Rapid Development of Custom Software Architecture Design Environments

Robert T. Monroe

August, 1999

CMU-CS-99-161

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee

David Garlan, Chair
Mary Shaw
Steve Cross
David Notkin, University of Washington

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Copyright © 1999 Robert T. Monroe

This research was sponsored by 1) the Defense Advanced Research Projects Agency (DARPA) the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF under contract no. F33615-93-1-1330, 2) DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0224, 3) fellowships from the National Science Foundation (NSF) and Eastman Kodak Corporation, and 4) Hewlett Packard. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of these organizations, the US government, or any other party.

DTIC QUALITY INSPECTED 2

DISTRIBUTION STATEMENT A1
Approved for Public Release
Distribution Unlimited

20000509 116

Keywords: software architecture, software design, software design tools, software architecture design environments, architecture description languages, configurable software, rapid software development

Preceding Pages Blank

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
COMPUTER SCIENCE

Rapid Development of Custom Software Architecture Design Environments

ROBERT MONROE

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

[illegible] THESIS COMMITTEE CHAIR — August 11, 1999 DATE

APPROVED:

[illegible] DEAN — August 27, 1999 DATE

Abstract

Software architecture provides a powerful way to manage the complexity of large software systems. It has emerged as a distinct form of abstraction for software systems with its own set of design issues, vocabulary, and goals. Like designers in other disciplines, software architects can gain significant leverage by using powerful and appropriate design environments and tools. The cost and difficulty of creating these powerful design tools, however, prohibit their use for many software development projects. One of the primary reasons for the difficulty and cost of building these tools is that tool developers generally need to build a significant amount of supporting infrastructure before they can make use of the important architectural design expertise that the tools encapsulate. This infrastructure includes both the concepts underlying the tools' functionality and the implementation of the tools themselves.

This dissertation describes a new approach to capturing and using architectural design expertise in software architecture design environments. A language and tools are presented for capturing and encapsulating software architecture design expertise within a conceptual framework of architectural styles and design rules. The design expertise thus captured is supported with an incrementally configurable software architecture design environment that specialized design environment builders and end-users can easily and quickly customize by specifying the architectural styles and design rules that the environment needs to support.

Dedicated to my twenties, frittered away in a Quixotic pursuit.

If I have seen further, it is by standing on the shoulders of giants.

– Sir Isaac Newton

Acknowledgements

Although my name graces the cover of this dissertation I would have been unable to complete it without the support and guidance of my family, friends, and colleagues. Please accept my thanks for your assistance and patience.

I would particularly like to recognize the following people:

David Garlan has been an excellent teacher and a helpful guide in navigating the vagaries of the research world. He is a whetstone for the intellect who pushed me to think like a scientist, to exceed the acceptable, and to aim for the superb. I can only hope that his lessons stuck.

In our many conversations over the past six years Mary Shaw has been immensely helpful in refining my thinking about software, engineering, and software engineering. She has been an excellent teacher of thinking, technology, BS detection, and the ability to enjoy your life outside of work.

As members of my thesis committee, Steve Cross and David Notkin generously gave their time and insight on how to improve the dissertation.

Rob Allen taught me how to survive graduate school, pointing out land mines before I stepped on them. He was also the key developer of the original Aesop system, out of which many of the Armani ideas grew.

Pete Su, Drew Kompanek, Greg Zelesnik, Rob DeLine, Eugene Fink, and too many other members of the SCS community to list, provided great assistance in focusing my thinking about this research and making it actually work. They played their roles as friends, sounding boards, and technical colleagues beautifully.

Dick Creps, Paul Kogut, Peyman Oreizy, Franklin Webber, Jun Li, and Peter Feiler all generously participated in the case studies described in this dissertation. Their willingness to try unproven software was courageous. The insight gained from these projects was valuable.

Finally, my wonderful wife Betsy has stood by me through six years of neurotic ramblings about software engineering, manic fits of depression and ebullience, and countless evenings home alone while I was working late at CMU. She has been my anchor to sanity and my provider of perspective throughout this journey.

Thank you.

Table of Contents

Chapter 1
Introduction

This dissertation demonstrates that it is possible to capture a significant and useful collection of *software architecture design expertise* with a language and mechanisms for expressing *design vocabulary, design rules,* and *architectural styles*. This captured design expertise can be used to incrementally customize software architecture design environments.

1.1 The role of software architecture

Software designers and developers have long realized the importance of powerful and appropriate abstractions for software systems. The architectural level of abstraction describes, at a relatively coarse granularity, the decomposition of a software system into its major components, the mechanisms and rules by which those components interact, and the global properties of the system that emerge from the composition of its pieces. There is growing recognition in the software design community that one of the critical steps for the successful completion and fielding of a major software system is the creation of a well defined and documented architecture [Gar95, RM97].

There are (at least) four major benefits to producing and documenting an architectural design:

1) **Analysis capabilities**. Given an appropriate set of analysis tools, system designers can flag likely problems and estimate global properties and capabilities of the system early in the development lifecycle. This capability allows the designers to perform an early analysis of whether the fielded system will be able to meet its requirements in a cost-effective way [All+98, SG98].

2) **System structure visibility**. An explicitly defined and documented architecture communicates "the big picture" of how the entire system will fit together to guide system developers in making lower-level design decisions. Developers that have a clear understanding of how their piece of the system fits with the system as a whole can insure that their components will integrate smoothly with the rest of the system and use the architecture to guide them in making good implementation decisions.

3) **Imposed discipline**. The process of producing an explicit architectural design requires architects to think about the system as a whole and how its pieces interact. This process will often uncover fuzzy thinking, poorly defined requirements, and important design issues that might otherwise be overlooked [ATT93].

4) **Maintaining conceptual integrity**. A system's architecture serves as its "conscience," guiding maintainers in making appropriate extensions and modifications to the system.

In this way, the architectural document exposes the "load-bearing walls" of the system [SG96, PW92].

Although architectural design is a critical factor in the successful design, development, deployment and maintenance of a software system, it requires significant time, effort, skill, and thus expense, to do well. Much of the challenge of producing an appropriate software architecture arises from the fundamental difficulty of determining the core abstractions to use in describing the system, accurately capturing those abstractions in a concrete representation that system developers and programmers can use as a blueprint for system implementation, and analyzing the architectural description to determine whether the system to be produced is likely to exhibit its desired properties. Because of the difficulty and cost of producing such an architectural specification, as well as the difficulty in quantifying the benefit of doing so, it is frequently difficult for software development organizations to justify producing, documenting, and thoroughly testing detailed software architecture specifications for the systems they develop.

To address the cost and difficulty of producing effective software architecture descriptions, numerous Architecture Description Languages (ADLs) have been developed[1]. Unfortunately, current ADLs have two critical limitations. First, they generally do not provide any mechanisms for describing planned or available evolutionary paths for the software system described. They can not describe the constraints under which the design was created and may be evolved, the invariants of the design that need to be maintained as the system evolves, nor the heuristics used in the system's design. Current ADLs tend to capture a snapshot of the design of individual systems at implementation or deployment time, without a roadmap to guide subsequent system maintenance and evolution. Because system maintenance and evolution costs are frequently greater than the cost of initial system design and development [Pfl87], the inability to support them directly is a significant limitation of existing ADLs.

The second limitation is that current ADLs tend to emphasize the ability to specify the design of a *single* software system and, in some cases, to analyze various properties of that single design. Many software development organizations, however, build families of related systems. As a result, the design and development of a software system is rarely a ground-up endeavor requiring the production of a fresh design and set of design concepts. Rather, the designers designing a new system or updating an existing system tend to reuse proven designs, design rules, and design vocabulary that their organization has developed or acquired in building previous systems. Unfortunately, few if any existing ADLs are equipped to capture this design expertise so that it can be reused to guide the building of new systems or the modification of existing system designs. As a result, this critical organizational software design expertise tends to be either kept in the heads of a few experts or, if it is written down at all, expressed informally in natural language documents. The value of existing ADLs is limited for many software development organizations, projects, and processes because they don't provide an effective way to capture this design expertise.

[1] See the related work section 3.1 for a detailed discussion of ADLs, their capabilities, and their limitations.

Both of these problems can be addressed with an Architecture Description Language that has mechanisms and constructs for capturing, archiving, retrieving, and reusing *architectural design expertise*. For the purposes of this thesis, architectural design expertise encompasses the concepts, models, and rules that skilled software architects use when specifying, constructing, or analyzing a software architecture. This includes generic design rules and vocabulary that are applicable to a family of systems, as well as the constraints, rules, and heuristics used in producing a single system and guiding how that system can evolve.

1.1.1 A case for automated software architecture design tools

The ability to explicitly capture and reuse software architecture design expertise is an important first step towards improving the state of the practice of building and evolving software. It is, however, only a first step. To take full advantage of this captured expertise software architects also need powerful design and analysis tools and environments. These tools can guide an architect in analyzing and reasoning about software architectures, testing an architectural specification for compliance with a set of design rules, or selecting an appropriate collection of vocabulary elements for a specific system design. Such a set of tools should allow software architects and developers to do their job more easily, quickly, and effectively.

The success of Computer Aided Design (CAD) tools in other disciplines such as mechanical engineering, building architecture, and VLSI design argues that when design tools capture the essential aspects of design in a given domain (that domain's design expertise) they can offer useful analyses, significant reuse of common design elements, and even design guidance and evaluation. Experience with these tools also demonstrates that, in general, as they are made more domain specific the tools provide greater leverage for the designers using them. The standard way to make design tools more domain specific is to encode design expertise from the target domain directly into the tools. For example, a VLSI CAD tool might include routing and layout algorithms, a large library of predefined VLSI components such as registers, busses, and memory blocks, rules for detecting and dealing with timing and impedance mismatches, and tools to simulate the expected behavior of a chip before it is fabricated. Each of these capabilities captures a collection of design expertise that tool users can take advantage of in producing their chip designs.

Appropriate software architecture design tools can provide software architects with similar leverage. Specifically, specialized software architecture design tools and environments promise to provide software architects with three benefits:

- **Reusable conceptual frameworks.** Specialized software architecture design environments capture and encode a conceptual framework for designing specific types of systems. Such a framework usually includes a vocabulary of building blocks, rules and semantics for composing those building blocks, and analyses that can be performed on systems developed with the environment. Much of the difficulty in producing a complex software system arises from the need to develop an appropriate abstract conceptual model as a basis for the system. These specialized environments allow a designer to reuse the building blocks and expertise that the tool has captured instead of having to create his own models from scratch for each new system.

- **Design guidance.** Selecting a well understood conceptual framework as a basis for system design guides architects by providing a vocabulary of proven design elements and suggesting appropriate ways to compose those elements. This foundation can be extended to capture the heuristics, guidelines, and contextual cues that experts take advantage of when designing the class of systems that the specialized environment supports. Further, by encoding information about the evolutionary paths and constraints built into a specific system's design, that system's original architect(s) can guide future system maintainers towards safe and appropriate system evolution.[2]
- **Design evaluation.** Constraining the architectural design space to a well understood conceptual framework provides a foundation for creating tractable and automatable design analyses and evaluations. The ability to evaluate the costs and benefits of various design approaches and alternatives early in the system design and development process allows designers to catch and/or prevent costly design mistakes when they are relatively inexpensive to fix.

Many organizations have recognized the potential benefits of producing and using automated software architecture design tools and environments. Consequently, a great deal of investment has been made in creating them. These tools range from generic environments that provide limited leverage over a wide variety of design domains, such as Rational Rose and the Unified Modeling Language [Qua98], to domain-specific design environments that provide a lot of analytical leverage and design guidance but are constrained to a much more limited scope of design. The latter group includes DARPA's Domain-Specific Software Architecture (DSSA) environments [MG92], MetaH [Ves94], ObjecTime's Real-time Object-Oriented Modeling (ROOM) tools [SGW94], and the Chimera framework for robotics software [SVK93].

1.1.2 Limitations of current approaches

Although specialized software architecture design environments offer significant promise, current approaches to designing and building them are inadequate for the following three reasons. First, it is rarely cost effective for software development organizations or projects to build tools and environments that are tightly customized to their specific design domain and problems. Historically, such software architecture design environments have had to be built from the ground up. As a result, they are expensive, difficult, and time-consuming to build. Because of the large up-front investment they require, such specialized design environments make economic sense only for projects and organizations that are able to use the environment for the production of many systems, thus amortizing the environment's development cost. Although they might benefit greatly from using a highly customized software architecture design environment, it is currently far too expensive for software architects in many domains and organizations to develop such customized tools.

A second problem with current approaches is that the environments produced by building from scratch tend to be brittle and difficult to evolve. As a designer's understanding of his domain and design techniques evolves, the tools that he uses need to be readily evolvable

[2] Although guidance in selecting an appropriate conceptual framework given a set of requirements is also an important issue in design guidance, that type of guidance is not directly addressed in this thesis.

also. The lack of a standard, generic way to encode design expertise requires each environment development project to (re)invent a representation for the design expertise that they capture and encode. Because they are developed for a single environment project, the schemas and mechanisms used to represent the environment's design expertise tends to be idiosyncratic, highly tool-specific, and difficult to modify or reuse. As a result, these tools tend to work well for a specific domain or style of architectural design (or perhaps a small set of domains), but they can not be readily adapted as the tool users' understanding of the domain evolves.

The third problem with the current approach to building specialized software architecture design environments is that it requires a great deal of tool-building expertise to construct such an environment. Building an effective design environment requires a specific set of skills and experience, which experts in designing other types of software may not possess. Conversely, a skilled developer of software architecture design environments is unlikely to possess the deep understanding of the target domain required to produce an environment that is tightly matched to that domain. As a result, an organization that wants a custom software architecture design environment is likely to end up with either a well-built environment that is a poor match for their domain, or a poorly crafted design environment that captures their design expertise but is not effective as a design tool.

As a result of these limitations, today's dominant model for building design environments is to have software designers adapt their problems to fit the concepts and models of the tools provided by the tool developers. This could take the form of using generic design tools that provide minimal leverage (e.g. "find the objects" using UML [BRJ98]), or attempting to use specialized tools that were designed to address a different problem or style of design. In both cases, a mismatch between the tool capabilities and the needs of the designer ensues. A more appropriate model, which is developed in this thesis, is to allow and encourage architects to easily, incrementally, and quickly adapt their tools to solve the design problems that they face.

1.1.3 The role of lightweight, incremental adaptation

Making the development and use of specialized architecture design environments practical and economically feasible for a wide variety of software development projects requires a new approach to their design and construction. Rather than building new design environments from scratch for every domain, it should be possible to easily and quickly adapt an existing design environment by incrementally adding new design expertise to it. The ability to incrementally adapt a design environment with domain-specific architectural design expertise provides three major benefits:

- **Simplified design environment construction**. Incrementally customizing a reusable environment that is designed for adaptation requires dramatically less effort than building a comparable environment from scratch. Because the bulk of the environment infrastructure is being reused, the cost, time, difficulty, and expertise required to produce a custom design environment can therefore be significantly reduced. As a result, the use of customized software architecture design environments becomes economically feasible for a broad array of software development projects that could not afford to produce such an environment with current technologies.

- **Design environment evolvability**. A design environment that is built to support incremental reconfiguration can evolve as its users' understanding of the domain and design techniques grows. By building evolvability into the environment from the beginning, the brittleness that plagues many custom tools and environments built from the ground up can be avoided.
- **End-user experimentation**. Using the incremental addition of design expertise as the primary mechanism for adapting a design environment provides an architect or domain expert with a great deal of environment customization capability and flexibility. The end-user does not need to know any significant details about the implementation of a design environment in order to adapt and customize it. He simply needs to understand the mechanisms for expressing design expertise and loading them into the configurable environment. As a result, customization decisions can be made by those who understand the domain and its design techniques best – the architects and domain experts using the environment. Further, because environment customization is a lightweight process, it is easy for an environment user to experiment with many different forms of design expertise and environment configurations.

In order to achieve these benefits, this dissertation presents a new approach to capturing software architecture design expertise and using it to incrementally customize software architecture design environments. The key to this approach is a language and conceptual framework for capturing design expertise, along with a flexible design environment infrastructure that can be easily, quickly, and incrementally configured with the language.

There are two fundamental challenges to supporting the incremental customization of software architecture design environments with encapsulated design expertise. The first challenge is simply developing a way to capture software architecture design expertise at all. This includes developing both a notation for expressing architectural design expertise and a conceptual framework that defines the relationships between the various constructs of the notation. The notation and framework must be sufficiently rich, flexible, and powerful to capture the important architectural design expertise for a broad range of architectural styles and design domains. Further, they must capture the expertise in a way that is straightforward for software architects to understand and use.

The second challenge lies in designing and building an extensible software architecture design environment infrastructure that can be incrementally adapted with this captured design expertise. Such an environment needs to be able to incrementally incorporate the design expertise captured in the notation and framework. To complicate matters, the environment needs to be able to deal with collections of design expertise that may be internally inconsistent, or even contradictory.

There appears, unfortunately, to be a fundamental tension that arises in attempting to simultaneously address these challenges. As the notation and conceptual framework is made more rich, flexible, and closer to a natural language it tends to become more difficult to provide automated tools that can process the design expertise. Throughout the remainder of this dissertation I describe how I address these tradeoffs and challenges to demonstrate my thesis that:

It is possible to capture a significant and useful collection of software architecture design expertise with a language and mechanisms for expressing design vocabulary, design rules, and architectural styles. This captured design expertise can be used to incrementally customize software architecture design environments.

To demonstrate this thesis, I present a design language and a configurable design environment called *Armani*. I then show how they can be used together to incrementally capture software architecture design expertise and develop customized software architecture design environments. To demonstrate the utility of the approach, I describe a set of case studies in which Armani is used to rapidly construct custom design environments.

1.1.4 Contributions

The research presented in this dissertation makes the following contributions to the field of Computer Science:

- **A technique** for dramatically reducing the time, cost, and difficulty of building a significant class of customized software architecture design environments. This technique benefits software architecture design environment builders by demonstrating how a variety of design tools can be built through principled, incremental adaptations to a common shared infrastructure. It benefits software development organizations by providing access to highly customized tools at a much lower cost than current development techniques allow. It benefits practicing software architects by providing them with tools that closely match their design domain. Finally, it benefits researchers studying software development tools by providing a general customization technique that can likely be extended to other design and problem domains.
- **A design language**. The dissertation describes a software architecture design language that is capable of incrementally capturing software architecture design expertise with modular and reusable language constructs. The design language is also a full-fledged architecture description language (ADL) capable of describing the structure of software architectures and the constraints and guidelines under which those systems were designed and may be evolved.

  The design language contributes to the software architecture research community by demonstrating that a first-order predicate logic-based constraint language can be used to define useful design rules to guide software design and evolution. Further, the language encodes an extensible framework for capturing software architecture design expertise. In addition to its benefit to researchers, the design language also benefits software development organizations by providing a way to capture and reuse the organizational design expertise they develop in building software systems. Finally, it benefits software architects by providing an explicit technique for capturing and expressing architectural design constraints in software architecture specifications.

- **A reference architecture.** The dissertation describes a reference architecture, or architectural style, for software architecture design environments that support incremental customization. It describes the architecture of the Armani design environment, discusses why a number of alternative architectures were not selected, and presents some fundamental tradeoffs of this style of architecture. This contribution is particularly useful for software tool builders who need adaptable, modular architectures for design tools and environments.

- **A set of case studies.** A set of detailed examples and case studies are presented to illustrate how the technique, language, and integration framework just described can be used to effectively capture software architecture design expertise and rapidly develop custom software architecture design environments. The case studies benefit people using Armani to design software architectures and build custom software architecture design environments. They are also useful for researchers interested in further exploration of the ideas presented in the dissertation.

1.2 Armani overview

The approach presented in this thesis for rapidly developing custom software architecture design environments has two primary technical components. The first component, a language and framework for capturing architectural design expertise and individual architectural designs, constitutes the core technical foundation for the approach. The language binds the foundational concepts and constructs available to software architects and environment designers for specifying architectures and capturing design expertise. The second component, a configurable and extensible design environment, can be customized with this design language to support specific styles of architectural design. Throughout this dissertation I will refer to the language and environment together as the *Armani System*, or simply *Armani*. When it is necessary to distinguish between Armani's language and design environment I will use the terms *Armani design language* and *Armani design environment*, respectively.

1.2.1 Critical requirements

In order for the Armani language and environment to achieve their goals of reducing the time, difficulty, and cost associated with building custom software architecture design environments, they must meet the following requirements.

1) **Incrementality**. A software architect using Armani should be able to incrementally adapt his or her Armani-based tools to make use of available design expertise, or to specify and add additional design expertise. Further, the incremental adaptation of an existing environment should be significantly quicker and easier than building a new environment from scratch.

2) **Power**. The Armani language and environment should be able to capture useful, non-trivial software architecture design expertise.

3) **Breadth.** The mechanisms provided by Armani should be capable of capturing a range of software architecture design expertise that is sufficiently broad to produce design environments for a diverse collection of software architecture design domains.

Strictly speaking, only the first of these requirements – incrementality – needs to be demonstrated for the thesis to hold. The requirements for power, breadth, modularity, and reusability simply assure that the environments incrementally developed with Armani will be useful for a sufficiently wide audience of software architects and environment developers. This dissertation will, however, demonstrate that the Armani approach to rapidly developing custom software architecture design environments satisfies all four of these requirements.

1.2.2 The Armani design language

The Armani design language can be used for capturing both software architecture design expertise and the architectural specification of individual software system designs. The language provides constructs for capturing three fundamental classes of architectural design expertise – *design vocabulary*, *design rules*, and *architectural styles*. A brief overview of each of these follows. A more complete description and specification of the Armani design language is provided in chapter 3.

- *Design vocabulary* is the most basic form of design expertise that can be captured with Armani, and possibly the most valuable. The design vocabulary available to a software architect specifies the basic building blocks for system design. Design vocabulary describes the selection of components, connectors, and interfaces that can be used in system design. As an example, the design vocabulary available for a naïve client-server style of design might include client and server components and an HTTP connector. Armani provides a rich predicate-based type system that environment designers can use to specify the design vocabulary, the properties of vocabulary elements, and the design invariants and heuristics that describe how the vocabulary elements can be used.

- *Design rules* specify heuristics, invariants, composition constraints, and contextual cues to assist architects with the design and analysis of software architectures. Armani makes the following aspects of a design rule independently modifiable: the specification of the rule itself, the policy for dealing with violations of the rule, and the scope over which the rule is enforced. Armani allows the association of design rules with a complete style, a collection of related design elements (such as all of the components in a system), a type of design element, or an individual instance of a component or connector. By making the scoping of design rules flexible and specifying their policy independent of the rule itself, Armani allows an architect to add, remove, modify, or temporarily ignore design rules as appropriate for various stages and types of design.

- *Architectural styles* provide a mechanism for packaging and aggregating related design vocabulary, rules, and analyses. An Armani style specification consists of the declaration of a set of design vocabulary that can be used for designing in the style, and a set of design rules that guide and constrain the composition and instantiation of the design vocabulary.

In addition to the constructs provided for capturing abstract design expertise, the Armani design language is also a full architecture description language in its own right. Architects can use Armani to describe the architectures of individual software systems, how those systems fit into a family of related systems, and how those systems are allowed to evolve over time. The language constructs for describing instances of software architectures are fully integrated with (and overlap) the language constructs used for capturing the design expertise used to customize the Armani design environment.

Using a single language for both tasks has a number of benefits. First, an architect needs to know only a single design language to both design software architectures and modify or update a design environment "on the fly". Second, it is relatively straightforward for automated tools to determine whether an individual architectural specification satisfies the design rules that were used in its design. This becomes a particularly important capability when modifications are proposed to the design that could violate some of the original design rules, as frequently occurs during the maintenance and upgrade stages of a system's lifecycle.

1.2.3 The Armani design environment

The Armani design language provides the key conceptual infrastructure for the rapid development of custom software architecture design environments by supporting the capture of abstract architectural design expertise. Converting this captured expertise into a working customized design environment, however, requires the additional infrastructure that the Armani design environment provides. Specifically, the Armani design environment provides the generic core infrastructure common to a large class of software architecture design environments, including a design representation database, a graphical user interface (gui), a tool integration mechanism, a generic design rule verification system, and tools to support end-user environment customization.

An Armani environment designer builds on this generic environment infrastructure to quickly develop a custom, specialized software architecture design environment. By (re)using this configurable generic infrastructure as a basis for a custom design environment, an Armani design environment builder begins his custom environment development project with a big head-start.

The basic development model for producing such a custom Armani environment takes advantage of this configurable infrastructure. An Armani environment designer uses the Armani design language to specify the design vocabulary and design rules for the target domain or style of architectural design. He then loads this design expertise directly into the Armani environment. In doing so, the environment configures itself to support the vocabulary and semantics of design in the target domain. The specified design vocabulary types and design rules for the domain are loaded and available for architects to use in designing software architectures with the environment. Further, because the structure and semantics for the target style (or styles) of design have been specified, the environment can provide basic design checks for semantic consistency such as type checking and confirmation that the design rules are satisified.

This environment customization process is both incremental and experimental. Design vocabulary and design rules can be added to an environment, removed from an environment, or modified at almost any point during the environment's creation or use as a design tool. As a result, both environment developers and end-users can quickly adapt their design environment to reflect newly discovered design expertise that needs to be incorporated into the environment.

The ability to dynamically load architectural styles, design vocabulary and design rules into an Armani environment provides one form of environment customization. In addition to this basic semantic customization, Armani can be configured and extended with rich visualizations for rendering specific types of design elements (e.g., to make a database component look different from a web-browser component). Likewise, additional design analysis or construction tools can be specified and/or linked into the environment to provide richer tooling capabilities. Like the basic semantic customization capabilities, visualization and tooling extensions can be added to the environment both statically when the design environment is initially created, or dynamically while the environment is in use.

1.2.4 Structure of the dissertation

The remainder of this dissertation expands and elaborates the discussion of the Armani approach to capturing and exploiting software architecture design expertise to incrementally customize architecture design environments. Chapter 2 provides an overview of how the Armani system can be used to capture architectural design expertise and rapidly create a custom design environment. Chapter 3 discusses related work. Chapter 4 describes the Armani design language used for capturing architectural designs and architectural design expertise. Chapter 5 details the design and architecture of the Armani configurable design environment. Chapters 6 through 8 outline and detail the steps taken to validate the thesis. This includes a detailed task analysis and a series of case studies that demonstrate Armani's capabilities. Chapter 9 evaluates the results of the thesis research and discusses open issues. Finally, Chapter 10 wraps the argument up with an evaluation of the thesis results and a discussion of promising directions for future work.

Chapter 2
A Quick Tour of Armani

This chapter provides a brief tour of the Armani system. It illustrates, from the environment user's perspective, how Armani can be used to rapidly and incrementally create highly customized software architecture design environments. Subsequent chapters will provide significantly more detail on the Armani language and environment, as well as detailed presentations of some specialized environments produced with Armani.

2.1 The generic Armani infrastructure

The Armani design environment provides the generic core infrastructure common to a large class of software architecture design environments. This core infrastructure includes a design representation database, a graphical user interface (gui), a tool integration mechanism, a generic design rule verification system, and tools to support end-user environment customization. In addition to being the base on which custom design environments are built, this core infrastructure also functions as a complete, albeit generic, design environment in its own right. Architects can use the generic environment to specify software system architectures without any further environment customization.

Figure 2.1 shows an example of how an architect can use this generic design environment to specify the architecture of a software system. In this example, an architect uses the reusable vocabulary elements stored in the palette to the left of the environment's main window to describe a simple instance of a software system with a few interacting components and connectors. Manipulating these instances of generic components and connectors with the environment's user interface tools allows the architect to specify the names of the components and connectors, the topology of their interactions, emergent properties of the system as a whole, and the properties of individual design elements (components, connectors, and their interfaces).

The primary drawback to using only the generic design environment is that it provides the designer only generic components and connectors to use as building blocks and minimal analytical capabilities or design guidance. There is little semantics associated with the design elements, just suggestive names and descriptions. To address these limitations, Armani builds on this common, generic base by providing the ability to incrementally customize the generic environment with new design vocabulary, design rules, and architectural styles. An environment designer can use these customization capabilities to create semantically rich building blocks and analyses.

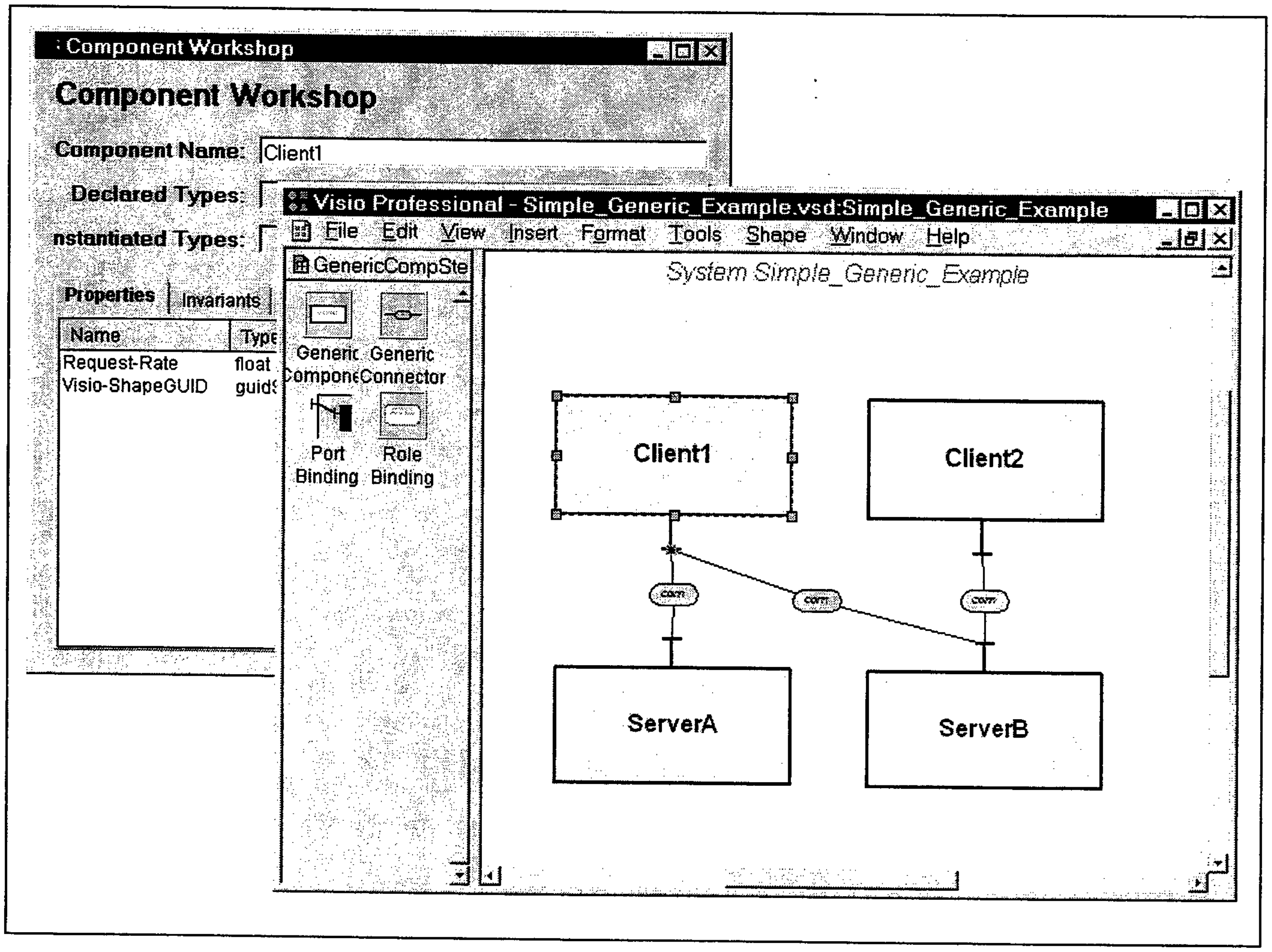
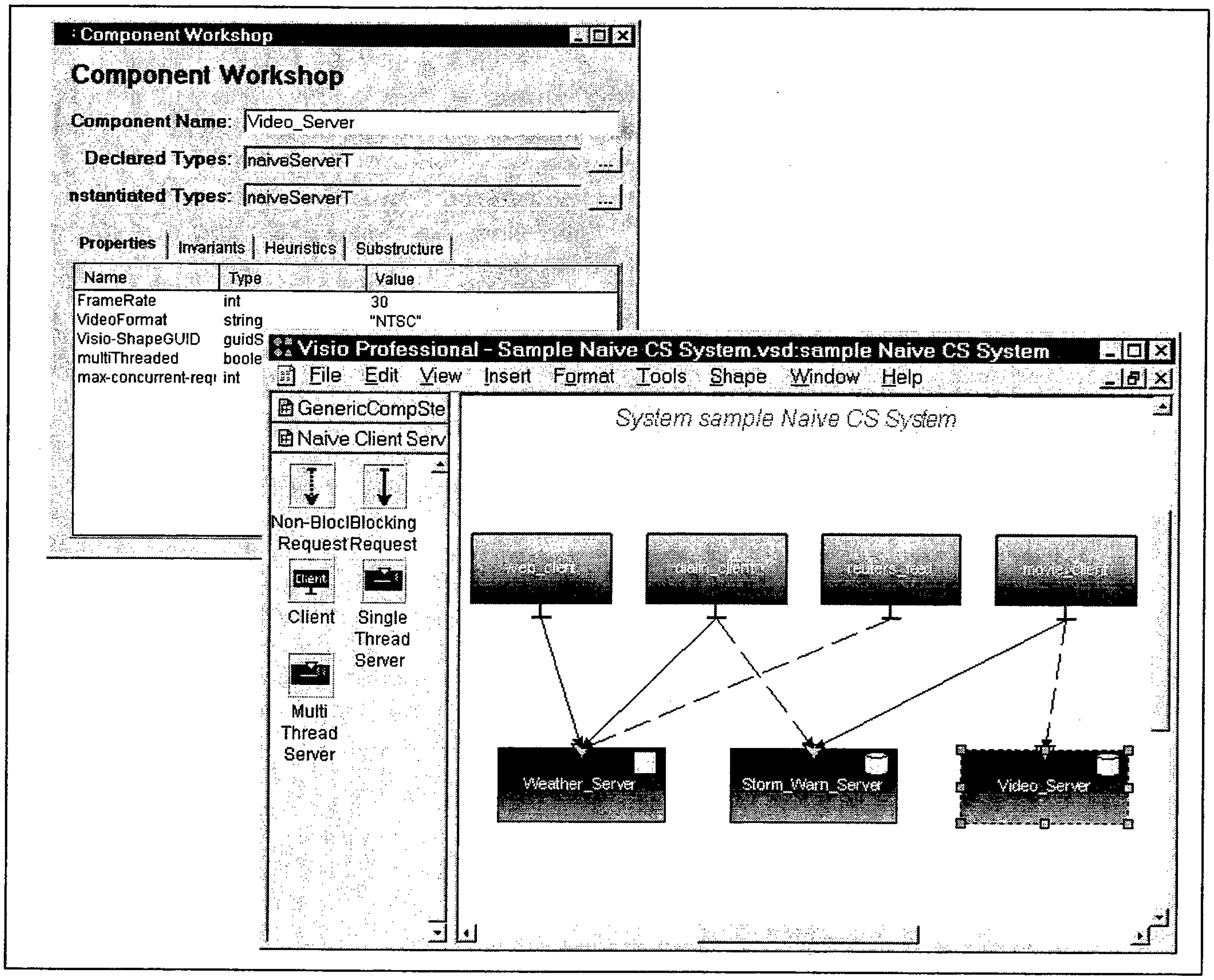


Figure 2.1: A simple design specified with the generic Armani design environment

2.2 Building an environment for a naïve client-server style

To illustrate how the generic Armani design environment can be customized to support design done in a specific architectural style, this section describes the steps that an environment developer (or an architect interested in creating a custom environment) would take to extend the generic environment to support the *naïve client-server* style. Though simple, this style is applicable to a wide range of traditional, two-tier client-server type systems.[3] Figure 2.2 shows a screenshot of the Armani design environment after it has been customized to support this style.

[3] I describe this style of architecture as a *naïve* client-server style because many of the details and design expertise that make the client-server style of design useful and powerful have been omitted to provide a clear example of how Armani is used. Chapter 7 provides a full description of a set of related styles that build on the naïve client-server style to create significantly more powerful and useful styles and design environments.



Component Workshop

Component Name: Video_Server

Declared Types: naiveServerT

nstantiated Types: naiveServerT

Properties | Invariants | Heuristics | Substructure

Name	Type	Value
FrameRate	int	30
VideoFormat	string	"NTSC"
Visio-ShapeGUID	guidS	
multiThreaded	boole	
max-concurrent-req	int	

Figure 2.2: The Armani design environment customized for the *naïve client-server* style.

The following three steps are all that are required to customize the generic Armani system to support the naïve client-server style.

1) **Capture the design expertise appropriate for naïve client-server systems with an architectural style specification**. The environment designer uses the Armani design language to capture this expertise. This style specification defines extensions to Armani's design vocabulary by defining three new component types called *clients*, *single-thread-servers*, and *multi-thread-servers*, two new connector types called *blocking-request* and *nonblocking-request*, and four new interface types that the clients, servers, and client-request connectors use. These vocabulary specifications include a definition of the structural semantics of the components, connectors, and interfaces, a set of properties that the individual clients, servers, and client-request's possess, and constraints on the values of properties and modifications allowed to the basic vocabulary elements.

Architects using the customized environment are therefore able to explicitly use *client* and *server* components that have well-defined structural semantics and properties as primitives in their designs. To illustrate this idea, the type definition for the client component type specifies, amongst other things, that all client instances have interfaces that can interact properly with client-request connectors and that all client instances have a property that describes the average rate at which they will send requests to their associated server(s). Likewise, the servers define how many concurrent clients they can handle, and how quickly they are able to process client requests.

The style specification also includes a collection of design rules that define valid architectural topologies, acceptable ranges for various design properties, and analyses that can be performed on client-server designs. Examples of the specific design rules provided in the naïve client-server style include disallowing client-client connections to ensure that all display information is received through a client-server connection, and a rule that maintains a workable ratio of clients to servers to insure that the servers are not swamped by client requests.

The case study presented in section 7.2.4 gives further details about this style specification. As the case study indicates, this simple, declarative, architectural style description requires only 35 lines of Armani code to specify and the entire environment (including its specification) took only 2.75 hours to create. This compares strikingly to a ground-up environment development project that could conceivably require hundreds of thousands, or even millions, of lines of code.

2) **Create custom visualizations for the style's components and connectors**. This step is optional, as standard visualizations can be applied to new types of components and connectors. Creating custom visualizations for specific component and connector types can, however, provide architects using the environment with rapid feedback about their designs. Armani's visualization engine is built on top of the Visio drawing environment [Visio99]. As a result, this step is performed using the visualization customization capabilities provided by Visio and specifying a mapping between the Visio-based visualization objects and the underlying Armani specifications of the components and connectors. A designer creating a custom visualization for a component or connector type can choose to make a trivial visual modification such as assigning unique colors to each type used (a matter of selecting the appropriate color from a dialog box). Alternatively, he can create arbitrarily rich visualizations for component and connector types using Visio's SmartShape™ technology [Visio99] and Armani's programmatic visualization interfaces.

3) **Load the defined style and visualizations into the environment**. This step is as simple as selecting a pair of files from a dialog box of the running Armani environment. The loaded files immediately configure the environment to support design in the naïve client-server style.

As these three steps indicate, it is straightforward to specialize the generic Armani environment to support specific styles of architectural design. Once this customization is complete, the environment provides a vocabulary of design elements that is tailored to this naïve client-server style (clients, servers, remote-procedure-calls), and a set of design rules that both guides architects working in this style towards appropriate design decisions and explicitly prevents them from making a number of poor design decisions.

2.3 Incrementally adapting the naïve client-server environment

Although the naïve client-server environment just described provides a more effective platform for designing client-server systems than the generic environment originally described, it can be customized even further as the environment user's understanding of client-server design evolves. Suppose that after using the naïve client-server environment for a while, it becomes apparent that the naïve style's two-tier client-server model has some significant limitations. Specifically, in the traditional two-tier client-server style of architecture, the display and user-interface functionality of an application resides in the client components and the generic data storage capability resides in the server components. The difficulty arises in deciding where to add the application-specific data processing capability.

Table 2.3 illustrates three architectural options for how this capability can be divided between clients and servers. The naïve-client-server style can support either of the first two options, "Thick Client/Thin Server" and "Thin Client/Thick Server". Each of these options, unfortunately, has significant drawbacks. In the Thick Client/Thin Server model the clients encapsulate the data processing capability of the system and the servers simply provide the raw data on which the clients operate. As a result, the clients tend to be large, complex programs in their own right. Furthermore, modifying the system's application-specific processing requires changing all of the clients in the system. Because the client components may be broadly deployed and distributed this can be a large, time-consuming, and logistically tricky task.

The Thin Client/Thick Server model addresses many of the limitations of the Thick Client/Thin Server model. The Thin Client model moves the application processing capability to the server components. The client simply displays information and all of the application processing takes place in the server components. As a result, modifications to the application processing logic can be made centrally by changing the server component. Unfortunately, the Thin Client model has two significant drawbacks of its own. The first is that it puts a much greater processing load on the server. Depending on the processing demands of the application and the expected ratio of clients to servers this might be a problem. A second, and more subtle issue, is that the data processing capabilities of the server are now more tightly intertwined with the data representation and storage capabilities of the server. If designed properly, a thick server is certainly capable of keeping these two functions separate and reasonably independent. Putting them in the same component, however, requires significant discipline on the part of system designers and programmers to keep the two functions independent.

A third alternative, the *Three-Tier Client-Server* style addresses the limitations of both the Thin Client and Thick Client models just described. In the three-tier style both the clients and the datastore servers are kept very specialized. The clients, called *data views*, simply display information for the user, and *datastore* servers simply store and retrieve data. All of the data processing capabilities required by the system are stored in a new type of component, the *application logic server*, that mediates the interactions between the clients and the datastore servers and provides the application's processing logic. With this approach, both the clients and the datastore server components retain the advantages of remaining "thin" - lower processing demands, highly specialized functionality, and easy component interchangeability and modification - without requiring that their counterparts become "thick" and suffer the subsequent limitations. As a further benefit, a system's application-specific data processing capability can be cleanly encapsulated in the system's application logic servers, improving system maintainability and modularity.

As users of the naïve-client-server style design environment (shown in figure 2.2) develop an understanding of the three-tier style of design they can incrementally evolve their environment to support design done in the new style. To do so, they simply need to update the design expertise loaded into the design environment. The process of evolving an existing design environment to support a new style of design is the same as the process described in the previous section for creating a new design environment. The first step is to add the new vocabulary types and rules for how those vocabulary types can be used. The second step, which is optional, is to add new visualizations for the design vocabulary elements. Finally, the new style specification and visualizations are loaded into the Armani environment and the modification is complete.

Depending on the degree of similarity between the original environment and the new environment this change can range from trivial, requiring the addition of a only a few lines of design rule specifications, to a significant rewriting of the basic design vocabulary and design rules. Because the original naïve-client-server provides a solid conceptual basis on which to build the three-tier style, updating the naïve-client-server environment to support the three-tier client-server style falls somewhere between these extremes.

The first step is to specify the new design vocabulary elements and design rules that will be the building blocks of the three-tier client-server style. The vocabulary of the three-tier style extends the naïve style with two additional types of components, an *application logic server* and a *datastore server*, along with a new *db-query* connector type. Application logic server components contain the application's data processing functionality. Datastore servers are databases that simply store and retrieve the raw data used by the system. Both of these new component types are *subtypes* of the naïve-client-server style's *server* component type. As a result, they are specified by describing how they extend the basic server component, rather than with a ground-up description. As a result, their descriptions require less than ten statements each. *Db-query*, the new connector type, is added as a connector type through which database queries are passed and the results of those queries are returned. The generic *client-request* connector supplied by the naïve-client-server style remains available in the three-tier style.

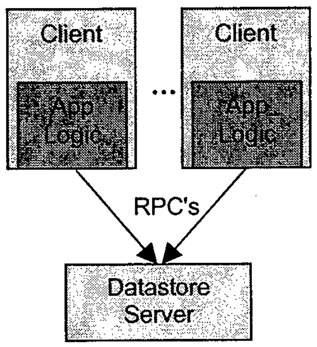
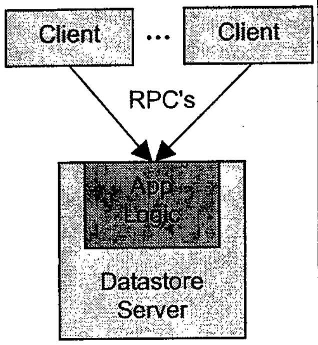
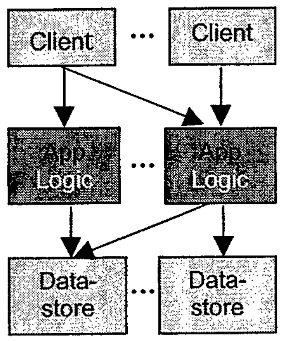
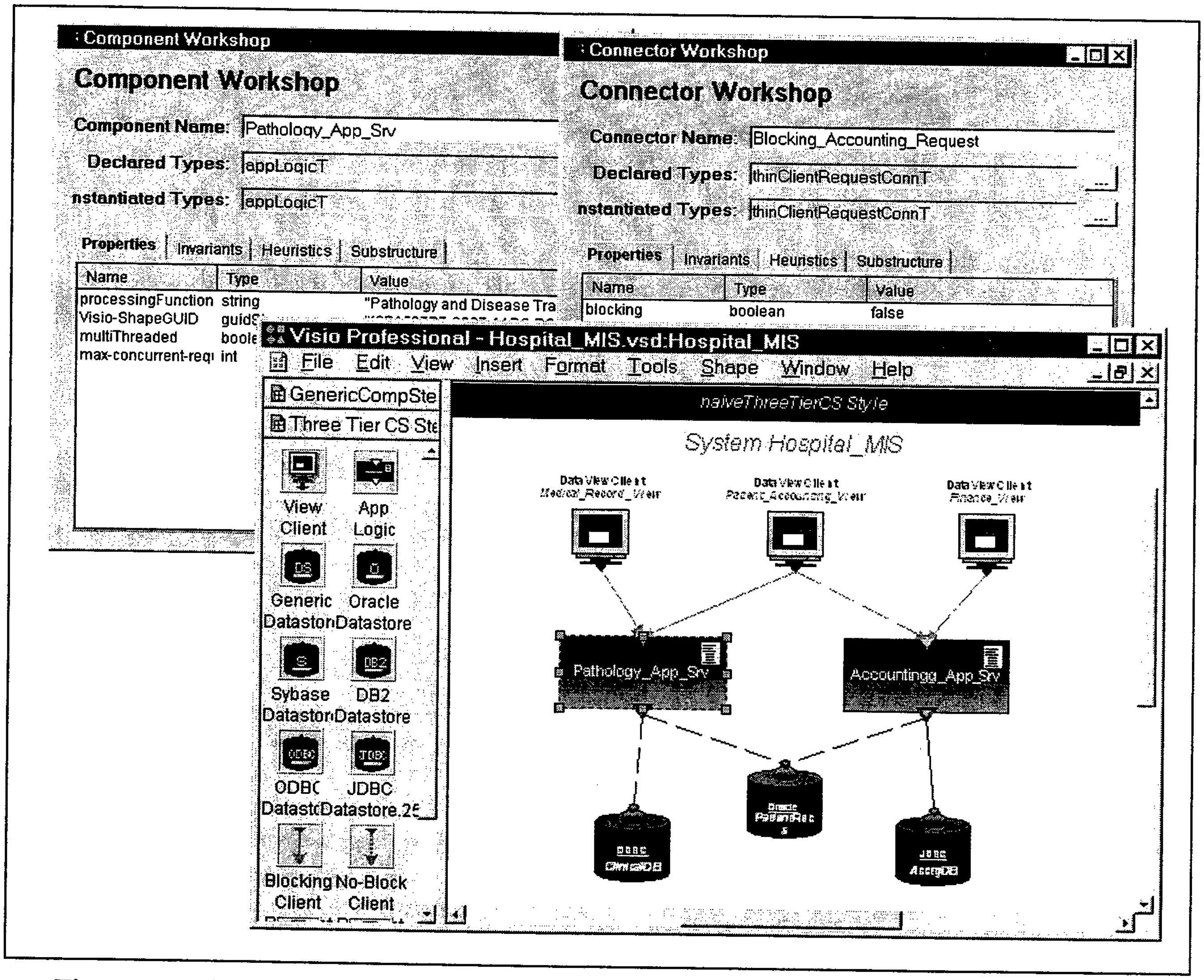
Approach	Thick Client/Thin Server	Thin Client/Thick Server	Three-Tier
Diagram	 Client Client App Logic App Logic RPC's Datastore Server		 Client Client App Logic App Logic Data-store Data-store
Description	Embed application logic in **client** components. Clients decide what data to retrieve from servers. Servers simply retrieve requested data.	Embed application logic in **server** components. Clients only send simple requests and display results. Server aggregates and processes data, returning processed results.	Add a third layer of abstraction and a new type of component, *App Logic*, that embodies the application logic and retrieves and aggregates data from the datastore(s).
Benefits	• Simpler datastore • Reduced server processing load	• Simpler clients • Updates to application logic made at server side propagate to all clients easily and quickly	• Clean separation of component responsibilities • Lightweight clients • Impact of modifications localized. • Easy propagation of component modifications
Drawbacks	• Large, complex clients • Updates to application logic in installed base of clients is difficult • Dependencies between client and server	• Data storage and representation is tightly tied to application data processing • Potentially high server processing requirements	• Potential performance hit due to extra layer of indirection • Additional potential points of failure.

Table 2.3: Overview of approaches for adding application logic to client-server systems

After specifying the additional vocabulary elements, it is necessary to add a set of design rules that describe legal configurations for systems in this style. Because one of the primary goals of the three-tier style is to separate the application logic from the client's display capabilities and the system's data storage capabilities, the first two rules that we add state that client components may only be connected to application logic components, and that datastore components may only be attached to application logic components. As a result, it is not possible to attach a client component directly to a data source. Next, we add topological constraints stating that all connections between clients and application logic servers must be made with client-request connectors and that all connections between application logic servers and datastores must be made with db-query connectors. Finally, we add rules that describe valid connector/component interface pairs.

In addition to the basic topological constraints, additional design rules are added that describe required performance characteristics and appropriate ratios of the various client and server components to maintain acceptable performance and reliability. This set of design rules configure the three-tier enabled design environment to constrain designs of systems created with the environment to those that would be expected from the informal style description presented earlier. As the new style is further refined, the environment can be easily and incrementally updated as well.



Component Workshop

Component Name: Pathology_App_Srv
Declared Types: appLogicT
nstantiated Types: appLogicT

Name	Type	Value
processingFunction	string	"Pathology and Disease Tra
Visio-ShapeGUID	guidS	[illegible]
multiThreaded	boole	
max-concurrent-req	int	

Connector Workshop

Connector Name: Blocking_Accounting_Request
Declared Types: thinClientRequestConnT
nstantiated Types: thinClientRequestConnT

Name	Type	Value
blocking	boolean	false

Figure 2.4: The Armani design environment customized after extending the *naïve client-server* style to support the *three-tier client-server* style

After specifying the conceptual additions required to evolve the client-server environment, we will need to add new visualizations for the datastore and application logic servers, as well as the new db-query type connector. These visualizations are created by drawing the new shapes with the Visio drawing tool and adding the shapes to the new style's palette of design elements. This step requires only that the environment developer drag the shapes from the drawing area and drop them on the palette. After the shapes are dropped on the palette the user is prompted to specify the mapping between the visual shapes and their underlying design types. At this point, the environment customization specification is complete and Armani will support design done in the three-tier client-server style. Figure 2.4 shows a screenshot of the completed three-tier client-server design environment, complete with a design built using the style.

2.4 Summary

This chapter has broadly illustrated the Armani approach by providing a walk-through for how the Armani system can be used to create custom software architecture design environments. Subsequent chapters will present the design language and environment in greater detail, followed by a set of detailed case studies that demonstrate Armani's breadth and utility.

Chapter 3
Related Work

There are four primary areas of related work that have inspired the direction of this dissertation, form a base on which it builds, and/or address similar problems. The first and most influential of these areas is software architecture description languages, tools, and environments. The second area is general purpose software specification languages. That is, languages for specifying software designs that are not specifically concerned with the architectural level of detail and abstraction. The third area of interest is computer-aided design and analysis tools used for designing both physical objects and software. Finally, the fourth area is studies of the abstract process of design. These design theories provide useful guidance for the creation of software architecture design tools. They also guide the selection of design tasks that are readily amenable to computational assistance.

3.1 Software architecture description languages, toolkits, and environments

Over the past decade numerous Architecture Description Languages (ADLs) have been created for describing the structure and behavior of software systems at the architectural level of abstraction. Most of these ADLs offer a set of tools that support the design and analysis of software system architectures specified with the ADL. As we shall see in the ensuing discussions, however, none of these ADLs offer sufficiently rich support for the lightweight and incremental capture of a broad range of software architecture design expertise, nor for incremental customization and adaptation.

3.1.1 Aesop

The Aesop system [GAO94] is a generic, configurable software architecture design environment that can be customized for use with specific architectural styles. Aesop's infrastructure consists of a graphical user interface (GUI) for manipulating visual representations of architectural designs, a database to store and manage designs, and an event system for integrating design tools. In their "raw" state, these building blocks constitute a generic architectural design environment that serves primarily as a box-and-line editor with no semantics underlying the boxes and the lines.

This generic Aesop environment can be extended to support specific styles of architectural design. To create a style-specific design environment, an environment developer creates a set of classes that encode the style's design vocabulary, configuration constraints, and visualizations. This collection of classes defines an *architectural style* in Aesop, which is the primary unit of abstraction for customizing the Aesop environment. These classes *programmatically* define their legal interactions with the environment through the environment's application programming interface (API). Once these classes have been written the environment developer generates a customized Aesop environment by compiling and linking his style-specific classes with the generic Aesop infrastructure.

An important implication of supporting only programmatic customization of the environment is that the process of customizing the environment requires a relatively deep understanding of the internal implementation of the environment infrastructure. Armani, on the other hand, allows architects to customize their environment by loading declarative descriptions of vocabulary and design rules into their environment.

The experience I gained developing and using the Aesop system provided the primary motivation and insight guiding this thesis research. Specifically, Aesop demonstrates that providing a generic, configurable design environment infrastructure can significantly reduce the amount of development effort associated with creating customized software architecture design environments. Further, Aesop demonstrated that the architectural style abstraction can be an effective way to capture broad design expertise.

Unfortunately, Aesop's model for capturing design expertise and customizing design environments also has the following five fundamental limitations.

