Experience with a Course on Architectures for Software Systems

David Garlan[1], Mary Shaw[1], Chris Okasaki[1],
Curtis M. Scott[1], and Roy F. Swonger[2]

[1] School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 14213 ***

[2] Digital Equipment Corporation
Nashua, NH 03062

Abstract. As software systems grow in size and complexity their design problem extends beyond algorithms and data structures to issues of system design. This area receives little or no treatment in existing computer science curricula. Although courses about specific systems are usually available, there is no systematic treatment of the organizations used to assemble components into systems. These issues – the *software architecture* level of software design – are the subject of a new course that we taught for the first time in Spring 1992. This paper describes the motivation for the course, the content and structure of the current version, and our plans for improving the next version.

1 Overview

The software component of the typical undergraduate curriculum emphasizes algorithms and data structures. Although courses on compilers, operating systems, or databases are usually offered, there is no systematic treatment of the organization of modules into systems, or of the concepts and techniques at an architectural level of software design. Thus, system issues are seriously underrepresented in current undergraduate programs. Further, students now face a large gap between lower-level courses, in which they learn programming techniques, and upper-level project courses, in which they are expected to design more significant systems. Without knowing the alternatives and criteria that distinguish good architectural choices, the already-challenging task of defining an appropriate architecture becomes formidable.

We have developed a course that will help to bridge this gap: *Architectures for Software Systems*. Specifically, the course:

*** Development of this course was funded in part by the Department of Defense Advanced Research Project Agency under grant MDA972-92-J-1002. It was also funded in part by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense) and Siemens Corporate Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government, the Department of Defense, Carnegie Mellon University, or Siemens.

- teaches how to understand and evaluate designs of existing software systems from an architectural perspective,
- provides the intellectual building blocks for designing new systems in principled ways using well-understood architectural paradigms,
- shows how formal notations and models can be used to characterize and reason about a system design, and
- presents concrete examples of actual system architectures that can serve as models for new designs.

This course adds innovative material to existing curricula on the subject of software architectures. It also helps define the field of software architecture by organizing and regularizing the concepts and by enabling the education of software designers in those concepts.

2 Background and Rationale

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; composition of design elements; scaling and performance; and selection among design alternatives.

This is the *software architecture* level of design. There is a considerable body of work on this topic, including module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, and formal models of component integration mechanisms. However, there is not currently a consistent terminology to characterize the common elements of these fields. Instead, many architectural structures are described in terms of idiomatic patterns that have emerged informally over time. For example, typical descriptions of software architectures include statements such as:

- "Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers." [S+87].
- "Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a client-server model for the structuring of applications." [FO85]
- "We have chosen a distributed, object-oriented approach to managing information." [Lin87]
- "The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors. ... A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by multiple compiler processes] before a final, merging pass recombines the object code into a single program." [S+88]

Other software architectures are carefully documented and often widely disseminated. Examples include the International Standard Organization's Open Systems Interconnection Reference Model (a layered network architecture) [Pau85], the NIST CASEE Reference Model (a generic software engineering environment architecture) [Ear90], and the X Window System (a windowed user interface architecture) [SG86].

It is increasingly clear that effective software engineering requires facility in architectural software design. First, it is important to be able to recognize common paradigms so that high-level relationships among systems can be understood and so that new systems can be built as variations on old systems. Second, detailed understanding of software architectures allows the engineer to make principled choices among design alternatives. Third, an architectural system description is often essential to the analysis and description of the high-level properties of a complex system. Fourth, fluency in the use of formal notations for describing architectural paradigms allows the software engineer to communicate new systems designs to others.

Regrettably, software architectures receive little or no systematic treatment in most existing software engineering curricula, either undergraduate or graduate. At best, students are exposed to one or two specific application architectures (such as for a compiler or for parts of an operating system) and may hear about a few other architectural paradigms, but no serious attempt is made to develop comprehensive skills for understanding existing architectures and developing new ones. This results in a serious gap in current curricula: students are expected to learn how to design complex systems without the requisite intellectual tools for doing so effectively.

We have developed a course to bridge this gap. This course brings together the emerging models for software architectures and the best of current practice. It examines how to approach systems from an architectural point of view. Other curriculum proposals have touched on this subject, but to our knowledge this is the first implementation of a full course in the area.

3 Philosophy and Course Overview

3.1 Objectives

We designed a course for senior undergraduates and students in a professional master's program for software engineering. By the end of this course, students should be able to:

- Recognize major architectural styles in existing software systems.
- Describe an architecture accurately.
- Generate reasonable architectural alternatives for a problem and choose among them.
- Construct a medium-sized software system that satisfies an architectural specification.
- Use existing definitions and development tools to expedite such tasks.
- Understand the formal definition of a number of architectures and be able to reason precisely about the properties of those architectures.
- Understand how to use domain knowledge to specialize an architecture for a particular family of applications.

3.2 Approach

We believe that important skills for designing complex systems can be provided by a course that examines systems from an architectural point of view. Specifically, our course considers commonly-used software system structures, techniques for designing and implementing these structures, models and formal notations for characterizing and reasoning about architectures, tools for generating specific instances of an architecture, and case studies of actual system architectures. It teaches the skills and background students need to evaluate the underlying architecture of existing systems and to design new systems in principled ways using well-founded architectural paradigms.

Since this is an entirely new course rather than a modification of an existing course, the major challenge in its development was to define and delimit its intellectual content. While the ability to recognize and use software architectures is essential for the practicing software engineer, there is to date no codified body of knowledge that deals specifically with this subject. Rather, relevant material is scattered over published case studies, standards reports, formal models, informal system documentation, and anecdotal experience. We have collected many of these sources, distilled them into a corpus of presentable knowledge, and discovered ways to make that knowledge directly usable by university students and the software engineering community at large.

