Enterprise meets Embedded

Michael Kircher
Siemens AG
Otto-Hahn-Ring 6
81739 Munich
Germany
michael.kircher@siemens.com

Christa Schwanninger
Siemens AG
Otto-Hahn-Ring 6
81739 Munich
Germany
christa.schwanninger@siemens.com

ABSTRACT
In this position paper we give a brief overview of existing technologies and concepts that play a role in reuse for constrained environments. We start of with considerations about reuse, continue with typical limitations in embedded systems and conclude with a set of contemporary technologies and concepts that support reuse in constrained environments.

Keywords
Reuse, Patterns, Frameworks, Components, Aspect Oriented Programming.

1 INTRODUCTION

Embedded devices hold more and more software, systems get connected and the demand for ever increasing functionality in small devices forces the vendors to shorter than ever development cycles. The requirements of software for embedded systems can only be fulfilled if reuse is made possible – something we struggle for in enterprise software since at least the middle 1980ties. Can we simply transfer the concepts from enterprise to embedded systems? We think this can be done - if we are aware of the constraints.

2 REUSE

The concept of reuse can be applied in two ways:

- Architecture – Architectural reuse means the reuse of the structure of separated responsibilities and their interaction patterns, e.g. three-tier architectures.
- Source code – Source code reuse encompasses libraries and generated code. Reuse by cut and paste is usually not desirable.

The two ways of reuse are supported in various combinations by the following well known concepts:

- Architectural patterns foster reuse at a purely architectural level. Some of the most prominent examples of architectural patterns are the Layers pattern [POSA1], as used in every protocol stack, and the Microkernel pattern [POSA1], e.g. implemented in embedded operating systems and middleware products.
- Frameworks support reuse at the architecture and source code level. They take influence on the architecture by imposing a certain control flow on the application. Prominent examples are AWT as GUI framework and ACE [Schm03] as network programming toolkit.
- Components that encapsulate application functionality are typically implemented and reused as libraries, but don't impose any architecture. To keep components independent from the application infrastructure, the infrastructure services are factored out, potentially in components as well. In enterprise systems component containers implement these infrastructure services. But in embedded systems component technologies such as J2EE or the CORBA Component Model [OMG03] are too heavy weight. Instead customized solutions, closely related to domain specific languages, are used; see also [Contribution by M. Voelter].
- Another way of code reuse is libraries that group related functionality, such as mathematical functions or operating system wrapper facades [POSA2].

Aspect Oriented Programming (AOP) [Kicz97] is orthogonal to these concepts. It supports the encapsulation of responsibilities that cut across the modularization artifacts like functions or objects. This additional modularization capability fosters reuse, since only components not "polluted" with functionality that is only valid in a certain context can be reused without change. The aspect oriented term for this kind of pollution is "tangling", which means that several concerns are implemented within one piece of code.

On the other hand, since aspect orientation allows the implementation of the crosscutting concern in a single module, the crosscutting concern's implementation also can be reused.

Aspects get woven into the base code at pre-compile, load or runtime. They make application independent component development possible, like component models do, since they allow to add infrastructure services without changing the component's code manually.

3 CONSTRAINED ENVIRONMENTS

The focus of this workshop has been quite broad regarding the definition of a constrained environment. Typical constraints of a software system are

- Memory footprint – Mass produced devices save on memory because every cent saved on hardware is a million more in earning.
- CPU cycles – CPU cycles are scarce because the faster a CPU is, the more energy is consumed.
- Communication bandwidth – The communication means in embedded devices have to be redundant and highly available, this leads to restricted communication bandwidth.
- Timing requirements – Mission and safety critical applications impose strict requirements regarding timing.
- Robustness – Embedded devices often have to work reliably under rugged conditions.

The constraints are typically non-functional requirements (hard real-time not considered), mostly derived from the limited availability of resources and Quality of Service considerations.

All environments have to cope with constraints. While enterprise systems mainly have to deal with scalability issues at various levels, embedded systems are heavily influenced by stringent resource availability. In contemporary software development projects, embedded systems increasingly make use of enterprise technologies such as object orientation and component technology, while enterprise systems start to adopt model driven architecture approaches, which have a long time history in embedded systems.

4 SOLUTION PROPOSAL

For the purpose of this workshop we limit our discussion on embedded systems, as they typically have more constraints than enterprise systems.

Going back to our list in section 2, the following concepts foster reuse in embedded systems:

- Architecture reuse is possible without limitations. The mentioned Layers and Microkernel patterns are well suited.
- Frameworks have to be highly customizable for embedded systems to avoid memory consumption by unused functionality. Design patterns such as Strategy [GoF], Bridge [GoF], Visitor [GoF] help to design for adaptability and flexibility. Both are necessary for efficient reuse.
- Components can be used as a concept, but resource intensive container frameworks are too heavy weight. Code generation can help to reduce the container functionality to what is actually needed.
- AOP can be used as technology to help to implement customizable component models. Crosscutting container functionality has to be implemented as aspects; the weaver generates the application from functional components and aspects.
- Resource management patterns [POSA3] cope with the reuse of system resources at system runtime. They describe when to acquire and when to release system resources, such as memory, threads, communication connections, and the like.

5 CONCLUSION

In this paper we gave a brief overview over contemporary concepts that support reuse, but not all of them are applicable in constrained environments, yet. Generally it can be said: technologies that substantially increase the runtime overhead or memory footprint are not applicable. Nevertheless, enterprise technologies such as component models and AOP are on their way to pervade the embedded domain.

REFERENCES

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns-Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[Kicz97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda and C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect Oriented Programming. In Proc. of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science Vol. 1241, pp. 220-242, 1997.

[OMG03] Object Management Group, CORBA Component Model Specification, http://www.omg.org/technology, 2003

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal, Pattern-Oriented Software Architecture-A System of Patterns, John Wiley and Sons, 1996

[POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture-Patterns for Concurrent and Distributed Objects, John Wiley and Sons, 2000

[POSA3] M. Kircher, P. Jain, Pattern-Oriented Software Architecture-Resource Management and Optimizations, John Wiley and Sons, 2004

[Schm03] D. C. Schmidt, Adaptive Communication Environment (ACE), http://www.cs.wustl.edu/~schmidt/ACE.html, 2003