



Agile Software Development of Embedded Systems

Version : 1.0
Date : 2007.02.14

Authors

Andrew Wils
Stefan Van Baelen

Status

Final

Confidentiality

Public

Agile Deliverable D.2.13

Agile Practices for Embedded Systems

Abstract

This document describes two practices that improve the agility of embedded software development. These practices represent current research topics that help cope with and embrace change and maintain a stable development pace. The practices are component-based development and agile Quality-of-Service constraints. While this research is focused on the development of embedded systems, we believe the practices can also be applied in a broader software development context. However, due to the lack of tool support, we do not expect the industry to apply the practices in the near future.



ITEA

INFORMATION TECHNOLOGY

FOR EUROPEAN ADVANCEMENT

Agile Practices for Embedded Systems

Andrew Wils and Stefan Van Baelen
K.U.Leuven DistriNet
Department of computer science
Celestijnenlaan 200 A, 3001 Leuven
andrew — stefanv @cs.kuleuven.be

February 14, 2007

Abstract

This document ¹ describes two practices that improve the agility of embedded software development. These practices represent current research topics that help cope with and embrace change and maintain a stable development pace. The practices are component-based development and agile Quality-of-Service constraints. While this research is focused on the development of embedded systems, we believe the practices can also be applied in a broader software development context. However, due to the lack of tool support, we do not expect the industry to apply the practices in the near future.

1 Introduction

Improving the agility of processes is not only solved at the process level. When used properly, technology is an important factor to create stable and predictable iterations and control changes in the software. XP defines values (courage, simplicity, communication, feedback) that guide the use technology. Still, XP's practices are more or less technology-neutral. The only constraints XP imposes on development technologies is that they can do not obstruct a daily, automatically tested build. This document introduces three domain specific technical practices to improve agility. The practices are inspired by embedded systems development. Embedded systems development brings forth some issues that are not specifically addressed by mainstream agile solutions such as XP. These issues are:

- Quality-of-Service (QoS) constraints
- the need to separate hardware from software

¹The described work is part of the EUREKA-ITEA AGILE project, and partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders)

- the need to modularize software
- product family development
- deployment on different hardware

The practices are provided as-is and meant to complement an existing agile process.

2 Component based development

Component based development: *Modularize software to control scalability and foster reuse.*

2.1 Introduction

Object-oriented development was not the revolution it was meant to be. The reuse of classes and objects is limited, and changes to the software still propagate beyond class boundaries. Especially in medium and large software projects, the limitations of object-oriented technology become clear. Large codebases prevent good insight in the global software architecture and complexity and dependency management becomes harder. For security and/or resource scarcity reasons, object-oriented technologies are not even an option in some embedded systems. These problems break a number of assumptions of traditional agile practices, making them ineffective. Common code ownership becomes difficult to maintain, and test driven development cannot reveal fundamental architectural deficiencies.

Component based development (CBD) [2] specifically addresses scalability and reuse issues. It provides a unit of composition that is suited for deployment and reuse. In CBD, software is built using and combining isolated components. Usually, the design process starts with an initial component architecture of the system. Each component encapsulates a reusable and deployable code block. Components communicate only through well-defined interfaces, eliminating hidden dependencies and tangling. The component implementation itself can be done using any programming language, although an object-oriented language is chosen.

2.2 Technical overview

Component architectures are present in both the academic and industrial world. Popular component technologies include COM/DCOM, .NET, Enterprise Java Beans, and the Corba Component Model. Most of the technologies allow to package a component code unit to make it deployable. Most component systems also offer a component container that takes care of a number of services, such as persistency and security.

Component systems that support dynamic composition [1] usually connect components with *connectors*. The latter encapsulate the communication between components, allowing anonymous communication, easy interception of messages and distribution transparency.

Since a component offers a coarse-grained code unit, the composition overview represents an insightful architectural model of the system. As the architecture should reflect the code, it is important to keep it consistent throughout the implementation process. A simple listing of components already offers an easy to update rudimentary view of the architecture, whereas a more insightful annotated model needs better tool support.

2.3 Required tools

A component run-time middleware is necessary to enjoy the container benefits and run-time deployment and adaptation. Also, to create the component architecture, modelling tools are useful. UML2.0 tools already offer support for modeling components, although middleware support is less mature.

2.4 Applicability

Use components if a lot of code reuse is possible: e.g. if you are building a software family or product line. Also, components are beneficial if a project is medium or large in size, or if performance scalability is an issue.

Avoid components when the overhead of putting them together is too great or when projects are small. Also, components (especially in dynamic component systems) introduce an additional resource overhead: choosing the right granularity is important.

3 Agile QoS constraints

Agile QoS Constraints: *Describe QoS adaptation orthogonally and use tools to generate adaptation code.*

3.1 Motivation

Although also present in regular software development, non-functional constraints are much more prominent in embedded systems development. An important non-functional constraint is maintaining Quality-Of-Service (QoS), so that the user experiences a guaranteed minimum level of quality (for example, in response time).