1) *Aesop's styles are heavyweight, monolithic and inflexible*. Aesop's style specifications capture interrelated design expertise in the form of imperative programming code. This expertise is carefully programmed as a coherent collection of classes that are intended to work together. The style specification is, however, scattered throughout a large body of environment implementation code. It is difficult for an environment designer trying to modify a style specification to cleanly separate the style's vocabulary, composition constraints, and analyses. As a result, modifying a style can have significant unintended consequences. Likewise, this tight intermingling of vocabulary, composition constraints, and analyses also makes it difficult to reuse pieces of a style definition in the creation of a new style.

2) *The semantics of an Aesop style specification are implicit in that style's environment implementation code*. As a result, it can be difficult to determine the semantics of the style itself, as they are tightly intermingled with implementation details. An architecture description language such as Armani that is explicitly geared towards capturing architectural style specifications can significantly ease the processes of creating an architectural style, understanding the semantics of a style, and extending the style to capture new design expertise.

3) *The range of design expertise that an Aesop style specification can express is limited*. In the Aesop system, and in the formal work on styles of Allen, Abowd and Garlan [AAG95], a style specification defines invariant aspects of all systems built in the given style. As a result, it is difficult to make use of design expertise that is not readily expressed as an invariant over all instances of systems built in the style. Specifying design heuristics, for example, can be difficult or impossible in Aesop.

4) *Inflexible scoping*. All vocabulary and composition rules in the Aesop system have a scope of a single style. Styles provide little assistance for capturing or using design expertise

that applies either across multiple styles, to only a context-dependent subset of the elements in a design, or that varies in applicability depending on context. As a result, it is not possible for architects to add localized constraints to a design or to use their judgement in relaxing or overriding rules without adapting the entire style itself. As the first limitation pointed out, adapting an Aesop style is a heavyweight operation that is likely to have repercussions beyond the intended modification.

5) *Limited support for experimentation*. Although a software architect using an Aesop-based style-specific software architecture design environment can experiment with designs done in a given style, Aesop's use of monolithic styles makes it difficult for a designer to adapt an existing style, create a new style, or experiment with style-independent design expertise. In Aesop's basic environment development process a "style developer" customizes a design environment for use with a particular style that he hands off to the architects who use it to design specific systems. The architects using the customized environment are given little in the way of tools for customizing the environment or quickly adding design expertise that they discover to the environment. When they want to evolve or improve the environment they generally need to send the request to the style developer who will make the changes and return to them an updated environment. As a result, lightweight, experimental environment evolution is not supported by Aesop.

To illustrate these limitations, consider a designer using Aesop who wants to specify that a particular configuration of components and connectors may have no cycles. She does not want to change the entire style, because this particular constraint is only applicable to one part of the system being designed. With Aesop, the style itself must be modified if this constraint is to be automatically maintained because all constraints must be defined as style-wide invariants. Further, even if our designer was willing to make the changes needed to the entire style, the monolithic and heavyweight nature of style specification in Aesop makes the process of adapting the style to support the changes prohibitively expensive.

Another case worth considering occurs when the system architect is not sure whether she should allow cycles in the systems she is designing. She would like to be able to experiment with a number of options before binding the decision. Aesop provides little support for experimentation of this sort. The process of adding constraints to a style requires significant programming and a relatively deep understanding of the internal structure of the Aesop system. The high cost of experimentation in style development discourages the designer from exploring a wide variety of options.

As the rest of this dissertation argues, Armani's model for explicitly and incrementally capturing design expertise and customizing the software architecture design environment addresses these five limitations. Armani addresses these issues by providing an architecture-specific declarative textual language that cleanly separates different aspects of architectural specification, by providing modular, orthogonal constructs for expressing architectural designs and design expertise, and by providing a much more flexible and incrementally adaptable design environment infrastructure than Aesop.

3.1.2 Acme

Acme [GMW97] is a generic, extensible software architecture description and interchange language. The Acme language grew out of a project to support the interchange of architectural descriptions between a wide variety of different architecture design and analysis tools. Because most of the target tools used their own idiosyncratic languages and representations for describing software architectures they did not naturally work well together. Acme was created to address this issue. Acme provides a flexible, structure-based architectural representation language that maps readily to many popular architecture description languages. Recognizing that many of the ADLs and tools that are designed to integrate with Acme would lose information if they only translated their structural information, Acme also supports the annotation of system architectures with non-structural information through its property construct.

As Acme evolved it became a useful architectural description language in its own right. Because it is generic, extensible, and designed to interact well with a broad variety of architectural design tools, the current version of Acme (version 1.0) provides a standard architectural representation that works well as a platform for building new architecture design tools and languages. Given this capability, the Armani design language was created as an extension to the Acme language. Armani builds on Acme by adding a more rigorous type system, a predicate language, and constructs for representing design rules in the form of design analyses, invariants, and heuristics.

Building Armani on top of Acme has two significant advantages. First, it leverages a significant amount of design and development effort that went into creating the Acme. As a result, Armani is able to avoid repeating many of the mistakes that were made (and subsequently rectified) in designing Acme. Second, because Armani is built as an extension to Acme, it is trivial to strip and/or translate Armani's additional constructs to create an Acme representation of an Armani system description. In this way, architectural specifications provided in Armani can be analyzed by other design tools that recognize the Acme standard.

The Acme language and its associated toolkit are effective in their role as the foundation for the Armani system. They are not, however, sufficient on their own to solve the problems that Armani addresses. Specifically, although Acme is useful as a platform for building and integrating sophisticated software architecture design tools and languages, the language does not provide the richness of expression that Armani does for capturing design expertise. Armani's extensions to Acme allow it to satisfy the goal of capturing software architecture design expertise in a way that Acme is unable to do on its own. A second fundamental limitation of Acme is that it neither provides nor specifies an architecture design environment. Acme defines a notational standard to which design tools and environments can be written but it does not provide those tools itself. As a result, Acme provides a solid generic infrastructure on which to build the Armani system, but it does not obviate the need for Armani.

3.1.3 Domain-Specific Software Architecture (DSSA)

DARPA's Domain-Specific Software Architecture program (DSSA) [MG92] provided early research recognition that reusable reference architectures hold significant promise for improving the design, analysis, production, and maintenance of complex software systems. The project was also an early attempt at capturing software architecture design expertise for specific application domains. The DSSA project developed reference architectures as well as design and analysis tools for a number of software development domains. As the project's DARPA origins suggest, the domains studied – avionics, command and control, GNC (guidance, navigation, and control), and adaptive intelligent systems – were selected for their strategic importance to the United States Department of Defense (DoD).

The DSSA approach to creating reference architectures and tools differs significantly from Armani's approach. The DSSA approach captures, stores, and organizes design expertise according to the application domain from which it was captured. An example of such a domain is the avionics domain, or military command-and-control systems. Armani, on the other hand, organizes design expertise according to the *style* of architecture for which the expertise is applicable. Styles capture common structures and approaches to design that can span multiple application domains. The client-server style and the master-subprogram style (see sections 7.2.3 and 7.2.5, respectively) are two examples of architectural styles that are applicable to multiple application domains. The distinction here is subtle but important.

A DSSA domain-specific reference architecture generally selects a single architectural style that is appropriate for the domain being studied. Doing so allows specialized architectural design and analysis tools to be developed for use in the chosen domain. The downside is that the expertise captured using the DSSA approach is applicable to only a narrow range of system design efforts. Further, because there is minimal shared conceptual basis or infrastructure between the various DSSA reference architectures, tools developed for one DSSA domain are not necessarily reusable or adaptable for use with another DSSA domain.

In Armani, on the other hand, the definition of an architectural style is orthogonal to the application domain in which the style will be applied. As a result, design expertise captured with Armani can be applied to a broader range of system design efforts than the expertise captured in a DSSA toolkit. Further, Armani's generic, adaptable infrastructure for creating custom software architecture design environments obviates the need for the ground-up tooling development effort associated with the DSSA projects. The cost of creating a style-specific architectural design environment with Armani is dramatically lower than the cost of producing a DSSA reference architecture and toolkit.

Overall, DSSA technology complements the Armani technology. Both domain-based design expertise and style-centric architectural design expertise play important roles in most large software development projects.

3.1.4 Automated design critiquing

Emerging work by Robbins and Redmiles [RHR98] refines Fischer's work on automated critiquing [Fis+87] and applies it to the software architecture design process. To demonstrate the approach, they have produced a software architecture design environment called *Argo* that integrates a critiquing capability with traditional design environment functionality. In Argo, *critics* are background tasks that continually monitor the state of a design and the actions that a designer takes to flag problems and to offer the designer suggestions and guidance.

As in Armani, the ability to capture and leverage software architecture design expertise is a central benefit of Argo. Argo, however, takes a significantly different approach than Armani. The design expertise that Argo captures is embedded in the implementation of its critics. These critics are implemented as Java classes that are tightly integrated with the rest of the Argo environment. As a result, Argo's expertise is readily modularized and composed, much as it is in Armani.

Critics provide a mechanism for capturing software architecture design expertise that is largely complimentary to the Armani approach. Critics, however, fall short of fully solving the problems addressed by Armani. First, critics are a relatively heavyweight mechanism for capturing design expertise. Creating them requires significant programming effort and a relatively deep understanding of the implementation of the environment into which they will be embedded. Although critics support an incremental way to add design expertise to a software architecture design environment, such incremental evolution remains out of reach of the environment's end-users. Second, the critics mechanism doesn't support the capture of constraints over specific design instances. Critics are fundamentally associated with a design environment rather than an individual design. As a result, critics are not appropriate ways to annotate architectural specifications with design rules to guide a single system's evolution. Finally, Argo lacks a unifying framework for aggregating and reasoning about the interactions of its critics. As a result, developing a formal foundation for understanding the design expertise captured by Argo presents a significant challenge.

3.1.5 Other ADLs

Numerous other architecture description languages (ADLs) have been developed to capture various aspects of architectural design expertise and provide architects with mechanisms for describing and reasoning about their designs. A brief synopsis of some prominent and relevant ADLs follows, followed by a comparison with Armani.

UniCon (UNIversal CONnector) [Shaw+95] is an architecture description language and toolset that supports the generation and analysis of software systems from architectural descriptions. To produce a software system, a UniCon user typically specifies the set of components that make up his or her system along with a description of how the components communicate and interact. These interaction paths are called *connectors*. UniCon supports a variety of architectural connectors that are commonly used in production software systems. Examples of these connectors include *Unix Pipes*, *Remote Procedure Calls* (RPCs), and *SQL Queries*. Because the interaction techniques used by these connectors are well understood and standardized, UniCon is able to generate code to create the system's connectors, the component packaging needed to hook its components together appropriately, and the system's build files. As a result, UniCon can dramatically increase the level of abstraction at which systems are actually constructed.

Rapide [Luc+95] provides an environment and language for modeling and reasoning about software architectures. A Rapide system model describes system structure and behavior. Rapide provides is the ability to simulate the behavior of the modeled system, given a stream of inputs. The resulting system behavior is represented as a Partially-Ordered Set (POSet) that defines sequences of events that could occur as a result of the system inputs. An architect can analyze these POSets to detect potentially anomalous emergent system behavior directly from the architectural specification.

Wright [AG96] is an architecture description language designed primarily to describe the protocols of interaction used within a system or throughout an architectural style. Like Rapide, Wright's primary analytical leverage comes from its ability to describe and analyze system behavior; particularly emergent system-wide behavior. Unlike Rapide, however, it supports static analysis of architectural specifications rather than simulation. As a result, Wright's analytical results are more comprehensive than the results generated by a Rapide simulation. That is, a Rapide simulation evaluates system behavior only for a specific sequence of inputs. A static analysis of a Wright specification, on the other hand, is shown to hold over all possible input sequences. Wright's support for the specification of architectural styles, rather than just specific system instances, dovetails nicely with its static analysis capabilities. The results of a Wright analysis of an architectural style's protocols of interaction demonstrates emergent properties that will hold for all systems built according to that style's specification.

Each of these ADLs support the description and analysis of one or more important aspects of software architecture design. They provide architects with the capability to express the design of a specific software architecture, analyze a set of properties of that architecture, and, in the case of UniCon, to generate working systems from the architectural specification.

Although each of these ADLs provide compelling capabilities, they have some significant limitations as well. First, all of these ADLs (willingly) trade off some flexibility and generality for greater analytic and generative capability. That is, although each ADL is good at providing its specific benefits, none of them are particularly general or extensible. Second, all of these tools required years of research and engineering to develop, yet none of them can be readily modified or adapted by their users as their needs evolve. As a result, the tools will require significant maintenance effort and expenditures if they are to be kept current as design needs change.[4]

The explicit decision to design Armani as a radically user-configurable design environment addresses the high cost of initially producing such an environment. By providing a configurable infrastructure and language that supports many of the functions and capabilities that had to be built from scratch for these ADLs, the effort to build new design and analysis tools and capabilities can focus on the providing the incremental benefits that these tools promise rather than rebuilding standard infrastructure and architectural concepts.

[4] An updated version of UniCon was released after the publication of [Shaw+95] that provides a much more flexible mechanism for adding new connectors to the environment. This is a big step towards greater flexibility, though it still requires a significant understanding of the implementation of the UniCon tools to successfully modify them.

The research presented in this dissertation explores a different part of the design space for software architectural design tools and environments than these three ADLs. Rather than trade off generality and broad applicability for additional power, Armani trades off some analytical power for broader generality, applicability, and evolvability. By building on top of the Acme interchange standard, however, Armani is able to address the need for powerful design analysis and generation capabilities that are not readily available with Armani. Specifically, an Armani description can encode design information for use by Rapide, UniCon, or Wright-based analyses. Rapide, UniCon, and Wright all provided significant lessons in the design and implementation of the Armani language and environment. As a result, Armani is able to leverage some of their analytical power while providing a more general and extensible set of architectural concepts and infrastructure.

3.2 General purpose software specification languages

There are many general software design and specification languages that are complementary to the architecture-specific languages described in the previous section. In most cases, these languages are intended for describing and specifying software at more detailed levels of abstraction than the architectural level. As a result, none of them are particularly good for describing software architectures.

The primary limitation these languages face for describing software architectures is that they don't support a sufficiently specialized set of constructs and concepts for dealing with architectural specification and analysis. Their primitives are generally geared towards specifying data structures and algorithms. An ADL, on the other hand, must describe the components of a system, how those components interact, and their non-functional properties. As a result, an architect trying to use such a language for architectural specification must generally expend a significant amount of time and effort building a set of architectural constructs from the language's primitives before using the language for architectural description.

Although they are not the ideal choice for many architectural specification tasks, many of these languages embody good ideas that were borrowed by and incorporated into the design of the Armani system. This section will discuss some of the most influential general purpose software specification and design languages, how they have influenced the design of Armani, and why they do not sufficiently address the problems that the Armani system tackles.

3.2.1 UML

The Unified Modeling Language (UML) [BRJ98] is a set of graphical notations for specifying the design of object-oriented software systems. UML was formed by merging a number of popular object-oriented design notations, including OMT [Rum+91], Booch [Boo94], and OOSE [Jac94], into a single collection of related object-oriented design notations. As a result, UML includes notations for describing the object structure of a system, behavioral diagrams to indicate how objects interact, and a notation for specifying use-cases. Due to its comprehensive scope and broad commercial tool support, UML has become a popular design notation amongst software engineers and architects building object-oriented systems.

Although at first glance UML appears to be a potential substitute for Armani, the two design languages are, in fact, quite different. UML provides a solid notation for specifying the detailed design of software systems. It is not, however, particularly effective as a high-level architecture description language for the following reasons:

Weak semantics. UML was intentionally created with informal semantics. Surprisingly, this has proven to be one of the language's greatest selling points for practicing software designers. Because the semantics of the notations are only weakly defined, designers using UML need not be particularly precise in their use of the notation. This can be a benefit if the notation is used primarily as a way for engineers to sketch designs on a whiteboard during a meeting. Unfortunately, UML's vague semantics are a weakness when the notation is used to specify a system architecture. Vagueness in the semantics of the notation is likely to lead to confusion amongst the engineers and programmers who have to implement the system. In the worst case this could lead to a complete inability to integrate the individual elements of the system.

Poor support for hierarchy and scaling. UML provides only weak support for the hierarchical decomposition of an architectural design. As a result, large UML designs tend to become complicated, with hundreds or even thousands of connected shapes on a single diagram. A design that is this complicated is difficult both for system implementers to understand and analysis tools to analyze. Armani, on the other hand, provides hierarchical decomposition as a fundamental language capability.

Tight ties to implementation structure. The UML notations are primarily intended to capture the implementation structure of a system. Further, it has only very weak support for the notions of interfaces that are separate from implementations. This is a benefit to tools that generate code skeletons from UML structural definitions. From an architectural design and analysis standpoint, though, it can be difficult to express abstract design concepts and interactions using UML. Armani, on the other hand, explicitly focuses on describing abstract architectural structure and interactions. In this way, UML and Armani can be used together in a complementary way. Armani is generally a better option than UML for specifying high-level, abstract architectural aspects of a software design. UML can be embedded within an Armani specification to describe the detailed, lower-level design of a system.

There is significant effort underway in the software architecture research community to integrate UML's popular implementation-level design concepts with more abstract software architecture design languages and tools [Rob+98]. For a detailed discussion on the relationship between architecture description languages and object modelling techniques see [Mon+97].

3.2.2 PVS

SRI's Prototype Verification System (PVS) [OS97] consists of a language for specifying system designs and a semi-automated theorem prover to verify properties of those designs. PVS's use of automated theorem proving allows it to support the specification and verification of complex system designs that would be difficult or impossible to verify with more traditional type-checking methods.

PVS is relevant to Armani primarily because, like Armani, the PVS type system supports the specification of complex predicates for capturing designs and design constraints [. Although many of the details of Armani's syntax and semantics differ significantly from those of PVS, the fundamentals of the two languages are actually similar enough that the PVS semantic core serves as the foundation for Armani's formal semantic specification. As a result, Armani is able to leverage and reuse a well established formal representation and logic as its semantic underpinnings.

PVS does not, however, solve the basic problem that Armani addresses – rapidly and incrementally developing custom software architecture design environments. PVS performs a single design task – formal verification and theorem proving – well for a variety of architectural domains. Although PVS provides helpful guidance and a formal foundation on which to build, it is not readily customized to work with specific architectural styles.

A second, and perhaps more subtle, reason that PVS is not by itself a sufficient basis for an extensible architectural design representation is that the result of a PVS verification is not necessarily binary. That is, the result of a verification test on a PVS specification can be "yes, the design is verified", "no, the design is not verified", or "maybe, if the following proof obligations hold". This behavioral characteristic arises because the PVS specification language is sufficiently rich to render typechecking a specification undecidable. As a result, integrating other interactive design tools with PVS can prove challenging. By insuring a binary and relatively fast response to a tool's query about whether a design satisfies its constraints, Armani can provide interactive feedback not only to human users but also to automated tools.

As a result, PVS and Armani are complementary technologies. The PVS system allows verification and validation of more complex designs than Armani at the expense of its utility as a general design representation and integration framework. Armani, on the other hand, provides a flexible, customizable, environment and tool integration infrastructure. One way in which the two technologies could be used together, for example, is to use PVS to determine the internal consistency of Armani type and style specifications. Determining whether an Armani style specification is internally consistent requires more theorem-proving capability than Armani possesses. PVS provides such a capability and Armani's integration framework should allow the two to work together. This issue is addressed in greater detail in Chapter 9.

3.2.3 Larch

Larch [GHW85] is a software specification language that supports the definition and composition of partial algebraic program and abstract data-types (ADTs). The aspect of Larch that is most relevant to the Armani design language is Larch's support for the capture of important aspects of ADTs in modular, independent units called traits. These traits can be composed to capture the semantics of full, rich ADTs.

The Armani design language makes use of and builds on Larch's ideas about trait composability and partial specification. Specifically, it uses a similar approach for constructing architectural styles and compound vocabulary types out of individual design rules, structural and property specifications. As a result, Armani can address the architectural design issues that Larch, with its emphasis on abstract data type specification, is not intended to capture.

3.2.4 Traditional formal software specification languages

In addition to the ADLs described above, there has been a significant effort by members of the software architecture research community to use formal methods to provide a mathematical foundation for describing software architectures and architectural styles. Formalisms such as Z [Spi89, AAG95], CSP [Hoa85], and the Chemical Abstract Machine [IW95] have been used to model software architectures. Although these formalisms can be useful for rigorous reasoning about architectural designs, they fail to provide either an easy way for designers and domain experts who are not deeply familiar with formal method to capture expertise, or mechanisms for automating the formalisms that they do capture in a working architectural design environment. Further, these formalisms often prove awkward for describing software architecture specification because their basic constructs are not geared for doing so. Although it is often possible to build software architecture constructs out of the primitives that these formalisms provide, much of the clarity that the formalisms exhibit for specification at lower levels of abstraction is lost in the process.

Although the Armani system does not emphasize the same degree of rigor for reasoning about architectures that these formalisms provide, it provides a greater degree of flexibility for capturing design expertise, realizing that expertise in a design environment, and experimenting and exploring the range of possible designs. Armani is, in fact, complementary with these more formal software architecture description models – formal specifications can be embedded in an Armani description to provide a more rigorous specification for various aspects or pieces of a system described with the Armani language and tools.

3.2.5 Law-Governed Systems

Minsky's work on Law-Governed Systems (LGS) [Min91, Min96-1, Min96-2] describes an approach for guiding the evolution of software systems with a collection of *laws*. These laws are generally represented as prolog-based rules that can be evaluated during system development, maintenance, or operation. By providing a framework for describing expected (and legal) system evolution paths a system's original designers can guide the subsequent evolution of the system and flag evolutionary paths that are likely to violate the system's basic foundations.

The LGS approach to capturing and enforcing system-wide evolutionary laws bears a striking resemblance to the Armani approach to capturing software architecture design constraints. Both approaches provide a framework and mechanisms for capturing software design expertise in the form of design rules and an automated checking mechanism to ensure that those rules are followed in the construction and evolution of software systems.

The Law-Governed System approach does not, however, fully address all of the issues addressed by the Armani system. First and foremost, Law-Governed Systems provide minimal direct support for dealing with architectural issues. Although they are designed to work at various levels of software system abstraction, they tend to emphasize issues that occur at more detailed (lower) levels of abstraction than the architectural level. As a result, Minsky's prolog-based laws trade off some of the power that comes from focusing on the architectural level of design for the generality of being applicable for all stages of software design and development.

Armani, on the other hand, focuses on specifying constraints on the evolutionary path of the architectural specification. Although Armani is not as broadly powerful for describing sub-architectural design constraints as LGS, its constructs, concepts, and design primitives provide architects with greater leverage for designing at the software architecture level of abstraction. An architect using LGS as the basis for architectural design needs to create a set of architecture-level primitives and design rules before LGS can be used effectively as an Architecture Description Language. This is a non-trivial, time consuming, and error-prone task. An architect using Armani, on the other hand, can immediately leverage its built-in architecture-level primitives and design rules.

Another limitation to the law-governed system approach is that, historically, the implementations of the law-governed system idea have been tightly tied to object-oriented system design and development techniques (c.f. [Min96-1] and [Min96-2]). It is not clear whether this is fundamental or accidental to the LGS approach, but the heavy emphasis on object-orientation, and method-call interactions in particular, limits the range of architectural styles that one can currently use to design a law-governed system. One of the implications of aligning the LGS approach tightly with object-oriented design is that although LGS provides a way of expressing laws at a wide range of abstraction levels (e.g. detailed implementation laws or high-level module relationships), it limits the overall system organization to a relatively small range of abstractions. Armani, on the other hand, focuses on capturing design constraints at the architectural level of abstraction, but allows a wide range of architectural abstractions for system organization.

Finally, Law-Governed Systems provide little or no support for the rapid creation and evolution of software architecture design environments. LGS provides an abstraction and modeling technique that can be realized and embedded in software design environments but this does not in any significant way to address the need for rapid and incremental environment development and modification capabilities.

3.2.6 Configurable programming environments

The idea of providing generators that can quickly create custom tools for software professionals certainly preceeds Armani. The Gandalf System [Hab+82] and the Synthesizer Generator [RT88] are software tool generators that were influential in the conception and creation of Armani.

The Synthesizer Generator allows tool builders to rapidly create structured editors that are customized for specific languages. These editors dynamically enforce the syntax of the language and provide constructive assistance to programmers using the tools. These editors also provide a programmatic interface for manipulating the attribute-grammar representation of a program.

Gandalf provides a package of infrastructure and generators that can be used to create custom software development environments. Like the Synthesizer Generator, a key part of the Gandalf software development environment generation process is the creation of a customized structure editor. In addition to the structure editor Gandalf also provides mechanisms for handling project management and source code version control.

Much like Armani, both of these tools guide software developers toward building correct software. Specifically, the structured editors created by these tools make it impossible (or at least very difficult) for a programmer to create a syntactically incorrect program, eliminating an entire class of programming errors.

Although both of these tools provide compelling support for creating customized programming environments, neither fully addresses the problems that Armani solves. First, both of the systems focus on providing programming support and, in the case of Gandalf, on supporting teams of programmers constructing systems. The tools are not intended to support the software architecture and design process. Likewise, neither tool supports the ability to explicitly express, capture, and make use of design expertise. Although the person generating custom tools makes use of significant expertise, that expertise is implicitly encoded in each of the generated tools.

Second, neither of these tools allow the end-user to perform any significant environment customization. Like Aesop, they both support a generation model in which the person who creates the custom environment is different from the person who uses the environment. Although the same person can obviously play both roles, the skills and perspective required varies greatly between the two tasks. Armani's support for dynamic end-user environment configuration is a powerful additional capability.

3.3 Computer aided design and analysis tools

The previous two sections have focused on related research in languages and techniques for specifying and reasoning about software designs. Although some of these languages are accompanied by significant toolkits, the primary contributions of these research projects tends to be the abstractions and frameworks that they provide for specifying and reasoning about software designs. Armani builds on this related work but it must also take advantage of research and development efforts others have conducted in building effective computational tools. In this section I present selected research in computer aided design and analysis tools from a wide variety of engineering and design domains.

3.3.1 CAD tools for various disciplines

Computer Aided Design (CAD) has proven to be an effective tool for designers in many disciplines. Architects, mechanical engineers, electrical engineers, and many graphic designers rely heavily on CAD environments to perform their jobs. Systems such as Cadence

[Cadence] for electrical engineering, AutoCAD for mechanical engineering [Autodesk], the Integrated Building Design Environment for building architects [Fen+94], and Corel Graphics [Corel] for graphic design capture the important aspects, models, and tools of design in their particular domains.

Because these systems were built to support design in significantly different domains from software architecture, they do not solve many of the specific issues faced by software architects. Exploring how these tools provide leverage for the designers using them, however, offered many insights into what makes a CAD tool effective, the degree to which design tools in other domains provide end-user configurability (they generally don't provide any), and the kinds of analytical capabilities that their users seem to expect.

The primary insight I gained from exploring CAD tools in other domains was that the tools provide a lot of leverage when they (a) capture and expose the domain's core design expertise, (b) provide analytical capabilities that a designer would have difficulty performing without the assistance of a computer, and (c) allow designers to quickly explore a variety of alternative scenarios before binding design decisions. The design tools that provided all of these capabilities tended to be very effective and popular with their users. Although the tools explored were not directly applicable to the software architecture domain, determining what made them effective proved very helpful in designing Armani.

3.3.2 Tools that manage constraints and rules

Constraint-based programming and rule-based programming have been studied extensively in the AI literature. Borning's Thinglab [Bor79] was one of the early constraint-based environments for manipulating models of the physical world. Thinglab simulated physical environments and allowed a user to specify constraints that needed to be maintained between different objects in the simulated world.

Like Thinglab, Armani allows designers to specify constraints that its environment enforces as a software architecture specification evolves. Unlike Armani, Thinglab was designed to be a very general constraint management system primarily for modeling entities in the physical world. As a result, Thinglab provided little or no support for the abstract concepts and constraints required in software architecture design. Although Thinglab's generality allowed it to be applicable to many domains, by focusing specifically on software architecture design, Armani is capable of providing architects with significantly more design leverage than a general purpose constraint management tool.

Fischer's work on *critics* (see also section 3.1.4) and configurable design environments for various industrial-design domains [Fis87,Fis+87] explored ways to capture design knowledge and design rules in custom, domain-specific design environments. The design environments that Fischer's research group built, however, were generally hand-crafted creations. Unlike Armani, they do not support significant end-user configuration, customization, or expression of design rules. Armani's support for end-user reconfigurability is central to its design and requirements. As a result, although Fisher's critics work provides many useful ideas for this research, it does not solve the full problem that Armani addresses.

3.3.3 Expert and rule-based systems

Rule-based programming systems such as OPS-5 [For81] and CLIPS [Gia93] are generic rule-based systems for capturing expertise. These tools are used by software developers to create expert systems for arbitrary domains. Traditional expert systems are monolithic entities that use inference techniques to derive conclusions from knowledge provided to them as input. These custom-built systems tend to be limited to very specific domains, such as diagnosing automobile problems and suggesting how to fix them.

Because traditional expert systems provide mechanisms for capturing expertise and using that expertise to solve specific problems they seem to be natural candidates for building software architecture design tools. Unfortunately, they fall short of meeting all of the goals and requirements laid out for Armani in three important ways.

First, one of Armani's most important capabilities is the ability to capture small, modular, and composable "snippets" of expertise that can be used (and reused) in an interactive design environment. Rule-based systems generally provide the capability to add or remove individual rules from the inference engine. Experience with these systems indicates, however, that they have a tendency to become very brittle when many rules are added to the system. Armani's modular and focused constructs for capturing design expertise go a long way towards addressing this limitation.

Second, there are many kinds of design expertise that are not readily captured in a set of implicitly fired rules. A prime example is the ability to analyze the throughput and latency of a system using queuing networks.[5] Expert systems traditionally only use collections of rules and knowledge (in the form of assertions) to capture expertise. In order to capture a wide range of design expertise, however, it is important that a custom software architecture design environment building system provide multiple mechanisms for capturing and exploiting design expertise. Armani provides this capability, but the closed-world model of most expert systems precludes them from doing so.

The final limitation of using a traditional expert system as the basis for rapidly developing custom software architecture design environments is the need to include humans in the decision making process. The Armani environment is intended to provide assistance to skilled human designers. Most expert systems, on the other hand, are intended to *be* the expert that makes the important decisions. This model does not work very well in the weakly constrained domain of software architecture. Architectural design decisions are rarely sufficiently clear-cut, mechanical, or well quantified to be amenable to a pure expert-system approach.

Although rule-based systems do not appear to completely obviate the need for tools and techniques to rapidly develop custom software architecture design environments, they could certainly be used as part of the environment infrastructure. As I discuss in Chapter 4, I selected a predicate logic formalism as the basis for capturing design expertise in Armani. It probably would have been possible to use a rule-based system as the underlying formalism

[5] Such an application of queuing networks is described in detail in section 7.2.4's case study.

for the Armani design environment instead but the predicate-based model more cleanly matched Armani's structural architecture specification model.

3.4 The design process

There is a significant body of work on the abstract process and cognitive aspects of design, including studies of design in engineering, industrial, and perceptual domains. Although the research that they describe is not always directly related to rapidly developing custom software architecture design environments, I made use of many of the principles put forth in these studies to guide the design and development of the Armani system. My research leverages this work on design processes rather than directly extending it.

Specific sources that I found particularly useful and interesting for this purpose include Winograd's book of essays on the software design process [Win96], Schön's work on "reflection in action" [Sch83], Petroski's work on engineering design [Pet85], and Norman's discussion of the design of everyday objects [Nor88]. Following the principles developed in these studies helped me design the Armani system to fit naturally into an architect's design process

In addition to these broad explorations of the design process, there are two areas of research specifically related to designing software architectures that significantly influenced my work on Armani – the Gamma, Helm, Johnson, and Vlissides' work on design patterns [Gam+95] and Rechtin's system architecting heuristics [Rec91, RM97]. In the next two sections I discuss these projects and how they relate to Armani in detail.

3.4.1 Design patterns and pattern languages

Design patterns have emerged as a popular mechanism for capturing and communicating proven design techniques that address specific problems. Based on the building architecture work of Christopher Alexander [Ale+77], a design pattern identifies a problem, provides a generic solution that can be tailored in many different ways, and discusses the implications (both good and bad) of using the pattern. Although individual design patterns are helpful, patterns provide significantly more leverage when a coherent collection of related patterns is available to guide the design process than they do in isolation. Such a collection of related patterns is called a *pattern language*.

Gamma, Helm, Johnson, and Vlissides' book "Design Patterns: Elements of Reusable Object-Oriented Software" [Gam+95], popularized the design patterns movement by presenting an initial pattern language for object-oriented software. Since that time, hundreds of articles and books have appeared describing pattern languages for nearly every aspect of software design and development, including software architecture [Bus+96]. There is clearly a wide base of popular support for the use of design patterns.

As I describe in detail in [Mon+97], design patterns and architectural styles are complementary constructs. Design patterns do not, however, solve all of the problems that Armani solves. First, they tend to be informal. Although this informality works well for communicating a general approach to solving a problem, it is very difficult to precisely specify designs with an informal specification and even harder to provide automated evaluation of a design to see if it conforms to such a specification. Second, design patterns tend to be good at solving local problems but limited at providing guidance for putting together a complete system. Finally, design patterns are intended to be primarily a communication medium between people. Although this approach makes them effective for exchanging designs and design knowledge between designers, it also limits their effectiveness as a conceptual foundation for configurable design tools.

Recognizing the complementary nature of design patterns and Armani's approach to capturing design expertise, I have tried to design Armani to be able to take advantage of design patterns. The full Armani design language presented in Chapter 4 does not explicitly provide a *design pattern* construct. It is, however, possible to capture much of the same effect of a design pattern by combining and by packaging related design vocabulary and design rule specifications. These implicit pattern specifications are generally more rigorous than the patterns found in [Gam+95] and related collections, though they lack the first-class status and much of the informal description provided by the explicit design pattern languages. Completing a full integration of design pattern concepts with Armani is a promising area for future research.

3.4.2 Rechtin's system architecting heuristics

In his books "Systems Architecting" [Rec91] and "The Art of Systems Architecting" [RM97], Rechtin outlines techniques and criteria for effective system architecting. One of his primary theses is that system architecting is a heuristic-driven process, regardless of the specific domain in which it is practiced (e.g. Aerospace, software, physical structures...). Throughout the books he presents the key issues facing system architects and offers 110 heuristics to guide architects in properly addressing these issues.

Though informal and far from comprehensive, these heuristics capture and provide a tremendous amount of useful advice for successful system architecting. Because of their informality they are difficult to encode in automated tools for use in evaluating architectural specifications, as Armani needs to do. Rechtin does, however, make a compelling case for the importance of heuristics as a mechanism for capturing architectural design expertise. As we will see in Chapter 4, the Armani design language builds on his ideas by providing a *heuristic* construct that architects can use to specify "soft" guidelines for designing software architectures.

Chapter 4
The Armani Design Language

Armani's design language defines the system's core concepts for specifying the design of individual software architectures and for capturing architectural design expertise. In this chapter I describe the Armani design language. I first argue the need for such a design language and lay out the requirements that the language must satisfy. Then I describe the key design decisions made in creating the language, arguing that the language both meets the requirements presented and provides the desired benefits. Rather than providing a detailed specification for the language (which can be found in [Mon98]), the discussion in this chapter emphasizes the interesting technical aspects of the language and the fundamental design decisions I made in creating it.

4.1 Armani design language requirements

The Armani design language defines the primitive building blocks for describing software architectures and architectural design expertise, along with rules for composing those primitives. Although selecting such a set of primitive constructs and composition rules is a critical first step in the development of an architecture design or analysis tool (or environment), it is a time consuming task that is difficult to do well. Armani relieves software tool and environment developers from this burden by providing a powerful and generic architecture design language that encapsulates such a set of design primitives and composition rules.

The Armani design language must meet five fundamental requirements. I established these requirements based on a series of conversations with practicing software architects, observations of numerous formal and informal architectural software systems specifications, and experience using a wide variety of architecture description languages. The first two requirements identify what the language must be able to describe – architectural instances and architectural design expertise. The subsequent three requirements define necessary characteristics of the design language for describing expertise and instances – expressiveness, analyzability, and incrementality.

Strictly speaking, only the second of these requirements – capturing design expertise – needs to be satisfied to demonstrate that the thesis holds. Experience with architecture description languages and design environments (both my own and that gathered from other sources), however, indicates that the other four requirements are necessary for the language to capture a sufficiently broad array of powerful design expertise and be useful to an appropriately wide audience, discharging the broad Armani requirements of incrementality, breadth, and power. Using the same language to capture instances of system designs and abstract design expertise also simplifies tooling and reduces the learning curve for architects who need to learn only one language for both purposes.

A detailed discussion of these requirements, the rationale behind them, and their potential benefits follows.

Requirement 1: Instances. *The design language must be capable of describing the architectural structure, properties, and topology of individual software system designs (architectural instances).*

An explicit and precise architectural specification of a software system provides multiple benefits throughout the system's lifecycle. Such a specification documents the design of the system, facilitating communication between the system's designers, developers, testers, installers, and maintainers. A precise architectural specification is also a prerequisite for effective automated architectural design evaluation and analysis. In addition to supporting automated analyses, a precise architectural specification is also helpful for designers who need to perform informal, back-of-the-napkin analyses. Finally, the process of explicitly defining a software system's architecture frequently exposes potential mismatches and discrepancies in the design.

In order to achieve these benefits, software architects must be able to explicitly, precisely, and unambiguously specify their design. There are three key parts to a system's architecture specification that the Armani design language needs to be able to specify. First, the *structure* and *topology* of a software system forms its core architecture. This includes defining the components that make up the system and the connectors through which the components interact. The language must, therefore, provide constructs for describing system structure and topology.

Second, the design language needs to be able to describe architectural *properties*. Properties fall into two classes - asserted properties and emergent properties. Asserted properties represent claims that the designer makes about individual design elements or the system as a whole. The value of an asserted property may represent measurements made in testing an implemented design element, estimates made by the architect, or a priori requirements that the system architect states the implementation must satisfy. Emergent properties, as their name suggests, are properties of a system that are derived from the interactions of the system's individual pieces.

Third, the design language needs to be able to describe *constraints* that must be maintained as a system's design evolves. Although maintaining a software system's conceptual integrity is critical to its overall success [Bro95], it is difficult to do so in the initial design of the system and even more difficult as the system design is updated and evolved. By allowing a system's original designers to explicitly state the fundamental constraints under which the system was developed, and which need to be maintained as the system evolves, system updates can be made in a principled and informed way. The Armani design language must, therefore, include the ability to associate explicit design constraints with software systems.

In addition to the three fundamental constructs just described, Armani's design language also needs to support encapsulation of design details and hierarchical decomposition. Architecture descriptions for large systems that don't make use of encapsulation and hierarchical decomposition tend to either become unwieldy because they contain too much detail at a single level of abstraction or analytically useless because they are too vague. Therefore, to keep the relationship between the elements of these systems both intellectually tractable and analytically useful, the architecture description language must support the encapsulation of subsystems. Appropriate encapsulation allows subsystems to be described in full detail at one level of abstraction and encapsulated for use as a single design element at more abstract levels of description.

Requirement 2: Design Expertise. *The Armani design language needs to be capable of describing software architecture design expertise that can be reused across multiple system designs and multiple design environments.*

As organizations build software systems they learn how to do so effectively. This learning process includes developing a deep understanding of the types of systems that the organization builds and discovering the pitfalls they face in building them. As part of this learning process, organizations generally develop a set of good practices and techniques for software design, as well as ways to avoid design problems. Such an organization has acquired *architectural design expertise*.[6]

These practices and techniques are generally developed informally and frequently reside only in the minds of a few software experts working on the project. Storing this expertise only in the heads of a few critical people has a number of negative implications for the software development organization. First, when these experts leave the organization the expertise that the organization has developed frequently leaves with them. Even if the experts stay in the organization, design details are easily forgotten over the duration of a system's operational lifetime. Finally, a lack of documentation describing expected design practice makes training people new to a development project more costly and difficult than it needs to be. Simply documenting the design expertise acquired by the organization should reduce the costs and disruption of staff turnover as well as speeding up the training of the organization's new designers and developers.

Although simply expressing an organization's architectural design expertise with natural language documents is a useful first step, there are three significant benefits to using a more formal language and framework for codifying the expertise. First, a domain expert using a formal architecture description language can build on the basic concepts and framework provided by the formalism. Using such a language supports creativity and productivity by freeing the designer to focus on the expertise he wants to express rather than searching for a way in which to express it. Second, the designer can be much more precise in the specification of the expertise with a formal language than with a natural language. Third, explicitly capturing architectural design expertise and architectural designs with a well-defined language is a necessary first step towards providing automated design guidance, evaluation, and analysis.

To achieve these benefits, Armani must provide a formal language for capturing architectural design expertise. Two fundamental kinds of design expertise are *design vocabulary* and *design rules*. To capture design vocabulary the language must provide constructs for defining the

[6] [ATT93] describes one way in which AT&T has informally captured and described some of this expertise.

structure, properties, and constraints associated with user-defined types of structural design elements. The types of vocabulary elements thus defined become the semantically meaningful building blocks used to specify instances of architectural designs. The language constructs for capturing design rules must at least be capable of defining constraints on how related types of design vocabulary elements may be composed and valid ranges for, and relationships between, the properties of those elements.

In addition to providing constructs for capturing individual "nuggets" of design expertise (with the vocabulary and design rule constructs), the Armani design language also needs to provide a way to aggregate related design expertise into coherent packages (or *styles*). This process primarily entails packaging vocabulary elements that are designed to work together with design rules that guide and constrain how the elements may be composed and interact. To make such a packaging scheme coherent the language must define the semantics of composing and aggregating the individual design expertise constructs.

Requirement 3: Expressiveness. *The notation and constructs supplied by the Armani design language must match the expressive needs of software architects.*

The Armani design language should facilitate the specification of software architectures and the capture of architectural design expertise. To do so, the Armani design language's constructs must map naturally to the concepts that architects use for designing software. This includes both the selection of primitive building blocks and the rules that define how those building blocks can be assembled and composed. If the gap between the language's constructs and designers' intuition is too great then the language will hinder good design rather than encouraging it. Conversely, if the language maps well to intuitive design concepts then it can provide designers with significant analytical and modeling leverage.

Another aspect of expressiveness is the richness and breadth of the designs and design expertise that the language can capture. The language must allow architects to capture non-trivial design expertise and specify precise and detailed descriptions of their architectural designs. Likewise, the language must be able to encode design expertise and system specifications for a broad range of architectural styles.

Requirement 4: Analyzability. *Armani's design language must support the evaluation and analysis of architectural descriptions.*

Much of the leverage that a custom software architecture design environment provides architects is derived from the environment's ability to evaluate and analyze architectural designs. Effective design analyses allow system architects to explore the expected properties of a proposed system, warn them of potential design problems, and help them determine whether their design is likely to produce a system that will meet its requirements. This capability allows architects to evaluate design options early in the system development lifecycle when multiple design alternatives can be explored relatively inexpensively.

The first two requirements stated only that the language be able to capture declarations of instances and design expertise. The Armani language itself does not need to be able to support the expression of operations or computations. The language need only be able to specify declarations about the structure, properties, and constraints of system instances, and abstract, modular, reusable design expertise. Although the language itself does not need to support the expression of operations, the systems and expertise specified with the language must be readily amenable to analysis by external tools.

Providing a sufficiently formal design language and a conceptual framework for capturing both the architectural specifications of individual software systems and abstract architectural design expertise is an important first step towards satisfying this requirement. Such a language and framework allows the precise specification of architectural designs that both automated tools and humans can evaluate. Further, design expertise captured with this language can be used by automated tools to analyze and evaluate specific system designs.

Although strong support for automated analysis appears to be a desirable language property, there is a fundamental tension between a language's analyzability and its naturalness of expression. In general, informal languages tend to be more intuitive and expressive than formal languages. Formal languages, however, are generally much more amenable to rigorous analysis. The design of the Armani language must find an appropriate balance between the need for analyzability and the need for language expressiveness.

Requirement 5: Incrementality. *The language must support incremental capture of architectural design expertise and incremental modifications to architectural specifications.*

In order to support the rapid customization of the Armani design environment, the Armani design language must be capable of encapsulating software architecture design expertise in modular units that can be incrementally composed and added to the design environment. As Chapter 5 discusses, building support for incremental specification into the design language that underlies the environment makes the job of customizing the environment quite straightforward.

The importance of having a modular and incremental language for capturing architectural design expertise extends, however, beyond its role in customizing the Armani design environment. The ability to collect design expertise in small, modular, and composable packets simplifies the conceptual effort required to capture and express the design expertise itself. Furthermore, it allows designers to select the expertise that they need for a specific project and to quickly adapt the basic concepts that they have to work with.

In addition to the need for incrementality in capturing software architecture design expertise, the language also needs to support incremental modification to individual architectural specifications. As the architecture of a software system evolves over time it is important that its design specification be modified to appropriately represent the changes to the system implementation. Building this capability into the architecture design language makes such specification changes easier, more straightforward, and therefore more likely to actually be performed under time and deadline pressure.

4.2 The Armani design language

The Armani design language meets these requirements. In the remainder of this chapter I present the key design decisions made in creating the Armani language, describe the core constructs of the language in the context of discussing these design decisions, and argue that the design decisions lead to a language that satisfies the stated requirements. Throughout the discussion I will illustrate the use of the language for two of its three roles – specifying the architectural design of individual software systems and capturing architecture design expertise. I defer a detailed discussion of the language's third role – creating custom software architecture design environments – to the following chapter.

4.2.1 Architectural structure

Selecting an appropriate set of core constructs for the Armani design language exposes some fundamental tensions. For example, Armani needs to offer sufficient analytical and expressive power for specific styles of design, yet still be flexible and extensible enough to support design work done in a broad range of architectural styles. Armani addresses this tension by providing a simple, generic language for describing *architectural structure* as its foundation for architectural specification. In the context of Armani, architectural structure defines the decomposition of a software system into its constituent components and the ways in which those components interact. This structure forms the context for doing all subsequent system specification and design.

The selection of generic structure as the foundation for architectural description satisfies the language's need to work over a broad range of architectural styles. To address the language's need to handle specific design issues in these various styles, Armani allows this generic structure to be augmented with style-specific design details using Armani's property and design rule constructs. As subsequent sections of this chapter will describe, the language also provides ways for architects to package and reuse the style-specific augmentations. This approach allows the Armani design language to finesse the fundamental tension between generality and style-specificity, getting the benefits of both while minimizing their drawbacks.

As outlined in Table 4.1, the basic structural constructs of the Armani design language are *components*, *connectors*, *ports*, *roles*, and *systems*. These constructs are generically referred to as *design elements*. *Attachments* define a point of interaction between a port and a role. As a result, the set of attachments associated with a system define that system's topology. All design elements can be hierarchically decomposed with the *representation* construct. An *abstraction map* defines the mapping from an "outer" element to the elements defined within its *representation*. A *binding* is a special kind of abstraction map in Armani that defines equivalence between two entities. All design elements can also be annotated with *properties* that capture selected non-structural aspects of the design elements. These seven constructs form the basic ontology for capturing the architectural structure of software systems. They also provide the context within which architectural design expertise is captured.

Selecting this collection of core structural constructs proved to be a critical and challenging step in defining the Armani design language. These constructs were selected based on a combination of: observations of practicing software architects who make heavy use of box and line diagrams (represented by components and connectors, respectively), studying the informal and intuitive techniques architects commonly use to represent their ideas (generally box and line diagrams coupled with informal natural language text), and experience using previous architecture description languages and module interconnection languages.

One of the first steps in designing a software architecture is dividing the functionality, responsibilities, and capabilities of the target system into a set of *components*. The components represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of commonly used components include clients, servers, filters, objects, blackboards, and databases.

A component's interfaces are defined by a set of *ports*. Each port identifies a point of interaction between the component and its environment. A port can represent an interface as simple as a single procedure signature or something more complex such as an event multi-cast interface point or a collection of procedure calls that must be invoked in a specified order. One of the interesting capabilities of Armani's component and port constructs is that a component can provide multiple interfaces to its environment. This approach separates the specification of what the component does from how it is packaged.

The decomposition of a system architecture into its constituent components is only the first step in the design and specification of a system architecture. A fundamental second step is defining the ways in which those components interact. Armani's *Connector* construct addresses the need to represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. Informally, they provide the "glue" for architectural designs and correspond to the lines in box-and-line descriptions. Examples of simple connectors include pipes, procedure calls, and message-passing channels. Connectors are not, however, limited to these simple binary interactions. They can also represent more complex interactions, such as a client-server protocol, an event system, or a SQL [DD97] link between a database and an application.

Like components, connectors have explicitly specifiable interfaces that are defined by a set of *roles*. Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as the *caller* and *callee* roles of an RPC connector, the *reader* and *writer* roles of a pipe, or the *sender* and *receiver* roles of a message passing connector. Other kinds of connectors may have more than two roles. For example, an event broadcast connector might have a single *event-announcer* role and an arbitrary number of *event-receiver* roles. Likewise, the message passing connector described above could be augmented with a third *probe* role that allowed other components to listen to all message traffic passing on the connector without being able to write messages.

Individual component and connector specifications are not particularly useful in isolation. They become useful and interesting, however, when they are composed to form *systems*. Armani therefore provides a *system* construct to represent configurations of components and connectors. A system includes (among other things) a set of components, a set of connectors, and a set of *attachments* that describe the topology of the system. An attachment associates a port interface on a component with a role interface on a connector.

Construct	Purpose
System	Describes a set of components and connectors and their interaction topology.
Component	Represents a system's primary computational elements and data stores.
Connector	Mediates the interaction between components.
Port	Defines an interface to a component.
Role	Defines an interface to a connector.
Property	Captures non-structural properties of design entities.
Representation	Supports encapsulation and hierarchical decomposition of designs.
Attachment	Defines a relationship between a port and a role. The set of attachments in a system defines that system's topology.
Abstraction Map	Defines the relationship between a design element and the elements contained in a *representation* of that element.

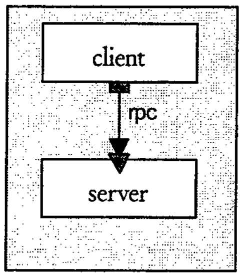
Table 4.1: Summary of Armani architectural instance primitives

To illustrate Armani's structure language, Example 4.2 describes a trivial architectural specification of a system with two components – a *client* and a *server* – connected by an *rpc* connector. The client component is declared to have a single *send-request* port, and the server has a single *receive-request* port. The connector has two roles designated *caller* and *callee*. The topology of this system is defined by the set of *Attachments*.

4.2.2 Representations

Complex architectural designs require hierarchical descriptions to make their specifications tractable for individual designers to understand, yet sufficiently detailed to guide system analysis, implementation, and testing. Armani supports the hierarchical decomposition of architectures. Specifically, any design element can be represented by one or more lower-level descriptions that provide additional detail. Each such description is termed a *representation*. A representation establishes and enforces an abstraction boundary between the structure, interfaces, and properties that a design element exposes to its environment and the details of the element's design and/or implementation. A representation consists of two parts: a *system* that describes the representation's structure, and a set of *bindings* that define a mapping between structure in the higher-level element encapsulating the representation and the lower-level elements defined in the representation's system.

```
System simpleCS = {
    Component client = { Port sendRequest }
    Component server = { Port receiveRequest }
    Connector rpc = { Roles {caller, callee} }
    Attachments {
                    client.sendRequest to rpc.caller ;
                    server.receiveRequest to rpc.callee }
}
```



Example 4.2: Simple client-server system specification in Armani text (left) and as a box-and-line diagram (right)

To illustrate, let us extend the simple client-server system described in Example 4.2. Suppose that the *server* component in this example encapsulates a complex sub-design. Specifically, the server component consists of three interacting components – a connection manager, a security manager, and a database. Figure 4.3 depicts this design extension graphically, and Example 4.4 provides the corresponding Armani textual specification.

Adding a representation to the server component allows the architect to provide details about the lower-level design of the server without compromising its higher-level abstraction as a single design element. When reasoning at an abstract level about the design it is appropriate to hide the server's complexity and simply think of it as a component that provides a specific service. If, however, the server represents a large subsystem it may be necessary to selectively reveal the detailed design of the server. The agent interested in seeing such design details could be a human architect or it could be an automated design analysis tool.

In addition to their role in encapsulating design complexity, representations can also be used to encode multiple views of architectural entities. By allowing individual design elements to have multiple representations Armani allows architects to specify different but complementary views of a single design element's sub-architecture, detailed design, or implementation.

The ability to encode multiple views raises some tricky challenges. One of the primary challenges is defining the relationships and rules for maintaining consistency between the different views. To keep the design language simple and flexible for a broad array of applications, Armani does not bind the details of the semantics of using representations to depict multiple views. Likewise, it does not provide specialized constructs for managing inter-view correspondences. Rather, it provides a basic infrastructure of structure and properties with which architects and environment designers can define their own view semantics. Developing a more powerful view mechanism lies outside of the scope of this dissertation but it is fertile ground for future work.

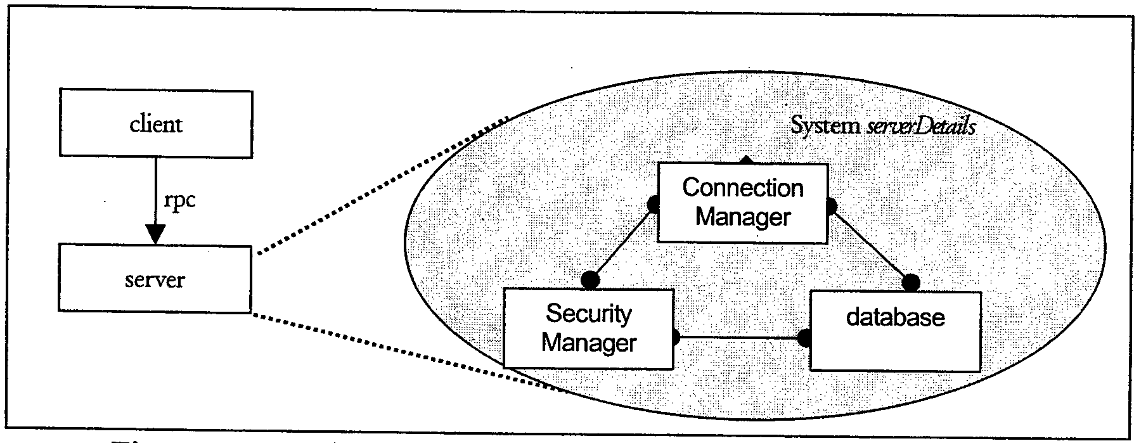


Figure 4.3: Graphical depiction of simple client-server system with server representation

4.2.3 Extending structural specifications with properties

Although structure and topology are critical aspects of a system's architecture, they are not, by themselves, sufficient for a complete architectural description. There are many other aspects of system design that can and should be captured in an architectural specification. The aspects that must be captured vary depending on the style of the system being described and the types of analyses that the system description needs to support. A system with hard real-time requirements, for example, needs a way to specify deadlines, scheduling doctrine, latency, and the performance profile of individual components and connectors. A client-server management information system, on the other hand, is probably more concerned with communication protocols, how the different stages of information processing are allocated between components, and supported database query languages than it is with the design concerns of hard real-time systems.

