

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
ARTIFICIAL INTELLIGENCE LABORATORY

## Aspects of the Real-World

(Position paper for the OOPSLA 2001 Workshop on Advanced Separation of Concerns)

Robert Laddaga, Paul Robertson, Howie Shrobe  
{rladdaga,paulr,hes}@ai.mit.edu

August, 2001

### 1 Introduction

The real world is a very complex place. While we were happy confining computers to their own isolated world, disconnected from the real world, we were able to accomplish great things. When we began to connect computers to the real world through sensors and effectors our honeymoon period with computation came to an abrupt end.

Now we find ourselves in an uncomfortable position: Most of our computational aspirations involve interactions with the real world but we have been notoriously unsuccessful in precisely those areas. So far we have not done well at making robots that can roam about unattended and computer vision systems have only been successful in highly constrained situations. On reflection it is not surprising that we have done so poorly. For a program to handle the full complexity of the world it would need to model all of that complexity—and that would be unmanageably huge.

The problem is not hopeless however because although the world is highly complex it doesn't ever expose its full complexity at the same time. Every situation that can be experienced can be viewed as a highly constrained situation. A situation in this case is a complex structure of special sub-cases. For example, interpreting a human speaker might depend upon several dimensions: The speaker, the background noise, the topic of conversation, and so on.

Each of these dimensions can be implemented as a collection of specialized methods for each special case that the program can handle. These different cases can be considered to be different aspects of the real world. When the problem is viewed in this way the problem of building a robust program becomes one of building a high level program that abstracts the structural aspects of the problem. This structure may be referred to as a “super routine”. At each point in the super routine where work must be done there is then a collection of methods that perform the specialized task appropriate to the circumstance. The methods that constitute the different specialized solutions can now be combined along with a decision procedure that effectively searches for the appropriate method in the existing circumstance.

Frequently it is suitable to decide upon the correct method on the basis of probabilities, or of expected utility functions. Our solution to building systems of this kind involves what we call *probabilistic dispatch* and is implemented as a special method combination operator. Methods for the different aspects are weaved together to make a complete program through the build-in mechanism of method combination. This is similar to the way the aspects are weaved together in AspectJ.

Probabilistic dispatch provides a natural way of implementing a dynamic domain architecture in which services are selected at runtime.

A Dynamic Domain Architecture structures a domain into service layers; each service is annotated with specifications and descriptions of how it is implemented in terms of services from lower levels. Like

other domain architectures, a Dynamic Domain Architecture provides multiple instantiations of each service, with each instantiation optimized for different purposes. Thus, it serves as a well-structured software repository. Typically, the application is a relatively small body of code utilizing the much larger volume of code provided by the framework. Typical domains of concern for military embedded software systems include sensor management, navigation guidance and control, electronic warfare, etc.

A Dynamic Domain Architecture is, however, different from the domain architectures developed in earlier DARPA programs (e.g. STARS and DSSA). In earlier systems, the Domain Architecture was a static repository from which specific instantiations of the services were selected and built into the run-time image of the application. Neither the models nor the deductions used to select specific instantiations of the services are carried into the runtime environment. In a Dynamic Domain Architecture, however, all the alternative instantiations, plus the models and annotations describing them are present in the run-time environment, and multiple applications may simultaneously and dynamically invoke the services.

Dynamic Domain Architectures allow late binding of the decision of which alternative instantiation of a service to employ. Like Dynamic Object Oriented Programming (DOOP) systems, the decision may be made as late as method-invocation time. However, Dynamic Domain Architectures go further than DOOP, allowing the decision to be made using much more information than simple type signatures. The models which describe software components are used to support runtime deductions leading to the selection of an appropriate method for achieving a service. Dynamic Domain Architectures recognize that in many open environments (e.g., image processing for ATR) it isn't possible to select the correct operator with precision, a priori. Therefore, Dynamic Domain Architectures support an even later binding of operator selection, allowing this initial selection to be revised in light of the actual effect of the invocation. If the method chosen doesn't do the job well enough, alternatives selections are explored until a satisfactory solution is found or until there is no longer any value to be gained in finding a solution.

