Reports in Informatics no. 208, October 2000, pp. 12, ISSN 0333-3590.
Department of Informatics, University of Bergen, N-5020 BERGEN, Norway

# A 2-tiered software process model for utilizing CASL*

Magne Haveraaen
Department of informatics, University of Bergen
P.O.Box 7800, N-5020 BERGEN, Norway

October 17, 2000

**Abstract**

The broader adoption of CASL will depend on its use being perceived as beneficial in the software development process. Traditionally the algebraic specifications community has focused on the use of algebraic methods for development of correct software within the standard *waterfall software process model* (and derivatives) for the implementation of software. Here we suggest a two-tiered softare process model, in which the first tier focuses on problem domain investigation and software architecture development, while the second tier is the implementation of software. Our experience has shown that algebraic methodology may play a significant role in the first tier, with significant improvements in software productivity as a result.

## 1 Introduction

Sofware engineering research [PCCW93] tells us that an organisation's ability to follow and reflect upon its software process is very important for the achievement of quality software. This is also stated in the international standard for software quality assurance [ISO97]. The first software process models were formulated in the 1960/1970s and focused on the development of application software. The most basic of these was the *waterfall software process model* [Roy70]. It is a very simple model and has obvious shortcomings. Variations were soon developed [Boe76], and these have been further refined over the years. But still the waterfall software process model identifies the central steps of application software development.

Formal methods are also concidered a useful tool in the ensurance of correctness and quality of software. Among these are algebraic methods, which the CoFI initiative aims at promoting. Algebraic methods arose from the mathematical disiplin of universal algebra, and have been seen as contributing to two phasets of mathematics:

*This research has been partially supported by ESPRIT working group 29432 (CoFI WG).

- investigation of problem domain concepts and
- investigation of structural relationships between concepts.

Being a formal method, much attention in computer science has been geared towards the use of algebraic methods in the development of correct software. This attention has produced much significant work, but has meant that the structuring aspects of algebraic methodologies have been receiving less attention.

A lack of interest in structural aspects of software may have been OK up until a decade ago, but with the current focus on software packages and software components, the structural aspects of software play an ever increasing rôle. Just think of program packages like office automation systems, where each program component is expected to seamlessly interact with all the others, sharing and exchanging data. Or concider software libraries like window packages, which supply data structures and operations for user interaction with application programs.

We may consider both kinds of software packages, i.e., program packages and library packages, as consisting of:

- types (data structures and data file formats) used for the exchange of data between entities in the software package, and
- operations (programs, procedures, methods) that perform the appropriate actions (such as formatting a page or updating the screen) on the data.

This is quite analogous universal algebra, where we have sorts and functions. Using algebraic terminology, we may consider a package *interface* (the syntactic declaration of types and operations) as a signature and the software package itself as a model for the interface. A software package specification limits the class of acceptable implementations of an interface, just as algebraic specifications limit the class of models for a signature.

Such an analogy obviously ignores many important difference between software and algebras, e.g., that software update (mutate / modify) data. But when looking at the structural side of software, these differences may initially be ignored, without causing problems for the broad picture.

When designing a software package for some problem domain, there are at least three major question to answer:

1. what are the concepts the package should support,
2. what is a good programming notation for these concepts, and
3. how can the concepts be implemented as software code.

This has spawned work in areas like *domain specific (embedded) languages* (DSL/DSEL) [Hud96] and *software architectures* [BST99]. Domain specific embedded languages provide the concepts from the problem domain with syntax in a general purpose programming language setting. Software architecture is the organisation of software into packages and modules and how to combine these in order to build specific software. Ideally software should be decomposed in such a way that different considerations may be confined to separate modules. Also, modules that provide alternative implementations of a concept should be easily interchangeable. This requires an extensive analysis of both the problem domain and of possible software structures. Doing this well may give very flexible software, easy to tailor for specific needs, as well as adapt to changing requirements – a characteristic of mature software design [Rac95].

Placing importance on the structural aspects of software, it then seems that we need a software process model that can address both

1. finding what concepts are useful and needed when approaching a problem domain with the purpose of developing a software package for it, and
2. how to implement the components of the software package as computer code.

Based on [HFJ99] we will suggest a 2-tiered software process model. The first tier is specifying the package we are to develop. Integrated with the specification process we will also include decomposing the package into sub-packages and providing an architecture for it. The second tier is building a concrete implementation (model) for the sub-packages. The algebraic specification language Casl has specific support for several of these issues.

