Université Catholique de Louvain

LINGI2252

Software Maintenance and Evolution

Mission 2: Improved prototype

Authors:
Group G
Gustin Simon
1171-14-00
Hallet Adrien
3276-13-00

Professor:
Mens Kim
Assistant:
Duhoux Benoît

November 15, 2018

1 Introduction

For the course *LINGI2252 – Software Maintenance and Evolution*, we were asked to improve our first prototype of a house automation system. Specifically, we had to add a parametrization component which would allow us to specify the values and states of the variability points at execution time, a command line interpreter which would allow us to update the state of the house at run time and to use relevant pattern designs to make the code more maintainable. We will discuss each of these extensions.

2 Parametrization component

The first extension we discuss is the parametrization component. It allows us to easily change the configuration of a house on two different executions without changing the code. It does this in a quite straightforward way: when the program executes, it starts by parsing a configuration file. It then creates a house that corresponds to the configuration described in this file.

Notice that we didn't decide to implement the parser as a class of its own but rather to place the parsing logic inside the class `House`. As we use an external library to parse the configuration file, we figured out it wouldn't be very interesting to make a parser that would be only a couple of lines long. As can be expected the configuration of the house is then done from within this class itself on basis of the parsed representation of the file. However, if we needed to parse files in other formats, making a class to handle the parsing would be a very good idea.

The configuration is indeed contained within an external file formatted in *JSON*. We chose this format for several reasons.

First it is well-known and heavily used as a format for configuration files. This is due to the fact that it is perfectly capable of handling this kind of task. If we had decided to make our own "configuration language", we could have very easily ended up with a syntax that would not fit all possible configurations perfectly, or with a flawed parser that would not work in certain cases.

Secondly, the fact that it is a well-known format also means that parsers exist in most languages. We didn't have to implement a parser by ourselves or to use one that would be incomplete or buggy.

Lastly, we are used to this format, which means we can work quickly and efficiently with it. The likelihood of us making an error while using it was thus smaller. This being said, using *YAML* or *TOML* would be perfectly legitimate choices, but we weren't as used to them as we are of *JSON*. Note that using the latter instead of one of those two formats prevents us from having comments in our configuration files, which can be considered a problem.

3 Command line interpreter

We discuss now the second extension we implemented, i.e. the command line interpreter. As we already said, this extension can be used to update the state of the house by typing commands during the execution of the program.

Each command is quite simple to interpret with the first word corresponding to the action to execute and the following words being the (optional) parameter(s) of the action. We designed them this way in order to simplify the implementation of the interpreter.

The implementation is made in a class named `Scenario`. Commands can be received in two different ways: either sequentially from the standard input, or from an input file which contains one command per line. Interpreting a command calls a method that executes the corresponding action to change the state of the house. This makes our code reusable: if we change our interpreter, we can call the same methods when we need to execute the corresponding actions.

4 Variability Points

4.1 House Layout

The main variability point is the ability to adapt each possible house. Our application can modify the house layout via the parametrization component. Thanks to the `HousePart` component, we can split the `House` in as many subcomponents as we want (either one per floor, one per room, one per square meter, whatever fits the needs). The implementation is pretty straightforward as the `House` is nothing but a dynamic list of `HousePart`.

```
public class House{
  private static House instance;
  private static String filename;

  JSONObject config;
  ArrayList<HousePart> housePartList;
  ...
}
```

This list is produced from the configuration file which lists the rooms and their components, as you can see in the short following example.

```
{
"house": {
    "houseParts":[
        {
            "name": "Entrance",
            "accessible-houseParts": ["Kitchen"],
            ...
        }
        ]
      }
    }
```