## 1.1 Domain modeling

The idea of domain architecture dates back to the Arpa Megaprogramming initiative where it was observed that software reuse could best take place within the context of a Domain Specific Software Architecture. Such an architecture would identify important pieces of functionality employed by all applications within the domain, and would then recursively identify the important functionality supporting these computations. In a visual-interpretation domain, for example, typical common functionality might include region identification, which in turn depends on edge detection, which in turn depends on filtering operations (eg, convolutions).

This process of identifying and structuring the common functionality is the first component of a process termed Domain Analysis. Domain Analysis structures common functionality into a series of "service layers," each relying on the ones below for parts of its functionality. The second component of Domain Analysis is the identification of variability within the commonality. Returning to our visual-interpretation example, there are several different approaches to region identification, dozens of distinct edge-detection algorithms, and many different ways to perform filtering operations. When looked at in even finer detail, there may be an even greater number of variant instantiations of any of these operations. Variations arise due to different needs for precision, time and space bounds, error management, and the like.

The power of Domain Analysis is that its identification of common functionality lets one view the code in a new terms: the bulk of the code is in the service-layer substrate and implements functionality common to many applications. Each application consists of a thin veneer of application-specific code, riding on top of this substrate of service layers. However, the substrate contains many variant instantiations of each service. Although each instantiation is relevant to only some of the applications, Domain Analysis lets us see these as variants of a common conceptual service.

## 1.2 Dynamic Object Oriented Programming

The Lisp community, with its close connection to artificial intelligence research, has independently discovered some of the same ideas, but has packaged them in a more dynamic but less formal framework. This approach was first identified and termed “super-routines” in a paper by Eric Sandewall [San79, SSS81]. Sandewall noted that it is often the case that a whole class of computations are instances of some very general pattern of computation, where the members of the class differ only in the details. He termed this higher-level structure a “super-routine” and noted that data-driven programming techniques could dynamically determine which subroutine was relevant at run time. As object-oriented programming ideas developed in the Lisp and Smalltalk communities, several researchers began to understand that the Dynamic OOP facilities common in these languages were exactly what was need to build a super-routine.

The high-level common services of a Domain Architecture are precisely the same idea as Sandewall’s notion of a super-routine (rediscovered in another context by another community a decade later). Unlike the Static Domain Architectures, the runtime environment of the systems Sandewall characterized all included many variant instantiations of the common services, and dynamically invoked a particular instantiation based on run-time conditions.

The mapping between the super-routine (or Domain Architecture) idea and the features of DOOP is straightforward: each high-level abstract operation (or Domain Architecture service) is identified with a generic function; the different instantiations are provided by different methods, each with a unique type signature. Method invocation performs the dynamic run-time selection of the appropriate instantiation of the service. This style of building extensible, domain specific architectures has become known as “open implementation” [Kic96].

## 2 Dynamic Domain Architectures

Our ideas on Dynamic Domain Architectures stem from several trends in AI and software engineering. The idea that programs can be viewed as instances of plans and that they exhibit a goal-directed structure dates back at least to the work on the Programmer’s Apprentice [RS76, Shr79, Ric81]. DDA frameworks build in diagnostic services which draw on the work on model-based diagnosis [DS82, dW87, dW89, Dav84]. The idea of formalizing a domain architecture and even the idea of dynamic invocation go back to a little referenced but nevertheless seminal paper by Sandewall [San79]. There is a host of more modern work similar in spirit to ours. These include ideas of decomposing systems in cooperating frameworks, using advanced object oriented techniques to build integration techniques, generating the integration code that links frameworks in to larger ensembles. We briefly survey some of that work below:

### 2.1 Subject-Oriented Programming

Subject-oriented programming [HO93] allows the natural specification of an application in terms of the composition of domain and task specific decompositions. The decompositions can cut across normal object-oriented organization involving partial definitions of classes and methods. These decompositions address subject-oriented design paradigms such as product lines, evolutionary development, and multi-team collaborations. Subject-oriented programming permits a user to specify the composition of these separate components as a set of rules for managing their combination and resolving conflicts.

