Points & Deviations -

A pattern language for fire alarm systems

Peter Molin and Lennart Ohlsson
University College of Karlskrona/Ronneby
Department of Computer Science and Business Administration
Soft Center
S-372 25 Ronneby
Sweden
Peter.Molin@ide.hk-r.se,Lennart.Ohlsson@ide.hk-r.se

Abstract

After an object-oriented framework for micro-processor based fire alarm systems had been implemented and the first instantiated system had reached product status an effort was undertaken to clearly document the architecture embedded in the framework. The result of this effort was a small pattern language which is presented in this paper. Important patterns in this language are the Point pattern, which covers the abstraction of sensors and actuators, and the Deviation pattern which deals with alarms, faults and other abnormal conditions. The described framework is intended to cover a spectrum of fire alarm systems ranging from small office systems to large complex systems intended for industrial multi-building plants.

Keywords : object-oriented framework, design pattern, pattern language, fire alarm system.

1 Introduction

The evolution of many real-time embedded systems started by transforming a hardware-only implementation of some functionality to a micro-controller based software solution. The high degree of flexibility offered by the software solution then typically lead to a continuous growth in functional capability, and what was once a small and conceptually simple device soon became a large and complex system.

Most design efforts in the early stages of this evolution are focused on performance optimization and optimal utilization of limited memory. While these issues are still important as the system grows, the design focus must now shift to the problem of managing complexity. Reliability, maintainability, portability, etc. are all affected by architectural choices. This paper originates from a major architectural redesign effort undertaken by the Swedish security company TeleLarm on a family of fire alarm system products, which over the years have followed the typical path of evolution described above. The project was initiated in 1992 in an attempt to take advantage of the potential benefits offered by object-oriented technology. The concrete goal of the project was to develop an object-oriented framework which was to be used for all fire alarm system products, ranging from small home and office systems to large complex systems intended for industrial multi-building plants. The first product based on the framework was completed during 1994, and since then approximately 500 systems per year have been shipped. Not a single fault has been detected after beta-testing, and the software has proven itself easy to modify, for example adaptations for different national standards have been made unexpectedly fast. In all, the project is so far considered highly successful.

After the project was finished we needed to better express the rationale for the design. Partly this need was driven by curiosity, but the most important reason was the conviction that an explicitly expressed rationale helps protect against entropic forces during maintenance and will increase the expected life time of the architecture. Somewhat unexpectedly, it turned out to be very difficult to express why the framework was designed as it was. We found that we had more difficulties and disagreements when trying to explain design decisions than we had had when we made them. It appeared that many decisions had been based on unconscious knowledge, but a knowledge which nevertheless was somehow shared. At this time the patterns movement had started to flourish, and we came to wonder whether our difficulties in expressing why the design was good had anything to do with Christopher Alexander’s notion of the "Quality Without a Name"[3]. Maybe patterns could be a useful tool to better understand our design rationale and make it more concrete and explicit. We decided to make an attempt. The result was the pattern language for fire alarm systems which is presented in this paper. It is written in the same spirit as Kent Beck's and Ralph Johnson's "Patterns Generate Architectures"[4], that it should be possible to use the patterns as a set of axioms from which the architecture can be logically derived. We do not think this is a goal which will necessarily be reached, but must still be strived for.

The remainder of the paper presents our pattern language which consists of five patterns. Each pattern is expressed in the classic pattern format with three sections: context, problem and solution. Following the pattern descriptions we end the paper with some concluding remarks.

2 The Deviation Pattern

The Deviation pattern gives a compact representation of the state of the system.

