```java
package fernuni.propra.main;


public class ParameterSet {
	private String runParameter;
	private String inputFile;
	private Integer timeLimit;
	private static String[] validRunParameters = {"s","sd","v","vd", "d"};
	private static final String formInputFileParameter = " if=\"pathToFile\"";
	private static final String formTimeLimitParameter = "l=timeLimit , where timeLimit is a positive Integer number";
	private static final String formRunParameter = "r=parameter";


	public ParameterSet() {

	}

	void setRunParameter(String runParameter) throws ParameterSetException {
		if (this.runParameter != null) {
			throw new ParameterSetException("Run parameter is already set. Please provide only one run parameter specification.");
		} else if (!isValidRunParameter(runParameter)) {
			String message = "Run parameter is not valid. Please provide a valid run parameter in the form " + formRunParameter+ ", where parameter is one of: ";
			for (String validParameter : validRunParameters) {
				message = message + validParameter;
				message = message + " ";
			}
			throw new ParameterSetException(message);
		} else {
			this.runParameter = runParameter;
		}
	}

	void setInputFile(String inputFile) throws ParameterSetException {
		if (this.inputFile != null) {
			throw new ParameterSetException("Path to input file is already set. Please provide only one input file specification.");
		} /*else if (!inputFile.startsWith("\"") || !inputFile.endsWith("\"")) {
			throw new ParameterSetException("The path to the input file you entered is: " + inputFile +System.getProperty("line.separator") + " The input file parameter needs to start and end with  \" . "
							+ "Please supply the path to the input file in the form: if=\"pathToFile\"");
		} */
		else {
			this.inputFile = inputFile.replace("\"", "");

		}
	}

	void setTimeLimit(int timeLimit) throws ParameterSetException {
		if (this.timeLimit != null) {
			throw new ParameterSetException("Time limit is already set. Please provide only one time limit specification.");
		} else {
			this.timeLimit = timeLimit;
		}
	}

	boolean isValidParameterSet() throws ParameterSetException {
		if(runParameter == null) {
			String message = "No run parameter provided. Please provide a valid run parameter in the form " + formRunParameter+ ", where parameter is one of: ";
			for (String validParameter : validRunParameters) {
				message = message + validParameter;
				message = message + " ";
			}
			throw new ParameterSetException(message);
		}
		if (runParameter.equals("v") || runParameter.equals("vd") || runParameter.equals("d")) {
			if (inputFile != null) {
				return true;
			} else {
				throw new ParameterSetException("No path to the input file is specified. Please provide the path to the input file in the form:" + formInputFileParameter);
			}
		} else if (runParameter.equals("s") || runParameter.equals("sd")) {
			if (inputFile == null ) {
				throw new ParameterSetException("No path to the input file is specified. Please provide the path to the input file in the form:" + formInputFileParameter);
			} else if (timeLimit == null) {
				throw new ParameterSetException("No time limit is specified. Please provide a time limit in the form:" + formTimeLimitParameter);
			} else {
				return true;
			}
		} else {
			String message = "Run parameter is not valid. Please provide a valid run parameter in the form " + formRunParameter+ ", where parameter is one of: ";
			for (String validParameter : validRunParameters) {
				message = message + validParameter;
				message = message + " ";
			}
			throw new ParameterSetException(message);
		}

	}

	private boolean isValidRunParameter(String runParameter) {
		boolean isValid = false;
		for (String validParameter :  validRunParameters) {
			if (validParameter.equals(runParameter)) {
				isValid  = true;
				break;
			}
		}
		return isValid;
	}

	String getRunParameter() {
		return this.runParameter;
	}

	String getInputFile() {
		return this.inputFile;
	}

	int getTimeLimit() {
		return this.timeLimit;
	}

	/*String[] getValidRunParameters() { // TODO
		String[] outStrings = new String[validRunParameters.length];
		for (int i = 0; i< validRunParameters.length; i++) {
			outStrings[i] = validRunParameters[i];
		}
		return outStrings;
	} */

}
```

```java
package fernuni.propra.main;

public class ParameterSetException extends Exception {

        public ParameterSetException() {
                // TODO Auto-generated constructor stub
        }

        public ParameterSetException(String message) {
                super(message);
                // TODO Auto-generated constructor stub
        }

        public ParameterSetException(Throwable cause) {
                super(cause);
                // TODO Auto-generated constructor stub
        }

}
```

```java
package fernuni.propra.main;

import java.util.Iterator;
import java.util.StringTokenizer;

import fernuni.propra.algorithm.UserSolveAAS;
import fernuni.propra.algorithm.UserSolveAASException;
import fernuni.propra.algorithm.UserValidateAAS;
import fernuni.propra.algorithm.UserValidateAASException;
import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
import fernuni.propra.file_processing.UserReadInputWriteOutputException;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.user_interface.UserDisplayAAS;

/**
 * Haupteinstiegspunkt der Anwendung.
 *
 * In der Main-Komponente m&uumlssen unter anderem die Eingabeparameter verarbeitet werden.</br>
 *
 * f&uumlr den Ablaufparameter "r" wird folgende Festlegung getroffen:
 * <ul>
 * <li>"s" (solve): f&uumlr die durch die XML-Datei beschriebene Probleminstanz wird eine L&oumlsung ermittelt. Die Positionen der Lampen werden in der angegebenen XML-Datei gespeichert. Wenn in der XML-Datei bereits eine L&oumlsung enthalten ist, so ist diese zu &uumlberschreiben.</li>
 * <li>"sd" (solve & display): wie "s", nur dass der Raum sowie die ermittelten Positionen der Lampen zus&aumltzlich in der grafischen Oberfl&aumlche gezeigt werden.</li>
 * <li>"v" (validate): durch diese Option wird gepr&uumlft, ob der in der angegebenen XML-Datei enthaltene Raum durch die ebenso dort angegebenen Lampen vollst&aumlndig ausgeleuchtet ist. Das Ergebnis der Pr&uumlfung sowie die Anzahl und Positionen der Lampen werden ausgegeben. Falls die angegebene XML-Datei keinen zul&aumlssigen Raum enth&aumllt, wird eine Fehlermeldung ausgegeben. Die Ausgabe erfolgt in der Kommandozeile.</li>
 * <li>"vd" (validate & display): wie "v", nur dass der Raum und die Lampen nach der Validierung zus&aumltzlich in der grafischen Oberfl&aumlche angezeigt werden.</li>
 * <li>"d" (display): der in der XML-Datei enthaltene Raum und die Lampen werden in der grafischen Oberfl&aumlche angezeigt. Falls die angegebene XML-Datei keinen zul&aumlssigen Raum enth&aumllt, wird eine Fehlermeldung auf der Kommandozeile ausgegeben.</li>
 * </ul>
 * Der Eingabedateiparameter "if" ist ein String, der den Pfad der Eingabedatei beinhaltet.</br>
 *
 * Der Parameter f&uumlr ein Zeitlimit "l" ist eine positive nat&uumlrliche Zahl, welche die maximale Rechenzeit in Sekunden angibt.
 */
public class Main {

        /**
         * Haupteinstiegsfunktion
         */
        private static final String generalHelpMessage = "Java -jar ProPra.jar r=runParameter if=\"pathToFile\" l=timeLimit \n \n "
                                + "The runParameter specified by r =runParameter is mandatory and must be one of s,sd,v,vd or d .\n "
                                + "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"
                                + "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.";



        public static void main(String[] args) {

                for (String arg : args) { // Debug
                        System.out.println(arg);
                }

                ParameterSet parameterSet = new ParameterSet();
                try {
                        for (String paramString : args) {
                                StringTokenizer st = new StringTokenizer(paramString, "=");
                                if(st.countTokens()==2) {
                                        String parameterKey = st.nextToken().trim();
                                        String parameterValue = st.nextToken().trim();
                                        switch(parameterKey) {
                                        case "r":
                                                parameterSet.setRunParameter(parameterValue);
                                                break;
                                        case "if":
                                                parameterSet.setInputFile(parameterValue);
                                                break;
                                        case "l":
                                                parameterSet.setTimeLimit(Integer.parseInt(parameterValue));
                                                break;
                                        default:
                                                throw new GeneralException();

                                        }

                                } else {
                                        throw new GeneralException();
                                }
                        }

                        if (parameterSet.isValidParameterSet() ) {
                                IRoom room;
                                UserReadInputWriteOutputAAS userReadWriteAAS;
                                UserDisplayAAS userDisplayAAS;
                                UserValidateAAS userValidateAAS;
                                UserSolveAAS userSolveAAS;
                                int numberOfLampsInSolution;

                                printMessageToConsole("Starting computation ...");

                                switch(parameterSet.getRunParameter()) {
                                case "s":
                                        userReadWriteAAS = new UserReadInputWriteOutputAAS(parameterSet.getInputFile());
                                        room = userReadWriteAAS.readInput();
                                        userSolveAAS = new UserSolveAAS();
                                        numberOfLampsInSolution = userSolveAAS.solve(room, parameterSet.getTimeLimit());
                                        userReadWriteAAS.writeOutput(room);
```

```java
                    printMessageToConsole("Computation finished ...");
                    break;
                case "sd":
                    userReadWriteAAS = new UserReadInputWriteOutputAAS(parameterSet.getInputFile());
                    room = userReadWriteAAS.readInput();
                    userSolveAAS = new UserSolveAAS();
                    numberOfLampsInSolution = userSolveAAS.solve(room, parameterSet.getTimeLimit());
                    userReadWriteAAS.writeOutput(room);
                    userDisplayAAS = new UserDisplayAAS();
                    userDisplayAAS.display(room);
                    printMessageToConsole("Computation finished ...");
                    printMessageToConsole("Number of lamps necessary:" + numberOfLampsInSolution); // TODO
                    printMessageToConsole(userSolveAAS.getRuntimeInformation().toString());
                    break;
                case "v":
                    userReadWriteAAS = new UserReadInputWriteOutputAAS(parameterSet.getInputFile());
                    room = userReadWriteAAS.readInput();
                    userValidateAAS = new UserValidateAAS();
                    userValidateAAS.validate(room);
                    printMessageToConsole(userValidateAAS.getResultString());
                    break;
                case "vd":
                    userReadWriteAAS = new UserReadInputWriteOutputAAS(parameterSet.getInputFile());
                    room = userReadWriteAAS.readInput();
                    userValidateAAS = new UserValidateAAS();
                    userValidateAAS.validate(room);
                    //System.out.println(userValidateAAS.getResultString());
                    printMessageToConsole(userValidateAAS.getResultString());
                    userDisplayAAS = new UserDisplayAAS();
                    userDisplayAAS.display(room);
                    break;
                case "d":
                    userReadWriteAAS = new UserReadInputWriteOutputAAS(parameterSet.getInputFile());
                    room = userReadWriteAAS.readInput();
                    userDisplayAAS = new UserDisplayAAS();
                    userDisplayAAS.display(room);
                    break;
                default:

                }

            } else {
                throw new GeneralException();
            }



        }catch (ParameterSetException e) {
            printMessageToConsole(e.getMessage());
            System.exit(0);
        } catch(NumberFormatException nfe) {
            printMessageToConsole("The timeLimit parameter specified by l=timeLimit is not an integer number");
        } catch(GeneralException ge) {
            printMessageToConsole(generalHelpMessage);
            System.exit(0);
        } catch (UserReadInputWriteOutputException e) {
            printMessageToConsole(e.getMessage());
            System.exit(0);
        } catch (UserValidateAASException e) {
            printMessageToConsole(e.getMessage());
            System.exit(0);
        } catch (UserSolveAASException e) {
            printMessageToConsole(e.getMessage());
            System.exit(0);
        }

    }

    private static void printMessageToConsole(String message) {
        System.out.println(message);
    }
}
```

```java
package fernuni.propra.main;

public class GeneralException extends Exception {

	public GeneralException() {
		// TODO Auto-generated constructor stub
	}

	public GeneralException(String message) {
		super(message);
		// TODO Auto-generated constructor stub
	}

	public GeneralException(Throwable cause) {
		super(cause);
		// TODO Auto-generated constructor stub
	}



}
```

```java
package fernuni.propra.file_processing;

import fernuni.propra.internal_data_model.IRoom;

/**
 * A provider of an algorithm that provides persistence to an {@link IRoom} instance.
 * @author alex
 *
 */
public interface IPersistence {
        /**
         * Used to read an {@link IRoom} that is present a certain location.
         * @param location : The location at which the {@link IRoom} is stored.
         * @return : The {@link IRoom}
         * @throws PersistenceException : thrown if an unexpected error occurred during the reading process.
         */
        IRoom readInput(String location) throws PersistenceException;

        /**
         * Used to persistently store the {@link IRoom} at a certain location.
         * @param room : The {@link IRoom}
         * @param location : The location at which the {@link IRoom} is stored.
         * @throws PersistenceException : thrown if an unexpected error occurred during the writing process.
         */
        void writeOutput(IRoom room, String location) throws PersistenceException;

}
```

```java
package fernuni.propra.file_processing;

import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.LineSegment;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Room;

import java.io.File;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.StringReader;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;


import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.input.SAXBuilder;
import org.jdom2.input.sax.XMLReaders;
import org.jdom2.output.Format;
import org.jdom2.output.XMLOutputter;

/**
 * A specific provider of persistence for an {@link IRoom} that stores/and reads  the {@link IRoom}
 * from an xml-file that adheres to the Document Type Definition (DTD) specified in Listing 1 of [1], i.e.
 *
 *      01 <?xml version="1.0" encoding="UTF-8"?>
 *  02 <!ELEMENT Raum (ID, ecken, lampen?)>
 *  03 <!ELEMENT ID (#PCDATA)>
 *  04 <!ELEMENT ecken (Ecke*)>
 *  05 <!ELEMENT lampen (Lampe*)>
 *  06 <!ELEMENT Ecke (x, y)>
 *  07 <!ELEMENT Lampe (x, y)>
 *  08 <!ELEMENT x (#PCDATA)>
 *  09 <!ELEMENT y (#PCDATA)>
 *  <p>
 *  {@link FilePersistence} makes use of
 *  the JDOM2-library (see http://www.jdom.org/)
 *
 * @author alex
 *
 * [1] Aufgabenstellung Programmierpraktikum SS 2020
 */
class FilePersistence implements IPersistence {
        private static final String DTDFileName = "DataModel.dtd";

    //all lamps are turned on initially
        @Override
        public IRoom readInput(String xmlFilePath) throws PersistenceException {


                Document document = null;
                InputStreamReader isr = null;
                Room outRoom = null;
                try {
                        File xmlFile = new File(xmlFilePath);
                        checkFileAvailability(xmlFile);

                        isr = new FileReader(xmlFile);
                        StringBuilder sb = insertDTDForValidation(isr);

                        //parse xml
                        SAXBuilder builder = new SAXBuilder(XMLReaders.DTDVALIDATING);
                        document = builder.build(new StringReader(sb.toString()));
                        Element roomNode = document.getRootElement();

                        String ID = roomNode.getChildText("ID");

                        Element cornersNode = roomNode.getChild("ecken");
                        if (cornersNode == null) { // if no corners are provided (which is valid according to DTD) an exception is thrown since no computation can be done
                                throw new PersistenceException("No corners provided. Cannot compute anything. Please provide an input file with a valid number of Ecken.");
                        }
                        List<Element> cornerNodes = cornersNode.getChildren("Ecke");
                        LinkedList<Point> corners = new LinkedList<Point>();
                        List<LineSegment> walls = new ArrayList<LineSegment>();

                        //loop over all corners
                        for(Element cornerNode : cornerNodes) {
                                Point tmpPoint = new Point(Double.parseDouble(cornerNode.getChildText("x")), Double.parseDouble(cornerNode.getChildText("y")));
                                // add wall
                                if (!corners.isEmpty()) {
                                        LineSegment newWall = new LineSegment(corners.getLast(), tmpPoint);

                                        testAndAddWallToWalls(newWall, walls);
                                }
                                // add corner
                                corners.add(tmpPoint);
                        }
                        // add last wall
                        LineSegment newWall = new LineSegment(corners.getLast(), corners.getFirst());
                        testAndAddWallToWalls(newWall, walls);

                        //add lamps
                        List<Lamp> lamps = getLamps(roomNode, walls);

                        outRoom = new Room(ID,lamps, corners);

                } catch (JDOMException e) {
                        throw new PersistenceException(e);
                } catch (NumberFormatException e) {
                        throw new PersistenceException(e);
                } catch (IOException e) {
                        throw new PersistenceException(e);
                } finally {
                        if (isr != null) {
                                try {
```

```java
                    isr.close();
                } catch(IOException e) {
                    throw new PersistenceException(e);
                }
            }
        }
        return outRoom;
    }


    @Override
    public void writeOutput(IRoom room, String xmlFile) throws PersistenceException {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(xmlFile);

            //build xml structure conforming with DTD definition
            Document outDocument = new Document();

            //root node
            Element roomNode = new Element("Raum");
            outDocument.addContent(roomNode);

            // write ID
            Element ID = new Element("ID");
            ID.addContent(room.getID());
            roomNode.addContent(ID);

            // write corners
            Element cornersNode = new Element("ecken");
            Iterator<Point> cornersOfRoomIterator = room.getCorners();
            while(cornersOfRoomIterator.hasNext()) {
                Point corner = cornersOfRoomIterator.next();
                Element cornerNode = new Element("Ecke");
                // write x,y
                Element xNode = new Element("x");
                Element yNode = new Element("y");
                xNode.addContent(String.valueOf(corner.getX()));
                yNode.addContent(String.valueOf(corner.getY()));
                cornerNode.addContent(xNode);
                cornerNode.addContent(yNode);
                cornersNode.addContent(cornerNode);
            }
            roomNode.addContent(cornersNode);

            Iterator<Lamp> lampIterator = room.getLamps();
            if (lampIterator.hasNext()) {
                Element lampsNode = new Element("lampen");
                while(lampIterator.hasNext()) {
                    Lamp lamp = lampIterator.next();
                    if (lamp.getOn()) { // lamps are only appended to output if they are turned on in the solution, i.e. they
//      are not only candidates but part of the best solution.
                        Element lampNode = new Element("Lampe");
                        Element xNode = new Element("x");
                        Element yNode = new Element("y");
                        xNode.addContent(String.valueOf(lamp.getX()));
                        yNode.addContent(String.valueOf(lamp.getY()));
                        lampNode.addContent(xNode);
                        lampNode.addContent(yNode);
                        lampsNode.addContent(lampNode);
                    }

                }
                roomNode.addContent(lampsNode);
            }

            XMLOutputter xmlOutputter = new XMLOutputter(Format.getPrettyFormat());
            xmlOutputter.getFormat().setEncoding("UTF-8");
            try {
                // actually write output
                xmlOutputter.output(outDocument, fos);
            } catch (IOException e) {
                throw new PersistenceException(e);
            }
        } catch(IOException ioe) {
            throw new PersistenceException(ioe);
        } finally { // clean up
            if (fos != null) {
                try {
                    fos.close();
                } catch(IOException e) {
                    throw new PersistenceException(e);
                }
            }
        }

    }

    /**
     * Returns a {@link List} of {@link Lamp}s that have been specified in the xml file. Uses JDOM-2.
     * @param roomNode : The "Raum" xml-Node of this valid inputfile
     * @param walls
     * @return
     * @throws IOException
     */
    private List<Lamp> getLamps(Element roomNode, List<LineSegment> walls) throws IOException {
        Element lampsNode = roomNode.getChild("lampen");
        List<Lamp> lamps = new LinkedList<Lamp>();
        if(lampsNode != null) { // contains lamps
            List<Element> lampNodes = lampsNode.getChildren("Lampe");
            for (Element lampNode: lampNodes) {
                Lamp tmpLamp = new Lamp(Double.parseDouble(lampNode.getChildText("x")), Double.parseDouble(lampNode.getChildText("y")));
                if (tmpLamp.isInsidePolygon(walls)) { // the lamp is actually positioned inside the room
                    lamps.add(tmpLamp);
                } else {
                    throw new IOException("Not all lamps are actually inside the room. Please provide a valid room layout");
                }
            }
```

```java
                }
                return lamps;
        }

        /**
         * Allows for format checking of the file content that is provided by an {@link InputStreamReader}. Inserts the DTD specification
         * after the first ">" (ASCII dez = 62) has been read -> works for xml-files. Otherwise the specification is appended to the end of the
         * file, which will produce nothing meaning full.
         * @param isr : The {@link InputStreamReader} obtained from an xml file
         * @return A {@link StringBuilder} that has the DTD-specification added to its xml-header.
         * @throws IOException : If read from the supplied {@link InputStreamReader} fails.
         */
        private StringBuilder insertDTDForValidation(InputStreamReader isr) throws IOException {
                StringBuilder sb = new StringBuilder();

                int c = -1;
                while((c = isr.read()) != -1) {
                        sb.append((char) c);
                        if (c == 62) {
                                break;
                        }
                }

                sb.append(System.getProperty("line.separator"));
                sb.append(readDTDFile());
                sb.append(System.getProperty("line.separator"));
                while((c = isr.read()) != -1) {
                        sb.append((char) c);
                }
                return sb;
        }


        private static void checkFileAvailability(File xmlFile) throws FileNotFoundException {
                if (!xmlFile.exists()) {
                        throw new FileNotFoundException("File not found at \"" +  xmlFile + "\". Enter a valid file path.");
                }
                if(!xmlFile.isFile()) {
                        throw new FileNotFoundException("Path does not point to a file. Enter a valid file path.");
                }
        }

        static void testAndAddWallToWalls(LineSegment newWall, List<LineSegment> walls) throws PersistenceException {
                //checks intersections and perpendicularity
                if (walls.isEmpty()) {
                        walls.add(newWall);
                        return;
                }
                if (!newWall.penetratesLineSegments(walls)) {
                        if (newWall.perpendicular(walls.get(walls.size()-1))) {
                                walls.add(newWall);
                        } else {
                                throw new PersistenceException("Sucessive walls are not perpendicular. Please provide a valid room layout!");
                        }
                } else {
                        throw new PersistenceException("Walls intersect. Please provide a valid room layout!");
                }
        }

        private String readDTDFile() throws IOException{
                InputStreamReader isr = null;

                StringBuilder sb = new StringBuilder();
                sb.append("<!DOCTYPE Raum [");

                try {
                        InputStream inputStream = getClass().getResourceAsStream(System.getProperty("file.separator") + DTDFileName);
                        isr = new InputStreamReader(inputStream);
                        boolean firstTagRead = false;
                        int c = -1;
                        while((c=isr.read())!=-1) {
                                if(firstTagRead) {
                                        sb.append((char) c);
                                }
                                else {
                                        if(c==62) {
                                                firstTagRead = true;
                                        }
                                }
                        }
                } catch (IOException e) {
                        throw new IOException(e);
                } finally {
                        if (isr != null) {
                                try {
                                        isr.close();
                                } catch(IOException e) {
                                        throw new IOException(e);
                                }
                        }
                }
                sb.append(System.getProperty("line.separator"));
                sb.append("]>");
                return sb.toString();
        }


}
```

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

```java
package fernuni.propra.file_processing;

public class UserReadInputWriteOutputException extends Exception {
        public UserReadInputWriteOutputException() {
                super();
                // TODO Auto-generated constructor stub
        }

        public UserReadInputWriteOutputException(String message, Throwable cau
se) {
                super(message, cause);
                // TODO Auto-generated constructor stub
        }

        public UserReadInputWriteOutputException(String message) {
                super(message);
                // TODO Auto-generated constructor stub
        }

        public UserReadInputWriteOutputException(Throwable cause) {
                super(cause);
                // TODO Auto-generated constructor stub
        }

}
```

```java
package fernuni.propra.file_processing;

public class PersistenceException extends Exception {

	public PersistenceException() {
		super();
		// TODO Auto-generated constructor stub
	}

	public PersistenceException(String message, Throwable cause) {
		super(message, cause);
		// TODO Auto-generated constructor stub
	}

	public PersistenceException(String message) {
		super(message);
		// TODO Auto-generated constructor stub
	}

	public PersistenceException(Throwable cause) {
		super(cause);
		// TODO Auto-generated constructor stub
	}

}
```

```java
package fernuni.propra.file_processing;

import fernuni.propra.internal_data_model.IRoom;

public class UserReadInputWriteOutputAAS {
        private final String location;
        private IPersistence persistence;

        public UserReadInputWriteOutputAAS(String location) {
                this.location = location;
                this.persistence = new FilePersistence();
        }

        public IRoom readInput() throws UserReadInputWriteOutputException {
                try {
                        return persistence.readInput(location);
                } catch (PersistenceException e) {
                        throw new UserReadInputWriteOutputException(e);
                }
        }

        public void writeOutput(IRoom room) throws UserReadInputWriteOutputExc
eption {
                try {
                        persistence.writeOutput(room, location);
                } catch (PersistenceException e) {
                        throw new UserReadInputWriteOutputException(e);
                }
        }

}
```

```java
package fernuni.propra.internal_data_model;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;


public class Room implements IRoom {
	private List<Lamp> lamps = new LinkedList<Lamp>();
	private final LinkedList<Point> corners;
	private boolean counterClockWise;
	private double minX, maxX, minY, maxY;
	private List<Wall> walls = new LinkedList<Wall>();
	private String ID;

	public double getMinX() {
		return minX;
	}

	public double getMaxX() {
		return maxX;
	}

	public double getMinY() {
		return minY;
	}

	public double getMaxY() {
		return maxY;
	}

	public Room(String ID, List<Lamp> lamps, LinkedList<Point> corners) {
		if (lamps != null) {
			this.lamps = lamps;
		}
		this.corners = corners;
		this.ID = ID;
		computeDimensionAndOrientation();

	}

	@Override
	public Iterator<Lamp> getLamps() {
		return lamps.iterator();
	}

	@Override
	public Iterator<Point> getCorners() {
		if (counterClockWise) {
			return corners.iterator();
		} else {
			return corners.descendingIterator();
		}
	}

	@Override
	public void addLamp(Lamp lamp) {
		lamps.add(lamp);

	}

	@Override
	public int getNumberOfLamps() {
		return lamps.size();
	}

	private void computeDimensionAndOrientation() {
		if (corners.isEmpty()) {
			throw new IllegalArgumentException("Room does not have any corners!");
		} else {
			minX = corners.get(0).getX(); maxX = minX;
			minY = corners.get(0).getY(); maxY = minY;
		}

		Point mostBottomMostRightPoint = null;
		for (Point corner :  corners ) {
			if(mostBottomMostRightPoint != null) {
				if( corner.getY() <= mostBottomMostRightPoint.getY()) {
					if (corner.getX()>mostBottomMostRightPoint.getX()) {
						mostBottomMostRightPoint = corner;
					}
				}
			} else {
				mostBottomMostRightPoint = corner;
			}

			if (corner.getX()< minX) {
				minX = corner.getX();
			} else if(corner.getX()>maxX) {
				maxX = corner.getX();
			}
			if (corner.getY()< minY) {
				minY = corner.getY();
			} else if(corner.getY()>maxY) {
				maxY = corner.getY();
			}
		}

		this.counterClockWise = isCounterClockWise(mostBottomMostRightPoint);
	}

	@Override
	public Iterator<Wall> getWalls() {
		if (walls.isEmpty()) {
			computeWalls();
		}

		return walls.iterator();
	}

	private void computeWalls() {
		Point firstCorner = null;
		Point previousCorner = null;

		Iterator<Point> cornersIterator = getCorners();
		int tag = 0;

		while(cornersIterator.hasNext()) {
			Point corner = cornersIterator.next();
			if (firstCorner == null) {
				firstCorner = corner;
			} else {
```

```java
                                Wall newWall = new Wall(previousCorner, corner, tag);
                                walls.add(newWall);
                        }
                        previousCorner = corner;
                        tag++;
                }
                Wall newWall = new Wall(previousCorner, firstCorner, tag);
                walls.add(newWall);
        }

        @Override
        public String getID() {
                return this.ID;
        }


        private boolean isCounterClockWise(Point mostBottomMostRightPoint) {
                // https://stackoverflow.com/questions/1165647/how-to-determine-if-a-list-of-polygon-points-are-in-clockwise-order/1180256#1180256
                int indexOfBMRMP = corners.indexOf(mostBottomMostRightPoint);
                Point previous;
                Point next;
                if (indexOfBMRMP == 0) {
                        previous = corners.get(corners.size()-1); //TODO this is faster with ArrayList?
