Reflections on the Object-Oriented Design of Embedded Systems

Antje von Knethen, AG Software Engineering, University of Kaiserslautern,
D-67653 Kaiserslautern, Germany, vknethen@informatik.uni-kl.de,
Barbara Paech, Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany, paech@iese.fhg.de

Abstract

The methodical development of object-oriented embedded systems is still an open issue. We discuss the design of complex control functionality through object interactions. Borrowing from the domain of information systems, we argue for the use of independent control classes.

1 Introduction

While the object-oriented paradigm is widely accepted in the domain of information systems, the object-oriented modeling of embedded systems is still in its infancy. There are only few established object-oriented methods and they leave many unresolved questions. This holds particularly for the question how a complex system functionality can be distributed among objects.

In the domain of information systems, the introduction of independent function objects has proved to be an answer to this question [Jac92, Pae99]. Function objects control the modification of data in different objects. In the following sections, we argue for transferring this idea to embedded systems, where instead of data modifications complex dependencies among sensors and actuators are controlled.

We discuss our ideas based on a case study taken from the domain of building automation.

2 The Case Study: Temperature Control Within a Building

The case study was developed with support of the SFB 501 "Development of large systems with generic methods" at the University of Kaiserslautern.

The following list describes a subset of the requirements for a temperature control system within a building:

- If a person enters an empty room, the room has to be heated-up to an adjustable comfort temperature in an adjustable time slot.
- If the last person leaves a room, the comfort temperature has to be maintained during an adjustable time slot.
- If the window in a room is open, the adjustable minimal temperature has to be maintained irrespective of the presence of persons in the room.
- If someone is present in a room, the comfort temperature is exceeded, and the outdoor temperature is lower than the current room temperature, then the window has to be opened.
- The adjusted temperature of the heating boiler must not exceed the maximum of the radiator temperatures requested by the different rooms.

3 Object-Oriented Design

The aim of object-oriented design is the realization of the requirements through interacting objects. Two kinds of designs can be distinguished: an application-oriented design that encompasses objects resulting from the application and a technical design that reflects the concrete hardware and the execution platform (e.g., with parallel or sequential processing). Here, we look at the application-oriented design only.

Figure 1 shows a first class diagram for the case study. We repeat classes in the diagram for layout reasons. To indicate particular members of a class (e.g. to distinguish the outdoor temperature sensor from the room temperature sensor) we use comment boxes.

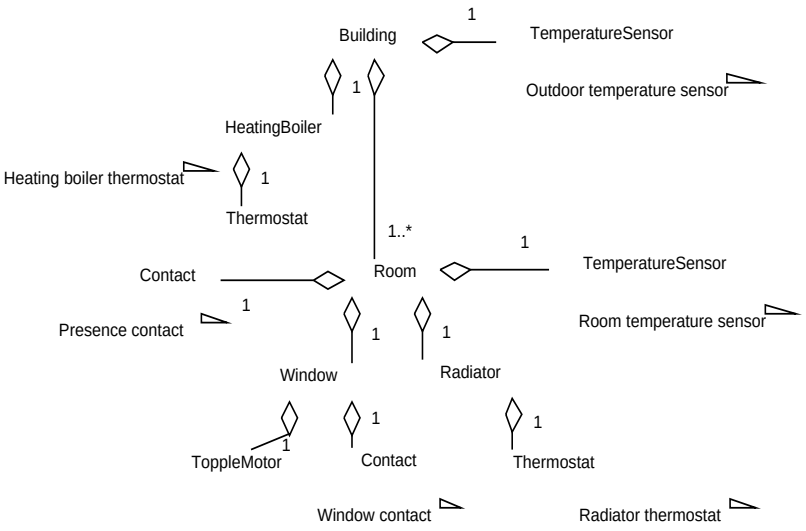

Figure 1: Class diagram for the case study

The class diagram mirrors the physical structure (the building architecture). A class is formed for each construct of the building (e.g., room or window). With aggregations, the physical structure is represented (e.g., a room has a window and a radiator). Furthermore, classes representing sensors and actuators are introduced (e.g., temperature sensor or thermostat). They are related through aggregations to the building constructs, in which they are located (e.g., each room has a temperature sensor or the building owns an outdoor temperature sensor). The sensor classes pick up the current values of the hardware sensors. The actuator classes adjust the hardware actuators.

This first class diagram results from requirements documents not given here, which describe the building architecture, the sensors, and the actuators. It does not contain any classes or operations implementing the behavior of the system (e.g., a class temperature control).

