_interface.RoomFrame;
27: import fernuni.propra.user_interface.RoomPanel;
28:
29: public class OriginalPartialRectanglesFinderTest {
30:     private IRoom mockRoom, room, room2, roomStar, roomHufeisen;
31:     private Point p1, p2, p3, p4, p5, p6, p7, p8;
32:     private Point pc1, pc2, pc3, pc4, pc5, pc6, pc7, pc8, pc9, pc10, pc11, p
c12;
33:     private Point p31, p32, p33, p34, p35, p36, p37, p38;
34:     private Wall w1, w2, w3, w4;
35:     private LinkedList<Point> corners, corners2;
36:     private static List<IRoom> rooms;
37:
38:     @BeforeClass
39:     public static void setupBC() {
40:
41:         String[] xmlPathesOK = {"instances/validationInstances/Selbstt
est_clockwise.xml", //0
42:                                "instances/validationInstances/Selbsttest_coun
terClockwise.xml", //1
43:                                "instances/validationInstances/Selbsttest_100a
_incomplete.xml", // 2
44:                                "instances/validationInstances/Selbsttest_100a
_incomplete.xml", //3
45:                                "instances/validationInstances/Selbsttest_100a
_solved.xml", // 4
46:                                "instances/validationInstances/Selbsttest_100a
.xml", // 5
47:                                "instances/validationInstances/Selbsttest_100b
.xml", // 6
48:                                "instances/validationInstances/Selbsttest_20a_
incomplete.xml", // 7
49:                                "instances/validationInstances/Selbsttest_20a_
solved.xml", // 8
50:                                "instances/validationInstances/Selbsttest_20a.
xml", // 9
51:                                "instances/validationInstances/Selbsttest_20b.
xml", // 10
52:                                "instances/validationInstances/Selbsttest_20c.
xml", // 11
53:                                "instances/validationInstances/Zufallsraum_144
_solved.xml" // 12
54:                                };
55:
56:         rooms = new ArrayList<IRoom>();
57:
58:         for(String xmlPath : xmlPathesOK) {
59:             UserReadInputWriteOutputAAS readAAS = new UserReadInpu
tWriteOutputAAS(xmlPath);
60:             try {
61:                 rooms.add(readAAS.readInput());
62:             } catch (UserReadInputWriteOutputException e) {
63:                 // TODO Auto-generated catch block
64:                 e.printStackTrace();
65:             }
66:         }
67:     }
68:
69:     @Before
70:     public void setUp() throws Exception {
71:         p1 = new Point(0,0);
72:         p2 = new Point(1,0);
73:         p3 = new Point(1,1);
74:         p4 = new Point(0,1);
75:
76:         p5 = new Point(0.5, 1.0);
77:         p6 = new Point(0.5, 0.5);
78:         p7 = new Point(0, 0.5);
79:
80:
81:
82:
83:         corners= new LinkedList<Point>();
84:         corners.add(p1); corners.add(p2); corners.add(p3); corners.add
(p4);
85:
86:         corners2= new LinkedList<Point>();
87:         corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2
.add(p5);
88:
89:         corners2.add(p6); corners2.add(p7);
90:
91:         room = new Room("test", null, corners);
92:         room2 = new Room("test", null, corners2);
93:
94:         pc1 = new Point(1,-1);
95:         pc2 = new Point(2,-1);
96:         pc3 = new Point(2,1);
97:         pc4 = new Point(1,1);
98:         pc5 = new Point(1,2);
99:         pc6 = new Point(-1,2);
100:         pc7 = new Point(-1,1);
101:         pc8 = new Point(-2,1);
102:         pc9 = new Point(-2,-1);
103:         pc10 = new Point(-1,-1);
104:         pc11 = new Point(-1,-2);
105:         pc12 = new Point(1,-2);
106:         LinkedList<Point> cornersStar = new LinkedList<Point>();
107:         cornersStar.add(pc1); cornersStar.add(pc2); cornersStar.add(pc3)
; cornersStar.add(pc4); cornersStar.add(pc5);
108:         cornersStar.add(pc6); cornersStar.add(pc7); cornersStar.add(pc8)
; cornersStar.add(pc9); cornersStar.add(pc10);
109:         cornersStar.add(pc11); cornersStar.add(pc12);

```

```

110:
111:         roomStar = new Room("star", null, cornersStar);
112:
113:
114:         p31 = new Point(-2,0);
115:         p32 = new Point(2,0);
116:         p33 = new Point(2,2);
117:         p34 = new Point(1,2);
118:         p35 = new Point(1,1);
119:         p36 = new Point(-1,1);
120:         p37 = new Point(-1,2);
121:         p38 = new Point(-2,2);
122:         LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
123:         cornersHufeisen.add(p31);cornersHufeisen.add(p32);cornersHufeisen.add(p33);cornersHufeisen.add(p34);cornersHufeisen.add(p35);
124:         cornersHufeisen.add(p36);cornersHufeisen.add(p37);cornersHufeisen.add(p38);
125:         roomHufeisen = new Room("hufeisen", null, cornersHufeisen);
126:
127:     }
128:
129:     @Test
130:     public void testFindOriginalPartialRectangles() {
131:         //Arrange
132:         OriginalPartialRectanglesFinder rectanglesFinder3 = new OriginalPartialRectanglesFinder();
133:         CandidateSearcher candidateSearcher = (CandidateSearcher) AbstractAlgorithmFactory.getAlgorithmFactory().createCandidateSearcher();
134:
135:         //Act
136:         IRoom testRoom = rooms.get(9);
137:         //IRoom testRoom = roomStar;
138:         ArrayList<RectangleWithTag> rectanglesWithTag = new ArrayList<RectangleWithTag>();
139:         try {
140:             rectanglesWithTag = rectanglesFinder3.findOriginalPartialRectangles(testRoom, null);
141:         } catch (OriginalPartialRectanglesFinderException e) {
142:             // TODO Auto-generated catch block
143:             e.printStackTrace();
144:         }
145:         RoomPanel roomPanel = new RoomPanel(testRoom);
146:         Color[] colors = {Color.blue, Color.red, Color.green, Color.yellow};
147:
148:
149:
150:
151:         IRoom testRoom2 = rooms.get(12);
152:         //Act
153:
154:         //IRoom testRoom = roomStar;
155:         ArrayList<RectangleWithTag> rectanglesWithTag2 = new ArrayList<RectangleWithTag>();
156:         try {
157:             rectanglesWithTag2 = (new OriginalPartialRectanglesFinder()).findOriginalPartialRectangles(testRoom2, null);
158:         } catch (OriginalPartialRectanglesFinderException e) {
159:             // TODO Auto-generated catch block
160:             e.printStackTrace();
161:         }
162:         RoomPanel roomPanel2 = new RoomPanel(testRoom2);
163:         Color[] colors2 = {Color.blue, Color.red, Color.green, Color.yellow};
164:
165:         List<RectangleWithTag> rectanglesWithTag3 = new ArrayList<RectangleWithTag>();
166:         try {
167:             rectanglesWithTag3 = (new CandidateSearcher()).reduceRectangles(rectanglesWithTag2);
168:         } catch (InterruptedException e) {
169:             // TODO Auto-generated catch block
170:             e.printStackTrace();
171:         }
172:
173:
174:         RoomFrame roomFrame = new RoomFrame(roomPanel2);
175:
176:         for (int i = 0; i<rectanglesWithTag3.size(); i++) {
177:             RectangleWithTag rec = rectanglesWithTag3.get(i);
178:             double width = rec.getP2().getX() - rec.getP1().getX();
179:             double height = rec.getP3().getY() - rec.getP1().getY();
180:             roomPanel2.addRectangle(String.valueOf(i), colors[i % 3], rec.getP1().getX(), rec.getP1().getY(), width, height);
181:             roomPanel2.repaint();
182:             //roomPanel2.removeLastRectangle();
183:         }
184:
185:
186:         try {
187:             Thread.currentThread().sleep(100000);
188:         } catch (InterruptedException e) {
189:             // TODO: handle exception
190:         }
191:
192:         fail("not yet implemented");
193:     }
194:
195:     @Test
196:     public void testGetAllTags() {
197:         // Arrange
198:         OriginalPartialRectanglesFinder rectanglesFinder = new OriginalPartialRectanglesFinder();
199:         HashSet<Integer> refSet = new HashSet<Integer>();
200:         refSet.add(0); refSet.add(1); refSet.add(2); refSet.add(3);
201:
202:         //Act
203:         try {
204:             rectanglesFinder.sortWallsToContainers(room);
205:             rectanglesFinder.constructOriginalPartialRectangles();
206:         } catch (WallContainerException e) {
207:             fail(e.getMessage());
208:         } catch (OriginalPartialRectanglesFinderException e) {
209:             fail(e.getMessage());
210:         }
211:
212:         //Assert
213:         assertTrue(rectanglesFinder.getAllTags().containsAll(refSet));
214:     }
215:
216:     @Test
217:     public void testSortWallsToContainers() {
218:         //Arrange
219:         OriginalPartialRectanglesFinder originalRectanglesFinder = new OriginalPartialRectanglesFinder();
220:         //Act
221:         try {
222:             originalRectanglesFinder.sortWallsToContainers(room);

```