                        next = corners.get(indexOfBMRMP+1);
                } else if(indexOfBMRMP == (corners.size()-1)) {
                        previous = corners.get(indexOfBMRMP-1);
                        next = corners.get(0);
                } else {
                        previous = corners.get(indexOfBMRMP-1);
                        next = corners.get(indexOfBMRMP+1);
                }

                double dx1 = mostBottomMostRightPoint.getX()-previous.getX();
                double dx2 = next.getX() - mostBottomMostRightPoint.getX();

                double dy1 = mostBottomMostRightPoint.getY()-previous.getY();
                double dy2 = next.getY() - mostBottomMostRightPoint.getY();

                double crossProduct = dx1*dy2 - dx2*dy1;
                return crossProduct > 0;
        }

        @Override
        public void replaceLamps(List<Lamp> lamps) {
                if (lamps != null) {
                        this.lamps = lamps;
                }
        }

        @Override
        public String printLampPositions() {
                int n = 1;
                String lineSeparator =  System.getProperty("line.separator");
                StringBuilder sb = new StringBuilder();
                sb.append("The room contains ");
                sb.append(String.valueOf(this.lamps.size()));
                String singPl = lamps.size() == 1 ? " lamp." : " lamps.";
                sb.append(singPl);
                sb.append(lineSeparator);
                sb.append("The lamps are located at:");
                sb.append(lineSeparator);
                for (Lamp lamp : lamps) {
                        sb.append("Lamp ");
                        sb.append(String.valueOf(n));
                        sb.append(" located at x=");
                        sb.append(String.valueOf(lamp.getX()));
                        sb.append("  y=");
                        sb.append(String.valueOf(lamp.getY()));
                        sb.append(". The lamp is ");
                        String onOff = lamp.getOn() ? "turned on." : "turned off.";
                        sb.append(onOff);
                        sb.append(lineSeparator);
                        n++;
                }
                String outString = sb.toString();
                return outString;
        }

}
```

```java
package fernuni.propra.internal_data_model;

public class LineSegmentException extends Exception {

	public LineSegmentException() {
		super();
		// TODO Auto-generated constructor stub
	}

	public LineSegmentException(String message) {
		super(message);
		// TODO Auto-generated constructor stub
	}

	public LineSegmentException(Throwable cause) {
		super(cause);
		// TODO Auto-generated constructor stub
	}

}
```

```java
package fernuni.propra.internal_data_model;

import java.util.Iterator;
import java.util.List;

public interface IRoom{
        Iterator<Lamp> getLamps();
        int getNumberOfLamps();
        Iterator<Point> getCorners();
        void addLamp(Lamp lamp);
        Iterator<Wall> getWalls();
        double getMinX();
        double getMaxX();
        double getMinY();
        double getMaxY();
        String getID();
        void replaceLamps(List<Lamp> lamps);
        String printLampPositions();

}

```

```java
package fernuni.propra.internal_data_model;

import java.util.ArrayList;
import java.util.List;

public class Point {
        private final double x;
        private final double y;
        private final static double TOL = 0.0001;
        private final static int PRECISION = 1000;
        public final static double INF = 100000000;

        public Point(double x, double y) {
                this.x = x;
                this.y = y;
        }

        public double getX() {
                return x;
        }

        public double getY() {
                return y;
        }

        public boolean isEqual(Point other) {
                if (other == null) return false;
                        return (Math.round(getX() * PRECISION) == Math.round(other.getX() * PRECISION))
                                        && (Math.round(getY() * PRECISION) == Math.round(other.getY() * PRECISION));
                //return (Math.abs(getX()-other.getX())  + Math.abs(getY()-other.getY()))  < TOL;
        }

        public boolean isOnLineSegment(Point p1, Point p2) {
                if (!p1.sameX(p2) && !p1.sameY(p2)) throw new IllegalArgumentException("Input is not a horizontal or vertical line!");
                boolean xAgrees = this.sameX(p1)
                                && this.sameX(p2);
                boolean yAgrees = Point.agrees(getY(), p1.getY())
                                && Point.agrees(getY(), p2.getY());
                boolean xInRange = isInRange(getX(), p1.getX(), p2.getX());
                boolean yInRange = isInRange(getY(), p1.getY(), p2.getY());
                return (xAgrees && yInRange) || (yAgrees && xInRange);
        }

        public boolean isOnLineSegment (LineSegment lineSegment) {
                return isOnLineSegment(lineSegment.getP1(), lineSegment.getP2());
        }


        boolean sameX(Point other) {
                return Point.agrees(this.x, other.x);
        }

        boolean sameY(Point other) {
                return Point.agrees(this.y, other.y);
        }

        boolean largerX(Point other) {
                return Point.isLarger(this.x, other.x);
        }

        boolean largerY(Point other) {
                return Point.isLarger(this.y, other.y);
        }

        public boolean isInsidePolygon(List<LineSegment> lineSegments) {
                ArrayList<LineSegment> arrayLinesSegments = new ArrayList<LineSegment>(lineSegments);

                // pre lineSegment must be a valid polygonial
                LineSegment testLineSegXP = new LineSegment(this, new Point(INF, getY()));
                LineSegment testLineSegYP = new LineSegment(this, new Point(getX(), INF));

                int intersectionCountXP = 0;
                int intersectionCountYP = 0;
                for (LineSegment lineSegment : lineSegments) {
                        try {
                                testLineSegXP.intersectionWithLinesegment(lineSegment);
                                if (isOnLineSegment(lineSegment.getP1(), lineSegment.getP2())) {
                                        return true; // if point is on wall -> point is in polygonial
                                } else {
                                        intersectionCountXP++;
                                }
                                /*if (lineSegment.getP1().isOnLineSegment(testLineSeg.getP1(), testLineSeg.getP2()) ||
                                        lineSegment.getP2().isOnLineSegment(testLineSeg.getP1(), testLineSeg.getP2())) {
                                        intersectedLineSegmentHasEndPointOnTestLineSegCount ++;
                                } */
                        } catch (LineSegmentException e) {
                        }

                        try {
                                testLineSegYP.intersectionWithLinesegment(lineSegment);
                                if (isOnLineSegment(lineSegment.getP1(), lineSegment.getP2())) {
                                        return true; // if point is on wall -> point is in polygonial
                                } else {
                                        intersectionCountYP++;
                                }
                                /*if (lineSegment.getP1().isOnLineSegment(testLineSeg.getP1(), testLineSeg.getP2()) ||
                                        lineSegment.getP2().isOnLineSegment(testLineSeg.getP1(), testLineSeg.getP2())) {
                                        intersectedLineSegmentHasEndPointOnTestLineSegCount ++;
                                } */
                        } catch (LineSegmentException e) {
                        }


                }

                if ((intersectionCountXP % 2) != 0 || (intersectionCountYP % 2) != 0) { // if number of intersections is odd -> point is in polygonial
                        return true;
        } else {
            return false;
        }
    }

    public boolean isInXRange(double xLow, double xHigh) {
        if (xLow>xHigh) throw new IllegalArgumentException("xLow > xHigh");
        return isInRange(getX(), xLow, xHigh);
    }

    public boolean isInYRange(double yLow, double yHigh) {
        if(yLow > yHigh) throw new IllegalArgumentException("yLow > yHigh");
        return isInRange(getY(), yLow, yHigh);
    }

    public boolean isInsideRectangle(Point p1, Point p3) {
        return isInXRange(p1.getX(), p3.getX()) && isInYRange(p1.getY(), p3.getY());
    }

    private static boolean agrees(double x, double x1) {
        return Math.abs(x-x1)<TOL;
    }

    private static boolean isLarger(double x, double x1) {
        return x-x1 > TOL;
    }

    private static  boolean isInRange(double x, double x1, double x2) {
        return (Math.min(x1, x2) < x + TOL) && (Math.max(x1, x2) > x - TOL);
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if(!(o instanceof Point)) {
            return false;
        }

        Point point = (Point) o;

        return isEqual(point);
    }

    @Override
    public int hashCode() {
        int result = 17;
        result = 31 * result + (int) Math.round(x * PRECISION);
        result = 31 * result + (int) Math.round(y * PRECISION);
        return result;
    }





}
```

```java
package fernuni.propra.internal_data_model;

import java.util.List;

public class LineSegment {
        private final static double TOL = 0.0001;
        private final Point p1;
        private final Point p2;


        public LineSegment(Point p1, Point p2) {
                // pre p1 != 0, p2 != 0
                this.p1 = p1;
                this.p2 = p2;
        }

        public Point getP1() {
                return p1;
        }

        public Point getP2() {
                return p2;
        }

        public boolean isHorizontal() {
                return p1.sameY(p2) && !p1.isEqual(p2);
        }

        public boolean isVertical() {
                return p1.sameX(p2) && !p1.isEqual(p2);
        }

        public boolean overlapsXrange(double xLow, double xHigh) {
                if (xLow>xHigh) throw new IllegalArgumentException("xLow > xHigh! Insert a valid range!");
                // pre xLow < xHigh
                boolean p1IsInRange = p1.isInXRange(xLow, xHigh);
                boolean p2IsInRange = p2.isInXRange(xLow, xHigh);
                boolean p1SmallerP2Greater = (new Point(xLow, p1.getY()).largerX(getP1())) &&
                                (getP2().largerX(new Point(xHigh, p2.getY()))) ;
                boolean p2SmallerP1Greater = (new Point(xLow, p2.getY()).largerX(getP2())) &&
                                (getP1().largerX(new Point(xHigh, p1.getY())));

                return  p1IsInRange || p2IsInRange || p1SmallerP2Greater || p2SmallerP1Greater;
        }

        public boolean overlapsYrange(double yLow, double yHigh) {
                if (yLow>yHigh) throw new IllegalArgumentException("yLow > yHigh! Insert a valid range!");
                // pre yLow < yHigh
                boolean p1IsInRange = p1.isInYRange(yLow, yHigh);
                boolean p2IsInRange = p2.isInYRange(yLow, yHigh);
                boolean p1SmallerP2Greater = (new Point(p1.getX(),yLow).largerY(getP1())) &&
                                (getP2().largerY(new Point(p2.getX(),yHigh)))
;
                boolean p2SmallerP1Greater = (new Point(p2.getY(),yLow ).largerY(getP2())) &&
                                (getP1().largerY(new Point(p1.getX(),yHigh)));

                return  p1IsInRange || p2IsInRange || p1SmallerP2Greater || p2SmallerP1Greater;

        }

        public boolean perpendicular(Point p1, Point p2) {
                double dx = this.p2.getX()-this.p1.getX();
                double dy = this.p2.getY()-this.p1.getY();

                double dxOther = p2.getX()-p1.getX();
                double dyOther = p2.getY()-p1.getY();

                double scalarProduct = dx * dxOther + dy * dyOther;

                return Math.abs(scalarProduct) < TOL ;
        }

        public boolean perpendicular(LineSegment other) {
                return perpendicular(other.getP1(), other.getP2());
        }

        public boolean penetratesLineSegments(List<LineSegment> lineSegments) {
                if (lineSegments.isEmpty()) return false;
                boolean penetrates = false;
                for (int j = 0; j < lineSegments.size(); j++) {
                        LineSegment tmpLineSegment = lineSegments.get(j);
                        //if (getP1().isOnLineSegment(tmpLineSegment) || getP2().isOnLineSegment(tmpLineSegment)) {
                        //      return false;
                        //}
                        try {
                                Point intersectionPoint = this.intersectionWithLinesegment(tmpLineSegment);
                                if (!intersectionPoint.isEqual(this.getP1()) &&
                                                !intersectionPoint.isEqual(this.getP2())) {
                                        penetrates = true;
                                        break;
                                }
                        } catch (LineSegmentException e) {

                        }
                }
                return penetrates;
        }

        public Point intersectionWithLinesegment(LineSegment other) throws LineSegmentException {
                if (perpendicular(other)) {
                        double x;
                        double y;
                        if (isHorizontal()) {
                                y = getP1().getY();
                                x = other.getP1().getX();
                        } else {
                                y = other.getP1().getY();
                                x = getP1().getX();
                        }
                        Point outPoint = new Point(x,y);
                        if (outPoint.isOnLineSegment(other) && outPoint.isOnLi
neSegment(this)) {
                                        return outPoint;
                                } else {
                                        throw new LineSegmentException("Line Segments
do not intersect!");
                                }
                        } else {
                                throw new LineSegmentException("Lines are not perpendi
cular cannot return (unique) intersection point!");
                        }

                }

        public boolean isEqual(LineSegment other) {
                return getP1().isEqual(other.getP1()) && getP2().isEqual(other
.getP2());
        }

}
```

```java
package fernuni.propra.internal_data_model;


public class Wall extends LineSegment{
        private int tag;


        public Wall(Point p1, Point p2, int tag) {
                super(p1, p2);
                this.tag = tag;
        }

        public Wall(LineSegment lineSegment, int tag) {
                super(lineSegment.getP1(), lineSegment.getP2());
                this.tag = tag;
        }

        public boolean isNorthWall() {
                return isHorizontal() && getP1().largerX(getP2());
        }

        public boolean isWestWall() {
                return isVertical() && getP1().largerY(getP2());
        }

        public boolean isSouthWall() {
                return isHorizontal() && getP2().largerX(getP1());
        }

        public boolean isEastWall() {
                return isVertical() && getP2().largerY(getP1());
        }

        public int getTag() {
                return tag;
        }



}
```

```java
package fernuni.propra.internal_data_model;

import java.util.HashSet;
import java.util.Iterator;

public class Lamp extends Point{
	private volatile boolean on;
	HashSet<Integer> tagsOfCoveredRectangles = new HashSet<Integer>();

	public Lamp(double x, double y) {
		super(x, y);
	}

	public Lamp(double x, double y, int tag) {
		super(x, y);
		tagsOfCoveredRectangles.add(tag);
	}

	public Lamp(double x, double y, HashSet<Integer> tags) {
		super(x,y);
		if (tags != null) {
			this.tagsOfCoveredRectangles = tags;
		}
	}

	public void addTag(Integer tag) {
		tagsOfCoveredRectangles.add(tag);
	}

	public HashSet<Integer> getCopyOfTags() {
		HashSet<Integer> outTags = new HashSet<Integer>();
		for (Integer tag : tagsOfCoveredRectangles) {
			outTags.add(tag);
		}
		return outTags;
	}

	public Iterator<Integer> iteratorTag() {
		return tagsOfCoveredRectangles.iterator();
	}

	public void turnOn() {
		on = true;
	}

	public void turnOff() {
		on = false;
	}

	public boolean getOn() {
		return on;
	}

	public Lamp deepCopy() {
		Lamp outLamp = new Lamp(this.getX(),this.getY());
		for(Integer tag : this.tagsOfCoveredRectangles) {
			outLamp.addTag(tag);
		}
		if (this.on) {
			outLamp.turnOn();
		}
		return outLamp;
	}

}
```

```java
package fernuni.propra.user_interface;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.geom.AffineTransform;


import javax.swing.JPanel;

import fernuni.propra.internal_data_model.IRoom;

public abstract class RoomPanelAbstract extends JPanel{
        private IRoom room;
        private double scale;

        public RoomPanelAbstract(IRoom room) {
                this.room = room;
                setBackground(Color.WHITE);
                setPreferredSize(new Dimension(1024, 768));
        }

        String getRoomID() {
                return room.getID();
        }

        @Override
        protected void paintComponent(Graphics g) {
                super.paintComponent(g);

                Graphics2D g2D= (Graphics2D) g;
                g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

                transformToRoomCoordinates(g2D);
                drawRoom(g2D);

                scaleBackToScreenCoordinates(g2D);
                drawLamps(g2D);
                drawRectangles(g2D); // TODO nur Test

                g2D.scale(1, -1);
                g2D.translate(0, -getHeight());


        }
        protected abstract void drawRectangles(Graphics2D g2D);

        protected abstract void drawLamps(Graphics2D g2D) ;

        protected abstract void drawRoom(Graphics2D g2D) ;

        private void scaleBackToScreenCoordinates(Graphics2D g2D) {
                AffineTransform myTransform;
                myTransform = AffineTransform.getScaleInstance(1/scale, 1/scale);
                g2D.transform(myTransform);
        }

        private void transformToRoomCoordinates(Graphics2D g2D) {
                double sx =  (0.9 * getWidth())/(room.getMaxX()-room.getMinX());
                double sy =  (0.9 * getHeight())/(room.getMaxY()-room.getMinY());

                scale = Math.min(sx, sy);
                double centerOffset = 0.5*Math.min(0.1*getWidth(),0.1*getHeight());
                AffineTransform myTransform = AffineTransform.getScaleInstance(scale, scale);
                g2D.transform(myTransform);
                myTransform = AffineTransform.getTranslateInstance(-room.getMinX() + centerOffset/scale,-room.getMinY() + centerOffset/scale);
                g2D.transform(myTransform);
        }

        protected IRoom getRoom() {
                return this.room;
        }

        protected double getScale() {
                return this.scale;
        }

}
```

```java
package fernuni.propra.user_interface;
import java.awt.BorderLayout;

import javax.swing.JFrame;

public class RoomFrame extends JFrame{
        RoomPanelAbstract roomPanel;

        public RoomFrame(RoomPanelAbstract roomPanel) {
                this.roomPanel = roomPanel;
                setTitle(this.roomPanel.getRoomID());
                add(roomPanel, BorderLayout.CENTER);
                pack();
                setDefaultCloseOperation(DISPOSE_ON_CLOSE);
                setLocationRelativeTo(null);
                setVisible(true);
        }


}
```

```java
package fernuni.propra.user_interface;
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Polygon;
import java.awt.Rectangle;
import java.awt.RenderingHints;
import java.awt.geom.AffineTransform;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import javax.swing.JPanel;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.LineSegment;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;

public class RoomPanel extends RoomPanelAbstract{
        private static final int PIXEL_LAMP_DIAMETER = 10; // in pixels
        private List<PlotRectangle> rectangles = new ArrayList<PlotRectangle>();

        public RoomPanel(IRoom room) {
                super(room);
        }

        private static class PlotRectangle { //TODO only for testing
                private double x,y,width,height;
                private Color color;
                private String name;

                public PlotRectangle(String name, Color color, double x, double y, double width, double height) {
                        this.x = x;
                        this.y = y;
                        this.width = width;
                        this.height = height;
                        this.name = name;
                        this.color = color;
                }
        }

        public void addRectangle(String name, Color color,double x, double y, double width, double height) { //TODO only for testing
                rectangles.add(new PlotRectangle(name, color, x, y, width, height));
        }

        public void removeLastRectangle() { // TODO for Debug
                if (rectangles.size() > 0) {
                        rectangles.remove(rectangles.size() -1);
                }
        }


        @Override
        protected void drawLamps(Graphics2D g2D) {
                double scale = getScale();
                IRoom room = getRoom();
                Iterator<Lamp> lampIterator = room.getLamps();
                while(lampIterator.hasNext()) {
                        Lamp lamp = lampIterator.next();
                        Color lampColor = lamp.getOn() ? Color.YELLOW :  Color.DARK_GRAY;
                        g2D.setColor(lampColor);
                        g2D.fillOval( (int) (lamp.getX() * scale) - (int) Math.round(PIXEL_LAMP_DIAMETER/2.0),
                                        (int) (lamp.getY() * scale) - (int) Math.round(PIXEL_LAMP_DIAMETER/2.0), PIXEL_LAMP_DIAMETER, PIXEL_LAMP_DIAMETER);
                        g2D.setStroke(new BasicStroke(2));
                        g2D.setColor(Color.BLACK);
                        g2D.drawOval((int) (lamp.getX() * scale) - (int) Math.round(PIXEL_LAMP_DIAMETER/2.0),
                                        (int) (lamp.getY() * scale) - (int) Math.round(PIXEL_LAMP_DIAMETER/2.0), PIXEL_LAMP_DIAMETER, PIXEL_LAMP_DIAMETER);
                }

        }

        @Override
        protected void drawRoom(Graphics2D g2D) {
                double scale = getScale();
                IRoom room = getRoom();
                g2D.setStroke(new BasicStroke(2/((float) scale)));
                Polygon p = new Polygon();

                Iterator<Point> cornerIterator = room.getCorners();
                while(cornerIterator.hasNext()) {
                        Point corner = cornerIterator.next();
                        p.addPoint((int) (corner.getX()), (int) (corner.getY()));
                }



                g2D.setColor(Color.ORANGE);
                g2D.fillPolygon(p);

                g2D.setColor(Color.BLACK);
                Iterator<Wall> wallIterator = room.getWalls();
                while(wallIterator.hasNext()) {
                        LineSegment wall = wallIterator.next();
                        g2D.drawLine((int) wall.getP1().getX(),(int) wall.getP1().getY(),
                                        (int) wall.getP2().getX(), (int) wall.getP2().getY());
                }


        }

        @Override
        protected void drawRectangles(Graphics2D g2D) { //TODO nur Test
                double scale = getScale();
                g2D.setStroke(new BasicStroke(2));
                for (PlotRectangle rectangle: rectangles) {
                        g2D.setColor(rectangle.color);
                        g2D.drawRect((int) (rectangle.x * scale), (int) (rectangle.y * scale), (int) (rectangle.width * scale), (int) (rectangle.height * scale));
                        g2D.drawString(rectangle.name, (int) ((rectangle.x + rectangle.width/2) * scale),
                                        (int) ((rectangle.y + rectangle.height/2) * scale) );
                }
        }


}
```

```java
package fernuni.propra.user_interface;
import fernuni.propra.internal_data_model.IRoom;

public class UserDisplayAAS {
        public void display(IRoom room) {
                RoomPanelAbstract roomPanel = new RoomPanel(room);
                RoomFrame roomFrame = new RoomFrame(roomPanel);
        }

}
```

```java
package fernuni.propra.algorithm;

import java.util.List;

import fernuni.propra.algorithm.runtime_information.IRuntimePositionOptimizer;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;

/**
 * A provider of an algorithm that finds a minimum set (and number) of tagged {@link Lamp}s that
 * illuminates an {@link IRoom} instance.
 * <p>
 * Implementing classes: {@link PositionOptimizer}
 * @author alex
 *
 */
public interface IPositionOptimizer {
        /** A method that initiates the computation of an optimal (i.e. a set with a minimum number of lamps
         *  that are turned on) set of lamps, represented by a
         *  {@link List}<{@link Lamp}>.
         * <p>
         * The set of {@link Lamp}s that should be minimized is supplied as a {@link List} of
         * tagged {@link Lamp}s. The tags of each {@link Lamp} uniquely denote the portion an {@link IRoom} that is
         * illuminated by each {@link Lamp}. The union of all such tags must be equivalent to all portions of the {@link IRoom}
         *  i.e. if the union of all tags associated with illuminated {@link Lamp}s is equal to the set of all tags, the {@link IRoom}
         *  must be illuminated.
         * <p>
         * Detailed runtime information can be stored to an {@link IRuntimePositionOptimizer} instance
         * <p>
         * The computation may be interrupted, by interrupting the executing thread. Implementations need to guarantee
         * that an {@link InterruptedException} is thrown as fast as possible in this case.
         * <p>
         * @param taggedCandidates : <{@link List}<{@link Lamp}> . The set of {@link Lamp}s that should be minimized is supplied as a {@link List} of
         *  tagged {@link Lamp}s. The tags of each {@link Lamp} uniquely denote the portion an {@link IRoom} that is
         * illuminated by each {@link Lamp}. The union of all such tags must be equivalent to all portions of the {@link IRoom}
         *  i.e. if the union of all tags associated with illuminated {@link Lamp}s is equal to the set of all tags, the {@link IRoom}
         *  must be illuminated.
         * @param runTimeInformation : Detailed runtime information can be stored to an {@link IRuntimePositionOptimizer} instance
         * @return A {@link List}<{@link Lamp}> that  represents the best solution (i.e. a set with a minimum number of {@link Lamp}s
         *  that are turned on) after the computation has finished.
         * @throws InterruptedException
         */
        List<Lamp> optimizePositions( List<Lamp> taggedCandidates,
                        IRuntimePositionOptimizer runTimeInformation) throws InterruptedException;

        /**
         * A method that allows to get the currently available best solution (i.e. a set with a minimum number of {@link Lamp}s
         *  that are turned on) as a {@link List}<{@link Lamp}>, where
         * a minimum number of {@link Lamp}s is turned on that still illuminates the {@link IRoom} represented by the tags of the Lamps.
         * <p>
         * Should only be called after {@link optimizePositions}. Otherwise no solution is available and 0 will be returned.
         * <p>
         * @return A list of {@link Lamp}s that represents the currently available best solution.
         */
        List<Lamp> getCurrentBestSolution();

        /**
         * Returns the number of turned on {@link Lamp}s in the currently available best solution.
         * <p>
         * Should only be called after {@link optimizePositions}. Otherwise no solution is available and 0 will be returned.
         * @return The number of turned on {@link Lamp}s in the currently available best solution.
         */
        int getNumberOfOnLampsBestSolution();

}
```

```java
package fernuni.propra.algorithm;

import java.util.Comparator;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import fernuni.propra.internal_data_model.Wall;

/**
 * An abstract class that represent a generic {@link Wall} container for a certain type of {@link Wall}s,
 * where the type of the {@link WallContainerAbstract} must be specified by implementing subclasses.
 * <p>
 * {@link Wall}s in the container are sorted according to total order specified by a {@link Comparator}<{@link Wall}>,
 * where the ordering must be specified by implementing subclasses. The total ordering is done
 * by mapping {@link Wall}s on double numbers and
 * using the total ordering defined by the ordering of real numbers.
 * <p>
 * {@link Wall}s can be added if they are of the correct type.
 * <p>
 * The {@link WallContainerAbstract} can also return the nearest ( in the sense of the specified ordering), valid wall, where
 * validity of a {@link Wall} must be specified by implementing subclasses.
 * <p>
 * Implements the template pattern, where templates are given for the two core functionalities, i.e. adding walls
 * and obtaining a nearest wall in some sense. Subclasses must fill in the blanks by implementing the abstract methods.
 * <p>
 * Implementing classes : {@link WallContainerEast}, {@link WallContainerNorth}, {@link WallContainerSouth}, {@link WallContainerWest}
 * <p>
 * @author alex
 *
 */
public abstract class WallContainerAbstract implements Iterable<Wall> {

        protected List<Wall> walls; // the walls in this container

        /**
         * Constructor
         */
        public WallContainerAbstract() {
                walls = new LinkedList<Wall>();

        }

        /**
         * A method that allows to add a {@link Wall} of the correct type to the container.
         * @param wall : the {@link Wall} to be added
         * @throws WallContainerException : thrown if wall is not of the correct type.
         */
        public void add(Wall wall) throws WallContainerException{
                if (!isCorrectWallType(wall)) throw new WallContainerException("Wall does not "
                                + "have the correct orientation for this wall container!");
                walls.add(wall);
                walls.sort(getComparator());
        }

        /**
         * A method that allows to search for the next {@link Wall} (in the sense of the total ordering defined
         * by the implementing subclass). The search can be further specified by defining a range of doubles,
         * whose meaning also needs to be specified by the implementing subclasses and the clients.
         * @param low : lower limit of the range
         * @param high : upper limit of the range
         * @param limit : limit to be used to find the next wall according to the ordering.
         * @return the next {@link Wall}
         * @throws WallContainerException : if range is not set correctly
         */
        public Wall getNearestWall(double low, double high, double limit) throws WallContainerException{
                if (low > high) throw new WallContainerException(); // not a valid range
                Iterator<Wall> iterator = walls.iterator(); // walls are ordered
                Wall nextWall;
                while(iterator.hasNext()) {
                        nextWall = iterator.next();
                        if (isValidWall(nextWall, limit, low, high)) { // wall fits the  range and is also the next wall according to the ordering
                                return nextWall;
                        }
                }
                return null;
        }

        /**
         * Provides access to all the walls in the container by returning an iterator.
         */
        public Iterator<Wall> iterator() { // TODO return only a copy?
                return walls.iterator();
        }

        /**
         * Computes whether a {@link Wall} has is indeed a subsequent - relative to the limit -  {@link Wall} with respect to to the
         * total ordering of this container.
         * @param wall
         * @param limit : limit that characterizes the wall and is used to decide if wall is indeed a subsequent {@link Wall} with respect to the
         * total ordering of {@link Wall}s in the container.
         * @param low : lower end of the range that is used to further specify the search for the next {@link Wall}
         * @param high : higher end of the range that is used to further specify the search for the next {@link Wall}
         * @return a boolean that shows whether the {@link Wall} is a valid {@link Wall} that matches the semantics of
         * {@link getNearestWall}
         */
        protected abstract boolean isValidWall(Wall wall, double limit, double low, double high);

        /**
         *
         * @return A {@link Comparator}<{@link Wall}> that specifies the total
 ordering that is used in this container.
         */
        protected abstract Comparator<Wall> getComparator();

        /**
         *
         * @param wall : the {@link Wall} to be checked.
         * @return a boolean that indicates whether wall has the suited type f
or this container.s
         */
        protected abstract boolean isCorrectWallType(Wall wall);
}
```

```java
package fernuni.propra.algorithm;
/**
 * This exception is thrown if something went wrong within the methods of a {@
link WallContainerAbstract}.
 * <p>
 * @author alex
 *
 */
public class WallContainerException extends Exception {

        public WallContainerException(String message) {
                super(message);
        }

        public WallContainerException(Throwable cause) {
                super(cause);
        }

        public WallContainerException() {
                super();
        }

}
```

```java
package fernuni.propra.algorithm;

import java.util.HashSet;
import java.util.Iterator;

import fernuni.propra.algorithm.runtime_information.IRuntimeIlluminationTester;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;

/**
 * A provider of an algorithm that tests if a room is illuminated by a number of lamps.
 * <p>
 * Implementing classes: {@link IlluminationTester}
 * <p>
 * @author alex
 *
 */
public interface IIlluminationTester {
        /**
         * Tests whether an instance of {@link IRoom} is illuminated, by the {@link Lamp}s that are part of that {@link IRoom}.
         * @param room : The {@link IRoom} instance to be checked (must contain information about the {@link Lamp}s)
         * @param runtimeInfo : A data structure of type {@link IRuntimeIlluminationTester} that can store runtime information.
         * @return A boolean that represents whether room is illuminated (true) or not (false).
         * @throws IlluminationTesterException
         */
        boolean testIfRoomIsIlluminated(IRoom room, IRuntimeIlluminationTester runtimeInfo) throws IlluminationTesterException;

        /**
         * Tests whether an a room is illuminated.
         * @param taggedLampsIterator : An Iterator for a set of {@link Lamp}s that are tagged. Each tag represents a portion
         * of the room that is illuminated by that {@link Lamp}.
         * @param allTags : The tags that represent all portions of the room. The union of all tagged portions of the room
         *                                       retrieves the room.
         * @param runtimeInfo : A data structure of type {@link IRuntimeIlluminationTester} that can store runtime information.
         * @return
         */
        boolean testIfRoomIsIlluminated(Iterator<Lamp> taggedLampsIterator, HashSet<Integer> allTags, IRuntimeIlluminationTester runtimeInfo);

        /**
         * Tests whether an a room is illuminated.
         * @param illuminatedTags : A set of tags that represents illuminated portions of the room. Each tag represents
         *                                                         a portion of the room. The union of all tagged portions of the room retrieves the room.
         * @param allTags : The tags that represent all portions of the room. The union of all tagged portions of the room
         *                                       retrieves the room.
         * @param runtimeInfo : A data structure of type {@link IRuntimeIlluminationTester} that can store runtime information.
         * @return
         */
        boolean testIfRoomIsIlluminated(HashSet<Integer> illuminatedTags, HashSet<Integer> allTags, IRuntimeIlluminationTester runtimeInfo);
}
```

```java
package fernuni.propra.algorithm;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import fernuni.propra.algorithm.runtime_information.IRuntimeCandidateSearcher;
import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
import fernuni.propra.algorithm.util.Rectangle;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;

/**
 *
 * A specific provider of an algorithm that can compute a {@link List} of potential {@link Lamp} positions
 * for an instance of {@link IRoom}.
 * <p>
 * The algorithm works as follows:
 * <p>
 * 1.) The original partial rectangles (instances of {@link RectangleWithTag}) of the room are constructed for {@link IRoom} according the method described in [1].
 *     This is delegated to {@link OriginalPartialRectanglesFinder}. The set of potential lamp positions is initialized as the returned set.
 * <p>
 * 2.)  All pairs of original partial rectangles are intersected. If an overlap is found,
 *              the resulting rectangle is added to the set of potential lamp positions and the tags of both original
 *              rectangles are added to the tags of the new rectangle
 * <p>
 * 3.)  The set of potential lamp positions is reduced by only keeping those lamp positions whose tags are not a subset of
 *              the tags of other rectangles in the set
 * <p>
 * 4.)  Steps 2.) and 3.) are repeated until the set does not change any more
 * <p>
 * 5.)  The potential lamp positions are the centers of the remaining tagged rectangles
 * <p>
 * 6.)  {@link Lamp} objects are created at these {@ Point}s and the lamps are tagged with the tags of
 *      the corresponding tagged {@link RectangleWithTag}, i.e. the tags of all original partial rectangles
 *      of the room that contain the {@link Lamp} are saved to the {@Lamp}s tags.
 * <p>
 * 7.)  A {@link List} of all such {@link Lamp}s is returned.
 *
 * <p>
 * Implemented interfaces and super classes: {@link ICandidateSearcher}
 *
 * <p>
 * <p>
 * [1]: Aufgabenstellung zum Grundpraktikum Programmierung im Sommersemester 2020
 *
 *
 *
 * @author alex
 *
 */
public class CandidateSearcher  implements ICandidateSearcher{


        public CandidateSearcher() {
        }

        @Override
        public List<Lamp> searchCandidates(IRoom room, IRuntimeCandidateSearcher runtimeCandidateSearcher) throws CandidateSearcherException, InterruptedException {
                List<Lamp> centersOfReducedRectangles = null; // the potential lamp positions
                try {
                        // find original partial rectangles
                        runtimeCandidateSearcher.startTimeOriginalPartialRectanglesFind(); // store runtime for construction of original partial rectangles
                        ArrayList<RectangleWithTag> originalRectangles =
                                        AbstractAlgorithmFactory.getAlgorithmFactory().createOriginalPartialRectanglesFinder().