Context:

The purpose of a fire alarm system is to survey a plant, a building, or some smaller unit like an office or an apartment. The system makes use of a number of sensors distributed across the area that is being surveyed. Each sensor is connected to one of several control units. The control units, each of which is an autonomous computing node, are all connected to a common communication network and they normally operate together as a single integrated system. A fire alarm system is the kind of system which does not do very much most of the time. The key function of the system is its ability to detect when something out of the normal occurs, i.e. a fire or an indication thereof, and generate an alarm. When this happens the fire alarm system takes appropriate actions such as alerting people in the building through alarm bells and text displays, invoke extinguisher systems, and automatically call the fire brigade. The system can handle several alarms simultaneously. Information about each alarm, such as time, location of detection or informative texts can be presented to fire brigade staff and other users. Which actions to take when an alarm is generated and how the alarm should be presented are to a large degree determined by national and international product standards, for example the European Standard concerning Fire Alarm Systems[2]. Other actions, such as the closing of fireproof doors, the automatic initiation of fire extinguishers or sending text information to presentation displays, are determined by the system configuration for the particular installation.

Although fire indications in the environment is the main thing to be detected by the system, it also continuously monitors itself for abnormal conditions internally. Such conditions may be faults, for example when some communication channel is broken or when a backup battery is failing. There is another category of less critical abnormal conditions, disturbances, such as temporarily disabled sensors or dirty sensors. When faults and disturbances are detected, they cause actions to be taken in a manner similar to how fire alarms are handled.

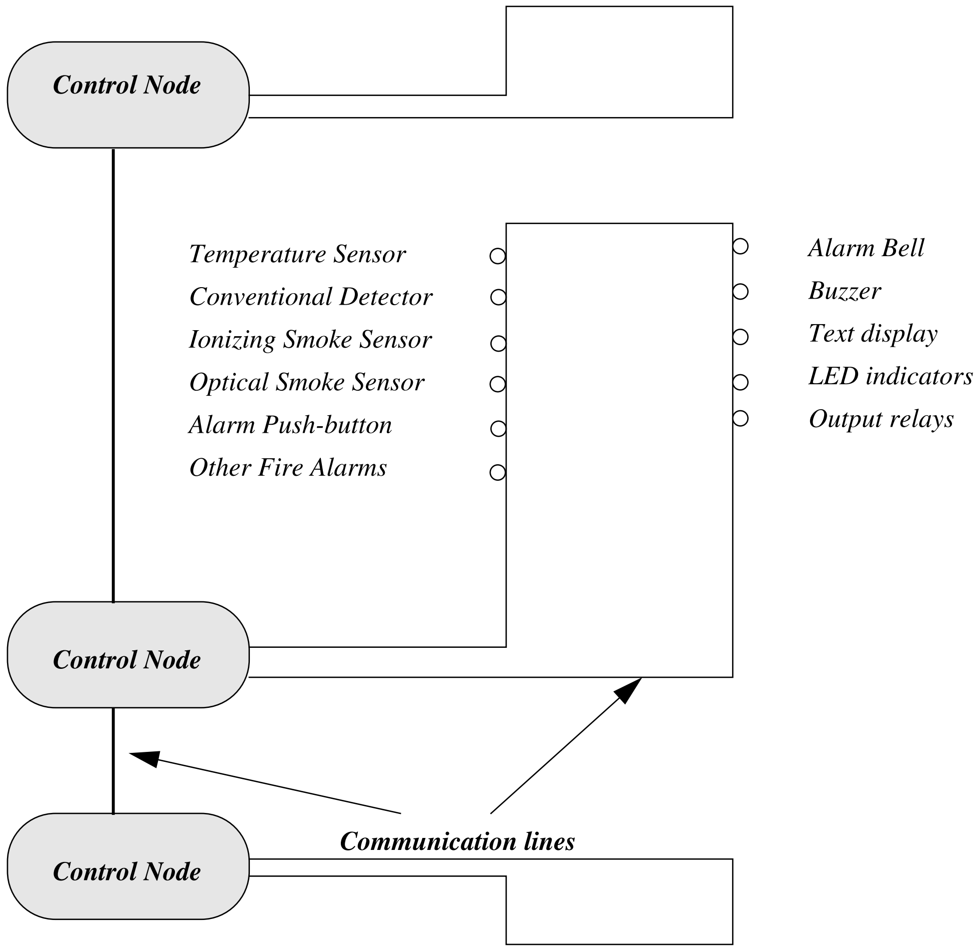

Fig. 1 System configuration example

Problem:

How do you implement the dependencies and information flow between alarm detection and actuators, user interfaces, and other outputs? This behaviour is completely independent of the distributed nature of the system; an actuator may depend on some particular input regardless of whether it is connected to the same control unit or some other control unit in the system. The problem is that the number of inputs in the system can be very large, and that the number of control units can also be large.