As these two examples illustrate, architects using a general-purpose architecture design language will likely want to specify a wide variety of non-structural design information. Attempting to include native language constructs to support all of the different types of design information that might be useful for all types of design is unlikely to be successful, as such a language would be overly complex and inflexible.

The Armani design language takes a different approach. It provides a flexible and generic *property* construct that can handle the expression of a wide variety of non-structural information. An Armani *property* is a typed attribute-value pair that can be associated with essentially all of the Armani design language constructs. Properties can even be associated with other properties, in which case they are called *meta-properties*. Meta-properties define properties of a property, rather than properties of a design element. Although the basic property structure is simple, complex property expressions can be created by composing individual properties and defining standard properties that all related design elements must posses.

```
System simpleCS = {
    Component client = { ... }
    Component server = {
            Port receiveRequest;
            Representation serverDetails = {
                System serverDetailsSys = {

                    Component connectionManager = {
                        Ports { externalSocket; securityCheckIntf; dbQueryIntf } }

                    Component securityManager = {
                        Ports { securityAuthorization;  credentialQuery; } }

                    Component database = {
                        Ports { securityManagementIntf; queryIntf; } }

                    Connector SQLQuery = { Roles { caller; callee } }
                    Connector clearanceRequest = { Roles { requestor; grantor } }
                    Connector securityQuery = {
                        Roles { securityManager; requestor } }

                    Attachments {
                        connectionManager.securityCheckIntf
                                    to clearanceRequest.requestor;
                        securityManager.securityAuthorization
                                    to clearanceRequest.grantor;
                        connectionManager.dbQueryIntf to SQLQuery.caller;
                        database.queryIntf to SQLQuery.callee;
                        securityManager.credentialQuery
                                    to securityQuery.securityManager;
                        database.securityManagementIntf
                                    to securityQuery.requestor;  }
                }
                Bindings { connectionManager.externalSocket to server.receiveRequest }
            }
    }
    Connector rpc  = { ... }
    Attachments { client.send-request to rpc.caller ;
                server.receive-request to rpc.callee }
};
```

Example 4.4: Extending the client-server example with a detailed representation for the server component.

```
System simpleCS = {
        Component client = {
            Port sendRequest;
            Properties {    requestRate : float = 17.0;
                            sourceCode : externalFile = "CODE-LIB/client.c" }}

        Component server = {
            Port receiveRequest;
            Properties { idempotent : boolean = true;
                            maxConcurrentClients : integer = 1;
                            multithreaded : boolean = false;
                            sourceCode : externalFile =  "CODE-LIB/server.c" }}

        Connector rpc  = {
            Role caller;
            Role callee;
            Properties { synchronous : boolean = true;
                            maxRoles : integer = 2;
                            protocol : WrightSpec = "..." }}

        Attachments {       client.sendRequest to rpc.caller ;
                            server.receiveRequest to rpc.callee }
}
```

Example 4.5: Simple-client-server system with properties

The decision to provide Armani with a simple set of property primitives from which complex properties can be built rather than a fixed set of predefined properties is fundamental to Armani's goal of achieving flexibility and supporting incremental adaptation. This approach allows architects to precisely define the important properties of their designs in terms that are appropriate for each individual system, style, or design environment.

The price of this flexibility and freedom is that architects need to put a non-trivial amount of effort into selecting and defining the set of properties to be used in their architectural specifications; more effort, in general, than they would be required to expend if all of the appropriate properties were simply built into the language. To address this drawback Armani includes a number of mechanisms that encourage the reuse and adaptation of previously defined properties both within an individual system specification and between different system specifications. Section 4.4's discussion of architectural styles and the Armani type system describes in detail how such a collection of properties can be defined, used, and reused.

To illustrate the use of properties in describing software architectures, Example 4.5 shows an extension to the simple client-server system given in Example 4.2. This example has been annotated with properties that describe characteristics of the clients, servers, and rpc's that form the example system's structure. These properties are annotated with type declarations. For the purpose of this discussion, a property type simply defines the set of valid values that can be assigned to a property. I provide a more detailed discussion of property types in Section 4.4.5.

Table 4.6 provides a set of sample properties that are applicable to different types of design elements and styles of design. This list illustrates how a variety of non-structural architectural data can be encoded with the property construct. The specific properties listed as examples are not defined by or built into the Armani design language. It is the responsibility of the system, style, or environment designer to define the meaning and semantics of the set of properties that are used and analyzed by tools in individual architectural specifications, styles, or environments.

4.2.4 Armani structural language syntax

The discussion to this point has presented a brief description of the Armani design language's constructs for specifying architectural structure and the properties of that structure. For reference, a short BNF describing these constructs is provided in Figure 4.7. The simplicity of the core structural language is reflected in the language's simple BNF. This minimal language specification is missing three key parts of the full language – type expressions, styles, and design rules. This syntax description will be extended later when these language features are described in detail. For reference, the full Armani BNF is included in Appendix A.

4.3 Design constraints

The Armani design language described to this point is effective for specifying the architectural structure and properties of a software system at a single point in time. To fully satisfy the first language requirement (capturing system instances), though, the design language must also be capable of describing constraints on how a system's architecture may evolve over time. The Armani language therefore includes constructs for annotating architectural specifications with design constraints. Constraints specified for individual systems bound the ways in which that system's architecture, or the properties of that architecture, can change over time. In doing so, they help subsequent developers maintain the system's conceptual integrity[7] as they update and evolve its design.

There are many different kinds of constraints that an architect might like to impose on a system's architecture. Useful architectural constraints include: limitations on modifications to a communications topology, restrictions on legal topological patterns of the system's structure graph, maintaining performance, reliability, security, fault-tolerance, and other parameters within acceptable ranges, and ensuring interface compatibility. The Armani language supports the specification of these categories of constraints and many others. As this chapter will lay out, and the discussion of case studies in Chapter 7 will elaborate, the Armani constraint language can capture a broad array of design constraints.

[7] See [Bro95] for a detailed discussion on the importance of maintaining conceptual integrity in the design of a software system.

Property to capture	Example usage

Whether a connector works synchronously or asynchronously.	Connector conn = { ...
	 Property Synchronous : boolean = true;
	 ... };

The number of bytes that can be stored in a connector's buffer.	Connector conn = { ...
	 Property BufferSize : int = 1024;
	 ... };

The query languages and protocols that a database understands. This property specification could be associated with a port that specifies the interface to a database component, or with the database component itself.	Component database = { ...
	 Property QueryProtocols : set{Protocol}
	 = {SQL,ODBC, JDBC};
	 ... };

The rate at which data is processed by a design element. This type of information could be associated with a component, a connector, an interface (a port or role), or a complete system. This property is specified as a record type that includes both the throughput value and the unit in which it is specified. Alternatively, if a standard unit for measuring throughput is established and used consistently throughout a specification then this property could be represented with a simple floating point value.	Component dspFilter = { ... *Property Throughput :* *Record [value:float; unit:flowUnitType]* *= [value=1500; unit=kb/second];* ... };

The protocol that a connector uses to mediate the interaction between a set of components. Because Armani doesn't include a native protocol specification language, this protocol can be specified as a string expression in the WRIGHT Architecture Description Language [Allen97] and stored as a property of an Armani connector. The protocol specification can then be passed for analysis to automated tools that understand WRIGHT. A similar approach can be used to embed specifications given in arbitrary other languages.	Connector unixPipe = { ...
	 Roles { source; sink }
	 Property protocol : WrightSpec =
	 "... source.write→buffer.write ...
	 [] buffer.write→sink.write ... ";
	 ... };

Table 4.6 Sample property specifications

To support the specification and enforcement of design constraints, Armani needs to address four critical language design issues. First, it must establish the basic formalism used for expressing constraints. The selected formalism must be reasonably familiar to its target audience, mathematically well-founded, and sufficiently expressive for capturing common architectural design constraints. Second, the process of evaluating whether a design satisfies its constraints must be readily automatable and computationally decidable. Third, it must be straightforward to integrate the constraint formalism with the structure and property constructs described in the previous section. This includes defining scoping rules, primitive building blocks, operators, and axioms for the constraint language. Fourth, the language must provide a way for designers to separate the truly critical design constraints from the constraints that are simply guidelines. For greatest effect, this distinction should be built directly into the design language's semantics.

To address these design issues, Armani provides a limited first order predicate logic-based constraint language in which design constraints are realized as predicates over architectural specifications. This predicate language is augmented with constructs for classifying constraints as either invariants or heuristics to indicate how rigorously they must be enforced. The following sections elaborate on the approach taken in creating the constraint language and the design decisions underlying it.

4.3.1 Constraint language formalism

The formalism selected for specifying design constraints in Armani must meet a number of somewhat conflicting requirements. Specifically, the formalism must be reasonably familiar to its target audience, mathematically well-founded, and sufficiently expressive for capturing common architectural design constraints. Additionally, the process of evaluating whether a design satisfies its constraints must be readily automatable and computationally decidable to support the Armani design environment's automated constraint management capabilities.

To meet these requirements, Armani uses a first order predicate logic (FOPL) formalism as the basis for its constraint specification sublanguage. FOPL is widely known and understood by the software architects and computer scientists who are likely to use the Armani language. As a result, the amount of effort required to learn the Armani design constraint language should be commensurate with the language's expected benefits. Likewise, FOPL is a well-understood mathematical formalism that can be readily reused in the context of specifying architectural design constraints. Reusing a well-understood formalism leverages a significant body of theoretical work and provides a rigorous mathematical basis for Armani constraint specifications.

FOPL also nicely matches Armani's expressive needs. Architectural descriptions in Armani are primarily descriptions of static structure and the properties of that structure. For example, constraints that specify required structure, recommended design patterns and topologies, and legal ranges for property values are easily expressed as first-order predicates.

Architectural Elements

```
System            ::=  System Name = { EntityDecl* } ;

EntityDecl        ::=  ComponentDecl
                       | Connector-Decl
                       | Port-Decl
                       | Role-Decl
                       | Property-Decl
                       | Rep-Decl
                       | Attachments-Decl

GenericDecl       ::=  PropertyDecl | RepDecl;

ComponentDecl     ::=  Component Name = {
                               (PortDecl | GenericDecl)* } ;

ConnectorDecl     ::=  Connector Name = {
                               (RoleDecl | GenericDecl)* } ;

PortDecl          ::=  Port Name = { GenericDecl * } ;

RoleDecl          ::=  Role Name = { GenericDecl * } ;

AttachmentDecl    ::=  Attachments { (PortName To RoleName)* } ;

Name              ::=  [a-zA-Z][a-zA-Z0-9_\-+]*
```

Properties

```
PropertyDecl      ::=  Property Name [ : TypeExpression ] = Value
                               [ << PropertyDecl+ >> ] ";"
                       | Properties {(Name [ : TypeExpression ] = Value
                               [ << PropertyDecl+ >> ] ";") * };
```

Representations

```
RepDecl           ::=  Representation [ Name ]= "{"
                               System
                               [ Bindings = { (Name To Name)* } ]
                               "}" ";" ;
```

Figure 4.7: Partial BNF for simple structural instance language

In many respects, therefore, FOPL is a natural choice for the formal foundation of Armani's constraint specification language. The formalism runs into trouble, however, with the requirement that it be readily automatable and computationally decidable. Although evaluating FOPL expressions is a fairly well understood process that has been widely researched and used in many previous applications (such as [Per89], [Min91], and [OS97]), determining whether an arbitrary set of predicates is true or false is not computationally decidable [End72]. The key aspect of FOPL that causes undecidability is its ability to quantify variables over infinite sets.

The Armani predicate language addresses this issue by insuring that predicate quantification is done only over finite sets. As a result, the algorithm for checking whether a design satisfies its constraints is straightforward and decidable. At a logical level, the constraint checker handles quantification by enumerating all of the values of the set quantified over and checks that the appropriate predicate holds for all of those values (in the case of a universal quantification), or for at least one of those values (in the case of an existential quantification). A detailed description of the design constraint checking algorithm is provided in [Mon98].

Although at first glance limiting quantification to finite sets might seem like a severe restriction in expressiveness, it has not proven to be a significant limitation in practice. Specifically, it did not present any problems in the case studies described in Chapters 7 and 8, nor has it been an issue in other explorations of the language's capabilities. In general, most of the interesting constraints that could be cast as quantifications over infinite sets can be readily recast as quantifications over finite sets or predicate expressions involving no quantification at all.

In addition to decidability, the speed with which arbitrary constraint expressions can be evaluated is also a concern. As I discuss in detail in section 9.1.4, the nature of the predicate evaluation that Armani supports allowed the use of straightforward constraint checking algorithms that generally ran very fast on typical architectural specifications. These algorithms ran fast enough to easily support interactive design checking.

The complexity of the analyses that need to be supported by the formalism are rather minimal in practice. The constraint checking mechanism only needs to be able to verify that a specific system instance satisfies its constraints. As I discuss later in this chapter and in [Mon98], this reasoning is quite straightforward. The Armani constraint checking formalism does not provide the ability to automatically detect whether a given system's constraints can *ever* be satisfied (that is, could something be changed in the design to make it correct). Likewise, it does not attempt to determine if a set of constraints can *never* be satisfied by any system (that is, do the system's constraints imply a contradiction). These types of analytical capability generally require full theorem proving capabilities, which is beyond the scope of Armani's capabilities.

Carefully scoping the types of constraints and analyses that the language's constraint checking system can evaluate insured that the evaluation performance of Armani's constraint checker did not become a significant issue in practice.

4.3.2 Extending the structural language with constraints

Having made the decision to use first order predicate logic as the foundation for constraint specification, it is necessary to integrate this constraint formalism with Armani's structure and property constructs. To achieve this integration, design constraints are realized in Armani as predicates over architectural specifications. Predicates can refer to the structure, topology, and properties of software systems or the individual design elements that make up a system. Constraint checking tools can evaluate whether a design's structure and properties satisfy its declared constraints.

First order predicate logic provides a natural match to the types of constraint predicates that Armani needs to support. As a result, the challenge in building an effective language for describing architectural design constraints on top of first order predicate logic lies in (1) insuring its decidability, (2) adding an appropriate set of primitive architectural predicates, operators, and axioms, and (3) defining a set of scoping rules that provide sufficient modularity and expressive capability without becoming overly verbose.

As discussed in the previous section, the task of evaluating FOPL expressions is made decidable in Armani by limiting quantification to finite sets. Armani insures that this limitation holds by making it syntactically impossible to define or construct infinite sets for the purpose of quantification. In a quantified predicate expression, the set to be quantified over can be defined by (1) explicit enumeration of the set's elements, (2) referencing an element's substructure, such as all of the ports of a component, (3) referencing a set-typed property, which will by definition always be finite, or (4) performing a sequence of operations that take one or more finite sets as arguments and return a single finite set. Because none of these mechanisms can be used to create infinite sets, Armani does not allow quantification over infinite sets.

The following two sections describe the primitive architectural predicates, operators, axioms and scoping rules added to FOPL by Armani.

4.3.2.1 Extending the first order predicate logic with architectural primitives

Armani defines a syntax and semantics for expressing design constraints as FOPL predicates. This includes a standard set of FOPL operators for logical, comparison, arithmetic, and quantification expressions. The semantics of these operators, detailed in [Mon98], are straightforward. Adapting FOPL for use in describing architectural design constraints, however, requires more care than simply adding a set of FOPL constructs to the design language's BNF. Tailoring the predicate language to describe architectural constraints requires the addition of primitive functions with architecture-level semantics. Fortunately, such an extension can be carried out in a straightforward way by simply adding a set of built-in functions to the predicate language.

Specifically, Armani adds twenty-four primitive functions that handle four key categories of architectural constraints. These constraint categories cover: system topology, properties, aggregation of sets of related architectural entities, and types (which will be discussed in detail in Section 4.4).[8] A set of functions for (finite) set manipulation are also included. Table 4.8 provides examples from each of these basic categories.

The first category of primitive architectural functions deal with system topology. These functions allow an architect to specify constraints such as how and with what a system's components can communicate, which components must be connected, which must *not* be

[8] A full description of all of Armani's primitive predicates is provided in [Mon98].

connected, etc. Topological functions can also be used to define and enforce parent-child relationships between entities in a design.

The second category of primitive functions are referencing and comparison operations for properties and substructure of design elements. The operator "." is used to identify specific element properties or children of a design element. For example, the property that determines whether a connector *P* is synchronous can be referring to as *P.synchronous*.

The third set of functions support the aggregation of related substructure. These functions allow sets of substructure to be referenced by name. For example, *P.roles* names the set of roles associated with the connector *P*. These functions provide a convenient shorthand for describing sets for quantification.

The fourth and final category of primitive architectural functions deal with architectural types. Specifically, they can be used to determine or select design elements based on the elements' type declarations.

Experience using Armani indicates that this is an appropriate collection of primitive architectural functions and referential capabilities. These functions and predicates can be

Function category and signature	**Function description**
Topology: *Connected(comp1,comp2)*	Returns True if component *comp1* is connected to component *comp2* by at least one connector, else it returns False.
Topology: *Reachable(comp1, comp2)*	Returns True if component *comp2* is in the transitive closure of *Connected(comp1, *)*, else it returns False.
Properties: *HasProperty(x, propertyName)*	Returns True if element *x* has a property called *propertyName*, else it returns False.
Properties: *<ElementName>.<PropertyName>*	Returns the value of the property identified by *<PropertyName>* in the element *<ElementName>*.
Aggregation: *<SystemName>.Connectors*	Returns a set containing all of the connector elements in the system identified by *<SystemName>*.
Aggregation: *<ConnectorName>.Roles*	Returns a set containing all of the roles of the connector identified by *<ConnectorName>*.
Types: *DeclaresType(elt, typeName)*	Returns True if the element identified by elt declares the type identified by typeName, else it returns False. A discussion of element types is provided in section **Error! Reference source not found.**.

Table 4.8 Examples of primitive architectural functions in Armani

composed within the FOPL framework to define complex and interesting design constraints. As architects gain further experience working with Armani they frequently discover additional constraints that they would like to be able to express but that are not readily represented with this set of primitive functions. To address this issue, Armani provides a mechanism by which architects can capture useful non-primitive functions that can be reused in multiple constraint specifications. These user-defined functions, called *design analyses*, will be discussed in detail later in this chapter.

The following examples illustrate how the built-in functions can be used to capture architectural design constraints.

The first predicate,

Connected(client, server);

checks that the components *client* and *server* are connected. The next predicate is defined in the scope of a system instance.

Forall conn : connector in systemInstance.Connectors | size(conn.roles) = 2;

It guarantees that all of the connectors in the system must be binary connectors (i.e., they must have exactly two roles). The following predicate specifies that all roles on all connectors in a system must be attached to a port, and further that the attached (port, role) pair must share the same protocol.

```
Forall conn : connector in systemInstance.Connectors | Forall r : role in conn.Roles |
    Exists comp : component in systemInstance.Components | Exists p : port in comp.Ports |
        attached(p,r) and (p.protocol == r.protocol);
```

The port and role protocol values are represented as properties of the port and role design elements.

In addition to describing the topological features of a system, predicates can describe and bound legal property values and types. The following examples demonstrate predicates that bound the legal value range for a property[9]

self.throughputRate >= 3095;

and that specify a relationship between multiple properties

comp.totalLatency == (comp.readLatency + comp.processingLatency + comp.writeLatency);

These examples are not intended to be an exhaustive exhibition of the types of predicates that can be expressed in Armani. Rather, they are intended as examples that give a flavor for the types of predicates that can be expressed over design instances with the Armani predicate language.

[9] The reserved keyword *self* refers to the design element instance (component, connector, port, role, or system) in which the constraint is scoped.

4.3.2.2 Scoping rules

Extending first order predicate logic with a set of architectural primitives is only a first step towards integrating the constraint language with the rest of the Armani design language. Another key issue is establishing the scoping and visibility rules for the constraints. Developing an appropriate set of scoping rules is challenging because they must satisfy the language's conflicting needs for modularity, expressive capability, and succinctness.

The importance of succinctness and expressiveness in language design are obvious. Succinct expressions are generally easier to write, read, and check for errors than verbose expressions. Likewise, if the language makes it difficult to express the desired constraints then it is likely to be an impediment to good design rather than an aid. A clean integration of the constraint language with the rest of the Armani language should, therefore, make it straightforward for an architect to express his or her desired design constraints and to do so in an elegant, concise way.

The rationale for supporting modularity in constraint expressions is a bit more subtle. The modularity of constraint expressions, however, plays a critical role in Armani's incremental modification capabilities. Keeping constraint specifications self-contained makes it much easier to move constraints around in a design, to shrink or expand the scope of applicability of the constraint, and to compose arbitrary constraints within a design. From an environment-modification perspective the addition to or removal of constraints from a design is greatly eased if those constraints are packaged as modular, independent entities. As section 4.4 will illustrate, this issue becomes increasingly important when constraint predicates are used in type and design rule specifications rather than simple instance specifications.

Armani achieves these goals by providing a few simple scoping rules that are broadly and uniformly applicable. An Armani design can be represented as a tree. Each node of the tree has precisely one parent and zero or more children. For example, a component *C* has a parent that is a system (call it *S*) and a set of children that include all of *C*'s ports, properties, and representations. Each of *C*'s children also has zero or more children.[10] It is important to note that the arcs of this tree represent parent-child relationships between elements rather than system connectivity. This tree structure is orthogonal to the connectivity graph Armani uses to represent system topology. This underlying tree representation can be used to describe arbitrary system connectivity graphs.

The first of these rules states that constraints can be specified in the scope of any design element or design element type declaration (remember that a design element is a system, component, connector, port, or role), or in the scope of the global design (which is defined as all things outside of the scope of any other declaration). As a result, all constraints are defined in an unambiguous scope.

The second rule defines name visibility and resolution. A constraint predicate can only refer to entities that are descendants of the constraint predicate's scope. In the example of Component *C* just given, a constraint defined in the scope of *C* could refer to any of the

[10] A more formal semantic specification of Armani's structure is provided in [Mon98]

ports, properties, or representations of *C* or any of *C*'s descendents in the structure graph. This constraint could not, however, refer to any other entities defined within System *S* or any of *C*'s siblings.

The scoping rules support succinctness and expressivity by allowing designers a great deal of flexibility in selecting an appropriate scope in which to declare each constraint. In general, as constraints are declared at higher nodes in the tree they can be defined for a more broad and/or specific selection of design elements. Conversely, as a constraint is moved down the tree its scope is refined and it is able to refer to and compare fewer entities, but the expression of that constraint can be made more concise, focused, and context-specific. In general, the rule of thumb that has arisen from using Armani to define constraints on design instances is that constraints should be pushed as far up the structure tree as needed to bring all of the necessary entities referred to by the constraint into scope, and no higher. Following this rule of thumb leads to modular, context-free, and reusable design constraint specifications that are relatively robust to system reconfiguration.

4.3.2.3 Wrap up

Armani's predicate language allows designers to annotate architectural specifications with design constraints. This predicate language also forms the foundation for Armani's type system and design rule specification capability. Section 4.4 will discuss this role of the predicate language in greater detail along with interesting design issues that it raises in the context of capturing architectural design expertise.

4.3.3 Invariants vs. heuristics

The constraint language just described provides a way for architects to precisely specify design constraints as predicates over a design or part of a design. All of the constraints that an architect might want to express about a design are not, however, equally important. Some constraints should never be violated. They specify a system's key design principles and assumptions; violating them may render the system unusable. Other constraints, however, can be viewed more as suggestions about how or whether aspects of the system can be changed. Violating these constraints will not necessarily prevent the basic operation of the system, though it may have other negative consequences.

It is, therefore, important that the Armani language allow architects to specify not only what it is that a design constraint is constraining, but also the rigor with which that constraint must be enforced. To address this need, Armani provides orthogonal constructs for specifying (1) the constraint itself, and (2) the constraint's enforcement semantics. The constraint itself is expressed as a predicate expression. The constraint's enforcement semantics are then defined by declaring whether the constraint is a *design invariant* or a *design heuristic*. A *design rule*, as defined by Armani, consists of a constraint expression and a declaration that the constraint is an invariant or a heuristic.

Design invariants, as their name suggests, specify constructs that must be maintained at all times. These often represent the basic assumptions that a system's constituent elements make about their environment and how they are able to interact with other design elements in the system. Design heuristics, on the other hand, specify rules of thumb used in designing the system. They can also be used to guide the modification of a system after it's initial development by clarifying and flagging design decisions.

The enforcement semantics of a constraint are not simply tooling issues. Rather, there is a fundamental semantic distinction between a design invariant and a design heuristic. The details of this distinction are captured by the design language's type system, which will be described more fully in section 4.4. Informally, though, invariant violations are type errors. An architectural specification that has one or more design invariant expressions that evaluate to false is not type-correct. The design contains a fundamental error. A specification with one or more heuristics that evaluate to false, on the other hand, may still be type-correct but it will generate a warning from Armani's type and constraint checker.

Example 4.9 illustrates the use of invariants and heuristics for representing design constraints. In this example, *MessagePath* is a connector that queues the messages it reads from its *source* role and writes them to its *sink* role in the order they were received at the source role. This simplified version of the connector has two properties – the size of the connector's queue buffer (in bytes) and its expected throughput (in messages per second). The connector specification also defines two constraints – one invariant and one heuristic – that define the range of acceptable values for these properties.

By specifying both an explicit present value and a legal range of potential values for the buffer size and expected throughput properties, the architect has described not only a snapshot of the initial system design (via the properties), but also the ways in which the connector can be modified and still fit within the overall system design (via the constraints). The invariant constraint provides strict limits on the acceptable sizes of the connector's *queueBuffer*. The heuristic that defines a relationship between the *expectedThroughput* and *queueBufferSize* properties, on the other hand, is provided as a guideline rather than a strict law.

```
System constraintExample = {
        ...
        Connector MessagePath = {
                Roles { source; sink; }
                Property queueBufferSize : int = 1024;
                Property expectedThroughput : float = 512;
                Invariant (queueBufferSize >= 512) and (queueBufferSize <= 4096);
                Heuristic expectedThroughput <= (queueBufferSize / 2);
        };
        ...
};
```

Example 4.9: MessagePath connector with invariants and heuristics

Figure 4.10 extends a portion of the simple BNF given in Figure 4.4 with productions that describe how design rules are added to the design language.

4.3.4 Summary of architectural structure discussion

The Armani design language described thus far satisfies the first of the requirements outlined in Section 4.1. Specifically, the language can describe the architectural structure of a software system, the topology and properties of that structure, and the constraints that bound the ways in which the structure can evolve. Additionally, it provides constructs for hierarchically decomposing architectural designs.

4.4 Capturing design expertise with predicate-based types

The second language requirement presented in Section 4.1 argues that, in addition to capturing individual system designs, Armani also needs to be able to capture, package, and reuse architectural design expertise *independent* of specific instances of system designs.

Conceptually, Armani divides abstract design expertise into two categories – *declarative* design expertise and *operational* design expertise. Informally, declarative design expertise describes the way that a design should (or must) be. That is, declarative design expertise is best categorized as those bits of design wisdom that can be articulated with a natural language statement such as "This design must ... " or "A *foo type* component is a component that ... ". Operational design expertise, on the other hand, consists of design evaluations and operations that are best expressed algorithmically. That is, they encapsulate algorithms for evaluating or modifying a design, rather than declarations about the design itself. A more succinct distinction between these two types of design expertise is that operational design expertise can describe actions for modifying a design but declarative design expertise does not, by definition, describe steps for modifying a design.[11]

The Armani design language directly supports the expression and checking of declarative expertise with predicate-based types. Specifically, Armani's type system captures declarative design expertise in the form of *design vocabulary*, *design rules*, and *architectural styles*. Because the Armani design language is declarative rather than algorithmic in nature, however, it is not well suited to capturing operational design expertise. To address this limitation, Armani provides an integration framework for linking external design tools into the Armani environment. This integration framework provides a complementary mechanism for capturing operational design expertise with independent tools. As I describe in Chapter 5, this approach has proven effective for capturing and encapsulating operational design expertise in the Armani environment.

In this section I present Armani's type system and describe how it can be used to capture declarative design expertise. I also discuss some of the key design decisions that make the approach work effectively and the implications of these decisions for capturing software architecture design expertise. I defer a detailed discussion of how operational design expertise is captured and used in the Armani environment until Chapter 5.

[11] For a detailed discussion on the issues related to separating and making use of both operational and declarative expertise in the related area of programming environments, see Kaiser's thesis [Kai85].

Architectural Elements

```
...

EntityDecl            ::= ComponentDecl
                          | Connector-Decl
                          | Port-Decl
                          | Role-Decl
                          | Property-Decl
                          | Rep-Decl
                          | Attachments-Decl

ComponentDecl         ::= Component Name = { (PortDecl | GenericDecl)* } ;

GenericDecl           ::= PropertyDecl | RepDecl | DesignRuleDecl;

...
```

Design Rule Productions

```
DesignRuleDecl        ::= ( Design )? ( Invariant | Heuristic )
                          DesignRuleExpression ";"

DesignRuleExpression  ::= <predicate expression, defined in Appendix A>
```

Figure 4.10: Partial BNF for simple structural instance language extended with design rule productions.

4.4.1 Capturing design expertise with architectural types

Armani uses a predicate-based type system to capture declarative architecture design expertise. As we will see throughout the rest of section 4.4, this approach provides a number of significant benefits for both the architects using Armani to capture abstract design expertise and environment developers who need take advantage of this expertise. These benefits include flexibility, analytic power, and composability.

Representing design vocabulary with predicate types allows architects the flexibility of associating complex structure and constraints with abstract vocabulary elements. Rather than providing only a mechanism for describing structure or properties as many other ADLs do, Armani's predicate-based type system allows architects to capture arbitrarily complex abstract design specifications in modular, reusable, units. As we will see, the nature of the type system allows architects to use many different approaches to dividing and capturing various aspects of design expertise. By providing many different ways to structure and capture their expertise, Armani allows designers to find or create a way to capture and structure their expertise so that it meets their specific needs.

A second benefit is that moving these sophisticated specifications into the type system allows them to be checked directly by the typechecker, which eliminates the need to write independent analysis tools for many kinds of analyses. The value of this capability should not be underestimated. Providing a sophisticated declarative language allows designers to specify their analyses based on declared properties and asserted goals. Rather than writing individual tools to verify that these desired aspects of a design instance hold, they can simply declare that they must hold and the Armani typechecker provides the verification capability.

Finally, predicate-based types provide a simple and intuitive formalism for composition. At a semantic level, each type specification defines a predicate. The semantic operation for combining types is simply a matter of creating a conjuntion of the logical expression that each of those types represents. This form of composition works well for declaring subtypes, for declaring that an instance satisfies multiple types, and for extending a typed instance with additional constraints (which has the effect of creating a new anonymous type). The modularity that the element type construct provides works particularly well with the compositionality of the formalism for capturing abstract, reusable design expertise.

The Armani design language described to this point can be readily extended to provide these benefits with a few straightforward additions. In the following sections I describe the extensions made to the instance language to support the capture of *design element types* (component, connector, port and role types) and *property types* (primitive, compound, and aliased property types).

As an aside, an important point to note about Armani's type system is that its primary purpose is to provide a form of checkable redundancy that assures the design constraints for a given type of design vocabulary are satisfied where that vocabulary is used. The type system provides a mechanism for ensuring that the system's fundamental design constraints are not violated as a design evolves over time (e.g. through system maintenance, upgrades, etc.). This role is significantly different than the role type systems typically play in programming languages. Programming language type systems are generally designed to provide statically-checkable guarantees of run-time program behavior (e.g., to insure that a function will not attempt to add a floating point value to an array of strings). The fact that there is no run-time realization of an Armani architectural specification significantly changes the purpose of the type system.

4.4.2 Declaring a design element type

In order to capture abstract design vocabulary, there are two classes of constraints that an element type specification must be capable of specifying. First, it needs to be able to specify the structure and properties that all instances of that type must possess. Requiring that all instances of a certain type contain specific structure and properties allows designers to define the aspects of that vocabulary element that remain constant across all instances of the type.

In addition to specifying these constants, a designer should also be able to precisely describe ranges of variability for instances of the type. To illustrate the distinction between these two types of constraints, consider the specification of the *client* type in Example 4.11. This specification states that although all instances that conform to the type *client* must have a property called *request-rate*, the value of that property can range from 0 to 100. These are very different types of constraint specifications. This variability represents the second class of constraint that Armani vocabulary specifications need to capture. To capture these bounds on instance variability, architects can associate invariants and heuristics with type specifications, just as they can with instance specifications. Invariants specified in an element type specification must hold for all instances declaring that type.

To address the need to express these constraints, an Armani element type specification thus defines the minimal set of structure and property fields that elements of a given type have, along with a set of invariants that must hold for all instances that satisfy the type.

In this section I illustrate how the Armani language uses its type system to capture these abstract design vocabulary descriptions and constraints. I do so by describing the syntax and semantics of *component*, *connector*, *port* and *role* types (collectively referred to as *design element types* or just *element types*). Armani *system* types, referred to as *architectural styles* (or simply *styles*), are discussed in section 4.4.6. Styles extend the capabilities of the design element types described in this section.

The informal syntax for declaring a design element type is: [12]

```
<Category> Type <TypeName> = {
      <Sequence of:    required structure and values
                       | properties
                       | explicit invariants
                       | explicit heuristics >
}
```

In the informal syntax given above, <*Category*> can be any of the literals *Component*, *Connector*, *Port*, or *Role*, and <*TypeName*> specifies a valid identifier. The body of the type declaration consists of a sequence of constraints by which instances of this type must abide. Informally, the meaning of the four kinds of constraint declarations that can be made within a type declaration are described below:

- **Required Structure**. The structural declarations in a type description T define the substructure that an element e of type T (written $e : T$) must have. Informally, for every port, role, or representation defined in T, an instance $e : T$ must have a corresponding port, role, or representation. The port, role, or representation defined in the instance must be defined with at least as much detail as its corresponding port, role, or representation in the type declaration. A more detailed specification of the semantics of required structure statements is given in table 4.12.
- **Required Properties**. A property p_i declared in a type declaration T specifies that an element $e : T$ must define the property p_I. Further, if p_I is declared to have a type and/or a value in T, p_I declared in $e : T$ must also have the same type and/or value. As with required structure, a more detailed specification of the semantics of property declarations is given in table 4.12.

[12] Complete syntactic specifications for Armani's type language are available in Appendix A's language BNF. The syntactic examples given throughout this section are informal abstract syntax specifications designed to show how the constructs can be specified and used.

- **Explicit Invariants.** In addition to the required structure and properties of a type, additional invariant constraints can be specified using Armani's constraint language. An element *e : T* must satisfy all of the invariant constraints defined in *T* in order to satisfy *T*'s predicate (and thus satisfy type *T*).
- **Explicit Heuristics** use the same predicate specification language as explicit invariants. Unlike invariants, though, heuristics are not considered in determining whether an element *e* satisfies a type *T*. Violations of type heuristics can be flagged during constraint analysis or analyzed by external tools, if desired, but the heuristics themselves are not part of a type's predicate. The heuristics construct provides architects and designers with a way to capture design "rules of thumb" that are less strict than invariants.

An element *e : T* satisfies type *T*'s predicate if *e* contains all of the required structure and properties specified in *T*, and *e* satisfies all of the invariant predicates defined in *T*.[13]

Type names are lexically scoped. Types may be declared within the global design namespace or within a style specification. Types with global scope are visible within all systems or styles declared in that global scope and types defined within a style specification are visible to all other declarations in the style, all of that style's substyles, and all system that claim to be built in that style.

The following example shows a type specification that declares constraints that must be satisfied by all instances of the type in the form of required minimal structure and predicates that must be maintained. Keywords are indicated with boldface type, comments with ligher text.

The *Client* type specification in example 4.11 imposes the following structural and invariant constraints on component instance *C : Client*:

Structural constraints:

- A *Client* instance must have a port called *request*, with a property called *protocol*. The protocol property must be of type CSProtocolT and have a value of *rpc-client*.
- A *Client* instance must have a property called *request-rate* of type *float*. The default value of 0.0 can be overriden with an *extended with {... }* clause, but the initial value for this property on all Client instances created with the *new* operator will be 0.0.

Invariant constraints:

- All ports of a client must have a property named *protocol*, which has a value of *rpc-client*. This is not redundant with the specification of the request port because a designer instantiating an instance of this type can add additional ports. This invariant insures that all of these additional ports have a protocol property with a value of rpc-client.
- There may be no more than 5 ports on a *Client* instance.

[13] Readers interested in a more detailed and rigorous specification of the syntax and semantics of Armani's type system are encouraged to see the full language specification provided in [Mon98].

- The request-rate property of a *Client* component must have a value greater than 0.

The heuristic constraint that the request-rate property of an instance of a *Client* component have a value less than 100 is not considered in determining whether that instance satisfies the *Client* type, though typechecking tools that evaluate instances of the client type should raise a warning that the heuristic has been violated if the instance's request-rate property has a value of 100 or greater.

```
Component Type Client = {

 // Declare the minimal structure that must exist. In this case, it says that an instance
 // of this type must have a port called request, and that port must have the protocol
 // rpc-client.
 Port Request = { Property protocol : CSProtocolT = rpc-client };

 // The next declaration says that a client must have a property of type "float" called
 // "request-rate." It also provides a default value for that property, which can be
 // changed when an instance of this type is created.
 Property request-rate : float << default = 0.0 >>;

 // Now specify the invariants that all elements that claim to satisfy this type must possess.

 // all ports must support the rpc-client protocol. This rule applies to all additional reports
 // that an instance of the type might add to the client.
 Invariant forall p in self.Ports • p.protocol = rpc-client;

 // there may be no more than 5 ports on a client
 Invariant size(self.Ports) <= 5;

 // The request rate must be a non-negative value
 Invariant request-rate >= 0;

 // Specify a heuristic indicating the request rate should not exceed 100
 Heuristic request-rate < 100;
}
```

Example 4.11: Declaring component type *Client*

Informal Element type Semantics

A type specification defines the minimal set of structure and property fields that elements of a given type have, along with a set of invariants that must hold for all instances that satisfy the type. Every type *T* can be converted to a boolean function F_t that takes a single element E as an argument. If the function $F_t(E)$ evaluates to true for element *E*, then element *E* satisfies type *T* (written *T*(*E*) in table 4.12). Each type's predicate function determines whether instance *E* satisfies the structural requirements and invariants of type *T*. Table 4.12 describes the informal semantics of structural declarations in an element type specification. [Mon98] provides a more detailed discussion of the semantics of Armani's type system.

4.4.3 Creating a simple instance of a typed architectural element

Design element type specifications capture reusable design vocabulary. Moving the design expertise that these type specifications capture from the abstract realm of reusable predicate specifications to concrete instances in a design requires only that a designer instantiates an instance of the type. The following syntax specification and examples illustrate this instantiation process.

Instances of the four basic architectural elements - components, connectors, ports, and roles, can be created with the following (informal) syntax:

```
<Category> <InstanceName> [: <TypeName> ] = <Value> ;

where
  <value> ::= ( { <sequence of property and structure specs.> } | new <TypeName> )
                    ( extended with <value> )*
```

Specifying an explicit type for an instance is optional. If no type is explicitly declared for an individual instance, then the type of that instance defaults to <Category>. Consider the following example of a component declared without an explicit type declaration:

```
Component C = { Port input; } ;
```

In this instance, the value of component *C* is *{ Port input }* which satisfies the constraints of the *Component* type, so this instance declaration is valid.

When an instance is explicitly typed, as in the following example, the value on the right hand side of the "=" token must satisfy the predicate defined by the declared type. Consider the following example:

```
Component C : Client = new Client;
```

In this example, a component *C* is declared to satisfy type *Client*. The value of *C* is defined using the Armani ***new*** operator. The expression ***new*** *<TypeName>* creates a value expression consisting of the minimal structure declared in the declaration of *<TypeName>* with default values applied to properties as specified in the type specification. Properties with no default value provided in the type declaration have undefined values in the instance generated.

Declaration Type	Example	Meaning
Structural element C with no type or value declaration	`Port C;`	Forall elements E s.t. E declares type T (written E:T), T(E) implies E has the element named C as a child.
Structural element C with a type but no value declaration	`Port C : t';`	Forall elements E s.t. E:T, T(E) implies E has the element named C as a child, and that C satisfies t' (t'(C))
Structural element C with a type and a value declaration	`Port C : t' = { Property j:t'' = bar};`	Forall elements E s.t. E:T, T(E) implies E has the element named C as a child, and t'(C) and C has the property j:t" with a value of bar.
Property named P with no type or value given	`Property P;`	Forall elements E st E:T, T(E) implies E has the property P of type "Property."
Property named P with a type t' specified, but no value given	`Property P : t';`	Forall elements E st E:T, T(E) implies E has the property P of type t'. P's value is unconstrained beyond the requirement that the value of P satisfy type t'.
Property named P with a type t' specified and a default value v given.	`Property P : t' <<default=v>>;`	Forall elements E st E:T, T(E) implies E has the property P of type t'. P's value defaults to v when a new instance of type T is created but the <<default = v>> clause is simply a convenience that the type has no obligation to maintain. The <<... >> notation specifies that "default = v" is a meta-property.
Property named P with a type t' specified and a value v assigned directly to the property	`Property P:t' = v;`	Forall elements E st E:T, T(E) implies E has the property P of type t' and P's value must be v. This statement declares a constant valued property for the type.

Table 4.12 Structural Specification Semantics

Using the Client type defined in example 4.11, the previous example creates a component with the following canonical structure:

```
Component C : Client = {
        Port Request = { Property protocol : CSProtocolT = rpc-client }
        Property request-rate : float = 0.0;
}
```

This default Client component satisfies the invariants and heuristics declared in the Client type definition.

It is possible to associate non-default values with an element created from a given type using the ***extended with*** *<value>* construct. The following example illustrates a client with an additional port and an additional property.

```
Component C' : Client = new Client extended with {
        Port ExtraPort = {  Property protocol : CSProtocolT = rpc-client;
                            Property primary-port = true };
        Property request-rate : float = 5.0;
}
```

This declaration would result in the creation of a new component C' with the following structure:

```
Component C' : Client = {
        Port Request = { Property protocol : CSProtocolT = rpc-client } ;
        Port ExtraPort = {  Property protocol : CSProtocolT = rpc-client};
                            Property primary-port = true;};
        Property request-rate : float = 5.0;
}
```

In this example, the default constructor is extended with new property values that either add new structure and values or override the default structure and value of the type. The value that is assigned to C' in this case is the unification of the structure declared with the *extended with {... }* clause and the structure that is created with the *new<TypeName>* constructor.

The basic unification algorithm is quite simple. An instance of the target design element is created. The structure and properties defined in the declared type and all of that type's supertypes are copied into the instance. Each entity (property, structure, or design rule) from the *extended with {... }* clause is then copied into the instance. For each term copied, if there is no structure or property already in the instance with the same name then the new term is copied directly without any problem. If there is, however, an entity with that name already in the instance then the algorithm checks if the new type, value, and substructure of the entity being copied is consistent with the structure or property that already exists in the instance. If they are consistent, or if the new entity adds additional information then the new new information is added and the next entity to add to the instance is selected and tested. If the entity to add is not consistent with the entity already in the instance, however, (such as if the extended with clause attempted to redefine the type of a property) then an instantiation error occurs and the unification algorithm aborts unsuccessfully. The complete algorithm for unifying substructure of an element using the *extended with {... }* construct is given in [Mon98] along with a detailed specification of the semantics of the *extended with* construct.

Types and instances

As this discussion indicates, one of the more unusual aspects of Armani's type system is that it blurs the distinction between types and instances. The syntactic declaration of an element type is very similar to the syntactic declaration of an element instance. Their semantic representations are very similar as well.

An implication of this approach is that architects using the language have a great deal of flexibility in deciding how to divide the specification of expertise between types and instances. They can specify all of the design details in the element instances themselves, using only minimal type specifications or they can create an abstract type for each individual element in the design and declare a single instance of each type. Alternatively, they can abstract common aspects of design elements into a variety of types and then mix-and-match those types amongst his design element instances to make it explicit that elements with common aspects share a common representation for those aspects.

In a given design situation, any of these alternatives might be appropriate. Armani's flexibility allows an architect to take advantage of all three of these approaches, or to create a hybrid approach that uses some combination the three. Because type and instance declarations are so similar, it is very easy to move a design concept from an instance into a type, or vice-versa. As a result, an architect can quickly experiment with a variety of design options before settling on a specific approach. Because the cost of modification is so low, however, once an approach has been selected, the cost of revisiting or changing the decision to use that approach is also relatively low.

4.4.4 Design element subtypes

The Armani design language's fifth requirement calls for the language to support incremental capture of architectural design expertise. One of the key ways that the language supports this requirement is with its flexible subtyping discipline. Design element types can be readily extended with additional structure, properties, and constraints to form new types. These new types encapsulate existing design expertise and extend it with additional expertise.

In order to maintain the composability and modularity it requires, Armani supports a strict form of subtyping that ensures substitutability between subtypes and supertypes. That is, if type *T'* is a subtype of type *T* (written $T' \leq T$), then any element that satisfies *T'* may be used wherever an element of type *T* is required. The following informal syntax describes Armani's subtyping construct.

```
<Category> Type <SubTypeName> extends <SuperTypeName>+ with {
      <Sequence of:    required structure and values
                       | properties
                       | explicit invariants
                       | explicit heuristics >
}
```

The semantics of this construct are straightforward. The new (sub)type <*SubTypeName*> consists of the unification of the structural requirements of all supertypes with the new structural declarations, and the conjunction of the invariant and heuristic predicates of all supertypes with the new invariant and heuristic declarations. As the syntax specification indicates, a type can have an arbitrary number of supertypes. All instances of the subtype are also instances of all of the supertypes and satisfy the constraints of both the supertypes and the constraints listed in the *extends ... with {...}* clause.

Consider the following example:

```
Component Type BlockingClient extends Client with {
        Port BlockingRequest = {Property protocol = rpc-client};
        Property blocking : boolean = true;
        Property timeout-sec : float << default = 30.0 >>;

        Invariant timeout-sec < 60.0;
}
```

An instance of a BlockingClient type component has all of the structure and rules to maintain that a Client type component would have. It also has the additional properties and rules given in this specification. Using the Client type definition from example 4.11, the previous type declaration is equivalent to declaring the BlockingClient type without subclassing as shown below:

```
Component Type BlockingClient = {
        Port Request = {Property protocol = rpc-client};
        Port BlockingRequest = {Property protocol = rpc-client};
        Property request-rate : float << default = 0 >>;
        Property blocking : boolean = true;

        Invariants {
                Forall p in self.Ports | p.protocol = rpc-client;
                Size(Ports) <= 5;
                request-rate >= 0;
                timeout-sec < 60.0;
        };

        Heuristic request-rate < 100;
}
```

4.4.5 Property Types

The discussion of the type system to this point has primarily described its use for design vocabulary elements. *Properties* of design elements can also be typed. The type system used for element properties uses a syntax and semantics similar to the design element type system's, though the constraints that can be imposed on properties are more limited than those that can be imposed on design elements.

As described in section 4.2.3, a property of a design element is a name with which a value and a type can be associated. The purpose of a property type is to define the range and structure of values that can be applied to the named property. A property type declaration can define an atomic type, an enumerated type, a compound type (set, sequence or record), or alias an existing type definition.

The declaration of a type can, but need not, be separated from the use of that type in specific properties. Explicitly named types are declared with the following (informal) syntax:

```
Property Type <TypeName> = <TypeStructure>;
```

<TypeName> is an identifier that is associated with <TypeStructure>. Semantically, <TypeStructure> specifies a predicate that defines the set of valid values for the type and in doing so defines the structure that values of the type must posses.

Typed property instances that use previously defined property types are declared with the following syntax:

```
Property <PropertyName> : <TypeName> = <PropertyValue>;
```

The property named <PropertyName> is associated with the element in whose scope it is declared. The type of <PropertyName> is explicitly specified with the ": <Typename>" notation.

All property instances in Armani must be typed. Although it is convenient to reuse previously defined property types or built-in atomic types, it is not necessary to do so. A property instance can declare an *anonymous* compound type, as the following example illustrates:

```
Component foo = {
     Property rate : Record [ speed : int; units : string ] =
                              [ speed : int = 100; units : string = "kb/s" ];
};
```

In this example property *foo.rate* has declared a new anonymous type – a record with the fields *speed* (of type *int*) and *units* (of type *string*). This new type is not visible to any other property or element (hence the term anonymous) but it specifies the structure that the value of the property must possess. Semantically, an anonymous type declaration in the context of a property instance specifies a predicate that the value of that property instance must satisfy.

Property Type Semantics

A property type, like a design element type, specifies a predicate that defines a set of valid values for instances of that type. An instance of a property is type correct if its value is an element of the set described by its type. The range of type predicates that can be defined for property types is more limited than those that can be defined for element types. Specifically, in the version of the Armani language completed to demonstrate this thesis a property type defines only structural predicates. It is not possible to associate arbitrary invariants with a property type and the language provides no support for property subtypes. This limitation is made in the interest of keeping the property type system relatively simple. Extending the property type language to include support for arbitrary invariants should, however, be reasonably straightforward. A more detailed semantic specification of Armani's property type constructs is provided in [Mon98].

4.4.6 Architectural Styles

Types and design rules provide mechanisms for capturing and encapsulating design expertise in the form of design vocabulary and constraints. Although individual types and design rules can be useful by themselves, expertise of this sort tends to be more useful when packaged as part of a coherent collection of related vocabulary and constraints. For example, defining a vocabulary element called a *server* is not nearly as useful as defining a full set of vocabulary and design rules for creating *client-server* systems. Armani's *style* construct provides the ability to aggregate and package related vocabulary and constraints.

An architectural style is fundamentally a system type – it defines a predicate against which system instances can be evaluated. In addition to their role as system types, however, styles also define namespaces for specifying vocabulary type declarations and design analyses. A style specification thus defines a set of vocabulary type definitions, a set of design rules, a set of design analyses,[14] and a set of minimal required structure that all systems built in that style must provide. Any or all of these sets may be empty. Styles obey all of the rules and semantics of types presented thus far, with some additional syntax and semantics to support the style's use as a namespace.

The informal syntax for defining a style is:

```
Style <style-name> = { <style-element>* } ;

  Or

Style <style-name> extends <super-style-name>+ with { <style-element>* } ;

  where:

<style-element> ::= <Sequence of:   required structure and values
                                    | required properties
                                    | explicit invariants
                                    | explicit heuristics
                                    | design analyses
                                    | type definitions >
```

The syntax and semantics for declaring individual type specifications, design rules, and design analyses were described earlier in the chapter. A style is a named collection (or a package) of such constructs. In its role as a system type, a style constrains the design of systems defined in that style by making design vocabulary available for use in the system instance, defining the required structure and design rules (invariants and heuristics) that all systems built in that style must provide and obey. Example 4.13 illustrates these constructs with a simple style specification.

[14] A *design analysis* is a named, parameterized predicate that design rules can invoke to perform common evaluations.

```
Style naiveClientServerStyle = {

  // define the style's vocabulary, port and role interfaces first
  Port Type naiveClientPortT = {...};
  Port Type naiveServerPortT = {...};
  Role Type clientSideRoleT = {...};
  Role Type serverSideRoleT = {...};

  // define the generic client vocabulary element
  Component Type naiveClientT = {
        Port sendRequest : naiveClientPortT;
  };

  // define the generic server vocabulary element
  Component Type naiveServerT = {
        Port receiveRequest : naiveServerPortT;
        Property multiThreaded : boolean << default : boolean = false; >>;
        Property max-concurrent-requests : int;
  };

  // define the generic binary client-server connector
  Connector Type csConnT = {
        Role clientSide : clientSideRoleT;
        Role serverSide : serverSideRoleT;
        Property blocking : boolean << default : boolean = true>>;
        Invariant size(self.roles) == 2; // all csConnT's are binary connectors
  };

  // limit the vocabulary types used in this style to  naiveClientT's, naiveServerT's and csConnT connectors.
  invariant forall comp : component in self.components |
        (declaresType(comp, naiveClientT) AND satisfiesType(comp, naiveClientT))
        OR (declaresType(comp, naiveServerT) AND satisfiesType(comp, naiveServerT));

  invariant forall conn : connector in self.connectors |
        declaresType(conn, csConnT) AND satisfiesType(conn, csConnT);

  // specify topological attachment constraints:
  invariant forall c1 : component in self.components |
        forall c2 : component in self.components | connected(c1,c2) ->
              (declaresType(c1,naiveClientT) AND declaresType(c2,naiveServerT))
              OR (declaresType (c1,naiveServerT) AND declaresType(c2,naiveClientT));

  // make sure that all of the attachments are valid...
  invariant forall a : attachment in self.attachments |
        (declaresType(a.port, naiveClientPortT) -> declaresType(a.role, clientSideRoleT))
        AND (declaresType(a.port, naiveServerPortT) -> declaresType(a.role, serverSideRoleT));
};
```

Figure 4.13: Naïve client-server style specification example illustrates style structure

System instances may make use of the design vocabulary and analyses packaged in a style by declaring that the system satisfies a style. The syntax for declaring that a system instance satisfies a style is analogous to declaring that a design element satisfies an element type:

```
System <SystemName> : <StyleName> = { <system-decl-body> } ;
```

When a system instance declares that it is designed in a specific style the names of all of the types and design analyses declared in that style are visible within the system instance. Further, all of the design rules contained in the style definition must hold over the system instance. That is, the design rules in the style definition take effect in the scope of the system instance, binding the concrete elements in the system instance to the appropriate abstract design rules of the style. Declaring that a system is designed in a specific style indicates that the design rules declared in that style must be maintained in the system instance. Failure to satisfy these constraints constitutes a type error.

The set of type specifications given in a style declaration provide vocabulary elements that can be used within system specifications in that style. The system definition is not, however, limited to using only the types provided by the style unless there is a design rule that explicitly limits the types of vocabulary that can be used. Design elements within the system instance that claim to satisfy a type defined in the style must, however, satisfy the type predicate given in the style definition.

Instantiating style instances

The *new* operator defined earlier in this chapter for creating minimal instances of simple element types can also be used with styles to create systems with the minimal required structure to satisfy the style specification. As with simple element types, the *extended with* construct can be used to extend the minimal structure provided by *new* and customize the created system. The basic syntax for creating a new minimal instance of a style follows:

```
System <sys-name> : <style-name> = new <style-name> [ extended with {...} ] ;
```

The semantics for using the new operator with systems are analagous to the semantics for using new with simple element types.