                                        findOriginalPartialRectangles(room, runtimeCandidateSearcher);
                        runtimeCandidateSearcher.stopTimeOriginalPartialRectanglesFind();

                        // reduce rectangles: result is non overlapping set of rectangles. Each rectangle contains all tags of
                        // original rectangles that it overlaps
                        List<RectangleWithTag> reducedRectangles = reduceRectangles(originalRectangles);

                        // create lamp objects at each potential position
                        centersOfReducedRectangles = new LinkedList<Lamp>();
                        for (RectangleWithTag rectangle : reducedRectangles) {
                                Point point = rectangle.getCenter();
                                Lamp lamp = new Lamp(point.getX(), point.getY());
                                //Iterator<Integer> tagsOfRectangleIterator = rectangle.getTagIterator();
                                Iterator<Integer> tagsOfRectangleIterator = rectangle.getCopyOfTags().iterator();
                                while(tagsOfRectangleIterator.hasNext()) {
                                        lamp.addTag(tagsOfRectangleIterator.next());
                                }
                                centersOfReducedRectangles.add(lamp);
                        }

                } catch (OriginalPartialRectanglesFinderException  e) {
                        throw new CandidateSearcherException(e); // chain exceptions
                }       catch (RuntimeExceptionLamps rte) {
                        throw new CandidateSearcherException(rte);
                }

                return centersOfReducedRectangles;
        }


        /**
         * Reduces an original set of tagged partial rectangles to a set of non-overlapping partial rectangles that contain all tags of
         * all original partial rectangles that intersect the final rectangle.
         * <p>
```

```java
	 * The algorithm works as follows:
	 * @param originalRectanglesTagged : The original rectangles
	 * @return A set of reduced rectangles as described above.
	 * @throws InterruptedException
	 */
	ArrayList<RectangleWithTag> reduceRectangles(ArrayList<RectangleWithTag> originalRectanglesTagged) throws InterruptedException{


		ArrayList<RectangleWithTag> reducedRectanglesLastIteration = originalRectanglesTagged;
		ArrayList<RectangleWithTag> reducedRectanglesCurrentIteration = null;

		boolean reductionOccured; // the set of rectangles has been further reduced in the current iteration
		do { // as long as set of rectangles can still be reduced

		reductionOccured = false;
		ArrayList<RectangleWithTag> intersectedRectangleWithTags = new ArrayList<RectangleWithTag>(); // set of rectangles with tag that can be constructed from intersections of last iteration rectangles
		for (int i = 0; i < reducedRectanglesLastIteration.size(); i++) {
			RectangleWithTag rectangleWithTagI = reducedRectanglesLastIteration.get(i);
			boolean intersectFoundI = false;
			for (int j = i+1; j< reducedRectanglesLastIteration.size(); j++) {
				RectangleWithTag rectangleWithTagJ = reducedRectanglesLastIteration.get(j);
				Rectangle overlappingRectangle = rectangleWithTagI.overlap(rectangleWithTagJ);

				if(overlappingRectangle != null) { // intersection detected
					intersectFoundI = true;

					// determine tags of overlap
					HashSet<Integer> tagsOfOverlap= new HashSet<Integer>();
					tagsOfOverlap.addAll(rectangleWithTagI.getCopyOfTags());
					tagsOfOverlap.addAll(rectangleWithTagJ.getCopyOfTags());

					// determine all rectangles that also contain center of overlapping rectangle and add that to the tags of the overlap
					for (int k = j+1; k < reducedRectanglesLastIteration.size(); k++) {
						RectangleWithTag rectangleWithTagK = reducedRectanglesLastIteration.get(k);
						if(overlappingRectangle.getCenter().isInsideRectangle(rectangleWithTagK.getP1(), rectangleWithTagK.getP3()) ) {
							tagsOfOverlap.addAll(rectangleWithTagK.getCopyOfTags());
						}
					}

					// add to all new rectangles
					RectangleWithTag overlappingRectangleWithTag = new RectangleWithTag(overlappingRectangle, tagsOfOverlap);
					intersectedRectangleWithTags.add(overlappingRectangleWithTag);
				}
			}
			// still need to keep rectangleWithTagI if no intersection with other rectangles is found
			if (!intersectFoundI) {
				intersectedRectangleWithTags.add(rectangleWithTagI);
			}
		}

		// determine those rectangles whose tags are not contained in another rectangle’s tags
		reducedRectanglesCurrentIteration = new ArrayList<RectangleWithTag>();
		boolean[] isMinRectangle = new boolean[intersectedRectangleWithTags.size()]; // isMinRectangle[i] = true -> tags of rec_i are not contained in another rectangle
		for (int i = 0; i<isMinRectangle.length; i++) {
			isMinRectangle[i] = true;;
		}

		for (int i = 0; i<intersectedRectangleWithTags.size(); i++) {
			RectangleWithTag rectangleWithTagI = intersectedRectangleWithTags.get(i);


			for (int j = i+1; j < intersectedRectangleWithTags.size(); j++) {
				RectangleWithTag rectangleWithTagJ = intersectedRectangleWithTags.get(j);
				boolean iSubsetOfJ = rectangleWithTagJ.getCopyOfTags().containsAll(rectangleWithTagI.getCopyOfTags());
				boolean jSubsetOfI = rectangleWithTagI.getCopyOfTags().containsAll(rectangleWithTagJ.getCopyOfTags());
				if (iSubsetOfJ && jSubsetOfI) { // equal
					isMinRectangle[i] = false;
				} else if (iSubsetOfJ) {
					isMinRectangle[i] = false;
				} else if(jSubsetOfI) {
					isMinRectangle[j] = false;
				}
			}

		}

		// keep only those rectangles whose tags are not contained in another rectangle’s tags for next iteration
		for (int i = 0; i<isMinRectangle.length; i++) {
			if (isMinRectangle[i]) {
				reducedRectanglesCurrentIteration.add(intersectedRectangleWithTags.get(i));
			} else {
				reductionOccured = true; // overlap detected
			}
		}

		// overwrite for next iteration
		reducedRectanglesLastIteration = reducedRectanglesCurrentIteration;


		} while(reductionOccured);

		return reducedRectanglesCurrentIteration;
	}
```

```


}
```

```java
package fernuni.propra.algorithm;

/**
 * Thrown if something went wrong within the Solve use case (optimized {@link
Lamp} positions are found for
 * an {@link IRoom})
 * @author alex
 *
 */
public class UserSolveAASException extends Exception {

        public UserSolveAASException() {
        }

        public UserSolveAASException(String message) {
                super(message);
        }

        public UserSolveAASException(Throwable cause) {
                super(cause);
        }

}
```

```java
package fernuni.propra.algorithm;

/**
 * An exception that is thrown if test that checks whether a room is illuminated or not fails due
 * to some unexpected error.
 * <p>
 * @author alex
 *
 */
public class IlluminationTesterException extends Exception {
        public IlluminationTesterException() {
                // TODO Auto-generated constructor stub
        }

        public IlluminationTesterException(String message) {
                super(message);
                // TODO Auto-generated constructor stub
        }

        public IlluminationTesterException(Throwable cause) {
                super(cause);
                // TODO Auto-generated constructor stub
        }


}
```

```java
package fernuni.propra.algorithm;

import java.util.Comparator;
import java.util.Iterator;

import fernuni.propra.internal_data_model.Wall;

/**
 * A specific container that stores north walls. Those {@link Wall}s can be specified by
 * two {@link Point}s in a horizontal-vertical coordinate system. The {@link Wall}s
 * in this container are ordered in ascending order with respect to the vertical component
 * (y-component) of their {@link Point}s.
 * <p>
 * The total ordering requested by {@link WallContainerAbstract} is such that walls
 * <p>
 * Extended classes and implemented interfaces: {@link WallContainerAbstract}.
 * <p>
 * @author alex
 *
 */
public class WallContainerNorth extends WallContainerAbstract{

        @Override
        protected boolean isValidWall(Wall wall, double limit, double low, double high) {
                return wall.overlapsXrange(low, high) &&  wall.getP1().getY()>=limit;
        }

        @Override
        protected Comparator<Wall> getComparator() {
                return new Comparator<Wall>() { // TODO: dont sort complete list -> find correct position and insert there
                        @Override
                        public int compare(Wall o1, Wall o2) {
                                if (o1.getP1().getY() < o2.getP1().getY()) {
                                        return -1;
                                } else if (o1.getP1().getY()>o2.getP1().getY()) {
                                        return 1;
                                }
                                return 0;
                        }
                };
        }

        @Override
        protected boolean isCorrectWallType(Wall wall) {
                return wall.isNorthWall();
        }
}
```

```java
package fernuni.propra.algorithm;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;

import fernuni.propra.algorithm.runtime_information.IRuntimeOriginalPartialRectanglesFinder;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;

public class OriginalPartialRectanglesFinder_old implements IOriginalPartialRectanglesFinder{

    private static double findWallTOL = 0.001;
    private HashSet<Integer> allTags = new HashSet<Integer>();
    private WallContainerEast wallContainerEast  = new WallContainerEast();
    private WallContainerNorth wallContainerNorth = new WallContainerNorth();
    private WallContainerWest wallContainerWest = new WallContainerWest();
    private WallContainerSouth wallContainerSouth = new WallContainerSouth();
    private ArrayList<RectangleWithTag> originalRectangles = new ArrayList<RectangleWithTag>();

    public OriginalPartialRectanglesFinder_old() {
        // TODO Auto-generated constructor stub
    }

    //public static OriginalPartialRectanglesFinder getOriginalPartialRectanglesFinder() {
    //      if (singleton == null) {
    //              singleton = new OriginalPartialRectanglesFinder();
    //      }
    //      return singleton;
    //}

    @Override
    public ArrayList<RectangleWithTag> findOriginalPartialRectangles(IRoom room, IRuntimeOriginalPartialRectanglesFinder rti) throws OriginalPartialRectanglesFinderException {
        try {
            sortWallsToContainers(room);
            constructOriginalPartialRectangles();
        } catch (WallContainerException | OriginalPartialRectanglesFinderException e) {
            throw new OriginalPartialRectanglesFinderException(e);
        }

        return originalRectangles;
    }

    @Override
    public HashSet<Integer> getAllTags() { // TODO findOriginalPartialRectangles needs to be called first
        return allTags;

    }

    void sortWallsToContainers(IRoom room) throws WallContainerException, OriginalPartialRectanglesFinderException {
        Iterator<Wall> wallIterator = room.getWalls();
        while(wallIterator.hasNext()) {
            Wall nextWall = wallIterator.next();
            if (nextWall.isEastWall()) {
                wallContainerEast.add(nextWall);
            } else if (nextWall.isNorthWall()) {
                wallContainerNorth.add(nextWall);
            } else if (nextWall.isWestWall()) {
                wallContainerWest.add(nextWall);
            } else if (nextWall.isSouthWall()) {
                wallContainerSouth.add(nextWall);
            } else {
                throw new OriginalPartialRectanglesFinderException("Wall orientation cannot be determined! Wall might not be horizontal or vertical");
            }
        }
    }

    void constructOriginalPartialRectangles() throws WallContainerException {

        int rectangleNo = 0;

        for( Wall northWall : wallContainerNorth) {
            double yNorth = northWall.getP1().getY();
            double westXLimit = northWall.getP2().getX();
            double eastXLimit = northWall.getP1().getX();

            Wall nextWestWall = wallContainerWest.getNearestWall(yNorth - findWallTOL,
                                    yNorth - findWallTOL, westXLimit);

            Wall nextEastWall = wallContainerEast.getNearestWall(yNorth - findWallTOL, yNorth - findWallTOL, eastXLimit);

            double xWest = nextWestWall.getP1().getX();
            double xEast = nextEastWall.getP1().getX();

            Wall nextSouthWall = wallContainerSouth.getNearestWall(xWest+findWallTOL, xEast-findWallTOL, yNorth);
            double ySouth = nextSouthWall.getP1().getY();


            rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
        }

        for (Wall eastWall: wallContainerEast) {
            double xEast = eastWall.getP1().getX();
            double southYLimit = eastWall.getP1().getY();
            double northYLimit = eastWall.getP2().getY();

            Wall nextSouthWall = wallContainerSouth.getNearestWall(xEast- findWallTOL, xEast - findWallTOL, southYLimit);
            Wall nextNorthWall = wallContainerNorth.getNearestWall(xEast- findWallTOL, xEast - findWallTOL, northYLimit);

            double ySouth = nextSouthWall.getP1().getY();
            double yNorth = nextNorthWall.getP1().getY();

            Wall nextWestWall = wallContainerWest.getNearestWall(ySouth+findWallTOL, yNorth-findWallTOL, xEast);
            double xWest = nextWestWall.getP1().getX();
```

```java

			rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
		}


		for (Wall westWall: wallContainerWest) {
			double xWest = westWall.getP1().getX();
			double southYLimit = westWall.getP2().getY();
			double northYLimit = westWall.getP1().getY();

			Wall nextSouthWall = wallContainerSouth.getNearestWall(xWest + findWallTOL, xWest + findWallTOL, southYLimit);
			Wall nextNorthWall = wallContainerNorth.getNearestWall(xWest + findWallTOL, xWest + findWallTOL, northYLimit);

			double ySouth = nextSouthWall.getP1().getY();
			double yNorth = nextNorthWall.getP1().getY();

			Wall nextEastWall = wallContainerEast.getNearestWall(ySouth+findWallTOL, yNorth-findWallTOL, xWest);
			double xEast = nextEastWall.getP1().getX();

			rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
		}

		for (Wall southWall: wallContainerSouth) {
			double ySouth = southWall.getP1().getY();
			double eastXLimit = southWall.getP2().getX();
			double westXLimit = southWall.getP1().getX();

			Wall nextEastWall = wallContainerEast.getNearestWall(ySouth + findWallTOL, ySouth + findWallTOL, eastXLimit);
			Wall nextWestWall = wallContainerWest.getNearestWall(ySouth + findWallTOL, ySouth + findWallTOL, westXLimit);

			double xEast = nextEastWall.getP1().getX();
			double xWest = nextWestWall.getP1().getX();

			Wall nextNorthWall = wallContainerNorth.getNearestWall(xWest+findWallTOL, xEast-findWallTOL, ySouth);
			double yNorth = nextNorthWall.getP1().getY();

			rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
		}
	}

	private int addOriginalPartialRectangle(int rectangleNo, double yNorth, double xWest, double xEast, double ySouth) {
		Point southWestCorner = new Point(xWest,ySouth);
		Point northEastCorner = new Point(xEast,yNorth);
		int tag = rectangleNo++;
		RectangleWithTag partialRectangle = new RectangleWithTag(southWestCorner, northEastCorner, tag);
		allTags.add(tag);
		originalRectangles.add(partialRectangle);
		return rectangleNo;
	}

	public Iterator<RectangleWithTag> iteratorOriginalRectangles() {
		return originalRectangles.iterator();
	}

	// TODO for tests
	Iterator<Wall> eastIterator() {
		return wallContainerEast.iterator();
	}

	Iterator<Wall> northIterator() {
		return wallContainerNorth.iterator();
	}

	Iterator<Wall> westIterator() {
		return wallContainerWest.iterator();
	}

	Iterator<Wall> southIterator() {
		return wallContainerSouth.iterator();
	}




}
```

```java
package fernuni.propra.algorithm;

/**
 * Defines an interface that provides the functionality of the program to othe
r applications as an API (Application Programming Interface).
 * <p>
 * Implementing classes: {@link Ausleuchtung}
 * <p>
 * @author alex
 *
 */
public interface IAusleuchtung {

        public abstract boolean validateSolution(String xmlFile);

        public abstract int solve(String xmlFile, int timeLimit);

}
```

```java
package fernuni.propra.algorithm;

/**
 * An exception that is thrown if the computation of original partial rectangl
es of an {@link IRoom} fails due
 * to some unexpected error.
 * @author alex
 *
 */
public class OriginalPartialRectanglesFinderException extends Exception {

	public OriginalPartialRectanglesFinderException() {

	}

	public OriginalPartialRectanglesFinderException(String message) {
		super(message);

	}

	public OriginalPartialRectanglesFinderException(Throwable cause) {
		super(cause);
	}


}
```

```java
package fernuni.propra.algorithm.util;

import fernuni.propra.internal_data_model.Point;

/**
 * A rectangle that does provides functionality to support the algorithm.
 * <p>
 * The rectangle is specified by providing the corner {@link Point}s,
 * where in an horizontal-vertical/x-y coordinate system, the points are
 * always arranged as
 * <p>
 * P1: bottom left
 * P2: bottom right
 * P3: top right
 * P4: top left
 * <p>
 * The functions {@link equals} and {@link hashcode} are overwritten so that
 * {@link Rectangle}s are consideres equal if their corner points are each equal. This also
 * supports usage in a {@link HashSet}.
 * <p>
 * @author alex
 *
 */
public class Rectangle {
        private Point p1,p2,p3,p4; // corner points

        public Rectangle(Point p1, Point p3) {
                if(!isValidRectangle(p1, p3)) {
                        throw new IllegalArgumentException("Rectangle not initialized correctly");
                }
                this.p1 = p1;
                this.p2 = new Point(p3.getX(),p1.getY());
                this.p3 = p3;
                this.p4 = new Point(p1.getX(), p3.getY());
        }

        /**
         * @return bottom left {@link Point}
         */
        public Point getP1() {
                return new Point(p1.getX(), p1.getY());
        }

        /**
         * @return bottom right {@link Point}
         */
        public Point getP2() {
                return new Point(p2.getX(), p2.getY());
        }

        /**
         * @return top right {@link Point}
         */
        public Point getP3() {
                return new Point(p3.getX(), p3.getY());
        }

        /**
         * @return top left {@link Point}
         */
        public Point getP4() {
                return new Point(p4.getX(), p4.getY());
        }

        /**
         * Computes the overlap of two {@link Rectangle} which is also a {@link Rectangle}.
         * The overlap is determined by finding the coordinates of P1, i.e. (x1, y1), and
         * P3, i.e. (x3,y3) as
         * <p>
         * x1 = max(this.p1.x, other p1.x),
         * y1 = max(this.p1.y, other p1.y),
         * x3 = min(this.p3.x, other p3.y),
         * y3 = min(this.p3.y, pther p3.y)
         * <p>
         * The resulting rectangle which is defined by P1, P3 is then checked for orientation
         * and for validity.
         * <p>
         * If the resulting rectangle is valid it is returned. If not then no overlap exists
         * and null is returned.
         * <p>
         *
         * @param other : a {@link Rectangle} that is compared with the calling {@link Rectangle}
         * @return The {@link Rectangle} that represents the overlap between the calling {@link Rectangle} and the parameter
         *                 other. If the rectangles don't overlap, then null is returned.
         */
        public Rectangle overlap(Rectangle other) {
                Point p1 = new Point(Math.max(this.p1.getX(), other.p1.getX()),
                                Math.max(this.p1.getY(), other.p1.getY()));
                Point p3 = new Point(Math.min(this.p3.getX(), other.p3.getX()),
                                Math.min(this.p3.getY(), other.p3.getY()));
                if(isValidRectangle(p1, p3)) {
                        Rectangle outRectangle = new Rectangle(p1, p3);
                        if(outRectangle.isCounterClockWise()) {
                                return outRectangle;
                        } else {
                                return null;
                        }
                } else {
                        return null;
                }

        }

        /**
         *
         * @return the center {@link Point} of the {@link Rectangle}.
         */
        public Point getCenter() {
                double width = p2.getX() - p1.getX();
                double height = p3.getY() - p1.getY();
                return new Point(p1.getX()+width/2.0, p1.getY()+height/2.0);
        }

        /**
         * Checks if {@link Point}s have counter clock wise orientation
         * by evaluating of the cross product P2P3 x P2P1
         * <p>
         * @return A boolean that shows whether the {@link Rectangle} as counter clock wise orientation.
         */
        boolean isCounterClockWise() {
                double dx1 = 0.0;
                double dx2 = p1.getX()-p2.getX();
                double dy1 = p3.getY() - p2.getY();
                double dy2 = 0.0;

                return dx1 * dy2 - dx2 * dy1 > 0;
        }

        @Override
        public boolean equals(Object o) {
                if (o == this) return true;
                if(!(o instanceof Rectangle)) {
                        return false;
                }
                Rectangle r = (Rectangle) o;
                return getP1().isEqual(r.getP1()) && getP2().isEqual(r.getP2())
                                && getP3().isEqual(r.getP3()) && getP4().isEqual(r.getP4()) ;
        }

        @Override
        public int hashCode() {
                int result = 17;
                result = 31 * result + p1.hashCode();
                result = 31 * result + p3.hashCode();
                return result;
        }

        /**
         * Checks whether the {@link Rectangle} is a valid {@link Rectangle} in the sense
         * that P1 might be the bottom left point and P3 the top right point.
         * @param p1 : P1 of the rectangle
         * @param p3 : P3 of the rectangle
         * @return A boolean that shows whether the {@link Rectangle} is a valid rectangle with nonzero volume
         */
        private static boolean isValidRectangle(Point p1, Point p3) {
                boolean isValidRectangle = p1.getX()< p3.getX() && p3.getY()> p1.getY();
                return isValidRectangle;
        }

}
```

```java
package fernuni.propra.algorithm.util;

import java.util.Collection;
import java.util.HashSet;

import fernuni.propra.internal_data_model.Point;

/**
 * A rectangle that can also be tagged, i.e. have a set of integers that represent the tags.
 * <p>
 * The {@link RectangleWithTag} is designed to represent an original partial rectangle of an {@link IRoom}
 * instance.
 * The tags typically represent the portions of the {@link IRoom} that are illuminated if the associated
 * {@link RectangleWithTag} is illuminated. This means that the union of the
 *  tags of all {@RectangleWithTag}s of an {@link IRoom} should be equal to all tags, i.e. all portions of the
 *  {@link IRoom}. The tags are stored internally as a {@link HashSet}<{@link Integer}>.
 * <p>
 * <p>
 * Extended classes: {@link Rectangle}
 * <p>
 * @author alex
 *
 */
public class RectangleWithTag extends Rectangle{

        private HashSet<Integer> tags = new HashSet<Integer>(); // the tags of the Rectangle

        /**
         * Constructor
         * @param p1 : left bottom {@link Point}
         * @param p3 : top right {@link Point}
         * @param initialTags : a {@link Collection} of initial tags.
         */
        public RectangleWithTag(Point p1, Point p3, Collection<Integer> initialTags) {
                super(p1,p3);
                if (initialTags != null) {
                        tags.addAll(initialTags);
                }
        }

        /**
         * Constructor
         * @param rectangle : A {@link Rectangle} that is used to create the new {@link RectangleWithTag}
         * @param initialTags : a {@link Collection} of initial tags.
         */
        public RectangleWithTag(Rectangle rectangle, Collection<Integer> initialTags) {
                this(rectangle.getP1(), rectangle.getP3(), initialTags);
        }

        /**
         * Constructor
         * @param p1 : left bottom {@link Point}
         * @param p3 : top right {@link Point}
         * @param initialTags : a single  initial tag.
         */
        public RectangleWithTag(Point p1, Point p3, Integer initialTag) {
                super(p1,p3);
                if (initialTag != null) {
                        tags.add(initialTag);
                }
        }

        /**
         * Constructor
         * @param rectangle : A {@link Rectangle} that is used to create the new {@link RectangleWithTag}
         * @param initialTag : a single initial tag.
         */
        public RectangleWithTag(Rectangle rectangle, Integer initialTag) {
                this(rectangle.getP1(),rectangle.getP3(), initialTag);
        }

        /**
         * Tests whether the tags of this {@link RectangleWithTag} contain a certain tag
         * @param tag : the integer tag that is to be checked
         * @return : a boolean that represents whether this {@link RectangleWithTag} contains the tag
         */
        public boolean containsTag(Integer tag) {
                return tags.contains(tag);
        }

        /**
         * Adds a certain tag to the tags of this {@link Rectangle}
         * @param tag : the integer tag to be added
         */
        public void addTag(Integer tag) {
                tags.add(tag);
        }

        /**
         * Returns a copy of the {@link RectangleWithTag}s tags.
         * @return : A {@link HashSet} with all copies of the tags of this {@link RectangleWithTag}.
         */
        public HashSet<Integer> getCopyOfTags() {
                HashSet<Integer> outTags = new HashSet<Integer>();
                for (Integer tag : tags) {
                        outTags.add(tag.intValue()); // boxing + unboxing TODO
                }
                return outTags;
        }

        /**
         *
         * @return sum of all tags for hashCode
         */
        private int getSumOfTags() {
                int result = 0;
                for (Integer tag: tags) {
                        result = result + tag;
                }
                return result;
        }

        @Override
        public boolean equals(Object o) {
                if (o == this) return true;
                if(!(o instanceof RectangleWithTag)) {
                        return false;
                }
                RectangleWithTag r = (RectangleWithTag) o;
                return getP1().isEqual(r.getP1()) && getP2().isEqual(r.getP2())
                                && getP3().isEqual(r.getP3()) && getP4().isEqual(r.getP4()) && getSumOfTags() == r.getSumOfTags() ;
        }

        @Override
        public int hashCode() {
                int result = 17;
                result = 31 * result + getP1().hashCode();
                result = 31 * result + getP3().hashCode();
                result = 31 * result + getSumOfTags();
                return result;
        }

}
```

```java
package fernuni.propra.algorithm;

import java.util.Comparator;
import java.util.Iterator;

import fernuni.propra.internal_data_model.Wall;

/**
 * A specific container that stores west walls. Those {@link Wall}s can be specified by
 * two {@link Point}s in a horizontal-vertical coordinate system. The {@link Wall}s
 * in this container are ordered in descending order with respect to the horizontal component
 * (x-component) of their {@link Point}s.
 * <p>
 * The total ordering requested by {@link WallContainerAbstract} is such that walls
 * <p>
 * Extended classes and implemented interfaces: {@link WallContainerAbstract}.
 * <p>
 * @author alex
 *
 */
public class WallContainerWest extends WallContainerAbstract{

        @Override
        protected boolean isValidWall(Wall wall, double limit, double low, double high) {
                boolean isValidWall = wall.overlapsYrange(low, high) &&  wall.getP1().getX()<=limit;
                return isValidWall;
        }


        @Override
        protected Comparator<Wall> getComparator() {
                return new Comparator<Wall>() {
                        @Override
                        public int compare(Wall o1, Wall o2) {
                                if (o1.getP1().getX() > o2.getP1().getX()) {
                                        return -1;
                                } else if (o1.getP1().getX()<o2.getP1().getX()) {
                                        return 1;
                                }
                                return 0;
                        }
                };
        }


        @Override
        protected boolean isCorrectWallType(Wall wall) {
                return wall.isWestWall();
        }

}
```

```java
package fernuni.propra.algorithm;

import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
import fernuni.propra.file_processing.UserReadInputWriteOutputException;
import fernuni.propra.internal_data_model.IRoom;

/**
 * Diese Klasse wird als API (Application Programming Interaface) verwendet. Das
 * bedeutet, dass diese Klasse als Bibliothek fÃ¼r andere Applikationen verwendet
 * werden kann.
 *
 * Bitte achten Sie darauf, am bereits implementierten Rahmen (Klassenname,
 * Package, Methodensignaturen) KEINE VerÃ¤nderungen vorzunehmen.
 * SelbstverstÃ¤ndlich kÃ¶nnen und mÃ¼ssen Sie innerhalb der Methoden Ã\204nderungen
 * vornehmen
 */
public class Ausleuchtung implements IAusleuchtung {

        /**
         * Ã\234berprÃ¼ft die eingegebene LÃ¶sung auf Korrektheit
         * @param xmlFile Dokument mit der LÃ¶sung, die validiert werden soll.
         * @return true, falls die eingelesene LÃ¶sung korrekt ist
         */
        @Override
        public boolean validateSolution(String xmlFile) {
                UserReadInputWriteOutputAAS userReadWriteAAS = new UserReadInputWriteOutputAAS(xmlFile);

                try {
                        IRoom room = userReadWriteAAS.readInput();
                        UserValidateAAS userValidateAAS = new UserValidateAAS();
                        boolean isIlluminated = userValidateAAS.validate(room);
                        return isIlluminated;
                } catch (UserReadInputWriteOutputException e) {
                        // TODO Fehlermeldung auf Konsole ausgeben?
                        //e.printStackTrace();
                        return false;
                } catch (UserValidateAASException e) {
                        // TODO Fehlermeldung auf Konsole ausgeben?
                        //e.printStackTrace();
                        return false;
                }

        }

        /**
         * Ermittelt eine LÃ¶sung zu den eingegebenen Daten
         * @param xmlFile Dokument, das die zu lÃ¶sende Probleminstanz enthÃ¤lt
         * @param timeLimit Zeitlimit in Sekunden
         * @return Anzahl der Lampen der ermittelten LÃ¶sung
         */
        @Override
        public int solve(String xmlFile, int timeLimit) {
                UserReadInputWriteOutputAAS userReadWriteAAS = new UserReadInputWriteOutputAAS(xmlFile);

                try {
                        IRoom room = userReadWriteAAS.readInput();
                        UserSolveAAS userSolveAAS  = new UserSolveAAS();
                        int numberOfLampsBestSolution = userSolveAAS.solve(room, timeLimit);
                        return numberOfLampsBestSolution;
                } catch (UserReadInputWriteOutputException e) {
                        // TODO Fehlermeldung auf Konsole ausgeben?
                        //e.printStackTrace();
                        return 0;
                } catch (UserSolveAASException e) {
                        // TODO Fehlermeldung auf Konsole ausgeben?
                        //e.printStackTrace();
                        return 0;
                }

        }

}
```

```java
package fernuni.propra.algorithm;

/**
 * Thrown if something went wrong with the validation of an {@link IRoom} inst
ance.
 * @author alex
 *
 */
public class UserValidateAASException extends Exception {

        public UserValidateAASException() {
        }

        public UserValidateAASException(String message) {
                super(message);
        }

        public UserValidateAASException(Throwable cause) {
                super(cause);
        }

}
```

```java
package fernuni.propra.algorithm;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import fernuni.propra.algorithm.runtime_information.IRuntimePositionOptimizer;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;

public class PositionOptimizer_old implements IPositionOptimizer{
        private static List<Lamp> currentBestSolution;
        private static int numberIlluminatedLampsBestSolution;
        private static IIlluminationTester illuminationTester = AbstractAlgorithmFactory.getAlgorithmFactory().createIlluminiationTester();

        public PositionOptimizer_old() {
        }

        @Override
        public List<Lamp> optimizePositions( List<Lamp> taggedCandidates, IRuntimePositionOptimizer runTimeInformation) throws InterruptedException{

                // all lamps are on -> illuminated
                currentBestSolution = taggedCandidates;
                numberIlluminatedLampsBestSolution = taggedCandidates.size();


                HashSet<Integer> allTags = new HashSet<Integer>();
                for (Lamp lamp : taggedCandidates) {
                        lamp.turnOff(); // make sure all lamps are turned off
                        Iterator<Integer> tagIterator = lamp.iteratorTag();
                        while(tagIterator.hasNext()) {
                                allTags.add(tagIterator.next());
                        }
                }

                ArrayList<Lamp> lamps = deepCopyLamps(taggedCandidates);

                //HashSet<Integer> illuminated = new HashSet<Integer>();


                searchSolution(lamps,0, allTags, 0,runTimeInformation);

                return currentBestSolution;



        }

        private void searchSolution(ArrayList<Lamp> lamps, int idx,
                        HashSet<Integer> allTags, int numberLampsOn, IRuntimePositionOptimizer runTimeInformation) throws InterruptedException {

                if (Thread.currentThread().isInterrupted()) {
                        throw new InterruptedException();
                }

                if(illuminationTester.testIfRoomIsIlluminated(lamps.iterator(), allTags, runTimeInformation)) { // valid solution found
                        if (numberLampsOn<=numberIlluminatedLampsBestSolution) {
                                System.out.println("Solution found with " + numberLampsOn + " lamps turned on.");
                                currentBestSolution = deepCopyLamps(lamps);
                                numberIlluminatedLampsBestSolution = numberLampsOn;
                        }


                } else { // not a valid solution
                        if (idx < lamps.size()) {
                                if(numberLampsOn<numberIlluminatedLampsBestSolution) {
                                        Lamp lamp = lamps.get(idx);
                                        lamp.turnOn();
                                        searchSolution(deepCopyLamps(lamps), idx+1, allTags, numberLampsOn+1, runTimeInformation);

                                        lamp.turnOff();
                                        searchSolution(deepCopyLamps(lamps), idx+1, allTags, numberLampsOn, runTimeInformation);

                                }
                        }
                }
        }


        private static ArrayList<Lamp> deepCopyLamps(List<Lamp> lamps) {
                ArrayList<Lamp> outLamps = new ArrayList<Lamp>(lamps.size());
                Iterator<Lamp> lampsIterator = lamps.iterator();
                while(lampsIterator.hasNext()) {
                        Lamp lamp = lampsIterator.next();
                        outLamps.add(lamp.deepCopy());
                }
                return outLamps;
        }

        private static HashSet<Integer> deepCopyHashSet(HashSet<Integer> hashSet) {
                HashSet<Integer> outHashSet = new HashSet<Integer>();
                for (Integer integer : hashSet) {
                        Integer outInteger = (int) integer;
                        outHashSet.add(outInteger);
                }
                return outHashSet;
        }

        @Override
        public List<Lamp> getCurrentBestSolution() {
                if (currentBestSolution == null) {
                        return null;
                }
                List<Lamp> outLamps = new LinkedList<Lamp>();
                Iterator<Lamp> lampIterator = currentBestSolution.iterator();
                while(lampIterator.hasNext()) {
                        outLamps.add(lampIterator.next().deepCopy());
                }
                return outLamps;

        }

        @Override
        public int getNumberOfOnLampsBestSolution() {
                return numberIlluminatedLampsBestSolution;
        }




}
```

```java
package fernuni.propra.algorithm;

/**
 * Thrown if something went wrong within the validation algorithm, i.e. the ch
eck
 * whether an {@link IRoom} is illuminated by its {@link Lamp}s.