```

ntainers(room);
223:                } catch (OriginalPartialRectanglesFinderExcept
ion e) {
224:                    fail(e.getMessage());
225:                }
226:                //Assert
227:                Iterator<Wall> east = originalRectanglesFinder
228:                .eastIterator();
229:                Iterator<Wall> north = originalRectanglesFinde
r.northIterator();
230:                Iterator<Wall> west = originalRectanglesFinder
231:                .westIterator();
232:                Iterator<Wall> south = originalRectanglesFinde
r.southIterator();
233:                //east
234:                boolean test11 = east.hasNext();
235:                Wall wallEast = east.next();
236:                boolean test12 = !east.hasNext();
237:                boolean test13 = wallEast.getP1().isEqual(w2.g
etP1()) && wallEast.getP2().isEqual(w2.getP2());
238:                boolean eastBool = test11 && test12 && test13;
239:                //north
240:                boolean test21 = north.hasNext();
241:                Wall wallNorth = north.next();
242:                boolean test22 = !north.hasNext();
243:                boolean test23 = wallNorth.getP1().isEqual(w3.
getP1()) && wallNorth.getP2().isEqual(w3.getP2());
244:                boolean northBool = test21 && test22 && test23
;
245:                //west
246:                boolean test31 = west.hasNext();
247:                Wall wallWest = west.next();
248:                boolean test32 = !west.hasNext();
249:                boolean test33 = wallWest.getP1().isEqual(w4.g
etP1()) && wallWest.getP2().isEqual(w4.getP2());
250:                boolean westBool = test31 && test32 && test33;
251:                //south
252:                boolean test41 = south.hasNext();
253:                Wall wallSouth = south.next();
254:                boolean test42 = !south.hasNext();
255:                boolean test43 = wallSouth.getP1().isEqual(w1.
getP1()) && wallSouth.getP2().isEqual(w1.getP2());
256:                boolean southBool = test41 && test42 && test43
;
257:                assertTrue(eastBool && northBool && westBool &
& southBool);
258:            }
259:            @Test
260:            public void testConstructOriginalPartialRectangles() {
261:                // Arrange
262:                OriginalPartialRectanglesFinder rectanglesFind
er = new OriginalPartialRectanglesFinder();
263:                OriginalPartialRectanglesFinder rectanglesFind
er2 = new OriginalPartialRectanglesFinder();
264:                //Act
265:                try {
266:                    rectanglesFinder.sortWallsToContainers
267:                    (room);
268:                    rectanglesFinder.constructOriginalPart
ialRectangles();
269:                } catch (WallContainerException e) {
270:                    fail(e.getMessage());
271:                } catch (OriginalPartialRectanglesFinderExcept
ion e) {
272:                    fail(e.getMessage());
273:                }
274:                //2nd room
275:                try {
276:                    rectanglesFinder2.sortWallsToContainer
277:                    s(room2);
278:                    rectanglesFinder2.constructOriginalPar
tialRectangles();
279:                } catch (WallContainerException e) {
280:                    fail(e.getMessage());
281:                } catch (OriginalPartialRectanglesFinderExcept
ion e) {
282:                    fail(e.getMessage());
283:                }
284:                //Assert
285:                Iterator<RectangleWithTag> rectIterator = rect
anglesFinder.iteratorOriginalRectangles();
286:                RectangleWithTag rec1 = rectIterator.next();
287:                RectangleWithTag rec2 = rectIterator.next();
288:                RectangleWithTag rec3 = rectIterator.next();
289:                RectangleWithTag rec4 = rectIterator.next();
290:                Rectangle ref = new Rectangle(p1, p3);
291:                boolean test1 = !rectIterator.hasNext();
292:                boolean test2 = rec1.equals(new RectangleWithT
ag(ref, 0));
293:                boolean test3 = rec2.equals(new RectangleWithT
ag(ref, 1));
294:                boolean test4 = rec3.equals(new RectangleWithT
ag(ref, 2));
295:                boolean test5 = rec4.equals(new RectangleWithT
ag(ref, 3));
296:                assertTrue(test1 && test2 && test3 && test4 &&
test5);
297:                Iterator<RectangleWithTag> rectIterator2 = rec
anglesFinder2.iteratorOriginalRectangles();
298:                RectangleWithTag rec2_1 = rectIterator2.next();
299:                RectangleWithTag rec2_2 = rectIterator2.next();
300:                RectangleWithTag rec2_3 = rectIterator2.next();
301:                RectangleWithTag rec2_4 = rectIterator2.next();
302:                RectangleWithTag rec2_5 = rectIterator2.next();
303:                RectangleWithTag rec2_6 = rectIterator2.next();
304:                Rectangle ref2 = new Rectangle(p1, new Point(1

```

```
,0.5));
317:           Rectangle ref3 = new Rectangle(new Point(0.5,0
), p3);
318:
319:           boolean test7 = !rectIterator2.hasNext();
320:           boolean test8 = rec2_1.equals(new RectangleWit
hTag(ref2, 0));
321:           boolean test9 = rec2_2.equals(new RectangleWit
hTag(ref3, 1));
322:           boolean test10 = rec2_3.equals(new RectangleWi
thTag(ref3, 2));
323:           boolean test11 = rec2_4.equals(new RectangleWi
thTag(ref3, 3));
324:           boolean test12 = rec2_5.equals(new RectangleWi
thTag(ref2, 4));
325:           boolean test13 = rec2_6.equals(new RectangleWi
thTag(ref2, 5));
326:
327:
328:           assertTrue(test7 && test8 && test9 && test10 &
& test11 && test12 && test13);
329:       }
330:
331: }
```

```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.HashSet;
6: import java.util.Iterator;
7: import java.util.LinkedList;
8: import java.util.List;
9:
10: import org.junit.Before;
11: import org.junit.Test;
12:
13: import fernuni.propra.internal_data_model.IRoom;
14: import fernuni.propra.internal_data_model.Lamp;
15: import fernuni.propra.internal_data_model.Point;
16: import fernuni.propra.internal_data_model.Room;
17: import fernuni.propra.internal_data_model.Wall;
18:
19: public class IlluminationTesterTest {
20:     private IRoom mockRoom, room, room2, roomStar, roomHufeisen;
21:     private Point p1, p2, p3, p4, p5, p6, p7, p8;
22:     private Point pc1, pc2, pc3, pc4, pc5, pc6, pc7, pc8, pc9, pc10, pc11, p
c12;
23:     private Point p31, p32, p33, p34, p35, p36, p37, p38;
24:     private LinkedList<Point> corners, corners2;
25:
26:     @Before
27:     public void setUp() throws Exception {
28:         p1 = new Point(0,0);
29:         p2 = new Point(1,0);
30:         p3 = new Point(1,1);
31:         p4 = new Point(0,1);
32:
33:         p5 = new Point(0.5, 1.0);
34:         p6 = new Point(0.5, 0.5);
35:         p7 = new Point(0, 0.5);
36:
37:
38:
39:
40:
41:         corners= new LinkedList<Point>();
42:         corners.add(p1); corners.add(p2); corners.add(p3); corners.add
(p4);
43:
44:         corners2= new LinkedList<Point>();
45:         corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2
.add(p5);
46:
47:         corners2.add(p6); corners2.add(p7);
48:
49:         room = new Room("test", null, corners);
50:         room2 = new Room("test", null, corners2);
51:
52:         pc1 = new Point(1,-1);
53:         pc2 = new Point(2,-1);
54:         pc3 = new Point(2,1);
55:         pc4 = new Point(1,1);
56:         pc5 = new Point(1,2);
57:         pc6 = new Point(-1,2);
58:         pc7 = new Point(-1,1);
59:         pc8 = new Point(-2,1);
60:         pc9 = new Point(-2,-1);
61:         pc10 = new Point(-1,-1);
62:
63:         pc11 = new Point(-1,-2);
64:         pc12 = new Point(1,-2);
65:         LinkedList<Point> cornersStar = new LinkedList<Point>();
66:         cornersStar.add(pc1); cornersStar.add(pc2); cornersStar.add(pc3)
; cornersStar.add(pc4); cornersStar.add(pc5);
67:         cornersStar.add(pc6); cornersStar.add(pc7); cornersStar.add(pc8)
; cornersStar.add(pc9); cornersStar.add(pc10);
68:         cornersStar.add(pc11); cornersStar.add(pc12);
69:
70:         roomStar = new Room("star", null, cornersStar);
71:
72:         p31 = new Point(-2,0);
73:         p32 = new Point(2,0);
74:         p33 = new Point(2,2);
75:         p34 = new Point(1,2);
76:         p35 = new Point(1,1);
77:         p36 = new Point(-1,1);
78:         p37 = new Point(-1,2);
79:         p38 = new Point(-2,2);
80:         LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
81:         cornersHufeisen.add(p31); cornersHufeisen.add(p32); cornersHufe
isen.add(p33); cornersHufeisen.add(p34); cornersHufeisen.add(p35);
82:         cornersHufeisen.add(p36); cornersHufeisen.add(p37); cornersHufe
isen.add(p38);
83:
84:         roomHufeisen = new Room("hufeisen", null, cornersHufeisen);
85:
86:         @Test
87:         public void testTestIfRoomIsIlluminatedIRoomIRuntimeIlluminationTester
() {
88:             //Arrange
89:             IIlluminationTester illuminationTester = AbstractAlgorithmFact
ory.getAlgorithmFactory().createIlluminationTester();
90:
91:             //Act
92:             boolean test1 = false;
93:             boolean test2 = false;
94:             boolean test3 = false;
95:             boolean test4 = false;
96:             boolean test5 = false;
97:
98:             boolean test6 = false;
99:             boolean test7 = false;
100:             boolean test8 = false;
101:             boolean test9 = false;
102:             boolean test10 = false;
103:
104:             try {
105:                 //Room 1
106:                 test1 = illuminationTester.testIfRoomIsIlluminated(roo
m, null);
107:
108:                 Lamp lamp = new Lamp(0.5, 0.5);
109:                 lamp.turnOff();
110:                 room.addLamp(lamp);
111:                 test2 = illuminationTester.testIfRoomIsIlluminated(roo
m, null);
112:
113:                 lamp.turnOn();
114:                 test3 = illuminationTester.testIfRoomIsIlluminated(roo
m, null);
115:
116:                 lamp.turnOff();
117:                 Lamp lamp2 = new Lamp(1.0, 3.0);
118:                 lamp2.turnOn();
119:                 room.addLamp(lamp2);

```

