On Teaching Software Architecture and Design

William Hankley
Kansas State University
Manhattan, KS 66506
hankley@cis.ksu.edu

Abstract

This paper describes a sophomore/junior level course on software architecture and design. The course covers general principles of system requirements and conceptual models, not just "programming skills". Key components of the course include use of UML (Unified Modeling Language) for description of system models, use of CASE (computer aided software engineering) tools for software modeling and development, GUI (graphical user interface) design, case studies of several kinds of software applications, writing skills, and programming assignments. Example systems include typical business data management software, soft real-time control of devices, direct manipulation visual models, and distributed computing. Programming is done primarily using Java, with some work using C++. With the foundation of object oriented structure, there is little difficulty in using the equivalent subset of C++. The course is offered within a track of software engineering (within a computer science/ information systems curricula); however, the course is also taken by electrical and computer engineers and some students from other engineering disciplines. The course is prerequisite to our capstone project course on software engineering. The architecture course does not cover general software engineering topics (such as testing, estimation, and management), but it does provide students in the capstone course with foundation skills for software design and development. This type of architecture and design course should be an appropriate second level course for engineers with a minor emphasis of software.

Introduction

This paper addresses the underlying issue of where and how and why architecture and design fit into the curriculum of courses for majors in computing and for service computing courses, which are often taken by engineering and science majors. Historically, the core courses of computing have dealt with programming, algorithms, and data structures, but very little with architecture and design. This pattern is examined in the next section.

What is software architecture? Within the scope of the course we have developed, the focus is on object-oriented software. Other paradigms for software design and development are discussed but not treated in depth. The implementation language for our core courses is Java, but that is not an essential aspect of the issues of this paper. Within this context, the architecture of a program consists of the selection of user operations, the user interface for operations, and the structure and behavior of program "components". An architecture is documented by various models, primarily visual models, and supporting narrative. Models represent different aspects of the architecture and different levels of abstract, such as domain models vs. implementation models. Many (but not all) of these models are defined by the Unified Modeling Language (UML) [19]. The treatment of these models within our course is presented in the course organization section. To achieve a more-than-novice understanding of architecture, it is not enough to just know about UML models; students must also have a familiarity with a variety of common domain models, common reusable "parts", and common patterns, and they must be able to understand the relationships between the various models.

To design (as a verb) is the process of building the models that comprise a software architecture. The resulting models themselves are also referred to as a design (noun). In the initial steps of creating domain models, the difficult task is in moving from understanding of the domain to the insight of creation of models. The best foundation for that task is familiarity with a range of models and an apprentice-like experience of building models and receiving constructive evaluation. Integral to completion of software design is the use of CASE tools, specifically, a rapid GUI (graphic user interface) builder and a complete UML builder tool. We use JBuilder and Together Control Center. The second difficulty of teaching software design is in checking and evaluating software models. Current tools provide little capability to check correctness of a model or consistency of the component models. Hence, a major focus of the course is on checking of models, either by peer groups or by the instructor and an assistant. Techniques for such checking are discussed in the course description.

The motivation in presenting our course on architecture and design is the conviction that these topics should be a core part of the preparatory curriculum for students majoring in computing and for those students from other disciplines who take selected courses in "programming". My observation is that once students master the elements of a programming language, most difficulties with programming are actually difficulties in forming the architecture, that is, difficulties in defining the program functionality and in defining and organizing the program "parts". All students who learn some programming should also learn about software architecture, because the architecture models are the most effective means by users, designers, and programmers can communicate requirements about software.

Background

What was originally the ACM model curriculum for computer science [1,2] and is now the ACM / IEEE guides to computing curricula [3,4] have been and remain the dominant models for structuring computing education, particularly in the United States. Originally, the single model was for computing science (as opposed to applied computing). The core courses were CS1 Programming, CS2 Algorithms and Data Structures, and Analysis of Algorithms. The 1991 guidelines defined core topic areas which would be incorporated into various courses. It identified several possible curricula, including computer science but also including "computer science with software engineering emphasis". It introduced a course on software engineering that dealt with a breadth of software engineering topics, but did not focus just on software architecture and design.

