Joint efforts to dispel an approaching modularity crisis.
Divide et impera, quo vadis?

Stephan Herrmann
Technical University Berlin,
stephan@cs.tu-berlin.de

Mira Mezini
Darmstadt University of Technology,
mezini@informatik.tu-darmstadt.de

Klaus Ostermann
Siemens AG CT SE 2,
Klaus.Ostermann@mchp.siemens.de

Abstract

In this paper we consider two important trends in improving separation of concerns: (a) the emergence of server-side component frameworks, and (b) the emergence of advanced approaches to software decomposition/composition. These two trends have emerged independently from each other, the first one in an industrial setting and the second one originating mostly from the object-oriented languages research community. Despite this independent development, both trends have quite some commonalities: not only do they follow the same goals, the key concepts are also basically the same. However, an effort to put both trends into a common reference frame, showing their commonalities, their differences, drawing boundaries on their application areas, analyzing how they complement each other and eventually profit from each other, etc., is still missing today. This paper is a modest effort to fill this gap.

1 Introduction

In tomorrow's software development, creating self–contained systems from scratch will rarely be found. Common tasks will be to enhance the (re-)usability of existing components and to compose systems out of existing (reusable) building blocks. This poses great challenges to the discipline of software engineering. There is a central issue which is as old as the "software crisis": the principle of separation of concerns. This is and will be our most important tool for managing complexity. If reuse and evolution are to be chief approaches in the component age, growing complexity calls for "Advanced Separation of Concerns". Approaches to separation of concerns from different areas, such as software architecture, requirement analysis, programming languages, component technology, etc. need to be put to work to *jointly* improve modularity in and across all phases of software development, to do this in multiple dimensions simultaneously, and to do this at different — especially larger — scales. If efforts are not combined we might be just at the dawn of a new crisis: a "Modularity Crisis".

In this paper we consider two important trends in improving separation of concerns: (a) the emergence of server-side component frameworks such as Enterprise JavaBeans (EJB) [14] and Corba Component Model (CCM) [12], and (b) the emergence of advanced approaches to software decomposition/composition such as AOP [5], Hyperspaces [15], Adaptive Plug-n-Play Components [10, 11], etc. These two trends have emerged independently from each other, the first one in an industrial setting and the second one originating mostly from the object-oriented languages research community. Despite this independent development, both trends have quite some commonalities: not only do they follow the same goals, the key concepts are also basically the same. However, an effort to put both trends into a common reference frame, showing their commonalities, their differences, drawing boundaries on their application areas, analyzing how they complement each other and eventually profit from each other, etc., is still missing today. This paper is a modest effort to fill this gap.

2 Server-Side Component Frameworks and Advanced Separation of Concerns

Server-side component models such as EJBs and CCM add powerful abstractions to the bare "distributed objects" layer, in order to support a clean separation of server-side application logic from other concerns such as distribution, transaction management, database connectivity, etc. Application logic is encapsulated in components, which are basically objects that do not explicitly deal with distribution or persistency issues. Beside the fact that the application code has to follow some conventions, the application logic is written as if it would run on the local machine and without any persistent storage of data.

Concerns such as distribution, persistency, access control, resource management, connectivity to external programs, etc. are implemented by application servers. Thus, application servers are to components what an operating system is for regular programs. We will use the attribute "cross-application" for referring to this set of concerns, as opposed to "application-specific" for referring to different concerns involved with application logic. The composition of the application-specific with cross-application concerns happens during the assembly and deployment phase of the process of developing server-side software. This is mainly a declarative process. The deployer of a component into a particular application server specifies in a separate "integration unit" – the so-called *deployment descriptor* – the connection points where certain cross-application concerns get "woven"[1] into the application logic encoded by the component.

[1]Although it is not found in the common component vocabulary, we use this term here in analogy with the AOP terminology

For instance, by specifying the attributes of the component that will be persistent together with their mapping to data in the database, the deployer weaves the application logic with that part of the application server which is responsible for persistency. Similarly, deployment descriptor specifications about the transactional features of the business objects will weave server's implementation of transaction management into the application logic.