```

117:         test4 = illuminationTester.testIfRoomIsIlluminated(roo
171:
172:         //Assert
173:         assertFalse(test1);
174:         assertTrue(test2);
175:     }
176:
177: }

m, null);
118:         Lamp lamp3 = new Lamp(1.0, 1.0);
119:         lamp3.turnOn();
120:         room.addLamp(lamp3);
121:         test5 = illuminationTester.testIfRoomIsIlluminated(roo
122:
123:         //Room Hufeisen
124:         test6 = illuminationTester.testIfRoomIsIlluminated(roo
mHufeisen, null);
125:         Lamp lamp4 = new Lamp(-1.5, 0.5);
126:         lamp4.turnOn();
127:         roomHufeisen.addLamp(lamp4);
128:         test7 = illuminationTester.testIfRoomIsIlluminated(roo
mHufeisen, null);
129:         Lamp lamp5 = new Lamp(1.5, 0.5);
130:         lamp5.turnOn();
131:         roomHufeisen.addLamp(lamp5);
132:         test8 = illuminationTester.testIfRoomIsIlluminated(roo
mHufeisen, null);
133:
134:     } catch (IlluminationTesterException e) {
135:     }
136:
137:
138:
139:     //Assert
140:     assertFalse(test1);
141:     assertFalse(test2);
142:     assertTrue(test3);
143:     assertFalse(test4);
144:     assertTrue(test5);
145:
146:     assertFalse(test6);
147:     assertFalse(test7);
148:     assertTrue(test8);
149:
150: }
151:
152: @Test
153: public void testTestIfRoomIsIlluminatedIteratorOfLampHashSetOfIntegerI
RuntimeIlluminationTester() {
154:     //Arrange
155:     IIlluminationTester illuminationTester = AbstractAlgorithmFact
ory.getAlgorithmFactory().createIlluminationTester();
156:     HashSet<Integer> allTags = new HashSet<Integer>();
157:     allTags.add(0); allTags.add(1); allTags.add(2); allTags.add(3)
;
158:     List<Lamp> lamps= new LinkedList<Lamp>();
159:     Lamp lamp1 = new Lamp(0,0,0);
160:     lamp1.turnOn();
161:     lamps.add(lamp1);
162:
163:     //Act
164:     boolean test1 = illuminationTester.testIfRoomIsIlluminated(lam
ps.iterator(), allTags, null);
165:     Lamp lamp2 = new Lamp(0,0,1);
166:     lamp2.addTag(2);
167:     lamp2.addTag(3);
168:     lamp2.turnOn();
169:     lamps.add(lamp2);
170:     boolean test2 = illuminationTester.testIfRoomIsIlluminated(lam
ps.iterator(), allTags, null);

```

```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import org.junit.Before;
6: import org.junit.Test;
7:
8: import fernuni.propra.internal_data_model.Point;
9: import fernuni.propra.internal_data_model.Wall;
10:
11: public class WallContainerNorthTest {
12:     Point p1,p2,p3,p4;
13:     Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;
14:
15:     @Before
16:     public void setUp() {
17:         //Arrange
18:         p1 = new Point(0,0);
19:         p2 = new Point(1,0);
20:         p3 = new Point(1,1);
21:         p4 = new Point(0,1);
22:         w1 = new Wall(p1,p2,0);
23:         w2 = new Wall(p2,p3,0);
24:         w3 = new Wall(p3,p4,0);
25:         w4 = new Wall(p4,p1,0);
26:
27:         w5 = new Wall(p2,p1,0);
28:         w6 = new Wall(p3, p2,0);
29:         w7 = new Wall(p4, p3,0);
30:         w8 = new Wall(p1,p4,0);
31:
32:         w9 = new Wall(p1,p1,0);
33:     }
34:
35:
36:     @Test
37:     public void testAdd() {
38:         //Arrange
39:         WallContainerNorth wallContainerNorth = new WallContainerNorth
40: ();
41:
42:         //Act
43:         boolean test1 = false;
44:         try {
45:             wallContainerNorth.add(w2);
46:             fail("WallContainerException expected");
47:         } catch (WallContainerException ex) {
48:             test1 = true;
49:         }
50:
51:         boolean test2 = false;
52:         try {
53:             wallContainerNorth.add(w3);
54:             test2 = true;
55:         } catch (WallContainerException ex) {
56:             fail();
57:         }
58:
59:         //Assert
60:         assertTrue(test1);
61:         assertTrue(test2);
62:
63:     }
64:
65:     @Test
66:     public void testGetNearestNorthWall() {
67:         //Arrange
68:         WallContainerNorth wallContainerNorth = new WallContainerNorth
69: ();
70:
71:         try {
72:             wallContainerNorth.add(w3);
73:         } catch (WallContainerException e) {
74:             // TODO Auto-generated catch block
75:             e.printStackTrace();
76:         }
77:
78:         try {
79:             wallContainerNorth.add(w5);
80:         } catch (WallContainerException e) {
81:             // TODO Auto-generated catch block
82:             e.printStackTrace();
83:         }
84:
85:         //Act
86:         Wall w10 = null;
87:         try {
88:             w10 = wallContainerNorth.getNearestWall(-1, 1, 0.5);
89:         } catch (WallContainerException e) {
90:             fail(e.getMessage());
91:         }
92:
93:         Wall w11 = null;
94:         try {
95:             w11 = wallContainerNorth.getNearestWall(-1, 1, 0.0);
96:         } catch (WallContainerException e) {
97:             fail(e.getMessage());
98:         }
99:
100:         Wall w12 = null;
101:         try {
102:             w12 = wallContainerNorth.getNearestWall(-1, 1, -0.001)
103: };
104:         } catch (WallContainerException e) {
105:             fail(e.getMessage());
106:         }
107:
108:         Wall w13 = null;
109:         try {
110:             w13 = wallContainerNorth.getNearestWall(-1, -0.5, -0.0
111: 01);
112:         } catch (WallContainerException e) {
113:             fail(e.getMessage());
114:         }
115:
116:         //Assert
117:         assertTrue(w10.getP1().isEqual(w3.getP1()) && w10.getP2().isEq
118: ual(w3.getP2()));
119:         assertFalse(w11.getP1().isEqual(w3.getP1()) && w11.getP2().isE
120: qual(w3.getP2()));
121:         assertTrue(w12.getP1().isEqual(w5.getP1()) && w12.getP2().isEq
122: ual(w5.getP2()));
123:         assertTrue(w13 == null);
124:     }
125: }

```



