# Smart House Simulation Tool

Cyryl Krzyska

Kongens Lyngby 2006

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

# Preface

This document is a result of an MSc project carried out between 18th April 2006 and 29th September 2006 at the department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU) under supervision of Associate Professor Christian D. Jensen, to whom I would like to thank for his guidance and support.

# Abstract

Smart house installations are becoming more and more popular all over the world. They help to improve the quality of a daily life, reduce costs and increase security.

The main goal of this project is to develop a JAVA tool for designing and simulating simple 2D model of a smart house. The tool provides the ability to design the physical boundaries of a house, position a set of sensors inside it, and observe the operation of a house through a set of scenarios, where human behaviour will trigger the sensors. Both sensors infrastructure and building of scenarios are designed in a way that enables their easy extension in the future. Program is logging its operation, giving ability to experiment with scenarios and check their validity.

As conclusions extension possibilities are outlined, with the most important of them being development of logic unit (programmable or adaptive), governing the house operation, making the software a real smart house simulator.

KEYWORDS:
Smart Houses, Intelligent Houses, JAVA Swing,

# Contents

# Chapter 1

# Introduction - Why Smart Houses?

Today's busy life, with days filled out almost to the last minute with an activities mixture, of both professional and private nature, leaves us tired when we get home in the evening. Yet coming home does not necessarily mean end of time- and energy-consuming repeatable activities, we shall do every evening before we head towards shower and bed - making sure all lights and hi-fi system are switched off, window blinds rolled down, and heating turned a bit down, to name just a few. It would be much nicer if 'something' checked for us if all windows and outside doors are properly locked, and informed us if it was not the case. Newly baked parents would certainly demand from the lights in the house to shine just at 30 percent of their nominal power at night, so their eyes are not annoyed by full brightness when they go to check why the baby cries at 3 o'clock in the morning. Implementation of the abovementioned scenarios is not rocket science – it requires only a well-thought network of interconnected sensors and devices and some logic that will govern that network and enable house owner to program some tasks he wants to get done automatically.

With electronics becoming cheaper and cheaper every single week, and many of such installations installed worldwide, there is steady increasing audience interested in the topic. Many families would like to try to check how their house could be improved. Some are considering installation of such technology in the house they want to build in the nearest future. There are a number of distributors offering smart house solutions, with a developed network of consultants offering help; however there are people that want to spend more time on choosing array of sensors and actuators that will best suit their needs – and they would appreciate software that will help them to design such array.

There are software packages available commercially, most of them designed for professionals, some with a price as high as 900 Euro [1] – hefty price for an individual just wanting to check if and how this new technology will work for his family.

A Smart House Simulation Tool developed during this MSc project comes up to these expectations. It allows a user to draw a floor plan of his choice, position various sensors on it, and define a variety of scenarios to check how these sensors will react to inhabitants' behaviour in different situations. Focus has been put to develop an extensible architecture, and a limited amount of features has been implemented to demonstrate possibilities of the framework.
The rest of this thesis is organized in the following way. Chapter 2 presents a classification of smart homes, with a brief division into programmable houses (possessing no intelligence),

and intelligent houses (with intelligence adapting to changing user behaviour). Chapter 3 will bring design description of a smart house simulation tool, whose implementation will be further elaborated in fourth chapter. Finally, evaluation and testing of the tool will be clarified and some conclusions drawn during project will be presented at the end of this paper.

# Chapter 2

# Smart Houses

The most basic reason for which people choose smart homes is to be able to automate simple things, like lighting, heating and air conditioning controls.

Another important aspect coming from the usage of a smart house is energy conservation. By proper adjustment (programming) of lights operated by movement sensors only lights that are actually needed are turned on.
Even greater savings can be achieved by fine tuning the heating system. Here several solutions exist – either programming the heating to be turned on during prescribed hours, or program the controller to heat only when people in the house are detected (e.g. basing on movement sensors). Heating can be also turned down during night, when it is not needed during inhabitants' sleep. Another possibility is to coordinate operation of window blinds and heating. In period of high sunlight window blinds should stay open to warm the room using sun energy. In winter, however, windows are source of greatest heat loss; hence window blinds should be kept closed.

Worth mentioning is also aspect of safety and security. Lights and/or audio equipment can be programmed to turn on randomly to simulate inhabitants' presence in the house, while they are away from home. Good programmed system will also note intruders in the house – by simply noting person presence in a previously unoccupied room. Possibility of turning on all light in the house by one button will be appreciated by people afraid of being alone in the house.

A myriad of other applications are possible, such as controlling kitchen appliances (dishwasher, fridge, coffee machine).

We could quote different possible usage scenarios and positive outcomes of smart homes forever, let us now divert into a brief classification of smart homes.

Smart houses could be divided into two main categories:

- Programmable houses – are those programmed to perform an action triggered by a sensor input,
- Intelligent houses – are those that possess some kind of intelligence that govern its operation.

## *2.1 Programmable Houses*

Programmable houses will be those, whose reactions are based only on simple sensor inputs, and possess no built-in intelligence. Such a house for a predefined input (from either sensor(s) or user controller(s)) has a predefined (programmed) set of actions to perform. Examples of such actions might be e.g. light bulb operated by a movement sensor, or selection of one of the predefined lighting settings by a user pressing button on his remote controller.

Virtually all of currently manufactured and sold 'smart house' systems belong to this group. With mass production it was possible to lower the end-user price to a level acceptable by middle-class customers – cost can lie between 50 and 120 (and up) Euro per square meter [2], depending on a set of desired features and controlled devices.

The biggest problem with this type of houses is that they have to be reprogrammed when some of the features become unnecessary and/or added. That presents a problem for many people and requires calling a technician to get the job done. Hence increasing tension to develop some smart home solution that - basing on artificial intelligence - will adapt its operation to changing user behaviour. A house that will not have to be first programmed and then reprogrammed all the time. A house that will think how to make a user satisfied, lower his workload and lower the house operation cost at the same time.

That tension leads to development of the houses that belong to the second category.

## *2.2 Intelligent Houses*

They represent the state-of-the-art technology. Those type of installations are driven by artificial intelligence, and instead of having to be programmed they are able to learn (i.e. program themselves) basing on observation of inhabitants behaviour over a period of time.

It one of the first successful implementations was well known Adaptive House developed by M. Mozer at University of Colorado back in 1998.

Some other examples that belong to the group of intelligent houses are:

- Georgia Tech Aware Home
- AIRE spaces at MIT
- Interactive Workspaces Project at Stanford
- Gaia project at UIUC
- MavHome project at UTA

Complete system could be decomposed down to building blocks, which could be grouped into following categories:

- Data mining
- Decision making

### 2.2.1. Data mining

It is an area known from e.g. statistics or pattern recognition. It could be defined as searching large volumes of data for patterns [3].

In case of intelligent house it will be implemented as looking for a specific behavior pattern through the saved logs of past activities, and trying to match the specific situation to a previously observed one.

One problem is that constantly growing amount of data (from sensors and actuators operation) demands a method for storage enabling short data access time. Second problem is to fine tune a method to 'mine' the data out from the database, so it can be used by the next stage of the intelligent house system. Several techniques to perform these operations are presented with some details in [4].

Having (possibly close) matched the current scenario to one from the past, that module passes the data to the next, far more complicated module that is responsible for making decisions.

### 2.2.2. Decision Making

The role of the Decision Making section is, as name implies, to make a decision of what action should be taken by the house, basing on the input from sensors and data fed by data mining unit. There is plenty of logic that could be implemented here. Decisions have to be performed accurately, because some everyday life situations can be potentially dangerous (like leaving a meal on cooker unattended).

In the abovementioned case different actions can be done:

1. Alert the inhabitant immediately by voice message or similar if he leaves kitchen and heads to another room.
2. Turn the heater down (or switch it off completely), thus eliminating danger without the need of user intervention.

Second aspect is how long house should let the cooker work unattended before any action is undergone. Time should be tailored to safety, it should always come first. But system should also learn that it could be a case that inhabitant still remembers his food is being cooked, and do not take any action for some (safe) time. In this case there has to be a delicate balance between safety and user satisfaction, that could be disturbed by voice messages and/or switching his food off.

Both of these decisions present a problem tailored for an Artificial Intelligence (AI) system, with possible solutions discussed in some recent papers, [5] gives a good overview of implemented solutions.
Developing logic for intelligent house is a major task, far beyond the capability of a single person, and is supposed to be carried out by a group of researchers, in most of the cases they are university scientists.

## *2.3 Smart House System/Simulator*

Smart house systems concentrate on providing assistance to an inhabitant while he is dealing with his daily life. Most simple solutions control only lighting and HVAC systems, basing on pre-programmed settings or simple sensor input (detection of movement, coupled with absence of natural light, triggers light on). With adding more types of sensors and more types of actuators operating new kinds of devices we can increase number of features in our smart house, enabling it to control more aspects of house operation, like automated curtains, kitchen appliances, and provision of security. The most important extension, still being in experimental stage and waiting for deployment in broad scale, is implementing intelligence to collect data from this complicated network of sensors, process them, and control devices in an adaptive way.

Those are the features necessary to create a smart house. In case of creating a smart house simulator, many things are simpler. We do not have to create the entire technical infrastructure, because action takes place on the computer display. However, we have to give user a way to define a house outline, to place and configure sensors, and a framework to simulate all events taking place in the house. In the next subchapter we present a more detailed list of requirements a smart house simulator should fulfil.

## *2.4 Requirements for the Simulator*

The smart house simulator is supposed to fulfil the following conditions:

1. Should be easy to use, to attract possibly broadest audience.
2. Should enable defining a house outline, so that user can define a house on his choice, instead of playing only with built-in templates.
3. Should enable placing sensors on the plan, and define their characteristics (e.g. detection radius and range for movement sensors).
4. Should enable defining a scenario, which may be:
    - single inhabitant moving,
    - several inhabitants moving,
    - influence of outdoor environment (weather and seasons) on the house environment (like indoor temperature and humidity),
    - combination of the above.
5. Should enable user to bind a specific action to specific sensor input (basic rule of operation of a programmable house). Example: inhabitant enters a room, is detected by movement sensor, the light in a room goes on.
6. Should possess ability to learn from repeated scenarios (basic rule of operation of an intelligent house). Example: House detects that on every afternoon inhabitant comes home, goes to his room and takes a nap. After several repetitions system learns to automatically turn on lights on his way to a room, then turns the lights off and switches off any ringing devices in the house.

## *2.5 Summary*

In this chapter, advantages of smart house technologies were outlined, together with a short classification of currently available solutions. Distinction between programmable houses and intelligent houses were outlined, basing primarily on the presence or absence of a self-learning logic unit, trying to fit the house infrastructure operation to inhabitant behaviour that may change over time.
Features implemented in current projects were outlined and summarized in section 2.4, and given as a brief requirements list such a smart house simulator should fulfil. Those requirements will be considered in the discussion of design issues that will follow in the next chapter.

# Chapter 3

# Simulation Tool - Design

The application should be designed to allow user interested in smart house technology to experiment with the smart house environment. Main idea is to provide a user with a freeware, simple to use tool to check if and how house of his choice could be improved to be a better place to live.

Because of time limitations many of the ideas from previous chapter (like designing and implementing programmable or intelligent logic) had to be dropped. What we found out to be feasible to accomplish during this MSc project is to create a program that could serve as a solid basis for future development. It lets the user define the outline his own apartment, position sensors inside it, and to observe the operation of the house to check if it fulfils his expectations. However, it is important to note that no matter how poor with features, it should be designed in a way allowing easy extension by adding new features.

The main application window contains three tabs:

- a drawing tab (Compose tab)
- a main simulation tab (Run tab)
- an event logging tab (Logs tab)

## *3.1 Compose tab*

It fulfils the first two requirements for a simulator. First, it gives user means to define an apartment outline, which may consist of walls, windows and doors. Also, sensors, such as movement, light, time and temperature sensors may be placed onto a plan.

