

# On The Real Value Of New Paradigms

Theodor Tempelmeier

Department of Computer Science  
Laboratory of Real-Time Systems  
Rosenheim University of Applied Sciences  
Hochschulstr. 1  
D-83024 Rosenheim  
tempelmeier@fh-rosenheim.de

Abstract. This is a critical assessment of some of the new paradigms of software engineering. The Unified Modeling Language, the notion of design patterns, and some ideas for future and more advanced modelling elements are investigated. This is done from a practical and theoretical point of view, with a focus on real-time and embedded systems development.<sup>1</sup>

## 1. Introduction

In computer science, new paradigms arise almost continually. In contrast to the real scientific breakthroughs these new paradigms are usually advertised with lots of euphoria, making developers uncritical about possible drawbacks and problems. Even if some new paradigms do not have intrinsic problems, they may suffer from a naive transfer to the domain of real-time and embedded systems development. Instead of such a naive transfer, the specific requirements of real-time and embedded systems development may necessitate a deviation from the original proposals. With this, the Unified Modeling Language, the notion of design patterns, and some ideas for future modelling elements are critically assessed in the following. Of course, this assessment is done with a focus on real-time and embedded systems development only. The author admits that former contact with safety-critical software may have resulted in a bias towards rigour, preciseness, and strong standards in doing real-time and embedded systems development.

It must be pointed out that in the view of the author object-oriented design is definitely the right way of building systems, and this also applies to embedded systems. Encapsulation, abstract data types, etc., and other concepts such as generics or templates, are a must when developing complex embedded systems. This is in fact also the way the more progressive companies do already develop embedded systems. The author is re-

---

<sup>1</sup> This contribution is an extended compilation of the author's presentations at the OMER and OMER 2 workshops [Te99],[Te01],[Br01]. The author would like to thank the anonymous reviewers of the workshops and the final proceedings for their invaluable and detailed remarks.

luctant, though, to accept inheritance as a dominating design principle. But at least an object-based approach, i.e. encapsulation without inheritance, is the right choice in the author's opinion, without any doubt. Hence, this contribution must not be misunderstood as a criticism of object-orientation in itself.

## 2. UML for Embedded Systems Development

### 2.1 Modelling with UML

The Unified Modeling Language (UML) has become the dominating modelling language in software development. UML has its merits, no doubt, in having unified the variety of slightly different but in essence similar notations. For embedded and real-time systems, the benefits of using UML are less clear [Se00]. The author feels that the UML has not been specified with too much concern for embedded and real-time systems development. Pragmatic approaches as in [MM98] underline the specific needs in this area. Generally, thinking about the application of the UML to embedded systems development actually involves questions on different issues:

1. Can something be modelled in UML in principle?
2. Can something be modelled in base UML (i.e. UML without extensions)?
3. Can something be modelled in UML in a "better" way than with previous notations, seen from a pragmatic point of view, i.e. considering only *secondary virtues* such as supply of tools, availability of trained engineers, etc.?
4. Can something be modelled in UML in a better way than with previous notations, seen from a fundamental point of view, i.e. solely with respect to the *concepts* in UML?

The answer to the first three questions is probably a plain "yes". If something can be modelled with certain concepts (say in the framework of ROOM [Se94]) then it can be modelled in UML by defining extensions (stereotypes) which exactly exhibit the same behaviour as the original concepts. UML just serves as an implementation vehicle, in this case. Concerning the second question, one may well assume that anything can be modelled with UML, given the plethora (and vagueness) of concepts in UML. And, of course, availability of tools and the like will profit from the unification process. This contribution only deals with the fourth question, both from a practical and theoretic point of view.

### 2.2 Software Requirements Specification

In a research and technology project of DaimlerChrysler Aerospace (now EADS) an integrated process for the development of control laws for complex aircraft configurations has been investigated. The flight control software from this project [Ro99b] will be used as an example to evaluate the UML for the requirements specification phase. In a small case study the UML (Version 1.0) has been investigated with respect to speci-

fying requirements for typical flight control software. The results are presented here in the form of three examples:

- definition of control surfaces
- control law block diagrams
- definition of external interfaces

*Definition of Control Surfaces.*

The main outputs of a flight control system are commands to the control surface actuators. Therefore, a requirements specification for flight control software would typically include a definition of these control surfaces. A possible UML definition for this is given in figure 1. A control surface “is a” primary or a secondary surface (inheritance relation) and so on. And a delta-canard aircraft “has” one rudder, four flaperons, and so on (aggregation or composition relation).

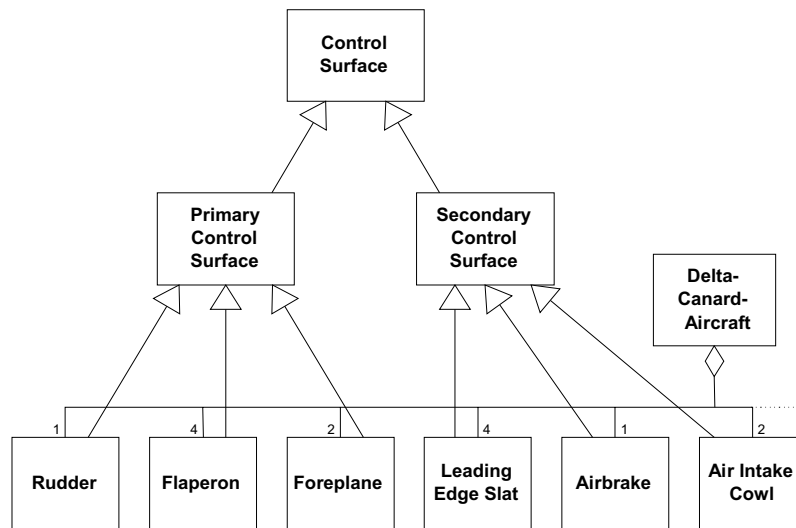


Fig. 1: Control Surfaces of an Aircraft in Delta-Canard-Configuration (UML Diagram)

Figure 2 shows as an alternative a sketch of an aircraft with the primary control surfaces emphasised in white and the secondary ones in dark. It can be seen that a simple figure is sufficient to convey at least the same information as the UML diagram. In fact, a much deeper conception of the physical situation is achieved by figure 2. The author would not consider it very helpful to rephrase such parts of the requirements in UML. (The situation might be different, if the system under investigation would not contain physically visible elements.)

