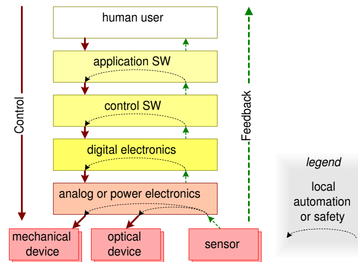
The Role of Software in Systems

-



Gerrit Muller

Buskerud University College

Den Dolech 2 (Laplace Building 0.10) P.O. Box 513, 5600 MB Eindhoven The Netherlands

gerrit.muller@embeddedsystems.nl

Abstract

Software is a dominating factor in the development of complex systems. It plays a crucial role in the performance of the final product at the one hand, while it contributes significant to the development cost and elapsed time of development. This paper will discuss the role of software in the broader system context. An improved understanding of the role of software enables the system architect, and the other stakeholders of the product creation process, to integrate the software development better. In this way hardware-software tradeoffs can be made, balancing performance, costs and risks.

Distribution

This article or presentation is written as part of the Gaudí project. The Gaudí project philosophy is to improve by obtaining frequent feedback. Frequent feedback is pursued by an open creation process. This document is published as intermediate or nearly mature version to get feedback. Further distribution is allowed as long as the document remains complete and unchanged.

All Gaudí documents are available at:
http://www.gaudisite.nl/

version: 1.2 status: concept July 1, 2011

1 Introduction

The relation between the software and system disciplines is difficult in many organizations. The poor relation between the disciplines results in gaps in the design and later in quality problems in the final systems. As a consequence software is in many organizations perceived as a problem and a bottleneck in product creation.

Part of the explanation is traditionally physical disciplines, e.g. mechanical, optical, or electrical engineering, dominated system design. Historically the engineers from these physical disciplines were confronted most with the application domain. These engineers have evolved into domain engineers.

In the modern world software has a significant impact on many system qualities, as we will show in this chapter. More and more customer value depends on software. Unfortunately, many software engineers have not yet build up sufficient knowledge of the physical aspects of their systems or of the application domain. At the same time the engineers from the physical disciplines, who dominate the system design, do not yet understand the jargon and the concepts from the "virtual" disciplines (software, digital electronics engineering).

2 Why is Software a Bottleneck in Product Development?

2.1 Growth of software effort

Software is a relative young discipline. The amount of software in systems is growing exponentially. The contribution of different disciplines to the system, measured in effort is shifting continuously. Figure 1 shows the growth of effort to make software and the related relative decrease of the other disciplines.

2.2 Roles of the disciplines in a system

The different disciplines do have an asymmetric relation when we look at the control in systems. Figure 2 shows a typical control hierarchy in a system. At the bottom we see the physical disciplines who realize physical devices and sensors. We prefer to keep these physical components independent from each other seen from control perspective. Safety provisions are the major exception to this rule.

The physical devices need actuation that is delivered by some analog (power) electronics, e.g. amplifiers. Note that there might be all kinds of conversions in between in the more complex reality, e.g. pressure in a hydraulic system, light in an optics system. Again we prefer to keep the analog electronics mutually independent. The analog electronics is controlled by digital electronics. The control stack continues with control software that sits on top of the digital hardware. Finally, application software determines what the control software should do. Hopefully, the human user is the person who is really in control.

Figure 1: The relative contribution of software effort as function of time

Note that in all layers there are several reasons to have short cuts from sensors to control:

Safety is always kept as simple and direct as possible, since any complexity introduces new safety risks. A good safety design carefully allocates safety functions to the different layers to achieve the desired safety while achieving the desired control flexibility.

Automation can be done on lower layers if this simplifies the overall design. Automation provides value when the higher level work flows are well understood and well defined.

Figure 2: The Control Hierarchy of a system along the Technology dimension

Performance is a special case of automation, where the short cut facilitates better performance, for example fast response times.

The software technology is in most modern systems the integrating technology, as shown by the control hierarchy. In the next section we will dive somewhat deeper in the relation between system qualities and software technology. In modern systems software technology determines to a high degree most system qualities. The inherent system qualities are often determined by the physical design, but the actually achieved quality is often determined by the way the software is constructed. For example, we can dimension a system with quite powerful motors to ensure high performance, but if the software does not fully utilize the motors, then the system performance is lower than can be expected from the physical design. Similarly for reliability that inherently is determined by the physical design. However, the software control may negatively impact reliability. For example, in a system with pumps, the software used a sequence where one of the pumps regularly ran dry. The consequence was that this pump failed often.