Substyles

Because the style construct is based on Armani's element type constructs, the type system's notion of subtyping extends to styles as well. Using this subtyping notion, a style can extend an existing style to make use of the types and design rules defined in the existing style. The existing style becomes the *superstyle*, and the newly defined style is the *substyle*. The following example illustrates such an extension:

```
Style super = { ... };

Style sub extends super with {
        Component type new-component = { ... };
        Invariant forall x in self.components • foo(x));
};
```

In this example a new style called *sub* extends an existing style called *super*. *Sub* consists of the union of the types, design rules, design analyses, and structure defined in both *super* and *sub*. A substyle may not redefine types or design rules named in the superstyle. It may, however, create new types that extend the types defined in a superstyle. Because the substyling operation only allows additional types, design rules, and structure to be added to a style, any system that satisfies the constraints of *sub* will also satisfy the constraints of super.

4.4.7 Multiple types

Implicit in the discussion of the previous three sections is the fact that Armani supports a typing discipline in which instances can declare and satisfy multiple types and subtypes can declare multiple supertypes. This capabilty allows designers to explicitly declare that a single design element plays multiple roles, or that a single element has the aspects and properties of multiple types of design elements. As a result, it is possible to abstract specific, common aspects of design elements into an appropriate collection of types and compose those aspects into instances with the desired properties by simply selecting an appropriate set of types for the given instance. The instance inherits all of the aspects captured in each of the types it declares that it satisfies.

For example, three orthogonal architectural aspects of components might be captured in three independent component types called *supports-transactions*, *multi-threaded*, and *persists-data*. Each of these component types defines just the properties and structure of a component that are required for the aspects that the type captures. To make a new instance of a database component that has all of these properties an architect could simply use the following declaration:

```
Component superDB : supports-transactions, multi-threaded, persists-data = ...;
```

The architect would, of course, probably also want to extend the instance with additional information specific to that instance. This capability is supported with the *extended with* construct.

This technique can also be used to create new subtypes that reuse all of the specification details of their supertypes. The previous example can be readily modified to define a new type of database component rather than an instance, as the following declaration illustrates:

```
Component Type superDBT extends supports-transactions,
                                multi-threaded,
                                persists-data with {...};
```

The ability to declare multiple types for instances and multiple supertypes for type declarations with such ease comes directly from the predicate foundation of Armani's type system. Predicates are highly modular and readily composed through conjunction. Likewise, this capability extends to all types of architectural element types, even styles.

This capability has proven to be particularly useful. The case studies in Chapter 7 and 8 illustrate that most of architects who have taken advantage of this capability have found it to be powerful, flexible, and intuitive.

Supporting this trivial composition of types introduces the opportunity for two kinds of conflicts – naming conflicts and conceptual mismatch conflicts. Naming conflicts are relatively easy to deal with. The use of an ambiguous name within a type or instance specification is simply an error. A language user can avoided naming conflicts by qualifying references to names that are shared by multiple supertypes (or declared in multiple types that the element instantiates). For example, the following pair of style declarations both define the client type. When the sample system instantiates a *client*-typed component it needs to unambiguously specify which of the *client* type declarations it is instantiating.

```
Style generic-cs = { ... Component Type client = {...} ... };
Style special-cs = { ... Component Type client = {...} ... };

System sample : generic-cs, special-cs = {
  Component generic-client : generic-cs.client = {...};
  Component special-client : special-cs.client = {...};
}
```

Qualification with a type name is required only where the lack of a qualifying identifier leads to ambiguity.

The second type of conflict that can occur when using multiple styles within a single system are conceptual mismatches. These conflicts occur because the styles being used are fundamentally incompatible with each other. An example of such a conflict is a system that merges a pipe-filter style, which requires that all components are filters and all connectors are pipes, with a client-server style that requires all components to be clients or servers and all connectors to be HTTP streams. Unless the required types are (accidentally) compatible with each other (e.g. instances of Filters satisfy the constraints of Client) non-empty system instances can not be created that satisfy the constraints of both styles.

It is up to Armani users to detect and avoid such conceptual conflicts. Fundamental conceptual conflicts will generally be readily apparent to the user because he or she is unable to instantiate the structure or properties that desired without creating type errors. In general, the ability to detect deep conceptual mismatches also requires a degree of taste, judgement, and experience on the part of the architect using the tool. Using multiple styles in a single system instance expands the vocabulary available for use in that system but generally constrains the design of the system further by introducing additional design constraints. As the previous example indicates, it is possible to overly constrain a design by using multiple styles. Tools can be developed to detect obvious style incompatibilities but they will not eliminate the need to be careful when using multiple styles for a single system instance.

4.4.8 Formal type system semantics

Throughout this chapter's presentation of Armani's type system I have repeatedly deferred a detailed discussion of the type system's formal semantics. Rather than repeat the lengthy presentation of the language's semantics that can be found in [Mon98], in this section I provide a high-level overview of the approach taken to formalize the type system's semantics.

Armani's predicate-based type systems can be represented denotationally with a set-based formalism. Using this formalism, all entities in the syntactic domain are mapped to elements and sets in the semantic domain. Typechecking is reduced to a test for set membership. The typechecking rules used are sound if they guarantee that all entities e that claim to satisfy a type T in the syntactic world correspond to an element in the semantic domain that is a member of the set defined by T in the semantic domain. More formally, the following equation must hold, where M defines the meaning function that maps entities from the syntactic domain to the semantic domain:

$$e : T \Rightarrow M(e) \in M(T)$$

Fortunately, Armani is not the first language to make use of a flexible predicate-based type system. As a result, defining the formal semantics for such a type system is basically a solved problem. Specifically, PVS [OS97] provides a detailed formal semantics for their predicate-based type language. Although PVS is significantly more complex and powerful than Armani (as discussed in section 3.2.2), we were able to slightly extend the semantics for a subset of the full PVS language to capture Armani's type semantics. The important aspects of this extension included adding support for record types to the PVS formal specification and modifying some of the semantic equations to reflect a subtle distinction between PVS and Armani type semantics. Specifically, instances that claim to satisfy a type in Armani may have additional structure and properties beyond those required by the type specification, whereas PVS requires that instances of a type have *exactly* the structure and properties required by the type.

4.5 Expressiveness, analyzability, and incrementality

The previous four sections presented the Armani design language and illustrated how it could be used to capture instances of software architecture specifications and software architecture design expertise. These capabilities address the language's first two requirements. In this section I argue that the language also satisfies its other requirements – expressiveness, analyzability, and incrementality.

4.5.1 Expressiveness

The notation and constructs supplied by the Armani design language must match the expressive needs of software architects.

The Armani design language presented in this chapter provides constructs for specifying abstract architectural styles and concrete instances of system designs, abstract types and concrete instances of components and connectors, abstract and concrete interfaces to components and connectors, properties that can be associated with any other construct, design constraints in the form of predicate expressions that can be enforced as heuristics or invariants, parameterized abstract design rules (called design analyses), and topological structure. All of these are captured through structural specifications and predicate expressions. All of these constructs support hierarchical decomposition and encapsulation.

As this list illustrates, the Armani design language provides enough constructs to specify architectural structure, properties, connectivity, and design rules in a variety of different ways without providing an overwhelming and semantically intractable array of constructs. All of this expressive capability emphasizes static structure and the ability to verify that properties of that structure hold.

The language's expressiveness requirement is given in terms of how the language's constructs match the expressive needs of software architects. Determining whether this is an appropriate or overly limited set of expressive capabilities is difficult without drawing on experience using the language to design software systems and capture architectural design expertise. Please see the case studies in chapters 7 and 8 for a detailed discussion of how these experiments indicate that Armani's design language is capable expressing designs and design expertise for a broad range of architectural styles. These case studies also indicate that the language provides sufficient depth of expression to capture interesting concepts within the styles.

Armani's broad array of constructs for capturing designs and design expertise, combined with the case study experiences using the language argue that the language satisfies its expressiveness requirement.

4.5.2 Analyzability

Armani's design language must support the evaluation and analysis of architectural descriptions.

Support for the analysis of architectural specifications lies at the core of the Armani design language. As a result, Armani's design language readily satisfies this requirement. The language supports two basic kinds of analysis. First, it has a "built-in" analytical capability provided by the language's typechecking system. Second, it has the ability to annotate architectural specifications with properties that can be analyzed by external tools.

Armani's typechecking process provides the language's primary "built-in" analytical capability. The typechecking process determines whether an instance of an architectural specification (a) satisfies all of its type declarations, and (b) satisfies all of its instance-specific design rules. Because Armani's type system and design constraint language allows architects to specify complex predicate constraints and types, the typechecking process provides sophisticated analytical capabilities when used properly.

Because all design rules that can be expressed in the Armani design language can be verified with Armani's typechecker, it is possible to evaluate arbitrarily complex design rules without having to make use of any analysis tools other than the Armani typechecker. As I discussed in detail in section 4.3, one of the primary issues in selecting an underlying formalism for capturing design constraints and creating a language for expressing those constraints was decidability. I had to limit the type-based analyzability of the language in two important ways to insure that the process of typechecking design specifications remained decidable. The first constraint is that quantifications over infinite sets are not supported. The second constraint is that the typechecker does not provide any meta-evaluation capability for inferring or proving properties about type and style specifications themselves. Armani's type analysis infrastructure cannot, for example, determine whether it is possible to create a valid instance of a given type *T*. Neither of these constraints proved to be particularly problematic in practice.

Although very powerful, Armani's typechecking capability neither enables nor provides all desirable forms of architectural analysis. Recognizing that only a core analytical capability should be built directly into the language infrastructure, architectural designs specified in the Armani design language can be arbitrarily annotated with properties to be analyzed and evaluated by external design tools. The determination, for example, of many emergent system properties are calculated with external tools. This approach allows the language to support arbitrary forms of analysis while limiting the number of domain-specific constructs that it needs to support natively. I discuss how Armani's environment infrastructure supports the integration of external analysis tools in detail in Chapter 5 and the case studies presented in chapters 7-8 illustrate the feasability and utility of this approach.

4.5.3 Incrementality

The language must support incremental capture of architectural design expertise and incremental modifications to architectural specifications.

The Armani design language provides a number of constructs and capabilities that allow it to meet this requirement. First, all of Armani's core constructs are explicitly designed to be incrementally composable. As a result, incrementally adding new properties, structure, design rules, type declarations, etc. to design elements and design element types is a straightforward operation. Likewise, incrementally adding the expertise and structure contained in a type specification to a subtype or an instance of a design element is trivial. As described earlier in this chapter, all that is required is simply declaring that the subtype or instance inherits from the supertype.

Second, Armani's modular language constructs underlie and enable this composability. Requiring that all design element and design expertise specifications be packaged in standard, discrete units makes it possible to provide a standard set of incremental integration rules. These rules define a framework for incrementally adding or removing structure, properties, and design rules from design element types and instances. These rules also support the incremental modification of the properties, substructure, and design rules themselves. By providing standard containers (design rules, components, etc.,) for the specification of designs and design expertise, it is easy for both machines and people to understand the meaning of composing these entities.

Third, Armani's subtyping discipline encourages designers to incrementally extend their design vocabulary and styles as needed. Types can be trivially extended with additional information by creating a subtype that defines only the incremental additions that the subtype makes to the supertype. Subtyping is a standard, well-understood mechanism for incremental extension. It works particularly well, however, with Armani's rich predicate language to support the incremental creation of design element types that capture exactly the expertise needed to meet a particular design goal.

As I discuss in Chapter 5, having a modular and incremental language for capturing architectural design expertise provides a superb foundation for building an incrementally configurable software architecture design environment. Its utility extends, however, beyond its role in customizing the Armani design environment. The ability to collect design expertise in small, modular, and composable packets simplifies the conceptual effort required to capture and express the design expertise itself. Furthermore, it allows designers to select the expertise that they need for a specific project and to quickly adapt the basic concepts that they have to work with.

4.6 Architectural specifications and implementation code

The Armani design language is fundamentally a declarative language for describing the abstract architectural structure and properties of software system designs, as well as constraints on the evolution of those designs. Because it is not intended to be used as a programming language, it does not provide constructs for describing program-level behavior or the algorithms used to implement component and connector functionality. By design, it is not possible to compile an arbitrary Armani specification into an executable system.

Although at first glance this disconnect between architectural design and program implementation might appear problematic, it provides at least two significant benefits. First, it allows architects to focus on large-scale questions about how a system's components are going to work together and reason about the system-wide properties that will emerge from the composition of the system's components and connectors. Working independent of the implementation code encourages architects to get the high-level, abstract design correct before worrying about the implementation details. Second, a great deal of design done at the architectural level of abstraction is concerned with composing entities for which the architect has no access to source code. The components and connectors with which he or she must work have been purchased from third-party suppliers who provide an interface and some form of description of the component or connector's behavioral and a-functional properties, but no source code. In these cases, mapping to the source code that implements the components is basically a non-issue, as the architect has no access to the code. The important architectural issues are determining the interfaces that the component or connector provides, it's properties, the rules that must be followed to successfully integrate the entity into a system specification, and evaluating how it will work in a proposed system design.

An important implication of this approach is that the language has a very different flavor than a programming language. Specifically, the Armani design language has no concept of an executing program, nor does it provide constructs for iteration or branching. An Armani description is simply a specification of a software system design or abstract design expertise.

Although Armani is not itself a programming language, it is possible to associate program code with an architectural specification. Specifically, source code and abstract behavioral specifications that could be used to generate souce code can be associated with design elements (or design element types) through Armani's property construct. Specifically, code and specifications can either be stored directly as the value of a property or the properties can store references to the code via a URL or file name. This has proven useful in cases where the architect working with Armani either has the source code for a component connector available, or will eventually need to implement that code.

Although Armani does not provide built-in mechanisms for rigorously and provably mapping architectural specifications to implementation code, developing ways to do so is an important direction for future work. Recent work on this topic by Moriconi et al [MQR94] provides a good start but it solves only part of the problem. UniCon's work on generating implementation code for common connection mechanisms [Shaw+95] provides another promising and less formal approach to automating the process of generating implementation code from architectural specifications.

4.7 Summary

The Armani design language described in this chapter meets its requirements. It provides a language and framework for capturing software architecture designs and design expertise. The language also supports the declarative and incrementally modifiable specification of instances of architectural designs. As a result, the Armani design language provides an infrastructure that demonstrates the first half of the thesis claim that *it is possible to capture a significant and useful collection of software architecture design expertise with a language and mechanisms for expressing design vocabulary, design rules, and architectural styles*. The case studies described in chapters 7 and 8 build on this introduction to the language and illustrate ways in which the language has been used to capture architectural styles, system descriptions, and other forms of architectural design expertise.

Chapter 5
The Armani Design Environment

The previous chapter demonstrated the first half of this dissertation's thesis: *it is possible to capture a significant and useful collection of software architecture design expertise with a language and mechanisms for expressing design vocabulary, design rules, and architectural styles*. This chapter illustrates how Armani addresses the second half of the thesis: *this captured design expertise can be used to incrementally customize software architecture design environments*.

I demonstrate this claim by developing a rapidly configurable software architecture design environment that can be incrementally customized with design expertise specifications captured in the Armani design language. In this chapter I lay out the requirements for such an environment. I then describe the architecture of the configurable Armani environment, show how it supports incremental customization, and discuss some key issues surrounding this approach. Finally, I argue that the Armani environment's architecture satisfies these requirements.

5.1 Design environment requirements

The Armani design environment's primary requirement is that an environment developer be able to rapidly reconfigure it with design expertise captured in the Armani language. This is the only requirement that needs to be met in order to demonstrate that the second claim of the thesis holds. In addition to support for incremental reconfiguration, four subsidiary requirements must also be met if the environment is to be broadly useful and sufficiently powerful – leverage, efficiency, support for external tool integration, and user-interface configurability. Taken as a whole, these five requirements appear to be applicable not only to the particular case of the Armani design environment, but also to a wide class of similar, highly-configurable systems.

Requirement 1: Incremental reconfiguration. *It must be possible to incrementally customize the Armani design environment to take advantage of architectural design expertise captured in the Armani design language.*

This is the Armani design environment's fundamental requirement. Architectural design expertise expressed in the Armani design language is valuable by itself as a set of human-readable design guidelines. This expertise becomes much more valuable, however, when Armani's tools process it and guide designers in creating and analyzing software architectures. The core capability Armani needs to provide is the ability to dynamically add, remove, and modify an environment's design expertise, updating the environment appropriately to reflect the changes.

One of the challenges introduced by this requirement is selecting the axes along which the environment can be reconfigured. As the previous chapter discussed, the Armani design language provides a basic set of design elements that the generic Armani environment uses as a baseline design vocabulary. The language also provides a variety of constructs for introducing new vocabulary, design rules, and design analyses. Linguistically, this customization can be done incrementally and at various granularities. An architect can adapt or extend design expertise at the level of styles, design element and property types, systems, or individual element instances. The Armani design environment needs to support all of the axes and granularities of configuration that the Armani design language supports. In addition, as requirements four and five describe, it also needs to support the inclusion or removal of external tools and the incremental reconfiguration of its user interface.

Requirement 2: Leverage. *The Armani design environment must provide architects with significant leverage for creating, evaluating, and manipulating designs and design expertise.*

One of the primary benefits of a good tool is that it extends and magnifies the tool user's capabilities. That is, the tool provides its user with leverage. To give architects this leverage for their design and analysis capabilities, Armani must provide a collection of tools for processing, manipulating, and analyzing designs expressed in the language. Specifically, it must provide at least the following tools:

- A parser to read textual design expertise, style, and system descriptions expressed in the Armani design language.
- Type and constraint checking tools to ensure that designs satisfy their type constraints and design rules.
- An error reporting system to alert the architect of problems or issues with designs.

In addition to these core tools, the Armani environment also needs to provide a graphical user interface for creating, displaying, documenting, and otherwise manipulating architectural specifications.

Requirement 3: Efficiency. *The Armani design environment must work efficiently enough to support the interactive creation, updating, and evaluation of architectural designs and design expertise.*

In addition to being configurable, the Armani design environment needs to provide sufficiently fast response times for interactive use on designs containing up to one thousand design elements (components, connectors, etc.). Note that this efficiency requirement is expressed completely in terms of the experience of an architect working on architectural specifications. Ensuring efficiency for faster response times or larger designs is beyond the scope or needs of this tool. Once a design reaches more than one thousand design elements it generally includes much more detail than is required at the architectural level of design.

Requirement 4: External tool integration. *The Armani design environment must allow "external" tools to access, manipulate, and evaluate Armani design representations.*

The design environment relies on its design language to capture declarative design expertise. The environment also, however, needs to be able to capture design analyses and operations that are best expressed algorithmically. As described in the previous chapter, this category of design expertise is called *operational* design expertise.

Operational design expertise is frequently encapsulated in legacy design and analysis tools. When no mechanism exists that encapsulates the desired operational design expertise, however, it is frequently useful to create new tools to capture it. For the purposes of this dissertation, I define *external tools* as tools that operate on Armani design representations without being part of the core Armani infrastructure. Both legacy tools and freshly-built point solutions are considered external tools.

To support the capture and use of operational design expertise, the Armani environment must allow external tools to access, manipulate, and evaluate Armani design representations.

Requirement 5: User interface configurability. *The Armani design environment's user interface must support user-defined graphical depictions of designs and design elements.*

Although the Armani design language provides a precise and flexible notation for describing software architectures and architectural design expertise, many architects prefer to work with graphical notations. To address this need, Armani must provide a user interface that architects can use to create and manipulate architectural specifications graphically.

The graphical notations that architects use to represent their designs can vary significantly between different styles and even between different individuals. It is therefore important that the environment's user interface be customizable and reconfigurable so that architects can match the visual depictions they would like to use for their design vocabulary and system specifications with the underlying semantic representations. It is also important that the architect be able to view either the graphical representation of a design or its underlying textual Armani representation. Taking this requirement one step further, architects should be able to select whether they would like to use a text-based interface or a graphical interface with Armani.

Part of the challenge in satisfying this requirement is determining which parts of the user interface should be fixed and which should be variable. The other aspect is the need to provide ways to gracefully integrate external tools' user interfaces with the core Armani environment's user interface.

5.2 The Armani design environment architecture

In this section I introduce the Armani design environment's architecture. First, I describe the core shared infrastructure that is common to all design environments. I then present extensions to this architecture that support the integration of external tools into the environment. Finally, I discuss how the architecture of this environment supports user interface customization.

5.2.1 Core shared environment infrastructure

All Armani design environments share a set of common infrastructure. This core infrastructure defines the basic, generic Armani design environment from which all custom Armani environments are derived. The heart of this common infrastructure is a component called the *architecture design representation*, or ADR. The ADR stores object-based, programmatically-manipulable, representations of Armani designs and design expertise.

In addition to the ADR, Armani provides a set of basic tools for manipulating, persisting, and evaluating Armani designs and design expertise. The following five tools round out the baseline shared infrastructure:

- A **parser** that reads textual design expertise, style, and system descriptions expressed in the Armani design language and converts them into an object-based representation in the ADR.
- An **unparser** that exports Armani design specifications from their object-based representations in the ADR to text.
- A **type manager** that verifies designs are type-correct and that they satisfy their design rules.
- An **analysis engine** that evaluates design analysis expressions in the context of specific designs or design elements.
- An **error reporting system** that alerts environment users of design problems.

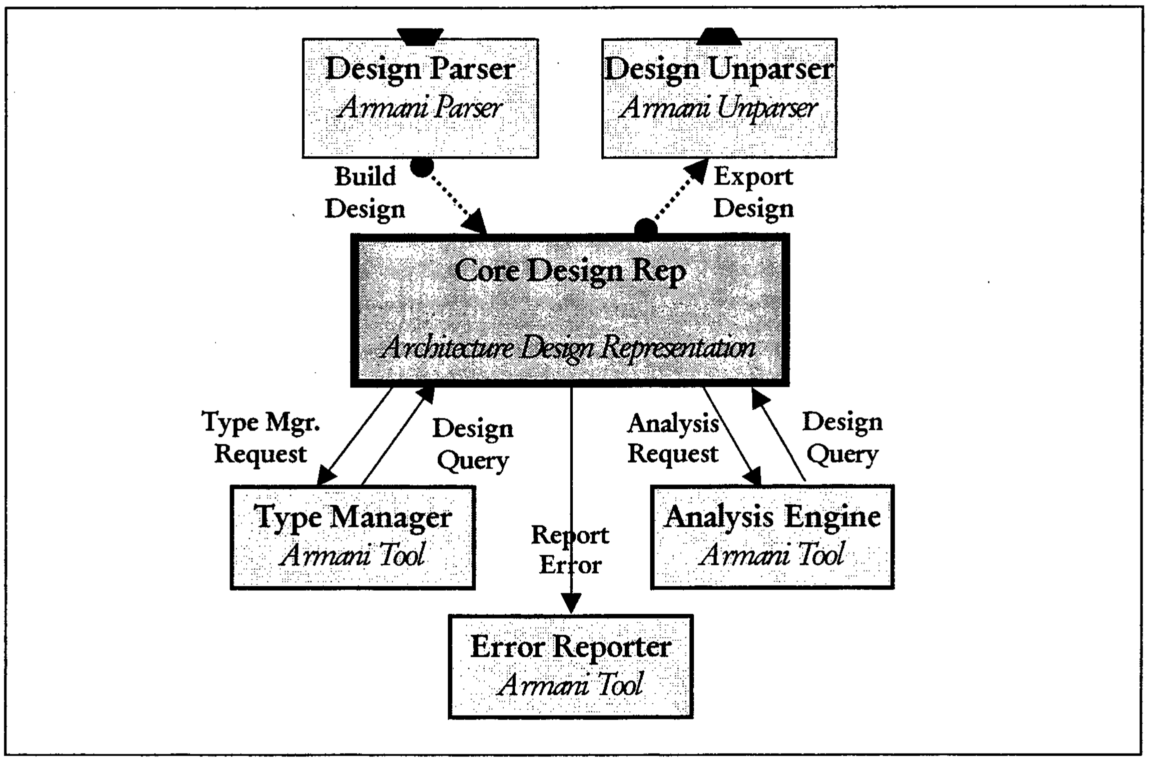


Figure 5.1: Architecture of the Armani design environment's core shared infrastructure

The diagram in figure 5.1 depicts Armani's basic architecture. As the figure indicates, all tools in a baseline Armani environment share a common architecture design representation component. This set of core tools supports the ability to capture and analyze both architectural specifications and architectural design expertise expressed in the Armani design language. It also provides the ability to evaluate individual designs to verify that they conform to their styles and design rules and to report any anomalies discovered in evaluating the design.

This baseline infrastructure is clearly missing some important components and functionality. The most obvious missing component is a user-interface. Even without a user interface, however, this collection of components can be compiled and executed. In its minimal configuration, the environment simply reads and parses Armani design specifications, evaluates the designs for type-correctness, and issues a report on any errors that it finds in the course of typechecking. All of this is done by invoking the tools through the operating system's command shell. Output is simply streamed to the standard output device.

This baseline environment encapsulates Armani's core incremental configurability capabilities. Architects and style developers can customize the expertise captured by the basic design environment by loading design vocabulary, rules, analyses, and style specifications captured with the Armani design language into the environment. In loading this expertise, an environment designer (or architect) modifies the vocabulary and semantics of design that the environment supports, as well as the set of design expertise available in the ADR. The *Type*

Manager and *Analysis Engine* components use this expertise in evaluating architecture specifications.

In the next section I will discuss how this basic infrastructure can be extended with additional components and tools.

5.2.2 Extending the environment with external tools

The ability to load the core design representation with design expertise is a first step towards rapid and incremental environment customization. This capability allows a designer to make both coarse-grained and fine-grained modifications to the design expertise encapsulated in his or her environment. The *declarative* design expertise captured in the styles, designs, design rules, and design analyses that can be loaded into the base environment encompasses an important and significant class of design expertise. Architects also, however, need to be able to take advantage of *operational* design expertise, or expertise that is best captured algorithmically (c.f. Section 4.4).

Operational design expertise is captured in an Armani environment by adding tools to the environment that perform the desired operations. These tools, referred to as *external* tools, fall into two broad categories – *legacy* tools that were built independent of the Armani environment but need to interoperate with the environment, and *Armani-specific* tools that were built explicitly as additions to the Armani environment.

Armani provides three types of connectors for integrating external tools with its design environment, each of which encapsulate a significantly different approach. All of these connectors can be used to integrate either legacy or Armani-specific tools, though their applicability for each varies widely. Figure 5.2 builds on Figure 5.1, illustrating how these connectors can be used to integrate three different external tools with the baseline Armani design environment.

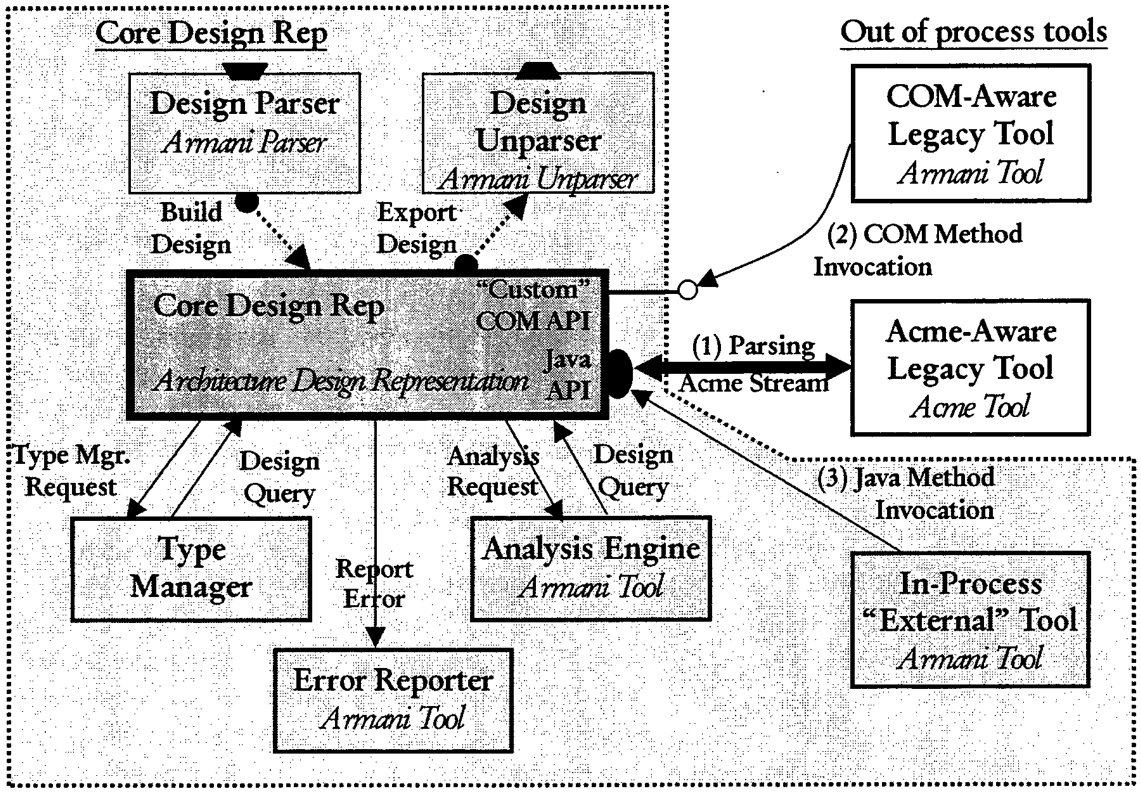


Figure 5.2: Three connection mechanisms for extending the Armani core shared infrastructure with external tools. Connector (1) streams Acme between the Armani, Core Design Rep and an Acme-Aware tool, connector (2) uses a COM-based interface to connect a legacy tool to the core infrastructure, and connector (3) uses direct Java method invocation to link interact an "external" tool that is loaded directly into the Core Design Rep's process with the Core Design Rep.

The three specific external tool integration connectors provided for integrating external tools with Armani are:

- **Acme.** The Acme tool integration connector streams textual Acme design representations between the Armani environment and an external tool. Acme, described in detail in section 3.1.2, is an interchange standard for architecture design specifications. As an emerging interchange standard, the Acme-based connector provides a low-cost integration mechanism for a wide selection of architectural design and analysis tools.

  When transferring a design from the Armani environment to the external tool, the connector converts an Armani design object into a textual Acme description of the design, and streams that textual description to the attached external tool. It is the attached tool's responsibility to interpret the Acme specification and perform its analysis or operation on that Acme description.

When a design is passed in the other direction, from the external tool to Armani's architecture design representation component, this process is reversed. The external tool writes a stream of Acme text to the connector. The connector then parses this stream, translates it to Armani, and builds an appropriate Armani design object in the environment's ADR.

This approach provides a loose integration between the environment and the external tool. This connector is most effective when the integrated tool requires only minimal interaction with the full environment. It is, however, an effective way to integrate legacy tools that support the Acme integration standard with the Armani design environment.

- **Tool-specific design rep API connector**. The second alternative for integrating external tools with the Armani design environment is to create a connector that provides a tool-specific interface to the architectural design representation component's API. The tool-specific interface provided by this connector generally exposes less functionality than the ADR's full API but provides more semantically-specific methods. It is the connector's responsibility to convert the external tool's requests made through the thin interface into an appropriate series of requests to the ADR. The connector then has to package the ADR's response and return it to the external tool in an appropriate format.

  As an example, consider a tool that walks over an architectural design to calculate system throughput. This tool is concerned only with retrieving the components and connectors in the design, discovering the performance properties of the individual components and connectors, and discovering the system's topology. Rather than expose the ADR's entire API to the tool, the connector only exposes methods to provide specific information about a system's components, connectors, topology, and performance properties.

  This integration approach has three benefits. First, it isolates the complexity of the interaction between the tool and the environment in a single connector. Isolating this complexity simplifies the implementation of both the ADR and the external tool and makes evolutionary changes to either the ADR or the tool easier. Second, it provides an additional degree of safety for the overall environment by restricting the operations that individual tools are allowed to perform on the shared design representation. Third, from the perspective of the environment end-user this approach provides a significantly tighter integration between external tools and the environment than the Acme-based integration described previously. Although this integration approach has many benefits, it also requires the most effort on the part of the environment developer.

  This integration technique is appropriate for integrating both legacy tools and Armani-specific tools. The tool-specific connector interfaces can be either written in Java or as COM interfaces. Tools that do not understand either of these standards can frequently be wrapped by a thin shell that exposes a COM interface to the outside world and interacts with the tool using the standard understood by that tool. Likewise, because COM can be used within or across process boundaries, this integration works for tools that run both within the ADR's process boundary and for tools that execute outside of the ADR's process boundary.

- **Direct API access to the ADR.** The third type of connection available for integrating external tools with the Armani design environment allows the external tool to directly invoke Java methods through the environment's core Architecture Design Representation API. With this approach, the external tool is loaded directly into the ADR's Java process, creating an "in-process external tool" that is given full access to the Architecture Design Representation's API. An in-process tool has the same status as the Type Manager, Analysis Engine, or Error Reporter tools in the baseline Armani environment.

  The Direct-API access to the ADR approach has two important limitations. First, the tool must be written in Java to integrate properly with the ADR. Second, the tool must be highly trusted because it will have full access to the internals of the Armani environment. As a result of these constraints, this approach is most appropriate for tools that are developed directly by the environment developer explicitly for the purpose of operating over Armani designs stored in the ADR.

5.2.3 Customized user interfaces

The previous two sections described how the architecture of the Armani environment allows custom environment designers to configure an environment and integrate external tools. Although these customizations form the foundation for Armani's configurability, it is also critical that Armani's user-interface (UI) be highly configurable to convey the underlying semantic modifications to architects using the tool.

To achieve this configurability, the Armani environment provides the ability to make both coarse-grained and fine-grained modifications to Armani's user interface. The coarse-grained adaptability derives from the design decision to implement Armani's user interface as a standard external tool that has no special privileges or status in the environment. As a result, environment developers can completely replace the environment's user interface. Although this appears to be a Draconian approach to customizing the UI, it allows environment designers to integrate the core Armani infrastructure with other tools that supply (and require) their own user interfaces. Likewise, it allows environment developers to experiment with radically different user-interaction techniques on top of the same basic environment infrastructure.

One of the implications of this approach to user-interface customization is that the core Armani infrastructure binds very few design decisions regarding how fine-grained configuration should be supported in a user interface. Fine-grained configuration includes such issues as associating icons with specific types of vocabulary elements, techniques for editing property values, etc. Each instance of an Armani user-interface is free to provide its own support for such customization. In the remainder of this section I will discuss how I used variations of the external tool connector types described in the previous section to integrate three different user interfaces with the core Armani infrastructure. I will also discuss the fine-grained user-interface customization capabilities that each of these interfaces provide. Figure 5.3 illustrates the architectural approaches taken to integrate each of these user interfaces with the core Armani environment infrastructure.

Command Line Interpreter

The initial Armani user interface was a textual, tty-based, command-line interpreter that allowed an architect to load, view, modify, export, and typecheck textual specifications of Armani designs and design expertise. Because the interpreter was written in Java, the most straightforward integration method available was to simply load the interpreter into Armani's ADR process and connect it to the ADR's API with a direct Java method invocation connector. This integration approach proved both simple and effective. Figure 5.3a illustrates the architectural approach taken to add the command-line interpreter user interface.

The Armani command-line interpreter provided only minimal fine-grained customization capabilities. Generating the interpreter with a parser generator [JCC99] provided the ability to easily add new commands to the interpreter. To extend the interpreter's command set, an environment developer simply had to add a new production to the interpreter's specification along with some Java code to execute when the command was invoked. Although this process required the environment developer to modify the interpreter's source code, the code was structured in such a way that these additions were straightforward and modular.

Overall, although it was rather feature-poor and somewhat lacking in visual appeal, the Armani environment and command interpreter front-end, coupled with a good text editor, proved to be a remarkably effective design tool.

AcmeStudio

The second user interface that I integrated with the Armani core infrastructure was the AcmeStudio design environment. AcmeStudio [Kom99] provides visualization, graph layout, and simple design analysis capabilities for designs defined in the Acme design language [GWM97]. Integrating AcmeStudio with Armani proved to be an effective way to quickly put a graphical user interface on top of the Armani core infrastructure. It also added useful functionality to the AcmeStudio by providing it with a powerful design rule checking mechanism.

Figure 5.3b depicts the architecture of the core Armani infrastructure integrated with the AcmeStudio. As this figure indicates, I used an Acme tool integration connector to link AcmeStudio to the Armani infrastructure. Using this integration approach, AcmeStudio writes textual Acme descriptions of its designs, types, and styles to this connector. The Armani infrastructure reads these specifications, translates them to Armani, evaluates them for type consistency, and returns the results of its analyses back to the AcmeStudio as a stream of Acme text.

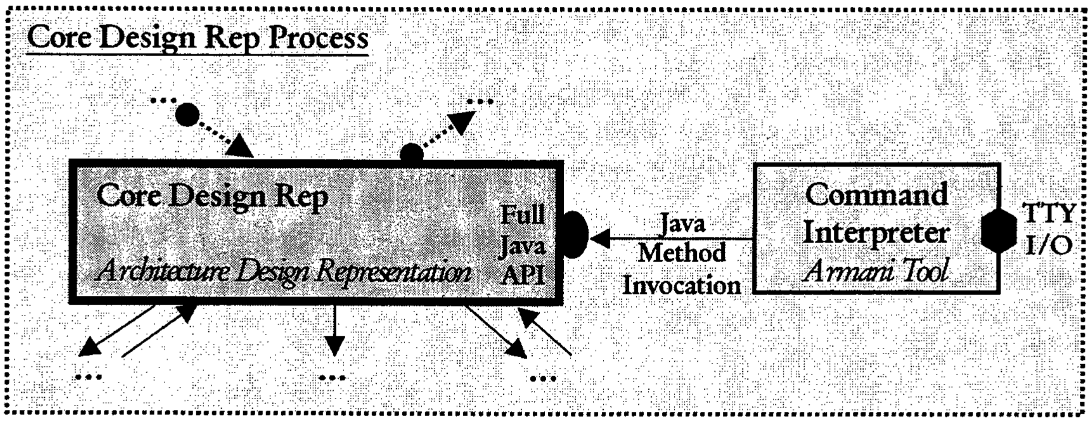


Figure 5.3a: Integrating a command interpreter user interface with a direct Java method invocation connector

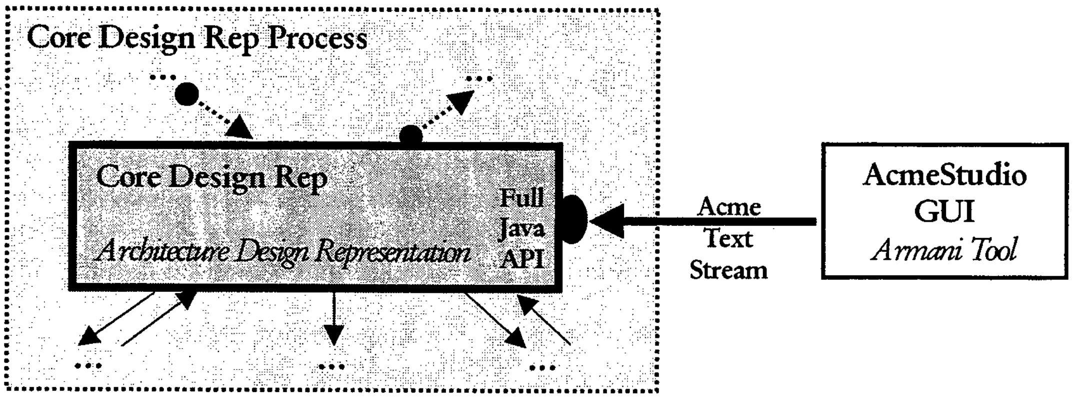


Figure 5.3b: Using a streamed Acme Text connector to integrate the AcmeStudio GUI with the Armani core design environment infrastructure.

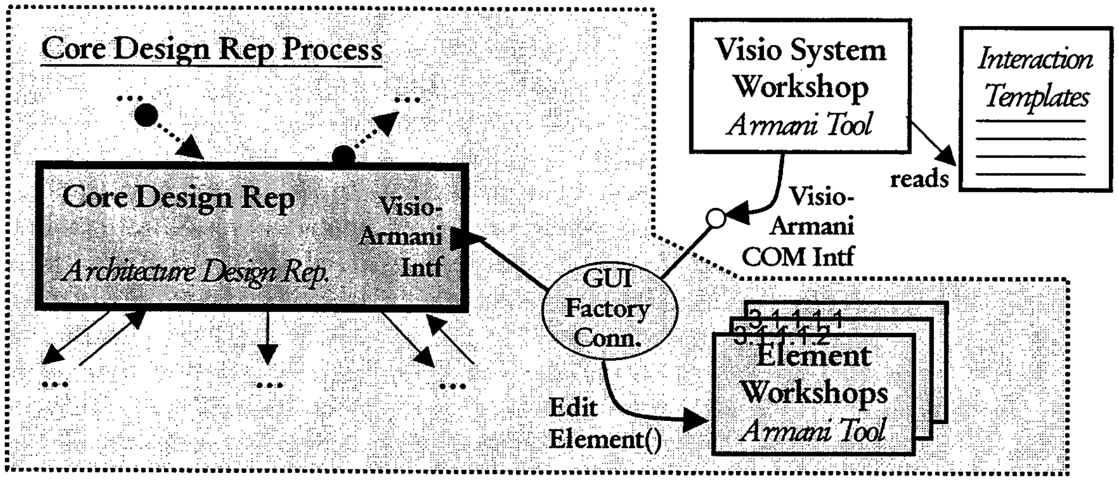


Figure 5.3c: Integrating the Visio-based, fine-grained configurable Armani interface with the GUI Factory connector and configurable editing workshops.

Figure 5.3: Three architectural options for integrating a user interface with the Armani core design environment infrastructure

AcmeStudio provides fine-grained user interface and design visualization customization. AcmeStudio's visualization capabilities are, however, geared towards Acme-based designs rather than Armani-based designs. As a result, Armani design rule and analysis constructs are encoded in AcmeStudio designs as properties that the Acme-to-Armani translation step converts into their appropriate Armani constructs. Acme does not give these constructs the first class status that Armani does, and the AcmeStudio user interface reflects this discrepancy.

Overall, this approach offered a quick way to provide a graphical editor for Armani designs.

Visio-based GUI

The third user interface that I integrated with the Armani core infrastructure was designed to be the "standard" Armani graphical user interface (GUI). This interface uses the Visio drawing package [Visio99] as the primary user interface for editing and visualization. In addition to the Visio-based design editor, this interface includes a number of Java-based GUI elements for editing individual design entities such as components, connectors, properties, and design rules.

Figure 5.3c illustrates the architectural adaptations I made to the core Armani infrastructure to integrate this GUI with Armani. As the figure indicates, components that provide a user interface for editing Armani semantic elements are called *workshops*. Visio provides the *system workshop* for editing system diagrams and the Java-based GUI components provide the workshops for editing individual design entities.

Because the various elements of this user interface are implemented with different languages and technologies, integrating them with each other and with the rest of the Armani environment infrastructure proved to be a challenge. The overall design of the Armani infrastructure, however, allowed me to address this challenge by creating a connector that managed all of the interactions between Visio, the Java-based workshops, and Armani's ADR. This connector, called the *GUI factory connector* refines the tool integration approach described for the *Tool-specific design rep API connector* in section 5.2.2. It uses the abstract factory pattern described in [Gam+95] to lazily create and invoke workshops for viewing and editing design entities.

The GUI factory connector keeps track of which visual entity corresponds to which semantic entity, and vice-versa. It is the responsibility of the factory connector to establish a direct connection between the editor and the entity being edited. When the GUI factory connector receives a request to edit a semantic entity, the connector finds or instantiates an appropriate workshop, asks that workshop to display itself and edit the requested entity. Depending on the implementation of the workshop, this step is done by handing the appropriate workshop a Java or a COM interface to the underlying semantic entity. Once this interface has been properly handed off, the GUI factory connector steps out of the way and lets the workshop directly access the underlying semantic object through this interface. As a result, after setting up the initial connection between a semantic entity and its editing workshop, the number of indirections that the workshop requires to access or update semantic information is minimal. This approach adds an additional start-up cost the first time an entity is edited or a workshop is opened, but the connection established between the workshop and edited entity is persistent through a session so the response time for future editing of that entity is nearly instantaneous.

This integration technique has four desirable characteristics. The first is that the semantic entities stored in Armani's ADR are not bound to any form of user-interface editing element until run-time. This late binding of editor selection cleanly separates semantic from visualization concerns and removes user interface issues from the implementation of the underlying semantic representation of architectural entities.

The second desirable characteristic of this integration approach is that, to the environment end-user, there appears to be a very tight integration between Armani's GUI, underlying semantic representation (ADR), and design checking tools. The environment provides very good interactive response and near-instantaneous feedback to user manipulations. The integration appears to the user to be much tighter than the integration of AcmeStudio and the Armani design checking tools. Although the appearance to the end-user is one of tight integration, the dependencies between the individual components of the Armani core infrastructure and GUI are minimal, allowing them to be maintained and evolved independent of each other.

The third desirable characteristic is that it supports a "medium-grained" form of user-interface customization. Rather than completely replacing the user interface, an environment developer can extend the existing GUI by creating new workshops for editing specific types of design elements. He can insure that the new workshop will be selected to edit the desired types of design entities by simply reconfiguring the dispatch table in the GUI factory connector. This type of configuration requires more work than the fine-grained customization capabilities provided by Visio, but significantly less work than building a new user interface from scratch.

Finally, using a configurable commercial drawing package such as Visio for a user-interface to the Armani core infrastructure provides the environment's end users with a vast array of fine-grained customization capabilities. In section 5.3.5 I discuss in detail how I was able to take advantage of Visio to provide this fine-grained user-interface customization capability.

5.3 Design environment discussion and evaluation

I now revisit each of the requirements laid out in the beginning of the chapter.

5.3.1 Incremental reconfiguration

Requirement: *It must be possible to incrementally customize the Armani design environment to take advantage of architectural design expertise captured in the Armani design language.*

Armani's core environment infrastructure is explicitly designed to support incremental adaptation. The environment provides a standard baseline infrastructure that leverages all of the Armani design language's built-in constructs and concepts (e.g. descriptions of design element types and instances, invariant and analysis specifications, etc.). An architect or environment designer can extend these basic constructs by loading additional design expertise into the environment.

The ability to load arbitrary Armani design language specifications into the core architecture design representation provides the environment developers and architects working with the Armani environment a very fine-grained customization capability. Individual design rules, vocabulary items, and analyses can be loaded, removed, or modified "on-the-fly". The design language's *style* construct also allows environment developers to use the same technique to make coarse-grained modifications to the environment's body of design expertise by loading (or removing) large collections of related expertise in a single operation. At a semantic level, for example, an environment can be trivially switched from supporting design done in a *dataflow* style to supporting design done in an *interacting processes* style by removing the first style from the environment and loading the second style into the environment.

The ability to incrementally reconfigure the design expertise used by the environment is a fundamental Armani feature that is, by itself, sufficient to minimally address the incremental reconfiguration requirement. Armani also provides two additional incremental reconfigurability capabilities that have proven very useful for custom environment developers. The first of these is the ability to easily link external tools into the environment so that it can take advantage of design expertise not readily captured in the Armani design language. The second is the ability to customize the user interface to reflect the changes in the environment's underlying design expertise, or even the tastes of individual designers. Sections 5.3.4 and 5.3.5 discuss, respectively, how each of these capabilities address Armani requirements.

5.3.2 Leverage

Requirement: *The Armani design environment must provide architects with significant leverage for creating, evaluating, and manipulating designs and design expertise.*

Although my fundamental thesis claim is simply that I *can* capture architectural design expertise with the Armani design language and that I *can* use that expertise to configure custom software architecture design environments, to be useful, these custom environments must provide the architects using them with design leverage. The Armani design environment, in both its baseline and custom forms, provides architects with at least four kinds of leverage:

1) **Evaluating Armani design language specifications.** The Armani environment provides a set of tools that allow an architect to take advantage of reusable design vocabulary, design rules, and styles when specifying architectural designs. The language processing and other environment infrastructure tools can check that architectural specifications are type-correct and that they satisfy all of the design rules used to create the system. This basic infrastructure also provides architects the ability to analyze Armani designs with analyses written directly in the Armani design language. These analyses, which can generally be written very succinctly, help architects discover whether their designs possess specific emergent, system-wide properties.

2) **Graphically creating, displaying, and manipulating designs**. The graphical user interfaces provided with the Armani environment allow designers to depict their architectural designs graphically. Although the benefits of using graphical depictions rather than textual descriptions have not been definitively proven, the use of graphical notations for specifying architectural designs and design patterns seems to match the informal ways that architects interact and discuss designs to solve design challenges. That is, the Armani GUIs support the ability to draw the ubiquitous box-and-line diagrams frequently found on whiteboards and in architecture design documents. Associating full Armani specifications with the boxes and the lines adds a significant semantic richness to the diagrams. Depicting architectural diagrams graphically also provides succinct design documentation that complements the detailed specifications available in a textual representation.

3) **Capturing and exploiting style-specific, system-specific, and domain-specific expertise.** The complete Armani system combines the Armani design language's ability to capture design expertise with the Armani environment's ability to use that expertise both analytically and constructively. This expertise can be used to evaluate individual designs for internal consistency, type-correctness, conformance to stylistic guidelines, and satisfaction of claims about the properties of the system. Captured design expertise can also be used constructively by designers who work with a palette of previously defined design elements and design rules that are customized for their specific design domain. These collections of design expertise help architects by providing consistent collections of components and connectors that are designed to work together, along with guidance on how they can be successfully composed. Further, the incremental nature of the language and the environment allows a designer to add the additional design expertise and notations that he discovers and develops in the course of architecting his systems. These additional vocabulary elements and design rules can then be reused on future projects.

4) **Integrating a suite of design tools**. The Armani environment can be extended by integrating (or building) external tools. This capability allows users of the Armani environment to take advantage of a universe of external design and analysis tools. The Armani environment's flexible integration infrastructure and Acme-compliance make it relatively easy to aggregate and take advantage of a wide variety of design tools within a single, custom Armani design environment.

By providing architects with these four areas of leverage, the Armani environment satisfies it's requirement to provide architects with design leverage.

5.3.3 Efficiency

Requirement: *The Armani design environment must work efficiently enough to support the interactive creation, updating, and evaluation of architectural designs and design expertise.*

System architects frequently have to give up system performance and efficiency to get generality and extensibility. The additional layers of abstraction that a highly generalized and flexibly extensible system require are usually inefficient and result in poorer performance than a custom-built, highly focused solution would provide. Understanding this tradeoff, I expected that I would have significant difficulty making Armani efficient enough to provide useful interactive feedback to designers working with Armani-based environments.

Specifically, there were two aspects of the Armani system that I anticipated were likely to cause significant performance problems. The first was Armani's ability to verify the type-correctness of designs and check that the designs satisfy their design rules. After experimenting with both incremental and batch algorithms for typechecking and evaluating design rules, however, I determined that this concern was unfounded. For the size of designs that Armani needs to be able to handle (designs with up to one thousand design elements), the simple batch algorithm that I used for typechecking runs quickly enough on current machines[15] to support interactive use.

It is possible to write pathologically complex design rules and analyses that cause Armani's simple type-checking algorithm to work very slowly, just as it is possible to write inefficient programs using any of a wide variety of programming languages. Experience from the case studies in Chapter 7 and Chapter 8, however, indicates that using these pathological approaches is rarely necessary and when they do arise they can generally be recast much more efficiently by rethinking the problem and/or solution. In situations where using such a pathological case appears to be unavoidable, designers have the option of rewriting their analyses or design rules directly in Java for efficiency. These Java-based rules and analyses can then be linked into the Armani environment and accessed from Armani specifications as "external" design analyses. Taking this step proved to be necessary in only one of the twelve case studies conducted – the study described in section 8.2.3.

The second aspect of the design that I anticipated was likely to cause performance problems was the separation made between the underlying design representation component and the user interface. Common wisdom holds that a graphical user interface (GUI) that is tightly integrated with its underlying data and analysis capabilities will provide much better user-interaction performance than a comparable system with a loosely integrated GUI. The degree to which this emerged as an issue varied with the three different user interfaces we put on Armani.

For the textual command-line interpreter this did not turn out to be an issue because the Armani back-end was generally able to produce its output at least as quickly as the operating system's i/o subsystem could display it to the user. When using an Acme connector to integrate the AcmeStudio front-end with the Armani infrastructure, however, the connection proved too slow to support interactive interoperation between the AcmeStudio GUI and the Armani back-end. To address this issue, all Armani analysis operations that the AcmeStudio user interface exposed to the user were presented as batch commands that could be run in the background without degrading interactive environment performance. This technique proved an effective way to use a loose and low-bandwidth connection between the two tools while hiding much of the loose separation from the environment's users.

[15] The "current machine" used as a testbed in this case is a 200Mhz Intel Pentium CPU with 64MB or RAM running Windows NT 4.0. By the time this dissertation was completed in 1999, though still usable, this machine configuration was somewhat dated and past its prime, yet Armani still ran sufficiently fast to support interactive use.

The most significant user-interface efficiency concern, however, was that the Visio-based user interface would provide acceptable performance and interactive feedback to the environment's users. This requirement was particularly important for the Visio-based GUI because of its role as Armani's primary user interface. Fortunately, the *GUI factory connector* that I created for this purpose provided sufficient performance. In this integration scheme, the GUI components store, update, and render all of the visualization information locally. Correspondingly, the core architecture design representation stores and handles all of the semantic information exclusively. This approach adds an additional start-up cost the first time an entity is edited or a workshop is opened, but the connection established between the workshop and edited entity is persistent through a session so the response time for future editing of that entity is nearly instantaneous.

Overall, this approach has proven to be an effective way to provide the loose integration between a user interface and its back end that is desirable for maintainability and separation of concerns. It has managed to do so without sacrificing the performance that is necessary in a user-centered design environment.

5.3.4 Integrating external tools

Requirement: *The Armani design environment must allow "external" tools to access, manipulate, and evaluate Armani design representations.*

Implementing the external tool integration connector described in Section 5.2.2 required that I address a number of important issues. The first of these issues is that Armani needs to provide a platform for quickly building new *Armani-specific* tools that operate directly on Armani design representations. At the same time, however, Armani also needs to support the integration of a wide variety of legacy tools that were not originally designed to work with Armani or its integration protocols. These two sub-requirements introduce a design tension and generally argue for different approaches to external tool integration.

The need to provide a platform for quickly developing tools that operate on a core design representation argues for providing a limited collection of standard, well-defined, ways for the tools to interact with the design representation. Providing a standard that binds tool interaction design decisions "correctly" for all tools reduces the work that tool builders need to do to create and integrate their tools with the environment. Armani supports this approach by providing a standard technique for writing external tools in Java that can be loaded directly into the architecture design representation (ADR) component's process. Armani also provides a standard by which these tools can register their presence and by which they can be invoked to perform their analyses. Although Armani explicitly defines the integration protocols that these tools use to interact with the design environment, the tools can also access the complete architecture design representation API if they need to. Access to this API is provided by the standard integration protocol.