Our approach focuses on developing four specific, related topic areas:

Classification: In order to use software architectures, it is first necessary to be able to recognize an architectural style and to describe a system in terms of its architecture. The tools required to describe and categorize common architectural models include notations for defining architectures and a taxonomy of existing models. In addition to introducing the student to these tools, this topic addresses the problem of architectural selection to solve a given software engineering problem. It covers both high-level architectural idioms (e.g., pipeline architectures) and specific reference models (e.g., the OSI layered model).

Analysis: Effective use of a software architecture depends on the ability to understand and reason about its properties (such as functional behavior, performance, developmental flexibility, evolvability, and real-time behavior). Such analysis can be applied to many kinds of architectural description, but it is particularly effective in the context of formal descriptions, where the power of mathematics can be exploited. This topic therefore covers techniques for analyzing an architecture. It introduces students to formal and informal methods and illustrates the ways in which formal analysis can be used to evaluate and select among architectural alternatives [Fi87] [GD90].

Tools: Certain architectures have evolved to the point where there is system support for defining applications using them and for executing those applications once they are built. Examples include Unix support for single-stream pipeline architectures, compilers for module interconnection languages (such as Ada package specifications), and IDL (Interface Description Language) readers and writers for shared data. Facility with such tools is a valuable skill for using the supported architectures in the context of current technology. Moreover, existing tools provide good illustrations of the kinds of automated support that we can expect to become pervasive as the field becomes more fully developed and populated with useful architectures.

Domain-Specific Architectures: Specific knowledge about an application domain can improve the power of the notations and tools for constructing systems in that domain. The same holds true for architectures, and there is active research and industrial development in the area of domain-specific software architectures [DSS90]. The course looks at a number of these to understand how domain knowledge can be exploited in designing an architecture tailored to a specific application family.

We rely heavily on case studies in each of these topic areas. These are used to motivate the importance and scope of architectural approaches, illustrate what has been done so far, and give students models for creating architectural descriptions of their own. In addition to examining existing case studies, students are expected to carry out a significant case study of their own. By doing this they practice applying the techniques of architectural description and analysis and contribute to the field by adding to the body of carefully documented architectural descriptions.

4 Course Description

In this section, we give an overview of each topic covered in the course. This information is summarized in Figure 1. Each row of the figure contains the lecture number, the major topic and subtopic covered in the lecture (as described below), the reading which the student is to have completed prior to attending the lecture, and the homework assignment (if any) to be discussed or turned in on that date. The assignments are numbered A1 through A4, with the course project due at the end of the semester. These are discussed in sections 4.3 through 4.5.

Introduction (2 lectures)

- *Orientation.* What is the architectural level of software design, and how does it differ from intra-module programming? Overview of the course.
- *What is a Software Architecture?* Constructing systems from modules. Some familiar kinds of architectures. Some common kinds of modules. [Sha90b, DK76, PW91]

Architectural Idioms (5 lectures)

- *Objects.* Information hiding, abstract data types, and objects. Organizing systems by encapsulating design decisions, or "keeping secrets." [PCW85, Boo86, WBJ90]
- *Pipes & Events.* Two architectural idioms: pipes and event systems. Pipes support a dataflow model. Event systems support loosely-coupled components interacting via event broadcast. [Par72, GKN88]
- *Multi-process Systems.* Organizing systems as collections of independent computations that run cooperatively on one or many processors. [And91]

Lecture	Topic	Subtopic	Reading	Assignment
1	Introduction	Orientation		
2		What is a SW Arch?	[Sha90b, DK76, PW91]	
3	Architectural	Objects	[PCW85, Boo86, WBJ90]	
4	Idioms	Pipes & Events	[Par72, GKN88]	
5		Multi-process Systems	[And91]	
6		Blackboards	[Nii86a, Nii86b]	
7		Heterogeneous Design	[Sha90a, Sha91]	A1 discuss
8	MILs	Classical MILS	[PDN86, LS79]	
9		Unix Pipes	[Bac86]	A1 due
10		SML & Ada	[H^{+}88, I^{+}83]	
11		Augmentations	[Per87, Gro91]	A2 discuss
12	Formal Models	Intro to Z	[Spi89b, Sha85]	
13		Industrial Experience	[GD90, HK91]	A2 due
14		Executable Specs	[Zav91]	
15		Event Systems	[GN91]	A3 discuss
16		Pipes and Filters	[AG92]	
17	Domain-Specific	Data Processing	[Fis91, RC86]	A3 due
18	Architectures	Distd, Heterogeneous	[BWW88, D^{+}91]	
19		Real-Time	[SG90, Sta88]	
20		Robotics	[HR90, SST86]	
21		Communication	[Tan81]	A4 discuss
22	Tools & Envts	Hints on Syst Design	[Lam84]	
23		Design Guidance	[Lan90]	A4 due
24		Arch. Transformers	[Bis87, BAP87]	
25		System Generators	[LS86, Joh86, BO91]	
26		Envt. Generators	[HGN91]	Proj due
27	Student Project			
28	Presentations			
29				

Fig. 1. Summary of Course Topics

- *Blackboards.* Sharing complex knowledge about a problem; making progress when you can't tell in advance what order to impose on the subproblems. [Nii86a, Nii86b]
- *Heterogeneous Design.* Designers don't have to limit themselves to a single architectural idiom. Examples of systems that use several idioms at various places in the system. [Sha90a, Sha91]

Module Interconnection Languages (4 lectures)

- *Classical MILS.* Historically, the earliest large systems were developed in procedural languages. The most common of the MILs reflect this in their emphasis on importing and exporting names of procedures, variables, and a few other constructs. [PDN86, LS79]
- *Unix Pipes.* The Unix paradigm connects independent processes by data flow.