Due to project time constraints experimenting we decided to limit our simulations to 2D apartment plans. Also, movement sensors with a 2D field-of-view are used in the simulation to simplify the implementation.

Very important feature of the drawing part should be an option to load a background image showing some existing apartment plan from a file before drawing. It will be included to help user to redraw existing apartment plan i.e. user may load an existing plan as a background and use the application to follow existing apartment outline.

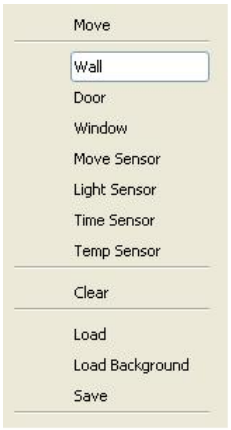



Fig.3.1: Draw Menu

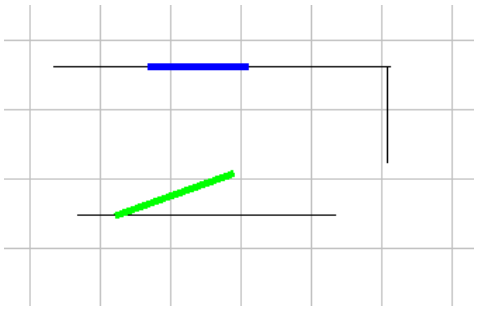


Fig.3.2: Example of drawing

Drawing will take place by means of choosing an apartment element and positioning them onto a plan. There are no constraints set on where a wall can be positioned. Other elements, i.e. doors and windows, may be put only on walls and the application does not allow putting them anywhere else.

Since in majority of cases walls are perpendicular with respect to each other, there will be an option to automatically position elements – it is enough to keep a Shift key pressed as elements are put onto a panel, and the application will automatically position them either vertically or horizontally. Hence there should be no need for a user to struggle to position walls precisely perpendicular – software will do it for him.

As far as movement sensors are concerned, user also sets their radius of detection.

There should be a possibility to clear any unwanted elements (by using Delete key). Also moving elements is implemented – it is important to note that wall is moved together with any doors and windows present on the wall.

Finally, possibility to save or load a plan is implemented, to facilitate possibility of future changes in the plan without the need of tedious redrawing.

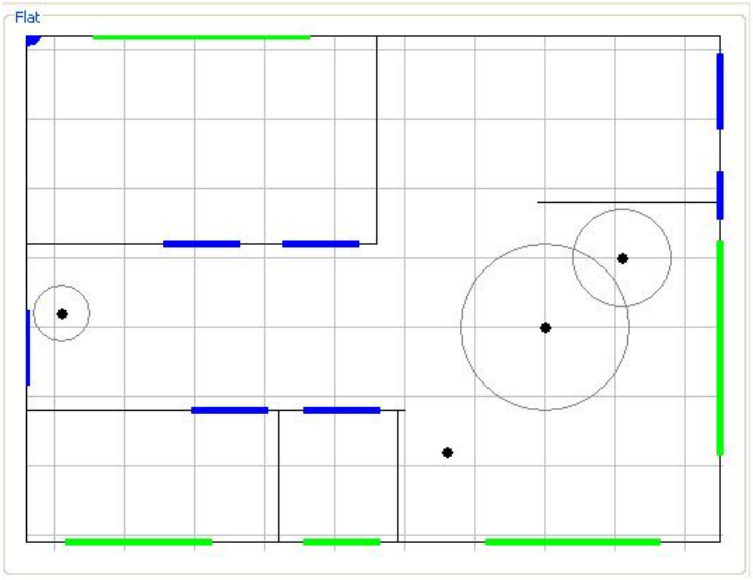



Fig 3.3: Floor plan example

## *3.2 Run tab*

This tab is designed to enable simulation of an operation of smart house sensors triggered by inhabitants' movement in the apartment. As inhabitant moves inside rooms, sensors placed on his way activate/deactivate, sending messages to a house logic that, when properly implemented and programmed in the future, may govern a network of actuators that perform useful function, such as provide intelligent lighting, heating and air conditioning control, and so on. Due to time constraints, currently program is only designed to generate log messages displayed on the screen and sent to a log file.

Generally there are two types of simulation from an end-user perspective:

- user may either do a simulation at runtime i.e. change a person position by means of mouse clicks
- user may design a scenario and run it.

The 'runtime simulation' is just triggered by a mouse click pointing to any position in the apartment. It provides a possibility to check user-environment interaction during simple consecutive movements of inhabitant in the apartment.

Scenarios give user a possibility to create a complex movement of an inhabitant and to check interaction of this movement with the house sensor network. Example scenario is depicted on a figure below.

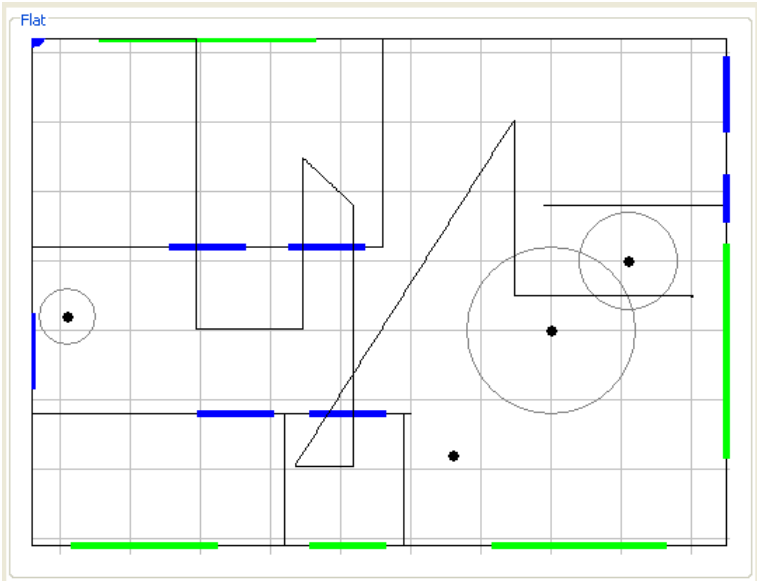



Fig.3.4: Example scenario

Scenarios designed by user may be saved to a file, which provides convenient method of storing them to be replayed in the future when needed. Common scenarios may be even published on the Internet. This gives the possibility to collect various scenarios, group them according to a various criteria and replay them.

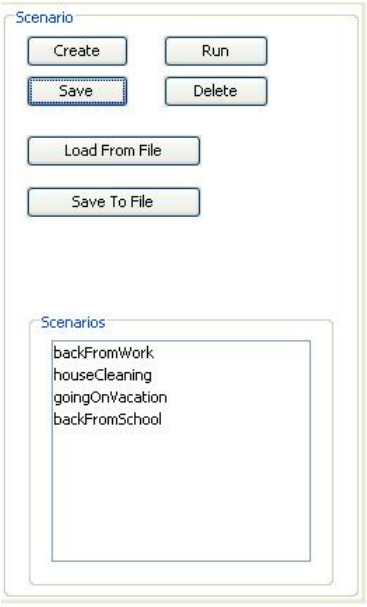



Fig.3.5: Scenario panel

In case the user is not satisfied with a particular sensors' placement, its coordinates and radius of operation can be changed directly using *Config* panel, without the need of coming back to drawing tab.

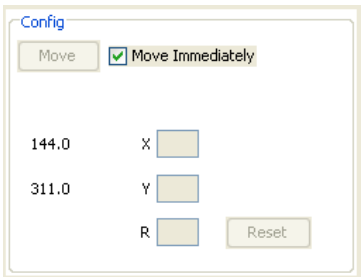



Fig.3.6: *Config* panel

For convenience *Config* panel provides also means to change sensors' parameters; modification is accomplished by right-click on a movement sensor present on an apartment plan and input of new parameters of text boxes in *Config* panel to set new values for a chosen sensor.

Only movement events may take place in a scenario in a current version of the application although application is extensible to incorporate other events in future versions (like climate changes).

User is provided with an option to temporarily turn off sensors of certain types without an apartment plan modification. To do that, *Sensors* panel shown below is introduced.

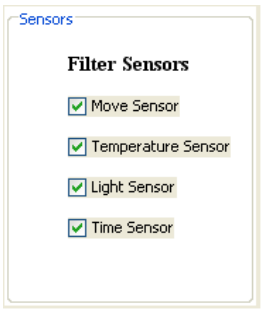



Fig.3.7: Sensors panel

There is also a *Temperature* panel where a user may simulate temperature changes. However it is rather experimental, as a scrollbar must be used to change temperature i.e. temperature changes may not take part in a scenario yet. A *Time* panel is put on a simulation tab only as a placeholder for future additions to the application; it is not used currently.

## *3.3 Logs tab*

Last programs' tab was introduced to allow user to check the validity of the implemented smart house environment operation. It shows log messages triggered by various events occurring during program runtime. These events may be:

- coordinates of user clicks during scenario creation
- sensors operation (activation/deactivation)
- listing of files with apartment description that are being loaded

Logs are also saved to a file, so that they may be further compared and analyzed independently from the application itself. Additionally the simulation tab contains a similar panel with a text area which displays currents logs:

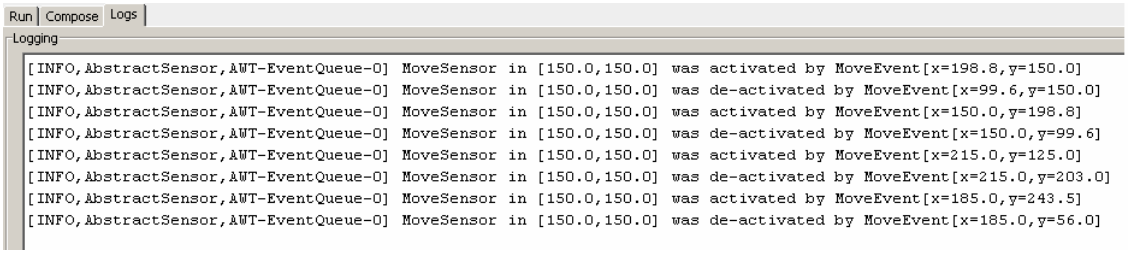



Fig.3.9: Log panel

The logging is declared declaratively at deployment time. The end-user may choose what type of events to log and may define which sensors are to log events. Although the configuration is very simple it requires a basic knowledge of a well known Log4J library.

## *3.4 Summary*

In this chapter the design scheduled for implementation was presented. Because of time constraints, some features (e.g. introducing actuators and implementing intelligence) have been omitted. Program is designed to enable user to draw an apartment outline consisting of walls, doors and windows, and position sensors in it. Floor plan may be saved for future use. User can also design a scenario consisting of inhabitants' moves around the house, to observe operation of sensors as inhabitant enters their field-of-view. Scenarios also have an option to be saved; hence they can be replayed and analyzed at any time.

# Chapter 4

# Simulation Tool - Implementation

Smart House Simulation Tool has been written in Java 6 beta 2 using:

- Core Java features and libraries
- Java Swing Components
- Log4J logging framework
- XMLBeans library
- Apache Jakarta Commons library
- JUnit framework
- Netbeans Matisse GUI builder
- Ant build scripts

## *4.1 General Structure*

General structure of the software is presented on the figure below. We can clearly observe user interaction triggers events, which are caught by GUI Events Dispatcher. Those events are further dispatched to services (which contain all application logic) through an intermediate Façade layer (three façades are currently implemented). All auxiliary stateless logic is grouped in Utility classes.