This paper is structured as follows: Next we recall the waterfall software process and propose the basic 2-tiered software process. In section three we give some examples of development according to the 2-tiered model. Finally we discuss the relationship to Casl in section four.

# 2 Software Process Models

Software process models, or software life cycle models, show how one envisages a piece of software comes about, and what will happen to it from its creation till it is taken out of use. There are many variations of process models, we will briefly present two: the standard waterfall model and our new package oriented model.

## 2.1 Waterfall Process Model

The waterfall process model is one of the oldest software life cycle models. Its origin may probably be traced back to [Roy70], but many variations are presented, see e.g. [Boe76]. It is still in many ways the reference model, which most other life cycle models either relate to or are variations of. Roughly speaking, the waterfall model implies that software development passes through the following phases:

1. *Requirements specification*: analyse the needs of a costumer and formulate them in the requirements document. This includes functionality, bounds on development cost, hardware resource requirements, etc. The requirement document will contain a

specification of the package as one of its parts. The finished requirement specification is a contract between the software developer and the costumer.

Included in the requirements phase is costumer requirement analysis and feasibility studies, and many modeling techniques have been developed for this purpose. Algebraic specification techniques such as those supported by CASL are among these.

2. *Design phase*: produce a detailed description of the data structures and algorithms in the form of a design document.

   The model oriented development methodologies advocated in [ST88, ST92] supports this steps.

3. *Coding*: express the design as program code in some programming language.

4. *System integration* (configuration and testing): putting the system's components together to form the package, and checking the interaction between its components. If the interface between the parts have *not* been carefully defined, system integration may become very expensive. The mismatches have to be resolved, perhaps forcing another iteration of design and coding.

   If the software development methodology does not place enough emphasis on interface design and conformance, this phase can be very costly. It is especially important that local consistency of components imply global consistency of the software package. Algebraic development methodologies may be of siginificant benefit here, as they are designed to provide the necessary compositionality of components and focus on interface development.

- *Acceptance testing*: the costumer checks that the requirements have been met, and that the package may be used in the way envisioned when the project was started. Acceptance testing is not a development phase as such, but is a major milestone in the life of a software system. After passing the acceptance test, the package goes into regular use and the development is finished.

5. *System maintenance*: this starts the moment the first costumer begins to use a piece of software. Errors are discovered, suggestions for improvements show up, and the need for modifying the functionality of the system become obvious. Maintenance is to improve the package according to these needs and wishes.

Each of the development stages are concluded by validating the result, in order to discover as many errors as possible and correct them in the phase where they were made. For instance, a mistake in the requirements phase which is not discovered until the acceptance test, may require the whole development project to begin from scratch, obviously adding considerably to the total costs of the system. In practice the distinction between the phases will not be so strict as suggested here, and there will be many feedback loops between the different phases.

The waterfall model is useful since it names important phases in software development. It also provides a good description of how a software development project should evolve. On the downside, it considers maintenance as one block, even though this usually is the major cost factor of a software system. Also, being oriented towards single development projects, it does not take into account reuse of code and the use of software libraries. Within this framework, preparing modules for a software library seems to add cost to a project without providing benefits, so many project managers have discouraged such activity. Thus adhering to the waterfall process model may encourage the development of monolithic pieces of software rather than generating reusable components.

## 2.2 A two-tiered software process model

The starting point for the proposed 2-tiered software process model is that any specific problem belongs to (at least) one problem domain. The normal situation is that we probably will want to solve more than jost one problem from this domain. Thus a proper understanding of the domain concepts and how to structure them is important. Two distinct structures need to be considered: the specification's structure which will evolve as the specification is built, and the implementation's structure, which should be subject to a proper investigation. Then the specific problems may be solved, using the domain specific concepts (which form a *domain specific language* – DSL) and infra-structure developed from the problem domain investigation. This was the idea we used in the project reported in [HFJ99].

The software processes for each of the two tiers may be idealised into linear process models. The first tier concerns development of a DSL and the DSL-supporting software library.

**Tier 1:** problem domain investigation and software infrastructure development.

1. *Establish an appropriate DSL*: define important properties of the problem domain concepts. Algebraic specification methods are obviously useful here. They identify interfaces (types and operations) and nail down the semantics of the DSL in sufficient detail, i.e., we may think of the DSL as a software package.

   Note that several DSLs may exist for a problem domain. It then becomes important to select a DSL which is fruitful from a software perspective, e.g., one which provides high level abstractions without sacrificing expressivity.