The organization of the processes and the style and tools for connection are substantially different. [Bac86]
- *Module Interconnection in Standard ML and Ada.* An important property of modern module interconnection languages is the ability to parameterize modules. This is represented by generics in Ada and functors in SML. [H+88, I+83]
- *Augmentations to Module Interfaces.* Future prospects for module interconnection. How to augment a module's interface so that it conveys more than signatures. [Per87, Gro91]

Formal Models of Software Architecture (5 lectures)

- *Introduction to Z.* Basic notation of the Z Specification Language. The schema calculus. [Spi89b, Sha85]
- *Industrial Experience with Formal Models.* Use of formal models to understand, document, and analyze system architectures in two major industrial case studies. [GD90, HK91]
- *Paisley.* Executable specification language that supports some elementary performance analysis. [Zav91]
- *Event Systems.* Formal model of event systems. Specialization of abstract formal models to describe specific systems. [GN91]
- *Pipes and Filters.* Abstract model of pipes and filters. Use of formalism to explain what a software architecture is and to analyze its properties. [AG92]

Domain-Specific Architectures (5 lectures)

- *Data Processing.* Architectures for management information systems. [Fis91, RC86]
- *Distributed, Heterogeneous Computing.* Applied pipe and filter architectures. Architectures to support flexible processor allocation and reconfiguration. [BWW88, D+91]
- *Real-Time System Architectures.* Real-time schedulers: rate-monotonic scheduling, cyclic executives, and others. Conditions under which a particular real-time architecture can be applied. [SG90, Sta88]
- *Architectures for Mobile Robotics.* Software organization of reactor-effector systems that operate in an uncertain environment. The CMU task control architecture. [HR90, SST86]
- *Layered Architectures for Communication.* Network protocols based on layered model of communication abstractions. Special emphasis on ISO Open System Interconnection (OSI) standard. [Tan81]

Tools, Environments, and Automated Design Guidance (5 lectures)

- *Hints on System Design.* Sage guidance and rules of thumb about designing good systems. [Lam84]
- *Automated Design Guidance.* The selection of a software architecture should depend on the requirements of the application. This example of a system shows how to make the structural design of a user interface explicitly dependent on the functional requirements. [Lan90]

– *Architecture Transformers.* Semi-automatic conversion of the uniprocessor version of a system to a multiprocesor version; not fully general, but works under clearly stated conditions. [Bis87, BAP87]
– *System Generators.* Automatic production of certain classes of systems from their specifications. [LS86, Joh86, BO91]
– *Environment Generators.* Automatic production of environments from descriptions of the tasks to be performed. [HGN91]

5 Assignments

5.1 Purpose

The purpose of the assignments, as in any course, is to help students master the material. Assignments serve the additional purpose of demonstrating the students' mastery of the material, thereby establishing a basis for evaluation.

Students begin by examining and understanding existing work in the area. Then they apply what they've seen and heard, first by trying to emulate it and then by performing analysis. Three kinds of assignments lead students through these activities.

First, the course is organized around written papers and lectures that present and interpret this material. We believe that the lectures are most useful if they provide interpretation, explanation, and additional elaboration of material students have already read and thought about. In addition to assigning readings, we provide guidance about the important points to read for and questions to help students focus on the most significant points in the reading.

Second, four two-week assignments ask students to apply the lecture material. Three of these assignments require students to develop small software systems in specific architectural styles. The fourth is a formal analysis task, which allows the students to work with a specific architectural formalism.

Third, students examine existing software systems to determine their architectures. We identified several systems of about 20 modules. For the final project of the course, each student team analyzed the actual system structure of one of these and interpreted the designer's architectural intentions.

We organized the students into teams of two (with one team of three because an odd number of students enrolled). This encouraged students to enhance their understanding through discussions with another student, reduced the amount of overhead required of any one student to get to the meat of a problem, and allowed us to partially compensate for differences in programming language and other related experience. The course included both undergraduate students and students in the Master of Software Engineering program; to the extent possible we paired undergraduates with graduates so that their experience would complement each other.

Since students often tend to spend most of their attention and energy on the components of a course that contribute to the final grade, we used the allocation of credit as a device to focus them on the most important activities. To this end, we included four factors in the grading basis. Here is the description of these factors as stated in the initial course handout:

- *Readings: (25%)* Each lecture will be accompanied by one or more readings, which we expect you to read *before* you come to class. To help you focus your thoughts on the main points of the reading we will assign a question to be answered for each of the reading assignments. Each question should be addressed in less than a page, due at the beginning of the class for which it is assigned. Each of these will be evaluated on a simple ok/not-ok basis and will count for about 1% of your grade.
- *Homework Assignments: (40%)* There will be four homework assignments. Each will count 10% of your grade. The first three will be system-building exercises. Their purpose is to give you some experience using architectures to design and implement real systems. You will work in groups of two (assigned by us) to carry out each assignment. To help clarify your designs we will hold a brief, un-graded design review for each assignment during class a week before it is due. Groups will take turns presenting their preliminary designs and getting feedback from the class and instructors. The fourth assignment will give you some practice using formal models of software architectures.
- *Project: (25%)* There will be a course project, designed to give you some experience with the architecture of a substantial software system. You will analyze an existing software system from an architectural point of view, document your analysis, and present the results to the rest of the class. Your grade will depend both on the quality of your analysis and also on the presentation of that analysis.
- *Instructors' judgement: (10%)*

5.2 Readings

No textbook exists for this course. Background material for the course consisted of readings, primarily from professional journals, selected to complement the lectures and discussions. The objective was for every student to read each paper before the corresponding class lecture.