 * <p>
 * @author alex
 *
 */
public class ValidateKException extends Exception {

        public ValidateKException() {
        }

        public ValidateKException(String message) {
                super(message);
        }

        public ValidateKException(Throwable cause) {
                super(cause);
        }

}
```

```java
package fernuni.propra.algorithm;

import java.util.Comparator;
import java.util.Iterator;

import fernuni.propra.internal_data_model.Wall;

/**
 * A specific container that stores south walls. Those {@link Wall}s can be specified by
 * two {@link Point}s in a horizontal-vertical coordinate system. The {@link Wall}s
 * in this container are ordered in descending order with respect to the vertical component
 * (y-component) of their {@link Point}s.
 * <p>
 * The total ordering requested by {@link WallContainerAbstract} is such that walls
 * <p>
 * Extended classes and implemented interfaces: {@link WallContainerAbstract}.
 * <p>
 * @author alex
 *
 */
public class WallContainerSouth extends WallContainerAbstract {

        @Override
        protected boolean isValidWall(Wall wall, double limit, double low, double high) {
                return wall.overlapsXrange(low, high) &&  wall.getP1().getY()<=limit;
        }

        @Override
        protected Comparator<Wall> getComparator() {
                return new Comparator<Wall>() { // TODO: dont sort complete list -> find correct position and insert there
                        @Override
                        public int compare(Wall o1, Wall o2) {
                                if (o1.getP1().getY() > o2.getP1().getY()) {
                                        return -1;
                                } else if (o1.getP1().getY()<o2.getP1().getY()) {
                                        return 1;
                                }
                                return 0;
                        }
                };
        }

        @Override
        protected boolean isCorrectWallType(Wall wall) {
                return wall.isSouthWall();
        }
}
```

```java
package fernuni.propra.algorithm;


import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.Timer;

import fernuni.propra.algorithm.runtime_information.IRuntimeInformation;
import fernuni.propra.algorithm.runtime_information.IRuntimeReader;
import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
import fernuni.propra.algorithm.runtime_information.RuntimeInformation;
import fernuni.propra.internal_data_model.IRoom;

/**
 * Use case that provides access to the solution algorithm, which allows to compute the optimal {@link Lamp}
 * positions for a given {@link IRoom} instance and a given time limit which has to be specified as an integer number
 * representing the seconds a solution is allowed to take.
 *
 * @author alex
 *
 */
public class UserSolveAAS {
        IRuntimeInformation runTimeInformation = new RuntimeInformation();

        /**
         * The interface to the solution algorithm. A separate thread is started to handle the algorithm that is
         *  interrupted after the time limit has been reached.
         *  <p>
         *  The solving is delegated to an instance of {@link SolveK} that controls the execution of the algorithm
         *  and makes the results available.
         *  <p>
         *
         * @param room : {@link IRoom} instance for which the optimal {@link Lamp} positions have to be found.
         * @param time : The time limit in seconds as an integer number. Negative numbers are treated as infinite
         *                                time limits.
         * @return The number of {@link Lamp}s that are turned on in the best solution.
         * @throws UserSolveAASException
         */
        public int solve(IRoom room, int time) throws UserSolveAASException {
                SolveK solveControl = new SolveK(room, runTimeInformation);
                try {
                        runTimeInformation.startTime();

                        if (time > 0) { // if time argument is smaller than zero time limit is ignored
                                Timer timer = new Timer(time * 1000, new ActionListener() {

                                        @Override
                                        public void actionPerformed(ActionEvent e) {
                                                solveControl.interrupt();
                                        }
                                });
                                timer.start();
                        }
                        solveControl.start();

                        SolveKException solveException = solveControl.testIfComputationFinished();

                        //exception from other thread
                        if(solveException != null) {
                                throw new UserSolveAASException(solveException);
                        }

                        int numberOfOnLampsBestSolution = solveControl.getNumberOfOnLampsBestSolution();
                        runTimeInformation.stopTime();
                        return numberOfOnLampsBestSolution;

                } catch (InterruptedException ie) {
                        throw new UserSolveAASException(ie);
                } catch (RuntimeExceptionLamps rte) {
                        throw new UserSolveAASException(rte);
                } finally {
                        solveControl.interrupt(); // stop solveControl thread
                }
        }

        /**
         * Provides access to runtime information.
         * @return A data structure of type {@link IRuntimeReader} that allows to obtain run time information.
         */
        public IRuntimeReader getRuntimeInformation() {
                return runTimeInformation;
        }

}
```

```java
package fernuni.propra.algorithm;

import fernuni.propra.algorithm.runtime_information.IRuntimeInformation;
import fernuni.propra.algorithm.runtime_information.IRuntimeReader;
import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
import fernuni.propra.algorithm.runtime_information.RuntimeInformation;
import fernuni.propra.internal_data_model.IRoom;
/**
 * Use case that provides access to the validation algorithm for an {@link IRoom} instance.
 * <p>
 * The test for illumination is delegated to an instance of {@link ValidateK} that controls the
 * execution of the algorithm and returns the result.
 * <p>
 *
 * @author alex
 *
 */
public class UserValidateAAS {
        private ValidateK validateK  = new ValidateK();
        private String resultString; // the result to be displayed.
        IRuntimeInformation runtimeInfo = new RuntimeInformation();

        /**
         * Provides the user with access to the validation algorithm
         * @param room
         * @return
         * @throws UserValidateAASException
         */
        public boolean validate(IRoom room) throws UserValidateAASException{
                try {
                        runtimeInfo.startTime();
                        boolean isIlluminated = validateK.validate(room, runtimeInfo);
                        runtimeInfo.stopTime();
                        resultString = computeResultString(room, isIlluminated);
                        return isIlluminated;
                } catch (ValidateKException e) {
                        throw new UserValidateAASException(e);
                } catch (RuntimeExceptionLamps e) {
                        throw new UserValidateAASException(e);
                }
        }

        /**
         * Computes a result string that can be displayed to the user.
         * @param room : {@link IRoom} instance that has to be checked.
         * @param isIlluminated : a boolean that represents whether the room is illuminated or not
         * @return
         */
        private static String computeResultString(IRoom room, Boolean isIlluminated) {
                String lineSeparator =  System.getProperty("line.separator");
                StringBuilder sb = new StringBuilder("The room ");
                sb.append(room.getID());
                String illuminatedOrNot = isIlluminated ? " is illuminated. " : " is NOT illuminated. ";
                sb.append(illuminatedOrNot);
                sb.append(lineSeparator);
                sb.append(room.printLampPositions());
                String outString = sb.toString();
                return outString;
        }

        /**
         * Can be used to get the result of the algorithm once it is available due to a prior call to
         * validate of the same instance.
         * @return A string that represents the result of the test.
         * @throws UserValidateAASException : e.g. if validate has not been called prior to this.
         */
        public String getResultString() throws UserValidateAASException{
                if (resultString != null) {
                        return resultString;
                } else {
                        throw new UserValidateAASException("No result available. Call validate first.");
                }

        }

        /**
         * Get runtime information.
         * @return a data structure of type {@link IRuntimeReader} that can be used to obtain runtime information.
         */
        public IRuntimeReader getRuntimeInformation() {
                return runtimeInfo;
        }

}
```

```java
package fernuni.propra.algorithm;

/**
 * An exception that is thrown if the search of potential candidates for lamp
positions fails due
 * to some unexpected error.
 * @author alex
 *
 */
public class CandidateSearcherException extends Exception {

        public CandidateSearcherException() {

        }

        public CandidateSearcherException(String message) {
                super(message);

        }

        public CandidateSearcherException(Throwable cause) {
                super(cause);

        }

}
```

```java
package fernuni.propra.algorithm.runtime_information;
/**
 * An interface that extends the extended interfaces so that implementing classes can declare
 * that they are capable of storing overall runtime information.
 * <p>
 * Implementing classes: {@link RuntimeInformation}
 * <p>
 * Extended interfaces: {@link IRuntimeCandidateSearcher}, {@link IRuntimePositionOptimizer},
 *                                      {@link IRuntimeIlluminationTester}, {@link IRuntimeReader}
 * <p>
 * @author alex
 *
 */
public interface IRuntimeInformation extends IRuntimeCandidateSearcher,
        IRuntimePositionOptimizer, IRuntimeIlluminationTester, IRuntimeReader{
        /**
         * Start the clock for the overall computation.
         * @throws RuntimeExceptionLamps : if not handled correctly
         */
        void startTime() throws RuntimeExceptionLamps;

        /**
         * Stop the clock for the overall computation.
         * @throws RuntimeExceptionLamps
         */
        void stopTime() throws RuntimeExceptionLamps;
}
```

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

```java
package fernuni.propra.algorithm.runtime_information;

/**
 * An interface that extends the extended interfaces so that implementing classes can declare
 * that they are capable of storing runtime information for the part of the algorithm that deals
 * with finding candidates for {@link Lamp} positions.
 * <p>
 * Implementing classes: {@link RuntimeInformation}
 * <p>
 * Extended interfaces: {@link IRuntimeOriginalPartialRectanglesFinder}
 * <p>
 * @author alex
 *
 */
public interface IRuntimeCandidateSearcher extends IRuntimeOriginalPartialRectanglesFinder{
        /**
         * Start the clock for the part of the algorithm that deals
         * with finding candidates for {@link Lamp} positions.
         * @throws RuntimeExceptionLamps : if not handled correctly
         */
        void startTimeCandidateSearch() throws RuntimeExceptionLamps;

        /**
         * Stop the clock for the part of the algorithm that deals
         * with finding candidates for {@link Lamp} positions.
         * @throws RuntimeExceptionLamps
         */
        void stopTimeCandidateSearch() throws RuntimeExceptionLamps;
}
```

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

```java
package fernuni.propra.algorithm.runtime_information;

/**
 * An interface that extends the extended interfaces so that implementing classes can declare
 * that they are capable of storing runtime information for the part of the algorithm that deals
 * with finding the original partial rectangles of the {@link IRoom}.
 * <p>
 * Implementing classes: {@link RuntimeInformation}
 * <p>
 * @author alex
 *
 */
public interface IRuntimeOriginalPartialRectanglesFinder {
        /**
         * Start the clock for the part of the algorithm that deals
         * with finding the original partial rectangles of the {@link IRoom}..
         * @throws RuntimeExceptionLamps : if not handled correctly
         */
        void startTimeOriginalPartialRectanglesFind() throws RuntimeExceptionLamps;

        /**
         * Stop the clock for the part of the algorithm that deals
         * with finding the original partial rectangles of the {@link IRoom}.
         * @throws RuntimeExceptionLamps : if not handled correctly
         */
        void stopTimeOriginalPartialRectanglesFind() throws RuntimeExceptionLamps;
}
```

```
package fernuni.propra.algorithm.runtime_information;

public interface IRuntimeReader {
        long getElapsedTimeCandidateSearch() throws RuntimeExceptionLamps;
        long getElapsedTimeOptimizePositions() throws RuntimeExceptionLamps;
        long getElapsedTimeOriginalPartialRectanglesFind() throws RuntimeExcep
tionLamps;
        long getElapsedTime() throws RuntimeExceptionLamps;
        long getElapsedTimeIlluminationTest() throws RuntimeExceptionLamps;
}
```

```java
package fernuni.propra.algorithm.runtime_information;

import java.util.concurrent.TimeUnit;

public class RuntimeInformation implements IRuntimeInformation, IRuntimeReader {
        private volatile long startTime = -1;
        private volatile long stopTime = -1;

        private volatile long candidateSearchStartTime = -1;
        private volatile long candidateSearchStopTime = -1;

        private volatile long originalPartialRectanglesFindStartTime = -1;
        private volatile long originalPartialRectanglesFindStopTime = -1;

        private volatile long optimizePositionsStartTime = -1;
        private volatile long optimizePositionsStopTime = -1;

        private volatile long illuminationTestStartTime = -1;
        private volatile long illuminationTestStopTime = -1;


        @Override
        public void startTimeCandidateSearch() throws RuntimeExceptionLamps {
                if (candidateSearchStartTime != -1 && candidateSearchStopTime != -1) {
                        throw new RuntimeExceptionLamps();
                }
                candidateSearchStartTime = System.nanoTime();

        }

        @Override
        public void stopTimeCandidateSearch() throws RuntimeExceptionLamps {
                if (candidateSearchStartTime == -1 || candidateSearchStopTime != -1) {
                        throw new RuntimeExceptionLamps();
                }
                candidateSearchStopTime = System.nanoTime();

        }

        @Override
        public long getElapsedTimeCandidateSearch() throws RuntimeExceptionLamps {
                if (candidateSearchStartTime == -1 && candidateSearchStopTime == -1) {
                        throw new RuntimeExceptionLamps();
                }
                return candidateSearchStopTime-candidateSearchStartTime;
        }



        @Override
        public void startTimeOriginalPartialRectanglesFind() throws RuntimeExceptionLamps {
                if (originalPartialRectanglesFindStartTime != -1 && originalPartialRectanglesFindStopTime != -1) {
                        throw new RuntimeExceptionLamps();
                }
                originalPartialRectanglesFindStartTime = System.nanoTime();

        }

        @Override
        public void stopTimeOriginalPartialRectanglesFind() throws RuntimeExceptionLamps {
                if (originalPartialRectanglesFindStartTime == -1 || originalPartialRectanglesFindStopTime != -1) {
                        throw new RuntimeExceptionLamps();
                }
                originalPartialRectanglesFindStopTime = System.nanoTime();

        }

        @Override
        public long getElapsedTimeOriginalPartialRectanglesFind() throws RuntimeExceptionLamps {
                if (originalPartialRectanglesFindStartTime == -1 && originalPartialRectanglesFindStopTime == -1) {
                        throw new RuntimeExceptionLamps();
                }
                return originalPartialRectanglesFindStopTime-originalPartialRectanglesFindStartTime;
        }


        @Override
        public void startTimeOptimizePositions() throws RuntimeExceptionLamps {
                if (optimizePositionsStartTime != -1 && optimizePositionsStopTime != -1) {
                        throw new RuntimeExceptionLamps();
                }
                optimizePositionsStartTime = System.nanoTime();

        }

        @Override
        public void stopTimeOptimizePositions() throws RuntimeExceptionLamps {
                if (optimizePositionsStartTime == -1 || optimizePositionsStopTime != -1) {
                        throw new RuntimeExceptionLamps();
                }
                optimizePositionsStopTime = System.nanoTime();

        }

        @Override
        public long getElapsedTimeOptimizePositions() throws RuntimeExceptionLamps {
                if (optimizePositionsStartTime == -1 && optimizePositionsStopTime == -1) {
                        throw new RuntimeExceptionLamps();
                }
                return optimizePositionsStopTime-optimizePositionsStartTime;
        }

        @Override
        public void resetTimeOptimizePositions() {
                optimizePositionsStartTime = -1;
                optimizePositionsStopTime = -1;

        }

        @Override
        public void startTimeIlluminationTest() throws RuntimeExceptionLamps {
```

```java
		if (illuminationTestStartTime != -1 && illuminationTestStopTime != -1) {
			throw new RuntimeExceptionLamps();
		}
		illuminationTestStartTime = System.nanoTime();

	}

	@Override
	public void stopTimeIlluminationTest() throws RuntimeExceptionLamps {
		if (illuminationTestStartTime == -1 || illuminationTestStopTime != -1) {
			throw new RuntimeExceptionLamps();
		}
		illuminationTestStopTime = System.nanoTime();

	}

	@Override
	public long getElapsedTimeIlluminationTest() throws RuntimeExceptionLamps {
		if (illuminationTestStartTime == -1 && illuminationTestStopTime == -1) {
			throw new RuntimeExceptionLamps();
		}
		return illuminationTestStopTime-illuminationTestStopTime;
	}


	@Override
	public void startTime() throws RuntimeExceptionLamps {
		if (startTime != -1 && stopTime != -1) {
			throw new RuntimeExceptionLamps();
		}
		startTime = System.nanoTime();

	}

	@Override
	public void stopTime() throws RuntimeExceptionLamps {
		if (startTime == -1 || stopTime != -1) {
			throw new RuntimeExceptionLamps();
		}
		stopTime = System.nanoTime();

	}

	@Override
	public long getElapsedTime() throws RuntimeExceptionLamps {
		if (startTime == -1 && stopTime == -1) {
			throw new RuntimeExceptionLamps();
		}
		return stopTime-startTime;
	}



	@Override
	public String toString() {
		String lineSeparator = System.getProperty("line.separator");
		StringBuilder sb = new StringBuilder("Runtime Information");
		sb.append(lineSeparator);
		sb.append("Total runtime: ");
		try {
			sb.append((double) Math.round((double) getElapsedTime() / 1_000_000_000 * 100)/100);
			sb.append(" s,");
		} catch (RuntimeExceptionLamps e) {
			sb.append("not available");
		}
		sb.append(lineSeparator);
		sb.append("thereof ");
		sb.append(lineSeparator);

		sb.append("searching for lamp position candidates: ");
		try {
			sb.append((double) Math.round((double) getElapsedTimeCandidateSearch() / 1_000_000_000 * 100)/100);
			sb.append(" s.");
		} catch (RuntimeExceptionLamps e) {
			sb.append("not available");
		}
		sb.append(lineSeparator);


		sb.append("optimizing lamp positions: ");
		try {
			sb.append((double) Math.round((double) getElapsedTimeOptimizePositions() / 1_000_000_000 * 100)/100);
			sb.append(" s.");
		} catch (RuntimeExceptionLamps e) {
			sb.append("not available");
		}
		sb.append(lineSeparator);

		sb.append("testing if room is illuminated: ");
		try {
			sb.append((double) Math.round((double) getElapsedTimeIlluminationTest() / 1_000_000_000 * 100)/100);
			sb.append(" s.");
		} catch (RuntimeExceptionLamps e) {
			sb.append("not available");
		}
		sb.append(lineSeparator);

		sb.append("constructing original partial rectangles: ");
		try {
			sb.append((double) Math.round((double) getElapsedTimeOriginalPartialRectanglesFind() / 1_000_000_000 * 100)/100);
			sb.append(" s.");
		} catch (RuntimeExceptionLamps e) {
			sb.append("not available");
		}

		String outString = sb.toString();
		return outString;
	}

}
```

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

```java
package fernuni.propra.algorithm.runtime_information;
/**
 * An interface that extends the extended interfaces so that implementing clas
ses can declare
 * that they are capable of storing runtime information for the part of the al
gorithm that deals
 * with checking whether an {@link IRoom} is illuminated.
 * <p>
 * Implementing classes: {@link RuntimeInformation}
 * <p>
 * Extended interfaces: {@link IRuntimeOriginalPartialRectanglesFinder}
 * <p>
 * @author alex
 *
 */
public interface IRuntimeIlluminationTester extends IRuntimeOriginalPartialRec
tanglesFinder{
        /**
         * Start the clock for the part of the algorithm that deals
         * with checking whether an {@link IRoom} is illuminated.
         * @throws RuntimeExceptionLamps : if not handled correctly
         */
        void startTimeIlluminationTest() throws RuntimeExceptionLamps;

        /**
         * Stop the clock for the part of the algorithm that deals
         * with checking whether an {@link IRoom} is illuminated.
         * @throws RuntimeExceptionLamps
         */
        void stopTimeIlluminationTest() throws RuntimeExceptionLamps;
}
```

```java
package fernuni.propra.algorithm.runtime_information;

/**
 * Thrown if a data structure for the storing of runtime information, e.g. {@link RuntimeInformation}, is used incorrectly.
 * E.g. if methods are called in a wrong order (time is stopped before a clock is started.
 * @author alex
 *
 */
public class RuntimeExceptionLamps extends Exception {

        public RuntimeExceptionLamps() {
                // TODO Auto-generated constructor stub
        }

        public RuntimeExceptionLamps(String message) {
                super(message);
                // TODO Auto-generated constructor stub
        }

        public RuntimeExceptionLamps(Throwable cause) {
                super(cause);
                // TODO Auto-generated constructor stub
        }


}
```

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

```
package fernuni.propra.algorithm.runtime_information;

public interface IRuntimePositionOptimizer extends IRuntimeIlluminationTester
{
        void startTimeOptimizePositions() throws RuntimeExceptionLamps;
        void stopTimeOptimizePositions() throws RuntimeExceptionLamps;
        void resetTimeOptimizePositions();

}
```

```java
package fernuni.propra.algorithm;

import java.util.Iterator;

import fernuni.propra.algorithm.runtime_information.IRuntimeIlluminationTester;
import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;

/**
 * A controll class that controls the program flow for the validation of an {@link IRoom} instance,
 * i.e. for the test whether that {@link IRoom} is illuminated by its associated {@link Lamp}s or not.
 * <p>
 * Delegates the algorithm to an instance of {@link IIlluminationTester}, which is obtained from a call to
 * the {@link AbstractAlgorithmFactory}.
 * <p>
 * Furthermore a data structure is given to {@link ValidateK} in order to store runtime information to that
 * data structure.
 * <p>
 * @author alex
 *
 */
public  class ValidateK {
        /**
         * Checks whether an {@link IRoom} instance is illuminated or not by delegating to {@link IIlluminationTester}.
         * @param room : the {@link IRoom} to be checked
         * @param runtimeInfo : {@link IRoom} the data structure to which {@link ValidateK} will write runtime information.
         * @return
         * @throws ValidateKException
         */
        boolean validate(IRoom room, IRuntimeIlluminationTester runtimeInfo) throws ValidateKException {
                try {
                        // turn all lamps on
                        Iterator<Lamp> lampIterator = room.getLamps();
                        while(lampIterator.hasNext()) {
                                lampIterator.next().turnOn();
                        }
                        runtimeInfo.startTimeIlluminationTest();
                        boolean isIlluminated = AbstractAlgorithmFactory.getAlgorithmFactory().createIlluminiationTester().testIfRoomIsIlluminated(room, runtimeInfo);
                        runtimeInfo.stopTimeIlluminationTest();
                        return isIlluminated;
                } catch (IlluminationTesterException e) {
                        throw new ValidateKException(e);
                } catch( RuntimeExceptionLamps rte) {
                        throw new ValidateKException(rte);
                }
        }



}
```

```java
package fernuni.propra.algorithm;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;

import fernuni.propra.algorithm.runtime_information.IRuntimeOriginalPartialRectanglesFinder;
import fernuni.propra.algorithm.util.Rectangle;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.LineSegment;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;
/**
 * Provides an algorithm to computes all orginal partial rectangles of an {@link IRoom} instance and to tag
 * these rectangles with tags that correspond to a portion of the {@link IRoom} that is illuminated if the associated
 * partial rectangle is illuminated.
 * <p>
 * The tags/portions correspond to {@link Wall}s of the {@link IRoom} that are
 * illuminated if the associated rectangle is illuminated.
 * <p>
 * Implemented interfaces : {@link IOriginalPartialRectanglesFinder}
 * <p>
 *
 * @author alex
 *
 */
public class OriginalPartialRectanglesFinder implements IOriginalPartialRectanglesFinder{

        private static double findWallTOL = 0.001; // necessary in order to correctly determine original partial rectangles TODO set it to size of room?
        private HashSet<Integer> allTags = new HashSet<Integer>(); // all portions of the room
        private WallContainerEast wallContainerEast  = new WallContainerEast();
        private WallContainerNorth wallContainerNorth = new WallContainerNorth();
        private WallContainerWest wallContainerWest = new WallContainerWest();
        private WallContainerSouth wallContainerSouth = new WallContainerSouth();
        private HashSet<Rectangle> originalRectangles = new HashSet<Rectangle>(); // all original partial rectangles of the room
        private ArrayList<RectangleWithTag> originalRectanglesTagged = new ArrayList<RectangleWithTag>(); // all tagged tagged original partial rectangles

        @Override
        public ArrayList<RectangleWithTag> findOriginalPartialRectangles(IRoom room, IRuntimeOriginalPartialRectanglesFinder rti) throws OriginalPartialRectanglesFinderException {
                try {
                        sortWallsToContainers(room);
                        constructOriginalPartialRectangles();
                } catch (WallContainerException | OriginalPartialRectanglesFinderException e) {
                        throw new OriginalPartialRectanglesFinderException(e);
                }

                return originalRectanglesTagged;
        }

        @Override
        public HashSet<Integer> getAllTags() {
                return allTags;

        }

        /**
         * Sorts all walls of the {@link IRoom} instance to suited containers depending on their orientation
         * <p>
         * Package access is granted for testing purposes.
         * <p>
         * @param room : The {@link IRoom} for which the {@link Wall}s need to be sorted
         * @throws OriginalPartialRectanglesFinderException : thrown if wall orientation of a wall of the {@link IRoom} cannot be
         *                  determined
         */
        void sortWallsToContainers(IRoom room)  throws OriginalPartialRectanglesFinderException {
                Iterator<Wall> wallIterator = room.getWalls();
                try {
                        while(wallIterator.hasNext()) {
                                Wall nextWall = wallIterator.next();
                                if (nextWall.isEastWall()) {
                                        wallContainerEast.add(nextWall);
                                } else if (nextWall.isNorthWall()) {
                                        wallContainerNorth.add(nextWall);
                                } else if (nextWall.isWestWall()) {
                                        wallContainerWest.add(nextWall);
                                } else if (nextWall.isSouthWall()) {
                                        wallContainerSouth.add(nextWall);
                                } else {
                                        throw new OriginalPartialRectanglesFinderException("Wall orientation cannot be determined! Wall might not be horizontal or vertical");
                                }
                        }
                } catch (WallContainerException wce) {
                        throw new OriginalPartialRectanglesFinderException(wce.getMessage());
                }

        }

        /**
         * Constructs the original partial rectangles for an {@link IRoom} and tags the rectangles with
         * the {@link Wall} indices that correspond to {@link Wall}s that are illuminated if the rectangle
         * is illuminated.
         * <p>
         *  Package access for testing purposes.
         * <p>
         * {@link sortWallsToContainers} needs to be called first.
         * <p>
         * @throws WallContainerException : {@link Wall} handling does not work.