```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.ArrayList;
6: import java.util.Iterator;
7: import java.util.LinkedList;
8: import java.util.List;
9:
10: import org.junit.Before;
11: import org.junit.BeforeClass;
12: import org.junit.Ignore;
13: import org.junit.Test;
14:
15: import fernuni.propra.algorithm.runtime_information.RuntimeInformation;
16: import fernuni.propra.algorithm.util.Rectangle;
17: import fernuni.propra.algorithm.util.RectangleWithTag;
18: import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
19: import fernuni.propra.file_processing.UserReadInputWriteOutputException;
20: import fernuni.propra.algorithm.util.RectangleWithTag;
21: import fernuni.propra.internal_data_model.IRoom;
22: import fernuni.propra.internal_data_model.Lamp;
23: import fernuni.propra.internal_data_model.Point;
24: import fernuni.propra.internal_data_model.Room;
25: import fernuni.propra.internal_data_model.Wall;
26:
27: public class CandidateSearcherTest {
28:
29:     private IRoom mockRoom, room, room2, roomStar, roomHufeisen;
30:     private Point pc1, pc2, pc3, pc4, pc5, pc6, pc7, pc8, pc9, pc10, pc11, p
c12;
31:     private Point p31, p32, p33, p34, p35, p36, p37, p38;
32:
33:
34:
35:     private static List<IRoom> rooms;
36:
37:     @BeforeClass
38:     public static void setupBC() {
39:
40:         String[] xmlPathesOK = {"instances/validationInstances/Selbstt
est_clockwise.xml", //0
41:                                "instances/validationInstances/Selbsttest_coun
terClockwise.xml", //1
42:                                "instances/validationInstances/Selbsttest_100a
_incomplete.xml", // 2
43:                                "instances/validationInstances/Selbsttest_100a
_incomplete.xml", //3
44:                                "instances/validationInstances/Selbsttest_100a
_solved.xml", // 4
45:                                "instances/validationInstances/Selbsttest_100a
.xml", // 5
46:                                "instances/validationInstances/Selbsttest_100b
.xml", // 6
47:                                "instances/validationInstances/Selbsttest_20a_
incomplete.xml", // 7
48:                                "instances/validationInstances/Selbsttest_20a_
solved.xml", // 8
49:                                "instances/validationInstances/Selbsttest_20a.
xml", // 9
50:                                "instances/validationInstances/Selbsttest_20b.
xml", // 10
51:                                "instances/validationInstances/Selbsttest_20c.
xml" // 11
52:                                };
53:
54:         rooms = new ArrayList<IRoom>();
55:
56:         for(String xmlPath : xmlPathesOK) {
57:             UserReadInputWriteOutputAAS readAAS = new UserReadInpu
tWriteOutputAAS(xmlPath);
58:             try {
59:                 rooms.add(readAAS.readInput());
60:             } catch (UserReadInputWriteOutputException e) {
61:                 // TODO Auto-generated catch block
62:                 e.printStackTrace();
63:             }
64:         }
65:     }
66:
67:     @Before
68:     public void setup() {
69:
70:         // build room star
71:         pc1 = new Point(1,-1);
72:         pc2 = new Point(2,-1);
73:         pc3 = new Point(2,1);
74:         pc4 = new Point(1,1);
75:         pc5 = new Point(1,2);
76:         pc6 = new Point(-1,2);
77:         pc7 = new Point(-1,1);
78:         pc8 = new Point(-2,1);
79:         pc9 = new Point(-2,-1);
80:         pc10 = new Point(-1,-1);
81:         pc11 = new Point(-1,-2);
82:         pc12 = new Point(1,-2);
83:         LinkedList<Point> cornersStar = new LinkedList<Point>();
84:         cornersStar.add(pc1); cornersStar.add(pc2); cornersStar.add(pc3)
; cornersStar.add(pc4); cornersStar.add(pc5);
85:         cornersStar.add(pc6); cornersStar.add(pc7); cornersStar.add(pc8)
; cornersStar.add(pc9); cornersStar.add(pc10);
86:         cornersStar.add(pc11); cornersStar.add(pc12);
87:
88:         roomStar = new Room("star", null, cornersStar);
89:
90:         // build room Hufeisen
91:         p31 = new Point(-2,0);
92:         p32 = new Point(2,0);
93:         p33 = new Point(2,2);
94:         p34 = new Point(1,2);
95:         p35 = new Point(1,1);
96:         p36 = new Point(-1,1);
97:         p37 = new Point(-1,2);
98:         p38 = new Point(-2,2);
99:         LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
100:         cornersHufeisen.add(p31); cornersHufeisen.add(p32); cornersHufe
isen.add(p33); cornersHufeisen.add(p34); cornersHufeisen.add(p35);
101:         cornersHufeisen.add(p36); cornersHufeisen.add(p37); cornersHufe
isen.add(p38);
102:         roomHufeisen = new Room("hufeisen", null, cornersHufeisen);
103:
104:     }
105:
106:
107:     @Test
108:     public void testSearchCandidates() {
109:         //Arrange
110:         CandidateSearcher candidateSearcher1 = new CandidateSearcher()

```

```

;
111:      CandidateSearcher candidateSearcher2 = new CandidateSearcher()
;
112:      CandidateSearcher candidateSearcher3 = new CandidateSearcher()
;
113:
114:
115:      //Act
116:      List<Lamp> candidates = null;
117:      List<Lamp> candidates2 = null;
118:      List<Lamp> candidates3 = null;
119:      try {
120:          candidates = candidateSearcher1.searchCandidates(room
Star, new RuntimeInformation());
121:          candidates2 = candidateSearcher2.searchCandidates(room
Hufeisen, new RuntimeInformation());
122:          candidates3 = candidateSearcher3.searchCandidates(room
s.get(9), new RuntimeInformation());
123:      } catch (CandidateSearcherException | InterruptedException e)
{
124:          // TODO Auto-generated catch block
125:          e.printStackTrace();
126:      }
127:
128:      //Assert
129:      assertTrue(candidates != null && candidates.size() == 1 && can
didates.get(0).isEqual(new Point(0.5,0.5)));
130:
131:      assertTrue(candidates2 != null && candidates2.size() == 2);
132:      assertTrue(candidates2.get(0).isEqual(new Point(-1.5,0.5)));
133:      assertTrue(candidates2.get(1).isEqual(new Point(1.5,0.5)));
134:  }
135:
136:      /** Checks if CandidateSearcher correctly determines the reduced tagge
d rectangles from a set of tagged rectangles that might overlap.
137:      * The reduced set contains the rectangles that result from overlappin
g. Only rectangles whose tags are not a subset of the tags of another rectangle are ke
pt.
138:      */
139:      @Test
140:      public void testReduceRectangles() {
141:          // Arrange
142:          ArrayList<RectangleWithTag> rectanglesWithTagIn = new ArrayLis
t<RectangleWithTag>();
143:          CandidateSearcher candidateSearcher = new CandidateSearcher();
144:          RectangleWithTag refRectangleWithTag = new RectangleWithTag(ne
w Point(-2,0), new Point(-1,1), 0);
145:          refRectangleWithTag.addTag(1);
146:          RectangleWithTag refRectangleWithTag2 = new RectangleWithTag(n
ew Point(1,0), new Point(2,1),1);
147:          refRectangleWithTag2.addTag(2);
148:
149:          // Hufeisenkonfiguration von Rechtecken
150:          RectangleWithTag rec0 = new RectangleWithTag(new Point(-2, 0),
new Point(-1, 2), 0);
151:          RectangleWithTag rec1 = new RectangleWithTag(new Point(-2, 0),
new Point(2, 1), 1);
152:          RectangleWithTag rec2 = new RectangleWithTag(new Point(1,0), n
ew Point(2,2),2);
153:          rectanglesWithTagIn.add(rec0); rectanglesWithTagIn.add(rec1);
rectanglesWithTagIn.add(rec2);
154:
155:
156:          // Act
157:
158:          ArrayList<RectangleWithTag> reducedRectangles = new ArrayList<
RectangleWithTag>();
159:          try {
160:              reducedRectangles = candidateSearcher.reduceRectangles
(rectanglesWithTagIn);
161:          } catch (InterruptedException e) {
162:              fail(e.getMessage());
163:          }
164:          // Assert
165:          assertTrue("Number of reduced rectangles is not correct.", red
ucedRectangles.size() == 2);
166:          assertTrue(reducedRectangles.get(0).equals(refRectangleWithTag
));
167:          assertTrue(reducedRectangles.get(1).equals(refRectangleWithTag
2));
168:
169:      }
170:
171:  }

```