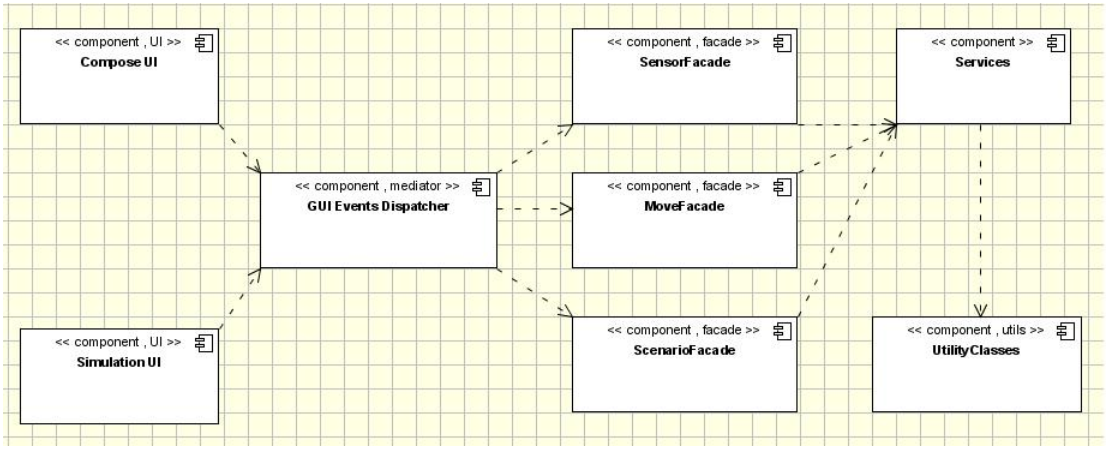



Fig.4.1: General structure

## *4.2 Packages*

All the sources are divided into several packages, each of them responsible for a part of logic:

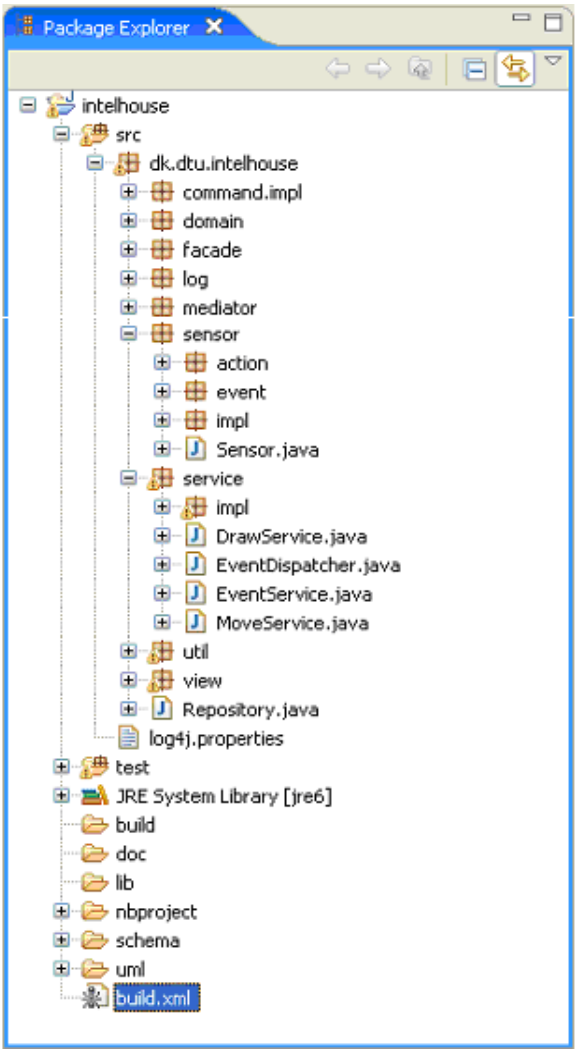



Fig.4.2: Packages

In order to keep the code clean several packages have been used:

- **Domain package** – contains classes used in a *Compose* tab which represent domain objects like walls, doors etc
- **View package** – contains classes that represents widgets used to display the data to an end-user
- **Mediator package** – contains classes that accept events caught by classes in *View* package. Those events are dispatched further to *Façade* layer.
- **Façade package** – contains classes that are main entry point to the applications' actual logic; they are responsible for executing use-cases and as such may wrap one or more services
- **Service package** – contains classes with the actual logic; usually to execute a given use case more than one service is used
- **Sensor package** – contains the whole sensor part i.e. sensors themselves, events that turn up during simulations and actions that are triggered by sensors in response to some events and an action executor

- **Util package** – contains classes with static utility methods only. Service layer uses this package as some stateless logic is delegated there.
- **Log package** – contains Log4J extension class
- **Command.impl package** – contains classes that implement *Command* pattern

In each top-level package one will find interfaces that state contracts between the application layers. Most top level packages include **impl** packages that contain the actual implementation of interfaces.

Sensor package structure, which will be covered in details later, is depicted as an example on the figure below.

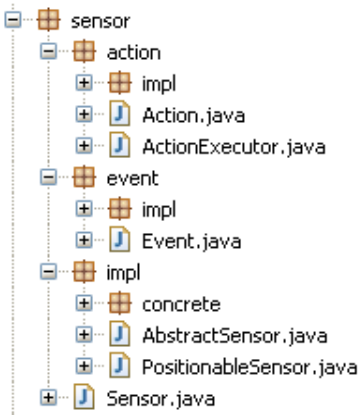



Fig.4.3: Sensor package

The main idea was to divide the logic into:

- Data processing part
- Data displaying part

One can see clearly the division if you take a look at the following two classes:

- DrawServiceImpl.java – It is used by other services. It accepts the data collected by other services and **pushes** that data onto a PaintPanel.java. Then is calls panel.repaint() method which suggests a Swing framework that it should trigger painting of a selected component.

- PaintPanel.java – It is used by a draw service i.e. the data that a panel is to display is set on it by a draw service. Therefore when a panel is triggered by Swing framework to display itself, the panel does **neither collects nor modify** data. It just assumes the data is already there and just paints it using Graphics object passed to it by Swing framework.

## *4.3 Mediator and Façades*

### 4.3.1 Mediator Pattern Implementation

The Mediator pattern [6] is implemented by a *MediatorImpl.java*. During the initialization of the application *MainFrame* creates GUI components like buttons, text fields etc. and registers them with mediator while at the application runtime GUI events are 'caught' by *MainFrame* (i.e. anonymous inner classes for listeners implementation [7] were used) and passed to a mediator.

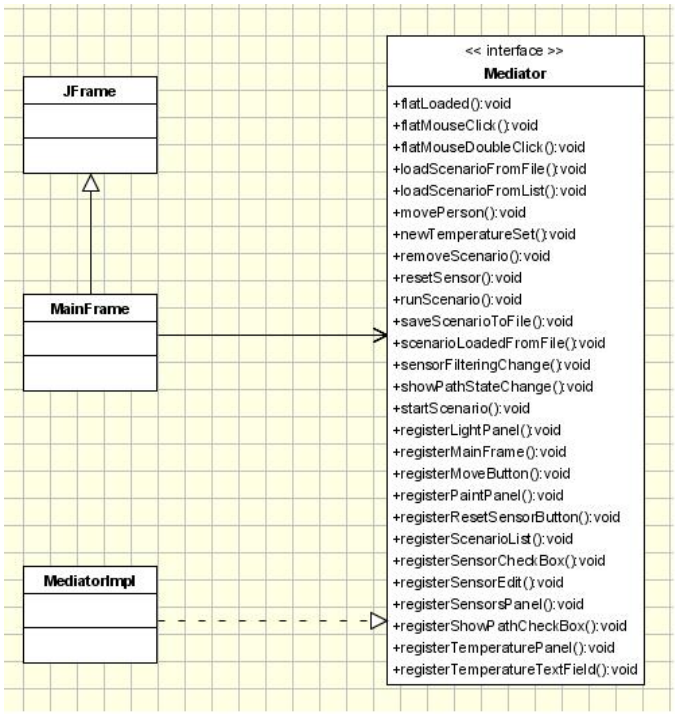



Fig.4.4: Mediator-widgets association UML class diagram

As it can be seen on the UML diagram, mediator methods are of two kinds:

- Ones that start with *register* which are part of *Mediator* pattern implementation
- All others, which are processing GUI events

### 4.3.2 Façade Layer

As mentioned previously, mediator acts as a layer between GUI and services. It:

- accepts events from *MainFrame*,
- converts GUI objects to domain objects (e.g. *ConversionUtils.java*),
- invokes services to complete a use-case.

During the development it turned out that it would be beneficial to introduce an additional level of indirection between a mediator and a services layer - for larger GUI applications a single mediator pattern may lead to a too large mediator class. The solution could be either to divide a 'master' mediator to a few mediators each responsible for only a part of GUI or to change the API of services layer. Second solution was chosen and wrappers around services were introduced. These wrappers are called *façades*. As Façade classes wrap more, their API is more coarse-grained, therefore it is easier for mediator to invoke a façade once than do several invocations of service layer methods.

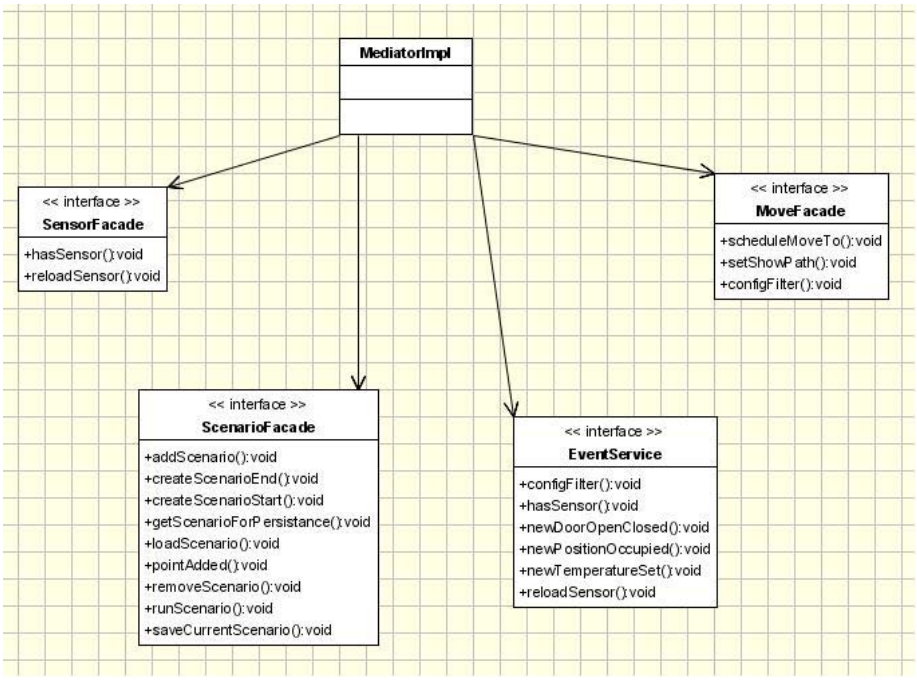



Fig.4.5: Mediator and scenarios interaction

Flexible approach to the design was preferred. Therefore mediator uses *EventService* directly as there were no problems to wrap this specific service. *EventService* API is quite easy to use and implementation of additional event façade would not provide any benefits.

