

NRC/SWT
Kimmo Raatikainen

5.1.2004

Background Material for WWI Architecture Cross-Issue

This document collects background readings on architectures, particularly on software architectures, and on their descriptions. The material is largely cut-and-past from various IEEE publications. Therefore, the document is only for internal use in WWI as background reading material.

The objective of this document is to provide a common terminology for discussions and decisions in the WWI Cross-Issue Workshop, Helsinki, January 22-23, 2004. The document does not promote any material in the document

Document History

Issue	Date	Editor	Comments
0.1	5.1.2004	Kimmo Raatikainen	Initial draft for NRC comments

NRC/SWT
Kimmo Raatikainen

5.1.2004

Background Material for WWI Architecture Cross-Issue

- 1. Introduction 3
- 2. “Software Architecture: Introducing IEEE Standard 1471,” IEEE Computer, April 2001, pp. 107-109. © 2001 IEEE 4
 - 2.1 IEEE 1471’s Architecture Description Requirements 5
- 3. Martin Fowler, “Who Needs an Architect?” IEEE Software, September/October 2003, pp. 11—13. © 2003 IEEE 6
- 4. Steve Vinoski, “Do You Know Where Your Architecture Is?” IEEE Internet Computing, September/October 2003, pp. 86—88. © 2003 IEEE 9
- 5. Nenad Medvidovic and Richard N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages,” IEEE Transactions on Software Engineering, Vol. 26, No. 1, January 2000, pp. 70—93. © 2000 IEEE 11
 - 5.1 ADL Classification and Comparison Framework 15
 - 5.1.1 Modeling Components 15
 - 5.1.2 Modeling Connectors 16
 - 5.1.3 Modeling Configurations..... 16
 - 5.1.4 Tool Support for Architectural Description..... 17
 - 5.2 Differentiating ADLs from Other Languages..... 17
 - 5.3 Conclusions 18
- 6. Debate on Model-Driven Architecture 19
 - 6.1 Axel Uhl, “Model Driven Architecture Is Ready for Prime Time”, IEEE Software, September/October 2003, p. 70&72 © 2003 IEEE..... 19
 - 6.2 Scott W. Ambler, “Agile Model Driven Development Is Good Enough,” IEEE Software, September/October 2003, pp. 71—73 © 2003 IEEE 20
 - 6.3 Cross-comments by Uhl and Ambler, IEEE Software, September/October 2003, p. 73 @ 2003 IEEE 21
 - 6.3.1 Axel Responds 21
 - 6.3.2 Scott Responds 21
- 7. Embedded Arcticels 22
 - 7.1 IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. 22
 - 7.2 Nenad Medvidovic, Marija Mikic-Rakic, Nikunj R. Mehta, and Sam Malek, “Software Architectural Support for Handheld Computing,” IEEE Computer, September 2003, pp. 66—73. .. 22
 - 7.3 Morgan Björkander and Cris Kobryn, “Architecting Systems with UML 2.0,” IEEE Software, July/August 2003, pp. 57-61. 23
- 8. Literature 23

NRC/SWT
Kimmo Raatikainen

5.1.2004

1. INTRODUCTION

Architectures are generally described in terms of components (computational elements), connectors (interaction elements), and their configurations. An architectural style further defines a vocabulary of component and connector types as well as a set of constraints on combining instances of those types in a software system. Examples of styles include black-board, C2, client-server, pipe and filter, and push-based. Selecting an appropriate architectural style is a key determinant of a software system's success.

Software architectures provide design-level models and guidelines for composing software systems. However, to be useful in a development setting, these models and guidelines require support for implementation and evolution.

The Software Architecture community still debates on proper architecture description languages. For example, most of the community does not—at least a couple of years ago—regard UML suitable for describing architectures. This is quite surprising since UML is the lingua franca of software modeling. More recently OMG has introduced Model Driven Architecture (MDA) to raise the abstraction level. In MDA, Platform Independent Models (PIMs) are used to provide a resource model including properties, capabilities and relations.¹

The IEEE has published a standard (IEEE 1471-2000) that gives recommendations what an architectural description must contain but leaves the description style open.

The rest of this document summarizes the following articles:

1. "Software Architecture: Introducing IEEE Standard 1471," IEEE Computer, April 2001, pp. 107-109. (Summary of IEEE Standard 1471-2000)
2. Martin Fowler, "Who Needs an Architect?" IEEE Software, September/October 2003, pp. 11—13. (A little bit provocative column.)
3. Steve Vinoski, "Do You Know Where Your Architecture Is?" IEEE Internet Computing, September/October 2003, pp. 86—88. (A little provocative column.)
4. Nenad Medvidovic and Richard N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," IEEE Transactions on Software Engineering, Vol. 26, No. 1, January 2000, pp. 70—93. (Presents a taxonomy for Architecture Description Languages.)
5. Axel Uhl, "Model Driven Architecture Is Ready for Prime Time", IEEE Software, September/October 2003, p. 70&72.
6. Scott W. Ambler, "Agile Model Driven Development Is Good Enough," IEEE Software, September/October 2003, pp. 71—73.
7. Cross-comments by Uhl and Ambler, IEEE Software, September/October 2003, p. 73.

The final section embeds (in pdf-format) the following articles:

¹ MDA Guide Version 1.0, OMG Document Number:omg/2003-05-01, 1st May 2003. See also IEEE Software Special Issue on Model-Driven Development, September/October 2003.

NRC/SWT
Kimmo Raatikainen

5.1.2004

- IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems
- Nenad Medvidovic, Marija Mikic-Rakic, Nikunj R. Mehta, and Sam Malek, "Software Architectural Support for Handheld Computing," IEEE Computer, September 2003, pp. 66—73. (An example of architecture description that uses UML.)
- Morgan Björkander and Cris Kobryn, "Architecting Systems with UML 2.0," IEEE Software, July/August 2003, pp. 57-61. (A brief introduction to UML 2.0)

The literature section suggests some additional readings.

2. "SOFTWARE ARCHITECTURE: INTRODUCING IEEE STANDARD 1471," IEEE COMPUTER, APRIL 2001, PP. 107-109. © 2001 IEEE

IEEE Standard 1471 identifies sound practices to establish a framework and vocabulary for software architecture concepts.

Architecture is used in various contexts to mean the instruction set of a central processor unit, the highest-level software modules in a large software system, or the overall structure of a business's information technology systems.

In 2000, the Computer Society approved IEEE Standard 1471, which documents a consensus on good architectural description practices.

Five core concepts and relationships provide the foundation for the approved IEEE 1471 version:

- Every system has an architecture, but an architecture is not a system.
- An architecture and an architecture description are not the same thing.
- Architecture standards, descriptions, and development processes can differ and be developed separately.
- Architecture descriptions are inherently multiviewed.
- Separating the concept of an object's view from its specification is an effective way to write architecture description standards.

