Software Architecture as a Combination of Patterns

Kent Petersson and Tobias Persson
Ericsson Microwave Systems

Dr. Bo I. Sanden
Colorado Technical University

Ericsson Microwave Systems in Sweden was confronted with the problem of constructing a radar system that could withstand the replacement of hardware and operating system software and be adaptable to different customers' functional requirements. This was accomplished by means of software architecture with highly independent and flexible components that is a combination of four design patterns: Layers, Pipes and Filters, Observer, and Model-View-Controller.

The computer science community has introduced design patterns as a means of capturing and documenting solutions to common software problems [1, 2]. An important aspect is the combination of patterns. In "Design Patterns" [1], the authors discuss how patterns *dovetail and intertwine* in good software. Nevertheless, combining patterns into a good overall architecture has received less attention than the individual patterns. In much of the literature, each pattern is often presented separately along with a discussion of how it fits into a resulting class diagram. This assumes that the class diagram exists when you present your patterns.

In this article, we present not only how different patterns are used in software architecture, but also how we evolved the architecture by successively integrating more patterns to deal with particular design problems. We present the individual patterns, their interaction, and the questions that arise when they are combined.

The example we use is a software subsystem of a new generation of surveillance radar constructed by Ericsson Microwave Systems (EMW) in Mölndal, Sweden. This subsystem contains the modules that handle tracking, communication with external devices, threat evaluation, etc. A major problem with the software of an earlier radar system was that the modules were too interdependent. If you wanted to use only one part for a new product, you often had to include all modules even though the functionality was not needed.

When the new architecture was constructed, one premise was to maximize reuse. The main reason was the business situation for the surveillance radar system. The trend was that each customer bought a small series but still required considerable tailoring to his or her needs. To encourage reuse, the different components had to be made more independent and flexible. A function had to be coupled to as few other components as possible, and direct dependencies had to be eliminated or minimized.

An important guideline in the architectural design was the requirement for modifiability. The following sources of modifications were especially considered:
1. Replacement of hardware and operating system.
2. Different customers' functional requirements.

> *"A principle of the Layers pattern is that the components in each layer only know of and use components in lower layers ... the components in the application layer only know and use components in the support layer ..."*

In the following, we first discuss how the system was made adaptable to new hardware and software, and then how it was designed to accommodate changes in customer requirements in general and requirements for presentation and control in particular.

Adaptability to New Hardware and Operating System

To facilitate future changes of hardware and operating system, a layered architecture was introduced. The system was structured in terms of components, which were placed in three layers with application-oriented components at the top and hardware and operating system dependent components at the bottom. This is a time-honored architectural style with the Open Systems Interconnection protocol stack as a classic example [3].

The Layers Architectural Pattern
The *Layers* pattern is described in [2]. In it, the system is structured as a number of layers that represent different levels of abstraction. We introduce the following three layers from top to bottom:
- The application layer containing the functionality required of the system.
- The support layer containing parts shared by the applications. An important role for this layer is to act as data storage for the applications. For maximum independence between applications, their input and output data is stored in the support layer. That way, the applications need not be directly aware of each other but instead depend on the database components in the support layer.
- The core layer containing components that depend on the operating system or the hardware.

A principle of the Layers pattern is that the components in each layer only know of and use components in lower layers. In our case, this means that the components in the application layer only know and use components in the support layer (and possibly the core layer). This principle automatically makes the components in the application layer independent, but sometimes forces you to introduce additional layers, potentially complicating the solution. For simplicity, it is sometimes reasonable to compromise and let some components in a layer know of each other. Still, a component should never need to know of a component in a higher layer.

Architectural Components
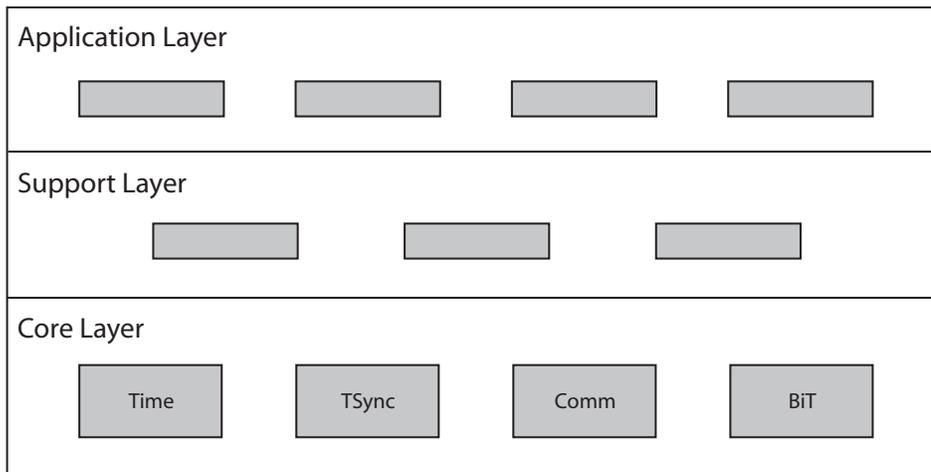The layered architecture is shown in