If every control unit would store the current status of every input sensor, it would place heavy demands on the memory capacity of each control unit. An alternative would be for each control unit to have only a proxy for remote inputs. Each proxy would consist of only a system wide reference and any request to it would be forwarded to the control unit where the input really existed. However, even storing as little as one reference per input in each control unit requires a lot of memory space.

On the other hand, the ratio of alarms, faults and disturbances to the total number of inputs is very low. Most of the sensors are in a normal state most of the time. All inputs need therefore not be *known* at each control unit since an *unknown* input can be assumed to be in a normal state. Only information about deviations from the normal state must therefore be globally accessible from every control unit.

Solution:

Therefore, represent each detected deviation from the normal state as a *Deviation* object. Use *Deviation* subclasses, such as *Alarm*, *Fault* and *Disturbance* to represent different kinds of deviations. Let deviations be the unit of distribution in the system in the sense that all deviations are replicated to all control units. Since the set of deviations defines the complete system state, this is immediately available on all nodes.

Different kinds of *InputPoints* are responsible for creating and deleting the corresponding *Deviations*. When an *AlarmPoint* is activated it creates an *Alarm*, when a *FaultPoint* is activated it creates a *Fault* etc..

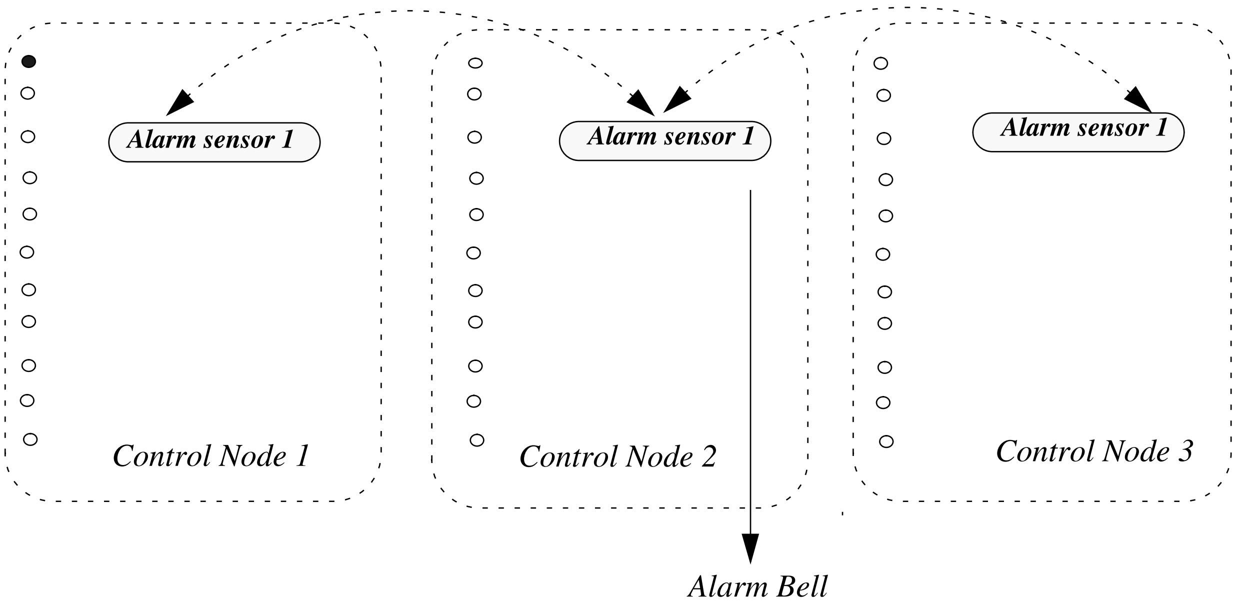

Fig. 2 Deviations in a distributed system.