IEEE 1471 uses civil architecture as a metaphor for the design of software-intensive systems. ... The architect is the client's trusted agent in coordinating all aspects of a building project, including the integration of structural, business, legal, and aesthetic concerns.

While the architect's role is broad, it does not extend to all of the building project's details. The architect's domain is the essential core, the aspects of the project that define usage, value, cost, and risk to within the client's tolerances.

IEEE 1471 defines architecture as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution." This definition incorporates the idea that there is a difference between an architectural description and an architecture. An architectural

NRC/SWT
Kimmo Raatikainen

5.1.2004

description is a concrete artifact, but an architecture is a concept of a system. An architecture embodies a system's fundamental aspects.

IEEE 1471 places normative requirements on architectural descriptions—the documents that describe a system's architecture. It does not standardize a system's architecture or the process for developing it. In this way, IEEE 1471 is more like a blueprint standard than a building code. It defines the equivalent of drawing and symbology conventions, although it does not define the full range of drawings needed to provide a description of any particular system.

IEEE 1471's most important elements are:

- a set of definitions for key terms such as architectural description, architectural view, and architectural viewpoint;
- a separation of the concepts of architecture and architecture description to facilitate establishing standards for describing architectures (analogous to blueprint standards) and standards for constructing systems (analogous to building codes or zoning laws); and
- content requirements for describing a system's architecture.

Because the architecture description concept has expanded to include budgets, business cases, and protocol definitions as well as physical component structure, it is evident that a single language is insufficient.

In IEEE 1471, a view is a collection of models that represent one aspect of an entire system. A view applies to only one system, not to generalizations across many systems. The standard introduces the concept of viewpoints to capture common descriptive frameworks across many systems. Viewpoints are the vehicles for writing reusable, domain-specific architecture description standards. They establish the languages or notations used to create a view, the conventions for interpreting it, and any associated analytic techniques that might be used with the view.

An architecture description must define the viewpoint for each view it contains. This concept is consistent with emerging practices in a number of software fields in which viewpoints define representation standards for specific stakeholders, such as the International Organization for Standardization's Reference Model for Open Distributed Processing².

2.1 IEEE 1471's Architecture Description Requirements

The new standard defines an architectural description's content requirements in terms of its elements. First, an architecture description must specify the system's stakeholders and identify their architectural concerns. Familiar concerns are

- Functionality. What does the system need to do?
- Performance. How will the system behave under heavy loads?

² ISO/IEC 10746 –2:1996, Information technology □ Open distributed processing □ Reference model: Foundations. ISO/IEC 10746 –3:1996, Information technology □ Open distributed processing □ Reference model: Architecture.

NRC/SWT
Kimmo Raatikainen

5.1.2004

- Security. Can the system adequately protect user information?
- Feasibility. Can we implement the system?

Second, an architecture description must be organized into one or more views of the system's architecture. A view is not just any glimpse of the system—it must address identified stakeholders' concerns and it must be well formed. To provide a minimal completeness check, at least one view must address each identified architectural concern.

Finally, an architecture description must provide the rationale for making key architectural decisions. This can take the form of presentations of trade-offs the architects considered, alternatives they didn't choose, or other analyses that led them to choose the architecture that the architecture description documents.

3. MARTIN FOWLER, "WHO NEEDS AN ARCHITECT?" IEEE SOFTWARE, SEPTEMBER/OCTOBER 2003, PP. 11—13. © 2003 IEEE

... even by our industry's standards, "architect" and "architecture" are terribly overloaded words. For many, the term "software architect" fits perfectly with the smug, controlling image at the end of *Matrix Reloaded*.

I define **architecture** as a word we use when we want to talk about design but want to puff it up to make it sound important. (Yes, you can imagine a similar phenomenon for architect.) However, as so often occurs, inside the blighted cynicism is a pinch of truth. Understanding came to me after reading a posting from Ralph Johnson on an Extreme Programming mailing list. It's so good I'll quote it all.

A previous posting said:

The RUP, working off the IEEE definition, defines architecture as "the highest level concept of a system in its environment. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces."

Johnson responded:

I was a reviewer on the IEEE standard that used that, and I argued uselessly that this was clearly a completely bogus definition. There is no highest level concept of a system. Customers have a different concept than developers. Customers do not care at all about the structure of significant components. So, perhaps an architecture is the highest level concept that developers have of a system in its environment. Let's forget the developers who just understand their little piece. Architecture is the highest level concept of the expert developers. What makes a component significant? It is significant because the expert developers say so.

So, a better definition would be "In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called 'architecture.' This understanding includes

NRC/SWT
Kimmo Raatikainen

5.1.2004

how the system is divided into components and how the components interact through interfaces. These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developers."

This would be a better definition because it makes clear that architecture is a social construct (well, software is too, but architecture is even more so) because it doesn't just depend on the software, but on what part of the software is considered important by group consensus.

There is another style of definition of architecture which is something like "architecture is the set of design decisions that must be made early in a project." I complain about that one, too, saying that architecture is the decisions that you wish you could get right early in a project, but that you are not necessarily more likely to get them right than any other.

Anyway, by this second definition, programming language would be part of the architecture for most projects. By the first, it wouldn't be.

Whether something is part of the architecture is entirely based on whether the developers think it is important. People who build "enterprise applications" tend to think that persistence is crucial. When they start to draw their architecture, they start with three layers. They will mention "and we use Oracle for our database and have our own persistence layer to map objects onto it." But a medical imaging application might include Oracle without it being considered part of the architecture. That is because most of the complication is in analyzing the images, not in storing them. Fetching and storing images is done by one little part of the application and most of the developers ignore it.

So, this makes it hard to tell people how to describe their architecture. "Tell us what is important." Architecture is about the important stuff. Whatever that is.

So if architecture is the important stuff, then the architect is the person (or people) who worries about the important stuff.

Architectus Reloadus is the person who makes all the important decisions. The architect does this because a single mind is needed to ensure a system's conceptual integrity, and perhaps because the architect doesn't think that the team members are sufficiently skilled to make those decisions. Often, such decisions must be made early on so that everyone else has a plan to follow.

Architectus Oryzus is a different kind of animal (if you can't guess, try www.nd.edu/~archives/latqgramm.htm). This kind of architect must be very aware of what's going on in the project, looking out for important issues and tackling them before they become a serious problem.

NRC/SWT
Kimmo Raatikainen

5.1.2004

In many ways, the most important activity of *Architectus Oryzus* is to mentor the development team, to raise their level so that they can take on more complex issues.

Architectus Reloadus is too common it's based on a flawed metaphor (see <http://martinfozler.com/bliki/BuildingArchitect.html>).

Remember Johnson's secondary definition: "Architecture is the decisions that you wish you could get right early in a project." Why do people feel the need to get some things right early in the project? The answer, of course, is because they perceive those things as hard to change. So you might end up defining architecture as "things that people perceive as hard to change."