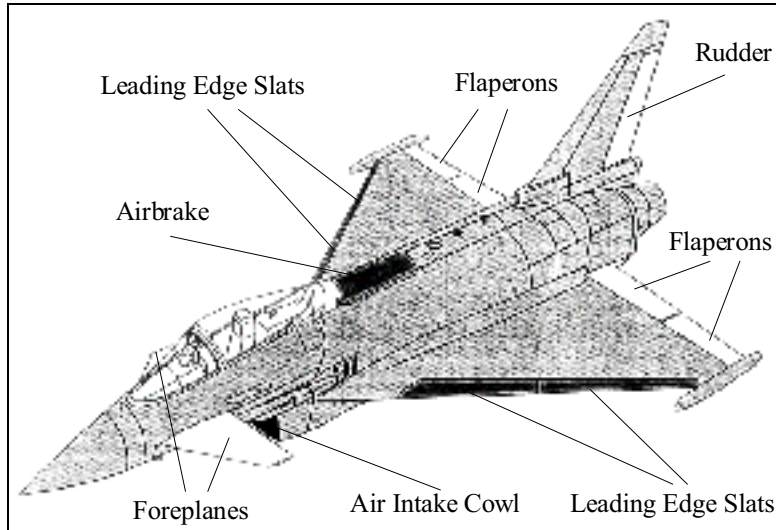


Fig. 2: Control Surfaces of an Aircraft in Delta-Canard-Configuration

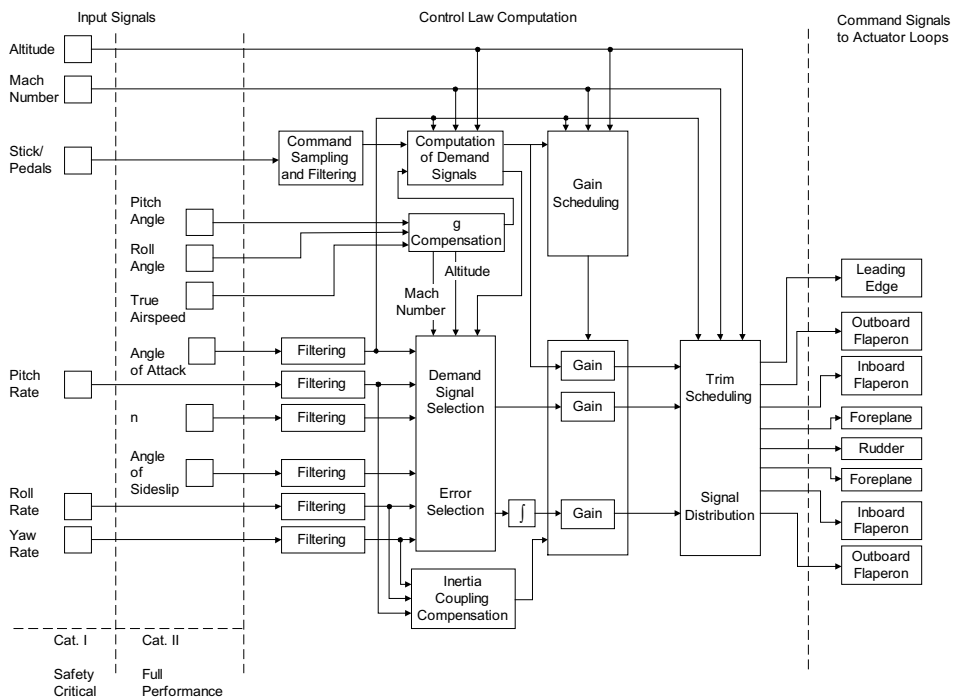


Fig. 3: A block diagram of flight control laws according to [Ka92]

### *Control Law Block Diagrams.*

Requirements for flight control law software are usually defined using control block diagrams, which essentially describe blocks and data flows between them. Each block represents a transformation of input data to output data. The detailed control law computations are described in an algorithmic language, e.g. in FORTRAN. It is essential that this specification is executable, in order to validate the control law design as far as possible before trying to put the real system under test. Obviously, the design of the control laws will go through some iterations, where new ideas and new parameters are tried out and validated (or rejected) until a stable version can be used as requirements for the actual flight control law software. The control block diagram serves as an invaluable aid for maintaining the overview over the control system as a whole. Figure 3 shows an example of such a diagram. If one tries to rephrase such diagrams in UML, the following difficulties arise.

- Obviously, the control block diagram is not a class diagram. Instead, the blocks constitute multiple instances of classes, e.g. the “Filtering” objects. The control block diagram thus rather is an “object diagram”. While it is possible to draw object diagrams in UML, only the association relation can meaningfully be used in this case. However, the association relation is only a line drawn between rectangles with almost no semantic information. UML’s object diagrams are thus not a suitable alternative for control block diagrams.
- Class diagrams do not help very much in this case, either. They would reveal surprisingly little information, for instance that a notch filter “is a” filter, etc. The complexity of the control block diagram lies in the functionality, i.e. in the contents of the block diagram elements and in their interdependencies. In contrast, UML seems to be more suitable for applications where the complexity lies in the class relationships, like in database applications, for instance.
- One could try to (mis)use other diagrams of UML, e.g. collaboration diagrams, sequence diagrams, or activity diagrams, but the author does not see any advantage in doing so, as compared to using well established control block diagrams.<sup>2</sup>
- Finally, use case diagrams could be used. It is still unclear to the author, whether use cases are just a recurrence of the once condemned functional decomposition models such as Structured Analysis, or whether they also contain some fundamental new ideas.