### 4.3.3 Façade Layer and Mediator Cooperation

To illustrate this section better a UML sequence diagram for a typical use-case is presented. It can be observed that user generated GUI events are caught by Java Swing framework and further passed to proper handlers in *MainFrame.java* which is actually a main frame in the application. *MainFrame.java* is responsible only for passing these events further to a mediator in a form of invoking proper mediator's methods. Mediator starts a use-case by invoking a façade that provides a convenient API. The use-case implementation details are hidden behind a façade, so mediator only accepts and processes a use-case result:

- Either success which typically results in an state change visible to an end user like a movement
- Or failure, which typically results in an error message displayed to an end user

Additionally there is a common *FacadeInvocationResult* class which serves as a result object of façade invocation. There is a mechanism on a mediator site that processes these results in a common and consistent way e.g. there is a mechanism that alerts a user with message dialogs:

```
private Boolean reactToResult(FacadeInvocationResult result) {
        return reactToResult(result, true);
}

private boolean reactToResult(FacadeInvocationResult result, boolean showMessageWhenOkSwitch) {
        final int status = result.getStatus().intValue();
        final String message = result.getMessage();
        reactToResult(message, status, showMessageWhenOkSwitch);
        return result.getStatusAsBoolean();
}

private void reactToResult(final String message, final int status, boolean showMessageWhenOkSwitch) {
        switch (status) {
        case FacadeInvocationResult.NOT_OK:
                JOptionPane.showMessageDialog(mainFrame, message, DIALOG_WARNING_LABEL, JOptionPane.WARNING_MESSAGE);
                break;
        case FacadeInvocationResult.OK:
                assert StringUtils.isNotBlank(message) : "I don't want empty messages to pass to an end-user";
                if (showMessageWhenOkSwitch) {
                        JOptionPane.showMessageDialog(mainFrame, message, "info", JOptionPane.INFORMATION_MESSAGE);
                }
                break;
        default:
                assert false : status;
        }
}
```

A façade provides a coarse-grained API and communicates with a mediator mainly by means of *FacadeInvocationResult* objects. However this is not enforced and when convenient is omitted, directly returning other objects to a mediator.

### 4.3.4 Move Use Case

Use-case is implemented in a façade i.e. it is a façade that knows what services to invoke and in which order. A façade does not do actual computations, like checking whether a given move path is possible, and does not do e.g. an actual drawing on a panel. The implementation details are present mostly in services.

However, when a use-case consists of several users' GUI actions, there is often a need for storing a state of where we are in a given use-case. A state is kept in a façade and is represented by simple objects i.e. there is no common, consistent way across the application to keep the state. For example when you consider a person movement between two points set by an end-user you will find a *moveIterator* object in a *MoveFacadeImpl* class. When inhabitant moves, a *moveIterator* state is changed by a façade (i.e. it fetches consecutive positions from an iterator) and a façade passes them to a move service. So in case of a scheduled move a state is represented by an iterator. Associations of move façade with services are depicted below.

On the next page a typical use-case is presented.

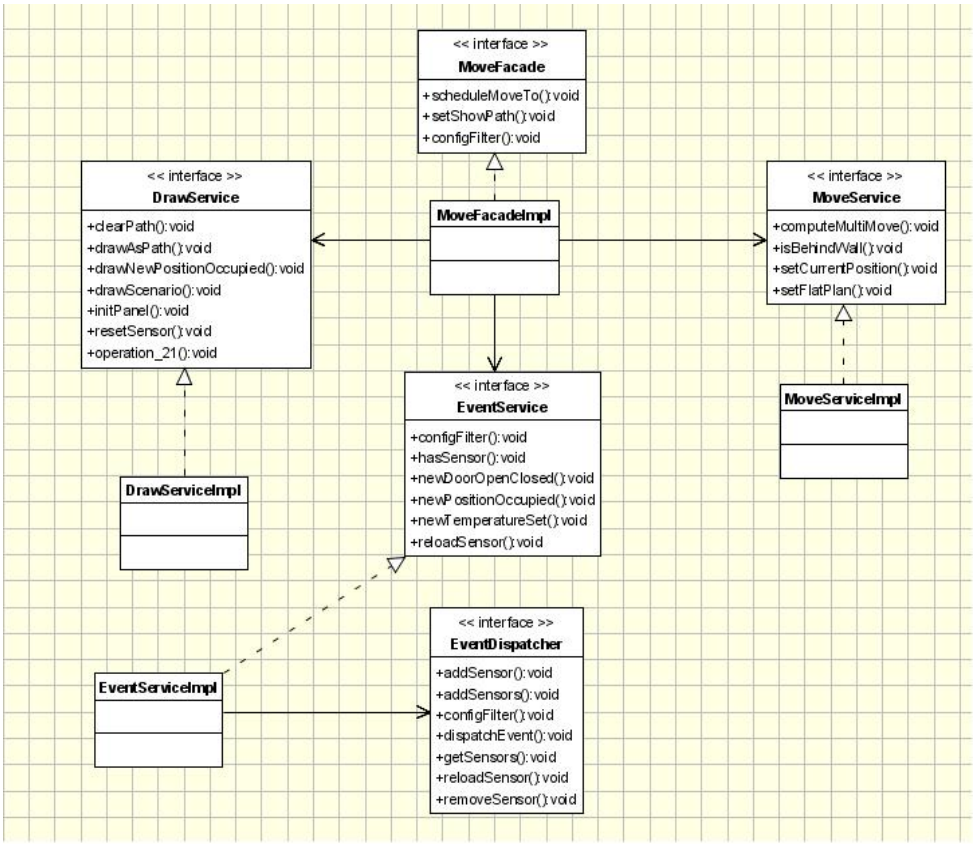



Fig.4.6: Move façade–services associations UML class diagram

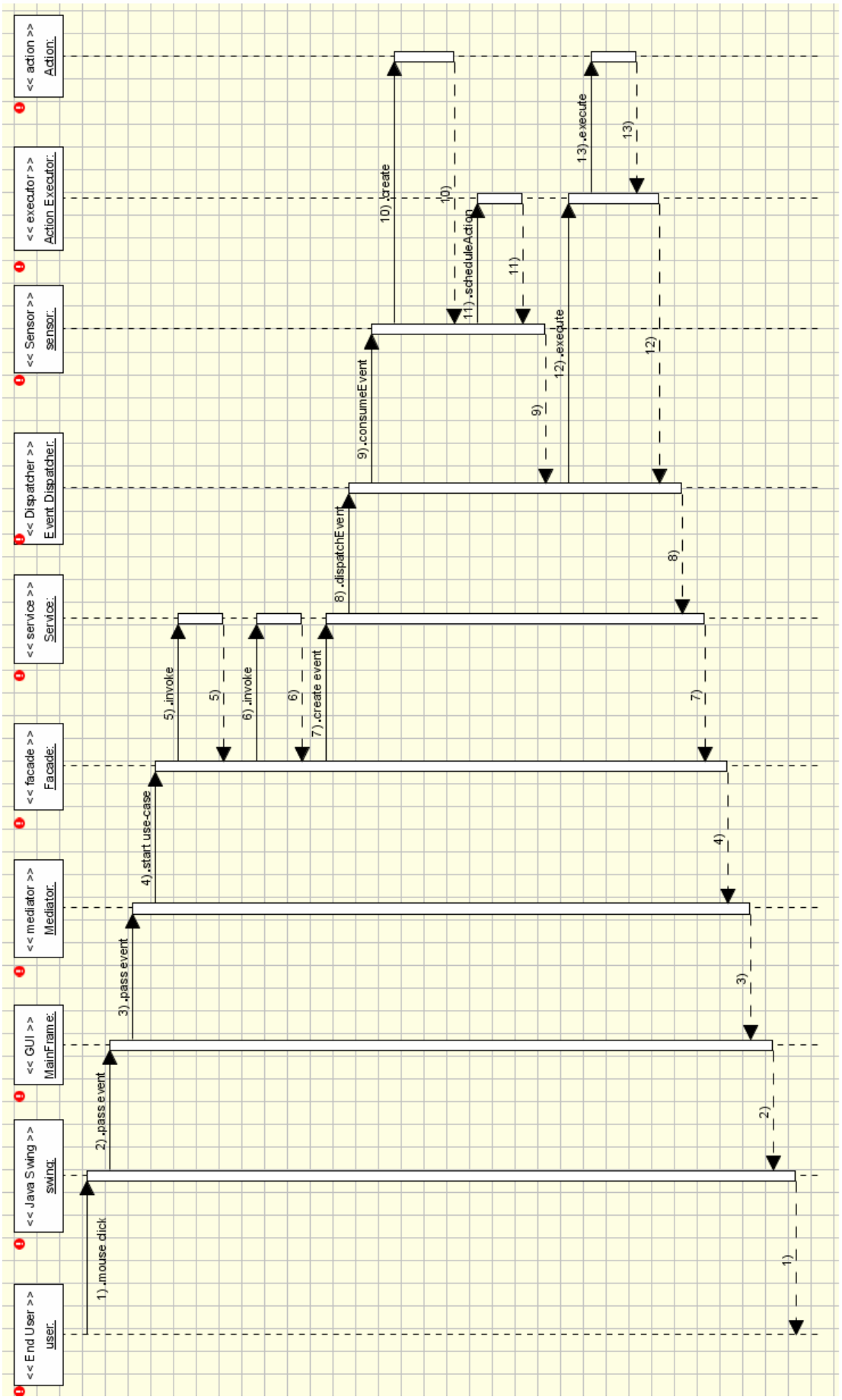



Fig.4.7: A move use-case UML sequence diagram

## *4.4 Defining House Outline*

As far a drawing is concerned, two approaches were considered:

- Pull approach – when implementation JPanel (*PaintPanel*) knows about a model i.e. a flat and knows how to interpret this domain model i.e. knows how to draw a model
- Push approach – when a service uses public *PaintPanel* setters to push a model graphical representation onto a panel object.

Both approaches have their pros and cons. Pull approach requires that GUI widgets contain the logic needed to extract the drawing information from a model. In this solution a double-dispatch approach may be employed to pass the information to a display widget about model changes. A widget may register itself as a listener to a model and each change to a model would result in an event firing caught by a widget. This approach requires non-trivial model and non-trivial widget.

Approach with an additional class (a draw service) which is informed about all model changes in an application domain form, was preferred. This class is responsible for converting these changes to low-level entities (like *java.awt.Shape*). These entities are further set on a presentation widget. This way a presentation widget needs only to iterate through low-level entities and need not have any special logic to display them.

Push approach was chosen as it makes an actual painting for a JPanel much easier.

A *PaintPanel* class has a few lists that store line objects (*Line2D*) which are set by a drawing service. This way drawing does not require from a panel much logic, in fact only a simple iteration through the lists is needed:

```
private Set<SensorImg> sensorLines = new HashSet<SensorImg>();
private Set<Line2D> wallLines = new HashSet<Line2D>();
private Set<Line2D> doorLines = new HashSet<Line2D>();
private Set<Line2D> windowLines = new HashSet<Line2D>();
private Set<Line2D> scenarioLines = new HashSet<Line2D>();
private List<Line2D> pathLines = new ArrayList<Line2D>();

public void paintComponent(Graphics g) {
        log.debug("paintComponent method start");
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        drawGrid(g2);
        drawFlat(g2);
        drawSensors(g2);
        drawScenario(g2);
        drawPath(g2);
        drawPerson(g2);
}
```

Below is an example how scenario paths are drawn:

```
private void drawScenario(Graphics2D g2) {
        for (Shape wall : scenarioLines) {
                g2.setColor(Color.ORANGE);
                g2.setStroke(new BasicStroke(1));
                g2.draw(wall);
        }
}
```

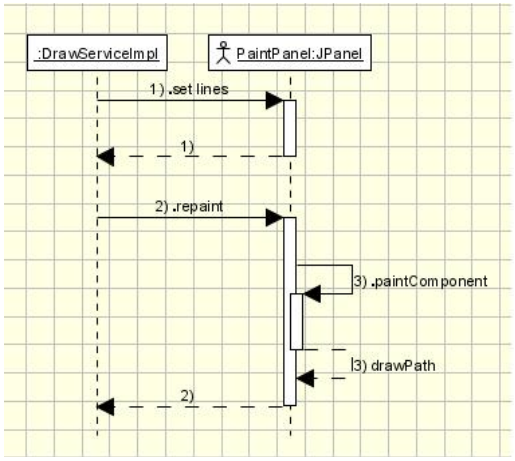



Fig.4.8: Push approach drawing UML sequence diagram

## *4.5 Scenarios*

The application provides the ability to group a set of moves as one entity – a scenario. A scenario may be saved to-, loaded from a file, and replayed. There is an independent entity in the code, *ScenarioFacade* that is responsible for dealing with scenarios and also serves for a mediator as an entry point to scenario logic.

