

Ensembles: Abstractions for A New Class of Design Problem

Kurt Wallnau and Judith Stafford

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213, USA
+1.412.268.{3265,5051}
{kcw,jas}@sei.cmu.edu

ABSTRACT

Trends in component-based software development point to increased use of pre-existing or purchased components. A consequence of this type of development is that systems are being composed of large-grained components over which the developer wields little control. This and other issues related to the use of commercial components has created a new class of design problem that is not addressed by traditional development methods and tools. In this paper we describe this class of design problem, and introduce *Ensemble*, a conceptual language that supports assembling software systems from commercial components.

INTRODUCTION

In recent years, software development using pre-existing or purchased components has become commonplace. This trend has far-reaching effects. Among these effects is that systems are being composed of large-grained components over which the developer wields little control. Moreover, the primary source of software components is the commercial marketplace, and market dynamics add new dimensions of design complexity and risk to software development. Software developers are, to a large extent, unaware of these new dimensions; they continue to approach software development using methods that served them well when constructing systems from “bespoke” components—i.e., components that are custom developed for a particular project.

We have developed Ensemble, a conceptual language that supports development of systems from pre-existing components. Ensembles describe a scope of interactions among components, and comprise a set of components and their interactions. Annotations are associated with the components and interactions in an ensemble. These annotations specify quality attribute

objectives of component interactions, and specify the component properties these objectives depend upon. Ensembles are also annotated and can be related to other ensembles. Ensemble annotations and relationships capture information about the state of a design activity, for example the feasibility of an ensemble, or contingency and repair relationships among ensembles. Thus, ensembles convey information both about a particular design, and about the state of the design process.

THE PROBLEM

Software design is at heart a decision-making process. There are two classes of design decision: *optimization* and *selection*. All software design problems exhibit both classes of decision. Systems developed primarily from custom-built software are dominated by optimization decisions, while systems designed from pre-existing software components are dominated by selection decisions. Current design methods are oriented towards optimization rather than selection.

Optimization

Optimization decisions arise when there are too many design options to itemize. For example, choosing a setting on a sixteen channel stereo equalizer requires an optimization strategy. In this situation, the fundamental design problem is to *generate* a feasible design. In bespoke systems, the designer is free to partition a system into components of arbitrary scope, to define component interfaces in arbitrary ways, and to select arbitrary mechanisms to support interaction of components. This freedom means there are an infinite number of design options, hence optimization.

Software engineering methods today are biased toward optimization decisions. Methods rooted in object-oriented analysis and design define optimization procedures that refine requirements statements (use cases, for example) into a collection of class specifications [8,15]. Component-based extensions of object-oriented methods add nuance to the optimization procedure, but do not fundamentally change the design approach [7]. Methods rooted in software architecture are likewise geared towards optimization, except that, in contrast to

object-oriented methods, architectural design methods are more robust in their treatment of quality attributes such as security, performance, and scalability, and trade-offs among these attributes [5,16,17].

Selection

Selection decisions arise when there is a bounded, and usually small set of *a priori* design options. In this situation, the fundamental design problem is to *select* the option that best satisfies specific design qualities. In systems built from pre-existing software components, the designer is not free to define the scope of components, their interfaces, and their interaction mechanisms, as these decisions have already been made by the component developer. This greatly restricted design freedom leads to the primacy of selection decisions.

Modern software engineering methods are deficient in their treatment of selection procedures. Where techniques are discussed at all, they tend to be based in multi-criteria decision theory [18], and even then these techniques are not well integrated with an overall design process. While multi-criteria analysis is a necessary foundation for any selection procedure, component selection introduces challenges not addressed by multi-criteria procedures alone. For example, multi-criteria procedures assume that selection decisions are independent. This is not the case with software components. Component selection decisions are often strongly interdependent, with one selection decision constraining others. These interdependencies are, in themselves, a principle source of selection criteria.

Commercialization

Today, and for the foreseeable future, the commercial marketplace is the primary source of software components. In fact, software components and the software component marketplace are inextricably linked. Szyperski observed that a component must be defined to fill a market niche so that it can be used in many situations [25]. However, Szyperski's notion of market was largely (although not completely) metaphorical. In contrast, our use of the term *component market* refers to something that demonstrably exists today, complete with component suppliers, component infrastructure providers, third-party component integrators, and, ultimately, consumers.