```

1: package fernuni.propra.algorithm;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.LinkedList;
6: import java.util.List;
7:
8: import org.junit.Before;
9: import org.junit.Test;
10:
11: import fernuni.propra.internal_data_model.IRoom;
12: import fernuni.propra.internal_data_model.Lamp;
13: import fernuni.propra.internal_data_model.Point;
14: import fernuni.propra.internal_data_model.Room;
15: import fernuni.propra.internal_data_model.Wall;
16:
17: public class UserValidateAASTest {
18:     private IRoom room, room2, roomStar, roomHufeisen;
19:     private Point p1, p2, p3, p4, p5, p6, p7, p8;
20:     private Point pc1, pc2, pc3, pc4, pc5, pc6, pc7, pc8, pc9, pc10, pc11, p
c12;
21:     private Point p31, p32, p33, p34, p35, p36, p37, p38;
22:     private Wall w1, w2, w3, w4;
23:     private LinkedList<Point> corners, corners2;
24:
25:     @Before
26:     public void setup() {
27:         p1 = new Point(0,0);
28:         p2 = new Point(1,0);
29:         p3 = new Point(1,1);
30:         p4 = new Point(0,1);
31:
32:         p5 = new Point(0.5, 1.0);
33:         p6 = new Point(0.5, 0.5);
34:         p7 = new Point(0, 0.5);
35:
36:
37:
38:         corners= new LinkedList<Point>();
39:         corners.add(p1); corners.add(p2); corners.add(p3); corners.add
(p4);
40:
41:         corners2= new LinkedList<Point>();
42:         corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2
.add(p5);
43:         corners2.add(p6); corners2.add(p7);
44:
45:         room = new Room("test", null, corners);
46:         room2 = new Room("test", null, corners2);
47:
48:
49:         pc1 = new Point(1,-1);
50:         pc2 = new Point(2,-1);
51:         pc3 = new Point(2,1);
52:         pc4 = new Point(1,1);
53:         pc5 = new Point(1,2);
54:         pc6 = new Point(-1,2);
55:         pc7 = new Point(-1,1);
56:         pc8 = new Point(-2,1);
57:         pc9 = new Point(-2,-1);
58:         pc10 = new Point(-1,-1);
59:         pc11 = new Point(-1,-2);
60:         pc12 = new Point(1,-2);
61:         LinkedList<Point> cornersStar = new LinkedList<Point>();
62:         cornersStar.add(pc1); cornersStar.add(pc2); cornersStar.add(pc3)
; cornersStar.add(pc4); cornersStar.add(pc5);
63:         cornersStar.add(pc6); cornersStar.add(pc7); cornersStar.add(pc8)
; cornersStar.add(pc9); cornersStar.add(pc10);
64:         cornersStar.add(pc11); cornersStar.add(pc12);
65:
66:         roomStar = new Room("star", null, cornersStar);
67:
68:
69:         p31 = new Point(-2,0);
70:         p32 = new Point(2,0);
71:         p33 = new Point(2,2);
72:         p34 = new Point(1,2);
73:         p35 = new Point(1,1);
74:         p36 = new Point(-1,1);
75:         p37 = new Point(-1,2);
76:         p38 = new Point(-2,2);
77:         LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
78:         cornersHufeisen.add(p31); cornersHufeisen.add(p32); cornersHufe
isen.add(p33); cornersHufeisen.add(p34); cornersHufeisen.add(p35);
79:         cornersHufeisen.add(p36); cornersHufeisen.add(p37); cornersHufe
isen.add(p38);
80:         roomHufeisen = new Room("hufeisen", null, cornersHufeisen);
81:
82:     }
83:
84:     @Test
85:     public void testValidate() {
86:         //Arrange
87:         UserValidateAAS userValidateAAS = new UserValidateAAS();
88:         Lamp lamp1 = new Lamp(0,0);
89:         roomStar.addLamp(lamp1);
90:
91:         //Act
92:         boolean test1 = false;
93:         boolean test2 = false;
94:         try {
95:             test1 = userValidateAAS.validate(roomStar);
96:             test2 = userValidateAAS.validate(room);
97:         } catch (UserValidateAASException e) {
98:             fail("Test result should have been found!");
99:         }
100:
101:         //Assert
102:         assertTrue("Test should have been correct!", test1);
103:         assertFalse("Test should have been not correct!", test2);
104:     }
105:
106: }

```



```

1: package fernuni.propra.user_interface;
2:
3: import static org.junit.Assert.*;
4:
5: import java.awt.Color;
6: import java.util.ArrayList;
7: import java.util.Iterator;
8: import java.util.List;
9:
10: import org.junit.Before;
11: import org.junit.Test;
12:
13: import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
14: import fernuni.propra.file_processing.UserReadInputWriteOutputException;
15: import fernuni.propra.internal_data_model.IRoom;
16: import fernuni.propra.internal_data_model.Lamp;
17: import fernuni.propra.internal_data_model.LineSegment;
18: import fernuni.propra.internal_data_model.Point;
19: import fernuni.propra.internal_data_model.Wall;
20:
21: public class RoomFrameTest {
22:     Point p1,p2,p3,p4,p5;
23:     Wall w1,w2,w3,w4,w5;
24:     List<Wall> walls;
25:     List<IRoom> rooms;
26:
27:     @Before
28:     public void setUp() {
29:         p1 = new Point (-1,-1);
30:         p2 = new Point (1,-1);
31:         p3 = new Point (1,1);
32:         p4 = new Point (-1,1);
33:         w1 = new Wall(p1, p2,0);
34:         w2 = new Wall(p2, p3,0);
35:         w3 = new Wall(p3,p4,0);
36:         w4 = new Wall(p4,p1,0);
37:         w5 = new Wall(p1, p3,0);
38:
39:         walls = new ArrayList<Wall>();
40:         walls.add(w1);walls.add(w2); walls.add(w3); walls.add(w4);
41:
42:         String[] xmlPathesOK = {"instances/validationInstances/Selbstt
43: est_clockwise.xml",
44:     45:         "instances/validationInstances/Selbsttest_coun
45: terClockwise.xml",
46:     46:         "instances/validationInstances/Selbsttest_100a
47: _incomplete.xml",
48:     47:         "instances/validationInstances/Selbsttest_100a
49: _incomplete.xml",
50:     48:         "instances/validationInstances/Selbsttest_100a
51: _solved.xml",
52:     49:         "instances/validationInstances/Selbsttest_100a
53: .xml",
54:     50:         "instances/validationInstances/Selbsttest_100b
55: .xml",
56:     51:         "instances/validationInstances/Selbsttest_20a_
57: incomplete.xml",
58:     52:         "instances/validationInstances/Selbsttest_20a_
59: solved.xml",
60:     53:         "instances/validationInstances/Selbsttest_20a.
61: xml",
62:     54:         "instances/validationInstances/Selbsttest_20b.
63: xml",
64:     55:         "instances/validationInstances/Selbsttest_20c.
65: xml"
66:     };
67:
68:     rooms = new ArrayList<IRoom>();
69:
70:     for(String xmlPath : xmlPathesOK) {
71:         UserReadInputWriteOutputAAS readAAS = new UserReadInpu
72: tWriteOutputAAS(xmlPath);
73:         try {
74:             rooms.add(readAAS.readInput());
75:         } catch (UserReadInputWriteOutputException e) {
76:             // TODO Auto-generated catch block
77:             e.printStackTrace();
78:         }
79:     }
80:
81:     @Test
82:     public void testRoomFrame() {
83:
84:         //Arrange
85:         IRoom mockRoom = new IRoom() {
86:             @Override
87:             public Iterator<Lamp> getLamps() {
88:                 List<Lamp> lamps = new ArrayList<Lamp>();
89:                 Lamp lamp = new Lamp(0.0,0.0);
90:                 lamps.add(lamp);
91:                 lamp.turnOn();
92:                 return lamps.iterator();
93:             }
94:
95:             @Override
96:             public int getNumberOfLamps() {
97:                 // TODO Auto-generated method stub
98:                 return 0;
99:             }
100:
101:             @Override
102:             public Iterator<Point> getCorners() {
103:                 List<Point> corners = new ArrayList<Point>();
104:                 corners.add(p1); corners.add(p2); corners.add(
105: p3); corners.add(p4);
106:                 return corners.iterator();
107:             }
108:
109:             @Override
110:             public void addLamp(Lamp lamp) {
111:
112:             }
113:
114:             @Override
115:             public Iterator<Wall> getWalls() {
116:                 return walls.iterator();
117:             }
118:
119:             @Override
120:             public double getMinX() {
121:                 return -1;
122:             }
123:         };
124:     }
125: }

```

```

115:         @Override
116:         public double getMaxX() {
117:             return 1.0;
118:         }
119:
120:         @Override
121:         public double getMinY() {
122:             return -1.0;
123:         }
124:
125:         @Override
126:         public double getMaxY() {
127:             return 1.0;
128:         }
129:
130:         @Override
131:         public String getID() {
132:             return "MockRoom";
133:         }
134:
135:         @Override
136:         public void replaceLamps(List<Lamp> lamps) {
137:             // TODO Auto-generated method stub
138:
139:         }
140:
141:         @Override
142:         public String printLampPositions() {
143:             // TODO Auto-generated method stub
144:             return null;
145:         }
146:
147:     };
148:
149:
150:     RoomPanel mockRoomPanel = new RoomPanel(mockRoom);
151:     mockRoomPanel.addRectangle("Nr.1", Color.BLUE, 0.5, 0.5, 0.25,
0.25);
152:     //RoomPanel roomPanel = new RoomPanel(mockRoom);
153:     RoomFrame mockRoomFrame = new RoomFrame(mockRoomPanel);
154:     try {
155:         Thread.currentThread().sleep(3000);
156:     } catch (InterruptedException e) {
157:         // TODO Auto-generated catch block
158:         e.printStackTrace();
159:     }
160:     mockRoomFrame.dispose();
161:
162:
163:     for (IRoom room : rooms) {
164:         RoomPanel roomPanel = new RoomPanel(room);
165:         RoomFrame roomFrame = new RoomFrame(roomPanel);
166:         try {
167:             Thread.currentThread().sleep(3000);
168:         } catch (InterruptedException e) {
169:             // TODO Auto-generated catch block
170:             e.printStackTrace();
171:         }
172:         roomFrame.dispose();
173:     }
174:
175: }
176:
177: }

```