At a fascinating talk at the XP (Extreme Programming) 2002 conference (<http://martinfozler.com/articles/xp2002.html>), Enrico Zaninotto, an economist, analyzed the underlying thinking behind agile ideas in manufacturing and software development. ... He saw agile methods, in manufacturing and software development, as a shift that seeks to contain complexity by reducing irreversibility—as opposed to tackling other complexity drivers. I think that one of an architect's most important tasks is to remove architecture by finding ways to eliminate irreversibility in software designs.

Here's Johnson again, this time in response to a draft of this article:

One of the differences between building architecture and software architecture is that a lot of decisions about a building are hard to change. It is hard to go back and change your basement, though it is possible.

There is no theoretical reason that anything is hard to change about software. If you pick any one aspect of software then you can make it easy to change, but we don't know how to make everything easy to change. Making something easy to change makes the overall system a little more complex, and making everything easy to change makes the entire system very complex. Complexity is what makes software hard to change. That, and duplication.

My reservation of Aspect-Oriented Programming is that we already have fairly good techniques for separating aspects of programs, and we don't use them. I don't think the real problem will be solved by making better techniques for separating aspects. We don't know what should be the aspects that need separating, and we don't know when it is worth separating them and when it is not.

Software is not limited by physics, like buildings are. It is limited by imagination, by design, by organization. In short, it is limited by properties of people, not by properties of the world. "We have met the enemy, and he is us."

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.

NRC/SWT
Kimmo Raatikainen

5.1.2004

4. STEVE VINOSKI, “DO YOU KNOW WHERE YOUR ARCHITECTURE IS?” IEEE INTERNET COMPUTING, SEPTEMBER/OCTOBER 2003, PP. 86—88. © 2003 IEEE

Life as a chief software architect is rarely fun. My job was to try to make sure the software underlying our products was flexible, fast, extensible, robust, consistent, cohesive, current, and devoid of duplication — and that it incorporated sound development practices. Sounds pretty obvious, as all software developers, development managers, and product managers strive for these qualities, right? Unfortunately, our old friend, the not-invented-here (NIH) syndrome, runs rampant throughout our industry. Sometimes, even when you put an architecture in place, managers and developers can still find ways to ignore it. I wish I had a dollar for every time I heard a variation of, “I don’t have time to make this software conform to our product architecture; I have to get it out the door now!” Such NIH can be the result of ignorance, hubris, or outright defiance, but whatever the cause, the end result is the same: nonexistent or incoherent software architecture.

Confusion over architecture, while not uncommon in software development in general, seems prevalent in middleware. I attribute this to the fact that middleware systems are typically distributed and heterogeneous. In general, distributed systems are difficult to design, implement, debug, and maintain. When you mix in multiples of hardware platforms, operating systems, protocols, applications, and vendors, the complexity can rise to the point where nobody really understands the whole system.

Treating technology as architecture just asks for trouble. Like all technologies and specifications, <X> eventually will fall out of favor, which will make this customer’s “architecture” need an upgrade. But given the customer’s penchant for technology, he is likely to switch to something new well before <X> heads to the middleware retirement home. Regardless of when he switches, the cost will not be trivial

Given the cost of redesigning a system whenever the underlying technology changes, I wonder why designers paint themselves into such corners. Conspiracy theory might promote the idea that they do it for job security, ... A lot of it is due to a simple lack of abstraction. Some designers can’t seem to disassociate the problems far enough away from the technologies they use to solve them. Others don’t feel they have the time to work out what the appropriate abstractions should be. Indeed, the “Get out of my way, I have to get it done yesterday” attitude ... pervades the IT industry. Did some ancient middleware prophet, glimpsing a future full of brittle middleware systems, coin the phrase “haste makes waste”?

Architecture specifies not only what the system is, but also what it isn’t. The more an architecture consists of abstractions, rules, and constraints that dictate what the system allows, while avoiding specific “technologies du jour,” the better the chance that it will age gracefully. I’ve helped create middleware architecture definitions that have evolved gracefully, for example, using two simple practices. The first is to write detailed definitions for important system abstractions, or metaphors, and make sure that all team members are familiar with them. The second is to write key internal system interfaces in Corba IDL.

Writing at the abstraction level IDL affords (rather than directly in Java or C++) lets me reuse interfaces across languages and express key service interface aspects without bogging down in implementation details.

NRC/SWT
Kimmo Raatikainen

5.1.2004

Poor abstraction skills can be particularly troubling in the context of service-oriented architectures.³ Developing an SOA requires that you first identify service abstractions, but this reverses the typical approach of writing applications first and then writing the specific services that application needs. With an SOA, the goal is to put appropriately abstracted services in place for reuse by numerous applications, rather than tightly coupling the services to a single application.

During my tenure as chief architect, my employer adopted extreme programming (XP).⁴

XP, and other agile software development approaches that have followed it (see www.agilealliance.org/home/), promote iterative development with a heavy focus on continuous code review, system and unit testing, and customer interaction. The reasoning behind these approaches is that software requirements are usually very fluid. This fluidity means that trying to draft requirements before writing any code only delays the code modifications that arise from the inevitable changes in those requirements, thus prolonging the project and ultimately raising its cost.

Soon after we adopted XP, a backlash arose from various camps. They complained that XP threw architecture and design out the window and replaced it with “cowboy programming.” They said that XP allowed developers the freedom to write whatever they wanted while providing them with a “get out of jail free” card — also known as refactoring⁵— should the resulting system not work as expected. Refactoring is intended to be a methodical way of improving software design and implementation without adversely affecting its external interface or usage, but it’s unfortunately often simply taken as an excuse to rewrite and reinvent. While XP certainly doesn’t advocate such nonsense, it can facilitate the “lone gunslinger” mentality, especially without strong communication among teammates. XP and agile methods count on each team member knowing what the other is doing.

The camps that resented our adoption of XP might have preferred if we had adopted the model-driven architecture approach.⁶ MDA, which follows in the traditions of structured programming and object-oriented programming, favors up-front design. Unlike these approaches, however, MDA treats code as something that tools generate from design models, rather than something that humans write. The arguments that advocates make for MDA sound logical enough. ... According to their argument, the natural progression is away from low-level programming, where you explicitly write code, toward high-level development through creating models of the system, leaving the “programming” to code-generation tools. These efforts include the use of analysis patterns⁷ and metadata to create a platform-independent model (PIM), from which the tools generate code — called a platform-specific model (PSM) — for a specific platform or middleware system.

MDA’s reliance on tools (which on the surface seems like a positive feature) might have deleterious practical effects. Middleware users know that standards help minimize lock-in to proprietary software and switching costs between standards-conforming products. Used correctly, standards also let organizations formed by mergers or acquisitions swiftly

³ S. Vinoski, “Service Discovery 101,” IEEE Internet Computing, vol. 7, no. 1, 2003, pp. 69–71.