Ignoring the effects of the marketplace on software engineering would be analogous to ignoring the complicating effects of friction in mechanical engineering. In particular, there are three qualities of commercial software components that together account for a significant share of the challenges posed by software components:

1. Commercial software components are *complex*.

Vendors package complexity to attract a market. However, components can become so complex that even experts do not know all their features. In practice, there will be gaps in engineering knowledge about component features and behavior, and this is a significant source of risk.

2. *Commercial software components are idiosyncratic.* While components may implement standard features, it is innovation that attracts consumers. Innovation results in knowledge that is component-specific. It also results in integration difficulties due to mismatches among innovative (and therefore non-standard) component features.
3. *Commercial software components are unstable.* New features must be introduced to generate sales and preserve product differentiation. Therefore, knowledge about components is not just vendor-specific—it also has a short half life. Moreover, design assumptions based on component features can be fragile and subject to untimely revision.

The combined effect of these component qualities introduces a new realm of challenges for designers, one that clearly outstrips multi-criteria selection procedures. In the following section we introduce component *ensemble* as a fundamental design abstraction. Ensembles expose component dependencies, and shift the emphasis from selecting individual components to selecting sets of components that work together.

ENSEMBLE METAMODEL

In this section we define a metamodel for ensembles. The metamodel is instantiated to produce models of ensembles (hence, *metamodel*). We use the Unified Modeling Language (UML) to define the metamodel. Later, we will also use UML to instantiate the metamodel, i.e., to produce models of ensembles. Other notations are possible, and in particular any architecture description language that supports connectors [2,12,25] will likely be adequate to describe ensembles.

Figure 1 presents the ensemble metamodel. We now discuss its key concepts.

Commercial Software Component

There is no shortage of definitions for software component [3,7,13,20,23]. We adhere to the broad consensus that components are binary implementations with an interface, but add additional criteria that reflect their origin in the component marketplace. A *commercial software component* is:

- an implementation of functionality
- distributed in binary form
- subject to third-party composition
- sold/licensed for profit by a vendor
- sold/licensed in identical (non-customized) form

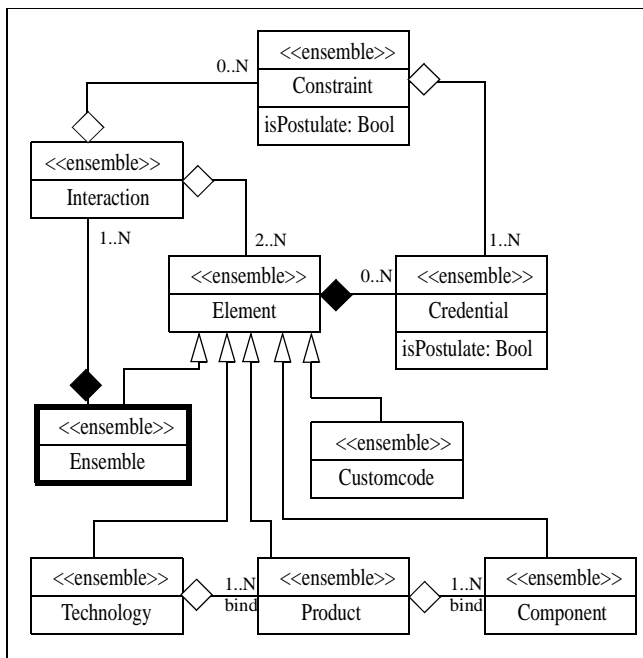


Figure 1: Ensemble Metamodel

Examples of commercial software components are Oracle relational database management system, Microsoft Word, and Netscape Navigator.

Credentials

To date, component vendors have been unwilling to provide complete and accurate specifications of their components—and there is no reason to expect this to change in the near future. Indeed, it may not be practical or possible for vendors to provide a complete component specification. Mary Shaw recognized this possibility, and suggested the idea of *credentials* [21]. Credentials are an open-ended and extensible way of describing the properties of a component. A *credential* is a 3-tuple $\langle \text{Property, Value, Credibility} \rangle$ where:

- *Property* denotes a visible property of a component, for example “encryption method” and “average response time.”
- *Value* denotes a measurement of a property on a particular component, for example “triple-DES encryption” and “50ms average response time.”
- *Credibility* denotes the source of knowledge about a $\langle \text{Property, Value} \rangle$ association, for example “from documentation.”

The list of credentials associated with a given component is arbitrary and extensible. *Credibility* provides a link between the what is known about a component, and how this knowledge was obtained. This link is important for designers to assess design risk, and to identify the need for contingency plans.