2. *Validate the suggested DSL*:

   - *Validate appropriateness of concepts*: formulate specific problems and solutions using the DSL. The DSL should be embedded into a programming language – forming a domain specific embedded language (DSEL) [Hud96] – to achieve full algorithmic expressivity when doing these investigations.

     Designing solutions to specific problems is a tier 2 activity. For validation purposes only the first couple of tier 2 phases are needed, i.e., problem specification and algorithmic design.

- *Validate implementability of concepts*: show the existence of data structures for the sorts and algorithms for the functions of the DSL.

  This is also a tier 2 activity, namely that of algorithmic design. This time the DSL itself is the problem specification. For validation purposes only the existence of a possible implementation needs to be established.

3. *Decide the architecture of a software package* for the DSL: organise the DSL into sub-packages and devise how the DSL package may be built from these.

   Software architecture design is one of the most important steps of this software process model. The step involves grouping the sorts and functions into sub-packages, studying their interplay and trying to parameterize the sub-packages as much as possible. The aim is to reduce the number of components and the complexity[1] of each by focusing on reusability. A good architecture design increases the versatility of the components, and simplifies the library itself. It may also greatly reduce the work needed to implement the full library. Categorical reasoning and software architecture expressions such as those found in CASL are useful for this purpose.

4. *Maintain the DSL package*: evolutionary implementation of the package and improvement of its components. Each of these developments are tier 2 tasks, only their overall administration is a tier 1 task. This phase includes initiating the implementation of interchangable variants of the sub-packages, as well as correcting (unavoidable) errors in the components. Many variants of any sub-package may coexist in the DSL library, as variants may adhere to different constraints on resource usage, hardware architecture, etc.

The first of these steps identifies the domain concepts and finds a notation for these. Once the specifications are in place, it is possible to experiment with the notation and the concepts. Such experiments include the description of specific problems and the design of algorithms that will use the library for solutions of the specific problems. This allows early checking of the usefulness of the concepts, and an adjustment of the concepts and notation at this stage is fairly cheap. We also have to check that the concepts are implementable, i.e., that there exists data structures and algorithms that approximate the concepts to a sufficient degree[2] before we can accept them as meaningful in a software context. Thus we have both outward (useability) and inward (implementability) requirements on a DSL as issues in the validation of it. The specification of the concepts create a DSL, while using this language for the solution of problems, we embedd it into a programming language, creating a DSEL.

There is seldom a unique DSL for a problem domain. Careful evaluation is needed to decide which DSL to choose. Evaluation criteria include software complexity and reusabil-

[1] *Software complexity* is a measure of the difficulty and cost of writing software. It does not relate to the resource usage at runtime which is the subject of traditional complexity theory.

[2] We use the word *approximate* here as the finite limitation of computing devices often means that we cannot find implementations that fully comply to a specification. The concept of a stack, e.g., is unbounded, whereas an implementation of a stack would have to be bounded.

ity, criteria which may be difficult to fully assess until the package has been developed and taken into use. Also the architecture for a given DSL will seldom be unique. Again criteria like software complexity and reusability come into play.

Tier 2 concerns the development of an implementation for a package (a suit of related programs or a library package), and is quite similar to the waterfall model. A typical starting point is the implementation of the sub-packages identified by the software process in tier 1, but also application programs for specific problems are developed using the tier 2 process model. When a tier 2 process commences, there exists some DSEL for the problem domain we are approaching. This DSEL may not be anything more than a programming language with its built in types and operations, or it may be based on a full fledged DSL providing high level concepts from the domain.

**Tier 2:** Implementation of package components.

1. *Specify the specific problem to be solved and its requirements* by identifying the components to be implemented, resource requirements etc. Normally this is simple since the components already have been specified in tier 1. If an operation (or program) that was not part of a specified package is needed, this may also involve functional specifications, but then the appropriate DSL for expressing this is available from the domain analysis perormed in tier 1.

   Requirements specification should not incur a full problem domain analysis, as this belongs to tier 1.

2. *Design components*:

   - *for every sort* to be implemented: decide *data structure*, *data invariant* and *abstraction function* relating the data structure to the specified model classes.
   - *for every operation* to be implemented: decide an *algorithm* such that it will, on the concrete data, maintain the data invariant and perform the computation required by the specification relative to the abstraction function.

   The relationship between data structures and algorithms were identified already in [Mor73], and are the standard correctness requirements on software with respect to a specification. There is normally no unique solution for the design. It is quite normal to analyse the suggested solution and iterate on alternatives to decrease implementation cost and increase runtime efficiency of the implementation.