⁴ K. Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley Longman, 1999. C. Poole and J.W. Huisman, “Using Extreme Programming in a Maintenance Environment,” IEEE Software, vol. 18, no. 6, 2001, pp. 42–50.

⁵ M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, 1999.

⁶ D. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003.

⁷ M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

NRC/SWT
Kimmo Raatikainen

5.1.2004

integrate their software systems, rather than having to scrap certain systems in their entirety and replace them. Unfortunately, the tools that MDA requires easily could shift vendor lock-in and ubiquity requirements from the middleware itself to the tools that create the middleware. It's true that tool standards under development might prevent this problem but, generally, it takes time for such standards to mature to the point of letting different vendors' tools interoperate cleanly.

XP and MDA advocates might not agree, but one thing the two approaches have in common is the idea of a metaphor. In XP terms, a metaphor is a word or short phrase that captures a project's central idea. XP metaphors are loosely similar to the names of the key entities in an MDA model. XP might use words to express metaphors whereas MDA uses modeling languages and diagrams, but the intentions are identical: both want to communicate the system architecture's key elements as succinctly, yet meaningfully, as possible.

Personally, I lean more toward XP than MDA. ... While I believe that someday we'll realize this goal of moving to that higher level of abstraction, I think it will be a good while before we get there. Mapping from one level of abstraction to another can be difficult in practice.⁸ ... We have a lot of systems to develop before that day comes, however, and until MDA is ready for prime time, I'll stick with the iterations, shared code, close customer involvement, and refactoring practices that agile methods like XP promote.

The two practices I described earlier — clearly defining system metaphors and defining interfaces or contracts using an abstract language like IDL — fit both XP and MDA. Of course your mileage might vary, but I found that when open and active communication among team members surrounds these practices, this approach easily avoids the need for wordy architecture documents or specialized tools to draw and store Unified Modeling Language diagrams.

As of mid-2003, I am no longer a chief architect. ... Perhaps this means that I finally get to sit on the other side of the NIH fence for a change?

Steve Vinoski is chief engineer of product innovation for IONA Technologies. He's been involved in middleware for 15 years. Vinoski is the coauthor of Advanced Corba Programming with C++ (Addison Wesley Longman, 1999), and he has helped develop middleware standards for the OMG and W3C. Contact him at vinoski@ieee.org.

5. NENAD MEDVIDOVIC AND RICHARD N. TAYLOR, "A CLASSIFICATION AND COMPARISON FRAMEWORK FOR SOFTWARE ARCHITECTURE DESCRIPTION LANGUAGES," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 26, NO. 1, JANUARY 2000, PP. 70—93. © 2000 IEEE

Abstract—Software architectures shift the focus of developers from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. Architecture description languages (ADLs) have been proposed as modeling notations to support architecture-based development. There is, however, little consensus in the research community on what is an ADL, what aspects of an architecture should be modeled in an ADL, and which of several possible ADLs is best suited for a particular problem. Furthermore, the distinction is rarely made between ADLs on one hand and formal specification, module interconnection, simulation, and programming languages on

⁸ S.Vinoski, "It's Just a Mapping Problem," IEEE Internet Computing, vol. 7, no. 3, 2003, pp. 88–90.

NRC/SWT
Kimmo Raatikainen

5.1.2004

the other. This paper attempts to provide an answer to these questions. It motivates and presents a definition and a classification framework for ADLs. The utility of the definition is demonstrated by using it to differentiate ADLs from other modeling notations. The framework is used to classify and compare several existing ADLs, enabling us, in the process, to identify key properties of ADLs. The comparison highlights areas where existing ADLs provide extensive support and those in which they are deficient, suggesting a research agenda for the future.

Software architecture research is directed at reducing costs of developing applications and increasing the potential for commonality between different members of a closely related product family⁹. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements (software components and connectors) and their overall interconnection structure. To support architecture-based development, formal modeling notations and analysis and development tools that operate on architectural specifications are needed. Architecture description languages (ADLs) and their accompanying toolsets have been proposed as the answer. Loosely defined, "an ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module"¹⁰. ADLs have recently become an area of intense research in the software architecture community¹¹.

Recently, initial work has been done on an architecture interchange language, ACME¹², which is intended to support mapping of architectural specifications from one ADL to another and, hence, enable integration of support tools across ADLs. Although, strictly speaking, ACME is not an ADL, it contains a number of ADL-like features.

There is, however, still little consensus in the research community on what an ADL is, what aspects of an architecture should be modeled by an ADL, and what should be interchanged in an interchange language¹³.

Another source of discord is the level of support an ADL should provide to developers. At one end of the spectrum, it can be argued that the primary role of architectural descriptions is to aid understanding and communication about a software system. As such, an ADL must have simple, understandable, and possibly graphical syntax, well-understood, but not necessarily formally defined semantics, and the kinds of tools that aid visualization, understanding, and simple analyses of architectural descriptions. At the other end of the spectrum, the tendency has been to provide formal syntax and semantics of ADLs, powerful analysis tools, model checkers, parsers, compilers, code synthesis tools, runtime support tools, and so on.

⁹ D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architectures," SIGSOFT Software Eng. Notes, vol. 17, no. 4, pp. 40-52, Oct. 1992. M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Apr. 1996.

¹⁰ S. Vestal, "A Cursory Overview and Comparison of Four Architecture Description Languages," technical report, Honeywell Technology Center, Feb. 1993.

¹¹ "Summary of the Dagstuhl Workshop Software Architecture," ACM Software Eng. Notes, D. Garlan, F.N. Paulisch, and W.F. Tichy, eds., pp. 63-83, July 1995. Proc. First International Workshop Architectures for Software Systems, D. Garlan, ed., Apr. 1995. Proc. Second International Software Architecture Workshop (ISAW-2), A.L. Wolf, ed., Oct. 1996. Proc. Third Int'l Software Architecture Workshop, J. Magee and D.E. Perry, eds., Nov. 1998.

¹² D. Garlan, R. Monroe, and D. Wile, "ACME: An Architecture Description Interchange Language," Proc. CASCON '97, Nov. 1997.

¹³ N. Medvidovic, R.N. Taylor, and E.J. Whitehead Jr., "Formal Modeling of Software Architectures at Multiple Levels of Abstraction," Proc. California Software Symp., pp. 28-40, Apr. 1996.

NRC/SWT
Kimmo Raatikainen

5.1.2004

An ADL must explicitly model **components**, **connectors**, and their **configurations**; furthermore, to be truly usable and useful, it must provide tool support for architecture-based development and evolution.

A discussion of the scope of software architectures and, therefore, ADLs, is given by Perry and Wolf¹⁴. Their conclusions are largely mirrored in the definition of architectures given by Shaw and Garlan¹⁵.

