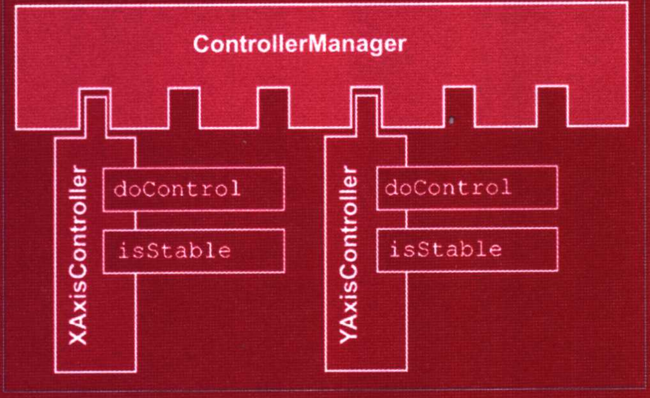
Alessandro Pasetti

LNCS 2231

Software Frameworks and Embedded Control Systems



Springer

Alessandro Pasetti

Software Frameworks and Embedded Control Systems

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Author

Alessandro Pasetti
ETH-Zentrum, Institut für Automatik
Physikstraße 3, 8092 Zürich, Switzerland
E-mail: pasetti@pnp-software.com

Dissertation der Universität Konstanz
Tag der mündlichen Prüfung: 8. Juni 2001

Referent: Prof. Dr. Wolfgang Pree
Referent: Prof. Dr. Kai Koskimies

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Pasetti, Alessandro:
Software frameworks and embedded control systems / Alessandro Pasetti. -
Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ;
Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2231)
Zugl.: Konstanz, Univ., Diss., 2001
ISBN 3-540-43189-6

CR Subject Classification (1998): C.3, D.2.11, D.2.13, J.2

ISSN 0302-9743
ISBN 3-540-43189-6 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10845761 06/3142 5 4 3 2 1 0

Preface

Professionally, the defining experience of my life was a period of nine years as control systems engineer with the European Space Agency (ESA). Given this background, it was perhaps inevitable that when, in 1999, I decided to start a new career as a software engineer I should choose as my area of work software architectures for embedded control systems. As it turned out, this was a lucky choice. After decades of academic neglect, embedded software is now beginning to receive the attention to which its ubiquity in everyday devices and its presence at the heart of many safety-critical systems entitles it. The novelty of the field means that much of the work that needs to be done consists in transferring to embedded systems the technologies and development practices that have become so prevalent in other fields. Following this line of research, I have concentrated on applying object-oriented software frameworks to embedded control systems. Framework technology has enjoyed wide currency for at least five years. Although it has proven its worth as a software reuse technique in many domains, it has been shunned in the embedded world mainly because it tends to be associated with lavish use of CPU and memory, both of which have traditionally been in short supply in embedded systems. Times are changing, though. Hardware advances are expanding the resources available to embedded systems and their adoption of framework technology no longer looks far-fetched or unrealistic.

The first objective of this book is to show how object-oriented software frameworks can be applied to embedded control systems. Software design may not be an art but it is certainly more of a craft than a science and teaching by example is in my view the best way to transfer design experience. I have accordingly chosen a case study as the means to make my point that framework technology can aid the development of embedded control software. The target application of the case study is the attitude and orbit control system of satellites. This domain is broader and less structured than the domains to which frameworks are normally applied. This led me to develop a concept of frameworks that was rather different from that proposed by other authors in that it gives a more prominent, or at least a more explicit, role to domain-specific design patterns. The second objective of the book is to discuss this new view of software frameworks and to present some methodological concepts that I believe can facilitate their development.

This book is therefore written with two audiences in mind: developers of embedded control software will hopefully be inspired by the case study in the second part of the book to apply framework technology to their systems, while researchers on software architectures might be moved to rethink their conceptualization of frameworks by the material presented in first part of the book. The link between first and second part lies in the use of the concepts introduced in the methodological sections of the book to describe the case study framework.

One of the functions of a preface is to offer a path through the book to prospective readers. The key chapters are 3 and 4, introducing the concepts of framelet and implementation case, and chapter 8, giving an overview of the design principles behind the satellite framework. Hurried readers could limit themselves to these three fairly self-contained chapters, which would suffice to give them a rough idea of both the methodological and technological contributions made by the book. The framework concept and the methodological guidelines for framework design are presented in chapters 3 to 7. The case study is covered from chapter 8 to the end of the book. Readers who are not familiar with satellite control systems should read chapter 2 in order to acquire the domain background necessary for an understanding of the case study.