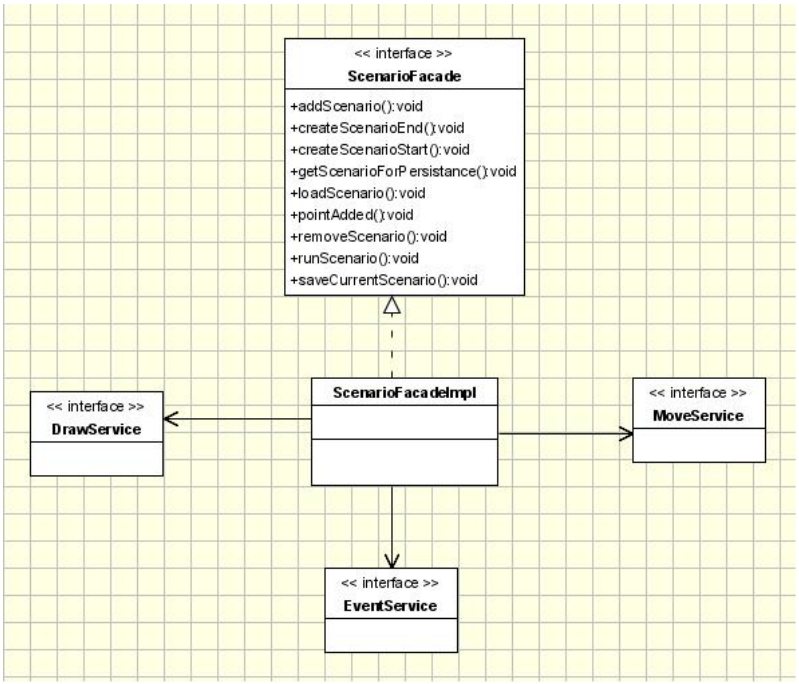



Fig.4.9: Scenario façade-service associations UML class diagram

The scenario façade uses Swing timer (explained in Threading Issues section) to implement scenario run simulation.

```
public void runScenario(final String scenarioName) {
        this.loadScenario(scenarioName);
        this.timer.start();
}

private class TimerListener implements ActionListener {
        public void actionPerformed(final ActionEvent e) {
                if (ScenarioFacadeImpl.this.moveIterator.hasNext()) {
                        final Point point = ScenarioFacadeImpl.this.moveIterator.next().getPoint2();
                        ScenarioFacadeImpl.this.doMove(point);
                } else {
                        ScenarioFacadeImpl.this.timer.stop();
                        this.timerStopped();
                        }
        }
}
```

There is a lot of logic in creating and replaying scenarios use-cases. *ScenarioFacade* uses all services to implement scenario use-cases.

```
private void doMove(final Point point) {
        this.drawService.drawNewPositionOccupied(point);
        this.moveService.setCurrentPosition(point);
        this.eventService.newPositionOccupied(point);
}
```

Scenarios may be created by an end user by means of the application, saved to external files and replayed somewhere in future. As scenarios are persisted in a user friendly XML format there may be created in any text editor and then loaded into the application. The application uses XML Beans library to parse scenarios from XML files:

```
private void processFile(final File selectedFile) {
        if (this.validateFile(selectedFile)) {
                Reader file;
                try {
                        file = new BufferedReader(new FileReader(selectedFile));
                        final XmlOptions options = new XmlOptions();
                        log.info("xmlbeans parsing the selected file");
                        final ScenarioDocument poDoc = ScenarioDocument.Factory.parse(file, options);
                        final Scenario scenario = poDoc.getScenario();
                        log.info("got scenario: " + scenario.toString());
                        final Position[] positions = scenario.getPositionArray();
                        final List<Point> points = ConversionUtils.convertToDomainObject(positions);
                                this.mediator.scenarioLoadedFromFile(points);
                } catch (final FileNotFoundException e) {
                        throw new RuntimeException(e);
                } catch (final XmlException e) {
                        throw new RuntimeException(e);
                } catch (final IOException e) {
                        throw new RuntimeException(e);
                }
        } else {
                throw new NotImplementedException();
        }
}
```

and to persist to XML files:

```
public void saveScenarioToFile() {
        final String scenarioName = (String) this.scenarioList.getSelectedValue();
        final List<dk.dtu.intelhouse.domain.Point> points = this.scenarioFacade.getScenarioForPersistance(scenarioName);
        try {
                final ScenarioDocument poDoc = ScenarioDocument.Factory.newInstance();
                final Scenario scenario = poDoc.addNewScenario();
                for (final dk.dtu.intelhouse.domain.Point point : points) {
                        final Position position = scenario.addNewPosition();
                        position.setX(new BigInteger(Integer.toString(point.getX().intValue())));
                        position.setY(new BigInteger(Integer.toString(point.getY().intValue())));
                }
                final String path = JOptionPane.showInputDialog(this.mainFrame, "Enter a file path to save a scenario");
                poDoc.save(new File(path));
        } catch (final IOException e) {
                throw new RuntimeException(e);
        }
}
```

## *4.6 Sensors*

### 4.6.1 Activation Framework

Sensors are the crucial part of the application. The requirement set on this part of the application was to support extensibility. To enable a plug-in approach to adding new kind of sensors and to force upon a stable architecture the following class hierarchy was constructed.

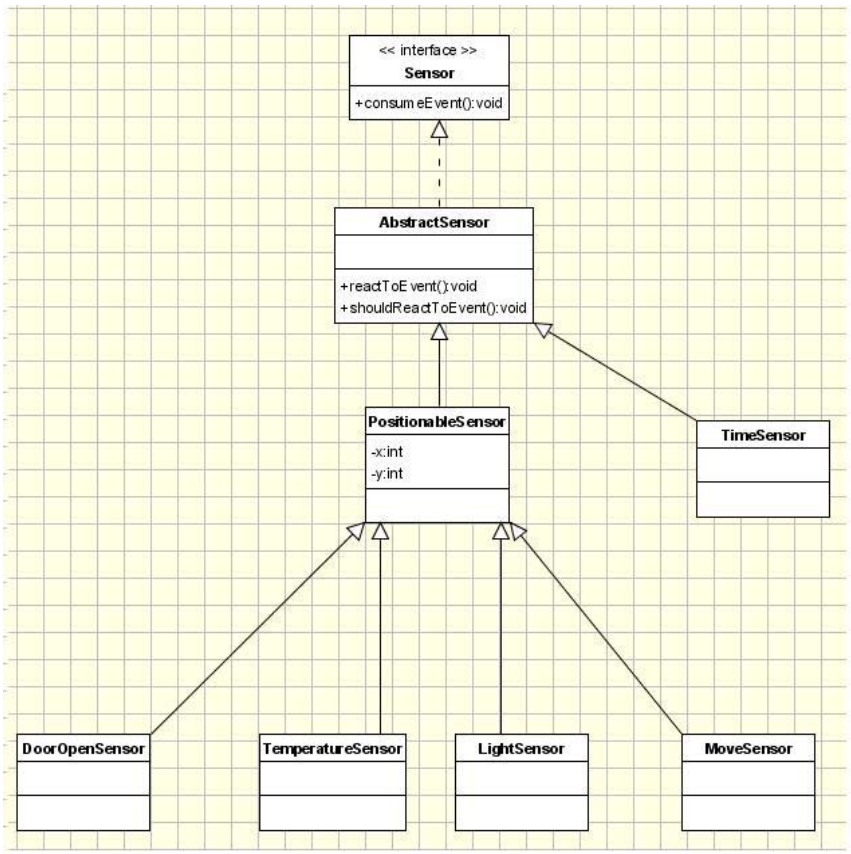



Fig.4.10: Sensor UML class diagram

A *Sensor* interface is a basic part of the hierarchy. It is the simplest one as well. It states the contract according to which all sensors must know how to react to a given event (all sensors have to define *consumeEvent* method). Although such a contract works smoothly with the rest of architecture, it gives a sensor developer a lot of freedom. To keep the architecture consistent and to provide a basic *Sensor* interface implementation there is an *AbstractSensor* abstract class that needs to be subclassed to add a new type of sensor quickly.

Therefore no one would practically want to implement the *Sensor* interface directly as there is already a convenient class to extend. This *AbstractSensor* class actually implements the sensor activation framework by providing:

- a template *consumeEvent(..)* method and connected to it: *basicShouldReactToEvent* and *isProperType* methods
- extensions points for sensor subclassing

The extension points include two methods:

- An abstract framework part – *reactToEvent* – that actually performs the sensor logic.
- A *shouldReactToEvent* method that is typically overridden in actual sensor implementations as it returns false by default.

The mechanism is based on a template method pattern [8]. Basically this means that a class that is higher in a hierarchy provides a template which is used by all classes that are lower in the hierarchy. When executing all classes lower in a hierarchy follows the pattern (a template method) set in a higher class.

There was introduced a distinction between an event that is dispatched to sensors and an action that may be triggered by sensors. An event is generated by *EventService*, which knows what actual event creates basing on other method invocations. This is the only responsibility of an *EventService*, as a generated event is then passed to an *EventDispatcher*. This is an *EventDispatcher* that passes events to sensors.

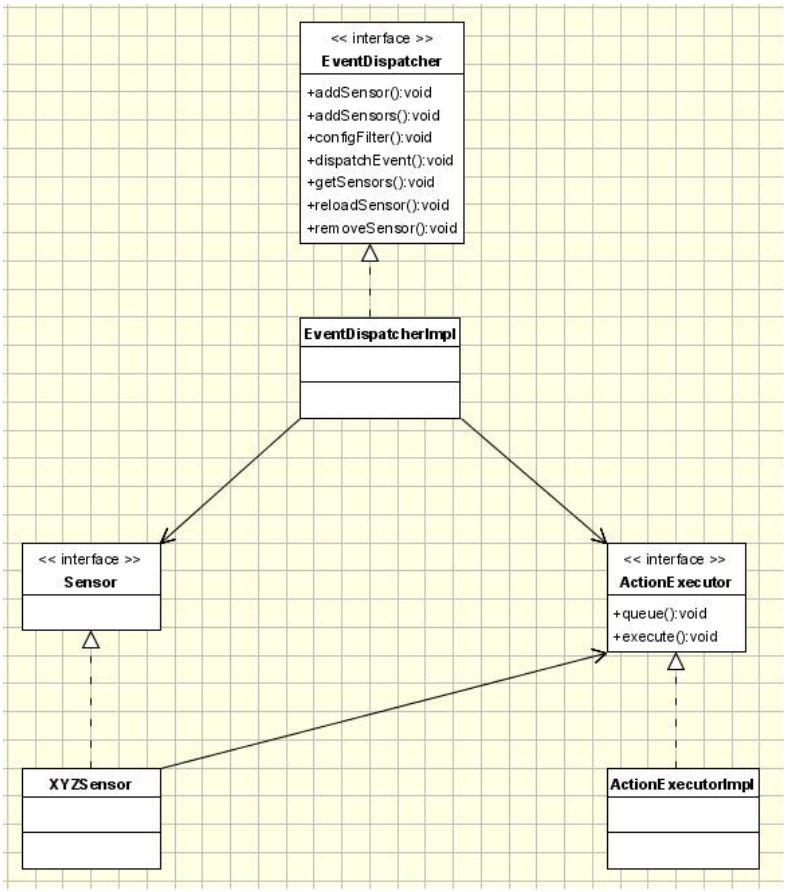



Fig.4.11: Event dispatching UML class diagram

Currently sensors are defined in XML files together with plan description.

### 4.6.2 Action Triggering and Execution

As one can see there are two methods in an action executor that may be used by a sensor to pass an action to execute:

- queue
- execute

A sensor may decide (or it may be extracted to a configuration file) that an action is to be executed immediately or queued to a later point when more actions are 'flushed'.