Figure 1: *Layered Architecture With Some of the Core Layer Components*

Figure 1. The following are some of the components in the Core layer:

- System Time (*Time*). This component allows the system to run at a user-defined time in a simulation mode. With multiple instances of this component, real time and simulated time can be used concurrently.
- Time Synchronization (*TSync*). With different parts of the system running on different hardware nodes, synchronizing the system time on each node is critical.
- Communication Protocols (*Comm*). This component includes components dealing with low-level communication protocols tailored for specific applications.
- Built-in Test (*BiT*). These modules handle hardware testing.

Adaptation to Customer Requirements

Customers have differing functional requirements for at least two reasons. First, different customers need more or less functionality depending on how they intend to use the radar system in their overall system structure. Some want a basic system while others want a deluxe version.

Second, customers must integrate the system they buy into a larger system whose interfaces vary considerably. This has led to a lot of reconstruction of the radar system over the years. The interface to the human operators is particularly variable.

To solve the problem with differing customer requirements, we designed architecture for a complete system with full functionality. It defines how the entire system would work even if EMW would only realize parts of it. This architecture is broken into subsystems that are sufficiently independent to allow for all reasonable variations. Designing a system that includes all possible variants is not a design pattern but a widely applicable design principle.

"The flexible and independent structure of the architecture was very useful when combining components to meet the customers' different needs and resulted in reduced cost as well as faster delivery of the projects."

The Data Flow Principle

The decomposition into subsystems was done according to the Data Flow Principle. In this common architectural style, you study how data flows through the system and divide it into subsystems at suitable points in the data flow. For example, radar can be divided into a transmitter, antenna, receiver, signal processor, data processor, or presentation, reflecting the path of a radar pulse through the system. Radar data processing has traditionally been structured along the same lines. One common structure is as follows: target tracking, data fusion, threat evaluation, and engagement planning. Most of those involved know and understand this structure.

The Design Pattern: Pipes and Filters

The pattern that supports data flow is *Pipes and Filters* [2]. It is intended to solve the problem of structuring a big system that transforms a stream of data. Designing such a system as a single component is unwise and makes the system inflexible and vulnerable to future changes. For this reason, the system is divided according to how data flows as shown in Figure 2. We primarily use this pattern to structure the application layer, but it also has implications for the support layer.

Composition of Layers and Pipes and Filters

The problem is how to fit both the Layers pattern and the Pipes and Filters pattern into the architecture. The Layers pattern structures the system vertically while Pipes and Filters structure it horizontally. We had to find an architecture that retains the desirable features of each pattern. Using Pipes and Filters directly on the application would make those application components that represent the filters aware of each other. This defeats one of the goals of the Layers pattern, namely application independence.

The solution was to put data storage components that work as buffers between the different applications – that is, the pipes in the Pipes and Filters pattern – in the support layer. Figure 3 shows the architecture with the Layers and Pipes and Filters patterns combined. This solution radically reduces the coupling between applications. By also making the components independent in terms of synchronization, we minimize the coupling between the components that produce and consume data. It is the case that a real-time system sometimes produces data faster than it can consume it, and a direct coupling between producer and consumer may then lead to the use of stale data. It is often better to discard the old data and continue with the relevant information.

With a separate data storage component we can isolate the problem and store valid data only without involving the producer or consumer. The producer pro-

Figure 2: *The Pipe and Filters Pattern*

Note: Arrows indicate the data flow between components.

duces data at one rate, and the consumer consumes it at a different rate. The data storage components ensure that the most current data are used.

Event Handling

The simple model with the components in the application layer decoupled by means of data storage components in the support layer works for data that are produced and consumed continuously. Other data, which may be associated with different events or operator controls, do not fit in this model. This is because changes occur rarely, but the application must react very quickly to them. In a data flow solution, an application that is dependent on a certain control would have to query the data storage component for its status quite often. Most of the time, the control would be unchanged. This is inefficient, but on the other hand, allowing the data storage component or some other application to send the information directly to another component in the application layer would create a strong dependence between components.

The solution is to introduce an event handling mechanism. Applications subscribe to an event by calling an event handler. When another application generates an event, which may carry with it other (changed) information, the subscribers are informed of the occurrence and can retrieve the additional information from the data storage components.