2.3 Characterization of disciplines

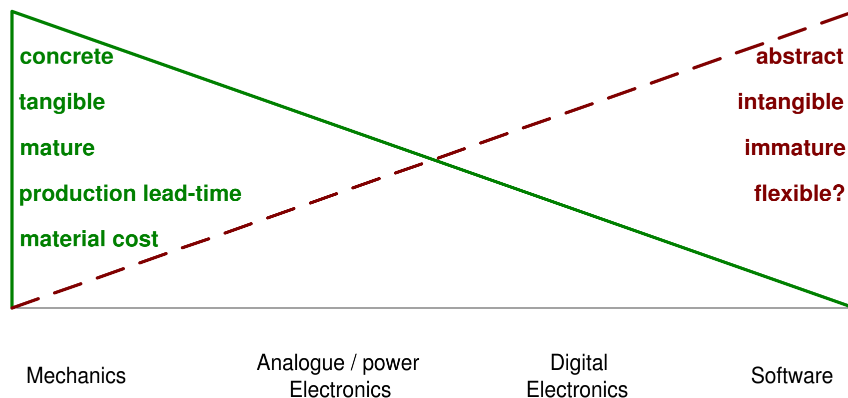


Figure 3: Characterization of disciplines, ordered along the level of abstraction

Physical disciplines work on aspects that can be touched, the subjects are tangible. Virtual disciplines work on abstract concepts, the subjects are intangible. Figure 3 shows the disciplines on an axes of decreasing tangibility and increasing abstractness. Mechanics is one of the older disciplines that is highly tangible. Analog (power) electronics is younger as discipline and less tangible. Digital electronics is again younger. Although the digital electronics itself can be touched, the circuitry itself is much more conceptual and abstract.

Figure 3 also provides a number of other characterizations that follow the same trend as tangibility and abstractness:

maturity The more tangible the more mature a discipline seems to be. Mature means here well known and founded; the discipline has an established and documented body of knowledge.

production lead time The physical world is constrained by nature. Processing and production of components have an inherent lead time. Software can be seen as infinitely fast. However, when testing, quality control and configuration management are included in the production lead time, then this lead time becomes strongly dependent on people, processes, and tools. Hence the question mark behind flexible at the right hand side of the figure.

material cost Physical systems do have inherent cost in the materials and its processing.

These differences in nature, especially *production lead time* and *material cost*, cause also differences in other business processes and the approach to life cycle aspects. For many physical components the logistics design is crucial for cycle time, stocks, and cost, where software does have zero reproduction cycle time, cost and infinite stocks.

3 System or Software Issues?

Systems can be specified in terms of their functionality and qualities. Most qualities of a system are strongly influenced or even determined by the software design. Figure 4 based on [4] shows a checklist for qualities. In this figure all qualities that have a strong or weak relation with the software design are highlighted.

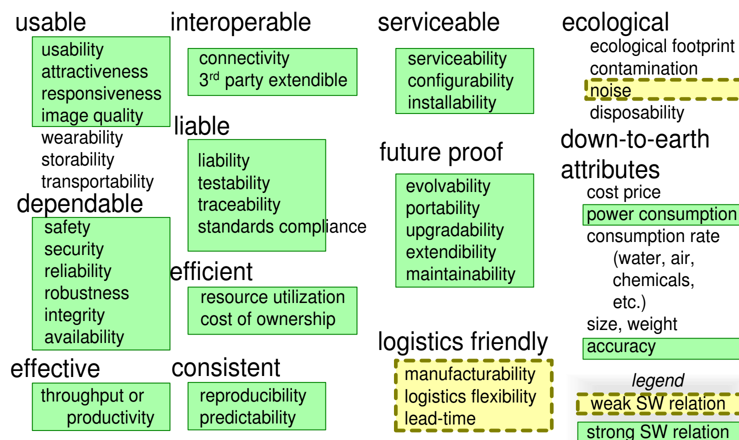


Figure 4: Quality Checklist annotated with the relation with software

During System Design the system is decomposed in subsystems and implementation technologies. The combination of subsystems and technologies together has

to realize the qualities. During this step the contribution or the role of a subsystem and implementing technology is determined.

Figure 5: System design aspects that are strongly SW related

Figure 5 shows the System level design aspects that are strongly related to software. Figure 6 shows a list of mechanisms used by SW engineers. These mechanisms facilitate the system level design aspects mentioned in Figure 5.