Currently *MoveSensor*s may react to any movement only in circular areas. However an extension to introduce an angle to slice a circular area can be easily added. The logic of finding whether such a sensor should trigger an action is implemented in a method overridden by *MoveSensor*:

```
protected boolean shouldReactToEvent(final Event event) {
        final MoveEvent evt = (MoveEvent) event;
        boolean shouldReact = false;
        final boolean pointInsideCircle = GeometryUtils.isPointInsideCircle(
evt.getX(),
evt.getY(),
this.radius,
this.x,
this.y);
        if (pointInsideCircle) {
                shouldReact = !this.moveService.isBehindWall(
new Point(this.x, this.y), new Point(evt.getX(), evt.getY()));
        }
return shouldReact;
```

As it can be observed the logic checks if an event took place inside an area and if that place is not hidden behind the wall. To introduce an angle, one would create an overloaded *isPointInsideCircle* method in *GeometryUtils* class:

```
public static boolean isPointInsideCircle(
final Double x, final Double y, final Double radius, final Double circleX, final Double circleY) {
        return Point2D.distance(x, y, circleX, circleY) < radius;
}
public static boolean isPointInsideCircle(
final Double x, final Double y, final Double radius, final Double circleX, final Double circleY, final Double angle) {
      // deal with an angle here
}
```

### 4.6.3 Sensors - Action Mapping

Sensor types should be independent from action types that follow sensor activation. From the application perspective it means that there should be an external, declarative mapping between a given sensor type and the action that it is going to trigger. Such a mapping may be done even at an instance level i.e. an activation of one sensor instance of Type A could result in a totally different action that the one triggered by another sensor instance of Type A. Currently the mapping is hard coded i.e. to change it one needs to recompile the code. However the application may be extended to read a mapping from an XML configuration file.

## *4.7 Logging Framework*

There is already a great logging framework developed by Apache Community [9]. It is used to log various events that occur throughout an application lifecycle. It is widely used as it is e.g. much easier to find a bug by looking through log files than by debugging. Log4J enables to declaratively configure how the events should be logged i.e.:

- where they should be stored
- what format of information (what thread, a date present, what method)
- what level of events should be stored (e.g. error, info, debug), as some levels may be omitted

There are plenty of log storage possibilities e.g. a file, a database, a console, JMS, Telnet, and many others. One may e.g. decide to use so-called SMTP appender to log specific events what would mean that information about those events would be sent by email, or an SMS appender to send some important log events directly to a mobile phone.

Very important feature is that it is possible to configure a storage type for certain events in a way that is transparent to an actual application code i.e. logging configuration can change at application deployment time without even touching an application code, because it is written in separate file **(%project_root%\src\log4j.properties)**.

In Log4J one may configure a logger basing on a java package. This capability was used to set a different level for log messages triggered by GUI elements. Most of events logged by widgets are connected to painting methods and it was not necessary to store all of them. Therefore there is a specific log level set for all classes in *dk.dtu.intelhouse.view* package:

```
log4j.logger.dk.dtu.intelhouse.view=ERROR, stdout, fileout, swing
```

As it can be observed, only events classified as ERRORs are logged.

At the beginning the application used only a console and a file as a mean to store events. Later, possibility to show events directly on a user interface (e.g. on a different tab) was added. As Log4J framework is extensible, it was enough to configure an *org.apache.log4j.WriterAppender* with my own *JTextAreaWriter*:

```
package dk.dtu.intelhouse.log;

import java.io.IOException;
import java.io.Writer;

import javax.swing.JTextArea;

public class JTextAreaWriter extends Writer {

private boolean closed = false;
        private JTextArea textArea;
        private StringBuffer buffer;

        public JTextAreaWriter(JTextArea textArea) {
                assert textArea != null;
                this.textArea = textArea;
        }

        // .. other methods

public void write(String string) throws IOException {
                checkIfClosed();
                getBuffer().append(string);
        }
public void flush() throws IOException {
                checkIfClosed();
                textArea.append(getBuffer().toString());
                textArea.setCaretPosition(textArea.getDocument().getLength());
                buffer = null;
        }
}
```

The integration of this new appender with the application was quite straightforward. It required only a modification of **log4j.properties** file (which contains a declarative logging configuration), actually adding:

```
# custom appender
log4j.appender.swing=org.apache.log4j.WriterAppender
log4j.appender.swing.layout=org.apache.log4j.PatternLayout
log4j.appender.swing.layout.conversionPattern=[%p,%c{1},%t] %m%n
```

During the application initialization *WriterAppender* was configured:

```
final Logger logger = Logger.getRootLogger();
WriterAppender appender = (WriterAppender) logger.getAppender("swing");
        appender.setWriter(new JTextAreaWriter(textArea));
```

, where *textArea* is a log textbox on the *Run* tab.

To better utilize logging capabilities, it is considered a good practice to always override *Object.toString()* method. A default (Object class) implementation returns a class name of an object followed by a memory address that it occupies. As an in-house implementation of *toString* methods may be tedious (especially when there are many fields declared in a class and, hence, much information should be output), it is suggested to use a well-known *commons-lang* library from Jakarta Apache which provides, among many others, *ToStringBuilder* utility class that uses reflection to implement a nice *toString* method. This is how it was used to build *toString* methods for e.g. *Point, Segment* classes:

```
public String toString() {
      return ToStringBuilder.reflectionToString( this, toStringStyle.SIMPLE_STYLE);
}
```

## *4.8 Flat and Scenario Definition*

XML standard is used to store a flat definition. An XML schema (XSD) was designed to support configuration of:

- doors
- wall
- doors
- windows
- sensors

Below part of the schema is presented (a scenario part is omitted)

```
<xs:schema id="flat" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="flat">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="22">
        <xs:element name="walls">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="wall" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="doors" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="door" minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                              <xs:attribute name="x1" type="xs:integer" />
                              <xs:attribute name="y1" type="xs:integer" />
                              <xs:attribute name="x2" type="xs:integer" />
                              <xs:attribute name="y2" type="xs:integer" />
                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="windows" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="window" minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                              <xs:attribute name="x1" type="xs:integer" />
                              <xs:attribute name="y1" type="xs:integer" />
                              <xs:attribute name="x2" type="xs:integer" />
                              <xs:attribute name="y2" type="xs:integer" />
                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="x1" type="xs:integer" />
                  <xs:attribute name="y1" type="xs:integer" />
                  <xs:attribute name="x2" type="xs:integer" />
                  <xs:attribute name="y2" type="xs:integer" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XMLBeans [10] XML processing library from Jakarta (actually donated by BEA.com) was used. The library has an interesting and a programmer friendly approach to processing XML files. It has its own 'compiler' which parses an XML schema and generates Java classes.

Basing on the supplied schema XML Beans tool created the classes of the types mentioned above. The XML Beans tool provides a command line utility to compile schemas. The tool was invoked by means of a batch script that is attached below.

```
C:\dev\xmlbeans-2.2.0\bin\scomp -out lib/model.jar -debug -verbose flat.xsd scenario.xsd
```

The tool creates java classes packed in a jar file titled **model.jar**. There are two types of classes generated. These Java classes contain:

- classes that represents an application domain (like walls, doors etc in case of IntelHouse e.g. a *Flat* class used in the code shown below)
- classes that are factories used at runtime to parse an actual XML configuration file(e.g. *FlatDocument.Factory* class used in the code shown below)

```
private void processFile(final File selectedFile) {
        if (this.validateFile(selectedFile)) {
                Reader file;
                try {
                        file = new BufferedReader(new FileReader(selectedFile));
                        final XmlOptions options = new XmlOptions();
                        log.info("xmlbeans parsing the selected file");
                        final FlatDocument poDoc = FlatDocument.Factory.parse(file, options);
                        final Flat flat = poDoc.getFlat();
                        log.info("got flat: " + flat.toString());
                        this.mediator.flatLoaded(flat);
                } catch (final FileNotFoundException e) {
                        throw new RuntimeException(e);
                } catch (final XmlException e) {
                        throw new RuntimeException(e);
                } catch (final IOException e) {
                        throw new RuntimeException(e);
                }

        }
}
```

The jar file with both types of classes is imported into the application project under Eclipse IDE and is used as any other third-party jar.

Thanks to having a concrete factory classes to parse XML configuration files, a programmer deals with a concrete API i.e. these factory classes return in our case concrete domain objects like walls, doors etc. that are further processed by a business logic without a need for additional class casting as it is shown in the code listed above.

Because of such approach it was quite easy to plug in a module with drawing capabilities (*Compose* tab). This is because there was a stated contract (in a form of XML schema) to define a flat. To persist a composed apartment plan, an object of Flat class created by XML Beans tool (basing on a provided schema) needs to be instantiated. Then it is enough to employ a factory created by XML Beans tool as well to persist it as listed below:

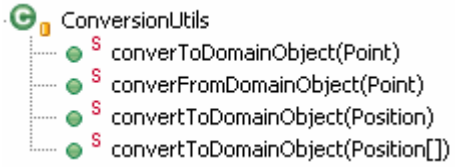
```
private void saveActionPerformed() {
        final int returnVal = this.fc.showSaveDialog(this);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
                final File file = this.fc.getSelectedFile();
                try {
                        final FlatDocument flatDoc = FlatDocument.Factory.newInstance();
                        this.saveFlatPlan(flatDoc, file);
                        ComposePanel.logger.info(flatDoc.getFlat());
                } catch (final Exception e) {
                        ComposePanel.logger.error(e);
                }
        }
}