In our previous work¹⁶, we attempted to identify the problems or areas of concern that need to be addressed by ADLs: representation, design process support, static and dynamic analysis, specification-time and execution-time evolution, refinement, traceability, and simulation/executability.

Understanding these areas and their properties is a key to better understanding the needs of software architectures, architecture-based development, and architectural description and interchange; a study of these areas is also needed to guide the development of next-generation ADLs. We demonstrated that each existing ADL currently supports only a small subset of these domains, and discussed possible reasons for that.

While we believe that this taxonomy gives the architect a sound foundation for selecting an ADL and orients discourse toward problem solving, it is still very much a preliminary contribution. Furthermore, our comparison of ADLs based on these categories did not reveal what specific characteristics and constructs render an ADL well-suited for solving a particular set of problems or whether certain constructs are complementary or mutually exclusive. Consequently, we believe that a feature-based classification and comparison of ADLs is also needed.

Luckham and Vera¹⁷ list requirements for an ADL, based on their work on Rapide: component abstraction, communication abstraction, communication integrity, which mandates that only components that are connected in an architecture may communicate in the resulting implementation, ability to model dynamic architectures, hierarchical composition, and relativity, or the ability to relate (map) behaviors between architectures.

As a result of their experience with UniCon, Shaw et al.¹⁸ list the following properties an ADL should exhibit: ability to model components, with property assertions, interfaces, and implementations, ability to model connectors, with protocols, property assertions, and implementations, abstraction and encapsulation, types and type checking, and ability to accommodate analysis tools.

Tracz¹⁹ defines an ADL as consisting of four "C"s: components, connectors, configurations, and constraints. This taxonomy is appealing, especially in its simplicity, but needs further

¹⁴ D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architectures," SIGSOFT Software Eng. Notes, vol. 17, no. 4, pp. 40-52, Oct. 1992.

¹⁵ M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Apr. 1996.

¹⁶ N. Medvidovic and D.S. Rosenblum, "Domains of Concern in Software Architectures and Architecture Description Languages," Proc. USENIX Conf. Domain-Specific Languages, pp. 199-212, Oct. 1997.

¹⁷ D.C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," IEEE Trans. Software Eng., vol. 21, no. 9, pp. 717-734, Sept. 1995.

¹⁸ M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," IEEE Trans. Software Eng., vol. 21, no. 4, pp. 314-335, Apr. 1995.

¹⁹ W. Tracz, "LILEANNA: A Parameterized Programming Language," Proc. Second Int'l Workshop Software Reuse, pp. 66-78, Mar. 1993.

NRC/SWT
Kimmo Raatikainen

5.1.2004

elaboration: justification for and definitions of the four “C”s, aspects of each that need to be modeled, necessary tool support, and so on. Tracz's taxonomy is similar to Perry and Wolf's original model of software architectures²⁰, which consists of elements, form, and rationale. Perry and Wolf's elements are Tracz's components and connectors, their form subsumes an architectural configuration, and the rationale is roughly equivalent to constraints.

Shaw and Garlan have attempted to identify unifying themes and motivate research in ADLs. Both authors have successfully argued the need to treat connectors explicitly, as first-class entities in an ADL²¹.

They²² also elaborate six classes of properties that an ADL should provide: composition, abstraction, reusability, configuration, heterogeneity, and analysis. They demonstrate that other existing notations, such as informal diagrams, modularization facilities provided by programming languages, and MILs [Module Interconnection Languages], do not satisfy the above properties and, hence, cannot fulfill architecture modeling needs.

Shaw and Garlan²³ identify seven levels of architecture specification capability: capturing architectural information, construction of an instance, composition of multiple instances, selection among design or implementation alternatives, verifying adherence of an implementation to specification, analysis, and automation.

Finally, Medvidovic et al.²⁴ argue that, in order to adequately support architecture-based development and analysis, one must model architectures at four levels of abstraction: internal component semantics, component interfaces, component interconnections in an architecture, and architectural style rules. This taxonomy presents an accurate high-level view of architecture modeling needs, but is too general to serve as an adequate ADL comparison framework. Furthermore, it lacks any focus on connectors.

Perhaps the closest the research community has come to a consensus on ADLs has been the emerging endorsement by a segment of the community of ACME as an architecture interchange language²⁵. In order to meaningfully interchange architectural specifications across ADLs, a common basis for all ADLs must be established. Garlan and colleagues believe that common basis to be their core ontology for architectural representation: components, connectors, systems, or configurations of components and connectors, ports, or points of interaction with a component, roles, or points of interaction with a connector, representations, used to model hierarchical compositions, and rep-maps, which map a

²⁰ D.E. Perry and A.L. Wolf, “Foundations for the Study of Software Architectures,” SIGSOFT Software Eng. Notes, vol. 17, no. 4, pp. 40-52, Oct. 1992.

²¹ R. Allen and D. Garlan, “A Formal Basis for Architectural Connection,” ACM Trans. Software Eng. and Methodology, vol. 6, no. 3, pp. 213-249, July 1997. M. Shaw, “Procedure Calls Are the Assembly Language of System Interconnection: Connectors Deserve First Class Status,” Proc. Workshop Studies of Software Design, May 1993. M. Shaw and D. Garlan, “Characteristics of Higher-Level Languages for Software Architecture,” Technical Report, CMU-CS-94-210, Carnegie Mellon Univ., Dec. 1994.

²² M. Shaw and D. Garlan, “Characteristics of Higher-Level Languages for Software Architecture,” Technical Report, CMU-CS-94-210, Carnegie Mellon Univ., Dec. 1994.

²³ M. Shaw and D. Garlan, “Formulations and Formalisms in Software Architecture,” Computer Science Today: Recent Trends and Developments. J. van Leeuwen, ed. Springer-Verlag, 1995.

²⁴ N. Medvidovic, R.N. Taylor, and E.J. Whitehead Jr., “Formal Modeling of Software Architectures at Multiple Levels of Abstraction,” Proc. California Software Symp., pp. 28-40, Apr. 1996.

²⁵ D. Garlan, R. Monroe, and D. Wile, “ACME: An Architecture Description Interchange Language,” Proc. CASCON '97, Nov. 1997.

NRC/SWT
Kimmo Raatikainen

5.1.2004

composite component or connector's internal architecture to elements of its external interface.

In ACME, any other aspect of architectural description is represented with property lists (i.e., it is not core).

ACME has resulted from a careful consideration of issues in and notations for modeling architectures.

5.1 ADL Classification and Comparison Framework

To properly enable further discussion, several definitions are needed. There is no standard, universally accepted definition of architecture, but we will use as our working definition the one provided by Shaw and Garlan²⁶:

Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

An ADL is thus a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation. ADLs provide both a concrete syntax and a conceptual framework for characterizing architectures²⁷. The conceptual framework typically reflects characteristics of the domain for which the ADL is intended and/or the architectural style. The framework typically subsumes the ADL's underlying semantic theory (e.g., CSP, Petri nets, finite state machines).