---

<sup>2</sup> It is often argued that UML collaboration diagrams could be used instead of control block diagrams or data flow diagrams. However, this is exactly what is meant by the term misuse above. In the UML specification [OM99] it is stated for collaboration diagrams that “... it is important to see only those objects and their interaction involved in accomplishing a purpose or a related set of purposes, *projected from the larger system* of which they are part for other purposes. A Collaboration defines a set of participants and relationships that are meaningful for a given set of purposes. The identification of participants and their relationships *does not have global meaning* [emphases by the author].“ In the view of the author this means that collaboration are to be used to describe different scenarios, not the whole system. Furthermore, the “message” arrows in UML collaboration diagrams clearly involve control flow (specifying the sender and receiver of the message). In contrast, data flow diagrams are somehow more abstract, just showing the flow of data and leaving open how this is accomplished (pushing or pulling the data, or even some other mechanism such as using global data). Pragmatically, if the UML is used, one should reinterpret its definition as it is done in [Go00], where “consolidated collaboration diagrams” are used and where “the information passed between objects” is shown to exhibit something similar to data flow diagrams.

*Definition of External Interfaces.*

Requirements for flight control software must include a definition of the external interface of the software, including the formats of the input and output values. The author considers Ada a good choice for specifying these formats and advocates its use starting with the requirements definition. The following examples repeat some of these probably well-known features of Ada.

```
type switch_values is (neutral, on, off);  
  for switch_values use (neutral => 1, on => 2, off => 4);  
  
C_Small_180 : constant := 180.0 * 2-11 ;  
type T_Fixed_180 is delta C_Small_180 range -180.0..180.0 ;  
  for T_Fixed_180'small use C_Small_180 ;  
  for T_Fixed_180'size use 12 ;
```

Such Ada definitions, firstly, have precise semantics according to the language definition. Secondly, the use of the representation clauses (“for ...”) allows a precise format definition down to the bit level. For instance, the first definition from above is an enumeration type, the logical values of which are tied to their bit representations in say a digital input or output register. The second example defines a fixed point type. The type covers a range from -180.0 to 180.0 with an increment of the constant C\_Small\_180. This results in a precision of 11 bits plus the sign bit. The for-clauses ensure that the compiler does not do better than requested, e.g. by using smaller increments and more bits than specified with the delta. This example could represent the format of a 12-bit analog to digital converter, for instance. Note the use of a delta which is not a power of two. This represents a common situation in practice. Other possible representation issues, e.g. arrangement of such 12-bit objects within 16-bit words or alignment with respect to memory block boundaries, are not shown here.

Most importantly, using such definitions in the software requirements specification automatically guarantees consistency between (this part of) the requirements with the final code, because a suitable and validated compiler has to implement the representation directives exactly as prescribed by the ISO standard of the Ada language. The author feels this method to be superior to rephrasing such requirements in some formal notation, and then perform proofs of consistency with the final Ada code. On the other hand, using UML would almost certainly be inferior, because there is no semantics and not even a syntax for describing types in UML.

The conclusion from this is as follows. Appropriate models for specification have been used in the engineering domain for quite some time (perhaps in contrast to the domain of business and administrative applications). It can not easily be seen how the UML would provide any fundamental improvement as compared to the current modelling approaches.

### 2.3 Software Design

The role of a modelling language in the design phase may be manifold. The two most important roles are

- the modelling language is used as design language down to coding, i.e. the modelling language is also used for “programming”
- the modelling language is used for design and the programming language is used for coding

The problem with the first approach is that most modelling languages do not have semantics as precise as it is necessary for programming. The problem with the second approach is that there may be a semantic discrepancy between the modelling language and the programming language (the programming language is in ultimate authority).

From the experience of applying object-based designs to embedded systems and from experience with various modelling languages and tools (e.g. HOOD, [Te98, Te94]), the author takes the following position.

- “Programming” in the modelling language is neither practicable nor reasonable nor desirable.
- The modelling language can be and should be used for a “visualisation” of the design. *This also implies that the modelling language resembles the design concepts within the programming language on a one-to-one, or “isomorphic” basis.* (Naturally, this is only valid if the programming language includes sound design concepts, e.g. programming languages such as assembler are not considered here. Note also that this is not about visualising the implementation, but about visualising (only) those elements in a programming language, which also resemble modelling concepts, e.g. objects, classes, packages, tasks or active objects and the like.)

Obviously, the UML fits the concepts of mainstream languages such as C++ or Java nicely. The author could accept UML for representing designs targeted to these languages, though not all concepts of embedded systems development (outside C++ and Java) can perhaps be described adequately. As an example, quite a few authors resort to symbols for concurrency which are outside the UML [Do98, Hr02, MM98].

If other target languages are used, e.g. Ada for safety-critical systems, the situation is different. Ada has sound design concepts (which are sometimes superior to C++ or Java concepts) which seem to be without counterpart in UML. This gives rise to the weird situation of *design modelling inversion*, where the language for modelling a design is less powerful (in terms of design concepts) than the programming language. The following three examples are given.