QoS-constraints are usually formulated in requirements documents, or in case of an agile approach, user stories. A test-driven development process mandates creation of tests that verify the Quality-of-Service in a number of simulations. However, a number of problems persist that distinguish QoS handling

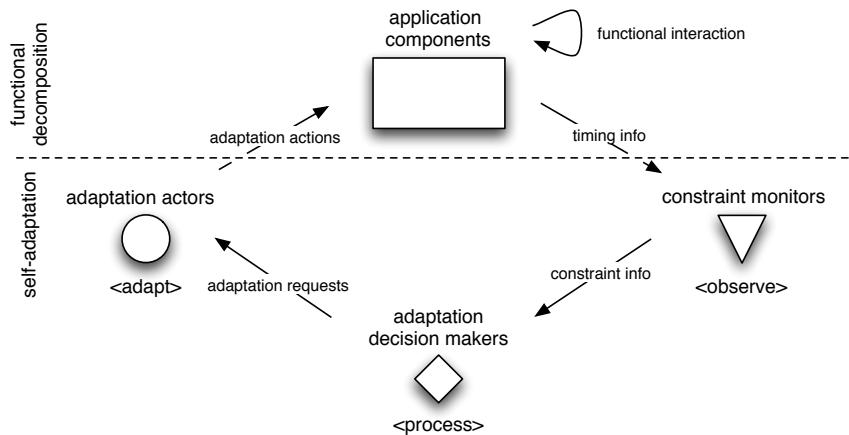


Figure 1: Architectural constructs and the “observe - process - adapt” cycle

from functional requirements. First, if QoS constraints are handled the same as functional requirements, they get ‘lost’ in the code. The lack of documentation in agile processes causes programmers to easily lose sight of orthogonal issues such as non-functional constraints. This tangling will make it increasingly difficult to find structural bottle-necks and manage performance of the application. Second, testing QoS constraints is not always sufficient: embedded software increasingly needs to be run in many changing resource conditions. Here, constraints need to be monitored when the software is deployed and running so as to steer QoS.

Fortunately, QoS constraints can be specified unambiguously in high-level descriptions. That is why we propose an Agile QoS Constraints practice:

make QoS management orthogonal: this way, it can be specified and handled separately

describe QoS constraints at the level of adaptation: this way, constraints and adaptation behavior can be easily expressed and linked

annotate QoS constraints and adaptation behavior in a translatable format: this way, QoS management code can be automatically generated

3.2 Technical overview

In what follows, we describe an example approach to describe QoS constraints in a component based diagram.

Figure 1 shows how this is achieved using special architectural constructs. The Figure introduces three constructs to complement application components:

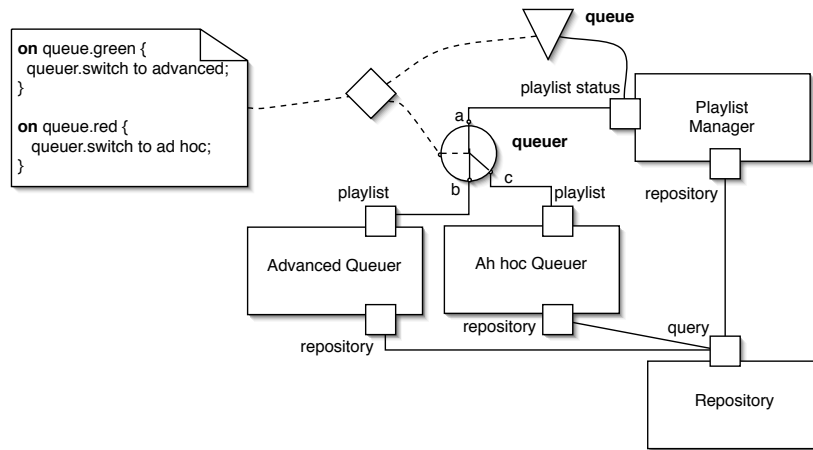


Figure 2: Example QoS adaptation diagram

constraint monitors specifying QoS constraints on message flows between application components and how these constraints must be checked at run-time;

decision makers detailing an adaptation policy to connect monitor evaluations with the appropriate adaptation actions.

adaptation actors specifying how application components and component compositions may be altered at run-time;

We call the architectural view that describes this monitoring-based adaptation the *QoS adaptation view*. This view is based on a component instance diagram but defines a run-time adaptation process with reified versions of the above constructs as follows. Decision makers encapsulate Quality-of-Service (QoS) levels for a component group. Constraint evaluations determine this QoS, but the decision maker may instruct adaptation actors to try to uphold it.

Figure 2 shows an example annotation describing a switch in queue algorithm that is linked to a Playlist timing constraint. When using a component-based middleware, the concepts can all easily be translated to run-time entities, effectively carrying out the QoS management. We refer to [3] for more details.

3.3 Required tools

Some support exists to express QoS constraints in UML tools, but additional tool support is needed to describe QoS adaptations and translate them to code.

3.4 Applicability

Recommended when QoS is an important requirement. Recommended if QoS monitoring and/or adaptation is required.

References

- [1] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.
- [2] C. Szyperski. *Component Software: Beyond Objectoriented Programming*. Addison-Wesley, New York, 1998.
- [3] A. Wils, Y. Berbers, T. Holvoet, and K. D. Vlaminck. Timing driven architectural adaptation. In *Proceedings of the conference on distributed and interoperable systems (DAIS)*, pages 242–255, 2006. accepted at the DAIS 2006 conference.