4.2 Controllers

In the configuration file, you can also modify the controllers (sensors, actuators and connected objects). You can place them and link a sensor to one or multiple actuators and/or connected objects. With such an implementation, you can link any kind of supported event (*e.g.: detecting motion, smoke, humidity, ...*) to any kind of supported action (*e.g.: opening a door, turning a*

light on, ...). As for the house layout, this has been done with a simple dynamic list of `Sensor` linking to a dynamic list of `Actuator` in a `HousePart`.

```
HousePart(House parent, JSONObject housePart){
  ...
  if (housePart.has("actuators"))
      this.actuators = parseActuators(housePart.getJSONArray("actuators"));
  else
      this.actuators = new Actuator[0];
  if (housePart.has("objects"))
      this.connectedObjects = parseConnectedObjects(housePart.getJSONArray("objects"));
  else
      this.connectedObjects = new ConnectedObject[0];
  if (housePart.has("sensors"))
      this.sensorsJSON = housePart.getJSONArray("sensors");
  else
      this.sensors = new Sensor[0];
  ...
```

Once again, the controllers' variability is handled by the JSON configuration file. Each HousePart can only have one sensor of each type. But that does not reduce the system capacities. If you want to have two different motion sensors in the same room, you just have to declare two different house parts for this room. In the following example, you can see two declared sensors. The first one is a motion detector. When triggered, it sends a signal to the audio alarm in the kitchen, which will be triggered only if it is enabled. The second sensor is a smoke detector which triggers an audio alarm in the entrance. Note the broadcast parameter. It signifies that every actuator of the same type in the house will also be triggered.

```
{"sensors": [
              {
                  "type": "motion",
                  "actions": [{"housePart":"Kitchen", "actuator":"audio-alarm"}]
              },
              {
                  "type": "smoke",
                  "broadcast": true,
                  "actions": [{"housePart":"Entrance", "actuator":"audio-alarm"}]
              }
          ]}
```

4.3 Disable controllers

This one is self-explanatory. You can dynamically disable/enable controllers at runtime, effectively changing the observable behavior of the system. This is simply a clever use of boolean conditions modified by the running `Scenario` within the `HomeController`.

```
public abstract class Controller {
  public String type;
  public double value = 0.0;
  private boolean enabled = true;
  private boolean inverted = false;
  ...
}
```

4.4 Inverting controllers

Maybe you don't want an action to execute when a sensor is triggered, but when it is not. In the parametrization component, you can define an inverted sensor. This effectively doubles the possibilities as each sensor can now have two different behaviors. For example, you could want your door to close when the humidity sensor detects rain. Once again, the implementation is simple with a boolean condition checking if the `Sensor` is inverted.

```
static void triggerActions(Sensor sensor){
    Actuator[] aList = sensor.getActuatorList();
    for(Actuator cActuator : aList){
        ...
        else { // Trigger one in given housePart
            if (!sensor.isInverted())
                cActuator.trigger();
            else
                cActuator.reset();
        }
    }
}
```