- *Protected objects.* Ada’s protected objects are not supported by UML. The keyword “protected” has a totally different meaning to the C++ or to Ada95 programmer. What would be the meaning of a keyword “protected” in UML?
- *Hierarchical libraries.* Hierarchical libraries can be realised by nesting, which causes some problems concerning recompilation, etc. Ada95 has a much smarter

scheme for hierarchical libraries which avoids such problems. How are hierarchical libraries handled in UML? Even if semantics in UML were simply tied to the library model of Java, one would still face the question of how to handle the generic hierarchical library units of Ada95, as there are no generics in Java. Further, it would still be open how to distinguish nested hierarchies from “smart” (i.e. only conceptually nested) hierarchies.

- *Abstract data types (in the meaning of a class without inheritance) as distinct from classes (with inheritance).* Ada95 offers both classes without and with inheritance (keyword “tagged”). There are good reasons for this distinction, for instance that one sometimes wants to avoid certain aspects of inheritance in safety-critical systems (see [Is96]), while there is no reason to refrain from using abstract data types. Suggestions to use the Java concept of “final” classes for the purpose of this distinction may be acceptable, but this seems not to be a concept within base UML, version 1.3.

It is possible, of course, to enrich UML with special notes and comments to reflect some additional concepts. Additionally, tools may employ a variety of switches to steer code generation in the desired way. But, this is exactly a repetition of the well-known situation with earlier notations and tools: The designer has in mind an exact idea of his or her design (in terms of the concepts of the target language, say, Ada). He or she then has to understand the design notation (UML), its semantics (maybe defined in terms of another language, say, C++ or Java), and the effects of the code generation switches of the tool (for instance about 50 switches for generating C++ classes in one of the leading UML tools). And then, maybe, the designer will get the design (tailored to his or her target language) he or she had exactly in mind from the very beginning. Such a procedure is hardly acceptable in practice.

As a conclusion, a one-to-one (or “isomorphic”) correspondence of design concepts in the modelling language and in the programming language is required (if the programming language itself includes a set of adequate design concepts as it is the case with Ada). This requirement does not seem to be fulfilled by the combination “Ada and UML”.

A final warning seems appropriate on the often heard presumption that a design is always to be independent of the programming language and the target operating system (called target environment for short). True, on a coarse level, the central ideas of a design can be independent of the target environment. This is often referred to as logical or abstract design. However, as the design evolves, more precision is added, and the final design will often be dependent of the target environment. Clearly, this does not refer to syntactic details, but to a dependency from the *concepts* in the target environment.

Consider task interaction as an example. The concepts in the target environment may range from simple semaphores to high-level concepts such as Communicating Sequential Processes (CSP). Such different concepts may for instance affect the number of

tasks in a design, especially when some tasks selectively wait for different conditions without knowing which condition is met first. (This works well with CSP, but may be quite difficult, otherwise.) However, in embedded and real-time systems, the number of tasks definitely is a design dimension, because it affects schedulability. The target environment thus does influence the design. To a certain degree, an abstraction layer or “middleware” can be used to encapsulate details of the target environment. However, an all-embracing abstraction layer for all situations has not yet been found. Additionally, a chosen abstraction layer may be unsuitable in a given situation due to performance reasons or it may be inferior to the native concepts in the target environment. So a dependency from different abstraction layers (which depend themselves on different target environments) remains.

As a second example, the class concept is taken. As shown in the following subsection, the semantics of a class in the UML should be different, depending on the target programming language. So how could a design, which is supposed to be independent of the target programming language, be constructed? One could use a UML class diagram, but this would *only look* as if it were language independent. In fact, the same class symbol would employ different semantics, depending on whether C++ or Java code is to be generated. Arguing that a view on such details is small-minded has to be rejected, at least for safety-critical systems where the lives of human beings are at risk.

## 2.4 Further Comments on the UML

It was after the original publication of the above that the author encountered even stronger criticism of UML. In [Br01] and during the UML «2000» conference [Co00], the lack of a firm semantical basis of UML was pointed out. Some statements of the latter author during his presentation are repeated here to underline how severe the situation is: “... just pictures, no semantics...”, “... confused notations ...”, “... deeply confused about the meaning of semantics: the semantics section [of the UML definition] is mostly about abstract syntax.” It was indicated that a definition of the semantics of UML would have to take into account differing semantics of target languages, i.e. a family of languages were necessary. As an example, different semantics of inheritance in C++ and Java was given in [Co00]. Indeed, even an innocently looking int or integer does have different semantics in C++, Java and Ada, due to a different handling of overflow, as pointed out in [Ro99b]. Hence, the position that the programming language be in ultimate control and that the design modelling language has to represent the semantics of the programming language, is held by the author even more firmly now. But the demand is not only that the semantics of the modelling language must be precisely defined. Rather, from a practical point of view, if the programming language contains sound design elements (as it is the case with Ada), then the modelling language has to represent these elements with the *same* semantics, i.e. in an isomorphic way as requested above.

Concerning the semantics of packages in the UML, criticism has arisen with respect to its vague definition, changing from version to version of the UML specification. This criticism has been confronted by one of the proponents of UML with the blunt announcement, that the semantics of packages in UML would probably change again [Hr01]. Such a situation is completely unacceptable to large-scale and long-term development of embedded systems, especially in a project where the target language offers a stable and near to perfect package concept, as it is the case with Ada. Using UML's fuzzy and inferior package concept in such a situation would result in design modelling inversion, which is clearly to be avoided. To be useful in long-term projects, the UML would have to reach a level of preciseness and stability comparable to the Ada language.

### **3. Design Patterns**

Design patterns have come into widespread acceptance with the book of Gamma et al. [Ga94]. Following this reference, a design pattern may be seen as “a description of communicating objects and classes that are customized to solve a general design problem in a particular context”. In the following the concept of design patterns and its application to embedded and real-time systems is discussed.

#### **3.1 The Value of Design Patterns**

Design patterns are an extremely valuable concept. There is no need for discussing or questioning the value of design patterns in the view of the author.