private void saveFlatPlan(final FlatDocument flatDoc, final File file) {
        final String xml = this.convertToXML(flatDoc);
        if (xml == null) {
                JOptionPane.showMessageDialog(this, "The plan doesn't have walls.");
                return;
        }
        Writer pw = null;
        try {
                pw = new PrintWriter(file);
                pw.write(xml);
        } catch (final FileNotFoundException e) {
                ComposePanel.logger.error(e.getMessage(), e);
        } catch (final IOException e) {
                ComposePanel.logger.error(e.getMessage(), e);
        } finally {
                if (pw != null) {
                        try {
                                pw.close();
                        } catch (final IOException e) {
                                ComposePanel.logger.warn(e.getMessage(), e);
                        }
                }
        }
}
```

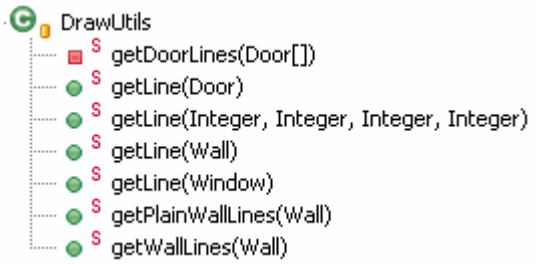
## *4.9 Utility Methods*

During the development some stateless logic was extracted from various classes. This resulted in an additional, U*til*, package with static utility methods only. Service layer uses this package to do e.g. geometry computation, *Swing* to domain objects conversion etc. To be compliant with general Java naming conventions, the classes end with *Utils* suffix

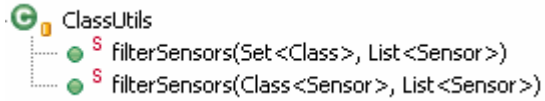
- *ConversionUtils* – used to convert Swing objects to the application domain objects and vice versa e.g. *java.awt.Point* to *dk.dtu.intelhouse.domain.Point*. This class makes both-way conversion between domain objects (the ones from *dk.dtu.intelhouse.domain* package) and user interface objects (e.g. ones from *java.awt* package). Methods *convertFrom* and *convertTo* follow naming convention depending on the way of the conversion, for example:
    - *convertFromDomainObject* – takes a domain object as a parameter and returns a non-domain object,
    - *convertToDomainObject* – takes a non-domain object as a parameter and returns a domain object.



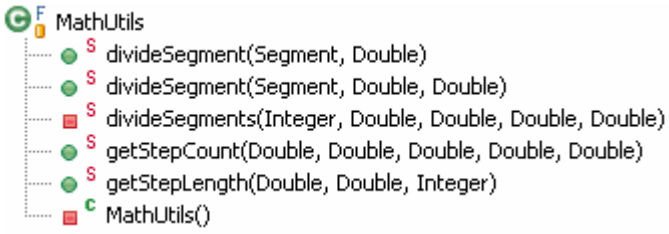

- *DrawUtils* – used to create ready to display objects from domain objects i.e. accepts a domain object as a parameter and returns a set of *java.awt.Line2D* objects that are further displayed by a *PaintPanel*. The application business logic operates on domain objects classes generated by XML Beans tool basing on the application XML schema. However as far as the user interface is concerned there is no distinction between a window, door or wall concepts; there are only lines, circles, colours etc. That is how the application widgets responsible for drawing (*PaintPanel.java*) make an actual drawing. Therefore there must be a place that does the actual conversion between logical concepts (a wall, door, and window) and pure drawable elements (a line, circle). The code that implements this conversion is placed in DrawUtils.java.



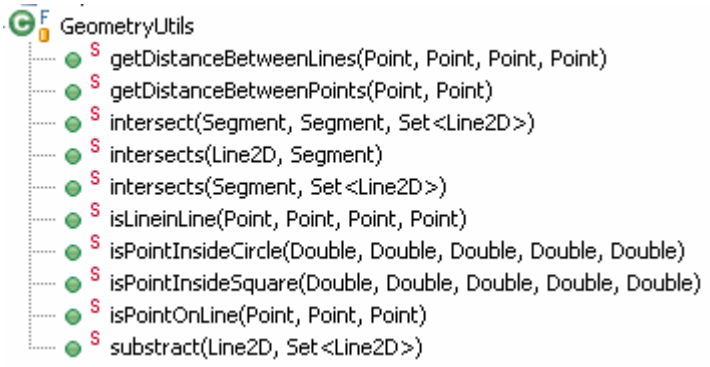

- *ClassUtils* – used to filter sensors basing on criteria passed as a parameter. The criteria are represented by the first parameter which means the class of a Sensor that should pass filtering. The method is helpful when we have a set of sensor objects and we want to filter it to get a set of filters only of the desired type (e.g. only *MoveSensors*) or set of classes (e.g. only *MoveSensor* and *TemperatureSensor*). This may be useful if we have many sensors placed in an apartment and we want to take into account only a subset of them in a given scenario.



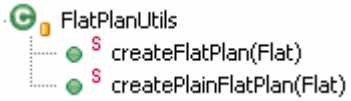
- *MathUtils* – used to do computational tasks to find move steps. The origin of this class comes from the fact that in no simulation we can present infinitely small steps. All computer simulations are based on some concrete steps of possibly very small value but greater than zero. In the simplest scenario the application presents a person moving straight from one point to the second point. However to simulate a move we need to divide this one big step into smaller ones and present the move smoothly. Still we need to find the lengths of those smaller steps and the class is used to compute the length of every single small step. We call a big step a segment.




- *GeometryUtils* – used to do computational tasks about geometry. The methods there are self-documenting and this was the preferred way during coding. *Subtract* method is the most complicated. It takes a line parameter from which it subtracts all lines from a second parameter. As a result we get a set of lines that when added to a second parameter would give exactly a first parameter.




- *FlatPlanUtils* – used to convert flat definition XMLBeans objects to application domain objects. The class is used to convert a *Flat* object (class generated by XML Beans tool based on the application XML schema) to a *FlatPlan* which is a domain object. There are two methods which differ by whether walls that go to a *FlatPlan* object are to be divided by doors they contain or not.



## *4.10 Threading Issues*

While developing the application we came across threading issues. Java Swing is based on its own threading model. According to the contract a developer should not create their own thread when using GUI based on Swing. However, Swing provides its own means to introduce application specific threads. It introduces an application specific thread that is to interact with GUI and a developer should 'plug-in' into the application using the specific Swing extension points or utility classes (we use *javax.swing.Timer* class to schedule periodical screen refresh when an end-user changes the position in the application GUI which is covered in details in [11]).

We introduced the application specific thread (*Timer*) to simulate a person moving in an apartment. A separate thread is started when end-user triggers an inhabitant movement. This thread periodically fires an event that is caught by a mediator.

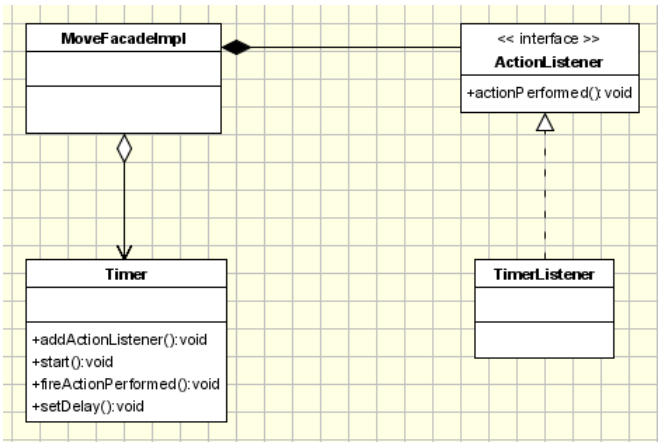



Fig.4.12: Timer thread - observer design pattern UML class diagram

The communication is based on a well-known Observer pattern [12]. There is a private class *TimerListener* defined in *MoveFacadeImpl*. *MoveFacadeImpl* creates and configures a timer and a listener object and sets a listener on a timer.

```
public MoveFacadeImpl(...) {
                // ...
                this.timerListener = new TimerListener();
                this.timer = new Timer(10, this.timerListener);
                this.timer.setInitialDelay(300);
}

private class TimerListener implements ActionListener {
        public void actionPerformed(final ActionEvent e) {
                if (moveIterator.hasNext()) {
                        final Point point = moveIterator.next().getPoint2();
                        doMove(point);
                } else {
                        timer.stop();
                }
        }
}
```

As it can be seen above, timer fires events until move iterator (which has the consecutive simulation positions) has a next position to move a person. When a person reached a destination point (i.e. an iterator does not have more positions to return), a timer is explicitly stopped.

## *4.11 JUnit Tests*

Continuous design and refactoring approach was preferred during work on the project. However, no refactoring should be done without a proper set of tests; otherwise one may easily introduce bugs into the existing code.

A few JUnit tests for utility classes were written. JUnit [13] is a de facto standard test framework in the Java world.

The tests are mainly implemented for utility methods which were considered to be the most error prone in the whole application. Moreover without a stable utility methods layer no service could return reliable results. Below a sample JUnit test is presented.

```
public class ClassUtilsTest {
        @Test
        public void testFilterSensors() {
                List<Sensor> sensors = new ArrayList<Sensor>();
                Set<Class> classes = new HashSet<Class>();
                List<Sensor> filteredSensors1 = ClassUtils.filterSensors(classes, sensors);
                Assert.assertEquals("filtered list size", 0, filteredSensors1.size());
                sensors.add(new MoveSensor());
                sensors.add(new MoveSensor());
                List<Sensor> filteredSensors2 = ClassUtils.filterSensors(classes, sensors);
                Assert.assertEquals("filtered list size", 0, filteredSensors2.size());
                classes.add(MoveSensor.class);
                List<Sensor> filteredSensors3 = ClassUtils.filterSensors(classes, sensors);
                Assert.assertEquals("filtered list size", 2, filteredSensors3.size());
        }
}
```

The test consists of:

- assertions that are put at the beginning of the tests (the input)
- actual logic (the code under test) invocation
- assertions that make sure that a result is what we expected

Another important test was about dividing a whole step into smaller parts which are seen as move steps by an end-user. A part of the test is presented below.

```
public class MathUtilsTest {

        private static Logger log = Logger.getLogger(MathUtilsTest.class);

        @Test
        public void testDivideSegment() {

                final int expectedSize1 = 11;
                Segment segment1 = new Segment(new Point(0.0, 0.0), new Point(-230.0, 0.0));
                testSegment(segment1, expectedSize1, 20.0);

                Segment segment2 = new Segment(new Point(230.0, 0.0), new Point(-229.0, 0.0));
                final int expectedSize2 = 22;
                testSegment(segment2, expectedSize2, 20.0);
        }