3 The Point Pattern

The Point pattern clearly separates the behaviour logic from input and output devices.

Context:

To survey the building, the system makes use of a number of sensors distributed at various points across the building. When an abnormal measurement value is recorded a fire is considered detected in that point.

There is a great variety of input sensors that may be used with the system. Fire is not a phenomenon which can be directly identified. Sensors must instead measure different secondary effects such as temperature or smoke intensity and detect certain patterns of variation. In conventional detectors this analysis is rather primitive and performed by the hardware whereas modern sensors uses more elaborate algorithms in software located in the Control Node. Each of the secondary effects can in turn be measured by different methods, for example smoke intensity can be measured by either ionizing or optical methods. The situation is similar, although not with quite as many variations, for actuators and other outputs like alarm bells, alarm transmitters, buzzers, light emitting diodes (LED) or text displays.

Yet another degree of variation among sensors and actuators is how they are connected to the system. Some are connected directly to the control units and can be accessed immediately. Others, are attached to some kind of device that is connected to some communication loop and has to be accessed via different communication protocols and/or dedicated device controllers. Sometimes one device contains only one sensor, but in other cases a single device packages a number of sensors or actuators, either of the same type or of different types.

Problem:

How can the behavioral logic of the system be clearly separated from the variation among input sensors and output actuators?

How actuators should be activated when fire is detected is partly defined by product standards and partly by how the system has been configured to behave at the particular site. In either case, this behaviour is independent of the kinds of sensors, their detection algorithms, access protocols and physical packaging.

Solution:

Therefore, model the boundary of the logical part of the system as a set of *Points*, either *InputPoints* or *OutputPoints*. A *Point* is something with a binary state, either active or inactive, without references to implementation details of how this state is represented or maintained.

Use the Bridge pattern[1] to have separate device objects to implement actual I/O-behaviour of *Points*. Let the pattern of how *Points* are associated with devices be determined by the characteristics of the physical devices. Device objects encapsulate access protocols and other details of how they perform actual I/O from the *Points* as well as whether they are polled or interrupt driven.

4 The Ticked Object Pattern

The Ticked Object pattern separates task logic from scheduling strategy without concurrent processes.

Context:

Most input devices to a fire alarm system are different kinds of sensors. These sensors are periodically sampled and evaluated. Since this is the major task of the system and no other tasks have critical response times, the most suitable control strategy is to use a main periodic loop. This avoids all the well known synchronization problems associated with alternatives based on multiple threads of control.

Problem:

The disadvantage of the main periodic loop is that there is no distinct division between the task logic and the scheduling algorithm. The design of the scheduling algorithm depends only on the deadline requirements and the CPU requirement of each task and not on what the task actually does. In a multi-threaded system this separation of concerns is very clear. But the price in terms of complexity and verification difficulties of introducing multi-threading is too high for achieving this distinction.

Solution:

Therefore, represent each periodic task as a *TickedObject*. This abstract class defines a pure virtual function *Tick*, and different kinds of tasks are implemented as subclasses which define the *Tick* operation as needed. Define scheduling classes which iterate over a collection of *TickedObjects* according to different scheduling strategies. These schedulers can range in complexity from simple cyclic schedulers to various priority schedulers to advanced schedulers which dynamically choose other schedulers to collaborate with. With the abstract coupling to the *TickedObject* interface, there is a clear separation of concern between task logic and scheduling strategy and they can easily

be changed independently.

By applying the Composite pattern[1] and making schedulers inherit from *TickedObject* the scheduling can be further modularized by building a hierarchy of different schedulers.