The deployment descriptor specifications influence the process of executing code "around" the component by the server. However, the main control as when and how these two categories of concerns are melted together remains in the responsibility of an application server. Different servers e.g., apply different strategies as whether the synchronization of application-logic attributes with corresponding data in the database will take place after any method call on the component or only when data is changed.

So far, we have considered only separation and late composition of cross-application concerns with application logic and nothing has been said about decomposing the application logic itself into smaller units. This is an important issue, though. The application logic of web-based software e.g., in domains such as order processing, warehouse management, etc. has its own complexity apart from the the complexity added due to distribution, security, database, etc. issues. Managing this complexity, also calls for an approach to software development as a process of assembling reusable components encoding some functionality within an application domain, which are possibly developed independently and by different providers.

Of the two component frameworks considered so far, the CCM model is more advanced, in supporting component-based development of application logic. A CCM component provides several facets and specifies a set of receptables (what it expects to be provided by the "external world"). In addition, a component has event sources, event sinks, and a set of customizable attributes. During the *component assembly* stage, receptables of a component `A` are connected to facets provided by some other component `B`. Similarly, event sources are connected to appropriate event sinks.

Regarding object oriented systems in general, the connections between the different parts of a software system are hidden deep in the implementation. The advantage of separating the inter-component connectivity from the source code is that independent components from different vendors can be combined by connecting their "plugs" once a common interface exists that one component provides and the other requires.

3 Object–Oriented Design Techniques and Language Models for Advanced Separation of Concerns

Prior and parallel to the emergence of component frameworks, the last decade of research in object-oriented programming has brought about advanced modularization techniques of object-oriented systems based on design patterns [4] and frameworks [3] as well as various advanced language models for software composition beyond classes, inheritance and object composition. These models approach the danger for a potential "modularity crisis" from an architectural, respectively, programming language perspective. The key in this development has been the observation that the basic object–oriented mechanisms for separation of concerns, inheritance and object composition, alone without at least obeying to certain architectural infrastructures in the design (which is basically what patterns and frameworks are about) are not enough to achieve an effective separation of concerns.

Not only it is hard in an object-oriented design to encapsulate concerns such as distribution, persistency, or error handling, which in fact desperately cut across the class structure of applications. When it comes to the implementation level, it is also hard to capture high-level functionality realizing use-cases from the analysis and design phases into encapsulated units [10, 15]: a class generally contains code realizing part of the functionality from several use cases, while the code that realizes a use case is spread around several classes. Code tangling and scattering are the result. Let alone being able to buy the implementation for certain use-cases in the form of components that are reusable within or across an application domain.

Design patterns and frameworks approach the "modularity crisis" *within* the object-oriented setting, rather than beyond it. They improve the reusability of object-oriented software by making variation points in the software explicit. However, these variation points have in general to be anticipated. With most of the design patterns, you have to plan today for tomorrows flexibility, although you (a) have only a vague understanding of tomorrow's requirements, and (b) you are not even sure that all the flexibility you built in the software will ever be needed.

Frameworks are partial systems which — within certain conditions — provide great flexibility by means of "upfront" design. By this, frameworks are reusable for a family of systems. Using frameworks as reusable *components* in the intended meaning is, however, hard because of the problems of combining independently developed frameworks [7, 16, 11].

In contrast to patterns and frameworks, the language models mentioned above (from now on referred to as *ASOC models*) approach the problem by extending the object-oriented model with new composition mechanisms. The key message of the ASOC models is (a) the principle of separation of concerns becomes really effective in managing complexity when it is supported by dedicated linguistic means of modularization along several dimensions of concerns, e.g., data, features (functions), cross-application concerns, etc., and (b) todays languages do not provide such linguistic means. Very broadly speaking, we have basically seen two different approaches to decomposing software so far: function-driven versus data-driven decomposition. Both support decomposition along a single dimension, while the structure of the systems we built is at least two dimensional [16, 10, 15].