         */
        void constructOriginalPartialRectangles() throws WallContainerException {

                // construct original partial rectangles for each north wall
```

```java
		for( Wall northWall : wallContainerNorth) {
			double yNorth = northWall.getP1().getY();

			// find west wall
			double westXLimit = northWall.getP2().getX();
			Wall nextWestWall = wallContainerWest.getNearestWall(yNorth - findWallTOL,
					yNorth - findWallTOL, westXLimit);
			// find east wall
			double eastXLimit = northWall.getP1().getX();
			Wall nextEastWall = wallContainerEast.getNearestWall(yNorth - findWallTOL, yNorth - findWallTOL, eastXLimit);

			// find south wall
			double xWest = nextWestWall.getP1().getX();
			double xEast = nextEastWall.getP1().getX();
			Wall nextSouthWall = wallContainerSouth.getNearestWall(xWest+findWallTOL, xEast-findWallTOL, yNorth);
			double ySouth = nextSouthWall.getP1().getY();

			// add this original partial rectangle
			addOriginalPartialRectangle(yNorth, xWest, xEast, ySouth);
		}

		// construct original partial rectangles for each east wall
		for (Wall eastWall: wallContainerEast) {

			// find north and south wall
			double xEast = eastWall.getP1().getX();
			double southYLimit = eastWall.getP1().getY();
			double northYLimit = eastWall.getP2().getY();

			Wall nextSouthWall = wallContainerSouth.getNearestWall(xEast- findWallTOL, xEast - findWallTOL, southYLimit);
			Wall nextNorthWall = wallContainerNorth.getNearestWall(xEast- findWallTOL, xEast - findWallTOL, northYLimit);

			// find west wall
			double ySouth = nextSouthWall.getP1().getY();
			double yNorth = nextNorthWall.getP1().getY();

			Wall nextWestWall = wallContainerWest.getNearestWall(ySouth+findWallTOL, yNorth-findWallTOL, xEast);
			double xWest = nextWestWall.getP1().getX();

			// add this original partial rectangle
			addOriginalPartialRectangle(yNorth, xWest, xEast, ySouth);
		}

		// construct original partial rectangles for each west wall
		for (Wall westWall: wallContainerWest) {

			// find north and south wall
			double xWest = westWall.getP1().getX();
			double southYLimit = westWall.getP2().getY();
			double northYLimit = westWall.getP1().getY();

			Wall nextSouthWall = wallContainerSouth.getNearestWall(xWest + findWallTOL, xWest + findWallTOL, southYLimit);
			Wall nextNorthWall = wallContainerNorth.getNearestWall(xWest + findWallTOL, xWest + findWallTOL, northYLimit);

			// find east wall
			double ySouth = nextSouthWall.getP1().getY();
			double yNorth = nextNorthWall.getP1().getY();

			Wall nextEastWall = wallContainerEast.getNearestWall(ySouth+findWallTOL, yNorth-findWallTOL, xWest);
			double xEast = nextEastWall.getP1().getX();

			// add this original partial rectangle
			addOriginalPartialRectangle(yNorth, xWest, xEast, ySouth);
		}

		// construct original partial rectangles for each south wall
		for (Wall southWall: wallContainerSouth) {

			// find east and west walls
			double ySouth = southWall.getP1().getY();
			double eastXLimit = southWall.getP2().getX();
			double westXLimit = southWall.getP1().getX();

			Wall nextEastWall = wallContainerEast.getNearestWall(ySouth + findWallTOL, ySouth + findWallTOL, eastXLimit);
			Wall nextWestWall = wallContainerWest.getNearestWall(ySouth + findWallTOL, ySouth + findWallTOL, westXLimit);

			double xEast = nextEastWall.getP1().getX();
			double xWest = nextWestWall.getP1().getX();

			// find north wall
			Wall nextNorthWall = wallContainerNorth.getNearestWall(xWest+findWallTOL, xEast-findWallTOL, ySouth);
			double yNorth = nextNorthWall.getP1().getY();

			// add this original partial rectangle
			addOriginalPartialRectangle(yNorth, xWest, xEast, ySouth);

		}
	}
	/**
	 * Finds tags for original rectangles and adds it to global original rectangles if this rectangle does
	 * not already exist
	 * @param yNorth
	 * @param xWest
	 * @param xEast
	 * @param ySouth
	 */
	private void addOriginalPartialRectangle(double yNorth, double xWest, double xEast, double ySouth) {
		Point southWestCorner = new Point(xWest,ySouth);
		Point northEastCorner = new Point(xEast,yNorth);
		Rectangle partialRectangle = new Rectangle(southWestCorner, northEastCorner);


		if (!originalRectangles.contains(partialRectangle)) { // same rectangle does not already exist -> add
			// determine all tags
			HashSet<Integer> allWallsCoveredByRectangleBoundary = findTagsOfAllCoveredWalls(partialRectangle);
			// add to rectangles
			originalRectangles.add(partialRectangle);
			// add to partial rectangles
			originalRectanglesTagged.add(new RectangleWithTag(partialRectangle, allWallsCoveredByRectangleBoundary));
			// include all tags of this new rectangle to the tags of the room
			allTags.addAll(allWallsCoveredByRectangleBoundary);
		}
	}

	/**
	 * Finds the tags of all covered walls of this original partialRectangle
	 * @param partialRectangle : the original partial rectangle to be checked
	 * @return a set of tags that corresponds to the tags of the {@link Wall}s illuminated by that rectangle
	 */
	private HashSet<Integer>  findTagsOfAllCoveredWalls(Rectangle partialRectangle) {
		HashSet<Integer> allWallsCoveredByRectangleBoundary = new HashSet<Integer>();

		//check east walls
		LineSegment eastWall = new LineSegment(partialRectangle.getP2(), partialRectangle.getP3());
		for (Wall wall : wallContainerEast) {
			if (wall.getP1().isOnLineSegment(eastWall) && wall.getP2().isOnLineSegment(eastWall)) { // wall of current rectangles covers a wall of of the room -> if the rectangle is illuminated then this wall is also illuminated
				allWallsCoveredByRectangleBoundary.add(wall.getTag());
				//allTags.add(wall.getTag());
			}
		}

		//check north walls
		LineSegment northWall = new LineSegment(partialRectangle.getP3(), partialRectangle.getP4());
		for (Wall wall : wallContainerNorth) {
			if (wall.getP1().isOnLineSegment(northWall) && wall.getP2().isOnLineSegment(northWall)) { // wall of current rectangles covers a wall of of the room -> if the rectangle is illuminated then this wall is also illuminated
				allWallsCoveredByRectangleBoundary.add(wall.getTag());
				//allTags.add(wall.getTag());
			}
		}

		//check west walls
		LineSegment westWall = new LineSegment(partialRectangle.getP4(), partialRectangle.getP1());
		for (Wall wall : wallContainerWest) {
			if (wall.getP1().isOnLineSegment(westWall) && wall.getP2().isOnLineSegment(westWall)) { // wall of current rectangles covers a wall of of the room -> if the rectangle is illuminated then this wall is also illuminated
				allWallsCoveredByRectangleBoundary.add(wall.getTag());
				//allTags.add(wall.getTag());
			}
		}

		//check south walls
		LineSegment southWall = new LineSegment(partialRectangle.getP1(), partialRectangle.getP2());
		for (Wall wall : wallContainerSouth) {
			if (wall.getP1().isOnLineSegment(southWall) && wall.getP2().isOnLineSegment(southWall)) { // wall of current rectangles covers a wall of of the room -> if the rectangle is illuminated then this wall is also illuminated
				allWallsCoveredByRectangleBoundary.add(wall.getTag());
				//allTags.add(wall.getTag());
			}
		}

		return allWallsCoveredByRectangleBoundary;
	}

	// for testing
	public Iterator<RectangleWithTag> iteratorOriginalRectangles() {
		return originalRectanglesTagged.iterator();
	}

	// for testing
	Iterator<Wall> eastIterator() {
		return wallContainerEast.iterator();
	}

	Iterator<Wall> northIterator() {
		return wallContainerNorth.iterator();
	}

	Iterator<Wall> westIterator() {
		return wallContainerWest.iterator();
	}

	Iterator<Wall> southIterator() {
		return wallContainerSouth.iterator();
	}




}
```

```java
package fernuni.propra.algorithm;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;

import fernuni.propra.algorithm.runtime_information.IRuntimeOriginalPartialRectanglesFinder;
import fernuni.propra.algorithm.util.Rectangle;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;

/**
 * tags all original partial rectangles with consecutive numbers
 * @author alex
 *
 */
public class OriginalPartialRectanglesFinder2 implements IOriginalPartialRectanglesFinder{

	private static double findWallTOL = 0.001;
	private HashSet<Integer> allTags = new HashSet<Integer>();
	private WallContainerEast wallContainerEast  = new WallContainerEast();
	private WallContainerNorth wallContainerNorth = new WallContainerNorth();
	private WallContainerWest wallContainerWest = new WallContainerWest();
	private WallContainerSouth wallContainerSouth = new WallContainerSouth();
	private HashSet<Rectangle> originalRectangles = new HashSet<Rectangle>();
	private ArrayList<RectangleWithTag> originalRectanglesTagged = new ArrayList<RectangleWithTag>();

	@Override
	public ArrayList<RectangleWithTag> findOriginalPartialRectangles(IRoom room, IRuntimeOriginalPartialRectanglesFinder rti) throws OriginalPartialRectanglesFinderException {
		try {
			sortWallsToContainers(room);
			constructOriginalPartialRectangles();
		} catch (WallContainerException | OriginalPartialRectanglesFinderException e) {
			throw new OriginalPartialRectanglesFinderException(e);
		}

		return originalRectanglesTagged;
	}

	@Override
	public HashSet<Integer> getAllTags() {
		return allTags;

	}

	void sortWallsToContainers(IRoom room) throws WallContainerException, OriginalPartialRectanglesFinderException {
		Iterator<Wall> wallIterator = room.getWalls();
		while(wallIterator.hasNext()) {
			Wall nextWall = wallIterator.next();
			if (nextWall.isEastWall()) {
				wallContainerEast.add(nextWall);
			} else if (nextWall.isNorthWall()) {
				wallContainerNorth.add(nextWall);
			} else if (nextWall.isWestWall()) {
				wallContainerWest.add(nextWall);
			} else if (nextWall.isSouthWall()) {
				wallContainerSouth.add(nextWall);
			} else {
				throw new OriginalPartialRectanglesFinderException("Wall orientation cannot be determined! Wall might not be horizontal or vertical");
			}
		}
	}

	void constructOriginalPartialRectangles() throws WallContainerException {

		int rectangleNo = 0;

		for( Wall northWall : wallContainerNorth) {
			double yNorth = northWall.getP1().getY();
			double westXLimit = northWall.getP2().getX();
			double eastXLimit = northWall.getP1().getX();

			Wall nextWestWall = wallContainerWest.getNearestWall(yNorth - findWallTOL,
					yNorth - findWallTOL, westXLimit);

			Wall nextEastWall = wallContainerEast.getNearestWall(yNorth - findWallTOL, yNorth - findWallTOL, eastXLimit);

			double xWest = nextWestWall.getP1().getX();
			double xEast = nextEastWall.getP1().getX();

			Wall nextSouthWall = wallContainerSouth.getNearestWall(xWest+findWallTOL, xEast-findWallTOL, yNorth);
			double ySouth = nextSouthWall.getP1().getY();


			rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
		}

		for (Wall eastWall: wallContainerEast) {
			double xEast = eastWall.getP1().getX();
			double southYLimit = eastWall.getP1().getY();
			double northYLimit = eastWall.getP2().getY();

			Wall nextSouthWall = wallContainerSouth.getNearestWall(xEast- findWallTOL, xEast - findWallTOL, southYLimit);
			Wall nextNorthWall = wallContainerNorth.getNearestWall(xEast- findWallTOL, xEast - findWallTOL, northYLimit);

			double ySouth = nextSouthWall.getP1().getY();
			double yNorth = nextNorthWall.getP1().getY();

			Wall nextWestWall = wallContainerWest.getNearestWall(ySouth+findWallTOL, yNorth-findWallTOL, xEast);
			double xWest = nextWestWall.getP1().getX();

			rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
		}

```

```java

                for (Wall westWall: wallContainerWest) {
                        double xWest = westWall.getP1().getX();
                        double southYLimit = westWall.getP2().getY();
                        double northYLimit = westWall.getP1().getY();

                        Wall nextSouthWall = wallContainerSouth.getNearestWall(xWest + findWallTOL, xWest + findWallTOL, southYLimit);
                        Wall nextNorthWall = wallContainerNorth.getNearestWall(xWest + findWallTOL, xWest + findWallTOL, northYLimit);

                        double ySouth = nextSouthWall.getP1().getY();
                        double yNorth = nextNorthWall.getP1().getY();

                        Wall nextEastWall = wallContainerEast.getNearestWall(ySouth+findWallTOL, yNorth-findWallTOL, xWest);
                        double xEast = nextEastWall.getP1().getX();

                        rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
                }

                for (Wall southWall: wallContainerSouth) {
                        double ySouth = southWall.getP1().getY();
                        double eastXLimit = southWall.getP2().getX();
                        double westXLimit = southWall.getP1().getX();

                        Wall nextEastWall = wallContainerEast.getNearestWall(ySouth + findWallTOL, ySouth + findWallTOL, eastXLimit);
                        Wall nextWestWall = wallContainerWest.getNearestWall(ySouth + findWallTOL, ySouth + findWallTOL, westXLimit);

                        double xEast = nextEastWall.getP1().getX();
                        double xWest = nextWestWall.getP1().getX();

                        Wall nextNorthWall = wallContainerNorth.getNearestWall(xWest+findWallTOL, xEast-findWallTOL, ySouth);
                        double yNorth = nextNorthWall.getP1().getY();

                        rectangleNo = addOriginalPartialRectangle(rectangleNo, yNorth, xWest, xEast, ySouth);
                }
        }

        private int addOriginalPartialRectangle(int rectangleNo,double yNorth, double xWest, double xEast, double ySouth) {
                Point southWestCorner = new Point(xWest,ySouth);
                Point northEastCorner = new Point(xEast,yNorth);
                Rectangle partialRectangle = new Rectangle(southWestCorner, northEastCorner);
                if (!originalRectangles.contains(partialRectangle)) { // same rectangle does not already exist -> add
                        int tag = rectangleNo++;
                        originalRectangles.add(partialRectangle);
                        originalRectanglesTagged.add(new RectangleWithTag(partialRectangle, tag));
                        allTags.add(tag);
                }
                return rectangleNo;
        }

        public Iterator<RectangleWithTag> iteratorOriginalRectangles() {
                return originalRectanglesTagged.iterator();
        }

        // TODO for tests
        Iterator<Wall> eastIterator() {
                return wallContainerEast.iterator();
        }

        Iterator<Wall> northIterator() {
                return wallContainerNorth.iterator();
        }

        Iterator<Wall> westIterator() {
                return wallContainerWest.iterator();
        }

        Iterator<Wall> southIterator() {
                return wallContainerSouth.iterator();
        }




}
```

```java
package fernuni.propra.algorithm;

/**
 * Thrown if an unexpected error occured during solution.
 * <p>
 * @author alex
 *
 */
public class SolveKException extends Exception {

	public SolveKException() {
		super();
	}

	public SolveKException(String message) {
		super(message);
	}

	public SolveKException(Throwable cause) {
		super(cause);
	}

}
```

```java
package fernuni.propra.algorithm;

/**
 * An abstract factory that configures the solution algorithm by instantiating specific instances of the
 * interfaces {@link ICandidateSearcher}, {@link IPositionOptimizer} and {@link IIlluminationTester} that are
 * to be used within the algorithm.
 * <p>
 * The {@link AlgorithmFactory1} implements the "abstract factory" (concrete factory) and "singleton" design patterns.
 * <p>
 *
 * Implemented interfaces and super classes: {@link AbstractAlgorithmFactory}
 *
 * @author alex
 *
 */
public class AlgorithmFactory1 extends AbstractAlgorithmFactory{
        private static AlgorithmFactory1 singleton;

        private AlgorithmFactory1() {};

        static AlgorithmFactory1 getAlgorithmFactory1() {
                if (singleton == null) {
                        singleton = new AlgorithmFactory1();
                }
                return singleton;
        }

        @Override
        public ICandidateSearcher createCandidateSearcher() {
                return new CandidateSearcher();
        }

        @Override
        public IPositionOptimizer createPositionOptimizer() {
                return new PositionOptimizer();
        }

        @Override
        public IIlluminationTester createIlluminiationTester() {
                return new IlluminationTester();
        }

        @Override
        public IOriginalPartialRectanglesFinder createOriginalPartialRectanglesFinder() {
                return new OriginalPartialRectanglesFinder();
        }

}
```

```java
package fernuni.propra.algorithm;

import java.util.List;

import fernuni.propra.algorithm.runtime_information.IRuntimeInformation;
import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;

// computes best solution in given time limit and replaces lamps with found best solution in room

/**
 * A control class that controls the program flow for the solution use case {@link UserSolveAAS}. It computes the
 * configuration of {@link Lamp}s that has a minimum number of illuminated {@link Lamp}s and still illuminates
 * a specific {@link IRoom} instance.
 * <p>
 * {@link SolveK} forwards certain tasks to instances of other classes.
 * <p>
 * The general algorithm works as follows:
 * <p>
 * 1.) A number of possible {@link Lamp} candidates is computed
 *     by forwarding to an instance of {@link ICandidateSearcher} that is obtained from
 *     the {@link AbstractAlgorithmFactory} singleton.
 * <p>
 * 2.) The candidates are then provided to an instance of {@link IPositionOptimizer} that is also
 *     obtained from the {@link AbstractAlgorithmFactory} singleton. The {@link IPositionOptimizer}
 *     finds an optimal configuration of lamp positions.
 * <p>
 * 3.) The currently available best solution replaces the lamps of the provided {@link IRoom} instance
 *     and the number of illuminated {@link Lamp}s in that best solution is stored.
 * <p>
 * 4.) The number of illuminated {@link Lamps} in best solution can be obtained.
 * <p>
 *{@link SolveK} extends {@link Thread} in order to make it possible to stop the computation externally (e.g. after
 * a time limit has passed) by calling the interrupt() method of an instance of {@link SolveK}. This sets the
 * interrupted Flag which is checked at several stages in the solve method. If the flag is set, an {@link InterruptedException}
 * is thrown, which leads to a stop of the algorithm. If a valid solution is available, this solution is stored as noted in 4.)
 * <p>
 * The computation must be started by calling the inherited start() method.
 * <p>
 * {@link SolveK} also supports the wait-notify mechanism for synchronizing, by allowing to let clients test whether the solution is already
 * available and waiting until the computation is done. So the algorithm provided can be started externally, and subsequently a client can wait until
 * 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
 * prematurely interrupts {@link SolveK}). Therefore, clients should check the return value of
 * {@link testIfComputationIsFinished} for an exception. Only if this return value is null, no exception has
 * occurred and a valid solution has been stored to the {@link IRoom} instance.
 * <p>
 *
 * <p>
 * Implemented superclasses : {@link Thread}
 * @author alex
 *
 */
public class SolveK extends Thread{
        private IRuntimeInformation runTimeInformation;
        private IRoom room;
        private ICandidateSearcher candidateSearcher;
        private IPositionOptimizer positionOptimizer;
        private boolean computationFinished;
        private volatile List<Lamp> bestSolution;
        private volatile int numberLampsOnBestSolution;
        private volatile SolveKException exception = null; // to be communicated to main thread

        /**
         * Constructor for {@link SolveK}
         * @param room : An {@link IRoom} instance for which an optimal {@link Lamp} configuration has
         *                                 to be found. The {@link Lamp} configuration is stored to this {@link IRoom} instance
         * @param runTimeInformation : An instance of {@link IRuntimeInformation} to which detailed runtime information can be stored.
         */
        public SolveK(IRoom room, IRuntimeInformation runTimeInformation) {
                this.room = room;
                this.runTimeInformation = runTimeInformation;
                this.candidateSearcher = AbstractAlgorithmFactory.getAlgorithmFactory().createCandidateSearcher();
                this.positionOptimizer = AbstractAlgorithmFactory.getAlgorithmFactory().createPositionOptimizer();
        }

        /**
         * Provides the steps 1.) and 2.) of the algorithm specified in the documentation of {@link SolveK}
         * @param runTimeInformation : An instance of {@link IRuntimeInformation} that can be used to save runtime information
         * @throws SolveKException : thrown if an unexpected error is thrown in the solution procedure
         * @throws InterruptedException : thrown if the solution procedure is interrupted. This is done intentionally to allow
         *                                  interruption by clients and is catched in run().
         */
        private void solve(IRuntimeInformation runTimeInformation) throws SolveKException, InterruptedException{
                List<Lamp> candidates;
                try {
                        // 1.) search lamp candidates
                        runTimeInformation.startTimeCandidateSearch();
                        try {
                                candidates = candidateSearcher.searchCandidates(room,
                                                runTimeInformation);
                        } catch(InterruptedException ie) {
                                runTimeInformation.stopTimeCandidateSearch();
                                throw new InterruptedException(ie.getMessage());
```

```java
            }
            runTimeInformation.stopTimeCandidateSearch();
            System.out.println("Number of candidates found: " + candidates.size());

            // 2.) find optimal configuration of lamps
            runTimeInformation.startTimeOptimizePositions();
            try {
                positionOptimizer.optimizePositions(
                        candidates, runTimeInformation);
            } catch (InterruptedException ie) {
                runTimeInformation.stopTimeOptimizePositions();
                throw new InterruptedException(ie.getMessage());
            }
            runTimeInformation.stopTimeOptimizePositions();

        } catch (CandidateSearcherException e) {
            throw new SolveKException(e); // something went wrong
        } catch (RuntimeExceptionLamps rte) {
            throw new SolveKException(rte); // something went wrong
        }
    }

    @Override
    public void run() {
        try {
            solve(runTimeInformation);
        } catch(SolveKException e) {
            this.exception = e;
        } catch(InterruptedException ie) {
        }

        //write output and set best solution
        bestSolution = positionOptimizer.getCurrentBestSolution();
        numberLampsOnBestSolution = positionOptimizer.getNumberOfOnLampsBestSolution();
        if(bestSolution != null) { // null if interrupted or exception at candidate searcher -> no solution available
            // optimal configuration is supplied to room
            room.replaceLamps(bestSolution);
        } else {
            exception = new SolveKException("Not enough time to compute a solution!"); // exceptions to be passed to client
        }
        setComputationFinished(true);
    }

    /**
     * Sets switch and also notifies those threads waiting for a result in the waiting queue
     * @param computationFinished
     */
    private synchronized void setComputationFinished(boolean computationFinished) {
        this.computationFinished = computationFinished;
        notifyAll();
    }

    /**
     * Allows clients to join the waiting queue and wait for a result
     * @return {@link SolveKException} that shows if a solution is available (return value is null) or not
     * @throws InterruptedException
     */
    public synchronized SolveKException testIfComputationFinished() throws InterruptedException{
        while(!computationFinished) {
            wait();
        }
        return exception;
    }

    /**
     * Clients can obtain the number of lamps that are turned on in the best solution.
     * @return Number of lamps that are turned on in best solution.
     */
    public synchronized int getNumberOfOnLampsBestSolution() {
        return numberLampsOnBestSolution;
    }


}
```

```java
package fernuni.propra.algorithm;

import java.util.HashSet;
import java.util.Iterator;
import java.util.List;

import fernuni.propra.algorithm.runtime_information.IRuntimeIlluminationTester;
import fernuni.propra.algorithm.runtime_information.RuntimeExceptionLamps;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;

/**
 * A specific provider of an algorithm that can check whether a room is illuminated by a set of {@link Lamp}s
 * or not.
 * <p>
 * Several methods are provided for this purpose.
 * <p>
 * The algorithm for {@link testIfRoomIsIlluminated}({@link IRoom} room, {@link IRuntimeIlluminationTester} runtimeInfo) works as follows:
 * <p>
 * 1.) Find all original partial rectangles by forwarding to {@link OriginalPartialRectanglesFinder} and determine the tags of all "parts" that consitute the room
 *     e.g. all walls.
 * <p>
 * 2.) Iterate over all {@link Lamp}s in room and compute the set of illuminated rectangles by checking if an illuminated
 *  {@link Lamp} is inside a rectangle and (if yes) adding the tags of that rectangle to the set of illuminated rectangles.
 * <p>
 * 3.) Check if the set of tags of illuminated rectangles contains all tags of the room.
 * <p>
 * <p>
 The algorithm for {@link testIfRoomIsIlluminated}({@link Iterator}<{@link Lamp}> taggedLampsIterator, {@link HashSet}<{@link Integer}> allTags, {@link IRuntimeIlluminationTester} runtimeInfo) works as follows:
 * <p>
 * 1.) Iterate over tagged lamps and construct a set of tags of illuminated original rectangles
 * <p>
 * 2.) Check if the set of tags of illuminated rectangles contains all tags of original rectangles
 * <p>
 * <p>
 The algorithm for {@link testIfRoomIsIlluminated}({@link HashSet}<{@link Integer}> illuminatedTags, {@link HashSet}<{@link Integer}> allTags, {@link IRuntimeIlluminationTester} runtimeInfo) works as follows:
 * <p>
 * 1.) Check if the set of tags of illuminated rectangles (illuminatedTags) contains all tags of original rectangles (allTags)
 * <p>
 * <p>
 * Implemented interfaces and super classes: {@link ICandidateSearcher}
 * @author alex
 *
 */
public class IlluminationTester implements IIlluminationTester{

         public IlluminationTester() {}


        @Override
        public boolean testIfRoomIsIlluminated(IRoom room, IRuntimeIlluminationTester runtimeInfo) throws IlluminationTesterException {
                try {
                        // find original rectangles
                        IOriginalPartialRectanglesFinder originalRectanglesFinder = AbstractAlgorithmFactory.getAlgorithmFactory().createOriginalPartialRectanglesFinder();
                        runtimeInfo.startTimeOriginalPartialRectanglesFind();
                        List<RectangleWithTag> rectanglesWithTag = originalRectanglesFinder.findOriginalPartialRectangles(room, runtimeInfo);
                        runtimeInfo.stopTimeOriginalPartialRectanglesFind();

                        // store all tags
                        HashSet<Integer> allTags = originalRectanglesFinder.getAllTags();

                        // compute set of tags of illuminated lamps
                        HashSet<Integer> tagsOfAllIlluminatedLamps = new HashSet<Integer>();
                        Iterator<Lamp> lampIterator = room.getLamps();
                        while(lampIterator.hasNext()) {
                                Lamp lamp = lampIterator.next();
                                if(lamp.getOn()) {
                                        for(RectangleWithTag rec : rectanglesWithTag) {
                                                if(lamp.isInsideRectangle(rec.getP1(), rec.getP3())) {
                                                        Iterator<Integer> tagIterator = rec.getCopyOfTags().iterator();
                                                        //Iterator<Integer> tagIterator = rec.getTagIterator();
                                                        while(tagIterator.hasNext()) {
                                                                tagsOfAllIlluminatedLamps.add(tagIterator.next());
                                                        }
                                                }
                                        }
                                }
                        }
                        // check if the set of tags of illuminated rectangles contains the tags of all rectangles
                        if(tagsOfAllIlluminatedLamps.containsAll(allTags)) {
                                return true;
                        } else {
                                return false;
                        }

                } catch (OriginalPartialRectanglesFinderException e) {
                        throw new IlluminationTesterException(e);
                } catch (RuntimeExceptionLamps rte) {
                        throw new IlluminationTesterException(rte);
                }
        }

        @Override
        public boolean testIfRoomIsIlluminated(Iterator<Lamp> taggedLampsIterator, HashSet<Integer> allTags, IRuntimeIlluminationTester runtimeInfo) {
                return illuminatedLampsCoverAllTags(taggedLampsIterator, allTags);
        }

        private static boolean illuminatedLampsCoverAllTags(Iterator<Lamp> tag
gedLampsIterator, HashSet<Integer> allTags) {
                // compute set of tags of illuminated rectangles
                HashSet<Integer> tagsOfAllIlluminatedLamps = new HashSet<Integer>();
                while(taggedLampsIterator.hasNext()) {
                        Lamp lamp = taggedLampsIterator.next();
                        if (lamp.getOn()) {
                                Iterator<Integer> tagIterator = lamp.iteratorTag();
                                while(tagIterator.hasNext()) {
                                        tagsOfAllIlluminatedLamps.add(tagIterator.next());
                                }
                        }
                }
                if (tagsOfAllIlluminatedLamps.containsAll(allTags)) {
                        return true;
                } else {
                        return false;
                }
        }

        @Override
        public boolean testIfRoomIsIlluminated(HashSet<Integer> illuminatedTags, HashSet<Integer> allTags,
                        IRuntimeIlluminationTester runtimeInfo) {
                return illuminatedTags.containsAll(allTags);
        }
}
```

```java
package fernuni.propra.algorithm;

import java.util.Comparator;
import java.util.Iterator;

import fernuni.propra.internal_data_model.Wall;

/**
 * A specific container that stores east walls. Those {@link Wall}s can be specified by
 * two {@link Point}s in a horizontal-vertical coordinate system. The {@link Wall}s
 * in this container are ordered in ascending order with respect to the horizontal component
 * (x-component) of their {@link Point}s.
 * <p>
 * The total ordering requested by {@link WallContainerAbstract} is such that walls
 * <p>
 * Extended classes and implemented interfaces: {@link WallContainerAbstract}.
 * <p>
 * @author alex
 *
 */
public class WallContainerEast extends WallContainerAbstract {

        @Override
        protected boolean isValidWall(Wall wall, double limit, double low, double high) {
                return wall.overlapsYrange(low, high) &&  wall.getP1().getX()>=limit;
        }


        @Override
        protected Comparator<Wall> getComparator() {
                return new Comparator<Wall>() {
                        @Override
                        public int compare(Wall o1, Wall o2) {
                                if (o1.getP1().getX() < o2.getP1().getX()) {
                                        return -1;
                                } else if (o1.getP1().getX()>o2.getP1().getX()) {
                                        return 1;
                                }
                                return 0;
                        }
                };
        }


        @Override
        protected boolean isCorrectWallType(Wall wall) {
                return wall.isEastWall();
        }
}
```

```java
package fernuni.propra.algorithm;

import java.util.List;

import fernuni.propra.algorithm.runtime_information.IRuntimeCandidateSearcher;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;

/**
 *
 * A provider of an algorithm that can compute a {@link List} of potential {@link Lamp} positions
 * for an instance of {@link IRoom}.
 *
 * <p>
 * Implementing classes: {@link CandidateSearcher}
 * <p>
 *
 * @author alex
 *
 *
 *
 */
public interface ICandidateSearcher {

        /**
         * A method that provides the functionality of {@link ICandidateSearcher} to callers. It returns a {@link List} of
         * {@link Lamp}s that are potential lamp positions at which lamps might be placed to illuminate the room.
         * @param room : an instance of {@link IRoom} for which the lamp positions are to be determined.
         * @param runtimeCandidateSearcher : an instance of {@link IRuntimeCandidateSearcher} to which runtime information can be saved
         * @return a {@link List} of
         * {@link Lamp}s that contains potential lamp positions for the provided {@link IRoom}
         * @throws CandidateSearcherException : thrown if an error occurs during execution
         * @throws InterruptedException : thrown if the executing thread is interrupted, e.g. to stop after a certain time
         */
        List<Lamp> searchCandidates(IRoom room, IRuntimeCandidateSearcher runtimeCandidateSearcher) throws CandidateSearcherException, InterruptedException;

}
```

```java
package fernuni.propra.algorithm;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import fernuni.propra.algorithm.runtime_information.IRuntimePositionOptimizer;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;

/**
 * A specific provider of an algorithm that finds a minimum set (and number) of tagged {@link Lamp}s that
 * illuminates an {@link IRoom} instance.
 * <p>
 * The algorithm works as follows:
 * <p>
 * 1.) Global (to this class) fields are introduced
 * that store the currently best solution ({@link List}<{@link Lamp}>) and the number of {@link Lamp}s
 * that are turned on in the currently best solution
 * <p>
 * 2.) All {@link Lamp}s in the supplied list are turned on (i.e. the {@link IRoom} represented by the portions
 *     represented by the tags of the {@link Lamp}s is illuminated. The number of illuminated {@link Lamp}s is
 *     consequently set to the size of the originally supplied set of lamps.
 * <p>
 * 3.) An index idx that can be used to navigate the set of {@link Lamp}s is introduced (idx = 0 initially)
 *     and the ideal configuration of {@link Lamp}s is computed by recursively calling the
 *     method searchSolution in the manner of a backtracking algorithm. The method follows the following pseudo-code
 *     <p>
 *     PROCEDURE searchSolution (lamps, idx) {
       <p>
       if ( check if all portions of room are illuminated) { // if not all other branches cannot illuminate the room either
       <p>
            if(number of turned on lamps < number of on lamps in best solution ) {
       <p>
            best solution  = lamps
       <p>
            number of lamps in best solution = number of turned on lamps }
                <p>
        if (idx < size of lamps) {
        <p>
            searchSolution(lamps, idx+1)
        <p>
            turn off lamp[idx]
        <p>
            searchSolution(lamps, idx+1)
        <p>
         }
        <p>
    }
    <p>
  }
 <p>
 * 4.) The computation can be interrupted by interrupting the exectuting thread. The computation will
 *     stop immediately with an {@link InterruptedException}.
 * <p>
 * 5.) The currently available best solution can now be obtained.
 *
 * <p>
 * Implemented interfaces and super classes: {@link IPositionOptimizer}
 *
 * @author alex
 *
 */
public class PositionOptimizer implements IPositionOptimizer{
        private static List<Lamp> currentBestSolution;
        private static int numberIlluminatedLampsBestSolution;
        private static IIlluminationTester illuminationTester = AbstractAlgorithmFactory.getAlgorithmFactory().createIlluminiationTester();

        public PositionOptimizer() {
        }

        @Override
        public List<Lamp> optimizePositions(List<Lamp> taggedCandidates, IRuntimePositionOptimizer runTimeInformation) throws InterruptedException
        {

                // all lamps are on -> illuminated
                currentBestSolution = taggedCandidates;
                numberIlluminatedLampsBestSolution = taggedCandidates.size();


                HashSet<Integer> allTags = new HashSet<Integer>();
                for (Lamp lamp : taggedCandidates) {
                        lamp.turnOn(); // make sure all lamps are turned on
                        Iterator<Integer> tagIterator = lamp.iteratorTag();
                        while(tagIterator.hasNext()) {
                                allTags.add(tagIterator.next());
                        }
                }

                ArrayList<Lamp> lamps = deepCopyLamps(taggedCandidates);
                searchSolution(lamps,0, allTags, numberIlluminatedLampsBestSolution,runTimeInformation);

                return currentBestSolution;



        }

        /**
         * Implements the backtracking algorithm as indicated in the commentary on the {@link PositionOptimizer} class.