To further ease the creation of new external tools, the Armani toolkit provides a significant collection of skeletal tools that perform a broad variety of common operations. Examples of the tasks performed by these skeletal tools include visiting each component or connector in a design (and possibly performing a tool-specific operation at each element visited), querying a design for all elements that have a specific property or property value, and retrieving a list of all of the types that are visible from a given scope. Tool developers can extend these skeletal tools to implement their specific operations. Providing pre-integrated implementations for these standard tool operations frees tool developers from rebuilding standard infrastructure and allows them to focus on the specific functionality that they want to add to their custom tools.

The tension surrounding the decision to provide standard tool integration protocols arises from the need to also support the integration of legacy tools. Legacy tools, by definition, do not follow the Armani environment integration standards. Recognizing this as an important issue, I designed multiple integration mechanisms for integrating external tools with the Armani environment. These integration mechanisms are encapsulated in the three types of external tool connectors described in section 5.2.2. Each of these connector types provide flexible intermediary connectors that can be adapted as needed to support the integration of legacy tools. The Acme connector provides a quick integration standard for other tools that support the Acme interchange standard [GWM97]. The tool-specific design rep API connector provides a custom environment developer with a place to embed the interface-bridging logic required to integrate the legacy tool with the Armani ADR.

By providing these three connector types and integration standards, the Armani environment supports the rapid development of custom environment-specific analysis tools without precluding the integration of legacy tools with the rest of the environment.

In addition to supporting the integration and development of multiple categories of external tools, these three types of external tool connectors also address the environment's need to progressively reveal the complexity of the Armani environment's implementation. It is possible for environment designers to create sophisticated custom environments by doing nothing more than loading design expertise captured with the Armani design language into the generic Armani environment. This customization process requires effectively no understanding of the implementation details of the Armani environment.

Once an environment developer decides to start integrating external tools, however, she needs to commit to learning something about the environment's implementation. The three external tool connector types reveal to the developer progressively more details about the environments implementation on an as-needed basis. The *Acme* connector, for example, requires almost no knowledge of the underlying environment implementation. It requires only the mastery of an extremely small API (primarily the methods readDesign(...) and translateToArmani(...)). Using the ADR's full Java API to write tools that will be loaded directly into the ADR's process requires a significantly greater understanding of the environment's implementation details. The full API can, however, generally be learned on an as-needed basis and specific tools will generally require the mastery of only a relatively small percentage of the full API. The depth of understanding of implementation details required to reuse a previously defined *tool-specific design rep API* connector falls somewhere between these two extremes. Implementing a new *tool-specific design rep API* connector, however, requires a relatively deep understanding of the Armani environment's implementation details.

A final issue that I faced in creating an appropriate set of mechanisms for integrating external tools with Armani was the need to distinguish between tools that can produce side-effects and those that can't. This distinction is important from the perspective of an architect using the Armani design environment because she needs to know whether making a request of the environment or one of its tools will simply provide a report on some aspect of the design, or whether it could actually change the design itself. A secondary driver for this issue is the ability to hook external tools into Armani's typechecking process to evaluate complex design analyses. All tools integrated with the typechecking process this way are fired implicitly during typechecking. Because the typechecker does not define the order in which these tools are run or how many times they will be run, it is critical that only side effect-free tools are invoked by the typechecker.

To address this issue, I provide Armani design language constructs that tool builders can use to distinguish between *operations* on design specifications, which can modify the design as a side effect, and design *analyses* that can not have any side effects. It is the responsibility of the tool writer to properly classify her tools using this taxonomy. Tools that perform operations on designs must be explicitly invoked by the user (or a proxy for the user). Side effect-free analysis tools, on the other hand, may be invoked either implicitly or explicitly.

5.3.5 Configurable user interfaces

Requirement: *The Armani design environment's user interface must support user-defined graphical depictions of designs and design elements.*

As section 5.2.3 described, the explicit decision to build Armani's user interfaces as loosely integrated external tools provides the environment with a great deal of coarse-grained user-interface customization. The fact that Armani supports three distinct user interfaces that all use the same core infrastructure effectively demonstrates that Armani satisfies this requirement with respect to coarse-grained configurability.

To support effective, lightweight, and incremental customization of architectural design environments, however, environment developers also need a lightweight set of mechanisms for customizing an existing user interface rather than completely replacing it. To address this need, the Visio-based Armani user interface provides the ability to make fine-grained modifications to the graphical depictions of designs and design elements.

Designing a user-interface infrastructure that provides this lightweight, fine-grained, incremental customization capability required me to address a number of important design issues. The fact that the Visio drawing package [Visio99] is implemented as a component that can be used as a complete front-end to other tools, such as the core Armani infrastructure, addressed many of the basic visual configuration issues. For example, Visio provides flexible customization capabilities for modifying the palette of visual entities that can be used in a drawing. These entities can be manipulated programmatically and by the end-user. As a result, the issues that I faced in making Visio an effective front-end for the Armani design environment proved to be more subtle than those related to actually drawing a design's graphical icons.

One of the primary issues I had to address in making the interface highly customizable was determining where the semantics for the design elements lived. Although not necessary for all external tools, the Visio interface stores negligible semantic information about the architectural entities that its diagrams model. All of the semantic structure for designs and design expertise is stored in Armani's core architecture design representation component. Armani systems are depicted in Visio as *drawings* and the system's components, connectors, ports, and roles are represented by the *shapes* that make up the drawing. A style's design vocabulary elements are stored on *palettes* from which an architect using the environment can select components and connectors to add to the system.

To keep a clean separation between the visual depiction of the design elements and the underlying semantics of those elements, the visual elements contain only a reference to the appropriate underlying semantic object that is stored in the environment's ADR. Likewise, the semantic objects stored in the ADR keep a reference to their appropriate visualization shapes. It is the responsibility of the *GUI factory* connector that binds Visio to the ADR to maintain these references appropriately.

The design vocabulary shapes stored on the editing palette use a different technique for interacting with the underlying ADR. The shapes stored on the palettes do not represent actual instances in a design; rather they are templates that store textual Armani expressions that describe how to create a new instances of a that component or connector. The shapes stored on the palette are associated with templates rather than types because it allows greater flexibility in mixing and matching types to create new components. It is, for example, easy to create a *transactional database* component template that will instantiate a component that satisfies both the *database* type and the *transactional* type without having to define a new type at the semantic level. When one of these shapes is copied from the palette to a drawing, the Armani template is sent to the ADR, where it is processed, an appropriate component or connector is added to the parent system, Visio adds an appropriate shape to the system's drawing, and the GUI Factory connector maintains the appropriate references between the visual shape and the underlying semantic object.

Overall, this approach has proven extremely flexible for performing fine-grained dynamic adaptations of design environments. The primary downside to this approach is that it requires the environment developer to do slightly more work to add some forms of user-interaction capability that require a tight integration between a design's underlying semantic representation and its visual depiction. Experience with the case studies described in Chapters 7 and 8, however, did not indicate that this limitation was particularly problematic.

Another important issue that this design raises is how to integrate other external tools that provide their own user interface with the Visio-based Armani environment and GUI. The basic problem is that if multiple tools expect to be the primary interaction mechanism with the underlying representation they can interfere with each other [GAO95]. Once again, the ability to provide a wide-range of integration techniques addresses this issue. In general, if other tools provide user interfaces that demand be the primary driver for interactions with Armani's underlying ADR, these tools are best integrated as tools running in completely different processes that communicate with the low-bandwidth Acme connector. In this way, the UI's can live in separate processes and they can each generally be adjusted to communicate only when appropriate. External tools, on the other hand, that provide user-interfaces that do not need to be the only one running in their process can generally be linked into the environment much more tightly using either of the other two connection techniques.

In addition to interoperability issues, integrating arbitrary tools in a single environment introduces significant issues regarding concurrent access to the shared design representation. To keep the scope of this research tractable for a single dissertation, I did not implement significant concurrency controls in the environment. Although an industrial strength design environment would need to support such capabilities (some of which are described in [GAO94]), the Armani prototype simply flags this as an issue that architects using the environment need to be aware of. In practice, the custom environments we used made relatively little use of implicit tool invocation. As a result, it was straightforward for architects using Armani to prevent concurrency problems by invoking only a single tool at a time.

5.4 Summary

The Armani design environment described in this chapter meets its fundamental requirements and demonstrates, by its existence, the second half of this thesis' claim that design expertise captured with the Armani design language can be used to incrementally customize software architecture design environments.

Chapter 6 Task Analysis

In the thesis statement I claim that it is possible to capture a significant and useful collection of software architecture design expertise with a conceptual framework of design rules and architectural styles, and further, that this captured design expertise can be used to incrementally customize software architecture design environments. The discussion of the Armani language and environment in chapters 4 and 5 demonstrate the feasibility of this claim – that it is possible to do so at all. In the next three chapters I describe a detailed task analysis and a set of experiments for evaluating the utility of the language, tools, and overall approach. To do so, I evaluate them for the following three properties:

- **Incrementality.** A software architect using Armani should be able to incrementally adapt his or her Armani-based tools to make use of available design expertise. Further, the incremental adaptation of an existing (possibly generic) environment should be quicker and easier than building a new environment from scratch.
- **Power.** The Armani language and environment should be able to capture useful, non-trivial software architecture design expertise.
- **Breadth.** The mechanisms provided by Armani should be capable of capturing a sufficiently broad range of software architecture design expertise. Coupled with the Armani infrastructure, this expertise should be capable of producing design environments for a wide variety of design domains and architectural styles.

I begin the validation of this thesis and the utility of Armani with a detailed analysis of the tasks required to capture design expertise, build, and incrementally adapt a customized software architecture design environment. The analysis evaluates both the traditional "ground-up" approach and the Armani approach and provides a comparison of the tasks and effort required to produce such an environment using each of the two methods. Section 6.1 of this chapter describes the results of this analysis. The purpose of this task analysis is to argue that, in theory, Armani fundamentally simplifies or eliminates many of the most time consuming and difficult tasks an environment designer must undertake with the traditional "ground-up" approach to creating a custom design environment.

To empirically validate the analysis results I conducted experiments in the form of two sets of case studies. In the first set of case studies I used the Armani system to construct a diverse collection of customized software architecture design environments. To demonstrate the breadth of expertise that Armani can capture, the styles selected for these case studies span the six "toplevel" categories of the taxonomy of architectural styles presented in [SC97]. Although not an exhaustive taxonomy, it captures most of the commonly used architectural styles. The structure and approach of the experiments also demonstrate the incremental nature of the language and tools. In these case studies I quantify the effort required to conceptualize and construct each of the case study environments. A detailed description of this experiment and its results are provided in Chapter 7.

The second set of case studies, described in Chapter 8, demonstrates that people other than the author can use Armani to produce useful and interesting software architecture design environments. In these case studies practicing software designers used the Armani system to model one or more architectural styles and used the Armani infrastructure to create custom software architecture design environment and tools for those styles. Although not as structured as the case studies described in Chapter 7, they serve to validate the utility of the overall Armani approach.

The experiments in both sets of case studies were designed only to evaluate Armani's effectiveness as a tool for rapidly creating custom software architecture design environments. They were not designed to evaluate the quality or effectiveness of the environments produced. Developing a set of experiments and metrics for objectively measuring amorphous concepts such as quality and effectiveness for such a broad variety of different environments simply proved too challenging to be reasonably completed in the scope of this dissertation.

6.1 Task analysis

This section describes an analysis of the tasks required to design and construct a customized software architecture design environment using both the "traditional approach" of building the environment ground-up, possibly using existing commercial off-the-shelf (COTS) components, and the Armani approach in which a configurable software architecture design environment is incrementally customized using a declarative architecture description language.

Each of the tasks described in this analysis is given a time estimate that includes best, average, and worst case times for each stage, along with a set of characteristics that help classify the types of projects that fall into each of these categories. The effort numbers assigned in these task analyses are rough estimates based on personal experience with teams building similar tools, informal estimates from other people who have built such tools, and the fact that comparable commercial and research tool development projects generally require multiple people working for one or more years.

These estimates can, of course, vary widely depending on the scope of the project and the power and polish desired in the environment produced. The estimates should, however, provide a rough idea of the likely time that each of the tasks will take under various conditions. Overall, the specific time estimates for the individual tasks are significantly less important than the specific tasks and subtasks required by each approach, and the units most likely used to measure these tasks (i.e., hours, days, months, or years).

An additional concern addressed by the task analysis is that the requirements of a prototype Armani environment are less demanding than the requirements of a commercial software architecture design tool such as ObjecTime [SGW94]. Commercial software packages must, for example, be built significantly more robust and feature-rich than the Armani environments created in this dissertation's case studies. To address this discrepancy in the task analysis I have attempted to include only those tasks required to build comparably robust and feature-rich tools. I have specifically excluded, for example, the lengthy testing processes required before releasing commercial tools.

6.1.1 The traditional approach

The current state-of-the-practice in constructing a software architecture design environment is to design and build the environment from the ground up. Throughout the rest of the task analysis, this approach will be referred to as *the traditional approach*. An environment designer using this traditional approach needs to perform at least four critical tasks – domain analysis, schema capture, implementation, and maintenance/evolution. This section describes these four fundamental tasks and some of the subtasks required to complete them. Each task is given a time estimate (in parentheses) for best, average, and worst case scenarios. Characteristics are given for each of the scenarios to illustrate and classify which of the scenarios a given project is likely to fall under.

The traditional approach involves (at least) the following tasks.

Task 1. Analyze the architectural design domain. In this stage the environment designer analyzes the design domain for which the design environment is to be built. The environment builder needs to ascertain the important aspects of the domain such as its standard design vocabulary, the design rules used in creating systems in this domain, and idioms and analyses that have been used successfully on similar systems built previously. In order to capture and express this domain analysis, the designer needs to either select an appropriate existing notation or create a new notation that directly reflects the concepts being captured.

Best case scenario (~1 week):

Well understood domain, environment analyst very familiar with domain, relatively homogenous design elements, existing design rules and analyses well documented, environment target-users enthusiastic about new tools. Easy integration with existing analysis and synthesis tools.

Average case scenario (~2 months):

Complex heterogeneous domain, but reasonably well understood. Many existing design rules and heuristics, but they are not well documented, analyst familiar with domain but not necessarily domain expert, environment target-users ambivalent towards new tools. Existing analysis and synthesis tools present moderate integration challenges.

Worst case scenario (~1 year or time to project cancellation):

Complex, poorly understood, heterogeneous domain. Few existing (effective) design rules and heuristics, almost none of which are documented. Environment designer/builder not very familiar with analysis domain. Environment target-users actively hostile towards new tools. Analysis and synthesis tools either non-existent, or available but running only on obsolete hardware (requiring significant porting or complex tool redesign).

Task 2. Find or create a language and schema for capturing and expressing the design vocabulary, rules, and analyses (collectively called the design expertise) to be used in the environment. This schema will be used both to document the architectural style that this environment is being defined to support, as well as provide a blueprint for implementing the environment and associated tools.

Best case scenario (~2 days):

Environment designer can use existing language and/or schema with which he is already an expert and the selected notation provides a good match to design expertise to be captured. The person performing the modeling is also the person who performed the domain analysis, or has at least been an integral part of the analysis

Average case scenario (~3 weeks):

Existing notation can be used to express design expertise schema, but match between notation and expertise to be expressed is tenuous and requires either significant effort to adapt the design expertise to the available notation, or the notation itself must be extended. Environment designer is familiar with the language/schema but not an expert at using it.

Worst case scenario (~6 months or time to project cancellation):

Unable or unwilling to use any existing schemas or design notations. Decide to invent a new notation from scratch in which the design schema will be described. Designer(s) inventing notation and schema are inexperienced at creating such notations and schema.

Task 3. Design, implement, test, and deploy the custom design environment. The cost and duration of this phase clearly has tremendous variability, depending on the power, capabilities, and polish demanded of the completed environment, along with the availability of appropriate reusable COTS components. This task basically includes all development costs and effort from the point where the desired design expertise and tools have been specified to the point where the environment is deployed and software architects are using it to design systems.

The specific steps required to complete this task vary greatly depending on the requirements of the environment, the quality of the work done in tasks one and two, the availability of appropriate reusable components, and the experience of the developers. In most cases, however, at least the following steps need to be taken to complete this task.

a) Define the requirements for the environment and tools based on the domain analysis and expertise capture completed in tasks one and two.
b) Specify the architecture and tool integration framework for the environment.

c) Figure out how to represent the captured architectural design expertise in the environment and its tools. Steps (b) and (c) will probably need to be iterated multiple times.
d) Implement and integrate the environment and tools. This may be primarily an exercise in finding and adapting a set of reusable components or, alternatively, this may involve building most of the environment's pieces from the ground up. As discussed in Chapter 5, this step will generally require the implementation of at least a design representation database, a graphical user interface, and a tool integration framework. Frequently, many other pieces will also have to be implemented to support the three primary elements.
e) Test and debug the environment. Iterate with step (d) as needed.
f) Document and deploy to users.

The precise duration and relative importance of each of these steps will vary from project to project, but they all require significant time, effort, and expertise to complete successfully.

Best case scenario (~1 month for 1 developer):

Small environment, COTS components available and used for most major parts of the system, COTS pieces integrate smoothly. Most analysis and synthesis tools are already built and need only be integrated with new environment, integration is smooth. Degree of polish and customization expected is relatively low. Expert developer who has built this kind of environment before and is reasonably familiar with all applicable COTS pieces.

Average case scenario (6 months for 2-3 developers):

Polished, robust environment required, some COTS components available, but significant portions of the environment have to be built by hand, some COTS components present significant integration difficulties. Significant portion of tools need to be built by hand or undergo difficult porting. Experienced developers, but not experts at building design environments.

Worst case scenario (> 1 year for 5-10 (or more) developers, or time to project cancellation):

Polished, robust environment required. Aggressive requirements. Few COTS components used (or even available). Many or most of the tools need to be built from scratch. "Not-invented-here" pervades development work, resulting in unnecessary "ground-up" development. Developers inexperienced and/or unfamiliar with the development domain. Poor analysis preceding development.

Task 4. Maintain, update, and modify the deployed design environment to capture and add additional design expertise, add additional tools and analysis capabilities, or evolve the schema to adapt to a changing understanding of the design domain.

Best case scenario (~1 day to 1 week for 1 developer):

Simple change request that doesn't disrupt any fundamental design decisions, original developer is maintaining code and providing updates, automated regression testing allows reasonable assurance that changes haven't introduced serious new bugs with minimal time required by developer.

Average case scenario (1 to 2 months for 1-2 developers):

Non-trivial change that requires rethinking of the original schema or design concepts. Schema and design expertise are hard-coded into design environment and distributed throughout the implementation so modifying them requires significant system archeology. Original developers are no longer maintaining the code. Little or no regression testing infrastructure to verify that changes won't disturb the rest of the system.

Worst case scenario (> or > > 2 months for 1 or more developers, or time to project cancellation):

Change requires complete rethinking of design expertise schema, design rules, and vocabulary, all of which is poorly hard-coded into the system implementation and distributed throughout the code. Original environment developers have left the organization. Original design is poorly documented. Change may require capabilities that a COTS component can't provide, necessitating expensive work-around or replacement.

Task 4 will be repeated as needed whenever updates or modifications to the system are required.

Task	**Approximate Time Required (in Engineer/Days, Weeks, Months, or Years)**		
	Best Case	**Average Case**	**Worst Case**
(1) Domain Analysis	1 week	2 months	1+ years
(2) Schema Capture	2 days	3 weeks	6+ months
(3) Design, implement, test and deploy environment	1 month	1 year	5+ years
Cumulative time to initial deployment	**~ 0.1 years (1.4 months)**	**~ 1.25 years**	**~ 6+ years (if not cancelled)**
(4) Time required for environment updates and modifications.	1-5 days	1-4 months	> or >> 2 months

Table 6.6.1: Breakdown of approximate engineer-years required to specify, design, build, and deploy a customized software architecture design environment using traditional ground-up approach.

6.1.2 The Armani approach

At a very abstract level, the four top-level tasks that an environment designer must undertake to build a customized software architecture design environment with Armani are the same as the top-level tasks required using the traditional approach. The specific sub-tasks required to complete each of these top-level tasks, however, can vary dramatically between the Armani approach and the traditional approach.

This section presents an analysis of the four top-level tasks required to create a custom Armani design environment along with a discussion of the key subtasks required for each of the top-level tasks. As in the previous section, each task is given a time estimate (in parentheses) for best, average, and worst case scenarios. Characteristics are given for each of the scenarios to help illustrate and classify which of the scenarios a given project is likely to fall under.

Task 1. Analyze the architectural design domain to capture design expertise – design vocabulary, constraints, heuristics, and analyses – for the target domain. As in the traditional approach, the environment builder needs to discover and/or articulate the important aspects of the domain such as its design vocabulary, the design rules used in creating systems in this domain, and idioms and analyses that have previously proven helpful. The times and tasks required to do this in Armani are comparable to those required with the traditional approach. Armani does not eliminate the need to deeply understand a design domain before constructing automated design tools for that domain. Armani can, however, provide some assistance with this task by providing a framework for doing the domain analysis and a collection of extensible, generic analyses.

Best case scenario (~1 week)

Well understood domain, environment analyst very familiar with domain, relatively homogenous design elements. Existing design rules and analyses well documented, environment target-users enthusiastic about new tools. Easy integration with existing analysis and synthesis tools. Analyst experienced with using Armani.

Average case scenario (~1 month):

Complex heterogeneous domain, but reasonably well understood. Many existing design rules and heuristics, but they are not well documented, analyst familiar with domain but not necessarily domain expert, analyst familiar with Armani. Environment target-users ambivalent towards new tools. Existing analysis and synthesis tools present moderate integration challenges.

Worst case scenario (~1 year or time to project cancellation):

Complex, poorly understood, heterogeneous domain. Few existing (effective) design rules and heuristics, almost none of which are documented. Environment designer/builder not very familiar with analysis domain. Environment target-users actively hostile towards new tools. Analysis and synthesis tools either non-existent, or available but running only on obsolete hardware (requiring significant porting or complex tool redesign). Novice analyst.

Task 2. Articulate the captured design expertise from the domain analysis in the Armani design language. This task can generally be done much faster with Armani than with the traditional approach because (1) much of the schema required for capturing architectural design expertise is already encoded in the Armani language, eliminating the traditional approach's subtask of creating such a schema, (2) the Armani language is explicitly designed for capturing this type of architectural design expertise, and (3) the domain analysis was performed with this language already selected as a target, reducing the likelihood of mismatch (same as best case in traditional approach).

Best case scenario (~0.5 days):

Small domain, experienced Armani modeler, all vocabulary and design rules expressed directly in Armani, no mismatches between expertise to express and the Armani language used to express it.

Average case scenario (~2-5 days):

More complex domain, relatively few design rules and analyses need to be written as external tools and linked into the environment. Domain modeler is familiar with Armani, rules. Minimal mismatch between expertise to express and Armani language/schema.

Worst case scenario (~2-3+ weeks or time to project cancellation):

Large, heterogeneous domain, many design rules and analyses need to be written as external tools and linked into the environment., novice modeler, significant mismatch between Armani language and expertise that needs to be expressed.

Task 3. Design, implement, test, and deploy the custom design environment. The difficulty and time required to complete this task can vary widely depending on the sophistication and polish expected of the final environment. In general, however, performing this task with the Armani system provides a dramatic savings in time, effort, and cost over the traditional approach because the infrastructure needed for such an environment does not need to be built from the ground up. Further, a significant amount of the customization is performed by simply loading the design expertise captured in task 2 directly into the generic environment infrastructure. The specific sub-tasks required to complete this task with the Armani system are:

g) Load the design vocabulary, design rules, heuristics, and analyses captured in tasks 1 and 2 into the generic Armani design environment. This step makes all of these design elements available for use as design building blocks in the custom environment.
h) Configure the user interface to map icons to their underlying semantic representations. Optionally, customized editing dialog boxes can be created for editing specific types of vocabulary elements.
i) Write and/or link-in any external design tools (analysis or synthesis) to be used with the environment.
j) Test the environment. Many consistency checks are provided by the Armani infrastructure to minimize the difficulty of testing the environment.
k) Deploy to users

As with the traditional approach's third step, the time, effort, and expertise required to complete this step varies widely depending on the richness of the tools that need to be written and the polish and customization desired in the user interface.

Best case scenario (~1 day for 1 developer):

Small, homogeneous design domain, small number of analyses defined directly in Armani or performed by existing tools that are easily linked in, experienced Armani developers, little customization required in the user interface.

Average case scenario (1-2 weeks for 1 developer):

Heterogeneous style, non-trivial customization and polish required in the user interface. Some design tools need to be written from scratch, others need to be linked in, developers have familiarity with Armani but are not experts.

Worst case scenario (2-4 months for 1 developers, or time to project cancellation):

Large, heterogeneous style, heavily customized and sophisticated user interface, many tools need to be written from scratch. Novice Armani developer(s), poor initial domain analysis.

Task 4. Maintain, update, and modify the deployed design environment to capture and add additional design expertise, add tooling and analysis capabilities, or evolve the schema to adapt to a changing understanding of the design domain. Along with step 3, this is the step that provides a huge savings over the traditional approach. In Armani, many modifications and customizations can be performed by the end-user as his or her understanding of the domain and design issues evolves. Specifically, additional design expertise in the form of vocabulary, design rules, heuristics, and analyses can be added to the environment by the end-user, on-the-fly, directly through the design environment itself. Further, this additional design expertise can be packaged and provided to other environment users if desired. Tools can also be dynamically and incrementally linked into the environment. Thus the environment's original developer(s) can be completely removed from the loop for many required modifications to the environment, dramatically reducing the time to turn a proposed change into an environment capability.

Best case scenario (~15 minutes for end-user):

User wants to add or modify a design rule, vocabulary element, or analysis. The change can be made by simply editing the appropriate schema element, after which Armani reconfigures itself with the new or updated design expertise.

Average case scenario (1 hour to 1 day for end-user):

User wants to make a significant modification to the domain schema, requiring changes to or additions of multiple design vocabulary elements, rules, analyses, or linked in tools. Perhaps a relatively straightforward design tool needs to be linked in or written.

Worst case scenario (1 or more weeks for 1 or more environment developers):

Change requires complete rethinking of design expertise schema, design rules, and vocabulary, and/or significant modification to the user interface and linked-in design tools. Original environment developers may have left the organization. Original design is poorly documented. Change may require capabilities that a COTS component can't provide, necessitating expensive work-around or replacement.

This final step will be generally be repeated many times throughout a design environment's lifecycle as updates or modifications to the system are required.

Table 6.2 summarizes the results of the Armani task analysis. Table 6.3 compares the results of the Armani task analysis to the results of the traditional approach's task analysis.

Task	Approximate Time Required (in Engineer/Days, Weeks, Months, or Years)		
	Best Case	**Average Case**	**Worst Case**
(1) Domain Analysis	1 week	1 month	1+ years
(2) Schema Capture	0.5 days	2-5 days	2-3+ weeks
(3) Design, implement, test and deploy environment	1 day	1-2 weeks	2-4 months
Cumulative time to initial deployment	**˜8 days**	**˜ 1.5 months**	**˜1.25 years (if not cancelled)**
(4) Time required for environment updates and modifications.	15 minutes	1 - 8 hours	1 or more weeks

Table 6.2: Breakdown of approximate engineer-years required to specify, design, build, and deploy a customized software architecture design environment using the Armani approach.

Task	Approximate Time Required (in Engineer/Days, Weeks, Months, or Years)					
	Best Case		**Average Case**		**Worst Case**	
	Traditional	Armani	Traditional	Armani	Traditional	Armani
(1) Domain Analysis	1 week	1 week	2 months	1 month	1+ years	1+ years
(2) Schema Capture	2 days	0.5 days	3 weeks	2-5 days	6+ months	2-3+ weeks
(3) Design, implement, test and deploy environment	1 month	1 day	1 year	1-2 weeks	5+ years	2-4 months
Cumulative time to initial deployment	**˜1.4 months**	**˜8 days**	**˜ 1.25 years**	**˜ 1.5 months**	**˜6+ years (if not cancelled)**	**˜1.25 years (if not cancelled)**
(4) Time required for environment updates and modifications.	1-5 days	15 minutes	1-4 months	1 - 8 hours	> or >> 2 months	1 or more weeks

Table 6.3: Comparison of approximate engineering time required to specify, design, build, and deploy a customized software architecture design environment using the traditional ground-up approach vs. the Armani approach.

6.2 Summary of task analysis and comparison of results

Assuming that the case studies presented in the following chapters uphold these task estimates, this analysis strongly argues that Armani eliminates many of the difficult, time consuming tasks required by the traditional ground-up approach to developing custom software architecture design environments. Although the specific numbers used to estimate the duration of each of these tasks are only rough estimates, the exact time estimated for each task is not critical. The qualitative comparison between the two approaches is sufficient to support the main points of the argument.

The primary insight to be gained from this analysis comes from the enumeration of the specific subtasks required to complete the top-level tasks using each of the two approaches. This detailed look at the specific tasks illustrates that Armani provides the greatest leverage in the implementation stage of environment creation. Armani also qualitatively changes the way in which expertise is captured (in the second stage), and the way in which an environment can be updated. In each of these stages, Armani not only reduces the number and difficulty of the tasks required to complete the stage, it changes *who* can complete the tasks. Specifically, Armani empowers its end-users to build and customize their design environments in a way that is very rarely found in design environments built with traditional approaches.

There are at least two implications that follow from this analysis. First, by either eliminating or drastically reducing many of the most time consuming tasks required by the traditional approach (defining a schema for capturing design expertise, designing and implementing the environment and tools), Armani allows environment designers to create and evolve their design environments much more quickly than they can by building from the ground up.

Second, Armani moves the bottleneck for producing an environment from implementation (the dominant timesink using the traditional approach), to domain analysis (the dominant timesink using the Armani approach). This is, in itself, an extremely valuable benefit. Armani allows environment developers to focus their effort and skills on figuring out *what* domain expertise and analytical capabilities they want the tools to support, rather than how they will implement support for the expertise that they capture.. As a result, they can spend more of their time figuring out what their tools should do and less of their time designing and implementing the tools.

Chapter 7
Structured Case Studies

The previous chapter argued that, at a conceptual level, creating a custom software architecture design environment using Armani eliminates many of the difficult and time consuming tasks such a project traditionally requires. This chapter describes a series of case studies undertaken to determine whether the assumptions made in the conceptual analysis of the previous chapter could be realized in practice.

As with the task analysis, these case studies seek first to demonstrate that the Armani approach can be used to capture design expertise at all, and second, to demonstrate that expertise so captured can be used to incrementally configure and customize the Armani design environment. Demonstrating these two capabilities provides a basic validation of the thesis that it is possible to capture a significant and useful collection of software architecture design expertise with a conceptual framework of design rules and architectural styles and, further, that this captured design expertise can be used to incrementally customize software architecture design environments.

This thesis, however, claims only that it is *possible* to incrementally capture design expertise and configure custom design environments with the Armani approach. Arguing for the *value* of the overall Armani approach also requires a demonstration that Armani is capable of capturing a broad variety of design expertise and that this expertise is powerful, non-trivial, and captured in such a way that it's useful in practice.

To make this argument I used the Armani system to construct custom design environments for eight different architectural styles. The selected styles and the capabilities of the environments produced support the claim that the technique provides sufficient breadth, power, and incrementality. These case studies both argue for the validity of the thesis and demonstrate the utility of the Armani approach.

7.1 Experimental structure

The basic structure used for this experiment was to perform a series of case studies in which I constructed custom software architecture design environments for eight different architectural styles (referred to as *test styles*). In each case study, I measured the time required to specify the style's design expertise and the time required to configure a design environment with that expertise (tasks two and three from the previous chapter's task analysis). In addition to the time required, I measured the number of vocabulary types defined, the number of design rules defined, total lines of code written or modified, and the number of visual shapes defined. The number of entities defined serves as a proxy for the size of the overall style specification effort.

Through careful selection of the test styles these experiments demonstrate the breadth, power, and incrementality of the Armani approach. The following criteria were used to select the test styles.

Availability of a published style description. Each of the test styles specified in these case studies is based on one or more published descriptions of that style. These published style descriptions provide informal specifications of the styles' design expertise (vocabulary, design rules, and analyses). The rigor with which the publications define the styles, as well as the match between the Armani conceptual framework and the format used for expressing the style's expertise varied widely from style description to style description. All styles selected had to have a rich enough informal description of the style that the expertise it contained could be used as the basis for creating a compelling design environment.

Using published sources as the basis for the styles provide three key benefits. First, starting with published sources provides a degree of external validation of the utility of the styles themselves. The design expertise embodied by these architectural styles has proven useful enough to somebody that it was worth capturing and publishing.

Second, using published sources as a foundation leverages other people's expertise. By attempting to capture expertise that had been previously published, I did not need to become a expert in each of the domains for which I built a case study environment. I needed only to fully understand the published specification. Likewise, I did not need to find an expert in each of these domains who was willing to learn the Armani tool and participate in this experiment. Finding multiple such experts was impractical given the budget, time, and scope constraints of the dissertation research.

Third, by using published descriptions of the target styles as a domain analysis, the time and effort required for the domain analysis task can be factored out of the case study experiments. As the previous chapter's task analysis indicates, Armani provides minimal leverage for completing the domain analysis step. Armani does not mitigate the environment designer's need to deeply understand the domain in which he or she is working. In these experiments the domain analysis phase consisted of finding appropriate published references, reading and understanding them deeply, and, if needed, discussing the published style specification with local experts. As a result, the metrics captured in these experiments reflect the time and effort required to create a custom design environment after the domain analysis has been completed.

Breadth. Selecting an appropriately broad variety of architectural styles for the case studies is critical for arguing the generality of the Armani approach. The discipline of software architecture is fairly young. As a result, the field as a whole has not yet established a complete catalog of all of the interesting or commonly used architectural styles. The Shaw/Clements architectural style taxonomy [SC97], however, outlines an initial proposal for what such a catalog might look like. I used this taxonomy as a basis for selecting test styles because it provides a clean and broad overview of many frequently used architectural styles. The taxonomy defines six "toplevel" categories of architectural styles – *data flow*, *call-and-return*, *interacting processes*, *data-centered repositories*, *hierarchical*, and *data sharing*. The authors augment this toplevel classification with more detailed examples of architectural styles that fit into each of these categories. Shaw and Clements claim that, though not exhaustive, these toplevel categories cover a significant portion of the commonly used architectural styles.

To demonstrate the breadth of architectural design expertise that Armani is capable of capturing then, at least one test style was chosen from each of the six toplevel architectural categories described in the taxonomy. In evaluating potential test styles from each of these categories I also made an effort to insure that, when evaluated as a whole, the set of architectural styles I selected from each of the toplevel categories addressed a broad range of architectural issues and captured a broad range of architectural attributes. As a result, the eight case studies described in this chapter demonstrate that Armani is capable of capturing and leveraging a wide variety of architectural design expertise.

Power. The power of the environments produced in these case studies varied significantly. For inclusion in these experiments, however, a published style description had to describe, or at least point the way towards, some compelling analytical or design guidance capabilities. To show the depth of the analytical power that can be included in an Armani environment, two of the styles (the Client-Server styles) were augmented with significant external analysis tools. This experimental approach demonstrates that an Armani environment can quickly and easily capture many forms of design expertise. It also demonstrates that significant, complex, modeling and analysis tools can be integrated with a customized Armani environment to capture and exploit design expertise that is not readily captured directly in the Armani design language.

Incrementality. These experiments demonstrate the incrementality of the Armani approach on two levels. At a coarse level, the basic process of creating a custom environment is simply the incremental adaptation of a generic Armani environment by loading styles and other captured design expertise into the environment. In this way, all of the test styles selected demonstrate the incremental nature of the Armani approach. At a more fine grained level, two of the eight test styles were selected because they are natural extensions and specializations of other, more generic, test styles. These style specializations were created by making incremental adaptations to existing styles so that the initial style's environment could capture additional design expertise.

Using these criteria, I selected eight architectural styles to use as test cases. For each of these styles I built a custom Armani-based software architecture design environment and tracked the time, effort, and tasks required to create it. Specifically, I measured the time spent creating the style and environment (broken down into time to capture abstract design expertise and time to customize the visualization and tooling of the environment), the number of new vocabulary entities defined, the number of design rules defined, total lines of code written, and the number of custom shapes defined for use in the customized graphical user interface.

At first glance, the fact that the creator of the Armani tool is also the person performing the case studies may cast doubt on the validity of the studies. This was, however, a reasonable approach for this phase of the validation. Having one person who was already expert with the Armani system perform all of the initial case studies factors issues of learning curves and differences in ability between individual environment developers out of the experimental results.

The case studies described in this chapter are only part of the overall thesis validation. Chapter 8 describes a set of "external" case studies in which people and organizations not associated with the Armani project used Armani to create their own software design tools and environments. The external case studies address the issue of whether people other than myself can effectively use the Armani tools and techniques.

These experiments are primarily intended to demonstrate that the overall Armani approach is feasible. A secondary goal is to demonstrate that the environments produced can capture useful design expertise for a broad variety of architectural styles. Thus the variable being tested in these experiments is the range and depth of design expertise that can be captured and exploited with Armani, rather than the skill of the environment developer using the tools.

7.2 Discussion of case studies

This section presents a detailed discussion of eight case studies that I conducted. I present the case studies according to how they fit into the six toplevel style categories defined in [SC97]. I defined at least one Armani style and environment for each of these six categories. For two of the categories I created a base style and then extended the base style to support additional capabilities with a more specific style.

Each case study begins with an overview of the style it captures – its interesting aspects, the issues facing designers working in that style, its key design vocabulary and design rules, and a citation of the sources from which the style definition was extracted. This is followed by a discussion of interesting issues raised in the process of customizing the Armani system to work with the style in question. Each of the case studies includes a figure with a screen-dump from the custom environment and an overview of some of the important statistics for the case study and environment.

7.2.1 Case study 1: a *data flow* style

For the first structured case study, I created a custom environment for the *batch sequential* architectural style. This style, which is popular for mainframe-based data processing applications, captures a common approach to building software systems that repeatedly perform a series of operations over large sequences of data records. I based this style specification on the discussion of "Sequential Processing Program Design" in Larry Best's book *Application Architecture: Modern Large-Scale Information Processing* [Be90]. Although Best's presentation of this style is somewhat informal, I found it straightforward to add the formality required by Armani to his style specification while retaining the style's basic concepts, vocabulary, and design rules.

Systemwide issues

Conceptually, systems built in the batch sequential style consist of a set of *data processing operation* components that perform one or more operations on a series of data records, a set of *input source* and *output sink* components that handle system input and output, and a set of *data stream* connectors that define and manage the flow of data through the system.

This structural breakdown provides a clean way for the architect to divide the functional processing needs of a batch sequential system amongst the data processing operation components. It also clearly identifies the dataflow paths through the system. These are two of the important system-wide issues that an architect working in the batch-sequential style must manage. Further style-wide design issues include: processing performance (both total system throughput and processing latency), design modifiability (how difficult is it to add, remove, or change the functional capabilities of the system, or replace an existing component with a component with different non-functional properties), data format compatibility (between components), connector interaction protocols, and error handling and recovery.

The design information that architects and tools need to reason about these issues is stored in the properties and design rules of the style's vocabulary elements - component, connector, port, and role types. With this approach, design details are stored locally with the design elements that make up a system. The emergent properties of the system are captured by the design rules and analyses that evaluate the properties of the system's constituent elements to address the system-wide issues just outlined.

Component issues

One of the early tasks an architect must perform in designing systems in this style is to define a functional decomposition of the processing to be done. This decomposition divides the overall system processing into a sequence of discrete functions. These functions are then mapped to a set of *data processing operation* components that execute the requested functions. Architects can specify and evaluate individual data processing operation components for: functionality provided, persistence of state between records, processing latency and throughput, error handling policies, and data validation policies. All of these aspects of individual components are represented with property annotations on the data processing operation component type specification.

Connector issues

After the basic functional decomposition has been performed and the functional responsibilities have been assigned to components, it is necessary to connect those components by feeding the output of each data processing operation into the input of one or more subsequent processing operations. Alternatively, if the processing has been completed then the output needs to be written to the appropriate storage or output device.

The basic connector used in this style is the *data stream*. All data stream connectors insure that data is delivered from an upstream component to a downstream component unmodified and in the order that it was received from the upstream component. Data availability is the

Batch Sequential Style
Category: Data Flow

Semantics Statistics
Primary component types:
- Data Processing Op
- Data Input Source
- Data Output Sink ...

Primary connector types:
- Data Stream
- Fan Out Data Stream
- Rendevous Data Stream ...

Sample design rule:
- All connectors must be Data Streams

Total types defined: 29
Design analyses defined: 3
Time to define: 3.5 hours
Lines of Armani code: 83

Environment Statistics
Total shapes defined: 14
Customization time: 5¾ hrs.

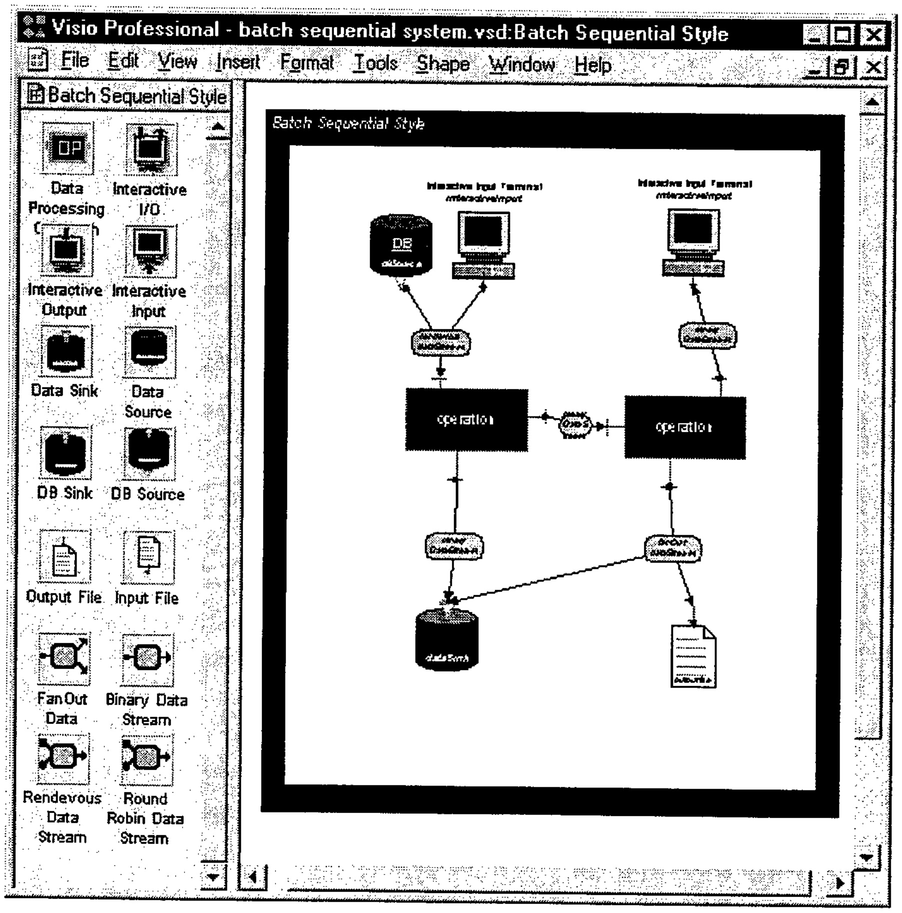


Figure 7.1: Overview of the Batch Sequential Style and environment

primary control signal used by data stream connectors to signal a transfer of control between components.

Data stream connectors need to be able to handle large volumes of data flowing between components that can have highly variable processing rates. As a result, they must be able to buffer large amounts of data and adapt to differences in processing speed between the different components. Three further issues that connectors need to address in this style are (1) whether the components attached to the connector will be pushing or pulling their data, (2) how the connectors avoid or react to buffer overflows, and (3) what policy is used for aggregating or replicating data streams with multiple input sources or output sinks. Numerous subtypes of the data stream connector type, which address these issues in various ways, are provided as basic system building blocks.

All three of these issues are captured in the properties of the connector types defined by the style. The style defines specific property types for describing push-pull behavior, buffer-overflow behavior, and data-replication behavior. It also defines a set of design rules that ensure that connectors and the components to which they are attached agree on these protocols and behaviors. As a result, architects working in this style can explore their options with respect to these issues by adjusting the values of these properties on their connector and component instances and testing whether their proposals satisfy the style's design rules.

Discussion and evaluation

As Figure 7.1 indicates, the process of defining this architectural style and customizing the generic Armani environment to support it was quick and straightforward, requiring only 9.25 hours of total development time. As mentioned earlier, this time measurement does not include the domain analysis effort (primarily reading and understanding Best's book) that preceded the style specification and environment development.

As the previous sections describe, I was able to capture and represent a wide variety of design options for this style using only Armani's property, design rule, and type constructs. The ability to define enumerated property types proved very useful because many of the properties that I needed to capture were easily represented with a discrete and fixed set of possible values. Furthermore, it was straightforward to identify which combinations of these values across components and connectors were valid and which were invalid, and to write corresponding design rules to insure valid combinations. For example, enumerated properties capture whether data is pushed or pulled across port/role pairs and whether that pushing or pulling is active or passive. Likewise, enumerated properties are used to define the three basic ways that a connector can handle buffer overflow – by blocking, dropping received records, or crashing.

Another finding from this case study was that the Armani design language proved remarkably effective at capturing the non-functional properties of an interesting set of connectors. The connectors in this style drive control flow with data flow, which is the opposite of traditional procedural connections. Further, they provide the ability to put multiple producers or multiple consumers on a single connection. Although I did not provide a formal behavioral protocol specification for the connectors, I was able to capture a wide variety of nonfunctional properties related to the connectors. I was also able to capture and enforce a few key aspects of the connector's communication protocol using enumerated properties and design rules. These aspects included ensuring appropriate push/pull and active/passive combinations between ports and the roles to which they were attached, buffer overflow policy, buffer sharing policy, transferred record format, and the model for handling multiple source roles that feed into a single sink (round-robin, rendezvous, or opportunistic).

The most encouraging result from this case study is that it proved straightforward to represent all of these things directly in the Armani design language. I did not need to use external analysis tools, language extensions, nor other embedded languages within my Armani style specification. Nor did I run into any significant limitations with the design language itself. This finding is encouraging because designers working in this style can address non-trivial, useful, and broad design issues.

7.2.2 Case study 2: a *hierarchical* style

In the second structured case study, I created a style that captures the popular architectural abstraction of layers. This style, imaginatively called the *layered style*, is an example of a

Layered Style

Category: Hierarchical

Semantics Statistics

Primary component type:
- Layer Component

Primary connector type:
- Inter-layer Request

Sample design rules:
- synchronicity and protocols of attached ports and roles must match
- requests can only be sent to "lower" level layers.

Total types defined: 11

Style-wide design rules defined: 3

Time to define: 2.25 hours

Lines of Armani code: 110

Environment Statistics

Total shapes defined: 6

Customization time: 5 hours

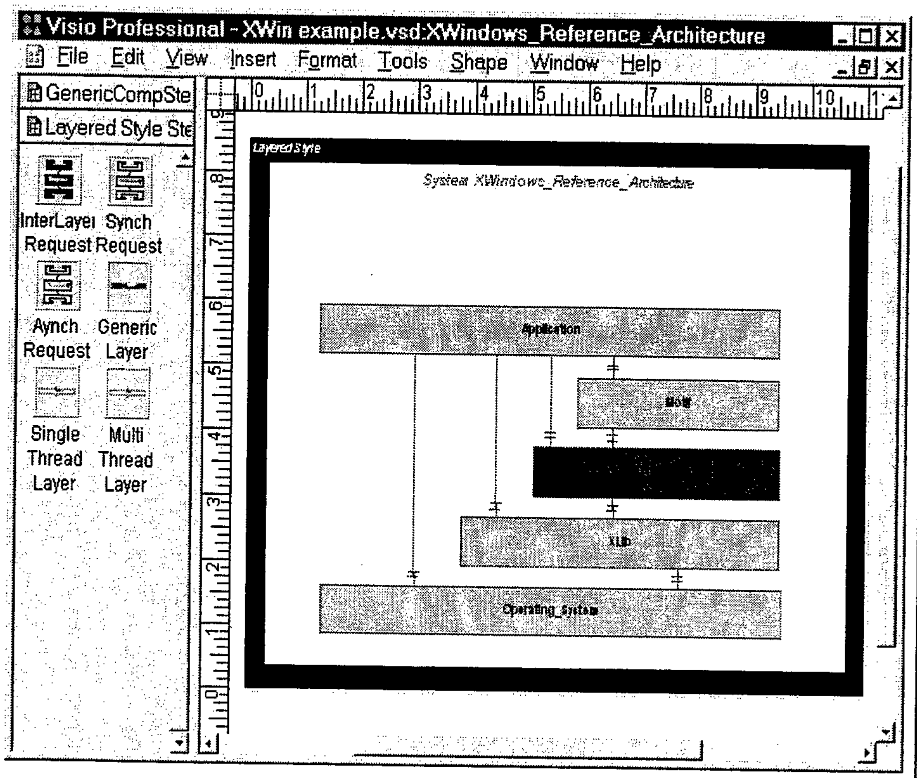


Figure 7.2: Overview of the Layered style and environment

hierarchical style in the Shaw/Clements taxonomy. I based this style on the high-level X-Windows architectural specification published in [HFB94]. Although it is used quite effectively in the design of the X-Windows system, the layer abstraction is also very general and broadly applicable. To take advantage of the style's generality I teased the fundamental "layerness" out of the X-Windows specification and used it to create a generic layered style.

Style overview

As Figure 7.2 details, the layered style is simple and straightforward. It has one primary component type – the *layer* – and one primary connector type – the *inter-layer request*. All layer components have properties that indicate their level in the layer hierarchy, whether they can handle asynchronous requests, and whether they are multi-threaded. Layers also have a set of ports that receive requests from components higher in the layer hierarchy and a set of ports that send requests to components lower in the layer hierarchy. Inter-layer requests are binary connectors that send requests from a higher-level layer to a lower-level layer and return responses in the other direction. They can be synchronous or asynchronous, and they can support many different request protocols. The universe of specific protocols accepted by the connectors can be extended as needed by the style's users.

The layered style defines a relatively small number of design rules that serve three primary purposes. First, they insure that each layer component represents an abstraction boundary in the overall design. A layer component can encapsulate multiple subsystems that perform the actual functionality provided by the layer. The functionality provided by each of these subsystems can only be accessed outside of the layer, however, through the layer's port interfaces. Other layers are unable to interact directly with the subsystems encapsulated within a layer. These design rules encourage architects to package their capabilities in common groups whose implementations can be readily replaced or modified without severely disrupting the overall system's architecture. Likewise, if a designer simply wants to bundle a group of components that form a logical layer in the design, he can do so by bundling the components in a representation of the layer component and making sure that the appropriate bindings are made between the layer's ports and the representation's ports.

The second purpose of the design rules is to constrain the topology of layered systems to insure that requests may only be sent from layers at a higher level of the hierarchy to layers at a lower level of the hierarchy. This rule insures that all requests will eventually "bottom-out" and be handled. It also eliminates circular dependencies between layers. The third purpose of the design rules is to make sure that the interacting layers agree on their protocol of interaction. This includes defining standards for message, request, and response structure and insuring agreement on synchronicity and handshaking issues.

Discussion and evaluation

The specific style and custom environment that I created in this case study is useful for broadly structuring and decomposing software systems. Specifically, it provides a way to carefully delineate where system capabilities are provided, how those capabilities can be accessed by the rest of the system, and how they can be composed. Due to its generic nature, though, the layered style does not, on its own, provide architects with a tremendous amount of design leverage.

One of the key observations from this case study is that the real power of the layered style emerges when it is combined with other styles that need to support a layering abstraction. Many systems can be viewed as being created in both a layered style *and* another style. It is useful, for example, to think of an n-tier client-server system as being both a data-repository centric system and a layered system with each tier represented by a logical layer. Armani's support for subtyping of style specifications makes this type of composition both feasible and straightforward. The layered client-server style just described, for example, could be defined in Armani by simply declaring a new style called *layered-client-server* that is a subtype of both the *layered* style and the *client-server* style. The new style declaration can be defined with the single line:

```
Style layered-client-server extends layered, client-server;
```

which would result in the creation of a new style that includes the types, properties, design rules, and default structure of both the layered and client-server super-styles. By making the layered style in this case study very generic I was able to both provide structure to systems that are primarily built as layers and also provide an orthogonal organizing principle for systems built in other styles that make use of the layer concept but do not rely on it as their fundamental building block.

Attempting to define the layered style so that it would be useful both as a powerful stand-alone style as well as a "supplementary" style that could be combined with other styles introduces what appears to be a fundamental tension. Creating a highly composable style requires the style designer to introduce only a minimal set of types, properties, and design rules, each of which are geared specifically towards capturing the design aspects that form the basis for the composable style. In general, the more minimal and focused the style specification, the easier it is to compose it with other styles. At the same time, however, the more rich, detailed, and fully specified the style, the more leverage that style (and its associated environment) can provide to the architects who use it.

In order to explore the issues surrounding this design tension, I created a couple of variations on the semantic specification of the layered style. For the first variation I defined a relatively rich and full-featured style specification for the layered style and built a custom Armani environment to support the style. In this version, the only vocabulary constructs available at the top level of system design are layer components and inter-layer request connectors. To further constrain the design space, the interaction protocols supported by the layers and inter-layer requests in this style are explicitly enumerated and fully specified. Architects are free to add representations to the layer components that define subsystems built in other styles, but they can not mix arbitrary types of components into the top-level system abstraction.

This first version of the layered style provided a constrained design space and a sufficient selection of vocabulary to work as the basis for a layered style Armani design environment. This version was not, however, particularly effective as a generic representation of "layerness" that could be combined and composed with other styles. To address this limitation, I created a second variation on the layered style that was significantly more abstract and less constrained than the original layered style. In this version of the style, I defined very abstract and generic design vocabulary types for layer components and inter-layer connectors, along with a much less rigorous set of design rules to govern valid layer topologies. These vocabulary elements had significantly less semantic structure than their corresponding types in the original style. In return for giving up their semantic richness, however, they became much simpler and more readily composed with the component and connector types of other styles through Armani's subtyping constructs.

Starting from the original layered style specification, I was able to experiment with a number of design alternatives and create the revised and more generic layered style with only about one hour of effort. The Armani language's modularity and incrementality proved instrumental in performing this style modification so quickly. I did not create a new environment for the second variation on the layered style because, on its own, the revised style provided significantly less leverage than the original layered style. Rather than form the basis for its own custom design environment, the second layered style variation is designed to be used as a supertype and "mixin" for other styles that need to capture some aspects of "layerness" in their systems.

Overall, this case study provided three interesting results. First, it demonstrated that Armani can be used effectively to create design environments for a basic architectural style with broad applicability. Second, it illustrated the tradeoff that style designers need to consider between creating powerful styles and environments for basic styles and creating very abstract basic styles that can be readily composed with other styles. Finally, it provided useful insight into how a generic set of design principles can be abstracted from a very specific and highly constrained style definition.