Already Meyer in its well-known book on object-oriented software construction [8] defines the *linguistic modular units principle*, as one of the modularity criteria, according to which "the language should be capable of expressing the structure by dedicated linguistic means for modularization". Unfortunately, the problem with this principle is that it is actually hard to use it as a judgment for the quality of a language, or of a language model regarding modularity, since one can have quite different perceptions of what "the structure" is. By postulating that the structure of any complex system can be expressed by means of two hierarchies expressing *has-a* respectively *is-a* relations, Booch [2] argued that object-oriented programming is *the* approach in coping with complexity.

In [9], Mezini argued that there are other context-dependent relations in the structure of a complex system which are not explicitly taken into account by Booch's structure. If these variations are to be expressed explicitly, the structure of complex systems has at least three rather than two dimensions. The object-oriented model which provides linguistic means for two dimensions only - (a) data types and (b) kind-of variations of a data type - would not fulfill Meyer's criteria anymore. The core idea of AOP [5] can be reformulated as follows: There are concerns in the structure of systems we build such as, distribution, tracing, etc. which cannot be expressed by dedicated linguistic means in object-oriented languages. Ossher and Tarr [15] postulate that, in general, the structure of software is N-dimensional; by supporting only two of these dimensions with linguistic means, existing object-oriented languages badly fail Meyer's modularity criteria mentioned above.

4 Putting it All Together

In this section, we put these two worlds into a common frame of reference. We point out that both worlds are needed in that they address similar problems at different scopes, and discuss how they complement (or otherwise profit from) each other. Note that the discussion is at a very high level of abstraction, aiming at establishing the "big picture" while leaving out of discussion technical details.

The question we consider is the following: "Do we still need new language models for advanced separation of concerns, or are they rendered obsolete by the emergence of component frameworks?" We try to falsify a position that might claim, that the complexity of multi–tier distributed applications can be managed simply be standard object–oriented concepts and a component framework for integration. We strongly believe that we need ASOC language models *and* component frameworks and that they can nicely complement each other. In the following we give some arguments that support our belief.

First, the areas addressed by component frameworks and language models are different. The primary focus of component frameworks is not on composing application logic out of pre-existing building blocks. The main focus is rather on the separation of application logic from cross-application concerns. Indeed, component frameworks also provide a fairly elaborated concept for specifying and connecting components. As pointed out previously, CCM components, e.g., explicitly declare their input (receptables) and output interfaces (facets). In the context of a "component assembly" receptables of one component are mapped to facets of another component, if there is one that matches.

However, the connection at the level of CCM components will generally be at a (very) large-scale. CCM components will in general be distributed over a network. Using the model as a substitute for respective language constructs for expressing and composing modules that encode different application-specific concerns is a very heavy-weight approach, which might damage the performance and scalability of applications built using a component framework.

The idea of explicit required and provided interfaces has been at the core of *Adaptive Plug-n-Play Components* ASOC model [10] even prior to the emergence of CCM. In this model, there is also an equivalent of the assembly process. Given two independent modules, that encode different application-specific concerns, an extra connector construct specifies how to map their interfaces. The model is even more elaborated than the corresponding CCM assembly model so far, in that it does not require conformance of the interfaces being mapped and provides linguistic means to fill the potential interface gap in a succinct way. In the Hyperspace model [15, 13] we also find a sort of "assembly" process in the form of a *hypermodule* definition, where you define the modules to be composed (called *hyperslices*) together with *composition rules* for combining them. Because of these features, the two models and derivates thereof provide a seamless decomposition/composition paradigm outside and inside the boundaries of components. The similarity of AOP "weaving" and CCM assembly has already been indicated in section 2. There remains a need for method support regarding the choice between ASOC and component technology during the design process. This question opens a quite interesting empirical research area.

