Implementation

- Description of the system components
- The implementation section should allow a skilled programmer to maintain the software
- **Remember, the most important documentation of the implementation is comments in the code!**

The physical setup

Since we needed real life data to train the system, the first task of the project was to create a physical system to start collecting data. We installed wireless switches and PIR sensors[1] in a David's appartment. The placebo switches were placed next to the normal switches controlling the light for each room, in all cases being the switch closest to the entrance. We installed a total of 10 motion sensors and 5 switches throughout the apartment, that collected data non stop for a period of two weeks.

The sensor setup consisted of three sensors in the living room, two sensors in the hallway, kitchen and bedroom, and one in the bathroom. When placing the sensors, we tried to provide as close to full coverage as possible, with special emphasis on making sure all the doorways were covered.

![Map of the testing environment with sensor and switch locations]
[Hellebaekgade]

The wireless nodes we have available communicate using the Zensys Z-Wave protocol. This protocol was chosen because we already prior to this project had designed and implemented a z-wave API in java. This greatly reduced the time and effort needed to setup and implement the data collection system.

ERHVERVSJURA

We setup a mini PC with a Z-Wave serial device, and configured all PIR sensor and switches to send notifications to the PC, when they where activated. The PC ran a Z-Wave API, which we added a listener to, so that sensor and switch event was logged to a SQL database.

Database table for sensor events

sensor_events	
id	Integer
timestamp	Timestamp

Database table for switch events

switch_events	
id	Integer
timestamp	Timestamp
status	Boolean

Simulator / AI interface

We have a smart house simulator available, which will be extended with an AI module, implementing the features discussed in this report. The simulator is implemented in scala, so an obvious choise would be to implement the AI in scala aswell. However work with the simulator in the initial stages of the project, showed that our programming speed in scala was too slow to get any meaningful amount of work done. The scala language is build upon Java, and both languages compiles to bytecode in *.class* files. A result of that is that Scala and Java interface very easily, and Scala code can invoke Java methods and vice versa. We chose to implement the AI in Java, working in a language we're well-versed in, to increase our productivity and quality of the code.

Configuration

Decision Matrix

Antallet af gange den klasse har skiftet navn… I've lost count… - svaret er 4

Event patterns

To make lookups based on the lastest event pattern, each new sensor event needs to be a matched to see if it's part of a pattern. As each sensor and switch event is received by the system, a list of the most recent event pattern is maintained in an EventList.

EventList determines if the lastest event is part of the pattern, and determines if a zone event has occured (if set to use zone events).

The invariant of the EventList, is that after an event is added, the event list contains the current event patttern. This pattern can then be used to determine if any switches should be turned on or off.

Correlation table

Correlation statistical generation
Correlation calculates the probability that a sensor is correlated a switch. It scans the database, and looks at the interval just after a switch is triggered. The sensors that triggered in the interval, are counted for that interval, in a way thay they're only counted once per switch event. If a multiple switch event are triggered in the same interval, the sensor events in the overlapping intervals should be counted for each of those switches. Having the number of times each switch is triggered, and each sensors triggeres with the given time interval, it's then calculated the probability that \(sensor_i \) is triggered, given that a \(switch_j \) was turned on atmost \(\Delta t \) ago. This gives the statistical correlation probability table.

To this the correlation confirmations in the database, is then added. Each row in the database table contains the acculated correlation correction for that switch / sensor pair. The correlation correction is simply added to the correlation based on the statistical data.

The resulting correlation table is allowed to have probabilities above 100%, which is inteded as destribed in

Correlation correction
When a switch is turned on, a timer is started for that switch. If a correlated sensor is triggered, it timer is extended. The duration is determined by the correlation between the sensor and the switch, higher correlation gives longer

timeouts. If the switch is turned off, the timer is stopped. If the timer runsout a timeoutevent is triggered, and the light is turned off, and a new timer is started, to verify that no manual overrides occur. If the a manual override occurs (e.g. the user turns the switch on again, while the timer is running), the system is "punished". The system increases the timeout time, by increasing the correlation between the switch and the first sensor triggered after the switch was turned off. If no manual override occurs, the system was correct in turning off the light, and lowers the timeout time, by reducing the correlation between switch and the last seen sensor before the switch was turned off.

These correlation corrections are stored in a separate table in the database. The correlation use for the timeout is based on both the statistical correlation, and these correlation corrections. The correlation corrections increase or reduce the correlation by 10 percent points. The system allows correlations higher than 100%, this gives the intended behavior that a switch may have a longer timeout than what is default.

Database table for correlation corrections

correlation_confirmation	
switch	Integer
sensor	Integer
correlation	Float

Timers and timeout

Timers are implemented in the Timer and Sleeper class. Sleeper is a fairly simple class, it sleeps starts a new thread, sleeps for a given time, then fires a timeout event to a given timeout listener. Timer simply holds a map, where each switch can set a timeout. Timer creates a sleeper object, and puts in the map. The sleepers can then easily be monitored and interrupted if needed.

To received the timeout events the SmartHouse class implements TimeoutListener.

1. Passive infrared sensors.