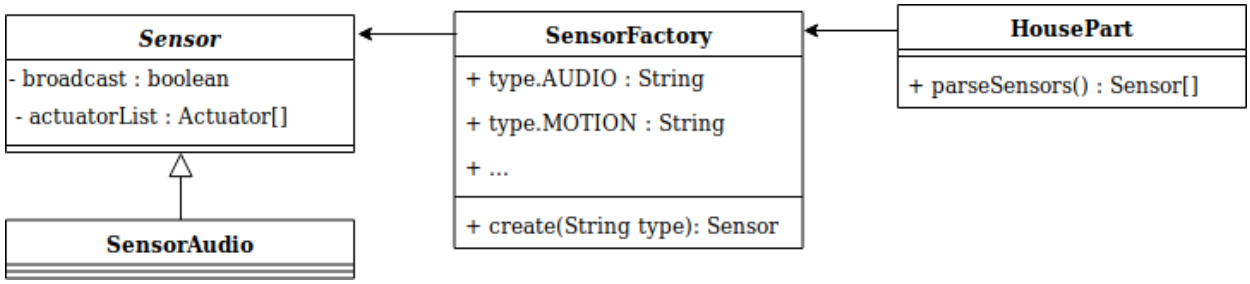
4.5 Summary

In brief, we have two major parametrization types. The static parameters are loaded with the JSON file (*e.g.: the house layout*), the dynamic parameters are real-time modifiable behaviors (*e.g.: the controller (de)activation*). We have implemented 8 different possible actuators that can independently be enabled, 13 sensors types that can be disabled, inverted and broadcasted. This means that for each house part we can have $(8*2)*(13*2^3) = 1664$ different possible combinations (some of them would of course be useless), leading to huge amount of possibilities for the entire house. It is important to note that the JSON only serves as an initializer for the system. Therefore the entire configuration is accessible in real time and can be modified (we could then erase the old configuration with a new one for the next initialization). Implementing user commands to introduce even more parametrization would be easily done thanks to that and the used Design Patterns (see next).

5 Design Patterns

5.1 Abstract Factory

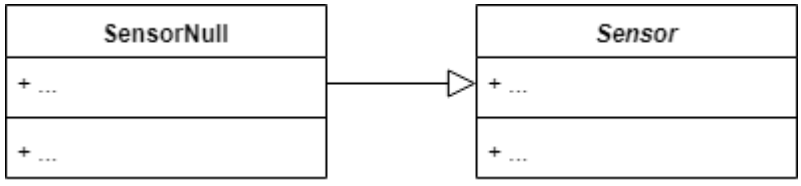
We have three factories to create the controllers. `SensorFactory`, `ActuatorFactory` and `ConnectedObjectFactory`. We do not have a higher-level factory because it would only reduce readability in the context. What motivated the use of this design pattern is that we had, by design, to parse the controllers from the parametrization component. This parsing was redundant in the code so we had to extract the method. Extracting the creation of multiple objects that inherit from the same parent was easier done with this factory pattern.




```
public static Sensor create(String type, boolean broadcast){
  switch (type){
      case Sensor.AUDIO:
          return new SensorAudio(broadcast);
      ...
      default:
          return new SensorNull();
  }
```

5.2 Null Object

When parsing controllers, we don't null erroneous controllers. Instead, we assign them a default `SensorNull` type. This allows to lighten the code by not having to heavily charge the code with defensive measures against wrong configurations that could not be parsed. The Null Object will be an object of the same type as his parent, but will fail quietly (simply by doing nothing) instead of throwing a `NullPointerException`.




```
public static Sensor create(String type, boolean broadcast){
  switch (type){
```

```
        case Sensor.AUDIO:
            return new SensorAudio(broadcast);
        case Sensor.BADGE:
            return new SensorBadgeDetector(broadcast);
        ...
        default:
            return new SensorNull();
    }
}
```

5.3 Facade

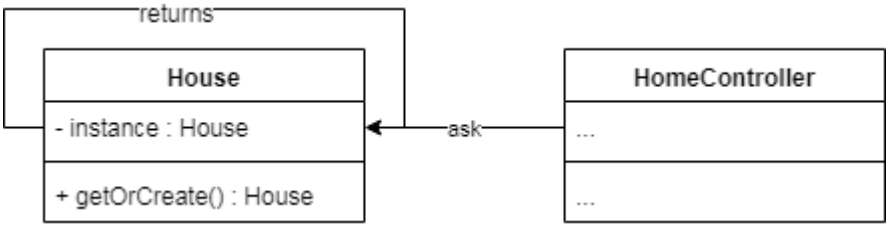
The `HomeController` is a facade to the system. It means that as a programmer you could interact and integrate the home automation system without caring about the concrete implementation of the houseparts, the parsers, the sensor communications, ... This pattern is very often used in large systems with many objects within a configuration and is almost always implicitly used with large polymorphed implementations. *Note that we don't put the class diagram here as the relevant part of the diagram is the entire implementation. Refer to the updated complete class diagram and note how you can interact with a single class to create the whole system.*

```
public static void main (String[] args){
    myHouse = House.getOrCreate("src/config_big.json");
    if (args.length == 0) {
        scenario = new Scenario(myHouse);
        scenario.userInput();
    }
    else if (args.length == 1) {
        userInputScenario = false;
        scenario = new Scenario(myHouse, args[0]);
        scenario.fileInput();
    }
}
```