7.2.3 Case study 3: an *interacting processes* style

In the third structured case study, I created an Armani style and custom environment for the C2 architectural style. The C2 style, described in detail in [Tay+96], is an example of an interacting processes style in the Shaw/Clements taxonomy. It provides a framework for building systems in which loosely coupled processes communicate by sending requests and notifications to other components through message bus connectors. [Tay+96] provides the following overview of the C2 style:

> "[The C2 style] is designed to support the particular needs of applications that have a graphical user interface aspect, but the style clearly has the potential for supporting other types of applications. ... [The C2 style] supports a paradigm in which UI components, such as dialogs, structured graphics models of various levels of abstraction, and constraint managers, can more readily be reused. A variety of other goals are potentially supported as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running concurrently in a distributed, heterogeneous environment without shared address spaces, architectures may be changed at runtime, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active and described in different formalisms, and multiple media types may be involved...
>
> The new style can be informally summarized as a network of concurrent components hooked together by message routing devices. Central to the architectural style is a principle of limited visibility: a component within the hierarchy can only be aware of components 'above' it and completely unaware of components which reside 'beneath' it."

As figure 7.3 indicates, I captured the C2 style in Armani with two key component types (*C2 Component* and *C2 GUI Component*), three connector types that define different kinds of message buses, twenty-eight supporting property, element, port and role types, and seven design analyses. After capturing the semantics of the style with the Armani design language, I customized the Armani environment to support the design visualizations commonly used for specifying systems in the C2 style. Capturing the key semantic properties of the C2 style in an Armani style specification and customizing the Armani environment to support the style proved to be a quick and straightforward task, requiring a total of only eight hours of effort.

Discussion and evaluation

The C2 style is particularly interesting as an Armani case study because the C2 style developers have also constructed a comprehensive environment for designing and implementing C2 systems. This environment, called *Argo* (pictured in figure 7.4) [RHR98], provides greater functionality than the comparable Armani environment. Specifically, it provides code generation capabilities, a software development process management tool, and a runtime environment. Armani is not designed to support these capabilities, so I did not include them in the Armani-based C2 environment. Because both Armani and Argo are capable of using the Acme interchange language [GWM97], however, it would be relatively straightforward to integrate Argo's other tools with Armani. I did not actually do such an integration for this case study but section 8.2.4 describes an external case study in which a member of the Argo development team integrated Armani with an Argo-based tool he developed.

C2 Style

Category:
Interacting Processes

Semantics Statistics

Primary component types:
- C2 Component
- C2 GUI Component ...

Primary connector types:
- Message Bus
- Notification Filtering Bus
- Prioritized Filtering Bus ...

Sample design rule:
- All connectors must be message or filtering buses

Total types defined: 33
Design analyses defined: 7
Time to define: 4 hours
Lines of Armani code: 110

Environment Statistics

Total shapes defined: 5
Customization time: 4 hours

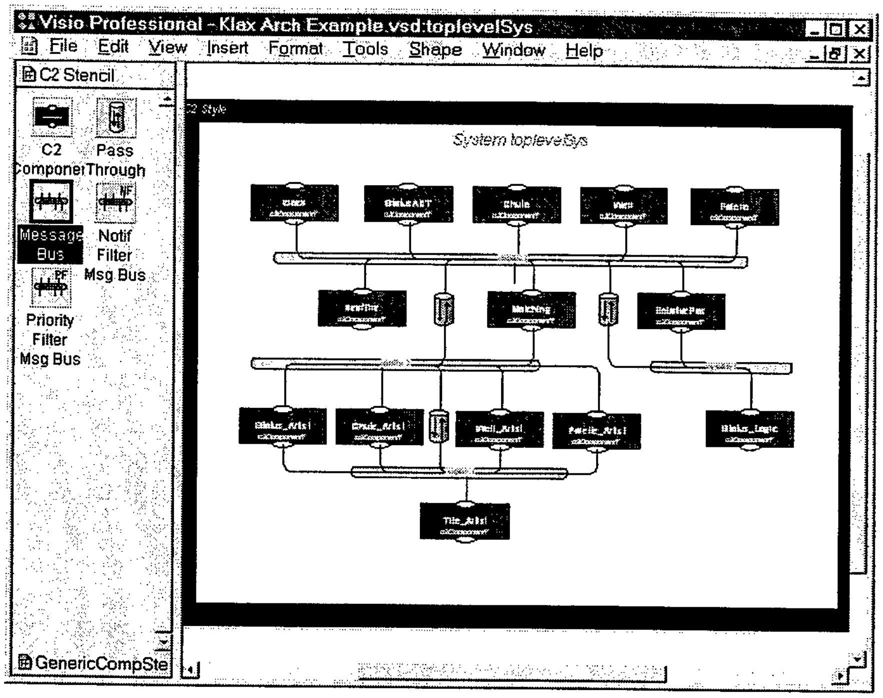


Figure 7.3: Overview of the C2 style and environment

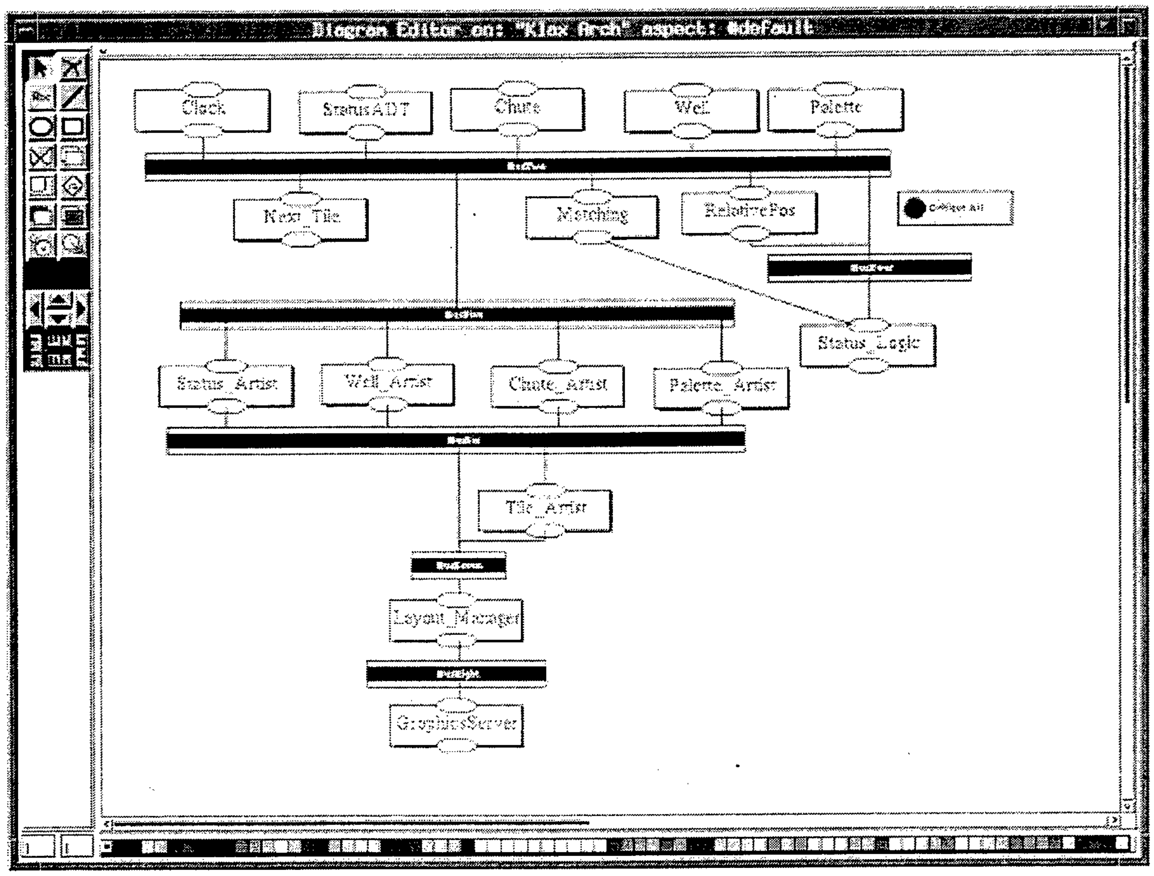


Figure 7.4: Screenshot from the *Argo* design environment for the C2 style

Although the Argo environment provides significantly more functionality than the Armani-based C2 environment, creating the Armani version required dramatically less time and effort than creating Argo. As Figure 7.3 outlines, the process of defining the C2 style in Armani and customizing the Armani environment to support the style took me only eight hours (again, this does not count time spent on domain analysis). For comparison, building the Argo environment has been an ongoing effort undertaken by multiple people over the past five or six years. Building all of the additional capabilities Argo provides (such as code generation and process management) into the Armani environment would clearly require significant additional effort. Even so, I was able to duplicate a significant portion of Argo's functionality (perhaps 30-50%) in the Armani-based C2 environment with a negligible percentage (<1%) of the development effort that went into Argo. Further, the basic Armani-based C2 environment can be significantly leveraged in creating or integrating such a set of complementary tools.

The Argo environment also provides an interesting comparison with the Armani-based C2 environment because, like Armani, Argo attempts to capture design expertise in a modular, composable way. Argo uses *design critics*, which are implemented as Java objects that observe a system design and the operations that are being performed on the design. These critics notify the user with suggestions when his or her design strays from the goals embodied in the critic. Because these critics are coded in Java and linked directly into the environment they are not readily modifiable by the architect using the environment. As a result, it is significantly more difficult for end-users to evolve and modify their design critics in Argo than it is to make comparable modifications to design rules with Armani.

One of the goals of this case study was to determine how well I could duplicate the capabilities provided by Argo's critics using Armani's type system, property construct, and design rules. To explore the answer to this question I found high-level specifications for eight Argo/C2 design critics in [RHR98] and attempted to express the expertise that these critics encapsulated in the Armani C2 style specification using only Armani's design language. Six of these eight design critics proved to be readily captured in Armani. These critics checked for: interface mismatches, connections that bypassed the message buses, overuse of memory resources, poor utilization of reusable components, too many components at a single level of abstraction, and the use of components that would not work with the code generation tool.

The critics that readily translated to Armani could all be cast as boolean questions answerable by a simple static analysis of a system's properties. Two of the critics, however, did not meet this criterion. The first of these provided component selection guidance and the second flagged system configurations that were likely to prove difficult to test. In both cases, the critics made use of significant external knowledge (knowledge not captured in the system specification being analyzed) and encapsulated their expertise as an algorithm to be run over an architectural specification that did not readily translate to a simple declarative statement. These critics would be better captured as external tools to be linked into an Armani environment than as collections of types and declarative design rules.

As a result, the critics experiments demonstrated that I can use Armani to recreate all of the critiquing capabilities found in the Argo-based C2 environment. Further, six of the eight critics were succinctly captured directly in the Armani design language and thus readily added by an end-user working with the customized Armani-based C2-environment.

7.2.4 Case study 4: two *data-centric repository* styles

For the fourth case study, I captured a pair of variations on the client-server architectural style. These styles were based loosely on the descriptions of client-server systems in [Ber92] and [OHE97]. The client-server style is widely used in modern database and management information systems. Variations on the three-tier client-server style are also currently popular for building world wide web and intranet-based systems. Although it could be argued that these styles fit into any of a number of different categories in the Shaw/Clements taxonomy, they seem to fit most cleanly in the *data-centric repository* category. Systems built in this style generally focus on one or more datastores (called *servers*) that *client* components can access and update.

As the ensuing discussion will illustrate, this case study has (at least) two interesting aspects. First, it demonstrates both how a generic Armani style can capture a basic design concept (such as "client-serverness") and also how that style can be enriched and extended to capture a much more complex, specific, and powerful style. Second, as a follow-on to the case study, a colleague integrated a powerful performance analysis tool with the Armani style, providing an initial proof of concept for Armani's tool integration capabilities.

The Naïve Client-Server style

The first style that I created for this case study was the *naïve client-server* style. As figure 7.5 indicates, this style provides two component types – *clients* and *servers*. Server components generally supply a system's persistent data storage and heavy processing capabilities. Design issues captured as properties in all server components include the services provided by the server, whether the server is multi-threaded, and the maximum number of concurrent requests that the server can handle.

Client components access the data stored in, and the services provided by, server components. Clients that provide a user-interface are represented by the special client subtype *gui-client*. Generic client components can also provide a user interface but they are not required to do so. Computational tools whose primary purpose is to make use of the data or services provided by the server are generally modeled with the generic client

Naïve Client-Server Style

Category:
Data-Centric Repository

Semantics Statistics

Primary component types:
- Naïve Client
- Naïve Server

Primary connector type:
- Client Request

Sample design rule:
- No peer-peer connections: clients may only be connected to servers and servers may only be connected to clients

Total types defined: 8
Style-wide design rules: 3
Time to define: 1.5 hours
Lines of Armani code: 39

Environment Statistics

Total shapes defined: 5
Customization time: 1.25 hrs

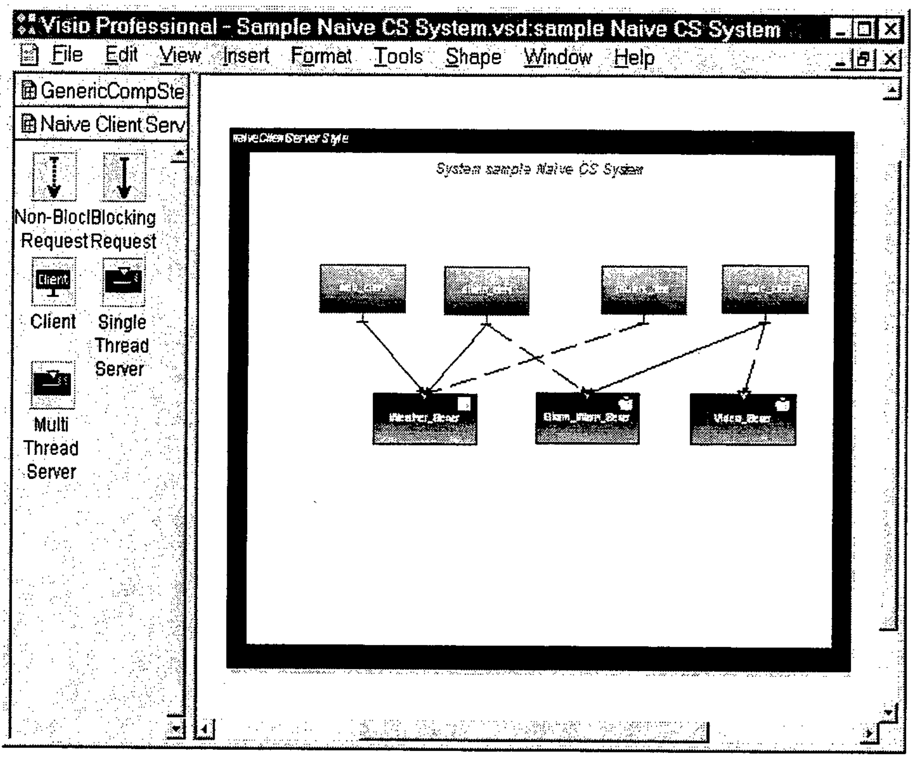

Visio Professional - Sample Naive CS System.vsd:sample Naive CS System
File Edit View Insert Format Tools Shape Window Help
GenericCompSte
Naive Client Serv
Non-Bloc Blocking Request Request
Client
Single Thread Server
Multi Thread Server
NaiveClientServer Style
System sample Naive CS System

Figure 7.5: Overview of the Naïve Client-Server style and environment

component type. The design issues that all client and gui-client components capture include specifying the requests that the client needs fulfilled, whether the client blocks waiting for a response to a request, and whether the client provides a user-interface.

Due to the simple communications model underlying the naïve client-server style, I needed to define only a single type of connector for the style. This connector type, called *client-request*, models a conduit through which messages from a single client are sent to a single server, and responses from that server are returned to the requesting client component. These connectors have only two roles, one for the client-side and one for the server side. The client-request type can be used to model both synchronous and asynchronous communication between the client and server, depending on the values assigned to various properties in the client-request type declaration. These connectors do not define the specific types of valid requests that can be sent from client to server. Instead, they are a neutral conduit through which all requests may pass. As we shall see with the three-tier extension to this style, this approach leaves style developers the option of creating new subtypes of the client-request connector type that handle only specific classes of client requests.

These type declarations are supplemented by three simple design rules that guide architects in creating well-formed designs. In the naïve client-server style, clients are allowed to communicate only with servers and servers are allowed to communicate only with clients. A simple design rule enforces this topological constraint. Likewise, a pair of design rules specify that all components in a naïve client-server design must be either clients or server-typed components and all connectors must be client-request connectors. Although the naïve client-server style is quite generic and simple, it provides a useful set of primitives for designing basic client-server systems.

The Three-Tier-Client-Server style

Chapter 2's introductory overview of the Armani approach describes some of the limitations of the naïve client-server style and introduced the *three-tier client-server* style as an evolution of the naïve client-server style. As table 2.3 described in chapter 2, the primary stylistic limitation addressed by the three-tier style is the need to separate the application-specific processing required by a system from its data storage and user-interaction components. To address this issue, I created a substyle of the naïve client-server style called the *three-tier client-server* style. This substyle defines three new component types, one new connector type, and one additional style-wide design rule to guide architects in proper usage of the new types.

The new component types defined in the three-tier style all extend component types defined in the naïve client-server style. Two of these component types – *data views* and *datastores* – are highly specialized versions, respectively, of the gui client and server component types defined in the naïve client-server style. The sole purpose of the *data view* component type is to provide a user interface to the system. Data view clients are explicitly not permitted to do any application processing. They are also the only components that are supposed to provide an interface to the user. Because of their limited functionality, data view clients are sometimes referred to as very *thin* clients. The sole purpose of the datastore component type is to provide persistent data storage. Like data views, datastore components are not allowed to perform any application processing. All application-specific processing functionality must

3-Tier Client-Server Style
Category:
Data-Centric Repository

Semantics Statistics
Primary component types:
- Data View Client
- Application Logic Server
- Datastore Server

Primary connector types:
- Application Request
- Datastore Query

Sample design rule:
- No direct Data View to Datastore connections

New types defined: 9
Additional design rules: 1
Time to define: 1.5 hours
Lines of Armani code: 23

Environment Statistics
New shapes defined: 12
Customization time: 2 hours

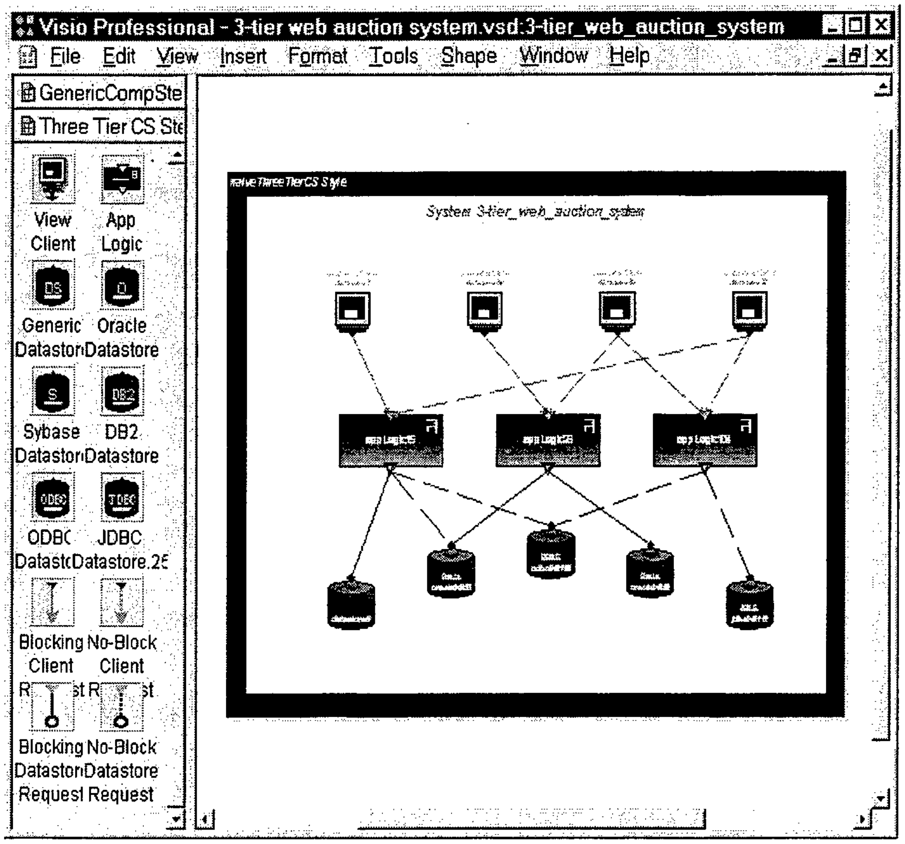

Visio Professional - 3-tier web auction system.vsd:3-tier_web_auction_system
File Edit View Insert Format Tools Shape Window Help
GenericCompSte
Three Tier CS Ste
View Client
App Logic
Generic Datastore
Oracle Datastore
Sybase Datastore
DB2 Datastore
ODBC Datastore
JDBC Datastore.25
Blocking Client Request
No-Block Client Request
Blocking Datastore Request
No-Block Datastore Request
Naive Three Tier CS Style
System 3-tier_web_auction_system

Figure 7.6: Overview of the Three-Tier extension to the Naïve Client-Server style and environment

be done by *application servers*, which are the third new type of component. Application servers are subtypes of both the client and server types defined in the naïve client-server style because they simultaneously act as servers for the data view components and as clients of the datastore components.

In addition to the three new component types, the three-tier style required the introduction of the new *datastore request* connector type. Datastore request connectors link application servers to datastores with a database query-and-update protocol and/or language. The generic client request connectors defined in the naïve client-server supertype proved appropriate for continued use as communication channels between data view and application server components.

Specializing the naïve client-server style to work as a three-tier style required the addition of only one significant style-wide design rule. This rule refined the superstyle's connection restrictions that allowed only clients to be connected to servers and vice-versa. Specifically, the new rule requires that all connections between application servers and datastores be made with datastore request connectors. Adding this design rule and simply enforcing the superstyle's design rules proved to be sufficient to capture the fundamental topological constraints of the three-tier style.

The introduction of these new types and design rule provides the three-tier style with a standard way to modularize and partition functionality and responsibilities throughout a distributed client-server system. Modifications made to the application-specific processing capabilities of a three-tier client-server system can be isolated and updated without disturbing the system's user interface or it's core data storage components. Likewise, the data storage or user interface technology can be updated or replaced without disturbing the core application logic.

In addition to the controlled portion of this case study that I conducted, a colleague of mine integrated her performance analysis tool [SG98] with the three-tier style environment. Because I did not do this portion of the experiment and the person who did the work did not track her effort precisely, I did not include the time and effort required to integrate this tool in the measurements I took for these case studies. The process of integrating the performance analysis tool required, however, only a couple of days of effort. The integration of this tool into the Armani environment significantly enhanced the utility of the completed environment by providing a queuing network-based static performance analysis of system throughput and latencies.

The process of integrating this legacy tool into the Armani environment was relatively straightforward. At the Armani semantic level, the developer had to require that all request connections were asynchronous and add some additional properties to the component and connector types defined for the style. These properties held information about the rate at which the individual components would generate requests, how long a server took to respond to a request, the buffer size and delivery delay of each connector, and the rate at which requests were introduced into the system as a whole. An architect using this tool is required to provide these values for the individual connectors. The analysis tool then uses the values for each of the individual design elements and computes processing rate, throughput, and latency values for each element in the system when it is running in a steady state. This analysis can identify potential performance problems, bottlenecks, and overall system overload.

Once the Armani style had been updated to provide the information that this analysis tool needed it took about a day to integrate the tool with the rest of the environment and have it providing useful analyses. This was a tight integration between the tool and the environment, with the analysis tool working directly on the Armani design representation. To further tighten the integration between the Armani environment and this tool, the integrator designed and coded a set of dialog boxes for dealing specifically with the performance-analysis related properties. Although this information was previously available in Armani's generic element workshops, the custom dialog boxes provided a more specific view of the performance attributes of both the individual design elements and the system as a whole. Creating these dialog boxes took less than one week of effort.

Discussion and evaluation

Perhaps the most encouraging finding from this pair of case studies was how easily and effectively I was able to extend a very generic style (the naïve client-server style) to create a much more focused and constrained substyle (the three-tier style). The three-tier style specification required minimal new Armani code to capture the additional vocabulary, constraints, and semantics of the substyle. The resulting substyle, however, provided a well-defined and quite specific framework for building three-tier client-server systems. This framework, the structural vocabulary, and the standard composition patterns of the three-tier style provide architects with significant design guidance and a leverage in creating three-tier systems.

The informal experiment in integrating an external design tool demonstrated that although not as easy as writing a design analysis directly in the Armani design language, integrating external analysis tools is feasible and relatively time-effective. Integrating the performance analysis tool with the three-tier style environment provided a powerful analytical capability to the custom Armani environment with relatively low effort.

7.2.5 Case study 5: two *call-and-return* styles

In the fifth case study, I created a pair of architectural styles and environments to support the *driver-subprogram* style described in [Be90]. These styles are variants of the generic Call-and-Return styles described in the [SC96] boxology paper. As with the batch sequential style described in the first case study, Best's description of the driver-subprogram and it's db-driver-subprogram substyle is too informal to translate directly into Armani. To capture the styles' essential concepts and constructs, I therefore had to add some additional formality to the style specification. As in previous case studies, adding the required formality proved to be straightforward. Adding this additional rigor required that I explicitly specify numerous design decisions that were implicit in the style's informal specification. Rather than diminishing the value of these style definitions, exposing these implicit design decisions significantly clarified the architectural specification of the styles.

The Driver-Subprogram style

The first style that I created in this case study was the *driver-subprogram* style, outlined in Figure 7.7. The key observation underlying the driver-subprogram style is that the infrastructure used in many large data processing applications is basically the same, though the specific tasks undertaken at each processing step vary from application to application. This is basically the same observation that underlies the batch sequential style described in section 7.2.1. The primary difference between the batch sequential and driver-subprogram styles lies in whether they are data driven (batch sequential) or control driven (driver-subprogram), and whether the processing emphasizes modifications to a stream of data (batch sequential), or transactions that might need to access one or more databases (driver-subprogram).

The first step for building systems in this style is to perform a functional decomposition of the system's processing tasks. This decomposition divides the processing into a set of discrete functions. These functions are then mapped to a set of *subprogram* components that execute the requested functions. Every system also has a exactly one primary *driver* compo-

Driver-Subprogram Style
Category: Call and Return

Semantics Statistics
Primary component types:
- Driver
- Subprogram
- Subdriver

Primary connector type:
- Processing Request

Sample design rule:
- A system has exactly one toplevel Driver component, but may have multiple Subdriver components.

New types defined: 18
Style-wide design rules: 3
Time to define: 3.5 hours
Lines of Armani code: 73

Environment Statistics
New shapes defined: 7
Customization time: 3 hours

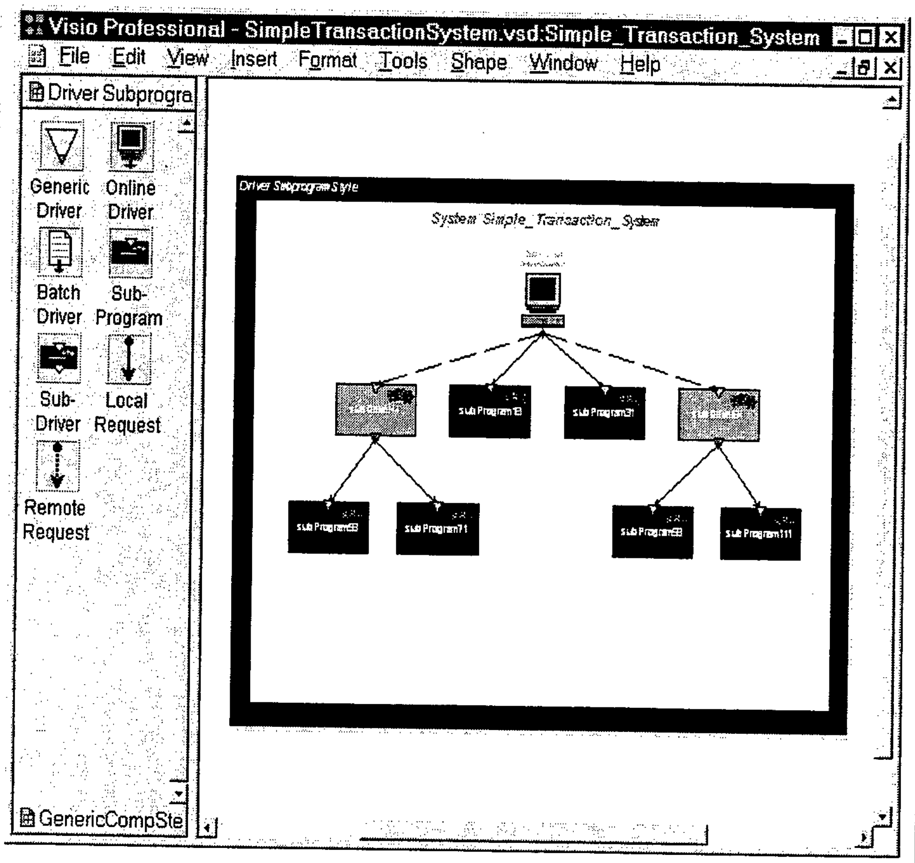

Visio Professional - SimpleTransactionSystem.vsd:Simple_Transaction_System
File Edit View Insert Format Tools Shape Window Help
Driver Subprogra
Generic Driver
Online Driver
Batch Driver
Sub-Program
Sub-Driver
Local Request
Remote Request
GenericCompSte
Driver Subprogram Style
System Simple_Transaction_System

Figure 7.7: Overview of the Driver-Subprogram style and environment

nent that triggers the execution of each of the subprograms. Unlike the batch sequential style, in which one specific sequence of operations are performed on all records, in the driver-subprogram style the driver can dynamically select which operation/subprogram will be performed at each step of the processing.

The driver component sends its requests to each of its subprograms over *processing request* connectors. These connectors transport requests from a single driver to a single subprogram. All connectors in this style are processing requests (or subtypes of processing requests). The key connector-related design issues that arise when working in this style include specifying the request or requests that will be sent through the connector, and specifying whether the requests will be sent synchronously, asynchronously, locally, or remotely. The Armani *processing request* connector type makes designers address these issues by requiring that they specify the requests that each instance of this connector type will convey. The style also defines multiple subtypes of the processing request connector type that capture synchronous, asynchronous, local, and remote variations on the base type.

Although there can be only one primary driver component in a driver-subprogram system, the style allows a system to have multiple *subdriver* components. A subdriver component is a non-leaf, non-root node in the system's structure graph. From the perspective of the (sub)driver sending it a request, a subdriver component appears to be a subprogram. From the perspective of the subprograms that it invokes, it appears to be a driver. The subdriver component type subtypes both the driver and subprogram component types to provide these facades. It can be thought of as an abstract functional unit that drives other subprograms to compute its function.

Given this palette of design rules, component and connector types, architects working with this style are able to address the key systemwide issues they face when building driver-subprogram systems. These issues include: establishing how functionality and responsibility are divided between different components, establishing appropriate modularity and abstraction boundaries to minimize the effects of anticipated system evolution paths, reuse of system infrastructure across similar systems, and the ability to cleanly separate business policies from system implementation. The Armani specification for this style provides both a framework for addressing these issues and a basic reusable design for building this type of system.

The DB-Driver-Subprogram style

The driver-subprogram style provides a standard infrastructure for a significant class of information processing applications. The style does not, however, provide any built-in structure for interacting with databases or handling transactions that persist across multiple requests to a driver's subprograms. To address this limitation, I used Armani's substyle construct to create a new architectural style called the *db-driver-subprogram* style that explicitly supports the specification of database and transaction management components and connectors.

As Figure 7.8 indicates, the db-driver-subprogram style defines three new component types and two new connector types to support database access and complex transactions. All of these types are provided in addition to the vocabulary elements (types) defined in the driver-subprogram superstyle (which are also available when working in the substyle). The three new component types defined in the style are the *database*, *db-access-subprogram*, and *transaction manager* types. As their name suggests, *database* components persistently store data that is shared by multiple subprograms (and even multiple systems). *Db access subprogram* components are subtypes of *subprogram* that support database queries and updates.

The *transaction manager* component type is the most architecturally interesting of the additional component types. Every db-driver-subprogram based system has a single transaction manager component that manages access to the databases. The transaction manager receives requests to begin, commit, and abort transactions from the system's primary driver. It then limits and allows the db-access-subprograms to access the system's databases appropriately. Having a single transaction manager that controls access to all of the databases allows the driver to invoke multiple subprograms within the context of a single transaction. In many situations this approach significantly simplifies the programming model for creating drivers to solve specific business problems.

The style also introduces two new connector types, both of which are subtypes of the *processing request* connector type. *Db-query-update* connectors allow db-access-subprograms to access database components. As their name suggests, they support requests to both query and update the database. Although these connectors provide the ability to request access to a database, the requests will only be honored if the transaction manager has given the requesting component permission to access the database. The second new connector type is the *transaction request*. A system's driver component interacts with the transaction manager through a transaction request connector. The transaction manager also uses transaction request connectors to interact with a system's databases.

The additional vocabulary and design rules introduced in this style proved effective at helping architects call out and explicitly make the key system-level design decisions they face when working in this style. The infrastructure carried over from the basic driver-subprogram style helps the architect address the issues that the substyle shares with the super style (determining the functional breakdown and processing requirements for a system's individual subprograms, separating business logic from infrastructure, etc.). The db-specific substyle also helps an architect address issues relating to the database structure of a system, the

DB-Driver-Subprogram Style

Category: Call and Return

Semantics Statistics

Extended component types:
- Transaction Manager
- Database
- DB Access SubProgram

Extended connector types:
- DB Query Update
- Transaction Request

Sample design rule:
- A system has exactly one Transaction Manager that must be connected to all databases.

New types defined: 12
Additional design rules: 6
Time to define: 2.0 hours
Lines of Armani code: 63

Environment Statistics

New shapes defined: 12
Customization time: 2 hours

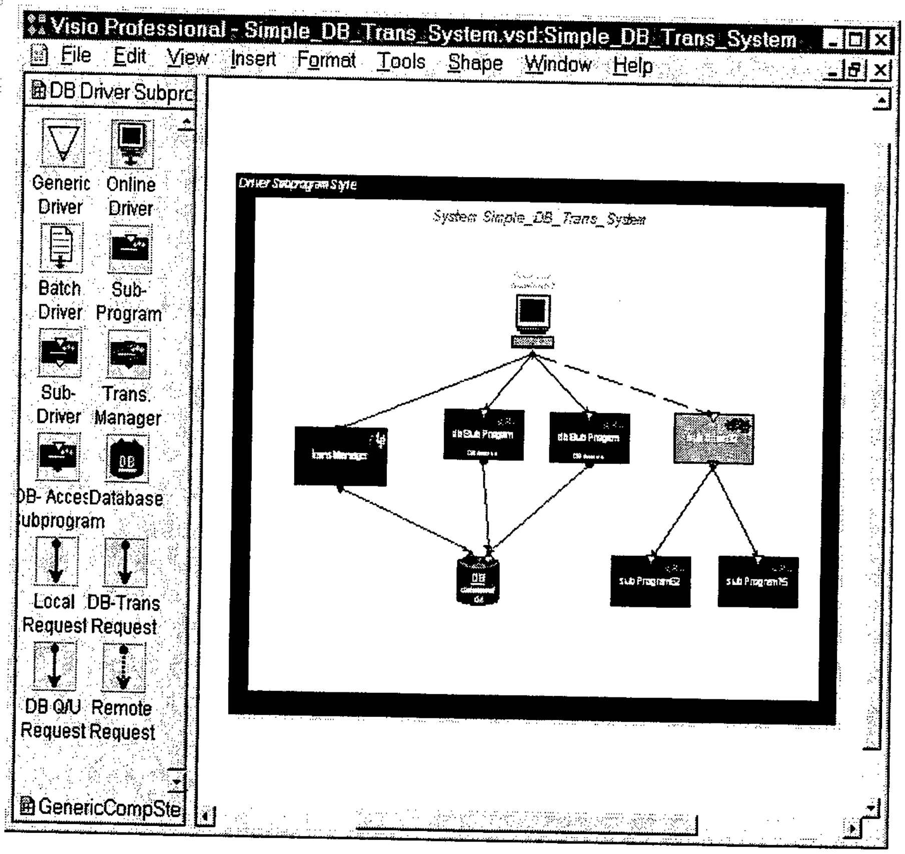


Figure 7.8: Overview of the Database extension to the Driver-Subprogram style and environment

transaction models that the databases (and entire system) will support, the database query protocols supported by the individual databases and required by the subprograms that access them, transaction abortion and error recovery schemes, and modeling of system performance. As this length list illustrates, the extensions that the substyle adds to the base style provide significant analytical and modeling power.

Discussion and evaluation

Overall, this proved to be an informative and compelling case study with (at least) four interesting findings. First, I was able to use Armani to capture a pair of proven styles that are commonly used for developing large information processing systems. Second, I was able to take a fairly generic Armani base style specification and naturally extend it to capture significant additional design expertise in a substyle. Third, I was once again able to capture significant design expertise with relatively small Armani specifications.

Finally, these styles (especially the db-driver-subprogram style) provide very detailed guidelines on how to structure systems. The styles provide not only vocabulary and design rules, but also a partially instantiated skeletal system structure to use as a starting point for building systems. An architect designing a system with these styles starts with a skeletal but extensible system structure already in place. The architect extends the skeletal specification by providing additional details about each of the core components and connectors. He can also add any additional design elements required by the system. As a result, the fact that the style binds many of the design decisions an architect faces in designing a system instance provides a great deal of design leverage at a relatively low cost.

7.2.6 Case study 6: a *data sharing* style

A traditional test of the expressive power of a programming language is to build a compiler both for and in that language. After creating a small, bootstrapping compiler in a different language, this allows all development of the new language processing tools to be done in the new language itself. For the final structured case study, I conducted a similarly recursive test with Armani. The Armani language is a design language rather than a programming language, so it is not appropriate for actually implementing a configurable design environment. I did, however, use it to create an architectural style that captures the vocabulary, design rules, and reference architecture for designing custom Armani environments. This style, called the *Armani environment style*, is an instance of a *data sharing* style, which is the sixth and final top-level category of the Shaw/Clements style taxonomy. This dissertation serves as the published specification for the Armani environment style.

Style overview

The Armani environment style captures and defines the core architectural concepts and structure used in creating the generic Armani environment, the constructs used to capture variability amongst different styles, and the mechanisms used to integrate external design tools with an Armani-based environment. Figure 7.9 outlines some of the key elements of the style and the standard, default structure it provides. Because Chapter 5 describes the design of the Armani environment in great detail, I will review only its fundamental design points here.

Armani Environment Style
Category: Shared Data

Semantics Statistics
Primary component types:
- Arch. Design Representation
- Armani Parser
- Armani Unparser
- Armani Tool

Primary connector type:
- Armani Text Stream
- Direct Design Rep Access
- Design Rep Tool Interface

Sample design rule:
- Exactly one Arch. Design Representation component per system

New types defined: 39
Additional design rules: 3
Time to define: 4.5 hours
Lines of Armani code: 147

Default Style Structure

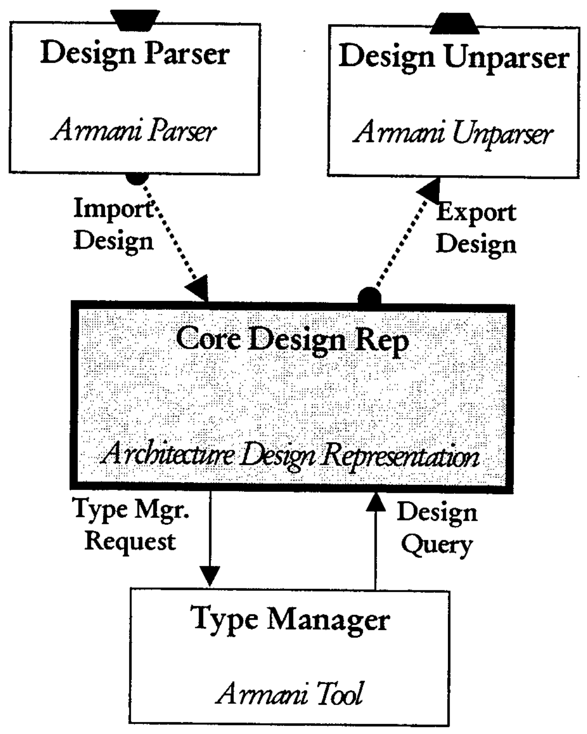


Figure 7.9: Overview of the Armani Environment Style

All system instances defined in the Armani environment style must have exactly one *architecture design representation* (ADR) component. This core component stores the architectural design representations that the environment's tools share, manipulate, and evaluate. The ADR stores the shared data that lies at the heart of the data sharing style. In addition to the Architecture Design Representation component, all design environment systems need to have exactly one *Armani parser* component and one *Armani unparser* component. The Armani parser converts textual Armani representations into object-oriented design representations stored in the ADR. The Armani unparser reverses this process, converting a design stored in the ADR into a textual Armani description. These textual Armani specifications are used both for persistent storage of Armani designs and also as a mechanism for loosely integrating design tools through the interchange of Armani design descriptions.

The fourth component that all Armani design environment systems need to include is a *type manager* that performs both type-checking and constraint management for the ADR. The Armani Type Manager is an instance of the *Armani tool* component type. All entities in an environment instance that can evaluate, manipulate, display, or generally operate on an Architecture Design Representation are modeled as Armani Tools. Three of the four primary component types used in the Armani environment style (ADR, Armani Parser, and Armani Unparser) must be instantiated exactly one time in each design. Environments generally have multiple instances, however, of the Armani Tool type. The standard way to customize and extend the generic Armani design environment with additional capabilities is to add Armani Tool components to the system. In addition to the Acme Type Manager, two additional examples of Armani Tool instances include the standard graphical user interface that ships with the Armani system and the performance analysis tool we added to the Armani system in the three-tier client-server style case study described in section 7.2.4.

The key component types used in this style are fairly straightforward. Much of the complexity of integrating a wide variety of tools to concurrently access and update a shared data repository (the ADR), however, is encapsulated in this style's connectors. The three primary connector types defined in this style are the *Armani text stream*, the *direct design rep access*, and the *design rep tool interface*. The Armani text stream connector simply streams textual Armani design descriptions from one component to another. These connectors are typically used by the system's parser and unparser to save and read Armani descriptions to the file system, or to transfer design descriptions between Armani Tools and the ADR. The direct design rep access connector provides an API that Armani Tools use to directly access the ADR.

The third connector type, the *design rep tool interface*, is the most interesting and complex of the primary connector types. These connectors have two primary uses. The first use is to provide Armani Tools with a limited, but potentially more semantically rich, interface to the ADR. The second use is to handle the interaction mismatch that arises when tool components were designed to use a different interaction mechanism than the API to the ADR. The connector that links Armani's standard Visio-based graphical user interface to the ADR is the canonical usage example for this type of connector. Design rep tool interfaces implement a very thin interface to the shared design representation that provides the Visio-based GUI with a relatively small collection of methods it can invoke on the design representation. All of these methods are directly related to the visualization and manipulation of design elements. The Visio-based GUI uses a COM-based integration mechanism [Box98] rather than the Java-based API provided by the ADR. To bridge this mechanism-mismatch, the connector provides one role with a COM interface and one role with a Java interface. The internals of the connector then provide the appropriate translations from COM requests to Java requests and vice-versa.

Discussion and evaluation

This case study proved to have three key interesting aspects. The first of these is that the style defines multiple rich and interesting connectors. These connectors, especially the *design rep tool interface* connector, address some significant component mismatch problems and provide useful transformations between request formats and semantics. The ability to capture specific API's in a set of role types and then add those roles to connectors in multiple combinations allowed me to separate the API-level interaction from the semantics that the connector itself captured. This connector type proved very effective at capturing standard, abstract interactions between the components that could be instantiated as needed to connect components that used disparate protocols (such as Java vs. COM).

The second interesting aspect of this case study was that, out of the eight styles created for these structured experiments, this style is the most specialized and focused. It also has the most detailed semantic specification. The style is not intended to be broadly or generally applicable. It provides a framework for designing and developing a very specific set of systems that share a lot of common infrastructure, a standard set of component interaction mechanisms, and a common core data representation. In exchange for choosing to work in this constrained style, architects and environment designers free themselves from having to research and make a wide variety of basic design decisions. These decisions have already been made in a standard and proven way. As a result, the designers can focus their efforts and energy on adding the specific customizations they need to the partially instantiated framework.

The third interesting aspect of this case study is that many of the types defined in the style correspond to concrete, implemented components. The configurable Armani environment described in this dissertation, for example, provides implementations for the architecture design representation, Armani parser, and Armani unparser components. It also provides implementations for the Armani text stream and direct design rep access connectors, as well as a number of instances of the design rep tool interface connector. Likewise, the COM and Java API port and role definitions model concrete, existing, implementation APIs. The application of Armani types to describe concrete, implemented, components demonstrates that the overall Armani approach is useful not only for blue-sky abstract modeling, but also as a technique for composing systems out of existing, proven systems. Although I did not create any significant generation tools to use with this system beyond the Armani environment itself, the highly focused, semantically rich nature of the style, combined with the existence of concrete component and connector implementations argues that it should be straightforward to do so.

7.3 Experimental results.

These case studies demonstrate that Armani is capable of capturing a broad range of interesting and powerful design expertise. In addition, the quantitative results of the case studies provide compelling evidence that Armani dramatically reduces the time, effort, and thus cost, of producing custom software architecture design environments.

Table 7.10 provides a detailed accounting of the time and effort expended producing custom design environments for these case studies. The rows of the table capture the four fundamental tasks required to create a custom software architecture design environment (from the task analysis in Chapter 6). The columns represent the six toplevel categories of architectural style that the case studies captured. Entries in the table specify the observed time to complete each task for each of the case studies. As indicated earlier, the domain analysis task was factored out of the case studies. Likewise, only two of the case studies (*data-centric repository* and *call-and-return*) included significant evolutionary modifications to an existing customized design environment. As a result, the time required for evolutionary maintenance was only recorded for these two case studies.

The broad result that table 7.10 illustrates is that in each of the case studies I was able to specify and implement a style-specific software architecture design environment with no more than one full day's development effort. Compared to the time required using the traditional ground-up approach, which generally requires months or years to produce comparable custom tools, the Armani approach provides significant savings.

The quantitative observations from these case studies corroborate the estimates of the time required to create such a set of environments using Armani. Table 7.11 compares the effort observed using Armani to construct the case study environments to the effort predicted by the previous chapter's task analysis for both an Armani-based approach and the traditional approach. In an attempt to make as fair a comparison as possible between the task analysis' estimates and the measured effort to develop the case study environments I assumed that each of the case studies fall into the "best-case" scenarios described in Chapter 6's task analysis framework. This assumption is supported by the facts that I am expert at using Armani, that the styles for which I was creating environments were all reasonably well defined, and that none of the environments created required significant integration with legacy analysis tools.

Environment development task	**Toplevel style category**					
	Call and Return	*Data-Centric Repository*	*Hierarchical*	*Data-flow*	*Data sharing*	*Interacting Processes*
(1) Domain analysis	Not tested	Not tested	Not tested	Not tested	Not tested	Not tested
(2) Schema capture	**3.5 hours**	**1.5 hours**	**2.25 hours**	**3.5 hours**	**4.5 hours**	**4.0 hours**
(3) Environment implementation	**3.0 hours**	**1.25 hours**	**5.0 hours**	**5.75 hours**	Not tested	**4.0 hours**
(4) Evolutionary maintenance	**4.0 hours**	**3.5 hours**	Not tested	Not tested	Not tested	Not tested

Table 7.10: Case study environment development effort log. Each cell indicates the time required to complete that stage of the case study.

Environment Development Task	Time Required (in Engineer/Days, Weeks, Months, or Years)		
	Armani case studies (observed times)	Armani task analysis (projections)	Traditional approach task analysis (projections)
(1) Domain Analysis	*Not tested*	1 week	1 week
(2) Schema Capture	Mean: **3.2 hours** Range: **1.5 to 4.5 hrs**	0.5 days	2 days
(3) Environment implementation	Mean: **3.2 hours** Range: **1.25 to 5.75 hrs**	1 day	1 month
(4) Evolutionary maintenance	Mean: **3.75 hours** Range: **3.5 to 4.0 hrs**	15 minutes	1-5 days

Table 7.11: Comparison of observed environment creation time using Armani with the times projected by the task analysis in Chapter 6. All projections are for "best-case" scenarios described in Chapter 6's task analyses.

The case study observations reported in table 7.11 argue strongly for the value of the Armani approach and the validity of the task analysis conclusions. The times projected in the task analyses are very rough estimates that are likely to have high variance in practice. The observed times for capturing a style's schema, however, were all at or below the estimated "best-case" time from the task analysis. The experimental numbers for the environment implementation task in the case study argue even more strongly for the effectiveness of the Armani approach and the validity of the task analysis (at least for the "best-case" version of the Armani task analysis). The observed environment implementation times were all significantly *lower* than the estimated best case, requiring only 3.2 hours on average per stylized environment.

The times observed for the evolutionary maintenance task in the case studies were higher than the projected effort required in the "best-case" scenario with Armani, but still a major improvement over the projected time required using the traditional approach. A quick investigation into the case studies reveals, however, that the evolutionary maintenance task was only measured for two case studies, providing a limited sample. Further, the evolutionary tasks undertaken in these case studies extended significantly beyond the description and estimating basis used in the task analysis. Specifically, the task analysis assumed the need to make a small incremental change and projected that an experienced Armani user could make such a change in as little as 15 minutes. The evolutionary modifications actually undertaken in these case studies, however, involved much more significant style and environment modifications. Each of these modifications effectively defined a new architectural style that was significantly more rich than the original style, in one case more than doubling the vocabulary elements defined in the initial environment. An alternative way to account for these case study tasks is to look at them as a series of smaller incremental evolutionary changes to an environment rather than a single atomic modification. Using this approach, the time and effort required for the evolutionary modification task is in line with the projections of the task analysis as well as a significant improvement over the time required to perform this task using the traditional approach.

Chapter 8
External Case Studies

The experiments presented in the previous chapter demonstrate that it is possible to incrementally build design environments for a broad variety of architectural styles using Armani. Further, they demonstrate that these rapidly-produced environments can encapsulate significant design expertise and analytical capabilities.

This chapter builds on these findings with a set of case studies in which software design environment developers who were not affiliated with me or my research group used Armani, or a portion of Armani, to build their own software architecture design tools and environments. These case studies, referred to as *external* case studies, illustrate how independent domain experts have used Armani to solve their design and tooling problems. In doing so they demonstrate that Armani can be used effectively by a variety of people and provide insight into Armani's strengths and weaknesses.

8.1 Experimental structure

In an unconstrained world, it would be desirable to extend the validation presented in the previous chapter with a set of controlled experiments in which real software architects used Armani to construct production-quality industrial design environments for their specific design domains. These experiments would be instrumented and controlled so that they yielded quantitative results that could be compared to the hypothesized values of the task analysis.

Given the scope, budget, and duration of this thesis research, however, such a study was not feasible. As a result, I took a more qualitative approach with these case studies. Specifically, I provided practicing software architects and software architecture researchers with the Armani toolset and environment infrastructure and observed how they used the tools to model their software systems, build custom design environments, and solve the software design problems that they faced. At the conclusion of each of these case studies I did a post-mortem analysis of the case study and the artifacts the participants had developed. I also held a wrap-up discussion with the participants to discuss their overall experience with the Armani approach and to solicit comments and suggestions for improvements. As we will see, the open-ended nature of these case studies proved to be effective at highlighting Armani's capabilities, strengths, and weaknesses.

The purpose of these case studies was to evaluate the effectiveness, power, and generality of the Armani system and the approach it embodies to rapidly developing custom software architecture design environments. Objectively determining the utility, polish, and power of the tools *produced* with the Armani system in each of the case studies was, however, explicitly beyond the scope of this thesis research. To this end, the case studies attempted to answer the following three questions:

1) Were the positive results of the case studies described in Chapter 7 due solely to the fact that the person conducting the case studies was also the primary developer of the Armani design language and design environment? Is it possible for other software architects and tool developers to effectively use Armani?

2) Did Armani's representation of architectural designs and design expertise allow the case study participants to capture the important design vocabulary, rules, and analyses for their specific domain of expertise? What did they find straightforward to represent and what did they find difficult to represent?

3) What aspects or features of the Armani environment did the case study participants find useful? What did they find unnecessary or counterproductive?

To ensure that the case studies could provide useful answers to these questions I required all of the case studies to satisfy four criteria. First, each study had to have one or more practicing software architects or tool builders (other than myself) who wanted to use Armani to solve a real and specific architectural design or tooling problem. Target problems included specifying the architecture for a significant software system or family of systems, creating new architectural design tools, and extending existing design tools. Second, to provide an accurate approximation of the motivations underlying "real" software design and development projects, case study participants had to have a selfish interest in the success of their project. Third, the pool of case studies selected had to span a variety of design domains and problem. Finally, the organization or person participating in the case study had to be willing to work with research prototype tools and to make the results of their experience using Armani available for publication in this dissertation.

These criteria insured that the external case studies addressed this chapter's three key questions. Having architects and tool developers other than myself use Armani to solve their own design and tooling problems insured that the case studies addressed the first question. Having the case study participants use Armani to solve problems in their areas of expertise addressed the second question. Finally, requiring the case study participants to have a selfish interest in the success of their project motivated them to deeply explore and test Armani's capabilities and expose its weaknesses, addressing the third question.

The Armani project was sufficiently well publicized to interest numerous people and organizations in using the tool and participating in the case studies. Four of these groups satisfied the case study selection criteria and formed the basis for the case studies described in this chapter. The four case studies selected included (1) modeling the Department of Defense's Global Transportation Network and building tools to assist the system's designers, (2) building a set of styles and an analytical tool suite for architectures that can be reliably and safely modified at runtime, (3) creating a software architecture design environment for analyzing a system's security and fault tolerance properties, and (4) integrating Armani with an architecture reconfiguration tool to handle architectural constraint specification and management.

Although the details of the experimental process undertaken for the case studies varied significantly from study to study, all of the studies followed the same basic steps:

Step 1: I worked with the target organization to identify a design or tooling problem that their organization faced and hypothesized how Armani could address it. If Armani did not appear to be a good match for their problem, or other issues such as an inability to disclose experimental results made the case study appear unattractive then we stopped here.

Step 2: After deciding that the case study was likely to be promising for both parties, I helped the case study participants install Armani at their site and together we charted a course for how they were going to use Armani to address their specific software design problems. Once they acquired a general understanding of how Armani worked, I tried to remove myself from the development process in order to let the users explore the tool and its capabilities on their own.