In the following sections, we discuss whether the behavior of a system should be implemented by one independent class, various independent classes or by operations of physical classes. The following criteria are important:

- The behavior of the system consists of controlling different dependencies among several sensors and actuators. The control of a dependency (called *control task* in the following) requires two decisions: The first decision determines whether an action(i.e., the control of actuators) is needed based on the (combined) sensor values (e.g., it is determined that cooling is necessary based on the values of the room temperature sensor and the presence contact). The second decision determines what action should be performed (e.g., the window has to be opened to cool off, if the outdoor temperature is lower than the comfort temperature, otherwise, nothing should be done).

- Various control tasks could conflict (e.g., the temperature control and a safety control: the temperature control wants to open the window to cool off and the safety control wants to close the window because of hazardous conditions). Conflicts have to be solved by priorities.
- Other classes can be informed of changing sensor values in an active or passive way (i.e., the values are requested periodically or the sensor class reports changes to controlling classes). Likewise, the control of an actuator class can be active or passive (i.e., an actuator class asks for new signals periodically or it gets the signals directly by controlling classes). Typically, passive sensors and actuators are preferred because periodical requests lead to a high number of object interactions.
- The determination of an action that should be performed does not only depend on the values of sensors, but also on properties of physical classes. For example, the control algorithm to calculate the heating-up temperature (the temperature guaranteeing that the comfort temperature in a room will be reached within an adjustable time slot) depends on the size of the room.
- The aggregation among physical classes will be interpreted as follows: the super class is able to call operations of the sub classes but not vice versa. This means in particular, that sensor values first get known to classes that aggregate a sensor class and that actuators are controlled by aggregating classes.
- Typically, the number of communication relations among classes should be reduced.

Strict observance of the given criteria leads to the following system structure: Classes on the top level of the hierarchy contain the controlling operations because they have the easiest access to all sensors and actuators. For example, a controlling operation "temperature control" could be assigned to the class *Building* because the outdoor temperature and the heating boiler can be accessed directly. This assignment is not intuitive because the temperature control takes place for each room separately. Furthermore, this is not realistic because various rooms of a building have to be maintained. This strong concentration of functionality and interaction only to few classes is not desirable.

If the control is moved to classes on lower levels of the hierarchy - in the example to the class *Room* or to the actuator classes directly - the following fundamental problems arise:

- If the control task is assigned to classes on upper levels (e.g., in the class *Room*), there is still the risk of overloading when additional control tasks (e.g., light) are integrated.
- If the control task is assigned to classes on lower levels, a high number of messages is necessary to transmit values from and to sensors and actuators (e.g., the value of the outdoor temperature sensor have to be transmitted to the building and from the building to the room or the radiator).
- If the hierarchy is taken strictly, classes on upper levels have to request values of sensors and actuators periodically and have to deliver the values to classes on lower levels (e.g., the building requests the value of the outdoor temperature sensor and delivers the value to the room). This implies a further overhead of communication.
- Transfer of values or periodical requests for values could be avoided by an introduction of communication relations among sensors and actuators. The relations can lead to an overload of these classes.

This discussions shows that the assignment of a whole control task to one physical class is unsatisfying. Thus, a partitioning of the control task could be useful.

In our example, one class could monitor the occupancy of the room and the room temperature. Based on these values, the class could determine whether the current temperature is too high, too low or fine. If the current temperature is too high, another class could monitor the outdoor temperature and determine whether the window can be used to cool off the room. A third class could activate the topple motor of the window to cool off the room.

These partial functions could be integrated in the physical classes. But still, there is the risk of an overload of the central class. In addition, it is difficult to locate the control task as a whole later (e.g., in the case of modifications).

Since also the assignment of partial control functions to physical classes does not lead to a satisfying solution, the introduction of control classes seems to be useful. This is comparable with the domain of information systems where the assignment of complex functionality to data classes has to be handled. In this case, the following guidelines should be applied:

- To locate the control functionality easily, a class should be introduced for each control task (e.g., a class *TemperatureControl*).
- To reduce communication relations among classes to sensors and actuators, auxiliary classes should be introduced. These auxiliary classes combine sensor values and bundle actuator signals. In our example, a class *Presence* could be introduced that stores the time a person has occupied a room and compares the time with the predefined time slots. Also, a class *RoomTemperature* could be introduced that compares the current room temperature with the aspired (comfort or minimal) temperature. Typically, the auxiliary classes are used by different control tasks (e.g., the class *Presence* is important for the light and the temperature control).
- The control classes should not be integrated in the aggregation hierarchy, because they communicate with the physical classes as well as with the sensors and actuators (through the auxiliary classes).
- If the control of actuators is complex, it is useful to introduce further auxiliary classes. These auxiliary classes allow to distinguish between the determination of the action that should be performed and the actual control of the actuators (not part of our case study).
- Priorities could be implemented by the actuator class, if only one actuator is affected. Otherwise, further auxiliary classes have to be introduced (not part of our case study).

Figure 2 shows the additional classes of our example.