3. *Code the components* as software using the desired DSEL.

4. *Configure the components* by specific choices of sub-packages for the DSEL being used in the implementation. Configuration choises will to a large extent determin runtime resource usage.

5. *Maintain the components* by correcting (unavoidable) errors and, to some extent, adapt the components to new requirements. In many cases changed requirements incurs a new tier 2 development, yielding a new implementation which will co-exist with the existing implementations in order to meet the changed requirements.

The first phase of tier 2 concerns the requirements specification of the package to be developed. If the process is initiated from tier 1, a specification already exists, and this phase becomes rudimentary, since only requirements like resource usage will need to be stated. Otherwise a specification for a new package needs to be developed. The appropriate DSEL may be used in the requirements specification. This is typical of the situation when a new application program is to be added to an existing suite of application programs for some problem domain.

Tier 2 is known as *programming by contract* [Mey91], and is possible using object-oriented languages or other programming languages with similar modularisation concepts. Note that programming by contract works both for implementing a fully specified package (whether a suite of application programmes or a library package) or for adding an extra operation to an already implemented package (such as developing an application program or an extra operation for a DSL).

Of course these tiers cannot be run through in linear order. Within each tier feedback loops are needed, and various forms of feedback between the two tiers must be organised. The maintenance phases of both tiers will obviously be intertwined.

Typically, once a problem domain DSL has been specified and its architecture decided, many independent tier 2 software processes will be spawned, as each of the sub-packages need to be implemented. Such spawning will continue as a regular part of mainting the DSL support library when new variations of sub-package implementations are needed. This provides an *evolutionary* approach to the implementation of a large software package. For instance, a DSL containg a dictionary as a sub-package may spawn several dictionary implementations as the need arises: linked list, sorted linked list, binary search tree, AVL tree, hash table, and so forth. Also a DSL (tier 1 process) may be refined as the development of application programmes (tier 2 processes) gives a better understanding of the problem domain.

A common case is that the development of implementations of the DSL developed in tier 1 will be driven by the needs of the application programs being developed. Then the cost of implementing the DSL may be spread out, even though the cost of investigating the DSL and designing its architecture must be taken initially.

# 3 Examples

In this section we give some examples of various software packages whose development may be organised according to the 2-tiered model presented above.

Tier 1, development of DSLs and software library architectures for some sample domains:

- Development of DSL for window handling, such as the X-window DSL. The library is organised around a few primitive operations and data structures. The other operations are defined in terms of these primitives.
- Development of DSL for paint and draw object handling, i.e., with curves, ovals, rectangles, composite objects etc. being manipulated (group, ungroup, shift, twist, stretch, . . . ).

Tier 2, implementing each of software packages defined by DSLs, and the development of application programs:

- Implementation of the X-window library for the X-window DSL. This was one of the early experimental graphical user interfaces.
- Implementtion of the Motif library for the X-window DSL. This is an X-window implementation with a different (Macintosh) look and feel.
- Implementtion of the CDE library for the X-window DSL. This is an X-window implementation with a yet another look and feel.
- An implementation of the paint and draw object DSL using a window handling DSL.
- Specification and implementation of a paint and draw application program using the graphical object DSL.

In [HFJ99] we reported on the development of the Sophus numerical software package architecture and some of its implementation. Sophus is a software package for high performance computing of partial differential equations (PDEs). Its development is a fully worked example following the 2-tier process model using algebraic methods in a rather untraditional domain. the use of .

- Traditional tier 1 investigation of the PDE problem domain has given a DSL which to a large extent is an integral part of most programming langauges: scalars (float, double etc.), arrays, matrices, discretised data and operations for the numerical algorithms on the various types of discretised data. The architecture is fairly flat, i.e., no component relly builds on others.
- The more algebraic tier 1 investigations yielded a coordinate free DSL for the PDE domain: scalars, vectors, tensors, manifolds, scalar fields, tensor fields. The Sophus software architecture builds tensors and tensors fields by tensor constructions from scalars and scalar fields respectively. It has several other internal reuses of components, which reduces the amount of work needed for its implementation.

A new tier 2 processes is spawned for each of the sub-packages identified by the Sophus library architecture, such that each sub-package may be implmented independently of the others. Further, the implementation of the Sophus library is evolutionary, in that library components are only implemented when needed by a new application.

- A new numerical discretisation methodology is added by a tier 2 development of a new implementation for the scalar field sub-package.
- Implementation of a PDE application program is a tier 2 process. This process may request – to the tier 1 process – the implementation of a new discretisation methodology or new versions of other sub-packages identified by the DSL.