The limitations of pure object-oriented programming have been recognized for a longtime in the LISP community. The Common Lisp Object System (CLOS) provides the ability to decompose applications along object and procedural lines. For example, multi-methods allow a developer to define a set of methods on a particular class or set of classes outside the usual class definition. This permits additions or modifications to object-oriented programs to be specified as separate files or libraries in language without resorting to outside compositional support. Mixin classes, method combination, and runtime namespaces increase the compositional power even further. Finally, procedural macros provide one more tool for specifying behavior that cross-cuts the usual object or procedural boundaries without tangling the source.

## 2.2 Product Lines, Scalable Libraries, and Software Generators

A product-line architecture (PLA) [BS99] is a design for a family of similar applications. A generator is a tool that takes a specification for a composition of scalable libraries and produces a high-performance application. In [SB98], the authors propose an object-oriented building block called a mixin-layer. The idea here is that often additions to software are not localized to a single class but instead span several classes. They show how to compose these layers using an implementation of the Gen Voca theory [BO92].

Our work demonstrates another set of techniques for implementing scalable libraries. In contrast to their work, our CLOS-style substrate is not as constrained as their object-centric foundation (e.g., C++). For example, multi-methods can already be added outside of class definitions.

## 2.3 Aspect-Oriented Programming (AOP)

Aspect-oriented programming [KLM + 97] is a style of programming that complements object-oriented and procedural programming by providing an alternative decomposition of a program into features of a particular domain, called aspects, that cross-cut multiple classes and/or procedures. These aspects can then be merged with the object-oriented and/or procedural parts of a program using a weaver to form an application.

AOP is a generalization of subject-oriented programming in that it goes beyond merely combinations of aspects, but addresses semantic and performance issues as aspects in their own right.

## 2.4 AspectJ

AspectJ [LK98] is an implementation of AOP for Java. It provides a mechanism for combining methods and fields defined in separate aspects. In particular, advise methods can be combined with existing methods and fields can be added. Aspects can be easily plugged in or out of applications by invoking a weaver at compile time. Unfortunately, this is a static operation. Pattern matching is used to specify the domain of the aspects, that is, the methods to which method combination is to be applied.

## 2.5 Refection and Metaobject Protocols

Refection and metaobject protocols [KdR91] are a powerful way to implement AOP. A reflective language is embodied in a base language and several meta languages which control the semantics and implementation of the given base language. The meta languages provide hooks that allow a user to implement cross cutting functionality, functionality to which no single base language has access.

# 3 Probabilistic Dispatch

Probabilistic dispatch is implemented as a special method combination operator. A family of method combination operators are provided to support a number of probabilistic dispatch scenarios. Users can implement additional operators.

Separate methods implement different aspects of each super-routine. For any given super routine then there exist a number of methods ( $M_0 M_1 \dots M_n$ ) that must be woven together by the method combination algorithm.

At runtime when the super-routine (generic function) is invoked the methods in the combined methods list are each asked to provide a plausibility number between 0 and 1, based on the invocation arguments, that determines the likelihood that the method is well matched to the current context that the program is operating in. A method is selected and invoked based on the plausibility and marked as used so that it doesn't get invoked a second time. Plausibility is programmer defined depending on the problem domain. Examples for definitions of plausibility include:

1. Probability of applicability of the method in the current world state.

2. The expected utility of the methods normalized to the [0,1] interval.

Each method implements a solution for an aspect of the real world. If the method cannot complete it performs a “callNextMethod)” in order to pass the responsibility on to another method.

The callNextMethod invocation causes the unused methods in the combined method to be re-polled for their plausibility (since knowledge of the state of the world may have changed) and again the a method is marked as used and invoked.