The satellite framework is presented as a set of design patterns that have been specifically tailored to the needs of satellite control systems. Each chapter in the case study part of the book covers one or a small number of related design patterns. These chapters are to a large extent independent of each other and could be read in any order or even in isolation of each other. One way to see this book (or at least its second part) is as a repository of design patterns for embedded control software development.

The appendix at the end of the book contains synoptic tables listing the architectural constructs offered by the satellite framework. The lists are cross-referenced and can serve as an aid to navigate the framework design presented in the book.

This book describes the satellite framework at the "design pattern level". The satellite framework also exists as a publicly available prototype and its full design documentation and code can be downloaded from the project web site[1]. Readers are invited to access this material but they should bear in mind that the code is offered without any guarantee of correctness and that its quality and completeness are those required for a proof-of-concept prototype. They should also be aware that the satellite framework project is a "living project" and its web site is constantly being updated as the framework design and implementation are reviewed and modified. Hence, as time progresses, inconsistencies will inevitably arise with the content of this book.

Having explained what the book offers to its readers, I should write a few words about what it expects of them. The focus is on *object-oriented* frameworks but no effort is made to explain the principles of object-oriented design which are assumed known to the readers. Similarly, familiarity with the design pattern concept is both assumed and necessary. Some background knowledge of framework technology would be useful but is not essential. No prior knowledge of satellite control systems is assumed, all the necessary background being offered in chapter 2. I have made every effort to make the case study as self-contained as possible

[1] Its address is subject to change. The site can be found with any internet search engine by searching for "AOCS Framework".

but I suspect that full appreciation of the solutions it offers requires at least a passing acquaintance with embedded control systems. Class diagrams are in standard UML and pseudo-code examples are written using C++ syntax with the addition of the interface construct from Java which I find simply too useful to ignore.

Finally, I wish to thank Wolfgang Pree and Kai Koskimies who reviewed the first draft of this book and Jean-Loup Terraillon, Ton van Overbeek, Richard Creasey, and David Soo who, in various capacities, supported the satellite framework project.

August 2001 Alessandro Pasetti

Contents

1 Introduction and Context

There are so many books in print that the first duty of an author must be to explain why a new one is needed. The present book is about the application of software framework technology to embedded control systems and its core is the description of a case study of a full object-oriented framework for a particular – but representative – class of embedded control systems. Its justification therefore requires arguing the case in favour of the distinctiveness of these systems. This is done in the next section with reference to embedded systems in general since their claims to distinctiveness encompass and subsume those of the narrower category of control systems. The following sections instead outline the long-term programme of research within which the results presented here were obtained. The material in the book only covers a small part of this programme (which, at the time of writing, is still on-going) but understanding the latter helps understand the motivation and intention behind the former. The chapter closes with a formulation of the objectives of the book and a summary of its contributions.

1.1 The Embedded Software Problem

It is probably true that, as many believe, nothing of any significance has been invented in computer science since the late sixties. Possibly in order to compensate for this dearth of new ideas, software engineers have proven exceptionally adept at introducing new words for old concepts and at attaching new meanings to old words. Whether for this or for other reasons, their discipline is plagued with terminological confusion. It is therefore fitting that this book should begin with a definition (with many more to follow in later chapters). In accordance with common usage, the expression *embedded system* is used to designate a computer system hidden inside a product other than a computer [95]. This definition can be turned inside out to say that an embedded system is a computer system that is used to control the product that contains it. Embedded systems are therefore the means by which the power of software is harnessed to make the behaviour of non-computer devices – from washing machines to satellites – more flexible and more controllable. The *embedded software* is the software that runs on an embedded computer and that is the ultimate source of this flexibility and controllability.

In what sense is embedded software different from other types of software and why does it deserve dedicated treatment? Answering this kind of question inevitably exposes an author to the twin dangers of overgeneralization and oversimplification. These dangers are especially acute in the present case because of the exceptional diversity of embedded systems. Still, even where there is diversity there can be average trends and general patterns and it is to them that the discussion that follows applies.

At first sight, the differences between embedded and non-embedded software have two aspects, functional and technological. Functionally and in a negative sense – looking at what embedded systems are *not* or do *not* have – embedded software is characterized by the complete lack, or at least sharply reduced importance, of the user interface. This already sets it apart from other software since the user interface in conventional systems accounts on the average for 50% of the total development and coding efforts and, when a graphical user interface is used, this proportion is even higher and can reach 90% [28].