To ensure this, a short homework assignment was set for each class. Each homework consisted of a few questions to be answered about the readings. These assignments were due at the beginning of the corresponding lecture and discussion. Though the single grade attached to a particular assignment would not significantly affect the course grade, the cumulative effect of these individual grades resulted in significant weight being placed on the readings.

A beneficial side effect of this policy was that it obviated the need for examinations. The incremental learning process was monitored and reinforced by the assignments, so there was no need for a final exam to measure student progress. As a result, end-of-semester energy could be productively directed to the course project.

Each reading was accompanied by hints which identified points to look for in each paper and gave advice on parts to ignore. These hints helped students to focus on the important concepts in each paper, and were particularly important because of the wide variety of notations and languages introduced in the readings.

Here are some examples of the hints we gave:

- In these readings you will be exposed to many different languages. You should not try to learn the specific syntax of each language, nor should you memorize

the specific features of each language. Rather, you should try to get a feel for the design space of module interconnection languages—what it is possible to represent and what it is desirable to represent.
- First and foremost, read to understand the blackboard model and the kinds of problems for which it is appropriate. Study Hearsay and HASP to see how the model is realized in two rather different settings. Look at the other examples to see the range of variability available within the basic framework.

The questions for each assignment also played an important role in focusing the intellectual energies of the students. The questions were structured to have the students understand the concepts involved, rather than simply read to complete the homework. Since the reading and homework combined were intended to take only a couple of hours, the questions dealt with major points and did not require deep thought or analysis.

Here are some examples of the questions we asked:

- What are the essential differences between the architectural style advocated by Parnas and that advocated by Garlan, Kaiser, and Notkin?
- What abstract data type does a pipe implement? What common implementation of that abstract data type is used to implement pipes?
- What is the problem addressed by sharing specifications in SML? Why doesn't this come up with Ada generics?
- What are the major abstractions of an interconnection model? How are these specialized in the unit and syntactic models?

5.3 Architectural Development Tasks

Believing that one must constructively engage a style to understand it, we assigned programming tasks in three different architectural idioms. For each task, we supplied an implementation in the required idiom that used several components from an available collection. The assignment required students to extend the implementation *in the same style* by reconnecting parts, using other components, or minimally changing components. The choice of this format was driven by two guiding principles:

- The attention of the students should be focused at the architectural level rather than at the algorithms-and-data-structures level. (Students should already know how to do the latter.)
- It is unreasonable to expect the accurate use of an unfamiliar idiom without providing illustrative sample code employing that idiom.

A pleasant side-effect of this choice of format was that problems more closely resembled software maintenance/reuse than building a system from scratch. In addition, we were faced with a considerable diversity of programming language background among the students. It's easier to work in an unfamiliar language if you have a working starting point.

To encourage cooperation and to balance unfamiliarity with particular programming languages and systems, students worked in pairs on the programming tasks. However, each task had a set of questions to be answered individually.

A major objective of this course is for students to leave with an understanding of the essential features of a given problem that make a particular architectural choice appropriate or inappropriate. To do this, we assigned variations of a single core problem for all three tasks, differing primarily in the features related to the choice of idiom. By assigning the same basic problem for each architectural idiom, we avoided the risk of students associating problem class X with architectural idiom Y, instead promoting understanding of the features of each problem that should lead the designer to choose that idiom. By varying the features related to the architectural choice, we also discouraged students from leaving each solution in the same basic architectural idiom, adding only the superficial trappings of the second idiom. For example, by changing the requirements on the system, we ensured that an event-driven solution would not merely be a pipes & filters solution "dressed up" to look like an event-driven system.

Because the problems involved not only the production of a working system but also the analysis of an architectural style, we held design reviews halfway through each assignment. These reviews were presented by the students in the class, with each team making one presentation sometime during the semester. The reviews were not graded; they thereby provided a means for the class to engage in discussions about the architectural style and for the instructors to guide the student solutions (both those being presented and those of the students watching the presentation) by asking pointed questions. These presentations were performed during class time, and their schedule is presented in Figure 1.

The core task chosen was the KWIC indexing problem[Par72, GKN88]. In this problem, a set of lines (sequences of words) is extended to include all circular shifts of each line, and the resulting extended set is alphabetized. This core problem was varied in each architectural idiom as follows:

Object-Oriented: This variation was *interactive*: a user enters lines one at a time, interspersed with requests for the KWIC index. Students were supplied with a system which generated the KWIC index without the circular shifts (i.e., a line alphabetizer) and asked to include the shifts. In addition, students were asked to omit lines which began with a "trivial" word (e.g., *and* or *the*).

Pipes & Filters: In this variation, students were asked to generate a batch version which generated KWIC indices of login and user names (as generated by the *finger* command). Students carried out two tasks. In one task, students used the Unix shell to connect "modules" such as the common Unix commands *finger*, *sort*, and *uniq*. A second task required them to connect the same modules in a pipe organization too complex to describe in the shell, so that they had to use raw pipes from within C. As before, they began with solutions which alphabetized lines but did not generate circular shifts.

Event-driven (implicit invocation): This variation extended the problem for the object-oriented architecture with a *delete* command. Students were required to reuse existing modules, augmenting them with event bindings to establish how they communicated.