Both *Quality Attributes* and *Design Aspects* are *System Level* issues, however most of these issues are predominantly influenced by the software. The System Architect should: define the system level **what**, co-design the system level **how** and be involved with the single technology or subsystem **how**.

Due to the strong Software impact the software architect should: understand/review the system level **what**, co-design the system level **how** and design the software **how**.

This requires significant domain know-how of the Software Architect, see [2].

Figures 5 and 6 contain too many design aspects and software mechanisms to discuss as part of this book. The main purpose of these lists is to show the variety of technology issues to be addressed by the software architect.

Many of the design aspects have a many to many relation to the software mechanisms. For example, the design strategies for *performance*, *safety*, and *security* relate to nearly all software mechanisms. Vice versa most software mechanisms penetrate throughout most software and relate back to most of the design aspects.

The software part of systems is complex in itself. The software is a construct

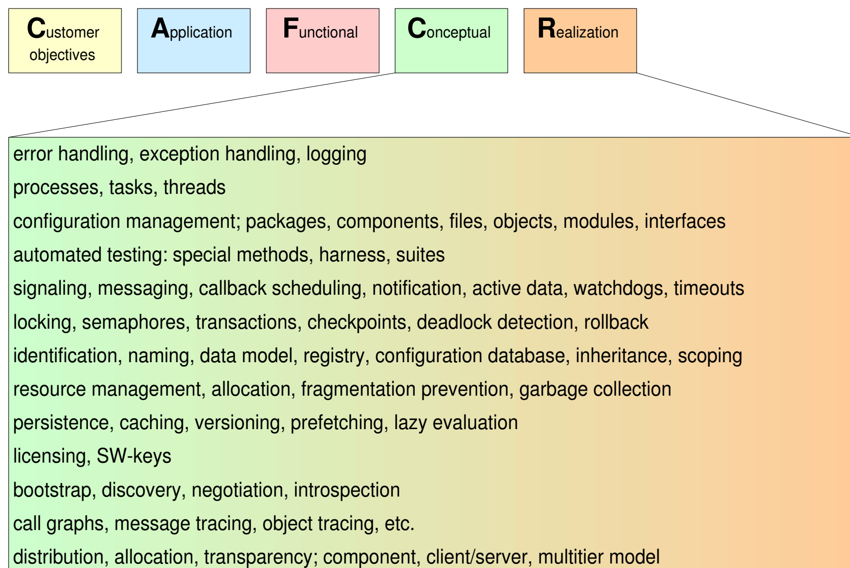


Figure 6: List of Software Mechanisms that are frequently applied to solve the system level design aspects

made by many people, stacking construct on construct. The risk is that software architects spend all their time internally in the software, while they also have to relate the software choices to the context, the system.

4 Acknowledgments

Jürgen Müller helped to sort out the attributes, aspects, mechanisms et cetera, which helps to position the Software Discipline in the System Development.

References

[1] IFIP. *Software Architecture; TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, 1999.

[2] Philip Kruchten. The software architect- and the software architecture team. In *Software Architecture; TC2 First Working IFIP Conference on Software Architecture (WICSA1)* [1], pages 565–583. This article describes required skills for architect and architecture team; traps and pitfalls; Personality profile based on Myers-Briggs Type Indicator.

[3] Gerrit Muller. The system architecture homepage. `http://www.gaudisite.nl/index.html`, 1999.

[4] Gerrit Muller. CAFCR: A multi-view method for embedded systems architecting: Balancing genericity and specificity. http://www.gaudisite.nl/ThesisBook.pdf, 2004.

History

Version: 1.2, date: August 9, 2010 changed by: Gerrit Muller

- replaced the text cetralized versus distributed by brief text about aspects and mechanisms

Version: 1.1, date: August 6, 2010 changed by: Gerrit Muller

- removed "complex" from title
- textual improvements
- changed status to concept

Version: 1.0, date: June 9, 2010 changed by: Gerrit Muller

- replaced lists by figures
- added introduction
- added text
- changed status into draft

Version: 0.3, date: August 5, 2002 changed by: Gerrit Muller

- minor changes

Version: 0.1, date: September 21, 2001 changed by: Gerrit Muller

- abstract and logo added

Version: 0, date: July 25, 2000 changed by: Gerrit Muller

- Created, no changelog yet