Postulate

A credential documents something that is known about a component. However, it is often just as important to document what needs to be discovered. For example, does a particular component support encryption? Perhaps it is known that it supports encryption, but what kind of encryption? In either case, instead of a credential we state a *postulate*. A postulate is structurally analogous to a credential, but is interpreted as a plan for producing a credential. A *postulate* is a 3-tuple $\langle \text{Property, Value, Plan} \rangle$ where:

- *Property* denotes a visible property of a component.
- *Value* denotes a (possibly empty) measurement of a property on a particular component.
- *Plan* denotes how a $\langle \text{Property, Value} \rangle$ association will be obtained.

The value of a property may be unknown or it may be hypothesized. The plan describes a procedure for validating the association of a property with a component, or for obtaining or confirming the value of a property.

Component Classifiers

We have found that, in practice, designers tend to think of components at three distinct levels of abstraction:

1. *Component*: as defined, above.
2. *Product*: a set of components from a single vendor.
3. *Technology*: a set of products in a market niche.

These abstractions reflect how designers make *graduated* selection decisions. For example, the designer may first select relational database *technology* rather than, say, object-oriented technology. Later, the designer may select from among relational database product vendors, for example Sybase, Informix, and Oracle. Still later, the designer may select a particular database *component*. This sequence of selection decisions involves a gradual commitment to the component that is ultimately selected. Different selection criteria may be used, and different levels of credibility may be required, at each step in the decision process.

These abstractions also reflect an implicit type model that is associated with components. In this light, it is often useful to associate postulates and credentials with technologies and products, and not just with components. For example, it might be known from documentation that all Netscape browsers support a particular version of X.509 digital certificates. This suggests an association of a credential such as $\langle \text{digital certificate, X.509v3, documentation} \rangle$ to a product called “Netscape Navigator.” Analogous illustrations can be devised for technologies, and for postulates.

In Figure 1, *Technology*, *Product*, and *Component* specialize *Element*, which possesses an arbitrary

trary number of *Credentials*¹. If a credential is, in fact, postulated, then the value of *isPostulate* is True. *Technology* classifies one or more *Product*, which in turn classifies one or more *Component*. The scope of a credential or postulate associated with a product is all components classified by that product. The scope of a credential or postulate associated with a technology is all products classified by that technology, and all components classified by these products.

Interactions and Constraints

Components are integrated so they can interact, i.e., exchange control and/or data. Component-based design is centered on interactions, not components. There are two reasons for this.

1. Designers have little or no control over the components in a system, but they do have control over how components are integrated.
2. Quality attributes pertain to component interactions as well as components. For example, *confidentiality* is inherently interactional.

While quality attributes may pertain to interactions, the mechanisms that enable interactions (and therefore the quality attributes of these interactions) are supplied by the components. Constraints provide the link between component properties and interaction properties. A *constraint* is a 4-tuple <Property, Value, {Credential}, Credibility>:

- *Property* denotes a property of an interaction.
- *Value* denotes a measurement of a property of an interaction.
- {*Credential*} denotes the set of component properties needed to achieve that property.
- *Credibility* denotes the source of knowledge about a <Property, Value> association, and a description of how {*Credential*} contributes to *Property*.

Constraints are analogous to credentials, except that constraints pertain to interactional properties whereas credentials pertain to component properties. Note that the credibility threshold for interactional properties is independent of the credibility of component properties.

To complete the analogy between constraint and credential, constraints also come in the form of postulates. We include a definition of postulated constraint for completeness, although its similarity with postulated credentials makes further elaboration superfluous. A *postulated constraint* is a 4-tuple <Property, Value, {Credential}, Plan> where:

- *Property* denotes a property of an interaction.
- *Value* denotes a measurement of a property of an interaction.
- {*Credential*} denotes the set of component properties needed to achieve *Property*.
- *Credibility* denotes how a <Property, Value> association will be obtained.

Ensemble

An *ensemble* is a scope of interactions among technologies, products, and components that cooperate to provide aggregate behavior. The most important point to note in this definition (and in the metamodel) is the central role played by interactions. A component defines a scope of functionality, while an ensemble defines a scope of interactions.