```

1: package fernuni.propra.main;
2:
3: import static org.junit.Assert.*;
4:
5: import org.junit.Before;
6: import org.junit.Ignore;
7: import org.junit.Test;
8:
9: public class MainTest {
10:
11:     @Before
12:     public void setUp() throws Exception {
13:     }
14:
15:     @Test
16:     @Ignore
17:     public void testUseCase_D() {
18:         //Arrange
19:         String[] commandLineParameters = new String[] {"r=d",
20:             "if=instances/validationInstances/Selbsttest_2
0b.xml"};
21:
22:         String[] commandLineParameters2 = new String[] {"r=d",
23:             "if=instances/validationInstances/Selbsttest_20b.xml"};
24:
25:         //Act
26:         Main.main(commandLineParameters);
27:         Main.main(commandLineParameters2);
28:         try {
29:             Thread.currentThread().sleep(1000);
30:         } catch (InterruptedException e) {
31:             // TODO Auto-generated catch block
32:             e.printStackTrace();
33:         }
34:         //Assert
35:     }
36:
37:
38:     @Test
39:     public void testUseCase_SD() {
40:         //Arrange
41:         String[] commandLineParameters = new String[] {"if=instances/v
alidationInstances/Selbsttest_100b.xml", "r=sd", "l=-15" };
42:
43:         //Act
44:         Main.main(commandLineParameters);
45:
46:         try {
47:             Thread.currentThread().sleep(4000);
48:         } catch (InterruptedException e) {
49:             // TODO Auto-generated catch block
50:             e.printStackTrace();
51:         }
52:
53:     }
54:
55:
56:     @Test
57:     public void testUseCase_V() {
58:         // Arrange
59:         String[] commandLineParameters = new String[] {"if=instances/v
alidationInstances/Selbsttest_20a_incomplete.xml", "r=v" };
60:
61:         //Act
62:
63:         Main.main(commandLineParameters);
64:         //Assert
65:     }
66:
67: }

```



```

1: package fernuni.propra.internal_data_model;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.ArrayList;
6: import java.util.List;
7:
8: import javax.sound.sampled.Line;
9:
10: import org.junit.Before;
11: import org.junit.BeforeClass;
12: import org.junit.Ignore;
13: import org.junit.Test;
14: import org.junit.runner.RunWith;
15:
16: import fernuni.propra.internal_data_model.LineSegment;
17: import fernuni.propra.internal_data_model.LineSegmentException;
18: import fernuni.propra.internal_data_model.Point;
19:
20: public class LineSegmentTest {
21:     Point p1,p2,p3,p4,p5;
22:     LineSegment l1,l2,l3,l4,l5;
23:     List<LineSegment> lineSegments;
24:
25:     @Before
26:     public void setUp() {
27:         p1 = new Point (0,0);
28:         p2 = new Point (1,0);
29:         p3 = new Point (1,1);
30:         p4 = new Point (0,1);
31:         l1 = new LineSegment(p1, p2);
32:         l2 = new LineSegment(p2, p3);
33:         l3 = new LineSegment(p3,p4);
34:         l4 = new LineSegment(p4,p1);
35:         l5 = new LineSegment(p1, p3);
36:         lineSegments = new ArrayList<LineSegment>();
37:         lineSegments.add(l1);lineSegments.add(l2); lineSegments.add(l3
); lineSegments.add(l4);
38:
39:     }
40:
41:
42:
43:
44:     @Test
45:     public void testGetP1() {
46:         //Arrange
47:         Point p1 = new Point (0,0);
48:         Point p2 = new Point (0,1);
49:         LineSegment linesegment = new LineSegment(p1, p2);
50:
51:         //Act
52:         Point px = linesegment.getP1();
53:
54:         //Assert
55:         assertEquals(p1, px);
56:
57:     }
58:
59:     @Test
60:     public void testGetP2() {
61:         //Arrange
62:         Point p1 = new Point (0,0);
63:         Point p2 = new Point (0,1);

```

```

64:         LineSegment linesegment = new LineSegment(p1, p2);
65:
66:         //Act
67:         Point px = linesegment.getP2();
68:
69:         //Assert
70:         assertEquals(p2, px);
71:     }
72:
73:     @Test
74:     public void testIsHorizontal() {
75:         //Arrange
76:         Point p1 = new Point (0,0);
77:         Point p2 = new Point (0,1);
78:         LineSegment linesegment = new LineSegment(p1, p2);
79:         LineSegment l2 = new LineSegment(null, p2);
80:
81:         //Act
82:         boolean isHorizontal = linesegment.isHorizontal();
83:         try {
84:             l2.isHorizontal();
85:             fail();
86:         } catch (NullPointerException ex) {
87:
88:
89:
90:         //Assert
91:         assertTrue(!isHorizontal);
92:
93:     }
94:
95:
96:
97:     @Test
98:     public void testIsVertical() {
99:         //Arrange
100:         Point p1 = new Point (0,0);
101:         Point p2 = new Point (0,1);
102:         LineSegment linesegment = new LineSegment(p1, p2);
103:
104:         //Act
105:         boolean isVertical = linesegment.isVertical();
106:
107:         //Assert
108:         assertTrue(isVertical);
109:
110:
111:     @Test
112:     public void testOverlapsXrange() {
113:         // Act
114:         boolean test1 = l1.overlapsXrange(0, 1);
115:         boolean test2 = l1.overlapsXrange(0.2, 2);
116:         boolean test3 = l1.overlapsXrange(-1, -0.001);
117:         boolean test4 = l1.overlapsXrange(-1, -0.000);
118:         boolean test5 = l1.overlapsXrange(1, 2);
119:         boolean test6 = l1.overlapsXrange(1.0001, 2);
120:         boolean test7 = l2.overlapsXrange(1.0, 1.0);
121:
122:         //Act, Assert
123:         try {
124:             l3.overlapsXrange(3, 2);
125:             fail();
126:         } catch (IllegalArgumentException ex) {
127:

```

```
128:         }
129:
130:         //Assert
131:         assertTrue(test1);
132:         assertTrue(test2);
133:         assertTrue(!test3);
134:         assertTrue(test4);
135:         assertTrue(test5);
136:         assertTrue(!test6);
137:         assertTrue(test7);
138:
139:     }
140:
141:     @Test
142:     public void testOverlapsYrange() {
143:         // Act
144:         boolean test1 = l2.overlapsYrange(0, 1);
145:         boolean test2 = l2.overlapsYrange(0.2, 0.4);
146:         boolean test3 = l2.overlapsYrange(-1, -0.001);
147:         boolean test4 = l2.overlapsYrange(-1, -0.000);
148:         boolean test5 = l2.overlapsYrange(1, 2);
149:         boolean test6 = l2.overlapsYrange(1.0001, 2);
150:         boolean test7 = l1.overlapsXrange(0.0, 0.0);
151:
152:
153:         //Act, Assert
154:         try {
155:             l3.overlapsXrange(3, 2);
156:             fail();
157:         } catch (IllegalArgumentException ex) {
158:
159:         }
160:
161:         //Assert
162:         assertTrue(test1);
163:         assertTrue(test2);
164:         assertTrue(!test3);
165:         assertTrue(test4);
166:         assertTrue(test5);
167:         assertTrue(!test6);
168:         assertTrue(test7);
169:     }
170:
171:     @Test
172:     public void testPerpendicularPointPoint() {
173:         //Act
174:         boolean test1 = l1.perpendicular(p2, p3);
175:         boolean test2 = l1.perpendicular(p1, p2);
176:         boolean test3 = l1.perpendicular(p1,p3);
177:
178:         //Assert
179:         assertTrue(test1);
180:         assertTrue(!test2);
181:         assertTrue(!test3);
182:
183:     }
184:
185:     @Test
186:     public void testPerpendicularLineSegment() {
187:         //Arrange
188:         LineSegment ls1 = new LineSegment(p1, p3);
189:
190:         //Act
191:         boolean test1 = l1.perpendicular(l2);
192:
193:         boolean test2 = l2.perpendicular(l4);
194:         boolean test3 = l1.perpendicular(ls1);
195:
196:         //Assert
197:         assertTrue(test1);
198:         assertTrue(!test2);
199:         assertTrue(!test3);
200:     }
201:
202:     @Test
203:     public void testLineSegmentDoesNotIntersectLineSegments() {
204:         //Arrange
205:         Point pt1 = new Point(2*p2.getX(), p2.getY());
206:         Point pt2 = new Point(0.2, 0.2);
207:         Point pt3 = new Point(0.4,0.2);
208:         Point center = new Point(0.5, 0.5);
209:         Point pt5 = new Point(-0.5, 0.5);
210:         Point pt6 = new Point(1.5, 0.5);
211:         Point pt7 = new Point(0.5,1.5);
212:         Point pt8 = new Point(0.5, -0.5);
213:
214:         LineSegment ls1 = new LineSegment(p2, pt1);
215:         LineSegment ls2 = new LineSegment(pt1, p2);
216:         LineSegment ls3 = new LineSegment(p1, p2);
217:         LineSegment ls4 = new LineSegment(center, pt5);
218:         LineSegment ls5 = new LineSegment(center, pt6);
219:         LineSegment ls6 = new LineSegment(center, pt7);
220:         LineSegment ls7 = new LineSegment(center, pt8);
221:         LineSegment ls8 = new LineSegment(pt2,pt3);
222:
223:         // Act
224:         boolean test1 = ls1.penetratesLineSegments(lineSegments);
225:         boolean test2 = ls2.penetratesLineSegments(lineSegments);
226:         boolean test3 = ls3.penetratesLineSegments(lineSegments);
227:         boolean test4 = ls4.penetratesLineSegments(lineSegments);
228:         boolean test5 = ls5.penetratesLineSegments(lineSegments);
229:         boolean test6 = ls6.penetratesLineSegments(lineSegments);
230:         boolean test7 = ls7.penetratesLineSegments(lineSegments);
231:         boolean test8 = ls8.penetratesLineSegments(lineSegments);
232:
233:         // Assert
234:         assertTrue(!test1);
235:         assertTrue(!test2);
236:         assertTrue(!test3);
237:         assertTrue(test4);
238:         assertTrue(test5);
239:         assertTrue(test6);
240:         assertTrue(test7);
241:         assertTrue(!test8);
242:
243:     }
244:
245:     @Test
246:     public void testIntersectionWithLinesegment() {
247:         //Arrange
248:         Point center = new Point(0.5, 0.5);
249:         Point pt5 = new Point(-0.5, 0.5);
250:         Point pt6 = new Point(1.5, 0.5);
251:         Point pt7 = new Point(0.5,1.5);
252:         Point pt8 = new Point(0.5, -0.5);
253:         Point pt9 = new Point(2.0, 0.0);
254:         Point pt10 = new Point(0.1, -4.0);
255:         Point pt11 = new Point(0.1, -6.0);
```