The building blocks of an architectural description are: 1) *components*, 2) *connectors*, and 3) *architectural configurations*.²⁸ An ADL must provide the means for their explicit specification; this enables us to determine whether or not a particular notation is an ADL. In order to infer any kind of information about an architecture, at a minimum, interfaces of constituent components must also be modeled. Without this information, an architectural description becomes but a collection of (interconnected) identifiers, similar to a "boxes and lines" diagram with no explicit underlying semantics.

5.1.1 Modeling Components

A **component** in an architecture is a unit of computation or a data store. Therefore, components are loci of computation and state²⁹. ... Each component may require its own data or execution space, or it may share them with other components. As already discussed, explicit component interfaces are a feature required of ADLs. Additional comparison features are those for modeling component *types*, *semantics*, *constraints*, *evolution*, and *nonfunctional properties*.

²⁶ M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.

²⁷ D. Garlan, R. Monroe, and D. Wile, "ACME: An Architecture Description Interchange Language," Proc. CASCON '97, Nov. 1997.

²⁸ "Architectural configurations" will, at various times in this paper, be referred to simply as "configurations" or "topologies."

²⁹ M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," IEEE Trans. Software Eng., vol. 21, no. 4, pp. 314-335, Apr. 1995.

NRC/SWT
Kimmo Raatikainen

5.1.2004

5.1.2 Modeling Connectors

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components, connectors may not correspond to compilation units in an implemented system. They may be implemented as separately compilable message routing devices, but may also manifest themselves as shared variables, table entries, buffers, instructions to a linker, dynamic data structures, sequences of procedure calls embedded in code, initialization parameters, client-server protocols, pipes, SQL links between a database and an application, and so forth³⁰. The features characterizing connectors are their *interfaces*, *types*, *semantics*, *constraints*, *evolution*, and *nonfunctional properties*.³¹

5.1.3 Modeling Configurations

Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether appropriate components are connected, their interfaces match, connectors enable proper communication, and their combined semantics result in desired behavior. In concert with models of components and connectors, descriptions of configurations enable assessment of concurrent and distributed aspects of an architecture, e.g., potential for deadlocks and starvation, performance, reliability, security, and so on. Descriptions of configurations also enable analyses of architectures for adherence to design heuristics (e.g., direct communication links between components hamper evolvability of an architecture) and architectural style constraints (e.g., direct communication links between components are disallowed).

Characteristic features at the level of architectural configurations fall in three general categories:

- qualities of the configuration description: *understandability*, *compositionality*, *refinement and traceability*, and *heterogeneity*;
- qualities of the described system: *heterogeneity*, *scalability*, *evolvability*, and *dynamism*;
- properties of the described system: *dynamism*, *constraints*, and *nonfunctional properties*.³²

Note that the three categories are not entirely orthogonal: Heterogeneity and dynamism each appear in two categories. Heterogeneity may be manifested in multiple employed formalisms in configuration descriptions and multiple programming languages in system implementations. Anticipated dynamism is a system property in that the system may be architected specifically to accommodate the (expected) dynamic change; unanticipated dynamism is a quality that refers to a system's general suitability for dynamic change.

³⁰ D. Garlan, R. Monroe, and D. Wile, "ACME: An Architecture Description Interchange Language," Proc. CASCON '97, Nov. 1997. M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," IEEE Trans. Software Eng., vol. 21, no. 4, pp. 314-335, Apr. 1995.

³¹ Although the comparison categories for components and connectors are identical, they were derived and refined independently of each other.

³² The term "quality" is used in the conventional, application-independent manner, e.g., as defined by Ghezzi et al. (C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Prentice Hall, 1991.). The term "property" refers to the characteristics of an application introduced to address specific requirements.

NRC/SWT
Kimmo Raatikainen

5.1.2004

The differences between the two pairs of features are subtle, particularly in the case of dynamism. While keeping the above categorization in mind, in order to maintain the conceptual simplicity of our framework and avoid confusion, we proceed by describing individual features; we include both notions of heterogeneity and dynamism under single respective headings.

5.1.4 Tool Support for Architectural Description

The motivation behind developing formal languages for architectural description is that their formality renders them suitable for reasoning and manipulation by software tools. A supporting toolset that accompanies an ADL is, strictly speaking, not a part of the language. However, the usefulness of an ADL is directly related to the kinds of tools it provides to support architectural design, analysis, evolution, executable system generation, and so forth. The importance of architectural tools is reflected in the on-going effort by a large segment of the community to identify the components that comprise a canonical "ADL toolkit"³³. Although the results of this work are still preliminary, several general categories have emerged. They reflect the kinds of tool support commonly provided by existing architectural approaches: active specification, multiple views, analysis, refinement, implementation generation, and dynamism.

5.2 Differentiating ADLs from Other Languages

In order to clarify what an ADL is, it may be useful to point out several notations that, though similar, are not ADLs according to our definition: high-level design notations, MILs, programming languages, object-oriented (OO) modeling notations, and formal specification languages.

The requirement to model configurations explicitly distinguishes ADLs from some high-level design languages. Existing languages that are sometimes referred to as ADLs can be grouped into three categories based on how they model configurations:

- *Implicit configuration languages* model configurations implicitly through interconnection information that is distributed across definitions of individual components and connectors;
- *In-line configuration languages* model configurations explicitly, but specify component interconnections, along with any interaction protocols, "in-line";
- *Explicit configuration languages* model both components and connectors separately from configurations.

The focus on conceptual architecture and explicit treatment of connectors as first-class entities differentiate ADLs from MILs³⁴, programming languages, and OO notations and languages (e.g., Unified Modeling Language, or UML). MILs typically describe the uses relationships among modules in an implemented system and support only one type of connection³⁵. Programming languages describe a system's implementation whose

³³ D. Garlan, J. Ockerbloom, and D. Wile, "Towards an ADL Toolkit," EDCS Architecture and Generation Cluster, Dec. 1998. <http://www.cs.cmu.edu/~spok/adl/index.html>.

³⁴ R. Prieto-Diaz and J.M. Neighbors, "Module Interconnection Languages," J. Systems and Software, vol. 6, no. 4, pp. 307-334, Oct. 1989.

³⁵ R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," ACM Trans. Software Eng. and Methodology, vol. 6, no. 3, pp. 213-249, July 1997. M. Shaw and D. Garlan, "Characteristics of Higher-Level Languages for Software Architecture," Technical Report, CMU-CS-94-210, Carnegie Mellon Univ., Dec. 1994.

NRC/SWT
Kimmo Raatikainen

5.1.2004

architecture is typically implicit in subprogram definitions and calls. Explicit treatment of connectors also distinguishes ADLs from OO languages³⁶.