Second, the view of the world supported by component frameworks is mostly a "closed" one. Component frameworks fail short when flexibility is required for cross-application concerns. Design decisions related to these concerns are locked into the application server - a monolithic block with no support whatsoever for application specific adaptability. Neither do applications have a say on the mechanism to be used for realizing individual cross-application concerns, although they know their needs better than anything else. Nor can the cross-application functionality layer be composed at will out of a set of cross-application concerns realized as off-the-shelf components in order to get the desired behavior.

In the beginning of the paper, we made the analogy between an application server and an operating system. Research on *open implementations* [1] has shown the problems with monolithic "closed" operating systems, TP monitors, etc. and has indicated the need and the power of more flexible systems, providing client applications control over their own implementation strategy via a well-designed auxiliary interface (e.g., [6, 17]). This allows the client to tailor the system's implementation strategy to better suit their needs, effectively making the module more reusable, and the client code simpler.

Component frameworks lack this kind of tailorability. The attribute "framework" is actually not justified when cross-application concerns are considered. There might be well the case that certain applications require some special semantics for the cross-application frameworks. Todays component frameworks provide no support for this kind of late, application-specific binding of mechanisms to cross-application policies. Once bean managed persistency, transaction, etc., or some cross-application functionality is required that is not implemented by the application server, e.g., some sort of tracing is needed, - that is, the application wants to control cross-application aspects - the programmer is basically left on his/her own with the separation of concerns. ASOC models could help to have aspects like persistence, distribution and transaction, be just "usual" off-the-shelf components that can be composed with other components in order to get the desired behavior

5 Summary

The aim of component and ASOC models is the same: Separation of concerns. Nevertheless, their scope is currently different: Component models focus on large scale distributed components, while ASOC models enable a more fine-grained separation on the programming language level. We have sketched some examples that show how both paradigms complement each other and could profit from each other.

However, we think that we have to take a step further. Our vision is that both worlds, components and ASOC, have to converge in order to let the dream of off-the-shelf components become true. In this ideal world, aspects like persistence, distribution and transaction, would be just "usual" off-the-shelf components that can be composed (woven) with other components in order to get the desired behavior.

To achieve this convergence, ASOC approaches need to be improved towards better applicability both to the full range of cross-application and application specific concerns and even more important larger scales. Components on the other hand need to become more flexible with respect to unanticipated behavioral adaptations.

References

[1] R. Barga and C. Pu. Reflection on a legacy transaction processing monitor. In *Reflection '96*, San Francisco, California, 1996.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 1991.

[3] M. Fayad, D. Schmidt, and R. Johnson. *Building Application Frameworks.* Wiley, 1999.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, 1995.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyvaskyla, Finland, 1997. Springer-Verlag.

[6] V. P. Lortz and K. G. Shin. Combining contracts and exemplar-based programming for class hiding and customization. In *Proceedings OOPSLA '94, ACM SIGPLAN Notices*, Oct. 1994. Published as Proceedings OOPSLA '94, ACM SIGPLAN Notices, volume 27, number 10.

[7] M. Mattson, J. Bosch, and M. E. Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10), October 1999.

[8] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

[9] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance.* Kluwer Academic Publishers, 1998.

[10] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, 1998.

[11] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice.* Kluwer, 2000. University of Twente, The Netherlands.

[12] Object Management Group. *CORBA Components Final Submission.* OMG TC Document orbos/99-02-05, 1999.

[13] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717), IBM Thomas J. Watson Research Center, 1999.

[14] Sun Microsystems. *Enterprise JavaBeans 2.0 Specification.* 2000.

[15] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. International Conference on Software Engineering (ICSE 99)*, 1999.

[16] M. VanHilst and D. Notkin. Using role components to implement collaboration-based design. In *Proceedings OOPSLA 96*, 1996.

[17] Y. Yokote, F. Teraoka, A. Mitsuzawa, N. Fujinami, and M. Tokoro. The Muse object architecture: A new operating system structuring concept. In *Operating Systems Review 25(2), April 1991*, 1991.