How the method is selected depends upon the method combination type. Initially we have three different probabilistic dispatch method combination operators:

1. **Deterministic:** The deterministic operator picks the most plausible method. If there are several methods that report the same highest plausibility the first of these methods is selected. The order is the order in which the methods were defined in the program.
2. **Random:** The random operator picks the most plausible method. If there are several methods that report the same highest plausibility one of the methods is selected at random.
3. **Monte-Carlo:** Rather than selecting the most plausible method the Monte Carlo method combination operator picks the one of the methods at random (weighted by the plausibility). This operator can be used in a loop invocation in which several results are sampled (Monte Carlo samples). This is useful when outright failure of a method and the subsequent invocation of callNextMethod is not likely but the result may not be optimal. The Monte-Carlo combinator permits the best solution provided by the methods to be approximated.

## 4 Conclusion

We have described an approach to expanding method dispatch to include a very general sort of probabilistic (or expected utility) computation of the appropriate set of methods for a given generic function call. That expanded form of dispatch allows us to dynamically control the application of method dispatch via virtually any computation that will return a probability or utility value.

This facility has very broad application, including dynamically changing method calling in response to a wide variety of environmental or historical considerations. It also lets us easily segregate, design, and program cross cutting concerns such as memory utilization, power utilization, or user or environmental preferences. The modularity and efficiency of AOP can be achieved with this mechanism.

Generic function dispatch is already a well understood facility in DOOP languages, so the extended facility is not be hard for programmers to learn. Since it involves no language changes, but only sufficient meta-object protocol capabilities to program method dispatch, it is easy to implement.

Potential applications include web search, transaction systems, embedded software and autonomous systems. It is a crucial foundation for self-adaptive software (self monitoring and self repairing software).

## 5 References

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. ACM TOSEM, October 1992.

[BS99] D. Batory and Y. Smaragdakis. Object-oriented frameworks and product-lines. Submitted for publication, 1999.

[Dav84] Randall Davis. Diagnostic reasoning based on structure and behavior. Artificial Intelligence, 24:347410, December 1984.

[DS82] Randall Davis and Howard Shrobe. Diagnosis based on structure and function. In Proceedings of the AAAI National Conference on Artificial Intelligence, pages 137 142. AAAI, 1982.

[dW87] Johan deKleer and Brian Williams. Diagnosing multiple faults. Artificial Intelligence, 32(1):97130, 1987.

- [dW89] Johan deKleer and Brian Williams. Diagnosis with behavior modes. In Proceedings of the International Joint Conference on Artificial Intelligence, 1989.
- [HO93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In Conference on Object-Oriented Programming: Systems, Languages, and Applications, pages 411428, 1993.
- [KdR91] Gregor Kiczales and Jim des Rivieres. The art of the metaobject protocol. MIT Press, Cambridge, MA, USA, 1991.
- [Kic96] Gregor Kiczales. Beyond the black box: Open implementation. IEEE Software, January 1996.
- [KLM + 97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Ak'sit and Satoshi Matsuoka, editors, ECOOP '97 — Object-Oriented Programming 11th
- [LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ(tm). In Serge Demeyer and Jan Bosch, editors, Object-Oriented Technology: ECOOP'98 Workshop Reader, volume 1543 of Lecture Notes in Computer Science, pages 398–401. Springer, 1998.
- [Ric81] Charles Rich. Inspection methods in programming. Technical Report AI Lab Technical Report 604, MIT Artificial Intelligence Laboratory, 1981.
- [RS76] Charles Rich and Howard E. Shrobe. Initial report on a lisp programmer's apprentice. Technical Report Technical Report 354, MIT Artificial Intelligence Laboratory, December 1976.
- [SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In European Conference on Object-Oriented Programming, 1998.
- [Shr79] Howard Shrobe. Dependency directed reasoning for complex program understanding. Technical Report AI Lab Technical Report 503, MIT Artificial Intelligence Laboratory, April 1979.
- [San79] Erik Sandewall. Why super routines are better than subroutines. Technical Report LiTH-MAT-R-79-28, Linkoping University, November 1979.
- [SSS81] Erik Sandewall, Claes Stromberg, and Henrik Sorensen. Software architecture based on communicating residential environments. In Fifth International Conference on Software Engineering, San Diego, 1981.