```
256:
257:
258:
259:     LineSegment ls4 = new LineSegment(center, pt5);
260:     LineSegment ls5 = new LineSegment(center, pt6);
261:     LineSegment ls6 = new LineSegment(center, pt7);
262:     LineSegment ls7 = new LineSegment(center, pt8);
263:     LineSegment ls8 = new LineSegment(p2, pt9);
264:     LineSegment ls9 = new LineSegment(pt10, pt11);
265:
266:     //Act
267:     Point test1 = null;
268:     try {
269:         test1 = l4.intersectionWithLineSegment(ls4);
270:     } catch (LineSegmentException e) {
271:         fail("An intersection point should have been found!");
272:     }
273:
274:     Point test2 = null;
275:     try {
276:         test2 = l2.intersectionWithLineSegment(ls5);
277:     } catch (LineSegmentException e) {
278:     }
279:
280:     Point test3 = null;
281:     try {
282:         test3 = l3.intersectionWithLineSegment(ls6);
283:     } catch (LineSegmentException e) {
284:     }
285:
286:     Point test4 = null;
287:     try {
288:         test4 = l1.intersectionWithLineSegment(ls7);
289:     } catch (LineSegmentException e) {
290:     }
291:
292:     boolean test5 = false;
293:     try {
294:         l1.intersectionWithLineSegment(l1);
295:     } catch (LineSegmentException e) {
296:         test5 = true;
297:     }
298:
299:
300:     Point test6 = null;
301:     try {
302:         test6 = l2.intersectionWithLineSegment(ls8);
303:     } catch (LineSegmentException e) {
304:         fail(e.getMessage());
305:     }
306:
307:
308:     boolean test7 = false;
309:     try {
310:         l1.intersectionWithLineSegment(ls9);
311:     } catch (LineSegmentException e) {
312:         test7 = true;
313:     }
314:
315:     boolean test8 = false;
316:     try {
317:         ls5.intersectionWithLineSegment(l4);
318:     } catch (LineSegmentException e) {
319:         test8 = true;
320:     }
321:
322:     //Assert
323:     assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test1 != null && test1.isEqual(new Point(0, 0.5)));
324:     assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test2 != null && test2.isEqual(new Point(1.0, 0.5)));
325:     assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test3 != null && test3.isEqual(new Point(0.5, 1.0)));
326:     assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test4 != null && test4.isEqual(new Point(0.5, 0.0)));
327:     assertTrue("No intersection point should have been found", test5);
328:     assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test6 != null && test6.isEqual(new Point(1.0, 0.0)));
329:     assertTrue("Not intersection point should have been found", test7);
330:     assertTrue("Not intersection point should have been found", test8);
331:
332:
333: }
```



```

1: package fernuni.propra.internal_data_model;
2:
3: import static org.junit.Assert.*;
4:
5: import org.junit.Before;
6: import org.junit.BeforeClass;
7: import org.junit.Test;
8:
9: public class WallTest {
10:     Point p1,p2,p3,p4;
11:     Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;
12:
13:     @Before
14:     public void setUp() {
15:         //Arrange
16:         p1 = new Point(0,0);
17:         p2 = new Point(1,0);
18:         p3 = new Point(1,1);
19:         p4 = new Point(0,1);
20:         w1 = new Wall(p1,p2,0);
21:         w2 = new Wall(p2,p3,0);
22:         w3 = new Wall(p3,p4,0);
23:         w4 = new Wall(p4,p1,0);
24:
25:         w5 = new Wall(p2,p1,0);
26:         w6 = new Wall(p3, p2,0);
27:         w7 = new Wall(p4, p3,0);
28:         w8 = new Wall(p1,p4,0);
29:
30:         w9 = new Wall(p1,p1,0);
31:     }
32:
33:     @Test
34:     public void testIsNorthWall() {
35:         //Act
36:         boolean test1 = w3.isNorthWall();
37:         boolean test2 = w1.isNorthWall();
38:         boolean test3 = w2.isNorthWall();
39:         boolean test4 = w9.isNorthWall();
40:
41:         //Assert
42:         assertTrue(test1);
43:         assertFalse(test2);
44:         assertFalse(test3);
45:         assertFalse(test4);
46:     }
47:
48:     @Test
49:     public void testIsWestWall() {
50:         //Act
51:         boolean test1 = w4.isWestWall();
52:         boolean test2 = w2.isWestWall();
53:         boolean test3 = w1.isWestWall();
54:         boolean test4 = w9.isWestWall();
55:
56:         //Assert
57:         assertTrue(test1);
58:         assertFalse(test2);
59:         assertFalse(test3);
60:         assertFalse(test4);
61:     }
62:
63:     @Test
64:     public void testIsSouthWall() {
65:         //Act
66:         boolean test1 = w1.isSouthWall();
67:         boolean test2 = w3.isSouthWall();
68:         boolean test3 = w2.isSouthWall();
69:         boolean test4 = w9.isSouthWall();
70:
71:         //Assert
72:         assertTrue(test1);
73:         assertFalse(test2);
74:         assertFalse(test3);
75:         assertFalse(test4);
76:     }
77:
78:     @Test
79:     public void testIsEastWall() {
80:         //Act
81:         boolean test1 = w2.isEastWall();
82:         boolean test2 = w4.isEastWall();
83:         boolean test3 = w1.isEastWall();
84:         boolean test4 = w9.isEastWall();
85:
86:         //Assert
87:         assertTrue(test1);
88:         assertFalse(test2);
89:         assertFalse(test3);
90:         assertFalse(test4);
91:     }
92:
93: }

```



```

1: package fernuni.propra.internal_data_model;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.ArrayList;
6: import java.util.Iterator;
7: import java.util.LinkedList;
8: import java.util.List;
9:
10: import org.junit.Before;
11: import org.junit.Test;
12:
13: public class RoomTest {
14:
15:     Point p1,p2,p3,p4,p5;
16:     LineSegment l1,l2,l3,l4,l5;
17:     List<LineSegment> lineSegments;
18:     LinkedList<Point> corners, cornersClockWise;
19:
20:
21:     @Before
22:     public void setUp() {
23:         p1 = new Point (0,0);
24:         p2 = new Point (1,0);
25:         p3 = new Point (1,1);
26:         p4 = new Point (0,1);
27:         l1 = new LineSegment (p1, p2);
28:         l2 = new LineSegment (p2, p3);
29:         l3 = new LineSegment (p3,p4);
30:         l4 = new LineSegment (p4,p1);
31:         l5 = new LineSegment (p1, p3);
32:
33:         corners= new LinkedList<Point>();
34:         corners.add(p1); corners.add(p2); corners.add(p3);corners.add(
p4);
35:
36:         cornersClockWise = new LinkedList<Point>();
37:         cornersClockWise.add(p1); cornersClockWise.add(p4); cornersClo
ckWise.add(p3); cornersClockWise.add(p2);
38:
39:         lineSegments = new ArrayList<LineSegment>();
40:         lineSegments.add(l1);lineSegments.add(l2); lineSegments.add(l3
); lineSegments.add(l4);
41:     }
42:
43:     @Test
44:     public void testGetWalls() {
45:         // Arrange
46:         Room room = new Room("test",null, corners);
47:         Room roomClockWise = new Room("test", null, cornersClockWise);
48:
49:         // Act
50:         Iterator<Wall> wallIterator = room.getWalls();
51:         Iterator<Wall> wallIteratorClockWise = roomClockWise.getWalls(
);
52:
53:         //Assert
54:         Wall w1 = wallIterator.next();
55:         Wall w2 = wallIterator.next();
56:         Wall w3 = wallIterator.next();
57:         Wall w4 = wallIterator.next();
58:
59:         Wall w5 = wallIteratorClockWise.next();
60:         Wall w6 = wallIteratorClockWise.next();
61:
62:         Wall w7 = wallIteratorClockWise.next();
63:         Wall w8 = wallIteratorClockWise.next();
64:
65:         boolean test1 = w1.isEqual(l1);
66:         boolean test2 = w2.isEqual(l2);
67:         boolean test3 = w3.isEqual(l3);
68:         boolean test4 = w4.isEqual(l4);
69:
70:         boolean test5 = w5.isEqual(l2);
71:         boolean test6 = w6.isEqual(l3);
72:         boolean test7 = w7.isEqual(l4);
73:         boolean test8 = w8.isEqual(l1);
74:
75:         assertTrue(test1 && test2 && test3 && test4);
76:         assertTrue(test5 && test6 && test7 && test8);
77:     }
78:
79:     @Test
80:     public void testGetLamps() {
81:         fail("Not yet implemented");
82:     }
83:
84:     @Test
85:     public void testGetCorners() {
86:         fail("Not yet implemented");
87:     }
88:
89:     @Test
90:     public void testAddLamp() {
91:         fail("Not yet implemented");
92:     }
93:
94:     @Test
95:     public void testGetNumberOfLamps() {
96:         fail("Not yet implemented");
97:     }
98:
99:     @Test
100:     public void testDimensions() {
101:         // Arrange
102:         Room room = new Room("test",null, corners);
103:         LinkedList<Point> corners2 = new LinkedList<Point>();
104:         Point p1 = new Point (-321.32,-432);
105:         corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2
.add(p4);
106:
107:         Room room2 = new Room("test",null, corners2);
108:
109:         //Act
110:         double xMin = room.getMinX();
111:         double xMax = room.getMaxX();
112:         double yMin = room.getMinY();
113:         double yMax = room.getMaxY();
114:
115:         double xMin2 = room2.getMinX();
116:         double xMax2 = room2.getMaxX();
117:         double yMin2 = room2.getMinY();
118:         double yMax2 = room2.getMaxY();
119:
120:         //Assert
121:         assertEquals(0.0, xMin, 0.0001);
122:         assertEquals(1.0, xMax, 0.0001);
123:         assertEquals(0.0, yMin, 0.0001);

```

./ProPra2020_workspace/Test_Component/src/fernuni/propra/internal_data_model/RoomTest.java

Fri Apr 17 11:51:31 2020