5 The Pool Pattern

The Pool pattern provides a uniform way of accessing all instances of a class, and make the distribution on different nodes transparent.

Context:

Although outputs and actuators can be programmed to depend on particular inputs, it is more common that they depend on aggregated conditions. The most important condition of this kind is of course whether there are any alarms in the system, but there are also many other cases. For example, certain indicators may show if there are any disabled sensors in the system, or if there are any faults present. These dependencies can be expressed in terms of subsets of the current deviations which fulfil certain criteria.

Another situation where there is a similar need is in the presentation of the system status to human operators. An operator can ask the system to list all active alarms, all faults which are not acknowledged or all dirty sensors which need to be replaced. In this scenario the user sees the system as a database to which he can make a number of different selection queries.

Problem:

How do you make operations on the set of all instances of a class, such as retrieving the subset which fulfil some criteria? In an application which uses a relational database this is a trivial problem, but although it is meaningful to speak about "the set of all instances" in object-oriented environment there is no explicit support for this notion in a language like C++. Furthermore, in a distributed system it is not only the set of all instances within one address space which is of interest but the set of all instances on all nodes in the whole system. Selection, iteration, etc. should be made on this complete set and the mechanism should be independent of where instances are located.

Solution:

For the relevant classes, collect all instances in *Pools*. A *Pool* is a virtual container which contains all instances of the class in the whole system and there is a copy of each pool on all nodes in the system. The instances which belong to the same node are actually contained in the *Pool*, whereas instances belonging to other nodes may be accessed according to different strategies such as duplication to all *Pools*, remote iteration, or by using the Proxy pattern[1]. Different strategies give different trade-offs regarding fault tolerance, memory consumption or response time. The strategy can be chosen differently for different *Pools*. Alarms for example, must be quickly available and also because a fire may damage the network should be duplicated immediately to all nodes, whereas rarely performed device monitoring involve a much larger number of instances but have lower demands on response times and availability and can therefore be implemented by remote iteration techniques.

6 Lazy State Pattern

The Lazy State pattern implements objects, whose state dependent behaviour is rarely executed, in a flexible and efficient manner.

Context:

In the fire alarm system the behaviour of *Points* may be affected by user commands. A user can for example disable one or more *InputPoints* which thereby do not generate alarms. Another example is test mode commands which put a set of *InputPoints* into a specific mode where an alarm condition only results in partial output behaviour. These commands do not affect normal operation, i.e. sensor evaluation. The command dependent behaviour only occurs after an alarm condition has been detected, and this occurs very rarely.

For each behaviour changing command there is an inverse command which should make the *Points* return to normal behaviour. For example, for each disable command there exist a corresponding re-enable command. Typically only a few commands are applied simultaneously, but these commands can nevertheless affect many *Points*.

This variation in *Point* behaviour is naturally modelled by finite state machines. Using the State pattern[1], each behaviour would be represented by a subclass to an abstract *PointState* class. However, in many cases different commands have a similar effect on the *Point*. One example is *OutputPoints* which can be disabled either by a specific command for an individual *OutputPoint* or by a command for all *OutputPoints* of a particular category, for example all alarm bells. Each of these similar commands have its own inverse, so if a *Point* is disabled both individually and because it is an alarm bell, it returns to normal behaviour only after both inverse commands have been applied. The commands therefore require partial command history to be stored, which in a pure state machine would require additional states. This state transition logic significantly increases the complexity of the state machine compared to states required to implement the variations in behaviour. Moreover, each of the various commands would requires a separate operation on the *Point* class, for example disableIndividually(), disableAsAlarmBell(), etc.

Furthermore, new commands can be easily be imagined, for example, disabling physical locations like rooms instead of a number of sensors corresponding to that room. Such changes are much more frequent than changes in basic alarm detection behaviour.

Problem:

How should the state machine behaviour of *Points* be designed in order to have a flexible state transition logic? It must be done in a way that does not make the *Point* interface polluted with command dependencies.