Step 3: In each of the case studies, the Armani users eventually ran into certain roadblocks and sought my advice on how to proceed. I took a consulting role at this point and helped them determine how they could best take advantage of Armani's capabilities to overcome their immediate problem. Generally, this simply required a fresh perspective, but sometimes this process uncovered bugs in the design and implementation of the Armani design language and environment. I recorded the problems and issues raised and, when we discovered a significant bug in Armani, I tried to fix it and return a fresh release of Armani as soon as possible. This proved to be a valuable mechanism for rapidly improving the Armani language and environment. In most of the case studies, we repeated this step numerous times until an effective and working tool, environment, or design emerged.

Step 4: Once the finished tool, environment, or design emerged from the case study, I conducted an informal post-mortem evaluation of the project. This evaluation examined Armani's role in the success or failure of the project, the capabilities that the participants were able to produce in the case study, the strengths and weaknesses of the Armani approach, and possible improvements to the Armani system.

For each of these steps, I worked as a passive observer attempting to answer the questions laid out at the beginning of this chapter. In some of the studies, such as the effort to model the Global Transportation Network, I also took an active role in helping the study participants use Armani effectively. In other studies, such as the effort to integrate Armani with an architecture reconfiguration tool, I provided almost no guidance beyond some initial assistance in installing the Armani distribution.

8.2 Case study details

Having laid out their structure and goals in the previous section, this section discusses each case study in detail.

8.2.1 Case study 1: Modeling the Global Transportation Network

In the first case study, two computer scientists from Lockheed Martin used Armani to model the architecture of the U.S. Department of Defense's Global Transportation Network (GTN). The Global Transportation Network, currently under development by Lockheed Martin and others, provides military logistics planners the capability to manage the movement of troops, cargo, patients, materiel, and vehicles throughout the entire Defense Transportation System (DTS). GTN tracks and schedules items moving through the system, as well as providing analysis capabilities to help logistics planners optimize system flow and resource utilization [LM98].

From a very high-level perspective, the baseline architecture for the GTN system consisted of a set of relational databases that keep track of all assets in the DTS, various analysis and scheduling tools that operate over these databases, a set of client application components that provide a user-interface to the system, a collection of data feeds bringing data into the system, and a proprietary middleware communications infrastructure that links these components together. Needless to say, GTN is a large, complex, system.

Lockheed Martin, as the prime contractor for the design and development of GTN, was interested in exploring how architecture design tools such as Armani could assist them in their efforts to design, build, and deploy large, complex, and heterogeneous software systems. When we began this case study, a prototype of the GTN system had already been developed and fielded. This experience exposed some design problems and limitations with the original design that needed to be addressed before the final system was deployed worldwide. We decided to use Armani to analyze proposed architectural modifications that would address the design issues raised by experience with the initial prototype. In the interest of keeping research projects off of the critical path for GTN's development, however, we agreed that the Armani modeling was best done as a supplementary design exercise rather than the project's primary system design effort.

To focus the scope and goals of the case study, we decided to attack two specific architectural problem that the GTN team faced. First, the GTN designers wanted to explore the implications of moving from a proprietary middleware communications infrastructure to a CORBA-based distributed object infrastructure[16]. Second, they wanted to explore the feasibility of migrating the client applications to Java-based applets that could be run in web browsers. To further refine the scope of the experiments, we decided to limit the initial investigation to determining how these changes would affect the performance and security aspects of the overall GTN system.

The scope of the GTN project is quite large and the system is expected to evolve and remain operational for many years. We therefore decided that it would be worthwhile to create a custom style and set of modeling tools that can be reused as the system evolves. To this end, the specific tasks of this case study involved defining an architectural style, building custom analysis tools to evaluate design alternatives, and using the Armani infrastructure as

[16] CORBA (Common Object Request Broker Architecture) is a standard framework for implementing distributed objects. The full CORBA specification can be found at the Object Management Group's website, www.omg.org. An introduction to CORBA for the lay-dummy can be found in [SO98].

an integration framework to leverage design tools created independent of Armani. Finally, after the design environment and style had been created, the Lockheed Martin scientists used the resulting environment to model and evaluate alternatives designs for the GTN system architecture. A brief overview and evaluation of this case study written by the participants is available in [KC98].

Creating the GTN style

After installing Armani, the Lockheed Martin team developed a style to capture the vocabulary and design rules used in the GTN system. Because they had previously modeled the existing GTN system design with the Aesop design environment[17] they decided against recreating the original GTN design in Armani. Rather, they decided to compare the performance and security properties of the new design to those calculated with Aesop for the original design.

GTN is a highly heterogeneous system designed to use a variety of component and connector technologies. The architects captured the diverse types of design elements and design rules for composing them with the Armani design language. These design elements included GTN's core database, which would be carried over from the previous design, its new CORBA-based distributed object middleware infrastructure, and the Java applets that were to serve as the clients in the new GTN system architecture. To capture this design expertise, the Lockheed Martin team created a single Armani style that captured the description of all of these key vocabulary elements. Table 8.1 describes the six core component types and three core connector types defined in the Armani-based GTN style.

Lockheed Martin GTN Architectural Style Overview

Component Types:

- *DatabaseServer*– manages and provides system data
- *Browser* – client that provides the system's user interface
- *WebServer* – intermediary that routes applets and HTML pages from the *DatabaseServers* to the *Browsers*.
- *CorbaComponent* – generic base type from which all Corba-compliant types are subtyped.
- *CorbaJavaApp* – Java-based applet that interacts with *CorbaComponents*.
- *CorbaSecurity* – GTN's basic security manager type specification.

Connector Types:

- *IIOP* – mediates the interaction between a pair of *CorbaComponents*.
- *JDBC* – allows web servers and *CorbaSecurity* components to access *DatabaseServers*.
- *HTTP* – transfers requests between *Browser* and *WebServer* components.

Table 8.1 Core component and connector types defined in the revised GTN style.

[17] For further information on the Aesop system please see section 3.1.1 in the related work chapter of this dissertation, or [GAO95].

Design rule specification in Armani	Discussion
Topology: Invariant Forall c1 : component in self.components | Forall c2 : component in self.components | Forall conn : connector in self.connectors | (*attached*(c1,conn) and *attached*(c2, conn)) -> (*validCORBAconnection*(c1, c2, conn) OR *validDBconnection*(c1, c2, conn));	Specifies valid combinations of connected (component, connector, component) triples. The analyses *validCORBAconnection* and *validDBconnection*, defined elsewhere by the style, establish which components can talk to each other and via which connector types. Along with two other related invariants, this invariant ensures valid system topologies.
Security constraint: Invariant Forall c1 : component in self.components | *DeclaresType*(c1, WebServer) -> Exists c2 : CORBASecurity in self.components | *Connected*(c1, c2);	Ensure that all WebServer components are attached to one or more CORBASecurity components to support authentication, authorization, and auditing system-wide.
System performance/load balancing: Heuristic Forall comp in self.Components | comp.componentUtilization <= 0.5;	Heuristic to indicate that no components in the system should be utilized more than 50%, in the average case. The value of the componentUtilization property of each component is calculated by the external queuing network-based performance analysis tool.
Overall system performance: Heuristic self.responseTime <= 10.0;	Heuristic that suggests the overall system response time should be less than 10.0 seconds.

Table 8.2 Selected design rules from the GTN architectural style

Architectural analyses

Specifying the GTN style in Armani provided a basis for modeling and analyzing the GTN system. To extend the utility of this model, the Lockheed Martin scientists linked a performance analysis tool into the GTN-customized Armani environment to analyze certain performance characteristics of the GTN system. This tool, described in [SG98], uses queuing networks to evaluate the flow of messages and requests through a system.

To support the performance analysis tool, the component and connector types in the GTN style were annotated with a few additional properties, such as the latency expected in connector types and the request response time expected for components. Using this approach, an architect specifies the performance characteristics of a system's individual components and connectors working in isolation. These values can be obtained either by instrumenting and measuring the element in question, if the concrete implementation of the design element is built and available for testing, by providing estimates of expected performance, or by setting performance requirements for a design element that is yet to be built. The performance analysis tool then uses the individual component and connector's performance information to compute the throughput, latencies, and potential bottlenecks for the entire system as a whole.

The Lockheed team successfully used the performance analysis tool to compare the performance properties of the old design to the performance properties of the proposed CORBA-based revisions. They also compared the performance properties of various architectural design alternatives for the revised GTN architecture. This analysis helped them rule out some of their proposed architecture modifications and fine tune the more promising design proposals (see [KC98]).

Estimating and evaluating the performance of a revised GTN system architecture was only part of the goal of the case study. System security (authorization, authentication, audit trail, etc.) in a defense system such as GTN is a critical concern for the system's designers and users. Moving from the proprietary middleware infrastructure of the previous design to the standard CORBA-based middleware infrastructure of the proposed design was only possible if doing so did not violate the system's security integrity.

The Lockheed Martin system modelers did not have a tool available to quantitatively analyze system security as they had done with the performance analysis. As a result, they were forced to take a different approach to using Armani to reason about security issues in their GTN designs. The approach they chose was to encode well known architectural patterns and best practices for building secure systems in the GTN style's vocabulary and design rules. This allowed the architects to demonstrate that they were using a well known and proven approach to creating secure systems, as well as providing visibility into the specific ways that GTN architectures addressed their security requirements.

The specific architectural pattern used to insure security in the revised GTN system design proposals required one or more *CORBASecurity* components to mediate the interactions between the GTN system's key components. Not surprisingly, requiring the *CORBASecurity* component to take such an active role in all transactions introduced a number of performance issues. The quantitative performance analysis tool integrated with the GTN-specific environment helped to address this problem by allowing the GTN architects to quickly experiment with the performance and security tradeoffs of multiple designs. This combination of quantitative design analysis tools, design patterns, and best practices allowed the Lockheed Martin team to build an effective and useful custom architectural design environment.

Evaluation

By using Armani to model a complex, heterogeneous style with real industrial applicability, this case study provided a useful validation of the overall Armani approach. The study provided three specific encouraging findings. First, after a few iterations the Lockheed Martin scientists were able to concisely specify both the GTN style and the GTN system architecture. Second, Armani and it's Acme subsystem proved an effective framework and infrastructure for integrating architectural design tools. Third, the environment designers were able to use the design expertise that they captured with the Armani design language and through the linked-in tools to perform useful analyses of the GTN system's performance and security.

Although the case study was an overall success, it illuminated two challenges facing the Armani approach. The first of these issues is that although Armani can provide significant leverage when used properly it does not relieve its users of the need to be skilled modelers and architects. This is especially true of those using Armani to specify architectural styles and create custom design environments, rather than simply using a custom Armani environment created by somebody else. The Lockheed Martin scientists had to iterate through multiple design alternative before finding an effective way to represent GTN's core vocabulary and design rules. Although the structure Armani provides appeared to assist them in this process, it still required significant intellectual effort to deeply understand the important issues in the GTN design.

The second issue was raised by one of the Lockheed Martin scientists participating in this case study. He found the declarative nature of Armani's design language challenging to take advantage of and use to its full potential. Specifically, he found it difficult to make the conceptual leap from specifying how a component should perform its task (as a programmer does) to simply declaring the desired structure, vocabulary, and design rules of an architectural design or style. This struggle was most apparent in the difficulty he had in specifying the design rules that defined topological and interface constraints. In the initial iteration he created unnecessarily complex analyses to verify interface matches. With a little guidance from me, however, he was able to rephrase these analyses by simply placing a few short design rules (on the order of 1-2 lines of code each) into the appropriate type definitions. Eventually, he acquired an appreciation for the simplicity of this approach, but getting to this point required a nontrivial amount of training.

Epilogue

The Lockheed Martin architects seemed to be satisfied with the modeling tools that they built on top of Armani and felt that the tools ended up providing them with some useful analytical leverage. It is, however, unclear how much effect their architectural analysis had on the eventual evolution of the GTN architecture. It does not appear that the tools they built have made the transition from an interesting research project to daily use within Lockheed Martin. Their experience with Armani and its associated performance analysis tool were, however, apparently successful enough that the tools were included in an internal Lockheed Martin follow-on project to continue pursuing the use of architecture design and analysis tools.

8.2.2 Case study 2: Building an analytic tool-suite

In the second case study, a computer scientist at the Software Engineering Institute (SEI) and a graduate student in Electrical and Computer Engineering at Carnegie Mellon University used Armani to define an architectural style called *MetaS* and to create a tool suite for analyzing system designs done in the MetaS style. The new style used the *Simplex* architectural style [WS97, Sha96] and Honeywell's *MetaH* architecture description language

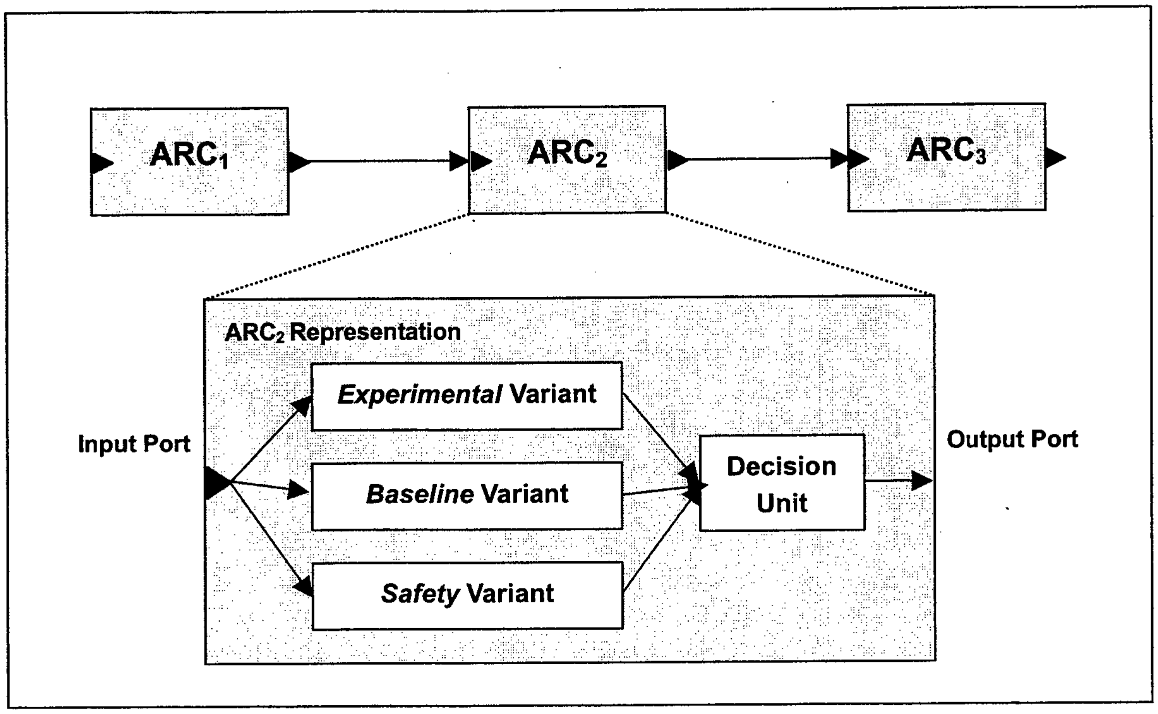


Figure 8.3 Simple MetaS architecture illustrating generic ARC representation

[Ves94] for its conceptual foundation. As the project grew, the developers also extended some of these analytic tools to work with other, more general, architectural styles.

The MetaS architectural style

The MetaS architectural style supports the safe, online upgrade of software components in mission-critical, real-time systems. The key technical concept supporting MetaS's safe run-time upgrade capability is the use of *Analytically Redundant Components*, or ARCs. An ARC consists of four subcomponents. Three of these subcomponents, called *variants*, provide redundant implementations of the component's functionality with different non-functional properties. The fourth subcomponent, called the *DecisionUnit*, monitors the output of the variants and selects the results of one of them for propagation to the parent ARC's output interfaces. Figure 8.3 provides a graphical depiction of a generic ARC's substructure. This diagram is based on the graphical depictions the case study participants used to describe the MetaS style's ARC's.

ARCs can have up to three variant subcomponents – an *experimental* variant that represents the least-proven version of the component, a *baseline* variant that represents a more proven but not necessarily bulletproof version, and a *safety* variant that represents the most reliable version of the component available. These three variants, along with the DecisionUnit, work together to ensure that the safety of the overall system is not compromised by any undiscovered bugs that a system upgrade might have introduced.

Although an ARC may contain multiple variants, precisely one of each ARC's variants is *preferred* at any given moment. As long as the preferred variant's results are within the range of acceptable output values its results are propagated to the ARC's output port or ports. If, however, the preferred variant's results stray outside the range of acceptable values then the ARC's DecisionUnit subcomponent can over-rule the preferred variant and switch to an alternative variant. At this point, the alternative variant selected becomes the new preferred variant. Because the MetaS style is intended for real-time, mission-critical systems this switching has to be performed while the system is running and without violating any of the system's real-time requirements. MetaS' ability to safely upgrade software components comes directly from the interactions of these four ARC subcomponents and their ability to do this real-time component switching.

As suggested by figure 8.3, the case study participants chose to model ARCs in Armani by creating an ARC component type. The ARC component type contains a representation with the three variants, a DecisionUnit component, a set of connectors to link the variants to the DecisionUnit, and appropriate bindings between the ports of the outer ARC and it's inner variants and DecisionUnits. Which variant is *preferred* is specified by a property placed on the DecisionUnit subcomponent.

The MetaS toolkit

After capturing the MetaS architectural style in an Armani specification, the case study participants built a set of tools for analyzing MetaS system designs. This toolkit provided analyses for insuring data format consistency throughout a MetaS system, analyzing the impact of design modifications, and maintaining configuration consistency constraints. A brief overview of each of these analytical capabilities follows:

Insuring data format consistency. The MetaS style provides a collection of properties, design rules and analyses for maintaining consistent data formats across all of a system's connected components. These properties, design rules and analyses provide a mechanism by which architects can specify the preconditions, postconditions, and obligations that ports and roles place on the data flowing through them. This capability was implemented entirely with native Armani constructs. As a result, the style developers were able to leverage Armani's type and constraint management system and provide this verification capability with minimal effort.

Impact analysis. The effects of modifying one aspect of a software system frequently ripple throughout the rest of the system in unexpected ways. To address this issue, the MetaS tool builders developed a tool to track dependencies throughout a system and analyze the impact of modifications to the system. Known dependencies are specified by annotating a MetaS system description with dependency properties. The style and tools support various types of dependencies, such as dataflow dependencies (e.g. preconditions, postconditions, and obligations), implementation dependencies, timing dependencies, etc. The impact analysis tool is capable of extrapolating from explicitly specified dependencies and inferring implicit dependencies to evaluate the effects of proposed changes.

The case study participants implemented the impact analysis tool as a collection of properties, design rules, and design analyses written in Armani, augmented with an analysis tool written in Java and linked into their tool suite through Armani's *external design analysis* construct. The decision to supplement the native Armani with an external Java-based tool was made based on efficiency concerns. When they implemented a simplified version of this analysis using only native Armani analysis functions they found that their implementation suffered from combinatorial explosion problems. The general quantification algorithms used by the Armani system simply considered too many alternatives as the systems grew large. To increase the efficiency of the analysis they re-implemented it directly in Java using the ArmaniLib's API. Explicitly specifying the analysis algorithm in Java increased the efficiency of the analysis by allowing the tool builders to use domain-specific knowledge to carefully prune the impact analysis search tree. As a result, they were able to significantly reduce the number of possibilities that the analysis had to consider.

The Armani design language's declarative nature allows an architect to specify *what* an analysis should calculate but it does not support the ability to carefully tune *how* that analysis performs the calculation. This experience with the MetaS impact analysis illustrates some of the inherent limitations of a purely declarative language and why it is important to be able to augment such a language with algorithmic specifications. I will discuss the implications of this approach on system performance and analysis development effort in greater detail in Chapter 9.

Maintaining configuration consistency constraints. The third key analytical capability provided by the MetaS tool suite is the ability to discover and maintain configuration consistency constraints. In a MetaS architecture, a *configuration* is defined as a system and a specification of which variants are preferred for each ARC. A single system architecture can therefore have many different configurations, some of which will likely be valid and some of which will likely be invalid. Furthermore, a system's configuration can change while the system is executing if one of the ARCs changes its preferred variant.

Because some system configurations may be invalid, it is important to be able to detect invalid configurations and prevent the system from attempting to transition into them. The MetaS run-time environment is designed to detect errors in individual ARCs and change the preferred variant for that component accordingly. Much of MetaS' power and utility comes from this observe-and-repair approach. This approach does, however, have its limitations. For example, it is possible that the variant selected to replace the failing variant will itself cause the system to go to an invalid configuration.

To address this problem, the MetaS run-time infrastructure needs to be able to verify that it is not trying to move the system to an invalid configuration. This check can be performed at run-time, just before the ARC switches its preferred variant. Because MetaS is designed for real-time-mission critical systems, though, this check must be very fast. Performing an on-line change impact analysis and verifying that all architectural constraints are satisfied is impractical for arbitrarily complex architectures. Although these analyses are relatively quick to perform at design time, Armani can not guarantee that they will be completed in a fixed number of milliseconds, as the MetaS runtime infrastructure requires. The MetaS tool suite avoids this problem by performing a static design-time analysis on system specifications to determine which configurations are valid and which are invalid. The results of this analysis are be cached by the running system so that verifying the validity of a proposed configuration change is a fast, simple, cache-lookup operation.

The MetaS tool suite developers implemented such a configuration consistency analysis tool. This tool tests all possible system configurations to determine which ones satisfy its design rules and data-format consistency checks. Unlike the other two analytical capabilities built for the MetaS tool suite, the consistency analysis was implemented as an independent tool that operates on Armani design representations through the ArmaniLib API. Once the analysis is complete, the tool annotates the Armani system with a property describing the system's valid configurations. To operate efficiently, this tool, like the impact analysis, had to be able to prune the configuration evaluation search tree. As a result, it was best implemented as an external tool rather than a design analysis captured with the Armani predicate language.

In implementing these three analytical capabilities, the tool developers realized that only the configuration constraint management analysis was specific to the MetaS style. The data format consistency and impact analysis capabilities were relatively generic and could be applied to systems built in other styles, provided that they had some of the characteristics of the MetaS style, such as dataflow-based connectors. Based on this realization, they chose to use a two-tiered approach to implementing the analyses. Specifically, they divided the MetaS style into two separate styles – a base style that contained the generic design analyses implementing the data format consistency and impact analyses, and a substyle (the MetaS style) that defined the MetaS-specific vocabulary and the interfaces to the configuration constraint analysis. Their experience with this approach argues favorably for Armani's tight integration of the style construct with its type system. By using subtyping with the style specification they were able to create a reusable collection of generic design analyses without any detrimental affect on the MetaS-specific style or the tool suite's capabilities.

Evaluation

Overall, this case study makes a strong argument for the claim that people other than myself can use Armani to capture powerful design expertise. The MetaS tool developers pushed heavily on the limits of what the Armani language, type system, and built-in analysis capabilities could represent and check. The use of Armani's representation construct, for example, to capture multiple variants of an ARC's lower-level design was an innovative use of the language for capturing a fundamental architectural concept. The developers also pushed on the use of *external design analyses* to extend the language's analytical capabilities. As a result, they were able to create an architectural style (the MetaS style) and tool suite capable of performing compelling, non-trivial analyses on designs done in that style. Furthermore, the results of these static, design-time analyses could be carried over to the MetaS run-time environment to guide MetaS' real-time dynamic reconfiguration and fault recovery.

Their experience implementing these analyses indicates that the Armani approach provides the ability to succinctly express complex analyses with Armani's declarative language while still allowing a tool developer to write more efficient forms of those analyses if and where required. The developers were able to capture all of the expertise and analyses they needed directly in the Armani design language. To address efficiency concerns, however, they re-implemented two of these analyses directly in Java and provided interfaces to the analyses through Armani's external design analysis construct.

As in the GTN case study, this experiment revealed that tool developers vary widely in their skill with using declarative languages. Once again, one of the participants in this study had a difficult time making the leap from specifying *how* a design rule should be enforced to simply specifying *what* the design rule was so that the Armani constraint manager could enforce it. Interestingly, the other primary participant in the study immediately picked up on the benefits of the declarative approach and used it to great effect, writing a remarkable number of small, useful design rules and analyses very quickly. The full implications of this case study with respect to the value of the declarative approach to design specification are unclear. Three lessons do, however, seem clear. The first is that the learning curve for using Armani effectively varies significantly between individuals. The second is that the declarative model is not an immediate and intuitive match to the way that all software designers model their software. The third is that the declarative approach can be very effective once an architect becomes familiar with it and adjusts his or her design technique to take advantage of its strengths.

A final observation from this case study is that it took significantly longer than the other studies. The development described here took a team of two people working part-time about five months to complete. There were a number of reasons for the extended duration of this study. First, the participants were not sure what they wanted to accomplish at the beginning of the study. They used the toolkit as a platform for experimenting with numerous possibilities. Second, they were not sure how they would go about accomplishing it. Again, the Armani infrastructure provided them with a testbed for exploring different approaches. Finally, the timeline for their project extended for a full year, so they were more concerned with fully exploring alternative analytical options than they were with simply building a tool suite quickly. The process of developing the MetaS style and its associated tool suite was therefore highly iterative and experimental. Armani's lightweight incremental design model proved to be very effective for rapid experimentation and prototyping of design and tooling ideas, providing a good match for the needs of the MetaS tool developers.

Epilogue

At the time of this writing, the participants of this case study are still actively extending their tool suite and exploring ways in which they can use the tools. They had, however, successfully used their tools to model and analyze two architectural systems. The first system was a research control-system project designed to control two carts with inverted pendulums that the controller had to prevent from falling. The second system was a portion of the avionics system from an F/16 fighter jet. In each case, the tools that they created were able to analyze the systems for reliability and fault tolerance properties, as well as evaluate run-time configurations to insure that the system did not attempt to reconfigure itself into an invalid configuration while it was running. Both of these projects were demonstrated at a Defense Advanced Research Projects Agency (DARPA) conference in the summer of 1999.

The case study participants are still using Armani to experiment with various analyses. The tools will be used as the basis for one of the participant's doctoral thesis. Once they have completed their experiments they intend to package and distribute the tools for general use by the software development community.

8.2.3 Case study 3: Security and fault-tolerance evaluation with DesignExpert

In the third external case study, a team of computer scientists from Key Software Corporation [Key99] used the Armani infrastructure to build a software design environment. This environment, called DesignExpert [WMK98], provides an integrated suite of tools for graphically editing architectural system designs and analyzing their fault-tolerance and security properties.

Unlike the previous case studies, the Key Software tool builders did not take advantage of the complete Armani system to build their custom software architecture design environment. Rather, they used Armani's design representation and analysis capabilities as an integration framework to tie together three existing design tools they had in various stages of development. The Key Software team chose this approach because they needed a way to integrate their existing analysis tools to form a common design and analysis environment without having to rebuild the tools from scratch. These design tools included a fault-tolerance analysis tool (called FTA), a security analysis tool (called SA), and a graphical architecture design editor (called GADE). The Armani design representation and analysis infrastructure provided a common design representation on which these tools could operate and through which they could share the results of their analyses.

Another difference between this case study and the two discussed previously is that I took a very hands-off approach on this project. Although I provided some initial guidance in using Armani to help get their project underway, most of my interactions with the Key Software team after the initial startup phase simply involved fixing Armani bugs they uncovered. The fact that this project was successful with very little guidance from me argues for my claim that people other than the me can use Armani to solve real problems.

To provide a common design representation for the various analysis and display tools of the DesignExpert environment, the Key Software team created an Armani style that captured the core semantic structures used by the environment and its constituent tools. Figure 8.4 outlines the core elements and properties captured in the DesignExpert style. As the figure indicates, the DesignExpert style stretches Armani's original intention of handling only software architectures by defining types to represent hardware components and connectors.

The analysis tools integrated into the DesignExpert tool suite require a system designers to specify a mapping from a system's *Software Processes* to its *Hardware Processors*. Because Armani does not provide built-in support for specifying this kind of mapping, the tool developers used Armani's property construct to capture the mapping. The DesignExpert environment's external analysis tools were then able to read and evaluate these system-wide properties and perform their analysis.

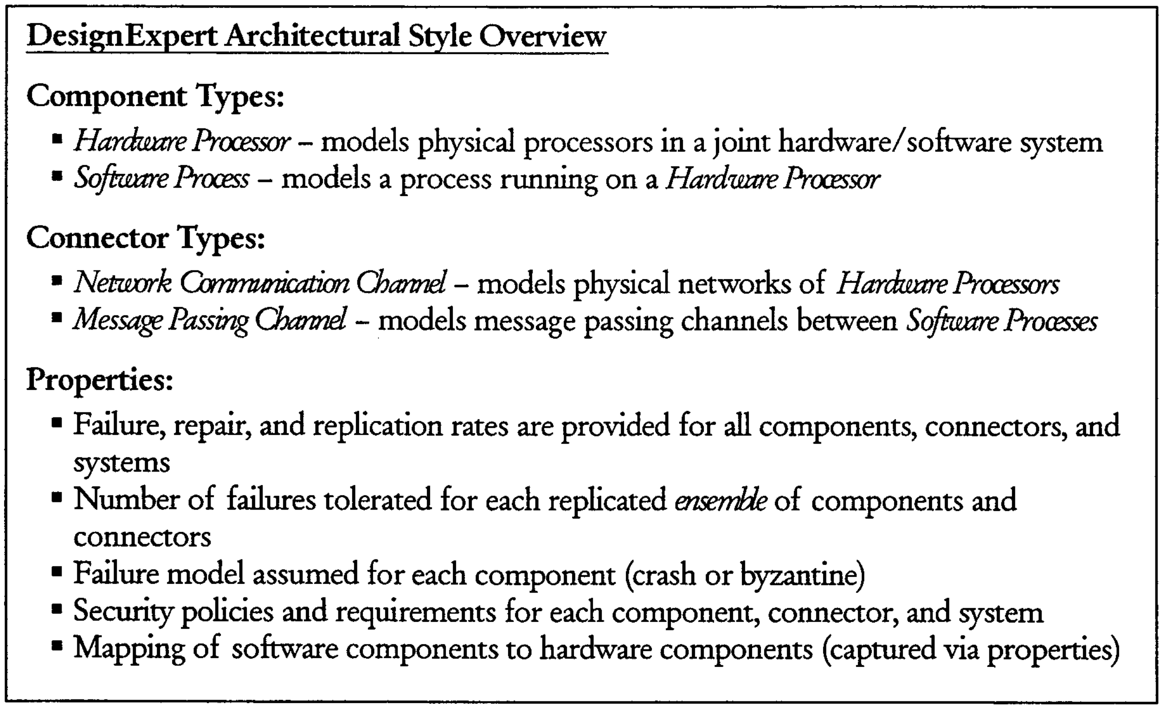
This core DesignExpert style served as an integration standard for three architectural design and analysis tools. The first of these tools, FTA, evaluates the fault-tolerance properties of system designs. FTA first does a Monte-Carlo simulation that exposes the effects of individual component failures and the probabilities that these individual failures will propagate and cause systemic failure. The results of these simulations are used to estimate system availability, Mean Time To Failure (MTTF), Mean Time Between Failures (MTBF), and Mean Time To Repair (MTTR). After completing one or more of these simulations, FTA provides a critique of the system's overall fault-tolerance and makes recommendations on ways in which the reliability of the system can be improved. If necessary, the tool also suggests additional simulations that should be run to improve the accuracy of the estimates.

The second analysis tool integrated with the DesignExpert environment is the Security Assistant (SA) tool. SA uses an expert-system to critique system designs, expose security flaws, and highlight places where the security requirements of a system are unlikely to be met by a proposed design. Like the FTA tool, SA provides architects with useful analytical capabilities and operates directly on Armani design representations.

The third tool that DesignExpert provides on top of the Armani infrastructure is an architectural visualization and editing tool (called GADE) that allows designers to graphically construct and edit system designs. GADE is tightly integrated with the Armani semantic representation, as well as the FTA and SA tools. As a result it supports animated visualizations of the FTA's Monte-Carlo simulation as well as the results of the SA's security analyses.

Evaluation

The DesignExpert team's need to capture the mapping between software and hardware architectures illustrated the limitations of encoding important semantic concepts in properties rather than as first-class Armani language constructs. Specifically, using properties to encode the mapping from software entities to the hardware entities on which they execute



DesignExpert Architectural Style Overview

Component Types:

- *Hardware Processor* – models physical processors in a joint hardware/software system
- *Software Process* – models a process running on a *Hardware Processor*

Connector Types:

- *Network Communication Channel* – models physical networks of *Hardware Processors*
- *Message Passing Channel* – models message passing channels between *Software Processes*

Properties:

- Failure, repair, and replication rates are provided for all components, connectors, and systems
- Number of failures tolerated for each replicated *ensemble* of components and connectors
- Failure model assumed for each component (crash or byzantine)
- Security policies and requirements for each component, connector, and system
- Mapping of software components to hardware components (captured via properties)

Figure 8.4 Summary of DesignExpert architectural style [WMK98].

works smoothly when an environment's analytical capabilities are encoded primarily in external analysis tools. If, on the other hand, the tool developers had written their analyses directly in the Armani design language, then specifying the design rules to manage and constraint these mappings could have become unwieldy as the number of components and connectors in a system grew large.

In this case study, however, the important analytical capabilities provided by DesignExpert came from its analysis tools rather than its built-in design rules. As a result, the property-based mapping proved an appropriate choice for this case study. The case study did, however, expose some significant limitations with this general approach to defining mappings. Experience with this case study and, to a lesser degree, with the case studies described in Chapter 7, indicates that a first-class mapping construct would be a useful addition to the Armani design language. To address this need, a group of researchers at Carnegie Mellon are currently exploring ways to add a native mapping construct to the Armani language. For the purposes of this dissertation, however, this effort should be considered "future work."

Overall, this case study successfully illustrated Armani's utility as a platform for creating and integrating architectural design and analysis tools. The DesignExpert developers found it straightforward to capture and represent the architectural features, characteristics, and properties required by their analysis tools with an Armani style. Having captured this expertise, the DesignExpert team was able to integrate their analysis tools with the Armani infrastructure to create a powerful design environment that supports two important types of design analyses – fault-tolerance and security – and a variety of interesting design visualization capabilities. Further, although the bulk of the work done in this case study involved building custom design tools on top of Armani, the Armani design language proved sufficient for, and effective at, capturing the important semantic concepts needed to support the external analysis tools.

Epilogue

The DesignExpert tool was handed off from its developers at Key Software to the Rome Air Force Research Labs. A group within Lockheed Martin's research organization also acquired the tool and integrated it into a prototype suite of Acme-based architectural specification and analysis tools. It is unclear how extensively the tool is currently used inside Rome Labs or Lockheed Martin. The tool builders at KeySoft, however, reported that they were able to use Armani successfully for modeling the fault-tolerance and reliability expertise that they needed to capture for the project. They also reported that Armani's simple and straightforward design and implementation allowed them to quickly prototype and build their tools.

8.2.4 Case study 4: Dynamic, run-time, architectural reconfiguration

In the previous case studies the participants followed the general approach outlined in this dissertation to create a custom Armani software architecture design environment. In this fourth and final case study, I describe how a software architecture researcher at The University of California at Irvine (UCI), took a different approach to incrementally customizing a set of design tools with Armani. Rather than using the entire Armani infrastructure as a basis for creating a custom design tool, he linked just the Armani parsing, design representation, and constraint management subsystems into his own design environment and toolset. In doing so, he demonstrated that the Armani infrastructure is sufficiently modular and decomposable that selected pieces of it can be reused to build design tools that extend beyond the original scope of Armani.

The environment developed at UCI and used as a basis for this case study is called ArchStudio [OMT98]. ArchStudio provides a design environment for representing and displaying architectural designs (through its Argo subsystem [RHR98]) as well as a component called the *Architecture Evolution Manager* (AEM) that controls the dynamic reconfiguration of system architectures. Software architects can use ArchStudio to design software system architectures and to dynamically modify the architecture of running systems created with the environment.

Prior to this case study, ArchStudio was designed exclusively for use with the C2 architectural style [Tay+96], also developed at UCI. Recognizing the need for a more general architectural modification tool, we decided to explore whether Armani would be an appropriate platform for building a similar but more generic tool for dynamic architectural reconfiguration of software systems. It quickly became apparent that a complete rebuild with Armani was unnecessary. Although Armani's flexible GUI and user interface were appealing, the real leverage that Armani could provide would come from the reuse of its design representation and verification infrastructure. It was also apparent that Armani's ability to represent a broad variety of architectural styles and perform generic architectural constraint management would be a valuable addition to ArchStudio. Therefore, we decided to integrate Armani's parsing, design representation, and constraint management subsystems with ArchStudio.

Because both ArchShell and Armani are fully compliant with the Acme architecture interchange standard [GMW97], integrating them was quick, straightforward, and relatively easy. Conceptually, as Figure 8.5 indicates, the integration required only the addition of the Armani infrastructure (parsing, design representation, and constraint management components) to the ArchStudio environment, the introduction of an Acme connector to connect Armani to ArchStudio's AEM component, and the definition of a basic and extensible Armani style for representing ArchStudio designs. The Acme connector used is simply a stream that carries textual Acme descriptions of the proposed architectural modifications from the AEM to Armani. Armani converts the Acme descriptions into Armani architectural specifications and evaluates the proposed modifications to see if they are consistent with the system's declared architectural style(s) and instance level design rules. After this evaluation is complete Armani returns either a success message or an error messages indicating where and why the proposed modifications violate the design rules of the system or its styles.

This integration allowed the combined ArchStudio/Armani environment to verify that proposed modifications to the running system did not violate the constraints of that system's architectural style or styles. By performing this check after a user or a tool requested an architectural change but before the change was actually implemented in the running system, the updated environment was able to catch architectural mismatch problems before they corrupted the running system. Armani's flexible and incremental language for

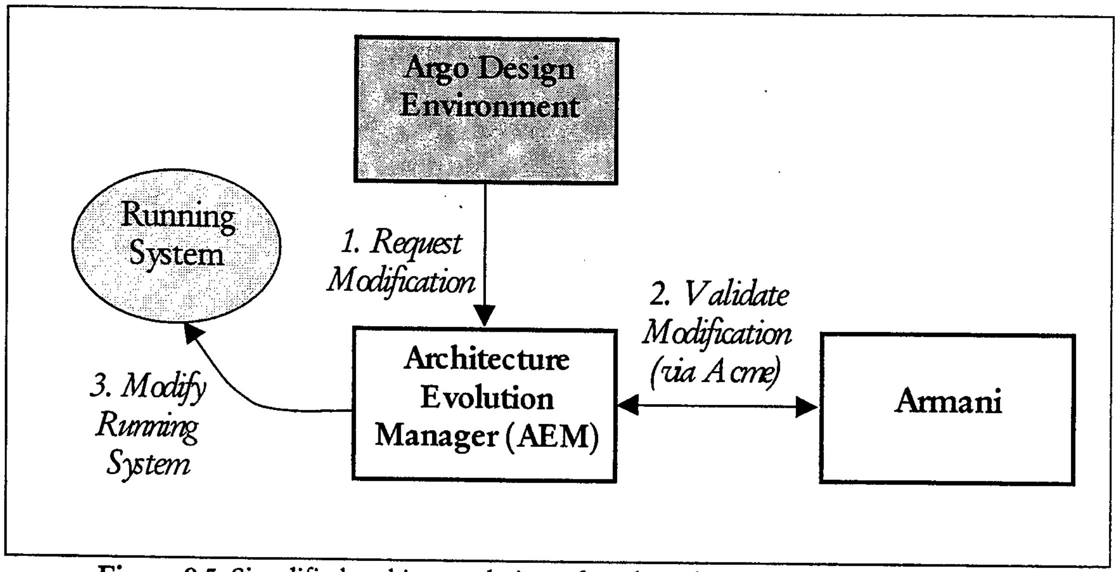


Figure 8.5 Simplified architectural view of ArchStudio's integration with Armani

expressing design rules also significant expanded the scope and variety of design constraints an architect could express in his or her ArchStudio-based design specification.

Evaluation

This case study was a resounding success. Adding the Armani core design representation and constraint management infrastructure to ArchStudio substantially improved the ArchStudio environment, providing significant additional functionality and power in exchange for only minimal effort on the part of the tool developer. Further, the integration of Armani's constraint management system into ArchStudio was done extremely quickly, the entire process requiring less than one week's effort.

The task of expressing ArchStudio's core concepts in an Armani style was straightforward and quickly handled, requiring only about two days worth of work. The ease with which the core concepts embodied in Armani meshed with ArchStudio was very encouraging, demonstrating the flexibility and power of Armani's basic model for representing software architectures, architectural styles, and design rules.

In addition to illustrating the flexibility of Armani's core constructs, this case study demonstrated that Armani's modular architecture allows tool developers to selectively and incrementally reuse specific pieces of the Armani system. As a result, Armani's support for incremental development extends beyond the ability to customize a generic Armani design environment. Armani also allows design tool and environment developers to extract, extend, and reuse the pieces of Armani that they find useful for meeting their specific tooling needs. Although Armani's constraint management system, for example, was not originally designed to handle dynamic run-time constraint checking, this capability was easily created by combining Armani with ArchStudio.

The Acme architectural interchange format was one of the key aspects that made this integration so easy. By providing a common representation for architectural specifications, along with a way to embed each tool's specification details in this common representation, Acme makes it straightforward to loosely couple architectural design tools. For the purposes of this case study, a loose coupling between the tools was sufficient to create the desired functionality in the combined environment.

Epilogue

The tool created in this case study proved to be very effective for defining and managing constraints on the run-time evolution of software architectures. The tool was successfully demonstrated at a large DARPA-sponsored conference in July, 1999. Although it is unclear how or if the tool built in this case study will be used outside of a research environment, it is sufficientyl powerful and useful to be a key element of the system that the UCI researcher built to demonstrate his doctoral thesis.

8.3 Summary and discussion

At the beginning of this chapter I laid out three questions that I hoped to answer with these external case studies. In this section I evaluate the overall results of the case studies and discuss the answers they provided to these questions. The first question was:

1) *Were the positive results of the case studies described in Chapter 7 due solely to the fact that the person conducting the case studies was also the primary developer of the Armani design language and design environment? Is it possible for other software architects and tool developers to effectively use Armani?*

The fact that the participants in these case studies used Armani to create a wide variety of interesting and useful design and analysis tools clearly indicate that it is possible for people other than me to effectively use the Armani design language and environment. Although I provided the case study participants with varying degrees of assistance to get them started with Armani and teach them how to use the tool, in each study the participants did the critical modeling, design, and toolbuilding work. The case studies were all a resounding success with respect to this question.

Having established that it is possible for people other than me to use Armani, the second question explores how well Armani meets the needs of custom design environment builders:

2) *Did Armani's representation of architectural designs and design expertise allow the case study participants to capture the important design vocabulary, rules, and analyses for their specific domain of expertise? What did they find straightforward to represent and what did they find difficult to represent?*

The results of the case studies provided a very positive answer to this question. One of the basic findings from the studies was that Armani's core concepts are surprisingly flexible and powerful. Participants in the case studies captured a broad variety of design vocabulary, design rules, analyses, and architectural styles with Armani. The nature of the expertise, for example, captured in the GTN case study was significantly different from the expertise captured to support ArchStudio's dynamic architectural reconfiguration style.

Likewise, the tools and environments created in the case studies encapsulated a wide variety of analytical capabilities. The fact that each of the environments or tools created in these case studies solved a real problem faced by software architects further strengthens the argument that the study participants captured real, useful, and non-trivial expertise with Armani.

One unfortunate finding was that there was a wide variance in the ease with which individual tool and environment developers were able to use Armani's declarative design language. Of the seven primary participants in these case studies (other than myself), two people picked up the power of the declarative approach right away and were able to use it to great effect. Two other people had a significant difficulty making the jump from specifying *how* to maintain a design rule, as they would in an imperative programming language, to simply specifying *what* the design rules were that they would like maintained. The experience of the remaining three participants fell somewhere in between these two extremes. As a result, the primary finding about what was difficult to represent with Armani was not an issue of what it was that the participants were trying to represent so much as who it was using the tool to represent it.

Therefore, the basic answer to the second question was that the case study participants generally found it possible to use the Armani design language to represent a broad variety of design expertise. The third question followed-up on this result by asking:

3) *What aspects or features of the Armani environment did the case study participants find useful? What did they find unnecessary or counterproductive?*

The most interesting finding related to this question is that in these case studies Armani's configurable graphic environment did not prove to be a critical feature. Rather, the case study participants found that Armani's semantic design representation, conceptual framework, and tool integration capabilities were more important than its GUI. This finding was somewhat surprising because my initial investigation into Armani's requirements, revealed that the need for a configurable GUI was high on the list of potential Armani users' needs.

There are at least four explanations for this finding. The first explanation is that the external case studies we undertook were not representative of the general target audience for Armani. The fact that we only conducted four case studies, and that the participants in three of these case studies already had existing graphical design editing tools that they wanted to integrate with Armani argues for this explanation. The second explanation is that Armani's core semantic and tool integration infrastructure was much more robust than its configurable GUI, especially for the earlier case studies. Developers' tendency to quickly discard software components that they perceive as buggy argues for this explanation. The third explanation is that a configurable GUI provides a snazzy demonstration for Armani, but that the real leverage Armani provides its users comes from its role as an integration framework and its ability to capture, model, and enforce design expertise. Experience working with the case study participants argues for this third explanation. The fourth explanation is that the tools that the case study participants were integrating with Armani provided their own user interfaces and they needed only Armani's representation and analysis capabilities. This was the case with the C2 and DesignExpert case studies, but not with the other two case studies. Overall, the evidence does not strongly support any of these explanations exclusively. Rather, I suspect that this finding is best explained by a combination of all four of them.

Another surprising finding from the case studies was the participants' willingness to drop below the level of the Armani design language and extend Armani's built-in design analyses with complex analytical functions written in Java. I had expected that most of the participants would want to work directly in the high-level declarative Armani design language if at all possible and only write new low-level analyses if absolutely necessary. What I found instead was that about half of the study participants preferred to capture only the basic vocabulary specifications and design rules in the Armani design language and then write their complex Java-based analytical tools to access the Armani design representation via the ArmaniLib's API.

The sample size is too small to draw general conclusions about how tool and environment developers will divide their development efforts between the Armani design language and the ArmaniLib API from these findings. It does, however, argue, that Armani's clean and coherent API for external tools is an important feature of the overall Armani system that environment and tool developers are likely to use frequently.

Overall, the case studies successfully demonstrated the utility and power of Armani, as well as its applicability to a wide variety of architectural styles and tasks.

Chapter 9

Discussion and Evaluation

The case studies presented in the previous three chapters demonstrate that the Armani project was broadly successful in meeting its goals. Armani allows software architects and architecture design environment builders to incrementally capture design expertise and to leverage that expertise in their tools and environments. In this chapter, I revisit some of the key design decisions I made in creating Armani. I first present and discuss a set of design decisions that proved to be highly effective. I then discuss some decisions that initially seemed promising but produced mixed results when implemented.

9.1 Design decisions that proved to be highly effective

Sections 9.1.1 to 9.1.4 describe four key design decisions I made in creating Armani that proved to be highly effective. The applicability of these decisions for other software development projects varies broadly. The discussion of each of these decisions, however, offers insights into ways to successfully build configurable software systems.

9.1.1 Supporting a rapid and iterative environment development process

Forty five years of software development history has illustrated that it is very difficult to correctly establish all of a system's requirements before the system is built. Likewise, it is nearly impossible to anticipate all of the implications of individual design decisions before the system has been built, integrated into its environment, and deployed. One of the standard ways of addressing this issue is to build software iteratively, using each iteration as a learning experience that helps clarify the requirements and the design of the next release of the software. Software design and development environments face these problems just like any other large software development project.

To address this issue, one of the premises underlying the Armani project was that the ability to rapidly iterate and experiment with multiple design alternatives provides environment developers and architects with significant leverage. Armani environment designers can rapidly and iteratively experiment with numerous alternatives for the design environment itself. Experimentation at this stage includes determining the appropriate design rules and vocabulary to support in a custom environment, selecting visualizations for design elements, and determining the analytical capabilities that the environment needs to provide. Because these design decisions are loosely bound in the environment they can be quickly revised or augmented as experience is gained using the environments. Armani pushes iterative evolution further than most systems by allowing not only the original environment developers to upgrade and evolve the system, but by also allowing architects using the environment to adapt it themselves to meet their needs. This rapid update capability allows good design ideas, analyses, and tools to be prototyped and quickly added to an environment. It also allows environment designers to revisit and quickly repair design decisions that proved less effective than hoped.

The advantages of this rapidly iterative approach spill over from the environment design task into the process of using a customized environment to design other systems. Architects using Armani can rapidly iterate through, and experiment with, a wide variety of design alternatives before binding design decisions for the systems they are producing with the environment. Supporting rapid iteration and experimentation of design alternatives reduces the risk associated with early design decisions by allowing a designer to quickly experiment with and test multiple alternatives before firmly binding a design decision.

The traditional alternative to using rapid, incremental evolution and configuration of software systems is to think very carefully through all of the system's requirements and the implications of various design decisions. Armani's support for rapid iteration and experimentation does not mitigate the need to deeply understand a system's requirements and the implications of design decisions. It does, however, help designers understand these issues better by allowing them to experiment with alternatives. It also reduces the risk associated with making these early decisions because they can be more easily rectified later if they prove to be poor decisions.

All of the experience and experiments with Armani argue strongly for the value of this rapidly iterative approach. Breaking from the traditional approach in this way turned out to be one of the best decisions of the entire project. Although it is unlikely that all software tool development projects would benefit from the extreme flexibility and late binding of design decisions that Armani provides, the approach is widely applicable and likely to be of great benefit to many different tool development projects.

Supporting an incremental and experimental approach to developing custom environments seems to significantly help environment developers build environments quickly, and experiment with multiple alternatives before binding design and implementation decisions. As I discuss in section 9.2.2, however, the decision to also put this incremental configuration power into the hands of the environments' end-users introduced some significant configuration and compatability issues.

9.1.2 Selecting first-order predicate logic as the formalism for design rules

Selecting first-order predicate logic (FOPL) as the formal foundation for expressing design rules and design analyses in Armani's design language proved to be an excellent design decision for (at least) three reasons. First, experience with Armani and other design tools indicates that FOPL is well understood by the architects and environment designers who make up Armani's target audience. It also appears to be a good match with their intuitions for expressing design rules. Further, the case studies described in Chapters 7 and 8 illustrate that design rules relating to system structure, topology, and properties are readily captured with predicates. Although a small portion the tool builders in these case studies had some difficulty adapting to the declarative nature of the Armani design language, none of them had any significant problem using predicates to express individual design rules..

Second, evaluating whether a FOPL expression holds over a set of assertions (given in the form of a system specification) is algorithmically straightforward. Likewise, the subset of FOPL used in Armani insures that checking whether a system's design rules hold is a decidable question. As discussed previously, the language insures decidability by disallowing quantification over infinite sets. This limitation, discussed in section 4.3.1, has not proven to be problematic in practice.

Third, FOPL predicates are readily specified independent of the environment in which they are eventually evaluated. As a result, using FOPL as the underlying design rule formalism made it possible to achieve the modularity, composability, and encapsulation of design rules required by the Armani design language.

Alternatives considered

Although FOPL clearly had many desirable properties, I also considered a number of other formalisms as a basis for the Armani constraint language. The alternatives included higher-order logics, temporal logics, and model-checking formalisms. All of these alternative candidate formalisms, however, had significant limitations. Solving the undecidability problem for higher-order logics, for example, proved to be more complex than solving the problem for first-order predicate logic. Because the higher-order logics did not present a sufficiently compelling increase in expressiveness to warrant the additional complexity they were removed from consideration.

The Armani constraint language's need to specify bounded ranges of valid design modifications led to the consideration of various temporal logics. Surprisingly, Armani's emphasis on static structure did not prove to be an effective match for the natural expressiveness of temporal logics. The expressive power gained by using temporal logic rather than FOPL was not sufficient to warrant the additional complexity it introduced into the language semantics and the automated checking tools.

I also considered various model-checking formalisms such as SMV [McM93] but their emphasis on states and state transitions did not provide a particularly natural match to the structural constraints that Armani had to be able to express. Likewise, their ability to explore enormous state spaces looking for any possible constraint violation was not necessary for the types of checking that Armani does. Armani constraints are simply a form of checkable redundancy for verifying the consistency of a static design specification. It is not necessary to demonstrate that an Armani description can *never* violate its constraints because it is a static specification. The critical check is that a specific design instance does not *currently* violate its constraints. This distinction is a subtle but important.

In light of the benefits of first-order predicate logic and the limitations of the alternative formalisms explored, selecting FOPL as the formalism underlying Armani's design rules proved to be a good design decision.

9.1.3 Appropriately scoped design rule checking capabilities

Using a predicate-based formalism for capturing design rules opens up the possibility of supporting a wide spectrum of different kinds of design checking. Solutions at the impoverished end of the spectrum do little more than check individual design specifications for syntactic well-formedness. Analyses at the other end of the spectrum support proofs about types, styles, and the composition of design elements. The verification techniques that live along this spectrum can provide computational validation tools, mechanisms to support human reasoning, or both.

In general, as a tool builder moves along this spectrum from providing very simple techniques to supporting sophisticated proofs, the power of the analyses increases as does the value of the analytical results. Unfortunately, there are also a number of trade-offs that she must make in order to get the more sophisticated analytical results. These trade-offs include: greater difficulty creating the analytical tools, a more focused scope of problems that the techniques addresses, a decrease in the speed with which results can be returned, and decidability problems.

Selecting an appropriate point on this spectrum for Armani's built-in analytical capabilities proved to be a crucial design decision. My goal was to provide as powerful a set of design rule checking capabilities as possible while still being able to meet the requirements laid out for the design language and configurable environment. The key requirements that seemed to be endangered by making the design rule checking too powerful were the ability to guarantee fast, interactive, environment performance and ensuring decidability of design rule checking (which was also a prerequisite for a good interactive environment).

Fortunately, I was able to create a set of design checking capabilities that provided designers with significant analytical power yet still met Armani's key requirements. This approach supports automated type checking techniques to verify that a specific design instance satisfies its design rules and all of the design rules stored in its types and styles. In the following Armani specification, for example, the Armani toolset can automatically determine that component A satisfies type T and that component B does *not* satisfy type T.

```
Component Type T = { Property X : int; }
System S = {
        Component A : T = { Property X : int = 7; };
        Component B : T= { Property Y : int = 7; };
};
```

As we have seen throughout the dissertation, the ability to determine whether an instance satisfies its design rules provides designers with significant analytical capability. A logical next step, then, is to ask whether a type or a style is internally consistent. In Armani, the question of whether a type is internally consistent can be reduced to the question of whether it is possible for *any* instance to satisfy the constraints of the type. The following Armani code snippet illustrates a type specification that is internally *inconsistent*. It is obviously not possible for an instance of type *T* to have a property *x* that is both less than 100 and greater than 100 at the same time.

```
Component Type T = {
        property x : int;
        invariant x > 100;
        invariant x < 100;
};
```

Unlike determining whether an instance satisfies a type, determining that an arbitrary type is inconsistent proves to be very difficult. As the previous example illustrates, however, it is frequently relatively straightforward for a human to make such a determination. Therefore, Armani provides an intellectual framework that a designer or architect can use to verify that the types and styles specified are internally consistent. This framework consists of the defined semantics for the language and the rules of predicate logic. One of Armani's key design principles is to have the computer perform the tasks at which machines are better than people and to have the architect perform the tasks that humans do better than machines. Following this principle, Armani does not support automated checking of internal type and style specifications consistency.