#### **3.2 Early Use of “Design Patterns”**

The impression that software design patterns have been “invented” in the early nineties is a misconception. Design patterns have been in use especially in embedded and real-time systems long before their widespread publicity, even if they have not been termed patterns explicitly. To give a few early examples from the domain of task interaction, [Bu84] includes “patterns” for dynamic connections between tasks, and [NS88] elaborates on Buffer, Transporter, and Relay Task Intermediary “patterns”.

#### **3.3 Domain and/or Target Technology Dependency of Design Patterns**

Design patterns are usually domain and/or target technology specific. While some design patterns may be independent of the target technology to be used, many design patterns will be heavily influenced by the actual target programming systems or by the conceptual world of their creators. Thus, in most titles it should read “design patterns for xyz kind of systems” instead of just “design patterns”. The following examples illuminate this issue.

- The main reference for design patterns [Ga94] does not even contain the notion of a task or an interrupt service routine (ISR). Obviously, the book is targeted to a domain different from real-time systems, namely to Smalltalk or C++ systems. Hence, one of the most basic design patterns of real-time systems, the Primary/Secondary Reaction pattern (see figure 4), is out of the realm of this book. (Note that this is not at all a case against the book or the authors.)

It must be emphasised that tasks and interrupt service routines are not just some low-level concepts, but can be implemented in an object-oriented manner. For instance, in Ada protected units are to be used to encapsulate the interrupt service routine [Bu98], and in C++ they can be encapsulated in classes (with some additional implementation effort) as suggested in [Ru98].

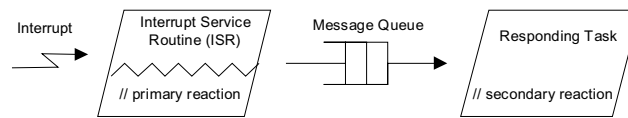


Fig. 4: The “Primary/Secondary Reaction” pattern in real-time systems

- The popular singleton design pattern (which ensures that only one instance of a class exists) may be quite different in module-oriented target languages such as Ada or Modula. A simple data object module (in contrast to a data type module which would represent a class) will automatically ensure the single instance property.
- Templates or generics are not considered with the patterns in [Ga94], because they “aren’t needed at all in a language like Smalltalk...”. Probably, some patterns, esp. the Template Method pattern, might look different (or might not be needed at all) for target languages with built-in template facilities.
- A Monitor Object pattern [Sc01] may reduce to a simple application of some feature of the target programming language, if the language directly supports such a (or a similar) concept.

### 3.4 Usefulness of Pattern Structure Diagrams in UML

The UML is widely (almost exclusively) used for visualising designs. It is still unclear, how UML can or should support the design of embedded and real-time systems in general (see [Se00, Do98]). As for the pattern structure diagrams in the (UML), these may be almost completely useless for describing patterns.

As a first example, it is observed that the very basic Primary/Secondary Reaction pattern cannot adequately be expressed in core UML. (Of course, it may be claimed that via UML’s extension mechanisms it can be expressed, as almost any concept or symbol is expressible with arbitrary UML extensions.)

Secondly, the usefulness of UML's pattern structure diagrams is demonstrated with the Priority Ceiling Pattern [Do00] (see figure 5). As easily seen, such a pattern structure diagram conveys almost no information at all with respect to the ideas behind the priority ceiling protocol. Such a design pattern diagram is completely useless. This is clearly not the fault of [Do00], but a shortcoming of the UML. A reasonable description of this pattern can of course be given with model elements outside the UML and with plain text [Do00].

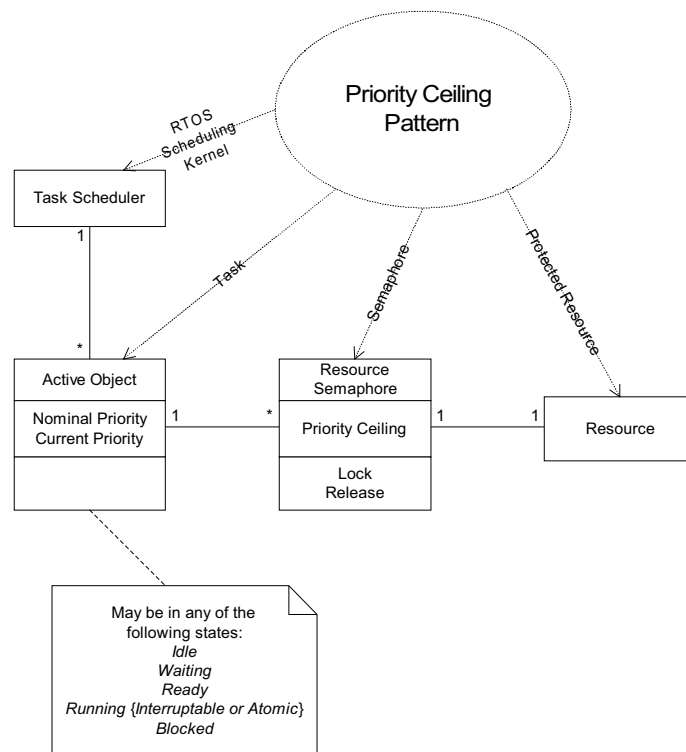


Fig. 5: Priority Ceiling Pattern (taken from [Do00])

### 3.5 Specific Properties of Embedded Systems

Embedded systems differ from other computer applications in a number of ways. For instance, embedded systems tend to be much more static than, say, a workstation application written in C++ or Smalltalk. This may mean that objects are allocated statically instead of being dynamically created and destroyed, that all program code is burnt in a read-only memory, that heap usage (if any) is minimised, etc. This also has its consequences concerning patterns. Patterns which induce significant run-time overhead may be unacceptable, other patterns may just be unneeded in such a context.

As an example from industrial practice, a seemingly trivial buffer object is considered (figure 6).