Solution:

Therefore, replace the member variable state in the State pattern[1] by a member function state(). Current state is thus not stored explicitly but computed each time state dependent behaviour is invoked.

The state function on Points is implemented by examining a *Pool* of *Commands* in the spirit of the Command pattern[1]. A *Point* is, for example, in a disabled state if there is a *DisableCommand* in the *Pool* which affects this *Point*. The execution of a *DisableCommand* consists of storing it in the *Pool*. The execution of a *ReEnableCommand* consists of locating the corresponding *DisableCommand* and removing it.

7 Design Considerations

In addition to the patterns above, there are a few more design considerations which have influenced the framework architecture.

7.1 Scheduling strategy

One very important design decision is to decide how to combine *Points* and *Devices* with *TickedObjects*. Which objects should be active, and regularly ticked and which should be passive and only called when appropriate data is available? How can the benefits of a polling strategy be combined with the benefits of a reactive system? The problem with a polling system is that there are synchronisation problems where the difference between polling and the time when data arrives must be compensated by a blocking wait. On the other hand, highly reactive systems are often based on concurrency, in which case there are well-known problems of process synchronisation and shared memory accesses.

The chosen strategy must also be efficient using small amount of memory and not wasting CPU-time unnecessarily. The strategy must also make it possible to fulfil real-time constraints of different nature, hardware access constraints as well as high level throughput and response time requirements.

The scheduling solution in the fire alarm system is based on a three-level structure, at the lowest level, an abstraction of the hardware is defined. Low level hardware manipulation and interrupt routines are hidden in that part. Specific hardware constraints are handled by low-level routines such as interrupt routines.

The second level consists of a number of active devices or device drivers that access the hardware abstraction level using a polling mechanism. The devices or device drivers are specified as ticked objects and scheduled accordingly. Response time on this second level is limited by the polling frequency and the scheduling algorithm at this level. Possible variation in polling frequency is covered by buffering between the lowest level and the device level. Some response time requirements can be handled directly by the device drivers.

The third level consists of the logic of the application. The objects on this level are defined in a reactive and passive way. Whenever a device or device driver find data to be handled by the application, the corresponding object is called with the data in question. Normal response to incoming data must pass this level before correct output can be generated.

The highest level is the most portable part where the logic and the behavioral functions can be reused on different platforms. The proposed solution makes it possible to chose different implementations in response to different performance requirements.

One example of the chosen strategy is that the *Points* are defined reactively with a method that handles incoming sensor data. The corresponding device driver is ticked, the hardware abstraction layer is polled, and the received data is made available to the corresponding *Point*. Another example is the user interface, where the device driver in this case handles a small keyboard and an LCD. The device driver is modelled as a ticked object, but the actual user interface logic is modelled reactively as a state machine. Whenever a key is read by the ticked device driver it is fed into the user interface logic as an event.

Control and scheduling considerations has been removed from the application level, which only handles incoming events. Scheduling decision are made close to the hardware, ensuring high performance.

7.2 Interleaved data processing

Another important design decision is related to performance. A fire alarm system spends the majority of its CPU cycles on periodically evaluating sample values from sensors. Sample data arrives from device controllers which poll a number of devices in a round-robin manner. The capacity of the system can be measured by the number of sensors it can evaluate per time unit. Since the worst-case response time corresponds to a case where a device just is read before it enters an alarm state, the capacity directly affects the worst case response time.

Evaluation of a sensor value is therefore the most performance critical operation which needs to be carefully designed. The problem is to achieve high performance by making the communication line and hardware work at highest possible rate. At the same time decisions must be made on a higher level, deciding if an alarm has occurred, in which case data should be written to other output devices, or if the system should continue by polling the next device. The sequence of operations could be reduced to a sequence of four parts, read-request, data-reading, analysis and decision. The first is to request the reading of a sensor, the second is to wait until data is available, the data must be analyzed, a decision must be made whether to continue polling or activate outputs.