5.4 Formal Modelling

To develop skill in understanding and manipulating formal models we assigned a task that required students to extend an existing formal model of a software architecture.

As with the architectural development tasks the formal modelling task builds on an existing base—in this case the formal model developed by Garlan and Notkin of event systems [GN91]. In this work the authors showed how a simple model of systems based on event broadcast could be specialized for a number of common systems, including Smalltalk MVC, Gandalf programming environments, the Field programming environment, and APPL/A.

The students were asked to perform similar specializations for two different architectures: spreadsheets and blackboard systems. In addition, they were asked to provide a commentary that answered the following questions:

1. What important aspects of the modelled architectures are (intentionally) left out of the model?
2. For the blackboard system, would it be possible to model some notion of "non-interference"?
3. For the spreadsheet system, is the *Circular* property defined in the events paper a useful concept? Why or why not?
4. Based on the formal models, briefly compare each of the two new systems with the other ones that were formally modelled. For example, you might explain which of the other systems are they most similar to.

5.5 Analysis and Interpretation of a System

In addition to the assignments described above, students also examined and described the architecture of a non-trivial system. About midway through the semester we asked each group of students to select a system from a list of candidates that we supplied. Alternatively, students could volunteer a system of their own, provided it met the criteria outlined below.

The students' task was to complete an architectural analysis of the chosen system by the end of the semester. This analysis was required to include the following components:

1. **Parts catalog:** A list of the modules in the system, making the interfaces explicit, together with an explanation of what each one does.
2. **Interconnections catalog:** A list of the connections between modules together with a descriptions of each.
3. **Architectural description:** A description of the system's architecture, using the vocabulary developed in the course.
4. **Critique:** An evaluation of how well the architectural documentation for the system matches the actual implementation.
5. **Revision:** Suggestions for ways that the system architecture could have been improved.

In addition to a written analysis students were expected to present their analysis to the class. We allotted three days at the end of the semester for this. The grade on the project was determined both by the written analysis and the presentation.

In selecting candidate systems we attempted to find systems that are tractable but challenging. Specifically, we applied the following criteria for selection:

- **Size:** Ten to twenty modules containing between 2,000 and 10,000 lines of code.
- **Documentation:** It should have enough system documentation that students do not need to start from raw code to do their analysis.
- **Resident guru:** There should be someone in the local environment who knows the system and can answer questions about its design and implementation.

6 Evaluation

6.1 Lessons from the Initial Offering

The first offering of the course ran during the spring semester of the 1991-1992 academic year at Carnegie Mellon University. Four undergraduate students and seven Master of Software Engineering students took the course. There were also half a dozen regular auditors. The lessons we have learned as a result of that offering are based upon the students' progress in learning the course material and on evaluation of the course by the students themselves.

Content. It is clear that a sufficiently large body of knowledge exists to support a course in software architecture. When we designed this version we were unable to include all the topics of interest, and we made some hard decisions among alternative materials for the topics we did include.

The specific topics of the course had varying success. The section on architectural idioms was particularly valuable. While the section on MILs was important to have included in the course, we now feel we spent too much time on that topic. Two lectures would have been more appropriate than the four that we scheduled. The formal methods segment worked out well, but for students with no exposure to Z, it required considerably more effort than the other sections. (Almost all of the master's level students had already had a course in formal methods.) The lectures on domain-specific software architectures were of mixed value, since these lectures were predominantly given by invited speakers. The section on tools and environments was reasonably successful, but suffered from the fact that there is relatively little material directly applicable to software architectures.

Many of the readings were quite good; others should be replaced. The best of the readings included Nii's survey of Blackboard Systems [Nii86a, Nii86b], Andrews' survey of distributed architectures [And91], Shaw's overview of architectural styles [Sha90a], Parnas' classic "Criteria" [Par72] and A7 papers [PCW85], the Perry Inscape paper [Per87], a paper on implicit invocation by Garlan, Kaiser, and Notkin [GKN88], Lampson's hints on system design [Lam84], and Lane's paper on the concept of the design space [Lan90]. The course syllabus would have been much better if we had been able to find good readings about Unix pipes, management information system architectures, the ISO Open Systems Interconnection model, and architectural tools. We would also like to find readings about object-oriented systems that deal specifically with architectural issues, rather than programming issues.

The assignments that involved construction and analysis of systems were generally quite valuable, as they gave students practice in applying the principles of the course. However, we felt that we could have chosen tasks that would have both challenged the students more than we did, and at the same time focused on some of the more important issues of architectural design.

Chosing good practical problems is one of the most difficult (and important) parts of developing such a course. Part of the problem centers around finding systems that are of the right size and complexity. On the one hand, it is important to find systems that are large enough to represent nontrivial architectures. On the other hand, there is a limit to the size of system students that can handle, particularly given the fact that we wanted students to have experience with several architectures.

Our approach to the problem was to give students a working system and a collection of parts that they could reuse and modify in adapting the system to the task assigned. Overall this was a good approach, although it takes a lot of preparation to make it successful. We attempted to use Booch components [Boo87] for the first and third assignments, and the standard Unix tools for the second assignment. However, many of the software needed in the solutions to the problems could not be found in these standard collections. As a result a majority of the code in the starting frameworks was developed by us from scratch. For example we used only one Booch component package (a total of 200 lines), but wrote 1400 lines of Ada and 300 lines of C for the frameworks provided to the students.

Another aspect of the problem of chosing good assignments is to find problems that exploit the target architecture but do not have a single solution. Our assignments were not sufficiently rich to accomplish this. Moreover, time constraints prevented us from making more parts available than were absolutely required for the assignments, so the students did not face the challenge of selecting an architectural solution from a rich parts kit.