         * @param lamps : the set of lamps for which an optimal configuration is to be found
         * @param idx : index of current iteration
         * @param allTags : a set of all portions of the room that need to be illuminated
         * @param numberLampsOn : number of lamps that are turned on in the provided lamps argument
         * @param runTimeInformation : a data structure that can be used to store runtime information
         * @throws InterruptedException
         */
        private void searchSolution(ArrayList<Lamp> lamps, int idx,
            HashSet<Integer> allTags, int numberLampsOn, IRuntimePositionOptimizer runTimeInformation) throws InterruptedException{
        if(Thread.currentThread().isInterrupted()) {
            throw new InterruptedException("Computation interrupted.");
        }

        if(illuminationTester.testIfRoomIsIlluminated(lamps.iterator(), allTags, runTimeInformation)) { /* valid solution found, else case
            does not need to be investigated since it all lamps with an idx larger than the supplied idx are already turned on*/
            if (numberLampsOn<numberIlluminatedLampsBestSolution) { // new best solution found
                System.out.println("Solution found with " + numberLampsOn + " lamps turned on.");
                currentBestSolution = deepCopyLamps(lamps);
                numberIlluminatedLampsBestSolution = numberLampsOn;
            }
            if (idx < lamps.size()) { // there are further configurations that can be investigated in this branch
                Lamp lamp = lamps.get(idx);
                //branch 1
                // lamp does not need to be turned on since it has been initialized as turned on
                searchSolution(deepCopyLamps(lamps), idx+1, allTags, numberLampsOn, runTimeInformation);

                //branch2
                lamp.turnOff();
                searchSolution(deepCopyLamps(lamps), idx+1, allTags, numberLampsOn-1, runTimeInformation);

            }

        } else { // not a valid solution, with all lamps > idx turned on

        }
    }

    /**
     * Provides funtionality to deep copy an ArrayList of {@link Lamp}s
     * @param lamps
     * @return a deep copy of the provided list of {@link Lamp}s
     */
    private static ArrayList<Lamp> deepCopyLamps(List<Lamp> lamps) {
        ArrayList<Lamp> outLamps = new ArrayList<Lamp>(lamps.size());
        Iterator<Lamp> lampsIterator = lamps.iterator();
        while(lampsIterator.hasNext()) {
            Lamp lamp = lampsIterator.next();
            outLamps.add(lamp.deepCopy());
        }
        return outLamps;
    }


    private static HashSet<Integer> deepCopyHashSet(HashSet<Integer> hashSet) {
        HashSet<Integer> outHashSet = new HashSet<Integer>();
        for (Integer integer : hashSet) {
            Integer outInteger = (int) integer;
            outHashSet.add(outInteger);
        }
        return outHashSet;
    }

    @Override
    public List<Lamp> getCurrentBestSolution() {
        if (currentBestSolution == null) {
            return null;
        }
        List<Lamp> outLamps = new LinkedList<Lamp>();
        Iterator<Lamp> lampIterator = currentBestSolution.iterator();
        while(lampIterator.hasNext()) {
            outLamps.add(lampIterator.next().deepCopy());
        }
        return outLamps;

    }

    @Override
    public int getNumberOfOnLampsBestSolution() {
        return numberIlluminatedLampsBestSolution;
    }




}
```

```java
package fernuni.propra.algorithm;

import java.util.ArrayList;
import java.util.HashSet;

import fernuni.propra.algorithm.runtime_information.IRuntimeOriginalPartialRectanglesFinder;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.internal_data_model.IRoom;
/**
 * A provider of an algorithm that can find the original partial rectangles for an {@link IRoom} instance.
 * <p>
 * Implementing classes: {@link OriginalPartialRectanglesFinder}
 * <p>
 * @author alex
 *
 */
public interface IOriginalPartialRectanglesFinder {
        /**
         * The original partial rectangles of an {@link IRoom} are tagged with {@link Integer}s that
         * denote "parts" (e.g. walls) of the room that are illuminated if the associated partial rectangle is illuminated.
         * If all tags are illuminated then the room is illuminated.
         * This method returns a set that contains all tags of all original partial rectangles of the {@link IRoom}.
         * <p>
         * The {@link findOriginalPartialRectangles} method needs to be called first in order for {@link getAllTags}() to work.
         * @return : {@link HashSet}<{@link Integer}> a set of all tags of the original partial rectangles of the {@link IRoom} parameter
         *                         of the previously called {@link findOriginalPartialRectangles} method.
         */
        HashSet<Integer> getAllTags();
        /**
         * Returns all original partial rectangles of an {@link IRoom} parameter and saves runtime information
         * to the instance of {@link IRuntimeOriginalPartialRectanglesFinder}. All rectangles are tagged with Integers that denote
         * the parts of the room (e.g. walls) that are illuminated if the rectangle is illuminated. If all tags are illuminated then it
         * must follow that the room is illuminated. Identical rectangles are only stored once.
         * @param room : the {@link IRoom} instance for which the original partial rectangles are to be determined
         * @param rt : a data structure of type {@link IRuntimeOriginalPartialRectanglesFinder} that can be used to store runtime information.
         * @return a list of tagged original partial rectangles of the room.
         * @throws OriginalPartialRectanglesFinderException
         */
        ArrayList<RectangleWithTag> findOriginalPartialRectangles(IRoom room, IRuntimeOriginalPartialRectanglesFinder rt) throws OriginalPartialRectanglesFinderException;

}
```

```java
package fernuni.propra.algorithm;


/**
 * Specifies an interface for the construction of parts of an algorithm defined by consistent
 * instances of {@link ICandidateSearcher}, {@link IPositionOptimizer} and {@link IIlluminationTester}.
 * <p>
 * Implements the abstract factory design pattern. Subclasses, i.e. "concrete factories "
 * must implement this interface.
 * <p>
 *
 * Extending classes: {@link AlgorithmFactory1}
 *
 *
 * @author alex
 *
 */
public abstract class AbstractAlgorithmFactory {

        /**
         * Provides an instance of a "concrete factory", that can deliver consistent {@link ICandidateSearcher},
         * {@link IPositionOptimizer} and {@link IIlluminationTester} objects.
         * @return An instance of a "concrete factory".
         */
        public static AbstractAlgorithmFactory getAlgorithmFactory() {
                return AlgorithmFactory1.getAlgorithmFactory1();
        }

        /**
         * Delivers an instance of {@link ICandidateSearcher} that works with the algorithms defined by the "concrete factory".
         * @return A consistent instance of {@link ICandidateSearcher}.
         */
        public abstract ICandidateSearcher createCandidateSearcher();

        /**
         * Delivers an instance of {@link IPositionOptimizer} that works with the algorithms defined by the "concrete factory".
         * @return A consistent instance of {@link IPositionOptimizer}.
         */
        public abstract IPositionOptimizer createPositionOptimizer();

        /**
         * Delivers an instance of {@link IIlluminationTester} that works with the algorithms defined by the "concrete factory".
         * @return A consistent instance of {@link IIlluminationTester}
         */
        public abstract IIlluminationTester createIlluminiationTester();

        /**
         * Delivers an instance of {@link IOriginalPartialRectanglesFinder} that works with the algorithms defined by the inheriting "concrete factor"
         * @return
         */
        public abstract IOriginalPartialRectanglesFinder createOriginalPartialRectanglesFinder();

}
```

```
package Algorithm_Component;

import fernuni.propra.algorithm.*;

import org.junit.Test;
import static org.junit.Assert.*;

/*
 * Informationen Ã¼ber das Unit-Testen mit Hilfe von JUnit finden Sie unter http://www.vogella.com/tutorials/JUnit/article.html.
 * In dem dort hinterlegten Dokument sind alle notwendigen Hilfsmittel erlÃ¤utert.
 *
 * Designen Sie Ihre Unit-Tests nach dem Arrange-Act-Assert-Prinzip
 */

public class API_Test_Validation {

        @Test
        public void validateFileHasToBeValid() {
                // Arrange
                IAusleuchtung api = new Ausleuchtung();
                // Act
                boolean solutionValid = api.validateSolution("");
                // Assert
                assertTrue("Ohne Angabe einer Datei wurde eine zulÃ¤ssige LÃ¶sung gefunden.", !solutionValid);
        }

        @Test
        public void validateTruePositive() {
                // Arrange
                IAusleuchtung api = new Ausleuchtung();
                // Act
                boolean solutionValid = api.validateSolution("instances/validationInstances/Selbsttest_20a_solved.xml");
                // Assert
                assertTrue("Eine zulÃ¤ssige LÃ¶sung wurde als nicht zulÃ¤ssig gewertet.", solutionValid);
        }

        @Test
        public void validateTrueNegative() {
                // Arrange
                IAusleuchtung api = new Ausleuchtung();
                // Act
                boolean solutionValid = api.validateSolution("instances/validationInstances/Selbsttest_20a_incomplete.xml");
                // Assert
                assertTrue("Eine unzulÃ¤ssige LÃ¶sung wurde als zulÃ¤ssig gewertet.", !solutionValid);
        }

}
```

```java
package fernuni.propra.file_processing;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.file_processing.FilePersistence;
import fernuni.propra.file_processing.PersistenceException;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.LineSegment;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Room;

public class FilePersistenceTest {
        Point p1,p2,p3,p4,p5;
        LineSegment l1,l2,l3,l4,l5;
        List<LineSegment> lineSegments;
        IRoom room;
        LinkedList<Point> corners;


        @Before
        public void setUp() {
                p1 = new Point (0,0);
                p2 = new Point (1,0);
                p3 = new Point(1,1);
                p4 = new Point(0,1);
                l1 = new LineSegment(p1, p2);
                l2 = new LineSegment(p2, p3);
                l3 = new LineSegment(p3,p4);
                l4 = new LineSegment(p4,p1);
                l5 = new LineSegment(p1, p3);
                lineSegments = new ArrayList<LineSegment>();
                lineSegments.add(l1);lineSegments.add(l2); lineSegments.add(l3); lineSegments.add(l4);

                corners= new LinkedList<Point>();
                corners.add(p1); corners.add(p2); corners.add(p3); corners.add(p4);

                room = new Room("test", null, corners);
        }


        @Test
        public void testTestWallAndAddToWalls() {
                //Arrange
                LineSegment lccw1 = new LineSegment(p1,p4);
                LineSegment lccw2 = new LineSegment(p4, p3);
                LineSegment lccw3 = new LineSegment(p3,p2);
                LineSegment lccw4 = new LineSegment(p2, p1);

                ArrayList<LineSegment> walls1 = new ArrayList<LineSegment>();
                ArrayList<LineSegment> walls2 = new ArrayList<LineSegment>();
                walls2.add(lccw1);
                ArrayList<LineSegment> walls3 = new ArrayList<LineSegment>();
                walls3.add(lccw1); walls3.add(lccw2);
                ArrayList<LineSegment> walls4 = new ArrayList<LineSegment>();
                walls4.add(lccw1); walls4.add(lccw2); walls4.add(lccw3);


                ArrayList<LineSegment> walls5 = new ArrayList<LineSegment>();
                ArrayList<LineSegment> walls6 = new ArrayList<LineSegment>();
                walls6.add(l1);
                ArrayList<LineSegment> walls7 = new ArrayList<LineSegment>();
                walls7.add(l1); walls7.add(l2);
                ArrayList<LineSegment> walls8 = new ArrayList<LineSegment>();
                walls8.add(l1); walls8.add(l2); walls8.add(l3);

                //Act, Assert
                try {
                        FilePersistence.testAndAddWallToWalls(lccw1, walls1);
                } catch(PersistenceException e) {
                        fail(e.getMessage());
                }
                try {
                        FilePersistence.testAndAddWallToWalls(lccw2, walls2);
                } catch(PersistenceException e) {
                        fail(e.getMessage());
                }
                try {
                        FilePersistence.testAndAddWallToWalls(lccw3, walls3);
                } catch(PersistenceException e) {
                        fail(e.getMessage());
                }
                try {
                        FilePersistence.testAndAddWallToWalls(lccw4, walls4);
                } catch(PersistenceException e) {
                        fail(e.getMessage());
                }

                try {
                        FilePersistence.testAndAddWallToWalls(l1, walls5);
                } catch(PersistenceException e) {
                        fail(e.getMessage());
                }
                try {
                        FilePersistence.testAndAddWallToWalls(l2, walls6);
                } catch(PersistenceException e) {
                        fail(e.getMessage());
                }
                try {
                        FilePersistence.testAndAddWallToWalls(l3, walls7);
                } catch(PersistenceException e) {
                        fail(e.getMessage());
                }
                try {
                        FilePersistence.testAndAddWallToWalls(l4, walls8);
                } catch(PersistenceException e) {
                        fail(e.getMessage());
                }

                //Assert
                assertEquals(walls1.get(0), lccw1);
                assertEquals(walls2.get(1), lccw2);
                assertEquals(walls3.get(2), lccw3);
                assertEquals(walls4.get(3), lccw4);
                assertEquals(walls5.get(0), l1);
                assertEquals(walls6.get(1), l2);
                assertEquals(walls7.get(2), l3);
                assertEquals(walls8.get(3), l4);

        }
```

```java


	@Test
	public void testReadInput() {
		//Arrange
		String[] xmlPathesOK = {"instances/validationInstances/Selbsttest_clockwise.xml",
					"instances/validationInstances/Selbsttest_counterClockwise.xml",
					"instances/validationInstances/Selbsttest_100a_incomplete.xml",
					"instances/validationInstances/Selbsttest_100a_incomplete.xml",
					"instances/validationInstances/Selbsttest_100a_solved.xml",
					"instances/validationInstances/Selbsttest_100a.xml",
					"instances/validationInstances/Selbsttest_100b.xml",
					"instances/validationInstances/Selbsttest_20a_incomplete.xml",
					"instances/validationInstances/Selbsttest_20a_solved.xml",
					"instances/validationInstances/Selbsttest_20a.xml",
					"instances/validationInstances/Selbsttest_20b.xml",
					"instances/validationInstances/Selbsttest_20c.xml"
		};

		String[] xmlPathesNOK = {"instances/validationInstances/Selbsttest_clockwiseNOK.xml",
					"instances/validationInstances/Selbsttest_counterClockwiseNOK.xml"
		};


		FilePersistence persistence = new FilePersistence();

		//Act, Assert
		for (String xmlFile: xmlPathesOK) {
			IRoom room = null;
			try {
				room = persistence.readInput(xmlFile);
			} catch(PersistenceException e) {
				fail(e.getMessage());
			}
		}

		for (String xmlFile: xmlPathesNOK) {
			IRoom room = null;
			try {
				room = persistence.readInput(xmlFile);
				fail("This xml file is not OK!" + xmlFile);
			} catch(PersistenceException e) {

			}
		}

	}

	@Test
	public void testWriteOutput() {
		//Arrange
		IPersistence persistence = new FilePersistence();

		//Act
		try {
			persistence.writeOutput(room,"/Users/alex/Desktop/test");
		} catch (PersistenceException e) {
			fail(e.getMessage());
		}
	}

	/*@Test
	public void testIsCounterClockWise() {
		Point p1 = new Point(0,0);
		Point p2 = new Point(1,0);
		Point p3 = new Point(1,1);
		Point p4 = new Point(0,1);
		List<Point> counterClockWise  = new ArrayList<Point>();
		List<Point> clockWise = new ArrayList<Point>();
		counterClockWise.add(p1); counterClockWise.add(p2); counterClockWise.add(p3); counterClockWise.add(p4);
		clockWise.add(p4); clockWise.add(p3); clockWise.add(p2); clockWise.add(p1);
		assertTrue(!FilePersistence.isCounterClockWise(clockWise, p2));
		assertTrue(FilePersistence.isCounterClockWise(counterClockWise, p2));
	}*/



}
```

```java
package fernuni.propra.file_processing;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

import fernuni.propra.internal_data_model.LineSegmentTest;
import fernuni.propra.internal_data_model.LineSegmentTestParameterized;
import fernuni.propra.internal_data_model.PointTest;

@RunWith(Suite.class)
@SuiteClasses({ FilePersistenceTest.class, LineSegmentTest.class, LineSegmentTestParameterized.class, PointTest.class })
public class FileProcessingTests {

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Room;
import fernuni.propra.internal_data_model.Wall;

public class PositionOptimizerTest {
        private IRoom mockRoom, room, room2, roomStar, roomHufeisen;
        private Point p1, p2, p3,p4, p5, p6, p7, p8;
        private Point pc1, pc2, pc3,pc4, pc5, pc6, pc7, pc8,pc9, pc10, pc11, pc12;
        private Point p31, p32, p33, p34, p35, p36, p37, p38;
        private Wall w1, w2,w3,w4;
        private LinkedList<Point> corners, corners2;

        @Before
        public void setup() {
                p1 = new Point(0,0);
                p2 = new Point(1,0);
                p3 = new Point (1,1);
                p4 = new Point(0,1);

                p5 = new Point(0.5, 1.0);
                p6 = new Point(0.5, 0.5);
                p7 = new Point(0,   0.5);



                corners= new LinkedList<Point>();
                corners.add(p1); corners.add(p2); corners.add(p3); corners.add(p4);

                corners2= new LinkedList<Point>();
                corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2.add(p5);
                corners2.add(p6); corners2.add(p7);

                room = new Room("test", null, corners);
                room2 = new Room("test", null, corners2);



                pc1 = new Point(1,-1);
                pc2 = new Point(2,-1);
                pc3 = new Point(2,1);
                pc4 = new Point(1,1);
                pc5 = new Point(1,2);
                pc6 = new Point(-1,2);
                pc7 = new Point(-1,1);
                pc8 = new Point(-2,1);
                pc9 = new Point(-2,-1);
                pc10 = new Point(-1,-1);
                pc11 = new Point(-1,-2);
                pc12 = new Point(1,-2);
                LinkedList<Point> cornersStar = new LinkedList<Point>();
                cornersStar.add(pc1);cornersStar.add(pc2);cornersStar.add(pc3);cornersStar.add(pc4);cornersStar.add(pc5);
                cornersStar.add(pc6);cornersStar.add(pc7);cornersStar.add(pc8);cornersStar.add(pc9);cornersStar.add(pc10);
                cornersStar.add(pc11);cornersStar.add(pc12);

                roomStar = new Room("star", null, cornersStar);


                p31 = new Point(-2,0);
                p32 = new Point(2,0);
                p33 = new Point(2,2);
                p34 = new Point(1,2);
                p35 = new Point(1,1);
                p36 = new Point(-1,1);
                p37 = new Point(-1,2);
                p38 = new Point(-2,2);
                LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
                cornersHufeisen.add(p31);cornersHufeisen.add(p32);cornersHufeisen.add(p33);cornersHufeisen.add(p34);cornersHufeisen.add(p35);
                cornersHufeisen.add(p36);cornersHufeisen.add(p37);cornersHufeisen.add(p38);
                roomHufeisen = new Room("hufeisen", null, cornersHufeisen);

        }

        @Test
        public void testOptimizePositions() {
                //Arrange
                IPositionOptimizer positionOptimizer = AbstractAlgorithmFactory.getAlgorithmFactory().createPositionOptimizer();
                IPositionOptimizer positionOptimizer2 = AbstractAlgorithmFactory.getAlgorithmFactory().createPositionOptimizer();
                ICandidateSearcher candidateSearcher = AbstractAlgorithmFactory.getAlgorithmFactory().createCandidateSearcher();

                List<Lamp> taggedCandidates = null;
                try {
                        taggedCandidates = candidateSearcher.searchCandidates(room, null);
                        Lamp lamp = new Lamp(0.0,0.0);
                        lamp.addTag(1);
                        taggedCandidates.add(lamp);
                } catch (CandidateSearcherException | InterruptedException e) {
                        fail("Candidates Searcher failed!");
                }

                List<Lamp> taggedCandidates2 = new LinkedList<Lamp>();
                Lamp lamp1 = new Lamp(0,0);
                lamp1.addTag(0);
                Lamp lamp2 = new Lamp(0,0);
                lamp2.addTag(1);
                Lamp lamp3 = new Lamp(0,0);
                lamp3.addTag(2);
                Lamp lamp4 = new Lamp(0,0);
                lamp4.addTag(3);

                Lamp lamp5 = new Lamp(0,0);
                lamp5.addTag(1); lamp5.addTag(2);
                Lamp lamp6 = new Lamp(0,0);
                lamp6.addTag(2);  lamp6.addTag(3);
                Lamp lamp7 = new Lamp(0,0);
                lamp7.addTag(3); lamp7.addTag(0);

                Lamp lamp8 = new Lamp(0,0);
                lamp8.addTag(1); lamp8.addTag(2); lamp8.addTag(3);
                Lamp lamp9 = new Lamp(0,0);
                lamp9.addTag(2);  lamp9.addTag(3); lamp9.addTag(0);
                Lamp lamp10 = new Lamp(0,0);
                lamp10.addTag(3); lamp10.addTag(0); lamp10.addTag(1);

                Lamp lamp11 = new Lamp(0,0);
                lamp11.addTag(0); lamp11.addTag(1); lamp11.addTag(2); lamp11.addTag(3);

                taggedCandidates2.add(lamp1); taggedCandidates2.add(lamp2); taggedCandidates2.add(lamp3); taggedCandidates2.add(lamp4);
                taggedCandidates2.add(lamp5); taggedCandidates2.add(lamp6); taggedCandidates2.add(lamp7); taggedCandidates2.add(lamp8);
                taggedCandidates2.add(lamp9); taggedCandidates2.add(lamp10); taggedCandidates2.add(lamp11);

                //Act
                List<Lamp> optimizedLamps = new LinkedList<Lamp>();
                try {
                        optimizedLamps = positionOptimizer.optimizePositions(taggedCandidates, null);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                List<Lamp> optimizedLamps2 = new LinkedList<Lamp>();
                try {
                        optimizedLamps2 = positionOptimizer2.optimizePositions( taggedCandidates2, null);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                //Assert
                for (int i = 0; i< optimizedLamps2.size()-1; i++) {
                        assertFalse(optimizedLamps2.get(i).getOn());
                }
                assertTrue(optimizedLamps2.get(optimizedLamps2.size()-1).getOn());

                assertTrue(optimizedLamps.get(0).getOn());
                assertFalse(optimizedLamps.get(1).getOn());
        }

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;

public class WallContainerEastTest {
        Point p1,p2,p3,p4;
        Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;

        @Before
        public void setUp() {
                //Arrange
                p1 = new Point(0,0);
                p2 = new Point(1,0);
                p3 = new Point(1,1);
                p4 = new Point(0,1);
                w1 = new Wall(p1,p2,0);
                w2 = new Wall(p2,p3,0);
                w3 = new Wall(p3,p4,0);
                w4 = new Wall(p4,p1,0);

                w5 = new Wall(p2,p1,0);
                w6 = new Wall(p3, p2,0);
                w7 = new Wall(p4, p3,0);
                w8 = new Wall(p1,p4,0);

                w9 = new Wall(p1,p1,0);
        }


        @Test
        public void testAdd() {
                //Arrange
                WallContainerEast wallContainerEast = new WallContainerEast();

                //Act
                boolean test1 = false;
                try {
                        wallContainerEast.add(w1);
                        fail("WallContainerException expected");
                } catch(WallContainerException ex) {
                        test1 = true;
                }

                boolean test2 = false;
                try {
                        wallContainerEast.add(w2);
                        test2 = true;
                } catch (WallContainerException ex) {
                        fail();
                }

                //Assert
                assertTrue(test1);
                assertTrue(test2);
        }

        @Test
        public void testGetNearestEastWall() {
                //Arrange
                WallContainerEast wallContainerEast = new WallContainerEast();

                try {
                        wallContainerEast.add(w2);
                } catch (WallContainerException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                try {
                        wallContainerEast.add(w8);
                } catch (WallContainerException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                //Act
                Wall w10 = null;
                try {
                        w10 = wallContainerEast.getNearestWall(-1, 1, 0.5);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }

                Wall w11 = null;
                try {
                        w11 = wallContainerEast.getNearestWall(-1, 1, 0.0);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }

                Wall w12 = null;
                try {
                        w12 = wallContainerEast.getNearestWall(-1, 1, -0.001);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }

                Wall w13 = null;
                try {
                        w13 = wallContainerEast.getNearestWall(-1, -0.5, -0.001);

                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }


                //Assert
                assertTrue(w10.getP1().isEqual(w2.getP1()) && w10.getP2().isEqual(w2.getP2()));
                assertFalse(w11.getP1().isEqual(w2.getP1()) && w11.getP2().isEqual(w2.getP2()));
                assertTrue(w12.getP1().isEqual(w8.getP1()) && w12.getP2().isEqual(w8.getP2()));
                assertTrue(w13 == null);

        }

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;

public class WallContainerWestTest {
        Point p1,p2,p3,p4;
        Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;

        @Before
        public void setUp() {
                //Arrange
                p1 = new Point(0,0);
                p2 = new Point(1,0);
                p3 = new Point(1,1);
                p4 = new Point(0,1);
                w1 = new Wall(p1,p2,0);
                w2 = new Wall(p2,p3,0);
                w3 = new Wall(p3,p4,0);
                w4 = new Wall(p4,p1,0);

                w5 = new Wall(p2,p1,0);
                w6 = new Wall(p3, p2,0);
                w7 = new Wall(p4, p3,0);
                w8 = new Wall(p1,p4,0);

                w9 = new Wall(p1,p1,0);
        }


        @Test
        public void testAdd() {
                //Arrange
                WallContainerWest wallContainerWest = new WallContainerWest();

                //Act
                boolean test1 = false;
                try {
                        wallContainerWest.add(w1);
                        fail("WallContainerException expected");
                } catch(WallContainerException ex) {
                        test1 = true;
                }

                boolean test2 = false;
                try {
                        wallContainerWest.add(w4);
                        test2 = true;
                } catch (WallContainerException ex) {
                        fail();
                }

                //Assert
                assertTrue(test1);
                assertTrue(test2);
        }

        @Test
        public void testGetNearestWestWall() {
                //Arrange
                WallContainerWest wallContainerWest = new WallContainerWest();

                try {
                        wallContainerWest.add(w4);
                } catch (WallContainerException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                try {
                        wallContainerWest.add(w6);
                } catch (WallContainerException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                //Act
                Wall w10 = null;
                try {
                        w10 = wallContainerWest.getNearestWall(-1, 1, 0.5);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }
                boolean test1 = w10.getP1().isEqual(w4.getP1()) && w10.getP2()
.isEqual(w4.getP2());

                Wall w11 = null;
                try {
                        w11 = wallContainerWest.getNearestWall(-1, 1, 1.0);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }
                boolean test2 = w11.getP1().isEqual(w4.getP1()) && w11.getP2()
.isEqual(w4.getP2());

                Wall w12 = null;
                try {
                        w12 = wallContainerWest.getNearestWall(-1, 1, 1.001);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }
                boolean test3 = w12.getP1().isEqual(w6.getP1()) && w12.getP2()
.isEqual(w6.getP2());

                Wall w13 = null;
                try {
                        w13 = wallContainerWest.getNearestWall(-1, -0.5, -0.00
1);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }
                boolean test4 = w13 == null;


                //Assert
                assertTrue(test1);
                assertTrue(!test2);
                assertTrue(test3);
                assertTrue(test4);

        }

}
```

```java
package fernuni.propra.algorithm.util;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.Point;

public class RectangleWithTagTest {
        private Point p1,p2,p3,p4;
        private RectangleWithTag rec1;

        @Before
        public void setUp() throws Exception {

                //Arrange
                p1 = new Point(0,0);
                p2 = new Point(1,0);
                p3 = new Point(1,1);
                p4 = new Point(0,1);
                List<Integer> initTags = new ArrayList<Integer>();
                initTags.add(1);
                rec1 = new  RectangleWithTag(p1, p3, initTags);
        }

        @Test
        public void testContainsTag() {
                //Act
                boolean test1 = rec1.containsTag(1);
                boolean test2 = !rec1.containsTag(2);

                //Assert
                assertTrue(test1);
                assertTrue(test2);
        }

        @Test
        public void testAddTag() {
                fail("Not yet implemented");
        }

        @Test
        public void testHashSet() {
                //Arrange
                List<Integer> initTags2 = new ArrayList<Integer>(); initTags2.add(2);
                RectangleWithTag newRectangleWithTag = new RectangleWithTag(p1, p3, initTags2 );
                HashSet<RectangleWithTag> rectanglesWithTags = new HashSet<RectangleWithTag>();
                rectanglesWithTags.add(rec1);


                //Act
                boolean test1 = rectanglesWithTags.contains(newRectangleWithTag);
                boolean test2 = rec1.equals(newRectangleWithTag);

                //Assert
                assertFalse(test1);
                assertFalse(test2);
        }

}
```

```java
package fernuni.propra.algorithm.util;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.Point;

public class RectangleTest {
	Point p1,p2,p3,p4;
	Rectangle rec;

	@Before
	public void setUp() {
		//Arrange
		p1 = new Point(0,0);
		p2 = new Point(1,0);
		p3 = new Point(1,1);
		p4 = new Point(0,1);
		rec = new Rectangle(p1, p3);
	}

	@Test
	public void testOverlap() {
		//Arrange
		Point p5 = new Point(0.5,0.5);
		Point p6 = new Point(1.5,0.5);
		Point p7 = new Point(1.5,1.5);
		Point p8 = new Point(0.5,1.5);
		Point p9 = new Point(0.5,2.0);
		Point p10 = new Point(-1, -1);

		Rectangle rec2 = new Rectangle(p5, p7);
		Rectangle rec3 = new Rectangle(p5, p3);
		Rectangle rec5 = new Rectangle(p3, p7);
		Rectangle rec8 = new Rectangle(p10, p9);
		Rectangle rec9 = new Rectangle(p1, new Point(0.5, 1.0));
		Rectangle rec11 = new Rectangle(p2, new Point(2*p2.getX(), 1.0
));

		//Act
		Rectangle rec4 = rec.overlap(rec2);
		Rectangle rec6 = rec.overlap(rec5);
		Rectangle rec7 = rec.overlap(rec);
		Rectangle rec10 = rec.overlap(rec8);
		Rectangle rec12 = rec.overlap(rec11);


		assertTrue(rec3.equals(rec4));
		assertTrue(rec6 == null);
		assertTrue(rec7.equals(rec));
		assertTrue(rec9.equals(rec10));
		assertTrue(rec12 == null);

	}

	@Test
	public void testGetCenter() {

		//Act
		Point result = rec.getCenter();

		//Assert
		assertTrue(result.isEqual(new Point(0.5, 0.5)));

	}

	@Test
	public void testIsEqual() {
		//Arrange
		Point p5 = new Point(2,0);
		Point p6 = new Point(2,2);
		Rectangle rec2 = new Rectangle(p1, p6);

		//Act
		boolean test1 = rec.equals(rec);