So far this has worked very well, even though we have had to improve the coordinate free DSL as new applications have demanded radically different approaches to the numerical methods used, see e.g. [GHW98]. The robustness of the Sophus architecture, developed using algebraic methodologies, has been demonstrated as it has remained unaltered through these radical changes. We have also shown that the cost of developing a new application may be brought down to 15% of the normal development cost, even counting all ammendments that have to be added to the Sophus software library by request of the new application [HFJ99]. This cost was shown for an application that traditionally would have required a full new application development process.

# 4 Discussion

We have preented a 2-tiered software process model:

**Tier 1:** problem domain investigation and software infrastructure development.

**Tier 2:** implementation of package components.

The former is actively pursued by the domain specific language communities [Hud96], but has not been identified as a software process linked to, but seperate from, the traditional software developmet process. The latter tier has traditionally been seen as the software development process [Roy70, ST88, Mey91].

Our experience, reported in [HFJ99], is that algebraic developmet methodologies [EGW98] can be very useful in several of the key steps in tier 1, mostly step 1 (DSL investigation) and step 3 (architecture investigation). Further, that the benefits of algebraic methods are much greater in the tier 1 process, domain investigation, than in the tier 2 process, package implementation. Tier 2 is closer to the traditional waterfal process model, but makes explicit reference to the domain and software architecture concepts developed in the tier 1 process. Specifically we will argue that CASL may be of use in the tier 1 software process.

It is well known that algebraic specification methodology is good for investigating language concepts [vDHK96], and thus CASL as an algebraic specification language is useful in identifying the DSL being developed in tier 1. But using a powerful language like Casl it is important to be aware of the constraints that follow from our need to embed the DSL into a programming language environment to give us a DSEL. This constraint implies that certain features of Casl should be used with caution. Also, a DSL will typically be a large package, and a well-structured specification is important to make its description understandable. It is important not to confuse the structure of the *specification of* the DSL with the structure of the *implementation of* the DSL. Casl has strong facilities for expressing both kinds of structure, unlike many other algebraic languages which are incapable of describing the structure of an implementation.

# References


[Boe76] B.W. Boehm. Software engineering. *IEEE Trans. Comput.*, C-25:1226–1241, 1976.

[BST99] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In Armando Martín Haeberer, editor, *Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 341–357. Springer, 1999.

[EGW98] Hartmut Ehrig, M. Gaiewsky, and Uwe Wolter. From abstract data types to algebraic development techniques: A shift of paradigms. In Francesco Parisi Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 1–17. Springer Verlag, 1998.

[GHW98] Phil Grant, Magne Haveraaen, and Mike Webster. Tensor abstraction programming of computational fluid dynamics problems. Technical Report CSR3-98, Department of Computer Science, University of Wales Swansea, Swansea SA2 8PP; United Kingdom, October 1998.

[HFJ99] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *Computing Surveys*, 28(4es):only available via http://www.acm.org/pubs/articles/journals/surveys/1996–28–4es/a196–hudak/a196–hudak.html, December 1996.

[ISO97] Quality management and quality assurance standards - part 3: Guidelines for the application of ISO 9001:1994 to the development, supply, installation and maintenance of computer software (ISO 9000-3:1997). Technical report, SIS - Standardiseringen i Sverige, 1997. UDK 658.56:681.3.06.

[Mey91] Bertrand Meyer. Design by contract. In Dino Mandrioli and Bertrand Meyer, editors, *Advances in object-oriented software engineering*, pages 1–50. Prentice Hall; Englewood Cliff N.J., 1991.

[Mor73] J.H. Morris. Types are not sets. In *Proceedings of the ACM Symposium on the Principles of Programming Lanugages*, pages 120–124, October 1973.

[PCCW93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. The capability maturity model for software, version 1.1. Report CMU/SEI-93-TR-24 and ESC-TR-93-177, Software Engineering Institute, Carnegie Mellon University; Pittsburgh, Pennsylvania 15213, 1993.

[Rac95] James Tiberius Racko. Choose me. *Software Development*, pages 81–84, April 1995.

[Roy70] Winston W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proc. WESCON*, pages 1–9. IEEE Press, New York, 1970.

[ST88] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[ST92] D. Sannella and A. Tarlecki. Toward Formal Development of Programs from Algebraic Specifications: Model–Theoretic Foundations. In W. Kuich, editor, *Automata, Languages and Programming*, pages 656–671. Springer LNCS 623, 1992.

[vDHK96] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.