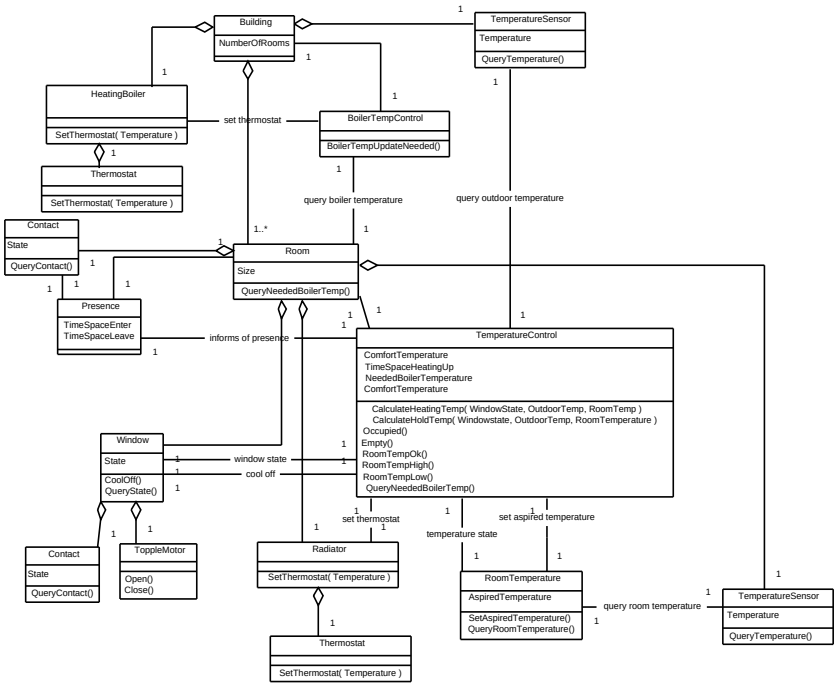

Figure 2: Class diagram extended by additional classes

The class *TemperatureControl* implements the control of the temperature in one room. The class decides which action should be performed. Furthermore, it controls the radiator to heat up and the

window to cool off. It requests the outdoor temperature to decide whether the window could be used to cool off. If the outdoor temperature is lower than the comfort temperature, the class will send a signal to cool off, otherwise, the class will do nothing.

The class *TemperatureControl* is activated by the auxiliary classes *Presence* and *RoomTemperature*. The class *Presence* requests the presence contact periodically and delivers the state of occupancy (occupied and empty) taking into account the adjustable time slots. The class *RoomTemperature* requests the room temperature sensor periodically taking into account the aspired temperature (comfort or minimal) and delivers the temperature state of the room (RoomTempOk, RoomTempHigh, RoomTempLow).

The class *BoilerTempControl* controls the temperature of the heating boiler. The class adjusts the thermostat of the heating boiler to the needed temperature. Therefore, the class requests all needed boiler temperatures from the different rooms and determines the maximum of the temperatures. Then, it adjusts the thermostat of the heating boiler to this value.

The class *BoilerTempControl* is activated by a room (initiated by the class *TemperatureControl*) that needs a new boiler temperature.

4 Related Work

OCTOPUS [AKZ96] is one of the few well-known object-oriented development methods in the domain of embedded systems. OCTOPUS combines the methods Fusion [CAB+93] and OMT (Object Modeling Technique) [RBP+91]. It starts with a description of the system functionality with use cases. Then, the system is decomposed into sub-systems. For each sub-system, an object model is developed during an analysis and design phase - similar to OMT. However, no guidelines on the distribution of complex system functionality is given. It is recommended to build a sub-system "hardware wrapper" to isolate the used hardware.

Another well-known object-oriented development method for embedded systems is ROOM(Real-Time Object-Oriented Modeling) [SGW94]. Basic modeling constructs are actors. Actors are system components which act independently and communicate with each other. The behavior of an actor is described with ROOMcharts. Similar to OCTOPUS, ROOM does not give guidelines on the distribution of complex system functionality.

In STATEMATE [HP98], a decomposition of activities as well as the control and data flow among different activities of the system are modeled. In the given examples, activities are only parts of control tasks between one sensor and one actuator. It is not clear how to proceed in the case of complex control tasks.

References

[AKZ96] M. Awad, J. Kuusela, and J. Ziegler. *Object-oriented Technology for Real-Time Systems.* Prentice Hall, 1996.

[CAB+93] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The Fusion Method.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[HP98] D. Harel and M. Politi. *Modeling reactive Systems with Statecharts.* McGraw-Hill, 1998.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[Pae99] B. Paech. *Aufgabenorientierte Softwareentwicklung - Integrierte Gestaltung von Unternehmen, Arbeit und Software.* Springer, im Erscheinen, 1999.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, 1991.

[SGW94] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-oriented Modeling.* Wiley, 1994.