		boolean test2 = rec.equals(rec2);

		//Assert
		assertTrue(test1);
		assertFalse(test2);

	}

	@Test
	public void testIsCounterClockWise() {
		//Arrange
		Rectangle rec2 = new Rectangle(p1, p1);

		//Act
		boolean test1 = rec2.isCounterClockWise();
		boolean test2 = rec.isCounterClockWise();

		//Assert
		assertFalse(test1);
		assertTrue(test2);
	}

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class RectangleWithTagTest {

        @Before
        public void setUp() throws Exception {
        }

        @Test
        public void testContainsTag() {
                fail("Not yet implemented");
        }

        @Test
        public void testAddTag() {
                fail("Not yet implemented");
        }

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Room;
import fernuni.propra.internal_data_model.Wall;

public class UserSolveAASTest {

        private IRoom room, room2, roomStar, roomHufeisen;
        private Point p1, p2, p3,p4, p5, p6, p7, p8;
        private Point pc1, pc2, pc3,pc4, pc5, pc6, pc7, pc8,pc9, pc10, pc11, p
c12;
        private Point p31, p32, p33, p34, p35, p36, p37, p38;
        private Wall w1, w2,w3,w4;
        private LinkedList<Point> corners, corners2;

        @Before
        public void setup() {
                p1 = new Point(0,0);
                p2 = new Point(1,0);
                p3 = new Point (1,1);
                p4 = new Point(0,1);

                p5 = new Point(0.5, 1.0);
                p6 = new Point(0.5, 0.5);
                p7 = new Point(0,   0.5);



                corners= new LinkedList<Point>();
                corners.add(p1); corners.add(p2); corners.add(p3); corners.add
(p4);

                corners2= new LinkedList<Point>();
                corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2
.add(p5);
                corners2.add(p6); corners2.add(p7);

                room = new Room("test", null, corners);
                room2 = new Room("test", null, corners2);


                pc1 = new Point(1,-1);
                pc2 = new Point(2,-1);
                pc3 = new Point(2,1);
                pc4 = new Point(1,1);
                pc5 = new Point(1,2);
                pc6 = new Point(-1,2);
                pc7 = new Point(-1,1);
                pc8 = new Point(-2,1);
                pc9 = new Point(-2,-1);
                pc10 = new Point(-1,-1);
                pc11 = new Point(-1,-2);
                pc12 = new Point(1,-2);
                LinkedList<Point> cornersStar = new LinkedList<Point>();
                cornersStar.add(pc1);cornersStar.add(pc2);cornersStar.add(pc3)
;cornersStar.add(pc4);cornersStar.add(pc5);
                cornersStar.add(pc6);cornersStar.add(pc7);cornersStar.add(pc8)
;cornersStar.add(pc9);cornersStar.add(pc10);
                cornersStar.add(pc11);cornersStar.add(pc12);

                roomStar = new Room("star", null, cornersStar);


                p31 = new Point(-2,0);
                p32 = new Point(2,0);
                p33 = new Point(2,2);
                p34 = new Point(1,2);
                p35 = new Point(1,1);
                p36 = new Point(-1,1);
                p37 = new Point(-1,2);
                p38 = new Point(-2,2);
                LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
                cornersHufeisen.add(p31);cornersHufeisen.add(p32);cornersHufei
sen.add(p33);cornersHufeisen.add(p34);cornersHufeisen.add(p35);
                cornersHufeisen.add(p36);cornersHufeisen.add(p37);cornersHufei
sen.add(p38);
                roomHufeisen = new Room("hufeisen", null, cornersHufeisen);

        }

        @Test
        public void testSolve() {
                //Arrange
                UserSolveAAS userSolve = new UserSolveAAS();

                //Act
                try {
                        userSolve.solve(room, 100);
                } catch (UserSolveAASException e) {
                        fail();
                }

                //Assert
                assertTrue(room.getLamps().hasNext());
        }

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;

public class WallContainerSouthTest {
	Point p1,p2,p3,p4;
	Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;

	@Before
	public void setUp() {
		//Arrange
		p1 = new Point(0,0);
		p2 = new Point(1,0);
		p3 = new Point(1,1);
		p4 = new Point(0,1);
		w1 = new Wall(p1,p2,0);
		w2 = new Wall(p2,p3,0);
		w3 = new Wall(p3,p4,0);
		w4 = new Wall(p4,p1,0);

		w5 = new Wall(p2,p1,0);
		w6 = new Wall(p3, p2,0);
		w7 = new Wall(p4, p3,0);
		w8 = new Wall(p1,p4,0);

		w9 = new Wall(p1,p1,0);
	}


	@Test
	public void testAdd() {
		//Arrange
		WallContainerSouth wallContainerSouth = new WallContainerSouth();

		//Act
		boolean test1 = false;
		try {
			wallContainerSouth.add(w2);
			fail("WallContainerException expected");
		} catch(WallContainerException ex) {
			test1 = true;
		}

		boolean test2 = false;
		try {
			wallContainerSouth.add(w1);
			test2 = true;
		} catch (WallContainerException ex) {
			fail();
		}

		//Assert
		assertTrue(test1);
		assertTrue(test2);
	}

	@Test
	public void testGetNearestWestWall() {
		//Arrange
		WallContainerSouth wallContainerSouth = new WallContainerSouth();

		try {
			wallContainerSouth.add(w1);
		} catch (WallContainerException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		try {
			wallContainerSouth.add(w7);
		} catch (WallContainerException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		//Act
		Wall w10 = null;
		try {
			w10 = wallContainerSouth.getNearestWall(-1, 1, 0.5);
		} catch (WallContainerException e) {
			fail(e.getMessage());
		}
		boolean test1 = w10.getP1().isEqual(w1.getP1()) && w10.getP2().isEqual(w1.getP2());

		Wall w11 = null;
		try {
			w11 = wallContainerSouth.getNearestWall(-1, 1, 1.0);
		} catch (WallContainerException e) {
			fail(e.getMessage());
		}
		boolean test2 = w11.getP1().isEqual(w7.getP1()) && w11.getP2().isEqual(w7.getP2());

		Wall w12 = null;
		try {
			w12 = wallContainerSouth.getNearestWall(-1, 1, 1.001);
		} catch (WallContainerException e) {
			fail(e.getMessage());
		}
		boolean test3 = w12.getP1().isEqual(w7.getP1()) && w12.getP2().isEqual(w7.getP2());

		Wall w13 = null;
		try {
			w13 = wallContainerSouth.getNearestWall(-1, -0.5, -0.001);
		} catch (WallContainerException e) {
			fail(e.getMessage());
		}
		boolean test4 = w13 == null;


		//Assert
		assertTrue(test1);
		assertTrue(test2);
		assertTrue(test3);
		assertTrue(test4);

	}

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import java.awt.Color;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import fernuni.propra.algorithm.runtime_information.RuntimeInformation;
import fernuni.propra.algorithm.util.Rectangle;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
import fernuni.propra.file_processing.UserReadInputWriteOutputException;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Room;
import fernuni.propra.internal_data_model.Wall;
import fernuni.propra.user_interface.RoomFrame;
import fernuni.propra.user_interface.RoomPanel;

public class OriginalPartialRectanglesFinderTest {
	private IRoom mockRoom, room, room2, roomStar, roomHufeisen;
	private Point p1, p2, p3,p4, p5, p6, p7, p8;
	private Point pc1, pc2, pc3,pc4, pc5, pc6, pc7, pc8,pc9, pc10, pc11, pc12;
	private Point p31, p32, p33, p34, p35, p36, p37, p38;
	private Wall w1, w2,w3,w4;
	private LinkedList<Point> corners, corners2;
	private static List<IRoom> rooms;

	@BeforeClass
	public static void setupBC() {

		String[] xmlPathesOK = {"instances/validationInstances/Selbsttest_clockwise.xml", //0
					"instances/validationInstances/Selbsttest_counterClockwise.xml", //1
					"instances/validationInstances/Selbsttest_100a_incomplete.xml", // 2
					"instances/validationInstances/Selbsttest_100a_incomplete.xml", //3
					"instances/validationInstances/Selbsttest_100a_solved.xml", // 4
					"instances/validationInstances/Selbsttest_100a.xml", // 5
					"instances/validationInstances/Selbsttest_100b.xml", // 6
					"instances/validationInstances/Selbsttest_20a_incomplete.xml", // 7
					"instances/validationInstances/Selbsttest_20a_solved.xml", // 8
					"instances/validationInstances/Selbsttest_20a.xml", // 9
					"instances/validationInstances/Selbsttest_20b.xml", // 10
					"instances/validationInstances/Selbsttest_20c.xml",   // 11
					"instances/validationInstances/Zufallsraum_144_solved.xml" // 12
		};

		rooms = new ArrayList<IRoom>();

		for(String xmlPath : xmlPathesOK) {
			UserReadInputWriteOutputAAS readAAS = new UserReadInputWriteOutputAAS(xmlPath);
			try {
				rooms.add(readAAS.readInput());
			} catch (UserReadInputWriteOutputException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}

	@Before
	public void setUp() throws Exception {
		p1 = new Point(0,0);
		p2 = new Point(1,0);
		p3 = new Point (1,1);
		p4 = new Point(0,1);

		p5 = new Point(0.5, 1.0);
		p6 = new Point(0.5, 0.5);
		p7 = new Point(0,   0.5);




		corners= new LinkedList<Point>();
		corners.add(p1); corners.add(p2); corners.add(p3); corners.add(p4);

		corners2= new LinkedList<Point>();
		corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2.add(p5);
		corners2.add(p6); corners2.add(p7);

		room = new Room("test", null, corners);
		room2 = new Room("test", null, corners2);


		pc1 = new Point(1,-1);
		pc2 = new Point(2,-1);
		pc3 = new Point(2,1);
		pc4 = new Point(1,1);
		pc5 = new Point(1,2);
		pc6 = new Point(-1,2);
		pc7 = new Point(-1,1);
		pc8 = new Point(-2,1);
		pc9 = new Point(-2,-1);
		pc10 = new Point(-1,-1);
		pc11 = new Point(-1,-2);
		pc12 = new Point(1,-2);
		LinkedList<Point> cornersStar = new LinkedList<Point>();
		cornersStar.add(pc1);cornersStar.add(pc2);cornersStar.add(pc3);cornersStar.add(pc4);cornersStar.add(pc5);
		cornersStar.add(pc6);cornersStar.add(pc7);cornersStar.add(pc8);cornersStar.add(pc9);cornersStar.add(pc10);
		cornersStar.add(pc11);cornersStar.add(pc12);

		roomStar = new Room("star", null, cornersStar);


		p31 = new Point(-2,0);
		p32 = new Point(2,0);
		p33 = new Point(2,2);
		p34 = new Point(1,2);
		p35 = new Point(1,1);
		p36 = new Point(-1,1);
		p37 = new Point(-1,2);
		p38 = new Point(-2,2);
		LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
		cornersHufeisen.add(p31);cornersHufeisen.add(p32);cornersHufeisen.add(p33);cornersHufeisen.add(p34);cornersHufeisen.add(p35);
		cornersHufeisen.add(p36);cornersHufeisen.add(p37);cornersHufeisen.add(p38);
		roomHufeisen = new Room("hufeisen", null, cornersHufeisen);

	}

	@Test
	public void testFindOriginalPartialRectangles() {
		//Arrange
		OriginalPartialRectanglesFinder rectanglesFinder3 = new OriginalPartialRectanglesFinder();
		CandidateSearcher candidateSearcher = (CandidateSearcher) AbstractAlgorithmFactory.getAlgorithmFactory().createCandidateSearcher();

		//Act
		IRoom testRoom = rooms.get(9);
		//IRoom testRoom = roomStar;
		ArrayList<RectangleWithTag> rectanglesWithTag = new ArrayList<RectangleWithTag>();
		try {
			rectanglesWithTag = rectanglesFinder3.findOriginalPartialRectangles(testRoom, null);
		} catch (OriginalPartialRectanglesFinderException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		RoomPanel roomPanel = new RoomPanel(testRoom);
		Color[] colors = {Color.blue, Color.red, Color.green, Color.yellow};




		IRoom testRoom2 = rooms.get(12);
		//Act

		//IRoom testRoom = roomStar;
		ArrayList<RectangleWithTag> rectanglesWithTag2 = new ArrayList<RectangleWithTag>();
		try {
			rectanglesWithTag2 = (new OriginalPartialRectanglesFinder()).findOriginalPartialRectangles(testRoom2, null);
		} catch (OriginalPartialRectanglesFinderException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		RoomPanel roomPanel2 = new RoomPanel(testRoom2);
		Color[] colors2 = {Color.blue, Color.red, Color.green, Color.yellow};

		List<RectangleWithTag> rectanglesWithTag3 = new ArrayList<RectangleWithTag>();
		try {
			rectanglesWithTag3 = (new CandidateSearcher()).reduceRectangles(rectanglesWithTag2);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}


		RoomFrame roomFrame = new RoomFrame(roomPanel2);

		for (int i = 0; i<rectanglesWithTag3.size(); i++) {
		RectangleWithTag rec = rectanglesWithTag3.get(i);
		double width = rec.getP2().getX() - rec.getP1().getX();
		double height = rec.getP3().getY() - rec.getP1().getY();
		roomPanel2.addRectangle(String.valueOf(i), colors[i % 3], rec.getP1().getX(), rec.getP1().getY(), width, height);
		roomPanel2.repaint();
		//roomPanel2.removeLastRectangle();
		}


		try {
			Thread.currentThread().sleep(100000);
		} catch (InterruptedException e) {
			// TODO: handle exception
		}

		fail("not yet implemented");
	}

	@Test
	public void testGetAllTags() {
		// Arrange
		OriginalPartialRectanglesFinder rectanglesFinder = new OriginalPartialRectanglesFinder();
		HashSet<Integer> refSet = new HashSet<Integer>();
		refSet.add(0); refSet.add(1); refSet.add(2); refSet.add(3);

		//Act
		try {
			rectanglesFinder.sortWallsToContainers(room);
			rectanglesFinder.constructOriginalPartialRectangles();
		} catch (WallContainerException e) {
			fail(e.getMessage());
		} catch (OriginalPartialRectanglesFinderException e) {
			fail(e.getMessage());
		}

		//Assert
		assertTrue(rectanglesFinder.getAllTags().containsAll(refSet));
	}

	@Test
	public void testSortWallsToContainers() {
		//Arrange
				OriginalPartialRectanglesFinder originalRectanglesFinder = new OriginalPartialRectanglesFinder();
				//Act
				try {
					originalRectanglesFinder.sortWallsToCo
```

```java
ntainers(room);
        } catch (OriginalPartialRectanglesFinderException e) {
            fail(e.getMessage());
        }

        //Assert
        Iterator<Wall> east = originalRectanglesFinder.eastIterator();
        Iterator<Wall> north = originalRectanglesFinder.northIterator();
        Iterator<Wall> west = originalRectanglesFinder.westIterator();
        Iterator<Wall> south = originalRectanglesFinder.southIterator();

        //east
        boolean test11 = east.hasNext();
        Wall wallEast = east.next();
        boolean test12 = !east.hasNext();
        boolean test13 = wallEast.getP1().isEqual(w2.getP1()) && wallEast.getP2().isEqual(w2.getP2());
        boolean eastBool = test11 && test12 && test13;

        //north
        boolean test21 = north.hasNext();
        Wall wallNorth = north.next();
        boolean test22 = !north.hasNext();
        boolean test23 = wallNorth.getP1().isEqual(w3.getP1()) && wallNorth.getP2().isEqual(w3.getP2());
        boolean northBool = test21 && test22 && test23;

        //west
        boolean test31 = west.hasNext();
        Wall wallWest = west.next();
        boolean test32 = !west.hasNext();
        boolean test33 = wallWest.getP1().isEqual(w4.getP1()) && wallWest.getP2().isEqual(w4.getP2());
        boolean westBool = test31 && test32 && test33;

        //west
        boolean test41 = south.hasNext();
        Wall wallSouth = south.next();
        boolean test42 = !south.hasNext();
        boolean test43 = wallSouth.getP1().isEqual(w1.getP1()) && wallSouth.getP2().isEqual(w1.getP2());
        boolean southBool = test41 && test42 && test43;

        assertTrue(eastBool && northBool && westBool && southBool);
    }

    @Test
    public void testConstructOriginalPartialRectangles() {
        // Arrange
        OriginalPartialRectanglesFinder rectanglesFinder = new OriginalPartialRectanglesFinder();
        OriginalPartialRectanglesFinder rectanglesFinder2 = new OriginalPartialRectanglesFinder();


        //Act
        try {
            rectanglesFinder.sortWallsToContainers(room);
            rectanglesFinder.constructOriginalPartialRectangles();
        } catch (WallContainerException e) {
            fail(e.getMessage());
        } catch (OriginalPartialRectanglesFinderException e) {
            fail(e.getMessage());
        }

        //2nd room
        try {
            rectanglesFinder2.sortWallsToContainers(room2);
            rectanglesFinder2.constructOriginalPartialRectangles();
        } catch (WallContainerException e) {
            fail(e.getMessage());
        } catch (OriginalPartialRectanglesFinderException e) {
            fail(e.getMessage());
        }


        //Assert
        Iterator<RectangleWithTag> rectIterator = rectanglesFinder.iteratorOriginalRectangles();
        RectangleWithTag rec1 = rectIterator.next();
        RectangleWithTag rec2 = rectIterator.next();
        RectangleWithTag rec3 = rectIterator.next();
        RectangleWithTag rec4 = rectIterator.next();
        Rectangle ref = new Rectangle(p1, p3);

        boolean test1 = !rectIterator.hasNext();
        boolean test2 = rec1.equals(new RectangleWithTag(ref, 0));
        boolean test3 = rec2.equals(new RectangleWithTag(ref, 1));
        boolean test4 = rec3.equals(new RectangleWithTag(ref, 2));
        boolean test5 = rec4.equals(new RectangleWithTag(ref, 3));

        assertTrue(test1 && test2 && test3 && test4 && test5);


        Iterator<RectangleWithTag> rectIterator2 = rectanglesFinder2.iteratorOriginalRectangles();
        RectangleWithTag rec2_1 = rectIterator2.next();
        RectangleWithTag rec2_2 = rectIterator2.next();
        RectangleWithTag rec2_3 = rectIterator2.next();
        RectangleWithTag rec2_4 = rectIterator2.next();
        RectangleWithTag rec2_5 = rectIterator2.next();
        RectangleWithTag rec2_6 = rectIterator2.next();
        Rectangle ref2 = new Rectangle(p1, new Point(1
,0.5));
                Rectangle ref3 = new Rectangle(new Point(0.5,0), p3);

                boolean test7 = !rectIterator2.hasNext();
                boolean test8 = rec2_1.equals(new RectangleWithTag(ref2, 0));
                boolean test9 = rec2_2.equals(new RectangleWithTag(ref3, 1));
                boolean test10 = rec2_3.equals(new RectangleWithTag(ref3, 2));
                boolean test11 = rec2_4.equals(new RectangleWithTag(ref3, 3));
                boolean test12 = rec2_5.equals(new RectangleWithTag(ref2, 4));
                boolean test13 = rec2_6.equals(new RectangleWithTag(ref2, 5));


                assertTrue(test7 && test8 && test9 && test10 && test11 && test12 && test13);
        }

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Room;
import fernuni.propra.internal_data_model.Wall;

public class IlluminationTesterTest {
        private IRoom mockRoom, room, room2, roomStar, roomHufeisen;
        private Point p1, p2, p3,p4, p5, p6, p7, p8;
        private Point pc1, pc2, pc3,pc4, pc5, pc6, pc7, pc8,pc9, pc10, pc11, pc12;
        private Point p31, p32, p33, p34, p35, p36, p37, p38;
        private LinkedList<Point> corners, corners2;

        @Before
        public void setUp() throws Exception {
                p1 = new Point(0,0);
                p2 = new Point(1,0);
                p3 = new Point (1,1);
                p4 = new Point(0,1);

                p5 = new Point(0.5, 1.0);
                p6 = new Point(0.5, 0.5);
                p7 = new Point(0,   0.5);





                corners= new LinkedList<Point>();
                corners.add(p1); corners.add(p2); corners.add(p3); corners.add(p4);

                corners2= new LinkedList<Point>();
                corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2.add(p5);
                corners2.add(p6); corners2.add(p7);

                room = new Room("test", null, corners);
                room2 = new Room("test", null, corners2);


                pc1 = new Point(1,-1);
                pc2 = new Point(2,-1);
                pc3 = new Point(2,1);
                pc4 = new Point(1,1);
                pc5 = new Point(1,2);
                pc6 = new Point(-1,2);
                pc7 = new Point(-1,1);
                pc8 = new Point(-2,1);
                pc9 = new Point(-2,-1);
                pc10 = new Point(-1,-1);
                pc11 = new Point(-1,-2);
                pc12 = new Point(1,-2);
                LinkedList<Point> cornersStar = new LinkedList<Point>();
                cornersStar.add(pc1);cornersStar.add(pc2);cornersStar.add(pc3);cornersStar.add(pc4);cornersStar.add(pc5);
                cornersStar.add(pc6);cornersStar.add(pc7);cornersStar.add(pc8);cornersStar.add(pc9);cornersStar.add(pc10);
                cornersStar.add(pc11);cornersStar.add(pc12);

                roomStar = new Room("star", null, cornersStar);


                p31 = new Point(-2,0);
                p32 = new Point(2,0);
                p33 = new Point(2,2);
                p34 = new Point(1,2);
                p35 = new Point(1,1);
                p36 = new Point(-1,1);
                p37 = new Point(-1,2);
                p38 = new Point(-2,2);
                LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
                cornersHufeisen.add(p31);cornersHufeisen.add(p32);cornersHufeisen.add(p33);cornersHufeisen.add(p34);cornersHufeisen.add(p35);
                cornersHufeisen.add(p36);cornersHufeisen.add(p37);cornersHufeisen.add(p38);
                roomHufeisen = new Room("hufeisen", null, cornersHufeisen);
        }

        @Test
        public void testTestIfRoomIsIlluminatedIRoomIRuntimeIlluminationTester() {
                //Arrange
                IIlluminationTester illuminationTester = AbstractAlgorithmFactory.getAlgorithmFactory().createIlluminiationTester();

                //Act
                boolean test1 = false;
                boolean test2 = false;
                boolean test3 = false;
                boolean test4 = false;
                boolean test5 = false;

                boolean test6 = false;
                boolean test7 = false;
                boolean test8 = false;
                boolean test9 = false;
                boolean test10 = false;

                try {
                        //Room 1
                        test1 = illuminationTester.testIfRoomIsIlluminated(room, null);
                        Lamp lamp = new Lamp(0.5, 0.5);
                        lamp.turnOff();
                        room.addLamp(lamp);
                        test2 = illuminationTester.testIfRoomIsIlluminated(room, null);
                        lamp.turnOn();
                        test3 = illuminationTester.testIfRoomIsIlluminated(room, null);
                        lamp.turnOff();
                        Lamp lamp2 = new Lamp(1.0, 3.0);
                        lamp2.turnOn();
                        room.addLamp(lamp2);
```

```java
            test4 = illuminationTester.testIfRoomIsIlluminated(room, null);
            Lamp lamp3 = new Lamp(1.0, 1.0);
            lamp3.turnOn();
            room.addLamp(lamp3);
            test5 = illuminationTester.testIfRoomIsIlluminated(room, null);

            //Room Hufeisen
            test6 = illuminationTester.testIfRoomIsIlluminated(roomHufeisen, null);
            Lamp lamp4 = new Lamp(-1.5, 0.5);
            lamp4.turnOn();
            roomHufeisen.addLamp(lamp4);
            test7 = illuminationTester.testIfRoomIsIlluminated(roomHufeisen, null);
            Lamp lamp5 = new Lamp(1.5, 0.5);
            lamp5.turnOn();
            roomHufeisen.addLamp(lamp5);
            test8 = illuminationTester.testIfRoomIsIlluminated(roomHufeisen, null);

        } catch (IlluminationTesterException e) {
        }



        //Assert
        assertFalse(test1);
        assertFalse(test2);
        assertTrue(test3);
        assertFalse(test4);
        assertTrue(test5);

        assertFalse(test6);
        assertFalse(test7);
        assertTrue(test8);

    }

    @Test
    public void testTestIfRoomIsIlluminatedIteratorOfLampHashSetOfIntegerIRuntimeIlluminationTester() {
        //Arrange
        IIlluminationTester illuminationTester = AbstractAlgorithmFactory.getAlgorithmFactory().createIlluminiationTester();
        HashSet<Integer> allTags = new HashSet<Integer>();
        allTags.add(0); allTags.add(1); allTags.add(2); allTags.add(3);
        List<Lamp> lamps= new LinkedList<Lamp>();
        Lamp lamp1 = new Lamp(0,0,0);
        lamp1.turnOn();
        lamps.add(lamp1);

        //Act
        boolean test1 = illuminationTester.testIfRoomIsIlluminated(lamps.iterator(), allTags, null);
        Lamp lamp2 = new Lamp(0,0,1);
        lamp2.addTag(2);
        lamp2.addTag(3);
        lamp2.turnOn();
        lamps.add(lamp2);
        boolean test2 = illuminationTester.testIfRoomIsIlluminated(lamps.iterator(), allTags, null);

        //Assert
        assertFalse(test1);
        assertTrue(test2);
    }

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;

public class WallContainerNorthTest {
        Point p1,p2,p3,p4;
        Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;

        @Before
        public void setUp() {
                //Arrange
                p1 = new Point(0,0);
                p2 = new Point(1,0);
                p3 = new Point(1,1);
                p4 = new Point(0,1);
                w1 = new Wall(p1,p2,0);
                w2 = new Wall(p2,p3,0);
                w3 = new Wall(p3,p4,0);
                w4 = new Wall(p4,p1,0);

                w5 = new Wall(p2,p1,0);
                w6 = new Wall(p3, p2,0);
                w7 = new Wall(p4, p3,0);
                w8 = new Wall(p1,p4,0);

                w9 = new Wall(p1,p1,0);
        }


        @Test
        public void testAdd() {
                //Arrange
                WallContainerNorth wallContainerNorth = new WallContainerNorth();

                //Act
                boolean test1 = false;
                try {
                        wallContainerNorth.add(w2);
                        fail("WallContainerException expected");
                } catch(WallContainerException ex) {
                        test1 = true;
                }

                boolean test2 = false;
                try {
                        wallContainerNorth.add(w3);
                        test2 = true;
                } catch (WallContainerException ex) {
                        fail();
                }

                //Assert
                assertTrue(test1);
                assertTrue(test2);
        }

        @Test
        public void testGetNearestNorthWall() {
                //Arrange
                WallContainerNorth wallContainerNorth = new WallContainerNorth();


                try {
                        wallContainerNorth.add(w3);
                } catch (WallContainerException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                try {
                        wallContainerNorth.add(w5);
                } catch (WallContainerException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                //Act
                Wall w10 = null;
                try {
                        w10 = wallContainerNorth.getNearestWall(-1, 1, 0.5);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }

                Wall w11 = null;
                try {
                        w11 = wallContainerNorth.getNearestWall(-1, 1, 0.0);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }

                Wall w12 = null;
                try {
                        w12 = wallContainerNorth.getNearestWall(-1, 1, -0.001);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }

                Wall w13 = null;
                try {
                        w13 = wallContainerNorth.getNearestWall(-1, -0.5, -0.001);
                } catch (WallContainerException e) {
                        fail(e.getMessage());
                }


                //Assert
                assertTrue(w10.getP1().isEqual(w3.getP1()) && w10.getP2().isEqual(w3.getP2()));
                assertFalse(w11.getP1().isEqual(w3.getP1()) && w11.getP2().isEqual(w3.getP2()));
                assertTrue(w12.getP1().isEqual(w5.getP1()) && w12.getP2().isEqual(w5.getP2()));
                assertTrue(w13 == null);

        }

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;

import fernuni.propra.algorithm.runtime_information.RuntimeInformation;
import fernuni.propra.algorithm.util.Rectangle;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
import fernuni.propra.file_processing.UserReadInputWriteOutputException;
import fernuni.propra.algorithm.util.RectangleWithTag;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Room;
import fernuni.propra.internal_data_model.Wall;

public class CandidateSearcherTest {

        private IRoom mockRoom, room, room2, roomStar, roomHufeisen;
        private Point pc1, pc2, pc3,pc4, pc5, pc6, pc7, pc8,pc9, pc10, pc11, pc12;
        private Point p31, p32, p33, p34, p35, p36, p37, p38;



        private static List<IRoom> rooms;

        @BeforeClass
        public static void setupBC() {

                String[] xmlPathesOK = {"instances/validationInstances/Selbsttest_clockwise.xml", //0
                                        "instances/validationInstances/Selbsttest_counterClockwise.xml", //1
                                        "instances/validationInstances/Selbsttest_100a_incomplete.xml", // 2
                                        "instances/validationInstances/Selbsttest_100a_incomplete.xml", //3
                                        "instances/validationInstances/Selbsttest_100a_solved.xml", // 4
                                        "instances/validationInstances/Selbsttest_100a.xml", // 5
                                        "instances/validationInstances/Selbsttest_100b.xml", // 6
                                        "instances/validationInstances/Selbsttest_20a_incomplete.xml", // 7
                                        "instances/validationInstances/Selbsttest_20a_solved.xml", // 8
                                        "instances/validationInstances/Selbsttest_20a.xml", // 9
                                        "instances/validationInstances/Selbsttest_20b.xml", // 10
                                        "instances/validationInstances/Selbsttest_20c.xml"    // 11
                };

                rooms = new ArrayList<IRoom>();

                for(String xmlPath : xmlPathesOK) {
                        UserReadInputWriteOutputAAS readAAS = new UserReadInputWriteOutputAAS(xmlPath);
                        try {
                                rooms.add(readAAS.readInput());
                        } catch (UserReadInputWriteOutputException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
        }

        @Before
        public void setup() {

                // build room star
                pc1 = new Point(1,-1);
                pc2 = new Point(2,-1);
                pc3 = new Point(2,1);
                pc4 = new Point(1,1);
                pc5 = new Point(1,2);
                pc6 = new Point(-1,2);
                pc7 = new Point(-1,1);
                pc8 = new Point(-2,1);
                pc9 = new Point(-2,-1);
                pc10 = new Point(-1,-1);
                pc11 = new Point(-1,-2);
                pc12 = new Point(1,-2);
                LinkedList<Point> cornersStar = new LinkedList<Point>();
                cornersStar.add(pc1);cornersStar.add(pc2);cornersStar.add(pc3);cornersStar.add(pc4);cornersStar.add(pc5);
                cornersStar.add(pc6);cornersStar.add(pc7);cornersStar.add(pc8);cornersStar.add(pc9);cornersStar.add(pc10);
                cornersStar.add(pc11);cornersStar.add(pc12);

                roomStar = new Room("star", null, cornersStar);

                // build room Hufeisen
                p31 = new Point(-2,0);
                p32 = new Point(2,0);
                p33 = new Point(2,2);
                p34 = new Point(1,2);
                p35 = new Point(1,1);
                p36 = new Point(-1,1);
                p37 = new Point(-1,2);
                p38 = new Point(-2,2);
                LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
                cornersHufeisen.add(p31);cornersHufeisen.add(p32);cornersHufeisen.add(p33);cornersHufeisen.add(p34);cornersHufeisen.add(p35);
                cornersHufeisen.add(p36);cornersHufeisen.add(p37);cornersHufeisen.add(p38);
                roomHufeisen = new Room("hufeisen", null, cornersHufeisen);


        }

        @Test
        public void testSearchCandidates() {
                //Arrange
                CandidateSearcher candidateSearcher1 = new CandidateSearcher()
;
                CandidateSearcher candidateSearcher2 = new CandidateSearcher();
                CandidateSearcher candidateSearcher3 = new CandidateSearcher();


                //Act
                List<Lamp> candidates = null;
                List<Lamp> candidates2 = null;
                List<Lamp> candidates3 = null;
                try {
                        candidates  = candidateSearcher1.searchCandidates(roomStar, new RuntimeInformation());
                        candidates2 = candidateSearcher2.searchCandidates(roomHufeisen, new RuntimeInformation());
                        candidates3 = candidateSearcher3.searchCandidates(rooms.get(9), new RuntimeInformation());
                } catch (CandidateSearcherException | InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                //Assert
                assertTrue(candidates != null && candidates.size() == 1 && candidates.get(0).isEqual(new Point(0.5,0.5)));

                assertTrue(candidates2 != null && candidates2.size() == 2);
                assertTrue(candidates2.get(0).isEqual(new Point(-1.5,0.5)));
                assertTrue(candidates2.get(1).isEqual(new Point(1.5,0.5)));
        }

        /** Checks if CandidateSearcher correctly determines the reduced tagged rectangles from a set of tagged rectangles that might overlap.
         * The reduced set contains the rectangles that result from overlapping. Only rectangles whose tags are not a subset of the tags of another rectangle are kept.