5.4 Singleton

The `House` object is a singleton as the automation system would only need one house. We used a singloton because the system is heavily hierarchized (house, house part, sensors, actuators) and the house is central to it all. If you need to access to another house part from the current one, you need to go back to the house object. Using a singloton ensures that we won't create a new object by mistake and we enforce the use of a single parent for the system hierarchy.

```
public class House{
  private House instance;
  public static House getOrCreate(String configFilename){
      ...
```




```
  }
  // Make constructor private to disable public/protected instanciation
  private House(String configFilename) {
    ...
  }
  ...
}
```

5.5 Lesser patterns

Some patterns that are less relevant for this report were used. We can cite :

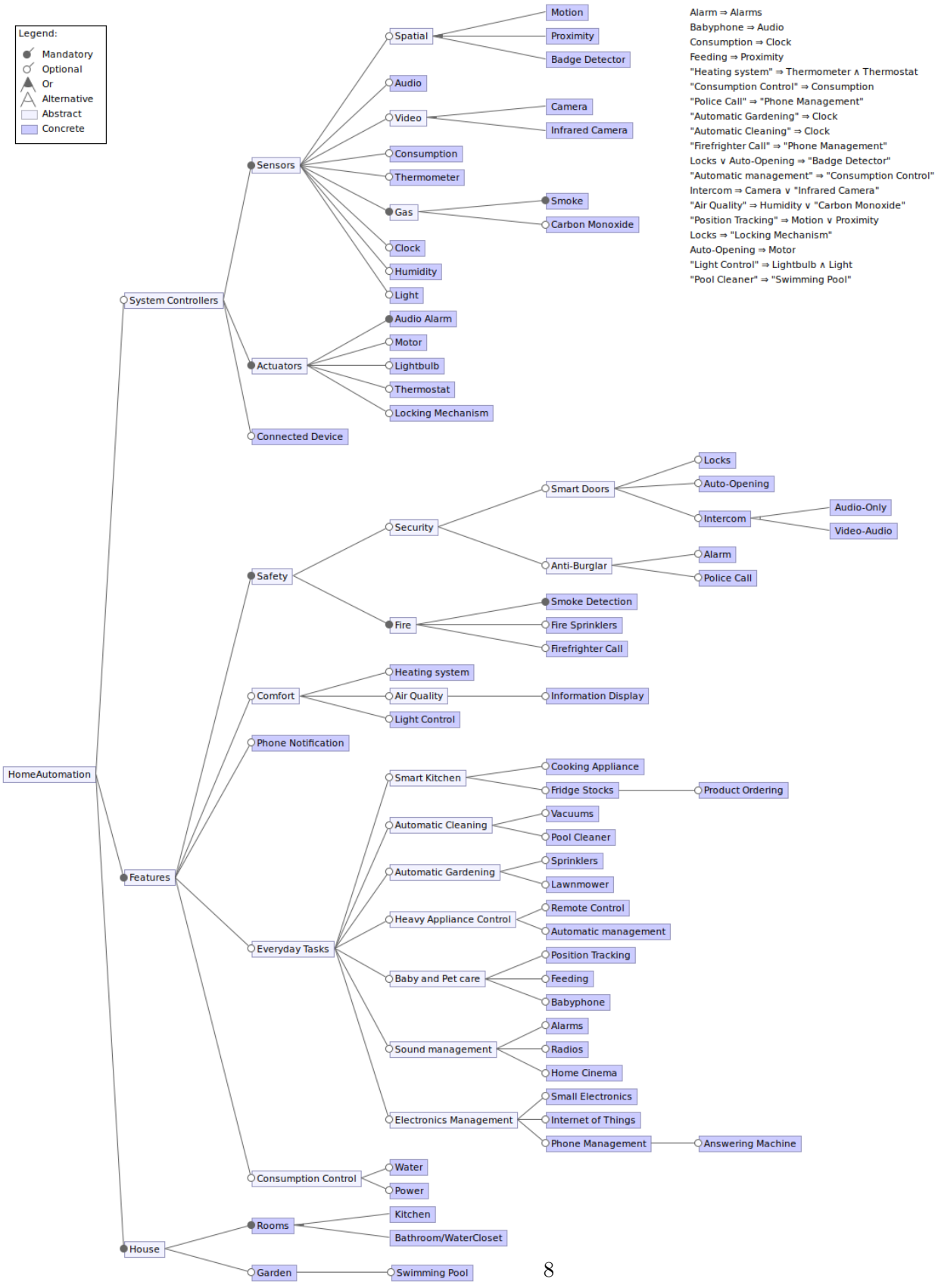
Builder According to the documentation, the builder pattern is a way to create, step-by-step, large and complex objects composed of multiple smaller objects. That is essentially what the `House` and `HousePart` do with the parsing. But we did not use the structure the pattern describes so it is only a pseudo-builder.