This kind of event handling reduces the coupling between the event-generating component and the subscribers; they need not be aware of each other. The approach is quite resilient to system changes. The event-handling mechanism works even if a certain subscriber is absent in a given variant of the system. Not even the event generator has to be present in all system variants, which will then lack certain functionality. A drawback is that event handling is an unstructured and dynamic way of information exchange comparable to exception handling and should be used restrictively.

The Design Pattern Observer

A design pattern that captures the idea behind the event handling described earlier is called *Observer* [1]. It is also referred to as *Publish/Subscribe* [2] and is used to synchronize the state of cooperating components. The component that detects the state of change publishes a message that is then forwarded to any number of subscribers.

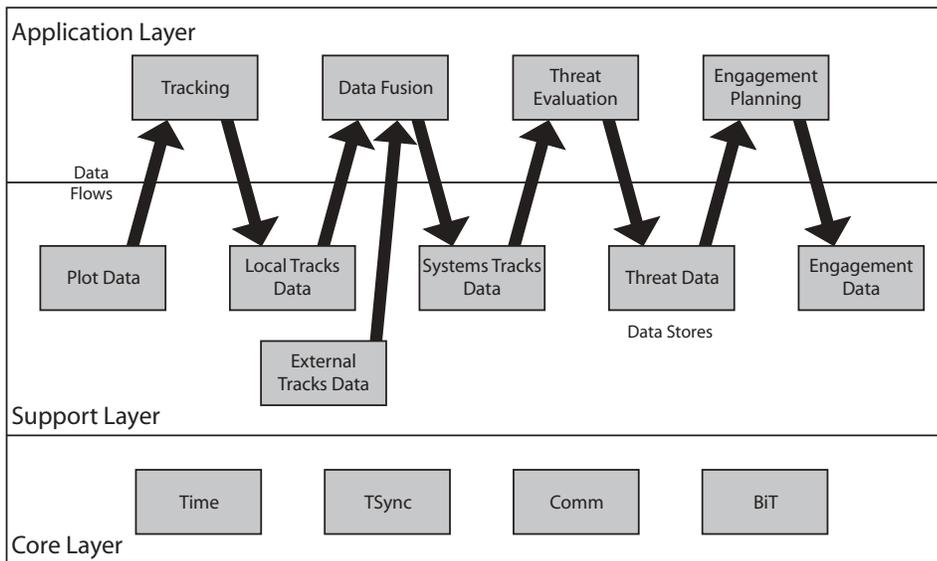The pattern solves the problem where many different components must be

Figure 3: *Combining the Layers Pattern with Pipes and Filters*

informed of a relatively rare event in a flexible way. The salient feature of Observer is the reduced coupling between the publisher and the subscribers.

Presentation and Control

Presentation and control are particularly susceptible to rework due to different customer requirements. Changes are difficult to predict because the requirements for presentation and control tend to be unique to each customer. The structuring of the user interface is often fundamental to the architecture of any application of which it is a part. There are two possibilities:

1. Separating presentation and functionality with the rationale that the presentation represents one cohesive unit, and functionality another. That is, certain functionality is considered more closely coupled to another functionality than to its own presentation. The *Model-View-Controller* (MVC) pattern captures this approach [2].

2. Tying functionality more closely to its presentation than to another functionality. A pattern that captures this view is *Presentation-Abstraction-Control* [2].

In our application, the concrete presentation and control are collected in one component: Operator Presentation and Control (OPC). This component is the entire program's interface to the human user. Changes concerning the concrete presentation and control are localized in one component according to MVC [2].

The Design Pattern MVC

The MVC pattern divides the world into a model, a presentation part, and a control