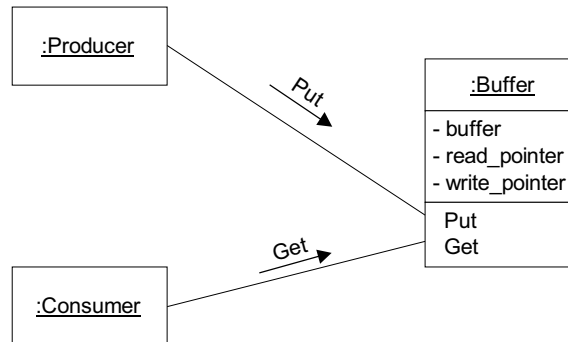


Fig. 6: A simple Producer/Consumer Buffer pattern

However, some typical embedded systems requirements make this an interesting object of study and, eventually, a design pattern:

- One part of this buffer pattern (the data acquisition part) is to be implemented as firmware on a specific hardware board.
- The buffer memory and the read and write pointers have to be at fixed memory addresses, forming the hardware interface.
- The other part of the pattern, i.e. further processing of the data, is implemented as software on a standard CPU.

This may result in a pattern as shown in figure 7. The buffer object is drawn on the edge of the hardware board to indicate that it is partly implemented as firmware, i.e. software on the board, partly as software on the main CPU. The firmware on the board comprises the “put” part, the buffer space, and the read and write pointers of the object, whilst the software on the main CPU implements the “get” part, accessing the buffer space and the read and write pointers via memory-mapped I/O.

Note that firmware and software are usually developed by different teams. Hence, it may be hard to detect such an object or class, spanning the responsibilities of different teams. Even more complexity arises when the firmware team also uses field-programmable gate arrays (FPGAs) to implement parts of the functionality in programmable hardware (e.g. [ST98]), now a common practice.

Further design decisions on the buffer object or class have to be made with respect to synchronisation, concerning genericity, and whether multiple instances should be allowed or not.

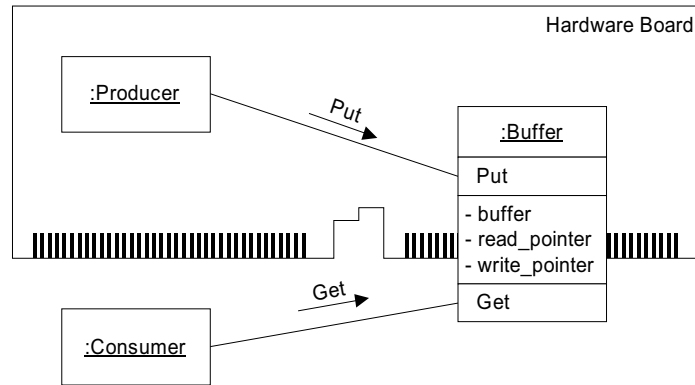


Fig. 7: A Producer/Consumer Buffer pattern with additional constraints of a real embedded system

To extend the above example, let us assume that in fact three consumers are given. These consumers interpret and eventually store or display the same data differently. It should be possible to individually switch these consumers on and off as desired with respect to user input. Note that the number of consumers is fixed at compile time. It was suggested to the author to solve this design task with the well-known observer pattern [Ga94]. This pattern involves an abstract and concrete server, an abstract observer, and concrete observers – the consumers. The consumers are notified by new data, and then updated. In fact, this pattern works for the given situation. However, such a solution is somewhat like an overkill. It involves a dynamically linked list (for the observers/consumers), dynamic memory allocation and deallocation (when switching consumers on or off), and, of course, inheritance, polymorphism, abstract classes, and the like. Apart from the involved overhead such a solution might also look difficult to the inexperienced, because – while being very flexible – the solution is not very explicit (cf. [Fo01] for the benefits of explicit designs). And part of the flexibility, the ability to bring in or take out new observers dynamically, is just not needed here. The author would instead suggest a simplified observer pattern for the given requirements, which would avoid the described drawbacks (see figure 8).

To summarise, design patterns are in many cases target or target technology specific. This means that for embedded and real-time systems the typical solution space with tasks and interrupt service routines, with fixed memory locations, with typical programming languages and operating systems, etc. has to be considered. More design patterns specific to the everyday design problems in embedded and real-time systems are needed!

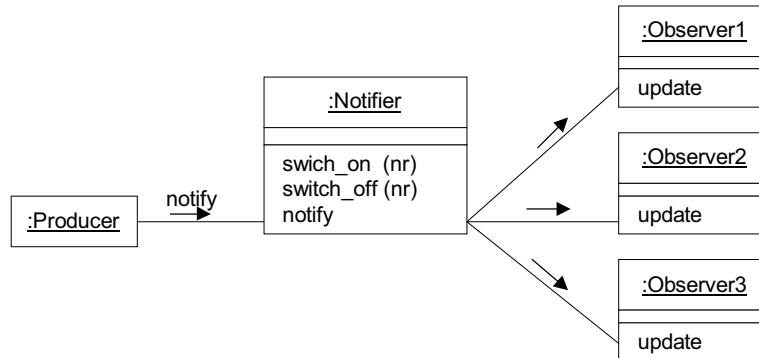


Fig. 8: A simplified observer pattern tailored to the real needs of a specific embedded system

#### 4. The Hot Spots: Future Modelling Elements and Current Practice

During the OMER workshops a variety of ideas about (and requests for) future modelling elements was discussed [Br01, Hr01, and contributions from the audience]. In the view of the author, some of these topics are really the “hot spots” of modelling embedded and real-time systems, recurring again and again. Hence they deserve some special attention.