As designs, styles, and types become complex and they are arbitrarily composed it can become difficult to determine whether a type is internally consistent. There is clearly some benefit to being able to automate the detection of such inconsistencies. Many of these inconsistencies can be detected using theorem proving techniques. The PVS theorem-proving system [OS97], for example, detects inconsistencies of this sort.

Unfortunately, although this technique can be used to find inconsistencies in type and design specifications, because it can run arbitrarily long to prove its theorems and requires sporadic input from the user, it is not particularly effective as a substrate for interactive design environments or fully automated analysis tools. Armani trades off the ability to prove the internal type consistency for the ability to definitively (and quickly) determine whether a given instance of a design specifies its type constraints.

The standalone Armani system provides most of the checking capability of PVS, but it does not support theorem proving. Integrating PVS with Armani would allow the Armani environment to support both an interactive environment for rapidly evaluating design instances and the ability to perform more sophisticated analyses of type and style consistency. Such an integration is beyond the scope of this dissertation, but a promising direction for future work.

9.1.4 Straightforward type and constraint checking algorithms

In implementing Armani's typechecking and constraint management systems, wherever possible I opted for simplicity and extensibility of implementation rather than run-time performance. This proved to be a good design decision.

One of the assumptions that I made for performance evaluation was that architectural specifications will generally be relatively small – on the order of tens or hundreds of components and connectors. I found that straightforward type-checking and design rule-checking algorithms were sufficient for designs of this size. In fact, the algorithms I implemented to do the checking proved sufficiently fast on designs with up to one thousand components and connectors. Evaluating designs with less than one hundred components and connectors for type and constraint satisfaction generally appear almost instantaneous to the architect using the tool. Typically, the time required to pass a request from the Visio-based GUI to the Armani constraint management engine was longer than the time required for the checking engine to validate the design. The complete design checking process, however, was still fast enough to support interactive evaluation.

Because the simple type and constraint checking algorithms executed so quickly, it was not necessary to implement more complex algorithms. Although these simple algorithms might not scale to designs with tens of thousands or millions of design elements, such a detailed specification is unlikely to be an *architectural* specification. An architectural specification describes a system abstraction that must be comprehensible to humans. A design with tens of thousands of interacting elements is not likely to satisfy this criterion. Armani specifications generally achieve scale in terms of total components and connectors through hierarchical decomposition and abstraction. That is, a component at one level of abstraction might be represented as a complete system (with Armani's *representation* construct) at a more detailed level of abstraction. Armani's type and constraint checking algorithms take advantage of this built-in notion of abstraction boundaries to scale performance as system designs get large. The algorithmic heavy-lifting generally occurs at a single level of abstraction in a design (ie within a single system description). By evaluating these encapsulated abstractions as a series of individual and largely independent entities, the algorithm scales linearly in the number of systems evaluated. The task of evaluating each individual system can, however, be arbitrarily complex depending on the complexity of the predicates declared to hold over that system.

Although the overall performance of these evaluation algorithms was acceptable, a performance issue arose in one of the case studies regarding the use of a declarative language. Specifically, one of the case study participants created an analysis that, when expressed naïvely in the Armani design language, evaluates an unnecessarily large search tree of potential system configurations. The tricky problem with Armani's declarative design rules that this case revealed is that it is not always clear when they will require inefficient and cumbersome evaluation processing. Experience using Armani, however, indicates that just as it is possible to write either very efficient imperative code or very inefficient imperative code, so too is it possible to write either relatively efficient or relatively inefficient declarative code. Gaining experience using Armani appears to help architects mitigate this problem. As the case study participants (and I) became more proficient at using the Armani design language it became much easier to write clean, efficient design rules. In general, these design rules also grew shorter, simpler, and more readable as we got better at writing them.

This problem is not unique to Armani. Other tools, such as model checkers, also face this issue. In many model checkers slight changes in a model's representation can result in huge variations (i.e., orders of magnitude) in run-time performance. Supplying users with a set of heuristics and a basic understanding of how their specification choices can affect run-time performance is an important first step in addressing this problem. As these heuristics become better understood, automated tools can be created to help designers make use of them. Although this is fertile ground for future work, solving this problem is beyond the scope of this thesis.

An important step that I took to address the need for tuning the evaluation efficiency of selected design rules was to provide an interface that tool developers and style designers can use to implement their design rules and analyses directly in Java. By providing this interface, tool developers can adjust where and how their design rules are evaluated and implement the evaluation mechanisms with an imperative algorithm instead of a declarative statement. The tool builders in the case study just described (and described in detail in section 8.2.2) used this approach to move his analysis into Java. By doing so he was able to dramatically prune the search tree and improve evaluation performance.

In this case study, as well as in others, however, this step was rarely required purely for performance reasons. As discussed in chapters 4 and 5, the Armani design language is not well suited for expressing all types of design analyses. In this instance, the analyses were moved from Armani to Java because the style designer found it easier to express the analyses imperatively rather than declaratively. Performance was a secondary issue, though in this case moving to a Java-based analysis that could prune its search tree improved performance dramatically. Combining a clean, declarative language for expressing most design rules with the ability to escape to an imperative language where necessary for performance or expressiveness concerns allows Armani to provide appropriate performance characteristics for a wide variety of designs and analyses.

9.2 Design decisions that yielded mixed results

Unfortunately, not all of the design decisions I made for Armani proved to be as effective in practice as I had intended. The results of two specific decisions proved to be particularly suprising. The first of these was the decision to use a completely declarative design language and the second of these was the decision to build extreme run-time flexibility into the environment. In the following two subsections I discuss each of these design decisions and their implications in greater detail.

9.2.1 Using a completely declarative design language

The Armani design language is fundamentally declarative. An Armani specification provides a blueprint from which a system can be built and a description of the properties of the system to be constructed. It does not provide an operational description of the steps required to build the system. Nor does an Armani specification describe the mechanism to use to verify that the constraints imposed on the design are met. Both of these tasks are left to the language processing tools that operate on Armani descriptions.

Using a declarative language frees an architect or environment developer from the need to specify how to enforce his design rules. He simply needs to declare what the design rules are and the environment will enforce them for him. In general, the size of declarative design rule specifications is significantly smaller than the amount of code required to describe the mechanism for checking those specifications in an imperative language. Likewise, the declarative nature of the language encourages, if not insures, modularity and principled composition of design elements. Integrating operational specifications of design vocabulary and design rules that were not originally designed to work together tends to be much more difficult and complicated than the relatively straightforward composition of declarative specifications [Kai85].

With all of these potential benefits, it seemed that a declarative design specification language would provide designers with a great deal of leverage for minimal cost. In training other people to use the Armani language, however, I discovered that this approach has a significant drawback. Specifically, I found that some designers who are primarily programmers by training and experience found it difficult to make the conceptual leap from specifying *how* to construct a design and what that design should *do* to simply specifying *what* the design should be. Although I suspect this difficulty is intertwined with the difficulty that case study participants had in making the transition from thinking at the programming level to thinking at the architecture level, I don't have any strong evidence other than my case study observations to support this hypothesis.

Throughout the case studies, approximately half of the participants found the declarative language powerful, easy, and natural to use. Approximately one quarter of the participants struggled with the declarative nature of the language initially but eventually discovered how to use it effectively. The most disappointing finding was that the final quarter of the participants were never really able to use the declarative language effectively. Even after acquiring significant experience with Armani they had enough difficulty expressing their ideas with the declarative language that they wrote most of their design rules and analyses directly in Java and imported them into the environment as external analyses. In almost all of these cases, I was able to help them write appropriate declarative statements that provided their desired capability after they had completed their experiments. This revision exercise demonstrated that the problem was not Armani's inability to capture this kind of expertise declaratively. Rather, the problem was that the Armani language did not provide a good match with these designer's mental concepts for how the expertise that they wanted to express should be captured.

Although the number of case studies was too small (approximately eight environment developers and architects participated) to be conclusive, this finding raises a concern with the Armani approach. The declarative nature of Armani provides a lot of leverage for those to whom thinking architecturally and declaratively comes naturally. Environment developers, however, who have trouble making the jump to a declarative architectural model will likely have trouble taking full advantage of Armani's potential benefits. Armani's support for writing external analyses and design rules directly in Java addresses this issue partially. Given an opportunity to redesign a second generation of the Armani system, I would strongly consider adding algorithmic extensions to the Armani design language.

9.2.2 Building extreme flexibility and reconfigurability into the environment

One of the original hypotheses underlying Armani was the idea that providing software architects with the ability to arbitrarily reconfigure their tools and environments would allow them to closely match their tools' semantic and visual design representations to the representations they used informally and independent of the tools. Driving this idea was the further hypothesis that allowing the architects and designers who actively use these tools to do the customizations themselves would unleash their creativity and allow them to build powerful, task-specific tools without having to be expert toolbuilders. In the course of conducting this research I created a configurable environment infrastructure with which these hypotheses could be explored. The goal of this research was not, however, to rigorously test whether these hypotheses held.

As the previous chapters have illustrated, Armani does indeed provide its end-users with tremendous flexibility and configuration capability. Experience using the tool also seems to confirm the hypothesis that it allows architects to incrementally customize their environments in powerful and useful ways without having to be expert toolbuilders. Whether this capability, when deployed on a large scale, will result in a groundswell of compelling custom design tools remains an open question. The case studies, however, point to the possibility of this outcome.

Although experience building and using Armani provides a compelling argument that these hypotheses hold, this experience unfortunately also illuminates two drawbacks to Armani's extreme reconfigurability. The first drawback is that although it is easy to modify an Armani environment, it is still difficult to create *great* customizations. The process of customizing an Armani environment is simple and straightforward; the changes can be made in very small increments; and the structure of the language and environment provide designers with significant guidance in making appropriate customizations. Defining or selecting appropriate expertise to load into the environment, however, still requires significant taste and judgement on the part of the person customizing the environment. Both visual and semantic customizations face this difficulty.

Armani's style construct goes a long way towards mitigating the seriousness of this problem by providing a mechanism for aggregating coherent collections of related design expertise and visualizations. One easy guideline that an architect can use to address this problem is to customize the environment only at the granularity of complete styles. By using only complete style specifications created by experts in those domains the likelihood that the expertise captured by the style will work together in a sensible way is greatly increased. As an architect becomes more confident in her ability to articulate design expertise in the Armani design language, she can begin to experiment with creating new styles that encapsulate her expertise and are applicable to her design domains.

The second drawback introduced by Armani's extreme configurability is that there is significant value in standardization amongst a group of software designers and developers on the tools (and the configurations of those tools), the vocabulary, and the design rules that they choose to design and build their system. Used properly, Armani encourages such a group to develop their own shared set of standard design expertise with a single agreed upon semantic definition and set of visual depictions. Used improperly, this capability results in chaos with all members of the development team customizing their individual tools until they have significantly different configurations, utilize little or no shared vocabulary or design rules, and provide different graphical depictions of design elements. This result is not necessarily undesirable in all design and development situations. In many organizations, however, this chaotic approach is unlikely to lead to the timely delivery of great software.

In order for Armani to be effectively deployed in a wide range of software design and development organizations, it is important that the organization set up guidelines and procedures for managing changes and customizations to the Armani environments used by its architects. Details of the strictness and specificity of these guidelines can vary significantly depending on the development organization's processes. Defining these guidelines is outside of the scope if this dissertation, but it certainly provides an avenue for useful future research.

Armani's radical flexibility and reconfigurability provides architects with the opportunity to build highly customized design tools that solve their specific design problems and closely match their conceptual models. Taking full advantage of this capability, however, requires those using the tool to also be vigilant of the approach's potential pitfalls.

Chapter 10
Conclusions and Future Work

10.1 Summary

In this dissertation, I have demonstrated my thesis claim that:

> *It is possible to capture a significant and useful collection of software architecture design expertise with a language and mechanisms for expressing design vocabulary, design rules, and architectural styles. Further, this captured design expertise can be used to incrementally customize software architecture design environments.*

To demonstrate this claim I built such a language and incrementally configurable software architecture design environment. In the first chapter I argued the need for rapidly customizable software architecture design environments and presented my plan for providing them. I illustrated how such an environment can be constructed and customized in Chapter 2. I reviewed relevant related work and concluded that all previous attempts at addressing this problem either failed to fully provide the required capabilities or solved a somewhat different problem (described in Chapter 3). In Chapter 4 I described the Armani design language and illustrated how it's constructs for expressing system descriptions, design vocabulary, design rules, and architectural styles can be used to capture both architectural specifications and design expertise. Having specified a language that addresses the first half of the thesis, I created the Armani configurable design environment and described in Chapter 5 how its architecture allows it to be rapidly reconfigured with design expertise specifications captured in the Armani design language. Chapter 5 also illustrates Armani's flexibility for integrating external tools (which can also contain significant design expertise) and its usefulness as a platform for building new custom design tools. Together, chapters 4 and 5 demonstrate that it is feasible to use these techniques and mechanisms for capturing design expertise and rapidly developing custom software architecture design environments.

To validate the overall approach and support the thesis claim I conducted a set of case studies. Chapters 6, 7, and 8, described these experiments and their results. In Chapter 6 I provided a detailed analysis and comparison of the tasks required to create a custom design environment using Armani vs. the tasks required to build a comparable environment from the ground-up using current methods and tools. The results of this analysis argue that if the time and effort estimates used in Chapter 6 hold up to experimental verification then Armani provides a significant advantage over the status quo. To validate the analytical results, I conducted eight case studies, described in Chapter 7. One of the key results of these case studies was that I had been overly conservative in my previous estimates. Armani's performance proved to be even better than the analysis had predicted. In addition to verifying the task analysis, the case studies in Chapter 7 demonstrated the breadth, power, and incrementality of the Armani approach. The fact that I conducted all of the case studies described in Chapter 7 by myself raises the issue of whether the results of those case studies were overly skewed by the fact that I both built the tool and tested it by building sample custom environments. To address this issue, I conducted four additional case studies in which other software architects, researchers, and tool builders used Armani to create custom design environments and tools that solved specific design problems they faced. These case studies also extended the previous chapter's demonstration of Armani's power, breadth, and incrementality. Chapter 8 summarizes the results of these case studies.

After presenting a detailed description of how Armani satisfies the thesis claim, along with an analysis and case studies to validate the claim, I provided a critical evaluation of interesting issues surrounding the Armani project in Chapter 9. Finally, in this chapter I discuss the thesis' contributions and describe opportunities for future research that this work has uncovered.

10.1.1 Contributions

Having explored a new approach to rapidly developing custom software architecture design environments, let us revisit the contributions that this research makes to the field of Computer Science. The research presented in this dissertation provides:

- **A technique** for dramatically reducing the time, cost, and difficulty of building a significant class of customized software architecture design environments. This dissertation's roadmap describing how to use this technique provides significant value, independent of the Armani implementation, to a variety of audiences. This technique benefits software architecture design environment builders by demonstrating how a variety of design tools can be built through principled, incremental adaptations to a common shared infrastructure. It benefits software development organizations by providing access to highly customized tools at a much lower cost than current development techniques allow. It benefits practicing software architects by providing them with tools that closely match their design domain. Finally, it benefits researchers studying software development tools by providing a general customization technique that can likely be extended to other design and problem domains.
- **A design language**. The dissertation describes a software architecture design language that is capable of incrementally capturing software architecture design expertise with modular, reusable, first-class language constructs. The design language is also a full-fledged architecture description language (ADL) capable of describing the structure of software architectures and the constraints and guidelines under which those systems were designed and may be evolved.

  The design language contributes to the software architecture research community by demonstrating that a first-order predicate logic-based constraint language can be used to define interesting and useful design rules to guide software design and evolution. Further, the language articulates and encodes an extensible framework for capturing software architecture design expertise. In addition to its benefit to researchers, the design language also benefits software development organizations by providing a way to capture and reuse the organizational design expertise they develop in building software systems. Finally, it benefits software architects by providing an explicit technique for capturing and expressing architectural design constraints in software architecture specifications.

- **A reference architecture.** The dissertation describes a reference architecture, or architectural style, for software architecture design environments that support incremental customization. It describes the architecture of the Armani design environment, describes the mechanisms that the environment's architecture provides for incremental adaptation, and discusses some fundamental tradeoffs facing architects working in this style. This architecture is proven, applicable, and appropriate for creating software architecture design environments beyond the Armani environment described in this dissertation. This contribution is particularly useful for software tool builders who need adaptable, modular architectures to use for design tools and environments.
- **A set of case studies.** A set of detailed examples and case studies are presented to illustrate how the technique, language, and integration framework just described can be used to effectively capture software architecture design expertise and rapidly develop custom software architecture design environments. The case studies benefit people using Armani to design software architectures and build custom software architecture design environments. They provide a framework for conducting validations of similar research in software design and development tools. Finally, they are useful for researchers interested in further exploration of the ideas presented in the dissertation.

10.2 Future work:

The process of designing, building, and using Armani introduces opportunities for further research. In this section I discuss the most promising of these.

10.2.1 Generalizing flexible configuration strategies

Armani provides a point solution to the general problem of separating the standard infrastructure shared by all members of a family of systems from the variable aspects of those systems. Specifically, it allows software architecture design environment developers to incrementally configure their design environments. The principles embodied in the Armani approach are, however, almost certainly applicable to other design domains. Although developing a general solution for configuring arbitrary families of systems is beyond the scope of this thesis, the research suggests three general principles for replicating this approach in other domains.

The first of these principles is that establishing the core concepts and underlying formalism as an initial step makes the creation of the configurable tools much easier. The hardest task in developing Armani was defining the Armani design language, its core concepts, and its extensibility semantics. Once I had developed the language and an infrastructure for processing the language it proved straightforward to implement the configurable environment on top of the language infrastructure. The key configurable aspects of the Armani environment were built directly into the language so I simply had to make the environment expose these configuration capabilities to the end user. The applicability of this approach to other domains is an open question and fertile ground for further research.

The second core principle is that the infrastructure that serves as the basis for the family of custom software systems needs to provide a useful set of capabilities. If the domain does not have a standard set of system capabilities common to all systems in the domain, then the domain is probably not a good candidate for using this approach. The minimal, unconfigured Armani design environment, for example, is a fully functioning design environment. It supports the representation of system designs, graphical design depiction and manipulation, design rule checking, and the integration of external tools. Further research is needed to develop guidelines for determining a domain's key common infrastructure pieces.

The third core principle is that it is important to provide multiple, complementary, mechanisms for configuring the core infrastructure. Armani provides a core set of tools and constructs common to all Armani environments, a language for configuring the design expertise stored in the environment, and mechanisms for integrating external tools and building new tools that couldn't be represented with the configuration language. Each of these mechanisms can be used to configure and create custom design environments. If an environment developer can't attain the desired configuration using one of these techniques, he can almost always achieve the configuration using one of the other techniques. These three techniques proved appropriate and useful for the domain of custom software architecture design environments. It is not clear, however, that they are optimal configuration techniques for all possible domains. Developing additional configuration techniques and providing guidance for mapping domain characteristics to configuration techniques both provide compelling opportunities for future research.

It appears that variations on the Armani approach can be applied with significant benefit to many different software development domains. Exactly which aspects of the approach need to be modified for different domains and which can be reused directly is an open question. Along the same lines, it is not clear whether Armani's use of a rich declarative language as the basis for the bulk of the configuration information is widely applicable beyond the domain of software architecture design environments. Significant future research is needed to address these questions. Ideally, these experiments will attempt to apply the broad Armani approach to creating configurable infrastructures in other domains and report on the effectiveness of the technique.

10.2.2 Integrating the Armani toolset with full software lifecycle processes

To limit the scope of this research, I did not directly explore Armani's roles in full software lifecycle processes. Likewise, my research does not provide any definitive answers on whether or how the Armani approach changes, or should change, these processes. Mature software development organizations, however, will only be able to take full advantage of Armani if they can integrate the tool into their software development, deployment, and maintenance processes. As a result, determining how the Armani design approach can be integrated with popular development processes, as well as discovering new processes enabled by Armani's flexibility and power, are both promising directions for future work.

There are four specific research topics within this general area that are likely to be fruitful. The first of these is developing techniques for relating system requirements to architectures. Providing the ability to directly track how an architecture addresses a system's requirements is a key capability, though figuring out how to do so effectively will require a non-trivial research effort. Armani's constructs for specifying software architectures and design rules potentially provide a useful platform for mapping such a relationship. Further, Armani's ability to rapidly experiment with different environment capabilities and system design options potentially introduces an opportunity to create a tight feedback loop between the requirements gathering task and the system architecting task. The process of designing a system to meet a set of requirements frequently sheds light on the requirements themselves. An additional avenue for further research is the development of processes that can take advantage of this potential tight feedback loop.

The second research topic is establishing more effective techniques for mapping architectures to implementation code. Ideally, this mapping process will allow system designers and developers to generate significant system implementation code in the process. Although architectural specifications are valuable and important as blueprints for how a system should be built, using a manual process to go from an architectural specification to the system implementation presents (at least) two problems. The first problem is architectural drift. Because the architectural specification is just a blueprint, it is possible and even probable that the completed system will not implement the architectural specification perfectly. This problem becomes more acute as a system evolves throughout its lifetime and the architectural documents fail to keep pace. The bigger and more complex the project is, the more likely this is to be true. The second problem is that implementation still requires a tremendous amount of effort relative to the architectural specification.

One obvious technique that could address both of these issues is providing tools for generating a significant body of implementation code from the architectural specification. Generating even skeletal code constructs could both significantly speed implementation time and improve the match between architectural specifications and the code that implements those specifications. One promising approach to making such generation feasible is to link tools that know how to produce code for common components and connectors given a set of parameters by an architect. The UniCon project [Shaw+95] takes this approach with its *experts* concept. UniCon experts describe how an architectural construct should be realized in implementation code. Although beyond the scope of this dissertation, developing a technique to couple UniCon's experts with Armani's design rules is a promising avenue for adding code generation capabilities to Armani.

The third research topic explores whether Armani, and other tools that emulate its extreme flexibility, introduces opportunities to use software processes that are radically different from those popular today. Armani allows a designer to quickly change not only the design of the system that he is building, but also the tools that he is using to build that system. The research I presented in this dissertation does not solve the problem of how best to harness this capability to build software better, faster, and cheaper. The ability to adapt your tools to solve the problem at hand instead of adapting your problem to fit the tools available, however, can fundamentally change the dynamics of software development processes. Specifically, assumptions about what kinds of tasks and capabilities are expensive and which are cheap may need to be re-evaluated. Such a re-evaluation can lead to useful insights into alternative processes and techniques. The opportunities for innovative software processes become even greater as additional software development tools take Armani's lead and are made highly configurable for customization by their end-users. Providing the Armani toolset to a larger audience of practicing architects and observing how they adapt their design process as a result of using the tools is a promising next step for exploring this topic.

The fourth opportunity for future process-related research lies in devising techniques and protocols for managing architects working with Armani. The freedom that Armani provides designers in configuring their design tools and environments may or may not be desirable at the organizational level. Requiring that all designers and developers use a standard set of tools and techniques, for instance, is often a good way to insure that designers can communicate and readily share designs and specifications.

At the same time, allowing all designers to customize their design tools with their own design vocabulary, design expertise, and visualization mechanisms offers the potential for chaos. Armani's ability to quickly distribute expertise and environment specifications between designers, however, also allows designers to quickly experiment with different environment configurations and share discoveries with their collaborators. The opportunity for chaos is present, but the opportunity for rapid and organic evolution of design techniques and tools is also present. The goal is to achieve this rapid organic evolution without suffering from excessive chaos. Developing techniques, protocols, and mechanisms for tuning and maintaining the desired level of flux in environment configurations is an important area for follow-on research. One promising direction is to attempt to incorporate a standard access control policy and/or model with Armani to define the specific permissions that each member of an architecture team has.

10.2.3 Building effective design tools

In the first chapter of this dissertation I argued that design tools are effective when they capture the fundamental *design expertise* of a specific design domain. Armani builds on this assumption by providing a language for capturing design expertise and a configurable software architecture design environment that can be incrementally customized with this captured expertise. The case studies presented in chapters 6, 7, and 8 explore and demonstrate Armani's ability to capture such design expertise and to use that expertise in creating custom Armani environments. The case studies do not, however, evaluate the effectiveness of these custom environments for designing software architectures in an industrial setting.

Although conducting such a set of experiments was beyond the scope of this thesis, this issue points to an important avenue for future work. Specifically, Armani relies on the environment developers and architects who use it to configure the generic environment with useful design expertise. Developing a better understanding of how and where different kinds of expertise provide architects with leverage should improve the quality of customized Armani environments. Along the same lines, providing a set of guidelines for effectively using Armani can help new environment developers become proficient with the tool more quickly.

Armani can be easily reconfigured, external tools can be readily integrated with the environment, and its extensible infrastructure can be used to rapidly develop and deploy experimental design tools. As a result, it provides an excellent platform for exploring what makes design tools effective and experimenting with a wide variety of alternative tools and design expertise.

10.2.4 Composable connectors

The availability of pre-built connector types and connection infrastructure is often a more important design decision driver than the availability of pre-built components. Building complex connectors can be very difficult, or impossible, without access to underlying system services. It is, for example, difficult to use a shared-memory connection between processes if the underlying operating system doesn't support inter-process shared memory blocks.

Dynamic composability of connector characteristics and generation of custom connectors could significantly alleviate this problem. It is important that this composability be provided at more than the formal level (though that's a good start). The model and technology must also support the creation of an executable, run-time connection infrastructure from the formal specifications. UniCon [Shaw+95] provides a good initial cut at connector generation, but it does not support the creation of new connection mechanisms through composition.

Throughout the process of designing the Armani environment's architecture, finding an appropriate set of reusable components proved less difficult than assembling a set of connection mechanisms that worked properly with the selected components. Without appropriate connection mechanisms, however, I was unable to take advantage of many of these reusable components. It was frequently easier to duplicate the component's functionality than it would have been to integrate them using existing connector technologies. In other cases, the performance of the integrated components was simply too unreliable to be usable in the production system. In these cases, the connection mechanisms were "black boxes" that frequently and mysteriously failed to work. Debugging these failures was almost impossible because the connection mechanisms were proprietary and exposed very few details to the developer. Providing tools and techniques for generating connectors with reliable, composable, functionality and properties would go a long way towards addressing these challenges.

10.2.5 Distribution and installation of component-based systems

One of the underlying experiments in designing the Armani environment infrastructure was to exploit the current state-of-the-practice in building component-based systems. Along these lines, I selected the Microsoft Java VM as the run-time environment for the Armani infrastructure, Visio as the basis for building the Armani GUI, and ActiveX [Cha96] and JavaBeans [Ham97] technologies to connect the components and wrap external tools. I needed to distribute about a half dozen components with the full Armani solution. For even with this small group of components, however, it proved effectively impossible to create an automated installation script that could reliably install and integrate all of the components.

My difficulties in putting together the Armani distribution had both a legal component and a technical component. At the heart of the legal difficulties was the fact that none of the component vendors I selected allowed me to distribute their components with the Armani release. They wanted to maintain full control over who received these components and how they were licensed. From a business perspective, this is not an unreasonable position. An important implication of this approach, however, is that anybody who wanted to install Armani at their local site had to first visit multiple places to purchase, download, and install half a dozen different components from multiple locations before they could even attempt to install the Armani system. Providing an automated script to do the purchasing, downloading, and installation required for these packages was effectively impossible.

I suspect that if I had been a large corporation selling hundreds of thousands of copies of this software, I probably could have negotiated licensing and distribution agreements with each of these vendors that would have allowed me to distribute their components, perhaps for a fee. As a lone graduate student trying to make a research prototype freely available over the internet, however, I had neither the time, resources, nor clout to make this solution feasible. The need to conduct these lawyer-intensive negotiations severely limits the scalability of current component-based system technologies.

Although the legal difficulties with distributing the full Armani system proved daunting, the technical difficulties proved even more challenging. Each of the component vendors rapidly put out new versions of their components, often introducing incompatibilities in the process. Unfortunately, when vendors introduce a new version of a component they frequently stop supporting older versions of it, or, even worse, they stop distributing the older versions at all.[18] As a result of this rapid upgrade cycle, it is effectively impossible to keep up with all of the possible combinations of different versions of *the same components* that a customer who downloads and attempts to install Armani is likely to encounter.

Allowing system developers to distribute these components directly with their system installation package would help to address this compatibility issue, because it would allow the installer to have a consistent set of components to install. This solution solves some of the distribution problems, but it is still not perfect. If the system to be installed uses a widely available component that is shared by multiple applications, installing a new version of that component on the target machine may disrupt previously installed applications. Microsoft's COM technology [Box98] has made significant progress in addressing the multiple versions problem. Taking advantage of this solution, however, requires a developer to completely buy-in to a large collection of proprietary Microsoft standards.

My experience distributing the Armani system indicates that the current state of the practice for distributing component-based systems with components from multiple vendors does not scale to systems with six distributable components. Failure to work for a system this small indicates that current distribution and installation solutions are highly unlikely to effectively scale to support the distribution of large, heterogeneous, component-based software

[18] Fortunately, in the only case I encountered where the vendor completely stopped distributing the older version of their component (on which Armani was dependent), I was allowed to distribute the old component with the Armani distribution. If I had not been so fortunate, I would have had to make significant, costly upgrades to Armani just to keep it available.

systems. Interestingly, this problem is not limited to large, complex, software systems. The state-of-the-practice techniques and infrastructure for distributing component-based systems are also problematic for small, inexpensive software systems produced by development organizations that lack the resources to negotiate with large component vendors.

Developing configuration management and distribution systems that address both technical and legal issues is a critical area for future research. Without dramatic improvements in this area, the optimistic projections of component-based software evangelists are unlikely to come to fruition and software development organizations are unlikely to be able to take significant advantage of component-based technologies.

10.2.6 Selecting appropriate styles and design expertise

Armani provides software architects and design environment builders with a great deal of flexibility for expressing, capturing, and reusing design expertise and architectural styles. It is not, however, particularly effective at guiding architects faced with a specific design problem in selecting appropriate architectural styles or collections of design expertise. This process still requires significant taste and judgement on the part of the architect.

Researchers working in other specialties have made progress on the general problem of mapping common design problems to understood solutions (to cite three examples, see the work in the design patterns community such as [Gam+95], Lane's thesis on user-interface development [Lan90], and Kruegger's work on selecting object-oriented database designs [Kru97]). There has, however, been little progress on this problem in the software architecture research community. Although solving this problem is outside of the scope of this thesis, experience with Armani underscores its importance as an area for future work.

10.3 Conclusion

In this dissertation I have demonstrated that it is possible to capture software architecture design expertise in small, reusable, and incrementally composable units. I've also shown that it is possible to build a configurable software architecture design environment that can be incrementally customized to support a wide variety of architectural styles by simply loading these design expertise units into the environment.

The case studies presented in this dissertation illustrate selected ways that this technique and technology can be used to inexpensively and quickly provide software architects with highly customized tools. The true power of Armani, however, remains to be discovered by the architects and environment developers who will use it to harness and leverage their creative skills and to create great software.

The mark of a great tool, it has been said, is its use in ways that its creator never imagined. Hopefully those who use Armani will use it not only as I've described in this dissertation but also as a springboard for creating powerful new design tools and techniques.

Chapter 11
Bibliography

[AAG95] Gregory Abowd, Robert Allen, and David Garlan, Formalizing Style to Understand Descriptions of Software Architecture. ACM Transactions on Software Engineering and Methodology, 4(4):319-64, October 1995.

[Ale+77] Christopher Alexander et al, A Pattern Language, Oxford University Press, 1977.

[All+98] Formal Modeling and Analysis of the HLA Component Integration Standard, Robert J. Allen, David Garlan, and James Ivers, *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), November 1998).*

[ATT93] Best Current Practices: Software Architecture Validation. AT&T Corp., 1993.

[Autodesk] AutoDesk Corporation website, www.autodesk.com.

[Ber92] Alex Berson, Client/Server Architecture, McGraw-Hill, New York, 1992.

[Be90] Laurence J. Best, Modern Large-Scale Information Processing, John Wiley & Sons, Inc., New York, 1990.

[Bus+96] Frank Buschmann, Regine Meunier (Contributor), Hans Rohnert (Contributor), Peter Sommerlad, Pattern Oriented Software Architecture : A System of Patterns, John Wiley and Sons, 1996.

[Boo94] Grady Booch, Object-Oriented Analysis and Design With Applications. Addison-Wesley, ISBN: 0805353402, 1994.

[Box98] Don Box, Essential COM, Addison-Wesley Pub Co; ISBN: 0201634465, 1998.

[BRJ98] Grady Booch, Jim Rumbaugh, and Ivar Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, ISBN: 0201571684, 1998.

[Bro95] Fred Brooks, The Mythical Man Month: Essays on Software Engineering, 25th Anniversary edition. Addison-Wesley, ISBN: 0201835959, July 1995.

[Cadence] Cadence Corporation website, www.cadence.com.

[Cha96] David Chappell, Understanding ActiveX and Ole, Microsoft Press; ISBN: 1572312165, 1996.

[Corel] Corel Corporation website, www.corel.com.

[DD97] Hugh Darwen and Chris J. Date, A Guide to the SQL Standard : A User's Guide to the Standard Database Language SQL. Addison-Wesley Pub Co; ISBN: 0201964260, April 1997.

[End72] Herbert B. Enderton. A Mathematical Introduction to Logic. Academic Press, 1972.

[Fen+94] Steven Fenves et al, Concurrent Computer-Integrated Building Design, Prentice Hall, Englewood Cliffs, N.J., 1994.

[Fis87] Gerhard Fischer, A Critic For Lisp, University of Colorado Technical Report CS-CU-365-87, June 1987.

[Fis+87] Gerhard Fischer and Andreas C. Lemke, Construction and Design Kits: Human Problem-Domain Communication, University of Colorado Technical Report CS-CU-366-87, June 1987.

[For81] Charles L. Forgy, "OPS5 User's Manual", Technical Report CMU-CS-81-135, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 1981.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Pub Co; ISBN: 0201633612, 1995.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom, Exploiting Style in Architectural Design Environments. In Proceedings of SIGSOFT '94: Foundations of Software Engineering, ACM Press, December 1994.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom, Architectural Mismatch, or, Why it's hard to build systems out of existing parts, in *Proceedings of the 17th International Conference on Software Engineering*, April 1995.

[Gar95] David Garlan, editor. Proceedings of the First International Workshop on Architectures for Software Systems. April, 1995.

[Gia93] Joseph C. Giarratano, CLIPS Users Guide, Available from the Software Technology Branch of the NASA Johnson Space Center. 1993. Available at: http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/expert/systems/clips/0.html.

[GMW97] David Garlan, Robert Monroe, and Dave Wile, Acme: An Architecture Description Interchange Language, Proceedings of CASCON '97, November 1997.

[Hab+82] Nico Habermann et al, The Second Compendium of Gandalf Documentation, Department of Computer Science, Carnegie Mellon University, May 1982.

[Ham97] Graham Hamilton (editor), Java Beans version 1.01 specification, available from http://www.javasoft.com/beans, July 1997.

[HFB94] Dan Heller, Paula M. Ferguson, and David Brennan, Motif Programming Manual (The Definitive Guide to the X Window System, Volume 6A). O'Reilly & Associates; ISBN: 1565920163, 1994.

[JCC99] JavaCC web site, http://www.sun.com/suntest/products/JavaCC/ .

[Jac94] Ivar Jacobson, Object-Oriented Software Engineering : A Use Case Driven Approach, Addison-Wesley, ISBN: 0201544350 , 1994

[KC98] Paul Kogut and Richard Creps, CORBA-Aware Environments, Internal Lockheed Martin report, available by personal request to paul.kogut@lmco.com.

[Kai85] Gail E. Kaiser, Semantics for Structure Editing Environments, Doctoral Dissertation, Carnegie Mellon University, 1985.

[Key99] Key Software Corporation website: www.keysoft.com.

[Kom99] Andrew Kompanek, AcmeStudio web site:
http://www.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html

[Kru97] Charles Krueger, Modeling and Simulating a Software Architecture Design Space, Ph.D. Thesis, Carnegie Mellon University Technical Report CMU-CS-97-158, December 1997.

[Lan90] Thomas Lane, User Interface Software Structures, Ph.D. Thesis, Carnegie Mellon University Technical Report CMU-CS-90-101, May 1990.

[Law87] Shari Lawrence Pfleeger, Software Engineering: The Production of Quality Software, Macmillan Publishing Co, New York, 1987.

[LM98] Lockheed Martin Corporation, Evolution Plan for the Global Transportation Network, Delivery 2/3. Publication USTCP 171-8.2, Lockheed Martin Corporation, March 1998.

[McM93] K. L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.

[MG92] Mettala, E. and Graham, M.. The Domain-Specific Software Architecture Program. Special Report, Carnegie Mellon University Software Engineering Institute CMU/SEI-92-SR-9. 1992.

[Min91] Naftaly Minsky, Law Governed Systems, The IEE Software Engineering Journal, September, 1991.

[Min96-1] Naftaly Minsky, Law-Governed Regularities in Object Systems; part 1: Principles. Theory and Practice of Object Systems (TOPAS), John Wiley, Vol. II, No. 4, 1996.

[Min96-2] Naftaly Minsky, Independent On-line Monitoring of Evolving Systems. *Proceedings of the 18th International Conference on Software Engineering*, pages 134-143, March 1996.

[Mon+97] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan, "Architectural Styles, Design Patterns, and Objects", IEEE Software, January 1997.

[MQR94] Mark Moriconi, Xiaolei Qian, and R.A. Riemenschneider. A formal approach to correct refinement of software architectures, : Technical report. SRI International. Computer Science Laboratory ; SRI-CSL-94-05, April 1994.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-Based Runtime Software Evolution. *Proceedings of the 1998 International Conference on Software Engineering (ICSE '98).* Kyoto, Japan, April 19-25, 1998.

[OHE97] Robert Orfali, Dan Harkey (Contributor), and Jeri Edwards (Contributor), The Essential Client/Server Survival Guide, John Wiley & Sons; ISBN: 0471153257, 1997.

[OS97] Sam Owre and Natarajan Shankar, The Formal Semantics of PVS. Technical Report CSL-97-2, SRI International, Menlo Park, CA, August 1997.

[Per89] Dewayne E. Perry. The Inscape Environment. *Proceedings of the Eleventh International Conference on Software Engineering*, Pittsburgh PA, May 1989.

[Pfl87] Shari Lawrence Pfleeger, Software Engineering: The Production of Quality Software, Macmillan Publishing Co, New York, 1987.

[PW92] Dewayne E. Perry and Alexander Wolf, Foundations for the Study of Software Architecture. ACM Software Engineering Notes, 17(4), October 1992, pp. 40-52.

[Qua98] Terry Quatrani, Visual Modeling With Rational Rose and Uml, Addison-Wesley, ISBN: 0201310163, January 1998.

[Rational] Rational Corporation website, www.rational.com.

[Rec91] Eberhardt Rechtin, Systems Architecting: Creating and Building Complex Systems, Prentice Hall, 1991.

[RT88] Thomas W. Reps and Tim Teitelbaum, The Synthesizer Generator: A System for Constructing Language-Based Editors, Springer Verlag, 1988.

[RHR98] Jason E. Robbins, David M. Hilbert, and David F. Redmiles., Software Architecture Critics in Argo. *Proc. of The 1998 International Conference on Intelligent User Interfaces (IUI'98)*, January 1998.

[Riv+96] Jose German Rivera, Alejandro Andres Danylyszyn, Charles B. Weinstock, Lui R. Sha, Michael J. Gagliardi, An Architectural Description of the Simplex Architecture, CMU Software Engineering Institute Technical Report CMU/SEI-96-TR-006, 1996.

[RM97] Eberhardt Rechtin and Mark Maier, The Art of Systems Architecting, CRC Press, Boca Raton, FL, 1997.

[Rob+98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, David S. Rosenblum.Integrating Architecture Description Languages with a Standard Design Method. Proceedings of the 18th International Conference on Software Engineering, 1998.

[Rum+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, Bill Lorensen. Object-Oriented Modeling and Design, Prentice Hall; ISBN: 0136298419, 1991.

[Shaw+95] Mary Shaw, Robert Deline, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions. on Software Engineering, April 1995.

[SC97] Mary Shaw and Paul Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. Proceedings of COMPSAC97, 1st Intternational Computer Software and Applications Conference, August 1997.

[SG96] Mary Shaw and David Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[SG98] Bridget Spitznagel and David Garlan, Architecture-Based Performance Analysis, *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, June 1998.

[SGW94] Bran Selic, Garth Gullekson, Paul T. Ward, Real-Time Object-Oriented Modeling, John Wiley & Sons, New York, 1994.

[Sha96] L. Sha, R. Rajkumar, and M. Gagliardi, Evolving Dependable Real Time Systems, *Proceedings of the 1996 IEEE Aerospace Applications Conference*. Aspen, CO, February 3-10, 1996. New York, NY: IEEE Computer Society Press, 1996.

[SO98] John Schettino and Liz O'Hara, CORBA for Dummies, IDG Books Worldwide; ISBN: 0764503081, October 1998.

[SVK93] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of Dynamically Reconfigurable Real-time Software Using Port-Base Objects. Technical Report CMU-RI-TR-93-11. Carnegie Mellon University Robotics Institute, July 1993.

[Tay+96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, pages 390-406, June 1996.

[Ves94] Steve Vestal, MetaH Reference Manual, Honeywell Technology Center, Minneapolis MN, 1994.

[Visio99] Visio Corporation home page. www.visio.com.

[WMK98] Franklin Webber, Joseph R. McEnerney, and Kevin Kwiat, The DesignExpert Approach to Developing Fault-Tolerant and Secure Systems. *Proceedings of the ISSAT International Conference on Reliability and Quality in Design*, 1998.

[WS97] Charles Weinstock and Lui Sha, Simplex overview website, http://www.sei.cmu.edu/activities/str/descriptions/simplex_body.html

[YC79] Edward Yourdon and Larry L. Constantine, Structured design : fundamentals of a discipline of computer program and systems design, Prentice Hall, 1979.

Appendix A
Armani Design Language BNF

BNF Meta-Syntax

Keywords are specified with bold text. Keywords are case-insensitive

Non-Terminals are specified with italics

(...)	Parentheses group tokens and productions
[...]	Indicates an optional production
(...)?	Indicates a sequence of zero or one elements (synonymous with [])
(...)+	Sequence of one or more elements
(...)*	Sequence of zero or more elements
|	Seperates alternative choices

Armani Grammar

```
ArmaniDesign              ::= ( TypeDeclaration
                                | FamilyDeclaration
                                | DesignAnalysisDeclaration )*
                              [ SystemDeclaration ]
                              <EOF>
```

Design Element Types:

```
FamilyDeclaration         ::= Family Identifier [ "(" ")" ] "=" FamilyBody [ ";" ]

FamilyBody                ::= "{" ( TypeDeclaration )* "}"

TypeDeclaration           ::= ElementTypeDeclaration | PropertyTypeDeclaration

ElementTypeDeclaration    ::=   ComponentTypeDeclaration
                                | ConnectorTypeDeclaration
                                | PortTypeDeclaration
                                | RoleTypeDeclaration

ComponentTypeDeclaration ::= Component Type Identifier "="
                                parse_ComponentDescription  [ ";" ]
                                |
                                            Component Type Identifier Extends
                                            Identifier ( "," Identifier )*
                                            With parse_ComponentDescription [ ";" ]

ConnectorTypeDeclaration  ::= Connector Type Identifier "="
                                  parse_ConnectorDescription [ ";" ]
                             |
                               Connector Type Identifier Extends
                               Identifier ( "," Identifier )*
                               With parse_ConnectorDescription [ ";" ]

PortTypeDeclaration     ::=  Port Type Identifier "=" parse_PortDescription [ ";" ]
                             | Port Type Identifier Extends Identifier ( "," Identifier )*
                               With parse_PortDescription [ ";" ]

RoleTypeDeclaration     ::=  Role Type Identifier "=" parse_RoleDescription [ ";" ]
                             | Role Type Identifier Extends Identifier ( "," Identifier )*
                               with parse_RoleDescription [ ";" ]

lookup_ComponentTypeByName     ::= Identifier

lookup_ConnectorTypeByName     ::= Identifier

lookup_PortTypeByName          ::= Identifier

lookup_RoleTypeByName          ::= Identifier

lookup_PropertyTypeByName      ::= Identifier
```

Design Elements:

```
SystemDeclaration       ::=  System Identifier ( ":" Identifier )? "=" systemBody [ ";" ]

SystemBody              ::=  ( New lookup_ComponentTypeByName  |
                               "{"
                               ( ComponentDeclaration | ComponentsBlock
                                 |  ConnectorDeclaration | ConnectorsBlock
                                 | PortDeclaration | PortsBlock | RoleDeclaration
                                 | RolesBlock | PropertyDeclaration | PropertiesBlock
                                 | AttachmentsDeclaration | RepresentationDeclaration
                                 | DesignRule
                               )*
                               "}"
                             )
                             [ Extended With SystemBody ]

ComponentDeclaration ::=  Component Identifier
                          [ ":" lookup_ComponentTypeByName ]
                          ( "=" parse_ComponentDescription ";" | ";" )

ComponentsBlock         ::=  Components "{"
                             ( Identifier
                               [ ":" lookup_ComponentTypeByName ]
```

```
                                        ( "=" parse_ComponentDescription ";" | ";" )
                                      )*
                                      "}" [ ";" ]

parse_ComponentDescription          ::= ( New lookup_ComponentTypeByName
                                          |
                                          "{"  ( PortDeclaration | PortsBlock
                                                       | PropertyDeclaration
                                                 | PropertiesBlock
                                                       | RepresentationDeclaration
                                                       | DesignRule )*
                                          "}"
                                          )
                                          [ Extended With parse_ComponentDescription ]

ConnectorDeclaration    ::= Connector Identifier
                            [ ":" lookup_ConnectorTypeByName ]
                            ( "=" parse_ConnectorDescription ";" | ";" )

ConnectorsBlock         ::= Connectors "{"
                            ( Identifier
                              [ ":" lookup_ConnectorTypeByName ]
                              ( "=" parse_ConnectorDescription ";" | ";" ) )*
                            "}" [ ";" ]

parse_ConnectorDescription ::=  ( New lookup_ConnectorTypeByName
                                  |
                                  "{" ( RoleDeclaration
                                        | RolesBlock
                                        | PropertyDeclaration
                                        | PropertiesBlock
                                        | RepresentationDeclaration
                                        | DesignRule )*
                                  "}"
                                  )
                                  [ Extended With parse_ConnectorDescription ]

PortDeclaration         ::= Port Identifier
                            [ ":" lookup_PortTypeByName ]
                            ( "=" parse_PortDescription ";" | ";" )

PortsBlock              ::= Ports "{"
                            ( Identifier
                              [ ":" lookup_PortTypeByName ]
                              ( "=" parse_PortDescription ";" | ";" ) )*
                             "}" [ ";" ]

parse_PortDescription   ::= ( New lookup_PortTypeByName
                              |
                              "{" ( PropertyDeclaration | PropertiesBlock
                                    | RepresentationDeclaration | DesignRule )*
                              "}"
                            )
                            [ Extended With parse_PortDescription ]
```

```
RoleDeclaration          ::= Role Identifier
                             [ ":" lookup_RoleTypeByName ]
                             ( "=" parse_RoleDescription ";" | ";" )

RolesBlock               ::= Roles "{"
                             ( Identifier
                               [ ":" lookup_RoleTypeByName ]
                               ( "=" parse_RoleDescription ";" | ";" ) )*
                             "}" [ ";" ]

parse_RoleDescription    ::= ( New lookup_RoleTypeByName
                              | "{" ( PropertyDeclaration | PropertiesBlock |
                                      RepresentationDeclaration | DesignRule )*
                                "}" )
                             [ Extended with parse_RoleDescription ]

AttachmentsDeclaration ::= [ Identifier "=" ]
                             Attachments "{"
                             ( Identifier "." Identifier to Identifier "." Identifier
                             [ "{" ( PropertyDeclaration | PropertiesBlock )* "}" ]
                             ";" )*
                             "}" ";"
```

Properties:

```
PropertyDeclaration      ::= Property parse_PropertyDescription ";"

PropertiesBlock          ::= Properties "{"
                              [ parse_PropertyDescription
                                ( ";" parse_PropertyDescription | ";" )*
                              ]
                             "}" [ ";" ]

parse_PropertyDescription  ::=      [ Property ] Identifier
                             ":" PropertyTypeDescription
                             [ "=" PropertyValueDeclaration ]
                             [  "<<" parse_PropertyDescription
                                 ( ";" parse_PropertyDescription | ";" )*
                                ">>"
                              |
                                "<<" ">>"
                             ]

PropertyTypeDeclaration    ::=      Property Type Identifier
                             ( ";"
                              |
                               "=" ( Int ";" | Long ";" | Double ";" | Float ";"
                                    |String ";" | Boolean ";" | Any ";"
                                    |Enum [ "{" Identifier ( "," Identifier )* "}" ]  ";"
                                    |Set [ "{" "}" ] ";"
                                    |Set "{" PropertyTypeDescription "}" ";"
                                    |Sequence [ "<" ">" ] ";"
                                    |Sequence "<" PropertyTypeDescription ">" ";"
```

```
                                        |Record "[" parse_RecordFieldDescription
                                          ( ";" parse_RecordFieldDescription | ";" )* "]" ";"
                                        |Record [ "[" "]" ] ";"
                                        |Identifier ";"
                                      )
                                   )

PropertyTypeDescription        ::=    Int | Long | Float | Double | String
                                   | Boolean | Any
                                   | Set [ "{" [ PropertyTypeDescription ] "}" ]
                                   | Sequence [ "<" [ PropertyTypeDescription ] ">" ]
                                   | Record "[" parse_RecordFieldDescription
                                          ( ";" parse_RecordFieldDescription | ";" )* "]"
                                   | Record [ "[" "]" ]
                                   | Enum [ "{" Identifier ( "," Identifier )* "}" ]
                                   | Enum [ "{" "}" ]
                                   | Identifier

parse_RecordFieldDescription          ::=    Identifier ( "," Identifier )*
                                             [ ":" PropertyTypeDescription ]

PropertyValueDeclaration       ::=    Integer_Literal | Floating_Point_Literal |
                                   String_Literal | False | True | AcmeSetValue |
                                   AcmeSequenceValue | AcmeRecordValue | Identifier

AcmeSetValue               ::=    "{" "}"
                                   | "{" PropertyValueDeclaration
                                         ( "," PropertyValueDeclaration )* "}"

AcmeSequenceValue          ::=  "<" ">"|
                                   "<" PropertyValueDeclaration
                                         ( "," PropertyValueDeclaration )* ">"

AcmeRecordValue            ::=  "[" RecordFieldValue ( ";" RecordFieldValue | ";" )* "]"

RecordFieldValue           ::=  Identifier ":" PropertyTypeDescription "="
                                   PropertyValueDeclaration
```

Representations and Bindings:

```
RepresentationDeclaration  ::=    Representation "{"
                                   SystemDeclaration
                                   [ BindingsMapDeclaration ]
                                   "}" [ ";" ]

BindingsMapDeclaration     ::=    Bindings "=" "{" ( BindingDeclaration )* "}" [ ";" ]

BindingDeclaration         ::=  [ Identifier "." ] Identifier to
                                   [ Identifier "." ] Identifier
                                   [ "{" ( PropertyDeclaration | PropertiesBlock )* "}" ] ";"
```

Design Rules and Analyses:

```
DesignRule               ::= ( Design )? ( Invariant | Heuristic )
                             DesignRuleExpression ";"

DesignRuleExpression     ::= QuantifiedExpression | BooleanExpression

QuantifiedExpression     ::= ( forall | exists) Identifier ":"
                             lookup_arbitraryTypeByName in
                             SetExpression "|" DesignRuleExpression

BooleanExpression        ::= OrExpression ( and OrExpression )*

OrExpression             ::= ImpliesExpression ( or ImpliesExpression )*

ImpliesExpression        ::= IffExpression ( "->" IffExpression )*

IffExpression            ::= EqualityExpression ( "<->" EqualityExpression )*

EqualityExpression       ::= RelationalExpression ( "==" RelationalExpression
                             | "!=" RelationalExpression )*

RelationalExpression     ::= AdditiveExpression
                             ( "<" AdditiveExpression | ">" AdditiveExpression
                              | "<=" AdditiveExpression | "=>" AdditiveExpression )*

AdditiveExpression       ::= MultiplicativeExpression
                             ( "+" MultiplicativeExpression
                              | "-" MultiplicativeExpression )*

MultiplicativeExpression ::= UnaryExpression
                             ( "*" UnaryExpression
                               | "/" UnaryExpression
                               | "%" UnaryExpression )*

UnaryExpression          ::= "!" UnaryExpression
                             | "-" UnaryExpression
                             | PrimitiveExpression

PrimitiveExpression      ::= "(" DesignRuleExpression ")"
                             | LiteralConstant | DesignAnalysisCall | Id

Id                       ::= Identifier ( "." Identifier )*

DesignAnalysisCall       ::= Id "(" ActualParams ")"

LiteralConstant          ::= IntegerLiteral | FloatingPointLiteral | StringLiteral
                             | true | false

ActualParams             ::= ( ActualParam ( "," ActualParam )* )?

FormalParams             ::= ( FormalParam ( "," FormalParam )* )?

ActualParam              ::= LiteralConstant | DesignAnalysisCall | Id
```

```
FormalParam        ::= Identifier ( "," Identifier )* ":"
                       (Identifier | Component | Connector | Port | Role
                        | Int | Float | String | Boolean )

SetExpression      ::= ( SetReference | SetFunction | LiteralSet
                        | SetConstructor )

SetReference       ::= Identifier ( ( "." Identifier ) | ( "." Components )
                        | ( "." Connectors ) | ( "." Ports ) | ( "." Roles )
                        | ( "." Representations ) | ( "." Properties ) )+

SetFunction        ::= ( Union | Intersection | Setdiff )
                       "(" SetExpression "," SetExpression ")"

LiteralSet         ::= ( "{" "}" |
                       "{" ( LiteralConstant | Id ) ( "," ( LiteralConstant | Id ) )*
                       "}")

SetConstructor     ::= "{" Select Identifier ":" lookup_arbitraryTypeByName in
                       SetExpression "|" DesignRuleExpression "}"
```