One issue involving the assignments was the use of multiple programming languages. We wanted to avoid giving the impression that there is a one-to-one mapping between programming languages and architectures. However, our parts kit (the Booch components) was in Ada, and our version of the Ada compiler did not provide a library for manipulating Unix pipes. This led us to use Ada for the first and third assignments, and C for the second.

Format. The division of the course into the major topics had some advantages, but overall was viewed as needing revision. The course lectures touch each of the architectural idioms three times: to introduce the idiom, to examine a suitable module connection language or tool, and to introduce a formal model and analysis technique useful within the idiom. In addition, some idioms reappear in domain-specific examples. In retrospect, a course organization that factors the course along the lines of architectural idioms seems more appropriate.

We were pleased to teach from selected readings. The readings allowed students to hear the ideas in the voices of their creators. Moreover, we believe the state of of knowledge in software architectures is not advanced enough to provide a single canonical picture of the various architectures. Further, we were able to order the topics appropriately to meet our ideas of how the topics should be presented, and to flexibly schedule the guest lecturers.

The chief disadvantages of using readings as source material are that notations and terminology vary from paper to paper, and that the architectural significance of a paper may not be well articulated. It is also a nuisance to deal with copyright considerations (though many of the papers carry blanket permissions for educational use).

Assigning questions on the class readings was a good idea. It served both to focus students' attention and to encourage them to do the reading in advance. This worked well enough that we could plan lectures that elaborated and interpreted the material rather than repeating it.

The task formats worked well in terms of the amount of time allotted and overall structure of the assignments. However, one challenge lay in the diversity of language experience among the students. In particular, Ada was familiar to some students, but new to others. While this course was not intended to be a "programming course," we found it necessary to provide a brief Ada help session for students without Ada experience. This session's effectiveness was limited because it was held outside normal class hours and was not directly graded. Ideally in an architectures course, however, all students should know the programming languages to be used for the assignments prior to entering the class.

Using groups to accomplish the assignments had generally positive results. By mixing graduate and undergraduate students on each team, the teams formed a balanced collection of strengths which enabled them to grasp the essentials of the programming assignments quickly. Also, having a team environment allowed the students to discuss the architectural issues among themselves in a more structured format than might otherwise have been available. We believe this would have been even more effective if a more challenging set of problems were presented.

Comments From Students. We distributed two course evaluations to students, one midway through the course and one at the end. Overall student responses were quite positive. Typical comments were: "I've noticed that I'm viewing problems in other classes from a different perspective," and "I now finally understand what we were doing when we built that system in the way we did."

At a more detailed level the students felt that the study of architectural idioms was most important, and that it provided a foundation for a body of knowledge to which they had not been exposed. They also said that the course encouraged a new perspective on software systems. Some students wanted more emphasis on the process of architectural design and guidance in choosing an architectural idioms. The general opinion was that more could be added to the course, but not at the expense of current material.

The readings were generally viewed as a valuable part of the course. Students appreciated the incentive to read them regularly, and they particularly appreciated the absence of a final exam. They also liked having lectures serve to elaborate the readings rather than repeating them—something that is only possible when the instructor can assume that students have actually read the readings.

The course required a significant amount of work, but the students thought it was worth the effort. They noted that, unlike most courses, the load is fairly level.

They had to pay attention to the course regularly, but they didn't wind up with massive deadline crunches.

The team organization was also judged favorably. We received no complaints about unequal workload within a team, although some team members commented that getting a consensus, even on a small team, took time.

6.2 Changes to Consider

While we are generally satisfied with the course as taught, we see some areas that we plan to change the next time we teach the course.

Content. While the course did not suffer from a lack of material, a number of areas could be added to the curriculum. In particular, study of heterogeneous and integrated architectural idioms would be appropriate, as would more study and practice in architectural design and decision making. One way that room could be made for this material is by reducing the time spent on module interconnection languages.

Another improvement would be to develop a more consistent terminology in our lectures on architectures. Within the area generally termed software architecture, there is a bewildering diversity of terms for similar concepts. As the discipline evolves and the terminology stabilizes, we would expect this problem to diminish.

Format. There are a number of format changes which we believe would improve the coherency and conceptual integrity of the course. They are:

1. Concentrate on a particular idiom for a span of 3-4 lectures to give the students a deeper understanding of each class of architecture. A consistent format for the study of each idiom might be:
   (a) Introduction of idiom and survey of class
   (b) Related languages or tools
   (c) Formal model
   (d) Case study and analysis
   This could be complemented by an assignment for each idiom, to give a single, coherent presentation of an architectural style.
2. The assignments could be better planned to emphasize the architectural nature of the projects, and minimize the "hacking" necessary to build a system. Assignments should allow the students to create unsuccessful solutions as well as different but successful solutions to the same problem. More emphasis should be placed on analysis of the assignment, encouraging students to reflect on the choices made and the reasons for these choices.

   The only way we see to create reasonable projects is to provide collections of ready-made components. Unfortunately, reasonable "parts kits" are scarce, and most existing parts kits do not clearly illustrate an architectural style. Moreover, in many cases (such as in event systems), architectural styles require tools beyond those provided even by a carefully constructed parts kit. This infrastructure is likely to be different for each architectural style, compounding the problem when multiple architectural styles are presented. Generally, in order to effectively teach multiple architectural styles within the constraints of a single-semester course, we need architectural tools as well as parts kits.