In a positive sense – looking at functional features that embedded systems have and other systems lack – embedded software is primarily distinguished by its direct interaction with hardware peripherals. In the case of embedded control systems, for instance, the software always has to manage a set of sensors from which measurements about the state of the variables under control are acquired and a set of actuators to which command signals are periodically sent. The need to respond to and control external hardware and external processes often results in timing requirements being imposed on embedded software. Embedded systems must then be built to ensure certain minimal response times or to guarantee a certain minimal throughput. Meeting such requirements can have a decisive impact on the architecture of the software and timing aspects are invariably among the design drivers of embedded software. Additionally, embedded systems are by their very nature designed to operate outside direct human supervision. Indeed, their task is often to replace or complement human supervision of a certain process or device. Stand-alone operation requires a high degree of autonomy and reliability to be built into embedded software.

Functional differences, though important, are not those that most stand out in a comparison between embedded and non-embedded software. The truly remarkable difference between the two is technological and relates to the vast gap that can be observed between the level of technology that is now common in general purpose computer systems and the level of technology prevalent in embedded applications. When seen from the point of view of mainstream computer science, embedded software projects often appear to be taking place in a time warp: the language of choice remains C (with generous sprinklings of assembler), the dominant architectural paradigm is only now shifting from procedural to object-based, software engineering tools are often ignored.

Technological backwardness is the norm even in fields that are perceived as "high-tech". In 1996, the author of this book was asked to take part in the review of a control system for a large satellite being developed by a major aerospace company for the European Space Agency and was astonished to find that its software was entirely written in assembler (using a modular approach – as one of the project engineers proudly declared). For another example, we can turn to the car industry. As recently as 1997, the engine control software of a very popular line of middle-sized cars consisted of about 300.000 lines of C code that was poorly structured because of its assembler heritage, and that, after years of changes to fulfill ever changing requirements, had become virtually unmaintainable [69].

Technological backwardness can be confirmed by inspection of some of the recent reference books on software for embedded systems such as [95, 46, 59, 7]. Although all four texts mention C++ or object-orientation (sometimes even in their titles) and give due emphasis to the virtues of the latter, the paradigm that underlies most of the material they present is the traditional object-based or modular one. The same impression can be derived from the "Embedded Systems Magazine", a prime reference for embedded engineers. Advanced software topics – object-orientation, component technology, real-time Java and others – are sometimes broached but their adoption in practice remains controversial and is clearly limited to a small minority of existing systems (for a typical example of this attitude, see the cover article of the issue of August 1999 with the eloquent title of: "Nuts to OOP!").

It might be expected that research would tend to redress the imbalance between embedded and other software. In reality this is far from being the case. Discipline folklore has it that while 99% of processors that are sold in a given year are used in embedded devices, only 1% of computer scientists are working on embedded software. These figures are difficult to verify but there is no disputing that the paradox they point to – the overwhelming majority of software research being focused on a narrow subset of all computer systems – is real.

Lack of interest on the part of software scientists can be largely explained by the limited resources available to embedded computer systems which typically have memories restricted to kilobytes and processor architectures of the CISC, 8- or 16-bit kind. Application memory and CPU margins are correspondingly tight and seldom leave any scope for applying the advanced software technologies that are the staple of academic publications and that are so widespread in desktop applications.

To some extent, this situation is self-perpetuating. Lack of research in embedded software leads to lack of innovation and to software systems that are comparatively small and simple. Since embedded devices are often mass-produced, profit margins are largely determined by hardware costs and manufacturers have an incentive to continue using the minimalist processor configurations that have stifled research and innovation in the first place. Perhaps more subtly, lack of interest in embedded software on the part of the research establishment has turned the subject into one of computer science's poor relations which hardly encourages software scientists to enter the field and attempt to redress the imbalances described above. The web of causal interrelationships between technical, economic and socio-cultural factors that contribute to keep embedded and other software apart is sketched in figure 1.1.

Recent developments, however, may herald a reversal of fortune for embedded software and its practitioners. On the one hand, the continuing decline in hardware costs is bringing more and more processing power within the budgetary envelope of embedded software projects while, on the other hand, ever expanding consumer expectations are putting increasing demands on the software, demands that can only be accommodated by resorting to the same kind of technology prevalent in the non-embedded world.

If disparity in the level of hardware resources were the only – or the main – factor differentiating embedded from other software, then there should be no reason to treat embedded software as "special": as the resources available to embedded developers are brought in line with those available to their desktop colleagues, one should see embedded and non-embedded software naturally converge towards the same technological level and towards the same design and implementation practices. This is unlikely to happen and this is a symptom of a more profound, albeit often overlooked, difference between much embedded and much non-embedded software.