Ensembles are the nexus of component-based design. They specify which components are involved, how they interact, and what component features are essential to those interactions. Ensembles demarcate what is known about components and their interactions and what remains to be discovered. There is, therefore, a direct connection between ensembles and an underlying design process. Ensembles do not merely document design decisions already made, but also help to structure and direct the design process itself. Ensembles are to component-based designers what navigation charts were to seafaring explorers.

APPLICATION

The ensemble metamodel, and the design processes it supports, evolved from interviews with designers, case studies, and trial application on industrial-scale projects during the period 1997-2001. For reasons of space we describe only two applications of ensemble: attribute-specific projections, and remedy management. Additional topics, along with a detailed case study, can be found in [24].

Design Documentation

Commercial components are designed to work in many application contexts. Only a small proportion of component features are typically used in any particular application. Prior to selecting a component, a designer must select which component features are pertinent. Often, the features which are pertinent are those that concern component interaction. Documenting an ensemble (or, for that matter, a component selection decision), requires that these pertinent features, and their relationship to interactions, be made explicit. We do this by means of ensemble *projections*.

Briefly, a projection is a model of a subset of the components and interactions in an ensemble. A useful heuristic is to create a separate projection for each ensemble interaction that bears a quality attribute

1. To conserve space in the figure we have not included the property, value, and credibility (or plan) attributes.

requirement (i.e., a constraint). Each projection, then, will document how a pattern of component interactions satisfies a quality attribute, and which component features are necessary for that pattern of interaction. The complete documentation of an ensemble will be found (among other artifacts) in the aggregation of these projections.

Figure 2 illustrates a projection described as a stylized UML collaboration diagram. This projection is focused on the *authorization* attribute of the *Applet* ensemble. (We know this by virtue of ensemble naming conventions summarized in Table 1.) The *authorization* qual-

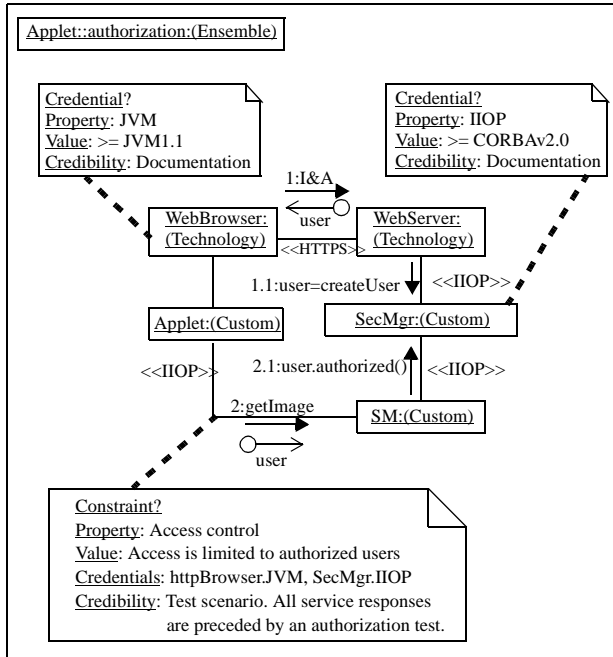


Figure 2: Attribute-Specific Ensemble Projection

ity attribute is described in the constraint at the bottom of the figure, which defines a *property* called “access control,” whose *value* is described as a restriction on an (in this case, implied) use case. The *credibility* states both what is needed to demonstrate confidentiality (a test scenario), and the criterion to be satisfied by that test (the authorization test, found in interaction 2.1:user.authorized() in the projection).

Table 1: UML Conventions for Ensemble Projections

Metamodel	Modeling Convention
Ensemble	Object instance of the metamodel class <i>Ensemble</i> . The name of the instance designates the subset of interactions modeled. If all interactions are modeled, the name of the instance is the name of the ensemble.

Table 1: UML Conventions for Ensemble Projections

Metamodel	Modeling Convention
Technology, Product, Component, Customcode	Object instances of the metamodel classes <i>Technology</i> , <i>Product</i> , <i>Component</i> , and <i>Customcode</i> , respectively.
Element	Never instantiated. <i>Element</i> is an abstract superclass in the metamodel.
Interaction	Not instantiated for binary interactions. Instantiated with distinguishing iconic representation for N-way interactions.
Credential	Structured annotation beginning with the keyword <i>Credential</i> , and including the keywords <i>Property</i> , <i>Value</i> , and <i>Credibility</i> .
Postulated Credential	Structured annotation beginning with the keyword <i>Credential?</i> , and including the keywords <i>Property</i> , <i>Value</i> , and <i>Credibility</i> .
Constraint	Structured annotation beginning with the keyword <i>Constraint</i> , and including the keywords <i>Property</i> , <i>Value</i> , <i>Credentials</i> , and <i>Credibility</i> .
Postulated Constraint	Structured annotation beginning with the keyword <i>Constraint?</i> , and including the keywords <i>Property</i> , <i>Value</i> , <i>Credentials</i> , and <i>Credibility</i> .