Figure 4: *The Architecture After Applying the MVC Pattern*
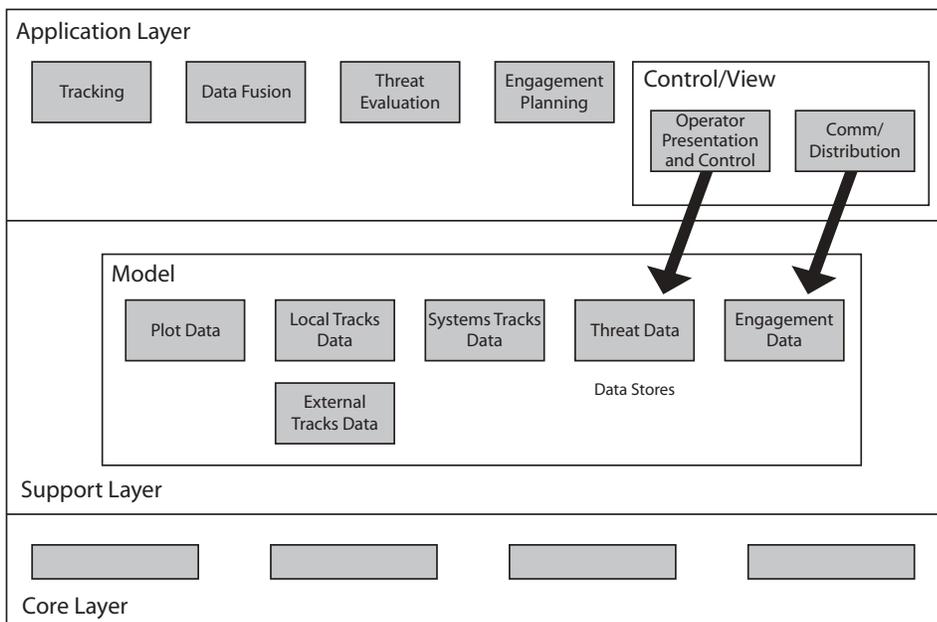
part. For compatibility with our own earlier designs, we used a variant, also described in [2], where the presentation and control parts are combined, while the model remains separate and consists of all the data storage components in the support layer.

The problem that MVC intends to solve is that user interfaces are especially vulnerable to change requests, and lead to many program modifications. A change to the presentation of one part of the system often forces a change to another part's presentation. It is hard to build a presentation with the required flexibility. Instead one can accept the situation and just structure the system so that all parts that handle the presentation are separated from the model.

Combining MVC With the Earlier Patterns

In the final architecture, MVC must obviously be combined with the earlier patterns. Figure 4 (see page 27) shows how it fits. The control/view part consists of two components: OPC for the actual presentation and Communication/Distribution for the distribution of data to external systems. The MVC pattern supports the idea that external communication is just another form of presentation. This means that external communication, which also tends to vary drastically between projects, can be handled in the same way as presenta-tion. Figure 4 shows how the components for presentation and external distribution are incorporated into the architecture. The data that are presented and distributed are in the data support layer, which is in accordance with MVC.

Conclusion

The resulting architecture shows that the four patterns, Layers, Pipes and Filters, Observer, and Model-View-Controller, can be combined quite elegantly in radar software. We have also shown that it is possible and beneficial to use design patterns in a large and very complex application. The same overall architecture has been used in seven different projects with quite different functionality in the human-machine interface, external communication, and command-and-control parts. In five of these projects, the design and implementation of the software is now completed. The flexible and independent structure of the architecture was very useful when combining components to meet the customers' different needs, and resulted in reduced cost as well as faster delivery of the projects.

We believe that you can expect to continuously improve and refactor the software architecture during its entire lifetime. For example, the OPC needs to send commands and controls to the application components in a more effi-cient way. Another example is the inner structure of the OPC component. Finally, it may be desirable to split the view and the controller of the MVC pattern into separate components.

In this article, we have only described a logical view of the components and ignored the mapping to physical processes and machines. Further work has been initiated to see how components can be structured and delivered as distributed applications that can execute on different nodes in a network.◆

Acknowledgment

References

1. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
2. Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley, 1996.
3. Day, J. D., and H. Zimmermann. The OSI Reference Model. Proc. of the IEEE. Vol. 71, Dec. 1983: 1334-1340.

About the Authors

Kent Petersson is currently working on software architectures at Ericsson Microwave Systems in Mölndal, Sweden. He has a background as associate professor at the Department of Computer Science at Chalmers University of Technology in Gothenburg. His research interests include program verification, type systems, and functional programming. Petersson received his degree in mathematics and computer science from the University of Gothenburg, Sweden.

Ericsson Microwave Systems AB
Surveillance and Communication Systems
SE-43184 Mölndal
Sweden
E-mail: kent.petersson@
         ericsson.com

Tobias Persson is manager of a software design group for Ericsson Microwave Systems and has worked at the company as a software engineer in radar projects. Persson has a Master of Science in computer science from the University of Gothenburg, Sweden.

Ericsson Microwave Systems AB
Surveillance and Communication Systems
SE-43184 Mölndal
Sweden
E-mail: tobias.persson@
         ericsson.com

Bo I. Sanden, Ph.D., is professor of computer science at Colorado Technical University in Colorado Springs. He spent 15 years as a software architect with UNIVAC and Philips. His main research interest is software design. Sanden has a Master of Science in engineering physics from the Lund Institute of Technology, Lund, Sweden, and a doctorate in computer science from the Royal Institute of Technology, Stockholm.

Colorado Technical University
4435 North Chestnut St.
Colorado Springs, CO 80907-3896
Phone: (719) 531-9045
Fax: (719) 598-3740
E-mail: bsanden@acm.org