```
124:         assertEquals(1.0, yMax, 0.0001);
125:
126:         //Assert
127:         assertEquals(-321.32, xMin2, 0.0001);
128:         assertEquals(1.0, xMax2, 0.0001);
129:         assertEquals(-432, yMin2, 0.0001);
130:         assertEquals(1.0, yMax2, 0.0001);
131:
132:     }
133:
134:
135: }
```



```

1: package fernuni.propra.internal_data_model;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.ArrayList;
6: import java.util.List;
7:
8: import org.junit.Before;
9: import org.junit.Test;
10:
11: import fernuni.propra.internal_data_model.LineSegment;
12: import fernuni.propra.internal_data_model.Point;
13:
14: public class PointTest {
15:     Point p1,p2,p3,p4,p5;
16:     LineSegment l1,l2,l3,l4,l5;
17:     List<LineSegment> lineSegments;
18:
19:
20:     @Before
21:     public void setUp() {
22:         p1 = new Point (0,0);
23:         p2 = new Point (1,0);
24:         p3 = new Point (1,1);
25:         p4 = new Point (0,1);
26:         l1 = new LineSegment (p1, p2);
27:         l2 = new LineSegment (p2, p3);
28:         l3 = new LineSegment (p3,p4);
29:         l4 = new LineSegment (p4,p1);
30:         l5 = new LineSegment (p1, p3);
31:         lineSegments = new ArrayList<LineSegment>();
32:         lineSegments.add(l1);lineSegments.add(l2); lineSegments.add(l3
); lineSegments.add(l4);
33:     }
34:
35:
36:     @Test
37:     public void testIsEqual() {
38:         //Arrange
39:         Point pt1 = new Point(0,0.01);
40:         Point pt2 = new Point(0.001,0.0);
41:         Point pt3 = new Point (131221.2,-500.7);
42:
43:         //Act
44:         boolean test1 = p1.isEqual(pt1);
45:         boolean test2 = p1.isEqual(pt2);
46:         boolean test3 = pt3.isEqual(pt3);
47:
48:         //Assert
49:         assertTrue(!test1);
50:         assertTrue(!test2);
51:         assertTrue(test3);
52:     }
53:
54:     @Test
55:     public void testIsOnLineSegmentPointPoint() {
56:         //Arrange
57:         Point pt1 = new Point(0,0.01);
58:         Point pt2 = new Point(0.001,0.0);
59:         Point pt3 = new Point (131221.2,-500.7);
60:         Point pt4 = new Point (2.0,0.0);
61:         Point pt5 = new Point (4.0,0.0);
62:
63:         //Act
64:
65:         boolean test1 = p1.isOnLineSegment (p1,p4);
66:         boolean test2 = pt3.isOnLineSegment (p1,pt1);
67:
68:         boolean test3 = false;
69:         try {
70:             test3 = p1.isOnLineSegment (p2,p4);
71:         } catch (IllegalArgumentException e) {
72:             test3 = true;
73:         }
74:
75:         boolean test4 = p1.isOnLineSegment (pt4, pt5);
76:
77:         boolean test5 = false;
78:         try {
79:             pt3.isOnLineSegment (p1, p3);
80:         } catch (IllegalArgumentException e) {
81:             test5 = true;
82:         }
83:
84:         //Assert
85:         assertTrue(test1);
86:         assertTrue(!test2);
87:         assertTrue(test3);
88:         assertFalse(test4);
89:         assertTrue(test5);
90:     }
91:
92:     @Test
93:     public void testIsOnLineSegmentLineSegment() {
94:         //Act
95:         boolean test1 = p1.isOnLineSegment (l1);
96:         boolean test2 = p1.isOnLineSegment (l4);
97:
98:
99:
100:         //Assert
101:         assertTrue(test1);
102:         assertTrue(test2);
103:
104:     }
105:
106:     @Test
107:     public void testIsInsidePolygon() {
108:         //Arrange
109:         Point center = new Point (0.5, 0.5);
110:         Point onLine = new Point (1.0,0.5);
111:         Point onCorner = new Point (1.0,1.0);
112:         Point out = new Point (2.0, -10.0);
113:
114:         //Act
115:         boolean test1= center.isInsidePolygon (lineSegments);
116:         boolean test2 = onLine.isInsidePolygon (lineSegments);
117:         boolean test3 = onCorner.isInsidePolygon (lineSegments);
118:         boolean test4 = out.isInsidePolygon (lineSegments);
119:
120:         //Assert
121:         assertTrue(test1);
122:         assertTrue(test2);
123:         assertTrue(test3);
124:         assertTrue(!test4);
125:     }
126:
127:     @Test

```

```
128:     public void testIsInXRange() {
129:
130:         //Act
131:         boolean test1 = p1.isInXRange(0.0, 0.0);
132:         boolean test2 = p2.isInXRange(1.0, 2.0);
133:         boolean test3 = p3.isInXRange(1.0001, 2.00);
134:
135:         boolean test4 = false;
136:         try {
137:             test4 = p4.isInXRange(2, 1.9);
138:         } catch (IllegalArgumentException e) {
139:             test4 = true;
140:         }
141:
142:         //Assert
143:         assertTrue(test1);
144:         assertTrue(test2);
145:         assertFalse(test3);
146:         assertTrue(test4);
147:     }
148:
149:     @Test
150:     public void testIsInYRange() {
151:         //Act
152:         boolean test1 = p1.isInYRange(0.0, 0.0);
153:         boolean test2 = p3.isInYRange(1.0, 2.0);
154:         boolean test3 = p4.isInYRange(1.0001, 2.00);
155:
156:         boolean test4 = false;
157:         try {
158:             test4 = p4.isInYRange(2, 1.9);
159:         } catch (IllegalArgumentException e) {
160:             test4 = true;
161:         }
162:
163:         //Assert
164:         assertTrue(test1);
165:         assertTrue(test2);
166:         assertFalse(test3);
167:         assertTrue(test4);
168:     }
169:
170: }
```

```

1: package fernuni.propra.internal_data_model;
2:
3: import static org.junit.Assert.*;
4:
5: import java.util.ArrayList;
6: import java.util.Arrays;
7: import java.util.Collection;
8: import java.util.List;
9:
10: import org.junit.Before;
11: import org.junit.BeforeClass;
12: import org.junit.Test;
13: import org.junit.runner.RunWith;
14: import org.junit.runners.Parameterized;
15: import org.junit.runners.Parameterized.Parameter;
16: import org.junit.runners.Parameterized.Parameters;
17:
18: import fernuni.propra.internal_data_model.LineSegment;
19: import fernuni.propra.internal_data_model.Point;
20:
21: @RunWith(Parameterized.class)
22: public class LineSegmentTestParameterized {
23:
24:     @Parameter(0)
25:     public LineSegment lp1;
26:     @Parameter(1)
27:     public boolean result1;
28:
29:
30:     // creates the test data
31:     @Parameters
32:     public static Collection<Object[]> data() {
33:         Object[][] data = new Object[][] { { new LineSegment(new Point(0,0), new
ew Point(1,0)), true },
34:         { new LineSegment(new Point(1,0), new Point(1,1)), false }, {
new LineSegment(new Point(1,1), new Point(0,1)), true },
35:         { new LineSegment(new Point(0,0), new Point(0,0)), false } };
36:         return Arrays.asList(data);
37:     }
38:
39:
40:     @Test
41:     public void testIsHorizontalParametrized() {
42:         //Act
43:         boolean isHorizontal = lp1.isHorizontal();
44:
45:         //Assert
46:         assertTrue(isHorizontal==result1);
47:     }
48:
49:
50:
51:
52: }

```