The credentials and constraint in Figure 2 a marked with the “?” suffix, indicating that they are postulates. This projection, then, models a design conjecture that must be validated. A demonstration of the access control constraint is required to validate, at least in part, the feasibility of the Applet ensemble.

Remedy Management

Validation of feasibility is of fundamental importance in component-based design precisely because so much is unknown or unspecified about how components behave alone or in an assembly. Given this uncertainty, it is hardly surprising that ensembles often fail to satisfy one or more constraint. However, all is not lost in such circumstances. In fact, such failures simply initiate design activities that are intended to discover one or more *remedies* for the failure.

The search for ensemble deficiencies and their remedies is a recurring if not central theme in component-based design. Remedies are, conceptually, a binary relation from ensembles to ensembles. One end of the relation is a deficient ensemble; the other end is an ensemble that, in some way, remedies the deficiency. We model these relations as UML associations¹.

1. In the interest of space we define remedies and so forth directly in UML terms. It must be noted that these definitions do not in any way depend upon UML.

Software architecture is not just about representation; it is also concerned with the relationship between the quality attributes of a system (reliability, performance, security, etc.) and its structure, and exploiting this structure to support analysis of quality attributes. Architectural styles have been the traditional link between structure and quality attributes. Styles are defined by a set of component types and their allowable patterns of interaction [1,4]. Ensembles also describe patterns of interactions among types of components, and link these patterns to the achievement of quality attributes.

Design Patterns

Even more important than deep expertise in any one technology is a general understanding of how several technologies can be combined. Good designers are familiar with a variety of patterns of component integration, and know the strengths and weaknesses of each. This is analogous to the idea of design patterns that emerged from the object-oriented (OO) research community [6,9]. The basic motivation of design patterns is to reuse proven patterns of interaction, whether at the computer program level or system design level.

One way ensembles differ from design patterns, and from architectural styles (another form of design pattern) is the source of the pattern. For ensembles, the source is the commercial component marketplace. Ensembles are “design patterns *du jour*.” In contrast, architectural styles and object-oriented design patterns are more stable, and, therefore, fundamental. Thus, there are different incentives for documenting ensembles than exist for architectural styles and OO patterns.

Component-Based Methods

Cheesman and Daniels’ base their method, *UML Components*, on the premise that the value of component-based development lies not in reuse of components, but in the development of highly adaptable systems [7]. Their method therefore focuses on how component specifications can be systematically derived from high-level models, and how component dependencies can be both minimized and made explicit. Their method does not, however, offer any guidance on designing systems from pre-existing components.

Like us, and unlike Cheesman and Daniels, Herzum and Sims believe that the essence of component-based development lies in the use of pre-existing components [14]. Unlike us, however, they do not believe such components exist. Instead, a *business component factory* must begin with bespoke components. As a result, Herzum and Sims focus on the reuse processes that will, in some indefinite future, result in a population of components from which systems can be constructed.

D’Souza and Wills’ *Catalysis* [8] and Rational’s *Unified Software Development Process* (RUP) [15], also focus on specifying bespoke components. Catalysis’ roots in formal approaches to object-oriented design is evident in its emphasis on rigorous specification rather than exploration and discovery. On the other hand, Catalysis recognizes the importance of treating patterns of interaction among components as first class design abstractions, and its notion of *kit* is the bespoke component analogue of ensembles. RUP also has its roots in object-oriented development, but relegates component abstractions to issues of software packaging and deployment, rather than fundamental design.

FUTURE WORK

There has been recent work to make UML more suitable as an ADL. At a minimum, this means addressing several aspects of UML that make it deficient as an ADL [11,19]. Extensions to the UML metamodel have been proposed to remedy these deficiencies, and at least one ADL, ACME [12], has been mapped to the revised UML metamodel. We plan to study using ACME, with its UML mappings, as a formalism for ensemble projections. This will simultaneously update our use of UML, and help us to understand the applicability of ADLs in general to component-based design.