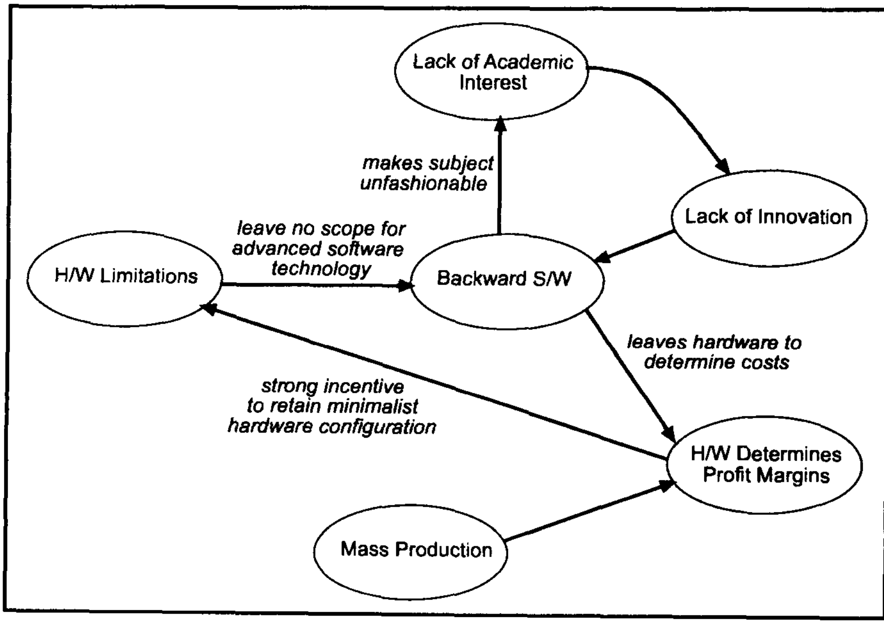


Fig. 1.1. Why embedded software lags behind other software

Software design is always a difficult undertaking but it is especially so when the persons doing the design have a poor understanding of the target application. This problem occurs everywhere but it is more severe in the embedded world. After all, the definition of embedded software given at the beginning can be paraphrased to read: "an embedded software is a piece software for a system other than a computer". The software for an embedded system consequently has a somewhat accidental character, it serves to embody functions that find their origin and motivation in the non-software aspects of the system. Software development for embedded applications is thus often characterized by the divorce between the roles of the software specialist – the person responsible for the software development – and the application specialist – the person responsible for specifying the application. The gap between the two is normally bridged by some kind of formal description of the application to be developed such as user requirements or use cases. The difficulty of exhaustively and unambiguously capturing the specification of a complex system means that, almost inevitably, as a project progresses, the interface between the software and application specialists becomes a source of personal attrition, schedule delays, cost overruns and specification misunderstandings. This problem is all the more serious because more and more of the functionalities of an embedded system are concentrated in its software and the split of work between the software and domain specialist therefore results in a situation where the heart of the application – its software – is designed, developed and tested by persons who are *not* application specialists.

It is worth to make this point by means of an example. First, consider the case of a software engineer who is asked to develop the software for, say, a new word processing application. As an input, he would probably be given a set of user requirements. Since he is familiar with word processors (because he has used them, because he has studied them at university, because he has read about them, etc) it is very likely that he would quickly understand them and would be able to assess their quality by, for instance, spotting mistakes, gaps, and ambiguities. Additionally, having a good "feeling" for word processing applications, he would be able to judge whether the requirements he has received are implementable and would quickly identify those that are critical and need special attention.

Suppose now that the same software engineer were asked to write the software for, say, a GPS receiver (a typical embedded application). GPS receivers are likely to be outside his typical domain of experience and knowledge and he would have to rely completely on the formal description of the GPS software that he receives from his customer. Firstly, he might have some trouble understanding it because he is not familiar with the vocabulary of that domain (whereas he was perfectly at home with the domain vocabulary of word processors). Secondly, he would not be in a very good position to assess the quality of the inputs he receives and to identify potential problems due to inconsistencies and misunderstandings. These problems would eventually emerge but they would emerge only during application implementation or, even worse, during testing.

It is obvious that, other things being equal, our software engineer would make a much worse job in the second than in the first case and that the cause of this is the fact that, in the second case, he lacks domain understanding. Our contention is that the scenario described in the second example typifies embedded systems in general and embedded control systems in particular.

Finally, and for completeness, it must be mentioned that in practice a common means for the application engineers to by-pass their software colleagues and avoid the problems and costs that arise at the interface between the two disciplines is for them to write their own code directly. This, however, is not a viable solution in any but the simplest cases: software development requires specialized skills that normally only specialists possess. The solution to the difficulties discussed above lies in optimizing the role of each of the two specialists in the application development process, not in ignoring the need for their cooperation.