In the following, these topics will be related to current practice. This will be done along the flight control software project mentioned in section 2.2. As part of this project control laws for a typical fighter aircraft have been implemented in Ada. The resulting software is called COLAda (Control Law Software in Ada) [Ro99a, Ro99b]. Flight control software represents one of the most demanding embedded real-time applications and should be of general interest with the advance of this technology into the automotive industry in the form of x-by-wire applications. So this is an example with a very realistic background.

##### 4.1 Inheritance and Polymorphism

In the design of COLAda object-oriented techniques have been used, where appropriate. Certain control law elements, e.g. filters, have been identified as candidates for objects, and corresponding abstract data types. (e.g. filter types) have been defined. Several objects of these types may be created as required by the control law application—the objects are just instances of abstract data types. The external behaviour of these objects is defined by the operations associated with the abstract data type. Of course all internal data are encapsulated in the objects.

According to a certain terminology, the design might be termed to be object-based, as no type extension (“inheritance”) is used. This is for the following reasons:

- Full use of inheritance, in particular polymorphism and dynamic binding (i.e. the use of class-wide types in Ada terminology), may cause certain problems in safety-critical systems (cf. [Is98]).
- There is hardly a need for using inheritance in the context of this project. The cases where variants of certain object types occur (e.g. first order and second order filters) can easily be covered without inheritance.

Though certain aspects of flight control software can be designed and implemented in an object-oriented way, and though the author very strongly favours such an approach, there must be a warning against hypocrisy with respect to object-orientation: The approach “everything is an object” does not seem very helpful. Over-simplistic approaches like in [Co95], where a case study of an object-oriented auto-pilot system is reported on, seem to be impractical for the implementation of real flight control software. And, of course, normal arithmetic operations and static typing are to be used in control law software. This is in contrast to the philosophy of radical object-oriented languages such as SmallTalk and Lisp, for instance, which are considered unsuitable to real-time safety-critical systems.

As discussed during the OMER-2 workshop [Br01], inheritance and its ramifications may indeed not be the desired way of building embedded real-time systems. In the view of the author, composition (of abstract data types) is to be preferred over inheritance in many cases. However, this has been the general guideline in large parts of the Ada and embedded systems community, anyway (e.g. [Ro92]).

## 4.2 Components and Composition

It has become clear that the class concept is too fine grained for building large scale systems [Br01]. Some larger building block, with a possibility to compose such blocks, is clearly needed. The term component will be adopted here for such building blocks, though this term is not precisely defined. It is unclear to the author, why something new should be necessary here. Rather, the demand during the OMER-2 workshop that one single concept should cover classes and components, seems very meaningful. It is possibly the influence of languages such as C++, which fosters requests for some additional silver bullet component concept. On the other hand, from the author’s experience with Ada, there seems to be an at least acceptable concept for structuring large systems. The Ada notion of a package does unify the class concept with the concept of larger building blocks. And the concept does have defined semantics (cf. section 2.4), and packages can be composed, since the advent of Ada95 even hierarchically in a very smart way.

## 4.3 Concurrency

The failure to adequately address concurrency issues in object-oriented development (e.g. in C++ or in UML) has also been addressed in [Br01]. As for the Ada projects the author has been involved in, there is in fact a concurrency model available. Notwith-

standing possible shortcomings, Ada's concurrency model is somehow very close to Hoare's Communicating Sequential Processes (CSP) – in the Ada95 version even highly performant – giving an level of abstraction and composability the author feels is sufficient. (This holds for concurrency within one processor, not for concurrency in a network of processors.) Note that Java, though more recent than Ada, has an inferior concurrency model, which does not scale – quite in contrast to CSP. Looking at the UML, the issue of concurrency once again gives rise to modelling inversion. That is, appropriate programming languages contain design features which are by far superior to the concurrency features of the “modelling language”.

## 5. Conclusion

As an overall conclusion, all the new and extensively marketed paradigms should be carefully evaluated with respect to exactness and completeness of the definition and concerning the suitability for real-time and embedded systems. In addition, proven and currently available technology must be included in a general assessment.

## References

- [Br01] Broy, M., Tempelmeier, T., Wirsing, M., Ziegler, J.: *OO Development of Distributed Embedded Systems – A Critical Assessment*. Panel discussion. OMER-2 (“Object-Oriented Modeling of Embedded Realtime systems”) May 9-12, 2001, Herrsching, Germany. The first author's position statement with similar content in German: Broy, M., Siedersleben, J.: Objektorientierte Programmierung und Softwareentwicklung – Eine kritische Einschätzung. (Object-oriented Programming and Software Development – A Critical Assessment.) Informatik-Spektrum, 25, 1, Februar 2002, p. 3-11.
- [Bu84] Buhr, R.J.A.: *System Design with Ada*. Prentice-Hall, Englewood Cliffs, 1984.
- [Bu98] Burns, A., Wellings, A.: *Concurrency in Ada*. 2<sup>nd</sup> Ed. University Press, Cambridge 1998.
- [Co95] Coad, P., North, D., Mayfield, M.: *Object Models. Strategies, Patterns, and Applications*. Yourdon Press, Englewood Cliffs, 1995.
- [Co00] Cook, S.: *The UML Family: Profiles, Prefaces and Packages*. In: Evans, A., Kent, S., Selic, B.: «UML» 2000 - The Unified Modeling Language. Advancing the Standard. Proceedings of the Third International Conference. York, UK, October 2000. Lecture Notes in Computer Science 1939. Springer, Berlin 2000. p. 255-264.
- [Do98] Douglass, B.P.: *Real-Time UML. Developing Efficient Objects for Embedded Systems*. Addison-Wesley, Reading 1998.
- [Do00] Douglass, B.P.: *Real-Time Design Patterns*. White Paper, I-Logix. On the internet <http://www.ilogix.com>. July 2000.
- [Fo01] Fowler, M.: *To be explicit*. IEEE Software, November/December 2001, 18, 6, 10-15.
- [Ga94] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading 1994.
- [Go00] Gomaa, H.: *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, Reading 2000.
- [Hr01] Hruschka, P., Björkander, M., Färber, G., Kopetzky, V.: *The Missing Concepts of UML for Designing Embedded RT Systems*. Talk Show. OMER-2 (“Object-Oriented Modeling of Embedded Realtime systems”) May 9-12, 2001, Herrsching, Germany.
- [Hr02] Hruschka, P., Rupp, C.: *Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML. (Agile Software Development for Embedded Real-Time Systems with the UML. In German)*. Hanser, München, 2002.