We have also recently shown that an OO language, such as UML, can be used to model software architectures if it supports certain extensions³⁷. These extensions are used to represent architectural abstractions that either differ (e.g., topological constraints) or do not exist (e.g., connectors) in OO design. Extending UML in such a manner is clearly useful in that it supports mapping of an architecture to a more familiar and widely used notation, therefore facilitating broader understanding of the architecture and enabling more extensive tool support for manipulating it. However, it is unrealistic to expect that UML could be extended to model every feature of every ADL; our initial experience indeed confirms this³⁸. Moreover, although UML may provide modeling power equivalent to or surpassing that of an ADL, the abstractions it provides will not match an architect's mental model of the system as faithfully as the architect's ADL of choice. If the primary purpose of a language is to provide a vehicle of expression that matches the intuitions and practices of users, then that language should aspire to reflect those intentions and practices³⁹. We believe this to be a key issue and one that argues against considering a notation like UML an ADL: A given language (e.g., UML) offers a set of abstractions that an architect uses as design tools; if certain abstractions (e.g., components and connectors) are buried in others (e.g., classes), the architect's job is made more (and unnecessarily) difficult; separating components from connectors, raising them both to visibility as top-level abstractions, and endowing them with certain features and limitations also raises them in the consciousness of the designer.

An ADL typically subsumes a formal semantic theory. That theory is part of the ADL's underlying framework for characterizing architectures; it influences the ADL's suitability for modeling particular kinds of systems (e.g., highly concurrent systems) or particular aspects of a given system (e.g., its static properties).

5.3 Conclusions

The main contribution of this paper is just such a definition and classification framework. The definition provides a simple litmus test for ADLs that largely reflects community consensus on what is essential in modeling an architecture: An architectural description differs from other notations by its explicit focus on connectors and architectural configurations.

Finally, neither the definition nor the accompanying framework have been proposed as immutable laws on ADLs. Quite the contrary, we expect both to be modified and extended in the future. We are currently considering several issues: providing a clearer distinction between descriptive languages (...) and those that primarily enable semantic modeling (...); ... But, what this taxonomy provides is an important advance toward answering the question of what an ADL is and why and how it compares to other ADLs.

³⁶ D.C. Luckham, J. Vera, and S. Meldal, "Three Concepts of System Architecture," Technical Report, CSL-TR-95-674, Stanford Univ., Palo Alto, Calif., July 1995.

³⁷ N. Medvidovic and D.S. Rosenblum, "Assessing the Suitability of a Standard Design Method for Modeling Software Architectures," Proc. First Working IFIP Conf. Software Architecture (WICSA1), pp. 161-182, Feb. 1999. J.E. Robbins, N. Medvidovic, D.F. Redmiles, and D.S. Rosenblum, "Integrating Architecture Description Languages with a Standard Design Method," Proc. 20th Int'l Conf. Software Eng. (ICSE '98), pp. 209-218, Apr. 1998.

³⁸ J.E. Robbins, N. Medvidovic, D.F. Redmiles, and D.S. Rosenblum, "Integrating Architecture Description Languages with a Standard Design Method," Proc. 20th Int'l Conf. Software Eng. (ICSE '98), pp. 209-218, Apr. 1998.

³⁹ M. Shaw and D. Garlan, "Formulations and Formalisms in Software Architecture," Computer Science Today: Recent Trends and Developments. J. van Leeuwen, ed. Springer-Verlag, 1995.

NRC/SWT
Kimmo Raatikainen

5.1.2004

6. DEBATE ON MODEL-DRIVEN ARCHITECTURE

Below are the “Point” and “Counterpoint” columns by Alex Uhl and Scott W. Ambler, respectively, as well as their cross-comments.

6.1 Axel Uhl, “Model Driven Architecture Is Ready for Prime Time”, IEEE Software, September/October 2003, p. 70&72 © 2003 IEEE

MDA is the next logical evolutionary step to complement 3GLs in the business of software engineering.

Software engineering has gone through the same dark ages as chip design and other areas of manufacturing and now has a good chance to make the transition to the same maturity level. Assembly language’s complexity and lack of expressiveness and portability taught us to use third-generation languages (3GLs), which—together with the corresponding compilers— we’ve come to take for granted as powerful and indispensable software development tools.

The reason we’re on the brink of moving toward the Object Management Group’s Model Driven Architecture (MDA) is that the technologies we use have grown significantly more complex over the last few years. Moreover, technology changes faster than the businesses we’re trying to support with this technology.

I’m not saying that we have to express each and every detail of our system specification in one or more UML models. But we’re much better off than we are with 3GLs alone: models don’t have to become platform dependent. A reasonable MDA tool lets you add system specifications at various abstraction levels and keeps them synchronized. The concept of marks (mapping-specific annotations that you can attach to model elements) helps keep the models themselves free from unnecessary platform specificities. With this kind of technology, we can provide each bit of system specification in the formalism that’s most appropriate for this purpose.

MDA is built on a solid foundation, including the Meta-Object Facility and UML, which are both well-adopted, ever-maturing formalisms for specifying metamodels and models. The agreement to use UML for most modeling activities takes us a lot farther than we were in the days when we could only agree to use ASCII for the programming languages. Today, you can automatically create UML profiles for a metamodel given in MOF. Several powerful technology- and domain-specific metamodels and corresponding UML profiles have already been standardized—...

Software engineers are seeing a strong pull toward model-centric development. ... MDA is here to stay—just as 3GLs were (and still are) some decades ago. We are now taking the next evolutionary step. From all I’ve seen, I’m convinced that MDA is the way to go. It’s ready for prime time.

Axel Uhl is a software architect in the team developing the architectural IDE ArcStyler at Interactive Objects Software and is pursuing a PhD in software architectures for scalable Internet search at Aachen University. He is actively contributing to the OMG’s MDA standardization efforts. Contact him at axel.uhl@io-software.com.

NRC/SWT
Kimmo Raatikainen

5.1.2004

6.2 Scott W. Ambler, “Agile Model Driven Development Is Good Enough,” IEEE Software, September/October 2003, pp. 71—73 © 2003 IEEE

Has it been 10 years already? The “uber-modeling tool” vision rears its ugly head yet again.

I’m not all that sure about the direction that model-driven development appears to be taking. Don’t get me wrong—I’m a firm believer in modeling. It’s just that I think that there’s a lot more to development than this. Here’s my point: We need to distinguish between generative MDD and Agile MDD. Generative MDD, epitomized by the Object Management Group’s Model Driven Architecture, is based on the idea that people will use very sophisticated modeling tools to create very sophisticated models that they can automatically “transform” with those tools to reflect the realities of various deployment platforms. Great theory—as was the idea that the world is flat. In my opinion, generative MDD is a lost cause for the current generation of developers. Agile MDD will be a struggle to pull off, but at least it has a chance of succeeding.