We are investigating ways to bundle *quality attribute reasoning frameworks* with software component technologies, so that the quality attributes of component assemblies can be predicted from the certified properties of the components [22]¹. Work on attribute-based architectural style (ABAS) [17] and, more recently, on *quality attribute mechanisms*, reinforces the central role of patterns of component interaction on achieving quality attributes. This suggests a natural link between ensembles and attribute reasoning frameworks. We plan to explore this link more fully in the coming months.

CONCLUSIONS

Assembling systems from pre-existing software components results in a new class of design problem, one based in selection rather than optimization. The software component marketplace is a confounding influence on this new class of design problem. Designers using commercial components are faced with limited and rapidly wasting knowledge of component behavior, integration mismatches arising from non-standard component features, and mutually-constraining selection decisions. Since the commercial market will continue to be the primary source of pre-existing components, these confounding influences must be addressed by software engineering methods. Current

1. See also <http://www.sei.cmu.edu/pacc/index.html>.

software methods have not caught up with the clearly recognizable shift in emphasis from bespoke components to pre-existing components.

We have described Ensemble, a conceptual language for meeting the challenges of commercial component-based design. Ensembles support design with partial and unstable knowledge of component properties. Ensembles help document designs by making explicit the links between component properties and component interaction. Ensembles also support graduated and incremental selection of technologies, products, and components. We aim, in future work, to augment the descriptive focus of ensembles with analysis and prediction of the quality attributes of component assemblies.

ACKNOWLEDGEMENTS

This work is sponsored by the US Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defence.

REFERENCES

1. G.D. Abowd, R. Allen and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319-364, October 1995
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213-249, July 1997.
3. F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord and K. Wallnau. Volume II: Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008. Software Engineering Institute, Pittsburgh, Pennsylvania, May 2000.
4. L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1998.
5. F. Bachmann, L. Bass, G. Chastek, P. Donahoe and F. Peruzzi, *The Attribute-Based Design Method*, Technical Report CMU/SEI-2000-TR-001, Software Engineering Institute, Pittsburgh, Pennsylvania.
6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture*, John Wiley & Sons, West Sussex, England, 1996.
7. J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, Boston, Massachusetts, 2000.
8. D. D'Souza, A. Cameron Wills, *Objects, Components and Frameworks with UML*, Addison Wesley, Reading, Massachusetts, 1998.
9. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, Addison-Wesley, Reading, Massachusetts, 1994.
10. D. Garlan, R. Allen and J. Ockerbloom. Architectural Mismatch: Why Reuse is So Hard. *IEEE Software*, 12(6):17-26, November 1995.
11. D. Garlan and A. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations, Third International Conference on the Unified Modeling Language (UML2000), pages 498-512. University of York, UK, 2-6, October 2000.
12. D. Garlan, R. Monroe and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169-183. IBM Center for Advanced Studies, November 1997.
13. G.T. Heineman and W.T. Councill (eds.). *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.
14. P. Herzum and O. Sims, *Business Component Factory*, OMG Press, John Wiley & Sons, Inc., 2000.
15. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, Massachusetts, 1998.
16. R. Kazman, M. Klein and P. Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Pittsburgh, Pennsylvania, August 2000.
17. R. Klein and R. Kazman. Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Pittsburgh, Pennsylvania, December 1999.
18. J. Kontio, *OTSO: A Systematic Process for Reusable Software Component Selection*, Technical Report CS-TR-3378, UMIACS-TR-96-63, University of Maryland College Park, MD 20742, December 1995.
19. N. Medvidovic and D. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. *Proc. First Working IFIP Conf. on Software Architecture*, pp. 161-182. San Antonio, Texas, February 1999.
20. J. Rumbaugh, G. Booch and I. Jacobson. *The Unified Modeling Language Reference Manual (UML)*. Addison Wesley Longman, Inc. December 1998.
21. M. Shaw. Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996.
22. J. Stafford and K. Wallnau, "Predictable Assembly from Certifiable Components," submitted to the IEEE/IFIP Working International Conference on Software Architecture, Amsterdam, The Netherlands, 2001.
23. C. Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, Boston, Massachusetts and ACM Press, 1998.
24. K. Wallnau, S. Hissam and R. Seacord, *Building Systems from Commercial Components*, Addison Wesley, Reading Massachusetts, to appear June 2001.
25. G. Zelesnik. UniCon Reference Manual. Technical Report CMU-CS-97-TBD, Carnegie Mellon University, 1997.