The capacity of such a system depends on the capacity of the hardware plus the time to analyze data. An alternative is to let low-level parts make a guess to poll the next sensor without waiting on higher levels for the correct decision. Even if this approach seems to increase alarm response time by one hardware cycle, it is on the contrary improving worst case response time. Assuming there are 100 polled sensors on one communication line, worst case response time corresponds to the handling of 99 normal values and one alarm value. Therefore the 99 normal situations need to be optimized, and the low-level decision guessing provide a higher degree of concurrency thereby minimizing overall worst case response time. This solution limits the worst case response time by allowing the hardware to operate at it's maximum rate.

There are in principle two ways of achieving this kind of parallelism, the first is by using request buffers from the analyzed points, where the old request is treated before the new is analyzed. The other solution is to hardcode the default polling strategy into the low-level parts, in which case only unusual actions must be sent from the *Point* level to the low level. There is one possible improvement in flexibility, where the default strategy, dynamically is sent to the low level part.

The three implementations differs in the amount of flexibility given to higher levels, the performance, and the memory usage. In the buffer solution the high level parts decides exactly what the default action is by sending corresponding requests to the buffer. In that case we have very high flexibility. Memory usage depends on the size of the request buffer, which must hold as many request depending on how higher part are scheduled and how the produces requests. Performance is affected only by the time to remove one request from the buffer. The hardcoded approach gives the best performance, and uses the minimum amount of memory because no buffer is used for default requests, but provides no flexibility. In the case of sending a strategy object to the low level, we have high flexibility, but performance is difficult to judge, it depends on the complexity of the strategy object.

To summarize this discussion, one major design principle was adopted, the interleaving of hardware and communication processing with data analysis, sacrificing best case response times but gaining worst case response time.

8 Concluding Remarks:

External circumstances determined that we designed the system first and then did patterns mining afterwards. Would we do differently a second time? Is it better to do the two activities more in parallel? The answer is yes and no. Although the design was heavily scrutinized during patterns mining we found no major flaws. So design quality could probably not be improved much. On the contrary, it could suffer from parallelization. Design is hard and require many rapid iterations. If each iteration would have to include the even harder task of making all decisions explicit, the iterations can not be as rapid, and there would not be time to do as many of them. On the other hand, a moderate dose of pattern exercises during design could well eliminate the need for some of the iterations made and thereby give time for making others.

Another reflection on this pattern work is concerned with the best level of generality for a pattern language. Our pattern language is specific to fire alarm systems, but most of its patterns are not that specific. They could probably be re-written for a wider range of systems: to other kinds of alarm systems, to various kinds of detection systems, and at least some of them to any kind of distributed measurement and control systems. Although the main purpose of our patterns mining was not to reach a wide audience, the writing of this paper rendered a slightly different perspective. Given that you want to make the mined knowledge to be as widely accessible as possible, what is the optimal level of generality? Stating the patterns in the most general terms possible make them more directly applicable to many situations. But the drawback is that something which is generally expressed is harder to understand to the level where the reader can actually apply it. In many cases a more specific story can be a better carrier of general knowledge; analogy can be more powerful than specialization. There is no conclusive answer to this question, no definitive criteria that can be applied, but we think it is better to err on the specific side than being too quick on going meta.

Acknowledgements

We would like to thank Östen Lundgren and Björn Svensson at TeleLarm for their support of the project and Rutger Pålsson and Magnus Håkansson for their design contributions.

References

[1] Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[2] CEC/TC72 N752:Final Draft prEN54-2

[3] Christopher Alexander, S. Ishikawa, M. Silverstein, M. Jacobsen, I. Fiksdahl-King and S. Angel, *A Pattern Language*, Oxford University Press, 1977.

[4] Kent Beck and Ralph Johnson, *Patterns Generate Architectures*, Proceedings of the 8th European Conferences on Object-Oriented Programming, Bologna, Italy, 1994.