        */
        @Test
        public void testReduceRectangles() {
                // Arrange
                ArrayList<RectangleWithTag> rectanglesWithTagIn = new ArrayList<RectangleWithTag>();
                CandidateSearcher candidateSearcher = new CandidateSearcher();
                RectangleWithTag refRectangleWithTag = new RectangleWithTag(new Point(-2,0), new Point(-1,1), 0);
                refRectangleWithTag.addTag(1);
                RectangleWithTag refRectangleWithTag2 = new RectangleWithTag(new Point(1,0), new Point(2,1),1);
                refRectangleWithTag2.addTag(2);

                // Hufeisenkonfiguration von Rechtecken
                RectangleWithTag rec0 = new RectangleWithTag(new Point(-2, 0), new Point(-1, 2), 0);
                RectangleWithTag rec1 = new RectangleWithTag(new Point(-2, 0), new Point(2, 1), 1);
                RectangleWithTag rec2 = new RectangleWithTag(new Point(1,0), new Point(2,2),2);
                rectanglesWithTagIn.add(rec0); rectanglesWithTagIn.add(rec1); rectanglesWithTagIn.add(rec2);


                // Act
                ArrayList<RectangleWithTag> reducedRectangles = new ArrayList<RectangleWithTag>();
                try {
                        reducedRectangles = candidateSearcher.reduceRectangles(rectanglesWithTagIn);
                } catch (InterruptedException e) {
                        fail(e.getMessage());
                }

                // Assert
                assertTrue("Number of reduced rectangles is not correct.", reducedRectangles.size() == 2);
                assertTrue(reducedRectangles.get(0).equals(refRectangleWithTag));
                assertTrue(reducedRectangles.get(1).equals(refRectangleWithTag2));

        }

}
```

```java
package fernuni.propra.algorithm;

import static org.junit.Assert.*;

import java.util.LinkedList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Room;
import fernuni.propra.internal_data_model.Wall;

public class UserValidateAASTest {
	private IRoom room, room2, roomStar, roomHufeisen;
	private Point p1, p2, p3,p4, p5, p6, p7, p8;
	private Point pc1, pc2, pc3,pc4, pc5, pc6, pc7, pc8,pc9, pc10, pc11, pc12;
	private Point p31, p32, p33, p34, p35, p36, p37, p38;
	private Wall w1, w2,w3,w4;
	private LinkedList<Point> corners, corners2;

	@Before
	public void setup() {
		p1 = new Point(0,0);
		p2 = new Point(1,0);
		p3 = new Point (1,1);
		p4 = new Point(0,1);

		p5 = new Point(0.5, 1.0);
		p6 = new Point(0.5, 0.5);
		p7 = new Point(0,   0.5);



		corners= new LinkedList<Point>();
		corners.add(p1); corners.add(p2); corners.add(p3); corners.add(p4);

		corners2= new LinkedList<Point>();
		corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2.add(p5);
		corners2.add(p6); corners2.add(p7);

		room = new Room("test", null, corners);
		room2 = new Room("test", null, corners2);


		pc1 = new Point(1,-1);
		pc2 = new Point(2,-1);
		pc3 = new Point(2,1);
		pc4 = new Point(1,1);
		pc5 = new Point(1,2);
		pc6 = new Point(-1,2);
		pc7 = new Point(-1,1);
		pc8 = new Point(-2,1);
		pc9 = new Point(-2,-1);
		pc10 = new Point(-1,-1);
		pc11 = new Point(-1,-2);
		pc12 = new Point(1,-2);
		LinkedList<Point> cornersStar = new LinkedList<Point>();
		cornersStar.add(pc1);cornersStar.add(pc2);cornersStar.add(pc3);cornersStar.add(pc4);cornersStar.add(pc5);
		cornersStar.add(pc6);cornersStar.add(pc7);cornersStar.add(pc8);cornersStar.add(pc9);cornersStar.add(pc10);
		cornersStar.add(pc11);cornersStar.add(pc12);

		roomStar = new Room("star", null, cornersStar);


		p31 = new Point(-2,0);
		p32 = new Point(2,0);
		p33 = new Point(2,2);
		p34 = new Point(1,2);
		p35 = new Point(1,1);
		p36 = new Point(-1,1);
		p37 = new Point(-1,2);
		p38 = new Point(-2,2);
		LinkedList<Point> cornersHufeisen = new LinkedList<Point>();
		cornersHufeisen.add(p31);cornersHufeisen.add(p32);cornersHufeisen.add(p33);cornersHufeisen.add(p34);cornersHufeisen.add(p35);
		cornersHufeisen.add(p36);cornersHufeisen.add(p37);cornersHufeisen.add(p38);

		roomHufeisen = new Room("hufeisen", null, cornersHufeisen);

	}

	@Test
	public void testValidate() {
		//Arrange
		UserValidateAAS userValidateAAS = new UserValidateAAS();
		Lamp lamp1 = new Lamp(0,0);
		roomStar.addLamp(lamp1);

		//Act
		boolean test1 = false;
		boolean test2 = false;
		try {
			test1 = userValidateAAS.validate(roomStar);
			test2 = userValidateAAS.validate(room);
		} catch (UserValidateAASException e) {
			fail("Test result should have been found!");
		}

		//Assert
		assertTrue("Test should have been correct!", test1);
		assertFalse("Test should have been not correct!",test2);
	}

}
```

```java
package fernuni.propra.user_interface;

import static org.junit.Assert.*;

import java.awt.Color;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.file_processing.UserReadInputWriteOutputAAS;
import fernuni.propra.file_processing.UserReadInputWriteOutputException;
import fernuni.propra.internal_data_model.IRoom;
import fernuni.propra.internal_data_model.Lamp;
import fernuni.propra.internal_data_model.LineSegment;
import fernuni.propra.internal_data_model.Point;
import fernuni.propra.internal_data_model.Wall;

public class RoomFrameTest {
        Point p1,p2,p3,p4,p5;
        Wall w1,w2,w3,w4,w5;
        List<Wall> walls;
        List<IRoom> rooms;


        @Before
        public void setUp() {
                p1 = new Point (-1,-1);
                p2 = new Point (1,-1);
                p3 = new Point(1,1);
                p4 = new Point(-1,1);
                w1 = new Wall(p1, p2,0);
                w2 = new Wall(p2, p3,0);
                w3 = new Wall(p3,p4,0);
                w4 = new Wall(p4,p1,0);
                w5 = new Wall(p1, p3,0);

                walls = new ArrayList<Wall>();
                walls.add(w1);walls.add(w2); walls.add(w3); walls.add(w4);


                String[] xmlPathesOK = {"instances/validationInstances/Selbsttest_clockwise.xml",
                                        "instances/validationInstances/Selbsttest_counterClockwise.xml",
                                        "instances/validationInstances/Selbsttest_100a_incomplete.xml",
                                        "instances/validationInstances/Selbsttest_100a_incomplete.xml",
                                        "instances/validationInstances/Selbsttest_100a_solved.xml",
                                        "instances/validationInstances/Selbsttest_100a.xml",
                                        "instances/validationInstances/Selbsttest_100b.xml",
                                        "instances/validationInstances/Selbsttest_20a_incomplete.xml",
                                        "instances/validationInstances/Selbsttest_20a_solved.xml",
                                        "instances/validationInstances/Selbsttest_20a.xml",
                                        "instances/validationInstances/Selbsttest_20b.xml",
                                        "instances/validationInstances/Selbsttest_20c.xml"
                };

                rooms = new ArrayList<IRoom>();

                for(String xmlPath : xmlPathesOK) {
                        UserReadInputWriteOutputAAS readAAS = new UserReadInputWriteOutputAAS(xmlPath);
                        try {
                                rooms.add(readAAS.readInput());
                        } catch (UserReadInputWriteOutputException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }

        }

        @Test
        public void testRoomFrame() {

                //Arrange
                IRoom mockRoom = new IRoom() {
                        @Override
                        public Iterator<Lamp> getLamps() {
                                List<Lamp> lamps = new ArrayList<Lamp>();
                                Lamp lamp = new Lamp(0.0,0.0);
                                lamps.add(lamp);
                                lamp.turnOn();
                                return lamps.iterator();
                        }

                        @Override
                        public int getNumberOfLamps() {
                                // TODO Auto-generated method stub
                                return 0;
                        }

                        @Override
                        public Iterator<Point> getCorners() {
                                List<Point> corners = new ArrayList<Point>();
                                corners.add(p1); corners.add(p2); corners.add(p3); corners.add(p4);
                                return corners.iterator();
                        }

                        @Override
                        public void addLamp(Lamp lamp) {


                        }

                        @Override
                        public Iterator<Wall> getWalls() {
                                return walls.iterator();
                        }

                        @Override
                        public double getMinX() {
                                return -1;
                        }

```

```
			@Override
			public double getMaxX() {
				return 1.0;
			}

			@Override
			public double getMinY() {
				return -1.0;
			}

			@Override
			public double getMaxY() {
				return 1.0;
			}

			@Override
			public String getID() {
				return "MockRoom";
			}

			@Override
			public void replaceLamps(List<Lamp> lamps) {
				// TODO Auto-generated method stub

			}

			@Override
			public String printLampPositions() {
				// TODO Auto-generated method stub
				return null;
			}

		};


		RoomPanel mockRoomPanel = new RoomPanel(mockRoom);
		mockRoomPanel.addRectangle("Nr.1", Color.BLUE, 0.5, 0.5, 0.25,
0.25);
		//RoomPanel roomPanel = new RoomPanel(mockRoom);
		RoomFrame mockRoomFrame = new RoomFrame(mockRoomPanel);
		try {
			Thread.currentThread().sleep(3000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		mockRoomFrame.dispose();


		for (IRoom room : rooms) {
			RoomPanel roomPanel = new RoomPanel(room);
			RoomFrame roomFrame = new RoomFrame(roomPanel);
			try {
				Thread.currentThread().sleep(3000);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			roomFrame.dispose();
		}

	}

}
```

```java
package fernuni.propra.main;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;

public class MainTest {

	@Before
	public void setUp() throws Exception {
	}

	@Test
	@Ignore
	public void testUseCase_D() {
		//Arrange
		String[] commandLineParameters = new String[] {"r=d",
				"if=instances/validationInstances/Selbsttest_2
0b.xml"};
		String[] commandLineParameters2 = new String[] {"r=d",
		"if=instances/validationInstances/Selbsttest_20b.xml"};

		//Act
		Main.main(commandLineParameters);
		Main.main(commandLineParameters2);
		try {
			Thread.currentThread().sleep(1000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		//Assert
	}


	@Test
	public void testUseCase_SD() {
		//Arrange
		String[] commandLineParameters = new String[] {"if=instances/v
alidationInstances/Selbsttest_100b.xml", "r=sd", "l=-15" };

		//Act
		Main.main(commandLineParameters);

		try {
			Thread.currentThread().sleep(4000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}


	@Test
	public void testUseCase_V() {
		// Arrange
		String[] commandLineParameters = new String[] {"if=instances/v
alidationInstances/Selbsttest_20a_incomplete.xml", "r=v" };

		//Act
		Main.main(commandLineParameters);

		//Assert
	}

}
```

```java
package fernuni.propra.internal_data_model;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.List;

import javax.sound.sampled.Line;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;

import fernuni.propra.internal_data_model.LineSegment;
import fernuni.propra.internal_data_model.LineSegmentException;
import fernuni.propra.internal_data_model.Point;

public class LineSegmentTest {
        Point p1,p2,p3,p4,p5;
        LineSegment l1,l2,l3,l4,l5;
        List<LineSegment> lineSegments;


        @Before
        public void setUp() {
                p1 = new Point (0,0);
                p2 = new Point (1,0);
                p3 = new Point(1,1);
                p4 = new Point(0,1);
                l1 = new LineSegment(p1, p2);
                l2 = new LineSegment(p2, p3);
                l3 = new LineSegment(p3,p4);
                l4 = new LineSegment(p4,p1);
                l5 = new LineSegment(p1, p3);
                lineSegments = new ArrayList<LineSegment>();
                lineSegments.add(l1);lineSegments.add(l2); lineSegments.add(l3
); lineSegments.add(l4);

        }



        @Test
        public void testGetP1() {
                //Arrange
                Point p1 = new Point (0,0);
                Point p2 = new Point (0,1);
                LineSegment linesegment = new LineSegment(p1, p2);

                //Act
                Point px = linesegment.getP1();

                //Assert
                assertSame(p1, px);

        }

        @Test
        public void testGetP2() {
                //Arrange
                Point p1 = new Point (0,0);
                Point p2 = new Point (0,1);
                LineSegment linesegment = new LineSegment(p1, p2);

                //Act
                Point px = linesegment.getP2();

                //Assert
                assertSame(p2, px);
        }

        @Test
        public void testIsHorizontal() {
                //Arrange
                Point p1 = new Point (0,0);
                Point p2 = new Point (0,1);
                LineSegment linesegment = new LineSegment(p1, p2);
                LineSegment l2 = new LineSegment(null, p2);

                //Act
                boolean isHorizontal = linesegment.isHorizontal();
                try {
                        l2.isHorizontal();
                        fail();
                } catch (NullPointerException ex) {
                }


                //Assert
                assertTrue(!isHorizontal);

        }



        @Test
        public void testIsVertical() {
                //Arrange
                Point p1 = new Point (0,0);
                Point p2 = new Point (0,1);
                LineSegment linesegment = new LineSegment(p1, p2);

                //Act
                boolean isVertical = linesegment.isVertical();

                //Assert
                assertTrue(isVertical);
        }

        @Test
        public void testOverlapsXrange() {
                // Act
                boolean test1 = l1.overlapsXrange(0, 1);
                boolean test2 = l1.overlapsXrange(0.2, 2);
                boolean test3 = l1.overlapsXrange(-1, -0.001);
                boolean test4 = l1.overlapsXrange(-1, -0.000);
                boolean test5 = l1.overlapsXrange(1, 2);
                boolean test6 = l1.overlapsXrange(1.0001, 2);
                boolean test7 = l2.overlapsXrange(1.0, 1.0);

                //Act, Assert
                try {
                        l3.overlapsXrange(3, 2);
                        fail();
                } catch (IllegalArgumentException ex) {

                }

                //Assert
                assertTrue(test1);
                assertTrue(test2);
                assertTrue(!test3);
                assertTrue(test4);
                assertTrue(test5);
                assertTrue(!test6);
                assertTrue(test7);

        }

        @Test
        public void testOverlapsYrange() {
                // Act
                boolean test1 = l2.overlapsYrange(0, 1);
                boolean test2 = l2.overlapsYrange(0.2, 0.4);
                boolean test3 = l2.overlapsYrange(-1, -0.001);
                boolean test4 = l2.overlapsYrange(-1, -0.000);
                boolean test5 = l2.overlapsYrange(1, 2);
                boolean test6 = l2.overlapsYrange(1.0001, 2);
                boolean test7 = l1.overlapsXrange(0.0, 0.0);


                //Act, Assert
                try {
                        l3.overlapsXrange(3, 2);
                        fail();
                } catch (IllegalArgumentException ex) {

                }

                //Assert
                assertTrue(test1);
                assertTrue(test2);
                assertTrue(!test3);
                assertTrue(test4);
                assertTrue(test5);
                assertTrue(!test6);
                assertTrue(test7);
        }

        @Test
        public void testPerpendicularPointPoint() {
                //Act
                boolean test1 = l1.perpendicular(p2, p3);
                boolean test2 = l1.perpendicular(p1, p2);
                boolean test3 = l1.perpendicular(p1,p3);

                //Assert
                assertTrue(test1);
                assertTrue(!test2);
                assertTrue(!test3);

        }

        @Test
        public void testPerpendicularLineSegment() {
                //Arrange
                LineSegment ls1 = new LineSegment(p1, p3);

                //Act
                boolean test1 = l1.perpendicular(l2);
                boolean test2 = l2.perpendicular(l4);
                boolean test3 = l1.perpendicular(ls1);

                //Assert
                assertTrue(test1);
                assertTrue(!test2);
                assertTrue(!test3);

        }

        @Test
        public void testLineSegmentDoesNotIntersectLineSegments() {
                //Arrange
                Point pt1 = new Point(2*p2.getX(), p2.getY());
                Point pt2 = new Point (0.2, 0.2);
                Point pt3 = new Point (0.4,0.2);
                Point center = new Point (0.5, 0.5);
                Point pt5 = new Point(-0.5, 0.5);
                Point pt6 = new Point(1.5, 0.5);
                Point pt7 = new Point(0.5,1.5);
                Point pt8 = new Point(0.5, -0.5);

                LineSegment ls1 = new LineSegment(p2, pt1);
                LineSegment ls2 = new LineSegment( pt1, p2);
                LineSegment ls3 = new LineSegment(p1, p2);
                LineSegment ls4 = new LineSegment(center, pt5);
                LineSegment ls5 = new LineSegment(center, pt6);
                LineSegment ls6 = new LineSegment(center, pt7);
                LineSegment ls7 = new LineSegment(center, pt8);
                LineSegment ls8 = new LineSegment(pt2,pt3);

                // Act
                boolean test1 = ls1.penetratesLineSegments(lineSegments);
                boolean test2 = ls2.penetratesLineSegments(lineSegments);
                boolean test3 = ls3.penetratesLineSegments(lineSegments);
                boolean test4 = ls4.penetratesLineSegments(lineSegments);
                boolean test5 = ls5.penetratesLineSegments(lineSegments);
                boolean test6 = ls6.penetratesLineSegments(lineSegments);
                boolean test7 = ls7.penetratesLineSegments(lineSegments);
                boolean test8 = ls8.penetratesLineSegments(lineSegments);

                // Assert
                assertTrue(!test1);
                assertTrue(!test2);
                assertTrue(!test3);
                assertTrue(test4);
                assertTrue(test5);
                assertTrue(test6);
                assertTrue(test7);
                assertTrue(!test8);

        }

        @Test
        public void testIntersectionWithLinesegment() {
                //Arrange
                Point center = new Point (0.5, 0.5);
                Point pt5 = new Point(-0.5, 0.5);
                Point pt6 = new Point(1.5, 0.5);
                Point pt7 = new Point(0.5,1.5);
                Point pt8 = new Point(0.5, -0.5);
                Point pt9 = new Point(2.0, 0.0);
                Point pt10 = new Point(0.1, -4.0);
                Point pt11 = new Point(0.1, -6.0);
```

```java



		LineSegment ls4 = new LineSegment(center, pt5);
		LineSegment ls5 = new LineSegment(center, pt6);
		LineSegment ls6 = new LineSegment(center, pt7);
		LineSegment ls7 = new LineSegment(center, pt8);
		LineSegment ls8 = new LineSegment(p2, pt9);
		LineSegment ls9 = new LineSegment(pt10, pt11);

		//Act
		Point test1 = null;
		try {
			test1 = l4.intersectionWithLinesegment(ls4);
		} catch (LineSegmentException e) {
			fail("An intersection point should have been found!");
		}

		Point test2 = null;
		try {
			test2 = l2.intersectionWithLinesegment(ls5);
		} catch (LineSegmentException e) {
		}

		Point test3 = null;
		try {
			test3 = l3.intersectionWithLinesegment(ls6);
		} catch (LineSegmentException e) {
		}

		Point test4 = null;
		try {
			test4 = l1.intersectionWithLinesegment(ls7);
		} catch (LineSegmentException e) {
		}

		boolean test5 = false;
		try {
			l1.intersectionWithLinesegment(l1);
		} catch (LineSegmentException e) {
			test5 = true;
		}


		Point test6 = null;
		try {
			test6 = l2.intersectionWithLinesegment(ls8);
		} catch (LineSegmentException e) {
			fail(e.getMessage());
		}


		boolean test7 = false;
		try {
			l1.intersectionWithLinesegment(ls9);
		} catch (LineSegmentException e) {
			test7 = true;
		}

		boolean test8 = false;
		try {
			ls5.intersectionWithLinesegment(l4);
		} catch (LineSegmentException e) {
			test8 = true;
		}

		//Assert
		assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test1 != null && test1.isEqual(new Point(0, 0.5)));
		assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test2 != null && test2.isEqual(new Point(1.0, 0.5)));
		assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test3 != null && test3.isEqual(new Point(0.5, 1.0)));
		assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test4 != null && test4.isEqual(new Point(0.5, 0.0)));
		assertTrue("No intersection point should have been found",test5);
		assertTrue("Intersection needs to be at x = 0.0, y = 0.5", test6 != null && test6.isEqual(new Point(1.0, 0.0)));
		assertTrue("Not intersection pint should have been found", test7);
		assertTrue("Not intersection pint should have been found", test8);
	}

}
```

```java
package fernuni.propra.internal_data_model;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class WallTest {
	Point p1,p2,p3,p4;
	Wall w1,w2,w3,w4,w5,w6,w7,w8,w9;

	@Before
	public void setUp() {
		//Arrange
		p1 = new Point(0,0);
		p2 = new Point(1,0);
		p3 = new Point(1,1);
		p4 = new Point(0,1);
		w1 = new Wall(p1,p2,0);
		w2 = new Wall(p2,p3,0);
		w3 = new Wall(p3,p4,0);
		w4 = new Wall(p4,p1,0);

		w5 = new Wall(p2,p1,0);
		w6 = new Wall(p3, p2,0);
		w7 = new Wall(p4, p3,0);
		w8 = new Wall(p1,p4,0);

		w9 = new Wall(p1,p1,0);
	}

	@Test
	public void testIsNorthWall() {
		//Act
		boolean test1 = w3.isNorthWall();
		boolean test2 = w1.isNorthWall();
		boolean test3 = w2.isNorthWall();
		boolean test4 = w9.isNorthWall();

		//Assert
		assertTrue(test1);
		assertFalse(test2);
		assertFalse(test3);
		assertFalse(test4);
	}

	@Test
	public void testIsWestWall() {
		//Act
		boolean test1 = w4.isWestWall();
		boolean test2 = w2.isWestWall();
		boolean test3 = w1.isWestWall();
		boolean test4 = w9.isWestWall();

		//Assert
		assertTrue(test1);
		assertFalse(test2);
		assertFalse(test3);
		assertFalse(test4);
	}

	@Test
	public void testIsSouthWall() {
		//Act
		boolean test1 = w1.isSouthWall();
		boolean test2 = w3.isSouthWall();
		boolean test3 = w2.isSouthWall();
		boolean test4 = w9.isSouthWall();

		//Assert
		assertTrue(test1);
		assertFalse(test2);
		assertFalse(test3);
		assertFalse(test4);
	}

	@Test
	public void testIsEastWall() {
		//Act
		boolean test1 = w2.isEastWall();
		boolean test2 = w4.isEastWall();
		boolean test3 = w1.isEastWall();
		boolean test4 = w9.isEastWall();

		//Assert
		assertTrue(test1);
		assertFalse(test2);
		assertFalse(test3);
		assertFalse(test4);
	}

}
```

```java
package fernuni.propra.internal_data_model;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

public class RoomTest {

        Point p1,p2,p3,p4,p5;
        LineSegment l1,l2,l3,l4,l5;
        List<LineSegment> lineSegments;
        LinkedList<Point> corners, cornersClockWise;


        @Before
        public void setUp() {
                p1 = new Point (0,0);
                p2 = new Point (1,0);
                p3 = new Point(1,1);
                p4 = new Point(0,1);
                l1 = new LineSegment(p1, p2);
                l2 = new LineSegment(p2, p3);
                l3 = new LineSegment(p3,p4);
                l4 = new LineSegment(p4,p1);
                l5 = new LineSegment(p1, p3);

                corners= new LinkedList<Point>();
                corners.add(p1); corners.add(p2); corners.add(p3);corners.add(p4);

                cornersClockWise = new LinkedList<Point>();
                cornersClockWise.add(p1); cornersClockWise.add(p4); cornersClockWise.add(p3); cornersClockWise.add(p2);

                lineSegments = new ArrayList<LineSegment>();
                lineSegments.add(l1);lineSegments.add(l2); lineSegments.add(l3); lineSegments.add(l4);
        }

        @Test
        public void testGetWalls() {
                // Arrange
                Room room = new Room("test",null, corners);
                Room roomClockWise = new Room("test", null, cornersClockWise);

                // Act
                Iterator<Wall> wallIterator = room.getWalls();
                Iterator<Wall> wallIteratorClockWise = roomClockWise.getWalls();


                //Assert
                Wall w1 = wallIterator.next();
                Wall w2 = wallIterator.next();
                Wall w3 = wallIterator.next();
                Wall w4 = wallIterator.next();

                Wall w5 = wallIteratorClockWise.next();
                Wall w6 = wallIteratorClockWise.next();
                Wall w7 = wallIteratorClockWise.next();
                Wall w8 = wallIteratorClockWise.next();

                boolean test1 = w1.isEqual(l1);
                boolean test2 = w2.isEqual(l2);
                boolean test3 = w3.isEqual(l3);
                boolean test4 = w4.isEqual(l4);


                boolean test5 = w5.isEqual(l2);
                boolean test6 = w6.isEqual(l3);
                boolean test7 = w7.isEqual(l4);
                boolean test8 = w8.isEqual(l1);

                assertTrue(test1 && test2 && test3 && test4);
                assertTrue(test5 && test6 && test7 && test8);
        }

        @Test
        public void testGetLamps() {
                fail("Not yet implemented");
        }

        @Test
        public void testGetCorners() {
                fail("Not yet implemented");
        }

        @Test
        public void testAddLamp() {
                fail("Not yet implemented");
        }

        @Test
        public void testGetNumberOfLamps() {
                fail("Not yet implemented");
        }

        @Test
        public void testDimensions() {
                // Arrange
                Room room = new Room("test",null, corners);
                LinkedList<Point> corners2 = new LinkedList<Point>();
                Point p1 = new Point(-321.32,-432);
                corners2.add(p1); corners2.add(p2); corners2.add(p3); corners2.add(p4);
                Room room2 = new Room("test",null, corners2);

                //Act
                double xMin = room.getMinX();
                double xMax = room.getMaxX();
                double yMin = room.getMinY();
                double yMax = room.getMaxY();

                double xMin2 = room2.getMinX();
                double xMax2 = room2.getMaxX();
                double yMin2 = room2.getMinY();
                double yMax2 = room2.getMaxY();


                //Assert
                assertEquals(0.0, xMin, 0.0001);
                assertEquals(1.0, xMax, 0.0001);
                assertEquals(0.0, yMin, 0.0001);
                assertEquals(1.0, yMax, 0.0001);

                //Assert
                assertEquals(-321.32, xMin2, 0.0001);
                assertEquals(1.0, xMax2, 0.0001);
                assertEquals(-432, yMin2, 0.0001);
                assertEquals(1.0, yMax2, 0.0001);

        }


}
```

```java
package fernuni.propra.internal_data_model;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

import fernuni.propra.internal_data_model.LineSegment;
import fernuni.propra.internal_data_model.Point;

public class PointTest {
        Point p1,p2,p3,p4,p5;
        LineSegment l1,l2,l3,l4,l5;
        List<LineSegment> lineSegments;


        @Before
        public void setUp() {
                p1 = new Point (0,0);
                p2 = new Point (1,0);
                p3 = new Point(1,1);
                p4 = new Point(0,1);
                l1 = new LineSegment(p1, p2);
                l2 = new LineSegment(p2, p3);
                l3 = new LineSegment(p3,p4);
                l4 = new LineSegment(p4,p1);
                l5 = new LineSegment(p1, p3);
                lineSegments = new ArrayList<LineSegment>();
                lineSegments.add(l1);lineSegments.add(l2); lineSegments.add(l3); lineSegments.add(l4);
        }


        @Test
        public void testIsEqual() {
                //Arrange
                Point pt1 = new Point(0,0.01);
                Point pt2 = new Point(0.001,0.0);
                Point pt3 = new Point(131221.2,-500.7);

                //Act
                boolean test1 = p1.isEqual(pt1);
                boolean test2 = p1.isEqual(pt2);
                boolean test3 = pt3.isEqual(pt3);

                //Assert
                assertTrue(!test1);
                assertTrue(!test2);
                assertTrue(test3);
        }

        @Test
        public void testIsOnLineSegmentPointPoint() {
                //Arrange
                Point pt1 = new Point(0,0.01);
                Point pt2 = new Point(0.001,0.0);
                Point pt3 = new Point(131221.2,-500.7);
                Point pt4 = new Point(2.0,0.0);
                Point pt5 = new Point(4.0,0.0);

                //Act
                boolean test1 = p1.isOnLineSegment(p1,p4);
                boolean test2 = pt3.isOnLineSegment(p1,pt1);

                boolean test3 = false;
                try {
                        test3 = p1.isOnLineSegment(p2,p4);
                } catch(IllegalArgumentException e) {
                        test3 = true;
                }

                boolean test4 = p1.isOnLineSegment(pt4, pt5);

                boolean test5 = false;
                try {
                        pt3.isOnLineSegment(p1, p3);
                } catch(IllegalArgumentException e) {
                        test5 = true;
                }

                //Assert
                assertTrue(test1);
                assertTrue(!test2);
                assertTrue(test3);
                assertFalse(test4);
                assertTrue(test5);

        }

        @Test
        public void testIsOnLineSegmentLineSegment() {
                //Act
                boolean test1 = p1.isOnLineSegment(l1);
                boolean test2 = p1.isOnLineSegment(l4);



                //Assert
                assertTrue(test1);
                assertTrue(test2);

        }

        @Test
        public void testIsInsidePolygon() {
                //Arrange
                Point center = new Point(0.5, 0.5);
                Point onLine = new Point(1.0,0.5);
                Point onCorner = new Point(1.0,1.0);
                Point out = new Point(2.0, -10.0);

                //Act
                boolean test1= center.isInsidePolygon(lineSegments);
                boolean test2 = onLine.isInsidePolygon(lineSegments);
                boolean test3 = onCorner.isInsidePolygon(lineSegments);
                boolean test4 = out.isInsidePolygon(lineSegments);

                //Assert
                assertTrue(test1);
                assertTrue(test2);
                assertTrue(test3);
                assertTrue(!test4);
        }

        @Test
```

```java
	public void testIsInXRange() {

		//Act
		boolean test1 = p1.isInXRange(0.0, 0.0);
		boolean test2 = p2.isInXRange(1.0, 2.0);
		boolean test3 = p3.isInXRange(1.0001, 2.00);

		boolean test4 = false;
		try {
			test4 = p4.isInXRange(2, 1.9);
		} catch(IllegalArgumentException e) {
			test4 = true;
		}

		//Assert
		assertTrue(test1);
		assertTrue(test2);
		assertFalse(test3);
		assertTrue(test4);
	}

	@Test
	public void testIsInYRange() {
		//Act
		boolean test1 = p1.isInYRange(0.0, 0.0);
		boolean test2 = p3.isInYRange(1.0, 2.0);
		boolean test3 = p4.isInYRange(1.0001, 2.00);

		boolean test4 = false;
		try {
			test4 = p4.isInYRange(2, 1.9);
		} catch(IllegalArgumentException e) {
			test4 = true;
		}

		//Assert
		assertTrue(test1);
		assertTrue(test2);
		assertFalse(test3);
		assertTrue(test4);
	}

}
```

```java
package fernuni.propra.internal_data_model;

import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameter;
import org.junit.runners.Parameterized.Parameters;

import fernuni.propra.internal_data_model.LineSegment;
import fernuni.propra.internal_data_model.Point;

@RunWith(Parameterized.class)
public class LineSegmentTestParameterized {

        @Parameter(0)
        public LineSegment lp1;
        @Parameter(1)
        public boolean result1;


    // creates the test data
    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] { { new LineSegment(new Point(0,0), n
ew Point(1,0)), true },
                { new LineSegment(new Point(1,0), new Point(1,1)), false }, {
new LineSegment(new Point(1,1), new Point(0,1)), true },
                { new LineSegment(new Point(0,0), new Point(0,0)), false } };
        return Arrays.asList(data);
    }


        @Test
        public void testIsHorizontalParametrized() {
                //Act
                boolean isHorizontal = lp1.isHorizontal();

                //Assert
                assertTrue(isHorizontal==result1);;
        }




}
```