Iterator Natively implemented in Java, we loop over our objects with the `for each` java structure, which uses iterators.

Interpreter Technically, the command line interpreter is an interpreter pattern as it translates human-readable commands to concrete system actions, but we feel the interpreter is not mature enough to be cited as a fully-implemented pattern.

Front Controller `Scenario` is a singleton (static) class which handles the requests from multiple sources (file and user input), translates them to system functions (`FakeEvent`) and logs the results to the system output. Once again, this was not proposed as a fully-implemented pattern, because the class is mainly a dispatcher and not a logger per se (the application is single-threaded and uses the standard output).

6 Conclusion


Legend:
Mandatory
Optional
Or
Alternative
Abstract
Concrete
Alarm ⇒ Alarms
Babyphone ⇒ Audio
Consumption ⇒ Clock
Feeding ⇒ Proximity
"Heating system" ⇒ Thermometer ∧ Thermostat
"Consumption Control" ⇒ Consumption
"Police Call" ⇒ "Phone Management"
"Automatic Gardening" ⇒ Clock
"Automatic Cleaning" ⇒ Clock
"Firefrighter Call" ⇒ "Phone Management"
Locks ∨ Auto-Opening ⇒ "Badge Detector"
"Automatic management" ⇒ "Consumption Control"
Intercom ⇒ Camera ∨ "Infrared Camera"
"Air Quality" ⇒ Humidity ∨ "Carbon Monoxide"
"Position Tracking" ⇒ Motion ∨ Proximity
Locks ⇒ "Locking Mechanism"
Auto-Opening ⇒ Motor
"Light Control" ⇒ Lightbulb ∧ Light
"Pool Cleaner" ⇒ "Swimming Pool"
HomeAutomation
System Controllers
Sensors
Spatial
Motion
Proximity
Badge Detector
Audio
Video
Camera
Infrared Camera
Consumption
Thermometer
Gas
Smoke
Carbon Monoxide
Clock
Humidity
Light
Actuators
Audio Alarm
Motor
Lightbulb
Thermostat
Locking Mechanism
Connected Device
Features
Safety
Security
Smart Doors
Locks
Auto-Opening
Intercom
Audio-Only
Video-Audio
Anti-Burglar
Alarm
Police Call
Fire
Smoke Detection
Fire Sprinklers
Firefrighter Call
Comfort
Heating system
Air Quality
Information Display
Light Control
Phone Notification
Everyday Tasks
Smart Kitchen
Cooking Appliance
Fridge Stocks
Product Ordering
Automatic Cleaning
Vacuums
Pool Cleaner
Automatic Gardening
Sprinklers
Lawnmower
Heavy Appliance Control
Remote Control
Automatic management
Baby and Pet care
Position Tracking
Feeding
Babyphone
Sound management
Alarms
Radios
Home Cinema
Electronics Management
Small Electronics
Internet of Things
Phone Management
Answering Machine
Consumption Control
Water
Power
House
Rooms
Kitchen
Bathroom/WaterCloset
Garden
Swimming Pool