3. The architectural analysis project should use examples that are familiar to the instructors and which have existing architectural documents for the students to start with.
4. The reports that the students produce from the architectural analysis project should emphasize the high-level structural and communications paradigms of the system, rather than specific functionality or detailed algorithmic analysis. To this end, the assignment of the project should specify the structure of the report, and provide good examples of existing architectural analyses.

6.3 Conclusions About Teaching Software Architecture

Software architecture is worth teaching. It can be taught in many ways. Based on our experience with the present course and previous experience with four semesters of graduate reading seminars in the area, we can draw some conclusions about teaching software architecture in any format.

- Architecture provides a bridge between theory and coding. In any program teaching system design, there are high principles of program construction which are difficult to relate to the small programming assignments that comprise the majority of the undergraduate experience. A course that presents students with the terminology of software architecture and that gives them concrete examples of systems to relate to specific architectural styles allows the students to relate these two disparate bodies of information more readily and concretely.
- Students seem capable of rapidly developing an aesthetic about architectures. They can identify systems in their own experience which match specific styles, and they can also identify flawed designs as examples of poorly-formed or poorly-understood architectures. They are quite capable of answering open-ended questions about the appropriateness of a specific architecture to a problem and defending their positions rationally and powerfully. Unity does not evolve among the students, however. Different students will promote different architectures for the same problem, depending upon their particular points of view.
- There is little concrete material available in any form to guide design decisions. Absent such material, students get little help in resolving point-of-view differences. Instructors should make every effort to present techniques for selecting among architectural alternatives, including even simple rules of thumb such as "consider an interpreter when you're designing for a machine that doesn't actually exist."
- There is enough substantive material to fill a course. The selection we made for this offering was based on the coverage of our graduate reading seminars. However, we recognize that there were a number of difficult choices in our selection which might well have gone another way. In our opinion, the field of software architectures is moving from a point where finding enough papers is difficult to one where the challenge is to select the appropriate complement of papers.

– We wish there were more organized surveys of the material than are currently present. Currently, the fragmented nature of the material requires that the students be carefully instructed on exactly which information within a given paper is appropriate to the subject at hand. This is compounded by the sheer size and disorganization of the current software architectures field. There are few papers which view problems from a purely architectural perspective, and the boundaries between architectural idioms are not always clear. We would like to see more papers presenting architectural analysis techniques, and more worked examples in specific architectures. We would also like to see more mature distributed systems architectures and more papers like Nii's [Nii86a, Nii86b] that survey a class of systems against a single architectural paradigm. We think this will come with a better understanding of the idioms that comprise software architecture.
– It is tempting to treat the subject of software architectures abstractly and present only idealized views of the various architectural idioms. Resist this. Students have weak intuitions about the high-level architectural abstractions. Every formal or abstract model must be related to a real example, so that the student not only learns the abstract view of the architecture, but also the characteristics of a concrete instance of that architecture.
– Practice in using models is important. Analyzing existing architectures without working within the specific architectural framework does not allow the student to recognize the strengths and weaknesses of individual architectural styles. It is not sufficient for a student to be able to recognize a specific idiom; the student must also be able to decide which idiom to apply to a particular problem. For that skill, analysis alone is not enough.

Acknowledgments

The authors would like to thank Robert Allen, Mario Barbacci, Marc Graham, Kevin Jeffay, Dan Klein, Reid Simmons, and Pamela Zave for participating as guest lecturers in the first offering of this course. We would also like to thank James Alstad for his participation in the development of the early curriculum of this course.

References

[AG92] Robert Allen and David Garlan. A formal approach to software architectures. Submitted for publication, January 1992.

[And91] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*, chapter 5.12, pages 111–119. Software Series. Prentice-Hall, 1986.

[BAP87] J. M. Bishop, S. R. Adams, and D. J. Pritchard. Distributing concurrent Ada programs by source translation. *Software—Practice and Experience*, 17(12):859–884, December 1987.

[Bis87] Judy M. Bishop. Ada profile charts in software development. *Journal of Pascal, Ada and Modula-2*, 8(2), October 1987.

[BO91] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems using reusable components. Technical Report TR-91-22, Department of Computer Science, University of Texas, Austin, June 1991.

[Boo86] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[Boo87] Grady Booch. *Software Components with Ada: Structures, Tools and Subsystems*. Benjamin/Cummings, Menlo Park, CA, 1987.

[BWW88] M. R. Barbacci, C. B. Weinstock, and J. M. Wing. Programming at the processor-memory-switch level. In *Proceedings of the 10th International Conference on Software Engineering*, pages 19–28, Singapore, April 1988. IEEE Computer Society Press.

[D+91] Doubleday et al. Building distributed Ada applications from specifications and functional components. In *Proceedings of TRI-Ada'91*, pages 143–154, San Jose, CA, October 1991. ACM Press.

[DK76] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[DSS90] *Proceedings of the Workshop on Domain-Specific Software Architectures*, July 1990.

[Ear90] Anthony Earl. A reference model for computer assisted software engineering environment frameworks. Technical Report HPL-SEG-TN-90-11, Hewlett Packard Laboratories, Bristol, England, August 1990.

[Fi87] Bill Flinn and ib Holm Sorensen. *CAVIAR: A Case Study in Specification*. Prentice Hall International, 1987.

[Fis91] Gary Fisher. Application portability profile -APP- The U.S. Government's open system environment profile. US Department of Commerce, April 1991. National Technical Information Service Special Report, 500-187.

[FO85] Marek Fridrich and William Older. Helix: The architecture of the XMS distributed file system. *IEEE Software*, 2(3):21–29, May 1985.