- [Is98] ISO/IEC: *Working Draft 3.8 - Programming Languages - Guide for the Use of the Ada Programming Language in High Integrity Systems*. ISO/IEC PDTR 15942, ISO/IEC JTC 1/SC22/WG9, October-29, 1998.
- [Ka92] Kaul, H.J.: *Flugsteuerungssystem Jäger 90* (Flight Control System of the Fighter 90. In German). In: G. Bürgener (Schriftleitung). Jahrbuch 1992 III der Deutschen Gesellschaft für Luft- und Raumfahrttechnik e.V. (DGLR). Deutscher Luft- und Raumfahrtkongreß 1992. DGLR Jahrestagung, Bremen, 29. September – 02. Oktober 1992. Deutsche Gesellschaft für Luft- und Raumfahrt e.V. (DGLR), Bonn 1992.
- [MM98] McLaughlin, M.J., Moore, A.: *Real-Time Extensions to UML. Timing, concurrency, and hardware interfaces*. Dr. Dobb's Journal, December 1998.
- [NS88] Nielsen, K., Shumate, K.: *Designing Large Real-Time Systems with Ada*. Intertext Publications & McGraw-Hill, New York 1988.
- [Om99] OMG: *Unified Modeling Language Specification*. Version 1.3, June 1999. On the internet or on CDROM in: Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading 1999.
- [Ro92] Rosen, J. P.: *What Orientation Should Ada Objects Take?* Communications of the ACM, 35, 11, November 1992, p. 71-76.
- [Ro99a] Roszkopf, A.: *Development of Flight Control Software in Ada - Architecture and Design Issues and Approaches*. Ada-Europe'99. International Conference on Reliable Software Technologies. June 7-11, 1999. Santander, Spain. Springer Lecture Notes in Computer Science, 1622. pp. 437-449. Springer, Berlin 1999.
- [Ro99b] Roszkopf, A., Tempelmeier, T.: *Aspects of Flight Control Software - A Software Engineering Point of View*. 24th IFAC/IFIP Workshop on Real-Time Programming, Schloss Dagstuhl, Saarland, Germany, May 31 - June 2, 1999. Pergamon, Elsevier Science, Oxford 1999. Also in: *Control Engineering Practice*, June 2000, 8 (2000), p. 675-680.
- [Ru98] Rusch, D.G.: *Encapsulating ISRs in C++*. Embedded Systems Programming, February 1998.
- [Sc01] Schmidt, D.C.: *Monitor Object – an Object Behavior Pattern for Concurrent Programming*, (updated October 10th) C++ Report, SIGS, planned to appear 2000. On the internet: <http://www.cs.wustl.edu/~schmidt/patterns-ace.html>. February 2001.
- [Se94] Selic, B., Gullekson, G., Ward, P.T.: *Real-Time Object-Oriented Modelling*. Wiley & Sons, New York 1994.
- [Se00] Selic, B., Burns, A., Moore, A., Tempelmeier, T., Terrier, F.: *Heaven or Hell? A "Real-Time" UML?* In: Evans, A., Kent, S., Selic, B.: «UML» 2000 - The Unified Modeling Language. Advancing the Standard. Proceedings of the Third International Conference. York, UK, October 2000. Lecture Notes in Computer Science 1939. Springer, Berlin 2000. pp 93-100.
- [ST98] Schrott, G., Tempelmeier, T.: *Putting Hardware-Software Codesign into Practice*. Control Engineering Practice, 6 (1998) 397-402. Volume 6, Issue 3, March 1998
- [Te94] Tempelmeier, T.: *An Overview of the HOOD Software Design Method*. In: Real Time Computing. NATO ASI Series F, Vol. 127. Halang, W.A. and Stoyenko, A.D. (eds.). Proceedings of the NATO Advanced Study Institute on Real Time Systems, Sint Maarten, Dutch Antilles, October 5-17, 1992. Pages 726-734. Springer, Berlin 1994.
- [Te98] Tempelmeier, T.: *Hierarchical Object-Oriented Design (HOOD) – Die Software-Entwurfsmethode der europäischen Raumfahrtbehörde ESA*. (Hierarchical Object-Oriented Design (HOOD) – The Software Design Method of the European Space Agency ESA. In German.) Kolloquiumsvortrag an der Universität Oldenburg, 29.6.98. On the internet: <http://www.fh-rosenheim.de/tempelmeier>.
- [Te99] Tempelmeier, T.: *UML is great for Embedded Systems – Isn't it?* In: P. Hofmann, A. Schürr (eds.): OMER ("Object-Oriented Modeling of Embedded Realtime systems") Workshop Proceedings. May 28-29, 1999, Herrsching (Ammersee), Germany. Bericht Nr. 1999-01, Mai 1999. Universität der Bundeswehr München, Fakultät für Informatik.
- [Te01] Tempelmeier, T.: *Comments on Design Patterns for Embedded and Real-Time Systems*. In: A. Schürr (ed.): OMER-2 ("Object-Oriented Modeling of Embedded Realtime systems") Workshop Proceedings. May 9-12, 2001, Herrsching, Germany. Bericht Nr. 2001-03, Mai 2001. Universität der Bundeswehr München, Fakultät für Informatik.