I believe that modeling is a way to think issues through before you code because it lets you think at a higher abstraction level. You can also do this by writing a test before you write functional code, along the lines of test-driven development. But this isn’t a TDD discussion, so I’ll say nothing more.

I’m also a firm believer in something that I call Agile Model Driven Development (AMDD). An agile model is just barely good enough—it meets its goals and no more. Because “just barely good enough” is relative, you can consider a sketch, a Unified Modeling Language statechart, or a detailed physical database model as an agile model in the right situations. Following an AMDD approach, I typically use very simple tools, such as whiteboards and paper, when I work with users to explore and analyze their requirements. Simple tools are easy to work with, inclusive (my stakeholders can be actively involved with modeling), and flexible, and they’re not constraining. They’re exactly what I need when I’m exploring the problem domain and identifying my system architecture.

Everyone doesn’t need, or want, sophisticated modeling tools. Notably with AMDD, programmers write the code progressively in step with the models—AMDD promotes an evolutionary approach, in which implementation occurs iteratively and incrementally.

To me, generative MDD is based on an incredibly wobbly foundation. First, we don’t yet have a standard modeling language that suffices for real-world needs, making it difficult for developers to work together effectively. Every system that I’ve ever built has both a user interface on the front end and a database on the back end, yet UML still doesn’t address these fundamental issues. Yes, the common rhetoric says all you need to do is apply a few stereotypes to existing UML diagrams, but as my work on a UML data modeling profile shows (see www.agiledata.org/essays/umlDataModelingProfile.html), there’s a lot more to it than this, and I’ve just scratched the surface (I’m also eager to receive feedback on this work).

Second, people simply don’t have the modeling skills. It’s hard enough to teach people how to create a sequence diagram or statechart on a whiteboard, let alone with a complicated modeling tool. We need to learn to crawl before we walk—or run. I’ll be impressed if most developers start sketching in this in the coming decade.

Third, the tools aren’t there. There are some interesting tools such as ..., but they’re not in wide use. The OMG appears to promote the idea that you can cobble a toolset together via the XML Metadata Interchange standard, but I question that too. For this vision to work, tool

NRC/SWT
Kimmo Raatikainen

5.1.2004

vendors would need to actually support the XMI specification as defined. ... Interestingly enough, numerous modeling tool vendors support XMI, yet I can't find a pair of tools that let me model in both and export back and forth without loss of information. Go figure.

The bottom line is that AMDD is a stretch for most developers and, at best, the executable MDD vision is viable in only a few situations. The MDD community should focus on what's practical and not on ivory tower theories.

*Scott W. Ambler is a senior consultant with Ronin International (www.ronin-intl.com). He is thought leader of the Agile Modeling methodology (www.agilemodeling.com), a contributing editor with Software Development (www.sdmagazine.com), and a member of Flashline's (www.flashline.com) Software Development Productivity Consortium. His latest book is *Agile Database Techniques* (John Wiley & Sons, 2004). Contact him at scott.ambler@ronin-intl.com.*

6.3 Cross-comments by Uhl and Ambler, IEEE Software, September/October 2003, p. 73 @ 2003 IEEE

6.3.1 Axel Responds

Scott and I agree that modeling generally is good. However, I disagree that generative MDD, as Scott calls it, or Model Driven Architecture is a lost cause. Several industrial software development projects have already proven it. Take, for example, Deutsche Bank Bauspar or the Austrian National Railroads, reporting total savings around 40 percent for their first MDA project.

With MDA, you can integrate any modeling language that you use with your domain experts by using the Meta-Object Facility (MOF), thus benefitting from automated model verification and transformation.

MDA does not prevent evolutionary approaches with an iterative and incremental development process. We use the MDA tool ArcStyler to develop ArcStyler itself in a team and proceed iteratively and incrementally without problems, particularly without MDA-related ones.

All MDA modeling languages are formally specified in the same metamodeling language MOF: a rock-solid foundation. UML profiles used for model representation are also standardized, as is the specification language for model transformations.

UML was intentionally kept concise, without domain specifics. Instead, UML offers lightweight extensibility, permitting the creation of UML profiles for metamodels formally specified in the MOF. You can even create profiles automatically using MDA.

I think that AMDD incurs the cost of modeling but stops before reaping the true benefits. I will always try to gain value from my models using MDA, as Scott does for the tests. I've seen MDA pay off so many times, in numerous real-world projects. So, I remain firmly convinced that it works.

6.3.2 Scott Responds

Sigh. The good news is that Axel and I didn't agree. That would have been boring, and in many ways it's because I'm focused on the present and Axel is focused on the future. The

NRC/SWT
Kimmo Raatikainen

5.1.2004

reason I'm not very excited about MDA is because I've heard similar visions in the past, visions that all failed miserably:

- Integrated Computer-Aided Software Engineering. I-CASE emerged in the 1980s and failed because the tools couldn't keep up with changing technology. Few developers had the requisite modeling skills or the desire to learn them, and although the tools generated 80 to 90 percent of the code, the extra 10 percent typically required 90 percent of the effort.
- Application Development Cycle. Not only wasn't the market ready for AD/Cycle, it saw through IBM's transparent veil and realized that its real goal for AD/Cycle was to sell products and services instead of the stated altruistic aims.
- Common Object Request Broker Architecture. Even though most CORBA vendors complied with the specification, at least partly, getting their "standard" ORBs (Object Request Brokers) to work together in practice was very difficult. Apparently, the vendors found significant marketing benefit in saying their tools were "CORBA compliant" yet little benefit in making it easy to work with their competitors' products.

I'm jaded when it comes to the MDA because I just don't see how it doesn't suffer from the same problems that sank I-CASE, AD/Cycle, and CORBA. Don't get me wrong, I would be very happy to see the MDA vision succeed because I'm clearly pro-modeling and, like most developers, like to work with good tools that increase my productivity. I'm just not going to hold my breath waiting.

7. EMBEDDED ARTICLES

7.1 IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.



IEEE-1471-2000.pdf

7.2 Nenad Medvidovic, Marija Mikic-Rakic, Nikunj R. Mehta, and Sam Malek, "Software Architectural Support for Handheld Computing," IEEE Computer, September 2003, pp. 66—73.

01231196-Software
ArchitecturalSupport4

NRC/SWT
Kimmo Raatikainen

5.1.2004

7.3 Morgan Björkander and Cris Kobryn, “Architecting Systems with UML 2.0,” IEEE Software, July/August 2003, pp. 57-61.



BjorkanderKobryn-U
ML2.pdf

8. LITERATURE

MDA Guide Version 1.0, OMG Document Number:omg/2003-05-01, 1st May 2003.

IEEE Software Special Issue on Model-Driven Development, September/October 2003.