        private void testSegment(Segment segment, final int expectedSize, double step) {
                List<Segment> segments = MathUtils.divideSegment(segment, step);
                Assert.assertEquals("segment list length", expectedSize, segments.size());
                Assert.assertEquals("last point check", segment.getPoint2(),
segments.get(segments.size() - 1).getPoint2());
        }
}
```

## *4.12 Suggestions for Future Development*

Obviously software needs further development, as it represents an early stage of software that is supposed to be a true simulation tool described in chapter 2.4.

Some other possible extensions improving software are listed below as suggestion.

Logging

One may think of an enhancement in a form of Log4j configured at runtime by an application end user. An end user could have a possibility to define what events to log by means of a user-friendly GUI interface (i.e. without having to manually edit the Log4j settings file). An end-user should also have a mean to choose where logs should be stored (Log4J even enables sending the events directly by email or putting them into database).

Advanced Enhancements

Another very interesting way, that we especially eagerly would like to see implemented, would be to be able to set a logical values to entities that are inside an apartment. Currently only walls and doors present any meaning to the application. However putting electrical facilities (like ovens, lighting switches, plugs, fridges) or others like water taps as first-class objects would gives many possibilities to do an interesting analysis. One could e.g. modify facilities positions in an apartment plan and replay the same scenarios for various facilities positions. This could lead us to a most efficient facilities positioning. By most efficient we mean the positioning for which the total person move for a given scenario is shortest.

# Chapter 5

# Program Testing and Evaluation

First part of this chapter is devoted to testing of the proposed solution, to ensure proper operation of the written code. In second part we try to evaluate the proposed solution, mainly considering ease of use and user impression.

## *5.1 Testing*

Several tests were conducted to verify the proper implementation.

### 5.1.1 Simple Movement Test

This first test was conducted to validate proper operation of one sensor in a room.

Room was set to have dimensions of 300x300; single movement sensor with detection radius 50 was placed in the center of the room (150,150).

Test is composed of two parts:

- Movement of an inhabitant along the line with X coordinate 150, simple calculations give us that sensor should detect inhabitant entering detection area in point with coordinates (150,100), and leaving area in point (150,200).
- Movement of an inhabitant along the line with Y coordinate 150, simple calculations give us that sensor should detect inhabitant entering detection area in point with coordinates (100,150), and leaving area in point (200,150).

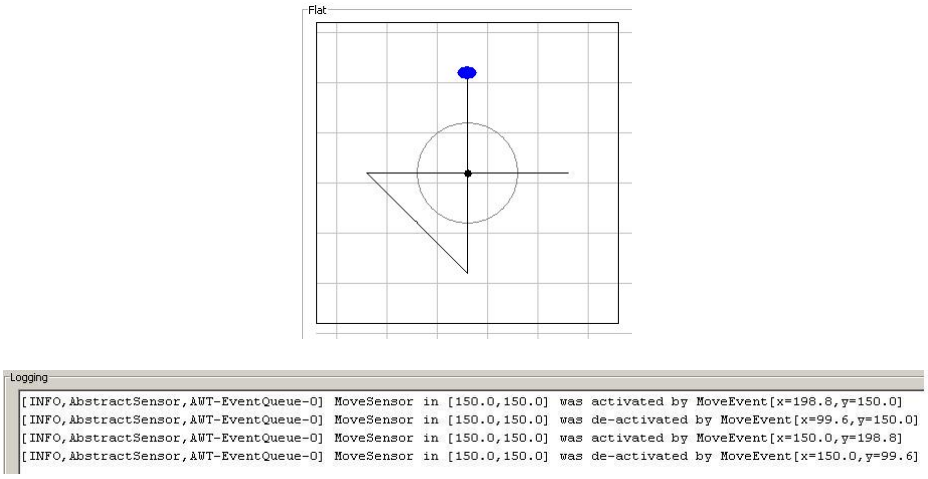



Fig.5.1: Test 1 Scenario and log

### 5.1.2 Wall Test

This test was conducted to make sure sensors don't detect moving objects that are hidden behind the wall.

Room with the same dimensions is used (300x300). Sensor with a detection radius of 100 is placed in the center of the room (150,150). Additionally there is a wall in the room built between points (200, 0) and (200,150), which intersects with a sensors' field-of-view. In the first stage of movement inhabitant moves behind the wall to check that sensor does not detect him. Afterwards, move takes place on the other side of the wall (not hidden from the sensor) to check that sensor operates properly.

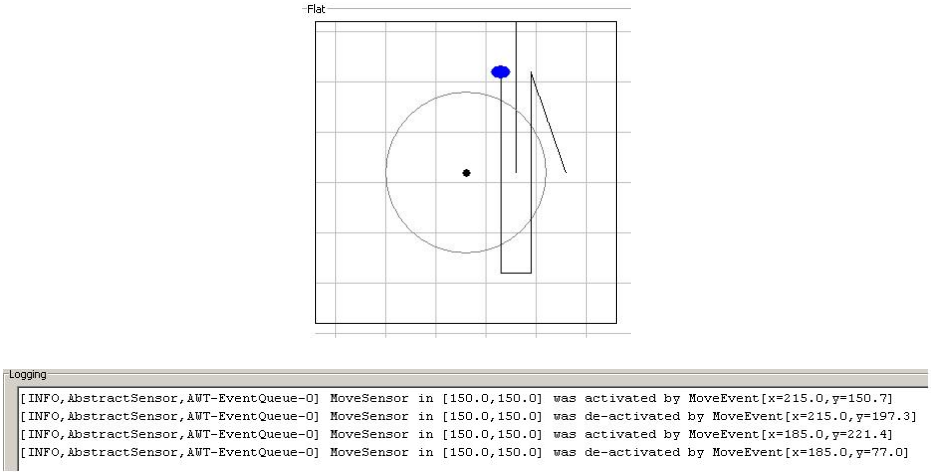



Fig.5.2: Test 2 Scenario and log

### 5.1.3 Door Test

This test was conducted to verify that movement sensors do detect objects through door opening

Test setup very similar to the one in test 2 is used; the only major difference is that a door is placed in an 'additional' wall. Also movement of inhabitant is designed in similar way as in previous case; first movement takes place in the area behind wall to observe the activating/deactivating sequence when inhabitant is visible to a sensor through the door opening. Afterwards inhabitant moves in the part of the room not covered by wall to ensure sensor is working properly.

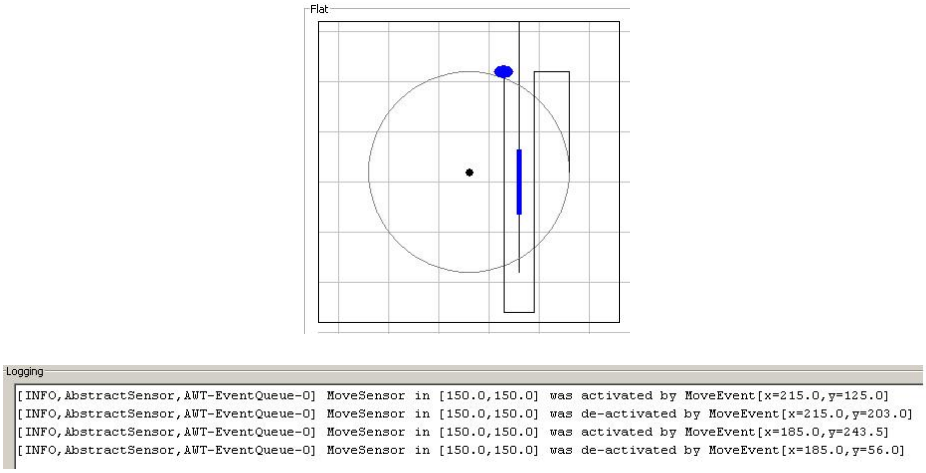



Fig.5.3: Test 3 Scenario and log

### 5.1.4 Multiple Sensors Test

This test was conducted to verify that inhabitants' movement can activate/deactivate more than one movement sensor.

Room of the size 400x300 and two sensors with detection radius of 80 placed at points (150,150) and (250,150) were used in the setup. Sensors' fields-of-view intersect, creating an area where movement will be detected by both sensors.

Test was composed of two parts:

- Movement of inhabitant along the line with Y coordinate 150, to observe that sensors first activate and later deactivate, as the inhabitant moves.
- Movement of inhabitant along the line with X coordinate 200, to observe simultaneous activation (and later deactivation) of sensors as inhabitant enters the point of intersection of their fields-of-view.

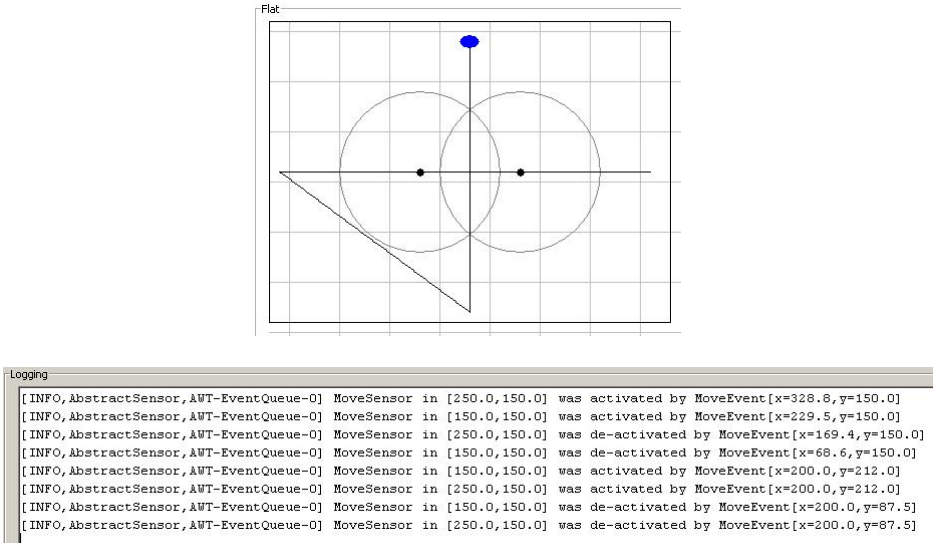



Fig.5.4: Test 4 Scenario and log

## *5.2 User Impression Evaluation*

Program was thoroughly tested in order to validate ease of use, which is of the most importance as the program is intended to be used by broader audience. Most tedious phase of program operation is a floor plan creation, which for simple house with 5 rooms and with a use of background image help takes on average about two minutes. Obviously more time is needed for more complex floor plans or in a case when user does not have image of an apartment, which could be used as a background for drawing. After apartment plan is created it may be saved and reused at any time, therefore there is no need to repeat this time consuming procedure every time program is run, which was appreciated by test users.

# Chapter 5

# Conclusions

With currently growing interest in smart houses technology, there is increasing number of people that would like to have software that will help them to have it implemented in their own homes. A couple of software packages are available on the market, but their price is usually way too high for a user that just wants to try it out – and that demand was one of the reasons for establishing this project.

Coming up to these expectations, during work on this project, current research in the field of smart homes was briefly presented. A set of requirements for a smart environment simulation tool was sketched which lead to a design and implementation of a Smart House Simulation Tool, allowing broad audience to experiment with smart house technology. It presents an easy-to-use and simple solution to people interested in the field.

It enables a user to draw a simple 2D apartment plan and place sensors of his choice in it. A movement of an inhabitant may be programmed, and sensors' operation triggered by inhabitants' behaviour may be observed and analysed, leading to development of the best solution suited for a particular individual.

During the development phase, a stress was put on a clean design. Much attention was held to keep the software readable and maintainable. This resulted in a form of self-documented code with possibly short and consistent classes and methods that have meaningful names. The architecture was based on well-known design patterns. As far as GUI layer is concerned, a popular *Mediator* entity serves as a single dispatching point for widget-generated requests, *Façade* acts as a common gateway to the application actual logic represented by a *Service* layer while some of error processing logic was extracted and delegated to *Helper* utility classes which are tested with additional JUnit code.

The usage of the broadly documented approaches to building GUI applications helped to omit many problems that would probably appear if we tried reinventing-the-wheel approach instead. Moreover, design patterns give a common nomenclature which makes a code readable. This means that any Java programmer that would like to further develop the code base should have no problem with understanding the application internals. As the *programming to interfaces* approach was chosen there are clean abstraction layers between functional parts of the code. Therefore, modifications of one layer internals have no influence on any other layer.

All the code is compliant with object-oriented programming art with all its powerful features. No magic constants are used, no publicly accessible class data, no overgrown *if-*

*else* blocks. Following the application execution flow is possible which is crucial if software is to be truly maintainable and extensible.

The tool design gives possibility of further extension, with the most recommended of them being implementation of a broader set of sensor variety, enabling the house to collect data from more events than just movement, introducing actuators (that will control devices in the house) and a programmable logic governing their operation. That will make this software be a truly programmable house simulator. As an advanced project, extension of this software giving it some form of Artificial Intelligence might be implemented.

# References

[1] http://www.konnex.org/knx-tools/ets/price-order/

[2] http://poradnik.smartech.pl/

[3] http://en.wikipedia.org/wiki/Data_mining

[4] Antony Galton ”Causal Reasoning for Alert Generation in Smart Homes” Designing Smart Homes, LNAI 4008, pp. 57–70, 2006.

[5] Rezaul Begg and Rafiul Hassan ”Artificial Neural Networks in Smart Homes” Designing Smart Homes, LNAI 4008, pp. 146–164, 2006.

[6] http://en.wikipedia.org/wiki/Mediator_pattern

[7] http://java.sun.com/docs/books/tutorial/uiswing/events/generalrules.html#innerClasses

[8] http://en.wikipedia.org/wiki/Template_method_pattern

[9] http://logging.apache.org/log4j/docs/

[10] http://xmlbeans.apache.org/

[11] http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html

[12] http://en.wikipedia.org/wiki/Observer_pattern

[13] http://junit.org/index.htm

# Appendix A: Usage guide

Program is available in two versions on an attached CD. One is a source code, which is placed in file **source code.zip**. Another one is executable version, placed in file **intelhouse.zip.**

Log file will be written to **d:\dev\log.txt**. If this location is unavailable for any reason (like lack of necessary file access permission) one will have to manually edit file **%project_root%\ log4j.properties** and change the following line in the file's last section: **log4j.appender.fileout.File= d:\dev\log.txt**, to any desired available path.

It is necessary to have a Java SE Development Kit (JDK) version 6 beta 2 installed in the system. It may be obtained from http://java.sun.com/javase/downloads/ea.jsp.

To run the program, simply execute **start.bat**.

To load a flat and try to walk in it:

1. **File->Open Flat**
2. Some predefined flat outlines are stored in **schema** directory e.g. select **sample.xml**. Flat outline will be loaded.
3. Move the person around the house to observe operation of sensors (either in a Log Events section or in a *Log* tab)

To load a scenario:

1. First load a flat (see above)
2. Load a scenario, either from **File->Load Scenario** or by using a button in section Scenario named **Load From File**
3. Some predefined scenarios are saved in **schema** directory, e.g. use **sampleScenario.xml**
4. Name the scenario as you wish, it will appear in the list of scenarios.
5. Select scenario from the list, then press **Run** button to run the scenario.

To design your own scenario:

1. Load a flat
2. Press **Create** button and design the inhabitants' movement
3. Save the scenario, either to a file or temporarily in a program
4. Select scenario from the list, then press **Run** button to run the scenario.