[GD90] David Garlan and Norman Delisle. Formal specifications as reusable frameworks. In *VDM'90: VDM and Z – Formal Methods in Software Development*, Kiel, Germany, 1990. Springer-Verlag, LNCS 428.

[GKN88] David Garlan, Gail E. Kaiser, and David Notkin. On the criteria to be used in composing tools into systems. Technical Report 88-08-09, Department of Computer Science, University of Washington, August 1988.

[GN91] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44. Springer-Verlag, LNCS 551, October 1991.

[Gro91] MIF Working Group. Master: A prototech module interconnection formalism. Draft of December 1991, 1991.

[H+88] Robert Harper et al. Introduction to Standard ML. Technical report, Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, March 1988.

[HGN91] A. Nico Habermann, David Garlan, and David Notkin. Generation of integrated task-specific software environments. In Richard F. Rashid, editor, *CMU Computer Science: A 25th Commemorative*, Anthology Series, pages 69–98. ACM Press, 1991.

[HK91] Iain Houston and Steve King. Experiences and results from the use of Z in IBM. In *VDM'91: Formal Software Development Methods*, number 551 in Lecture Notes in Computer Science, pages 588–595. Springer-Verlag, October 1991.

[HR90] Barbara Hayes-Roth. Architectural foundations for real-time performance in intelligent agents. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 2:99–125, 1990.

[I+83] J. D. Ichbiah et al. Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(16 (Part B)):8:1–16, 13:1–21, June 1983. Chapters 8 (Modules) and 13 (Generic Program Units).

[Joh86] Stephen C. Johnson. YACC: yet another compiler-compiler. In *UNIX Programmer's Supplementary Documents*, volume PS1, pages 15:1–33. University of California, Berkeley, 1986.

[Lam84] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–28, January 1984.

[Lan90] Thomas G. Lane. A design space and design rules for user interface software architecture. Technical Report CMU/SEI-90-TR-22 ESD-90-TR-223, Carnegie Mellon University, Software Engineering Institute, November 1990.

[Lin87] Mark A. Linton. Distributed management of a software database. *IEEE Software*, 4(6):70–76, November 1987.

[LS79] Hugh C. Lauer and Edwin H. Satterthwaite. Impact of MESA on system design. In *Proceedings of the Third International Conference on Software Engineering*, pages 174–175. IEEE Computer Society Press, May 1979.

[LS86] M. E. Lesk and E. Schmidt. LEX—a lexical analyzer generator. In *UNIX Programmer's Supplementary Documents*, volume PS1, pages 16:1–13. University of California, Berkeley, 1986.

[Nii86a] H. Penny Nii. Blackboard systems Part 1: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(3):38–53, Summer 1986. Reprinted with corrections by AI Magazine.

[Nii86b] H. Penny Nii. Blackboard systems Part 2: Blackboard application systems and a knowledge engineering perspective. *AI Magazine*, 7(4):82–107, August 1986. Reprinted with corrections by AI Magazine.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Pau85] Mark C. Paulk. The arc network: A case study. *IEEE Software*, 2(3):61–69, May 1985.

[PCW85] David L. Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

[PDN86] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.

[Per87] Dewayne E. Perry. Software interconnection models. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 61–68, Monterey, CA, March 1987. IEEE Computer Society Press.

[PW91] Dewayne E. Perry and Alexander L. Wolf. Software architecture. Submitted for publication, January 1991.

[RC86] Sridhar A. Raghavan and Donald R. Chand. Applications generators & fourth generation languages. Technical Report TR-86-02, Wang Institute and Bentley College, February 1986.

[S+87] Alfred Z. Spector et al. Camelot: A distributed transaction facility for Mach and the Internet - an interim report. Technical Report CMU-CS-87-129, Carnegie Mellon University, June 1987.

[S+88] V. Seshadri et al. Semantic analysis in a concurrent compiler. In *Proceedings of ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.* ACM SIGPLAN Notices, 1988.

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[SG90] Lui Sha and John B. Goodenough. Real-time scheduling theory and Ada. *Computer*, pages 53–62, April 1990.

[Sha85] Mary Shaw. What can we specify? Questions in the domains of software specifications. In *Proceedings of the Third International Workshop on Software Specification and Design*, pages 214–215. IEEE Computer Society Press, August 1985.

[Sha90a] Mary Shaw. Elements of a design language for software architecture. Unpublished position paper, 1990.

[Sha90b] Mary Shaw. Toward higher-level abstractions for software systems. In *Data & Knowledge Engineering*, volume 5, pages 119–128. Elsevier Science Publishers B.V., North Holland, 1990.

[Sha91] Mary Shaw. Heterogeneous design idioms for software architecture. In *Proceedings of the Sixth International Workshop on Software Specification and Design, IEEE Computer Society, Software Engineering Notes*, pages 158–165, Como, Italy, October 25-26 1991.

[Spi88] J. Michael Spivey. *The Fuzz Manual.* Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 1988.

[Spi89a] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

[Spi89b] J.M. Spivey. An introduction to Z and formal specification. *Software Engineering Journal*, 4(1):40–50, January 1989.

[SST86] Steven A. Shafer, Anthony Stentz, and Charles E. Thorpe. An architecture for sensor fursion in a mobile robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2002–2010, San Franciso, CA, April 1986.

[Sta88] John A. Stankovic. Misconceptions about real-time computing. *Computer*, Vol.21(10):10–19, October 1988.

[Tan81] Andrew S. Tannenbaum. Network protocols. *ACM Computing Surveys*, 13(4):453–489, December 1981.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.

[Zav91] Pamela Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):212–225, March 1991.