During 1990's decade, our curriculum, like many others, had a capstone software engineering and senior project course. Over the years it became clear that students were learning design issues while they were building their senior project; they did not have a foundation of design experience to bring to bear on their project work.

The latest ACM/IEEE guidelines, currently in draft form [4], also define areas of the body of knowledge for degrees relating to computing. Section 7, Introductory Courses, discusses the option of programming early vs. programming delayed. For this paper, we assume beginning courses deal with programming. The section presents three distinct approaches, starting with imperative languages, object-oriented, or breadth first. Obviously, this paper deals with the object-oriented paradigm. Section 7.6.2 of the guidelines presents two options for objects-first programming, a traditional programming, data structures, and algorithms sequence that emphasizes experience in building data structures vs. a shorter sequence of programming and object methodology that emphasizes use of standard data structures. Among intermediate level courses, the draft defines the course CS290r Software Development [6] that includes event-driven programming (4 hrs), human-computer aspects and requirements (18 hrs), software design (2 hr), standard "parts" and environments (5 hrs), and other topics (6 hrs). Among the courses in the draft, that course is closest in focus to the presentation of this paper, but it has much more emphasis on human-computer aspects rather than architecture and design. The course presented herein is suggested as an alternative version of the CS290r course.

One concern about existing programming courses is expressed in Section 7 of the guidelines [5]:

> "Introductory programming courses often oversimplify the programming process to make it accessible to beginning students, giving too little weight to design, analysis, and testing relative to the conceptually simpler process of coding. Thus, the superficial impression students take from their mastery of programming skills masks fundamental shortcomings that will limit their ability to adapt to different kinds of problems and problem-solving contexts in the future."

In this vein, I believe it is incomplete to teach design without the accompanying study of architecture. In that case, it is striking that software architecture is not even a core subject in the curriculum guidelines. In the body of knowledge organization, software architecture is one sub-item, without supporting details, within the software design item of the software engineering area.

One impetus for recognizing software architecture as a core part of the body of knowledge of computing arises in the work of Garland and Shaw and others at Carnegie Mellon University [13, 14]. They have presented a model course on software architecture [12]. That course focuses on what I call architecture paradigms, such as data flow models, a blackboard model, rule-based model, etc. I incorporated material from the CMU course into the initial version of our architecture course. We concluded the focus of CMU material without supporting laboratory work was too abstract and too difficult for beginning level students. Students needed first more experience with object-oriented models that they could relate to programs in the language they knew.

During the same time period of the 1990's, UML (Unified Modeling Language) emerged as the common visual notation for representing software models. Many introductory programming texts and data structure texts now show at least the class diagram model of UML. Yet, such texts do not give much emphasis to the design process of building software models.

Course Particulars

The course Software Architecture is offered each fall and spring semester. It is the third required course in the programming sequence, with programming and data structures and algorithms as prerequisites. It is required for majors in Computer Science, Information Systems, and Computer Engineering. A few graduate students take the course as preparation for the Master of Software Engineering program. Enrollment is around eighty students per semester. A graduate assistant helps with in-class evaluations, with grading, and with consulting.

The learning goals for the course are the achievement of the following:

(1) knowledge of concepts of architecture models (UML and extensions)
(2) experience with a breadth of architecture types
(3) experience with a breadth of reusable parts and patterns
(4) skill in building abstract software models
(5) skill in using standard design parts within CASE tools
(6) proficient completion of an integrated project of design and implementation.

The learning units to achieve the goals are shown in Figure 1 and an example breakdown of class hours is shown in Table 1. The specific content has varied as the course has evolved. The class is taught as two 75 minute sessions per week, so that a class "hour" is nominally 50 minutes. As in most courses, periods start with some administrative items (such as discussing assignments and returning papers), so the effective work "hour" may be 45 minutes. As indicated in Figure 1, the learning units are covered in parallel tracks, often one day for concepts and abstract models and the other day for implementation topics (during the first part of the semester) and project evaluation (during the second part of the semester). In Table 1, the notes in the "joint with" column, indicate that topics are not covered as separate items, but are combined. Typically, presentation of details of one of the UML models (such as sequence diagrams) is knit together with discussion of one of the application domain examples (such as the a library example of a rental system) and that is followed by a class exercise where in students construct another instance of the UML model. In the assignments column, "P" represents a programming assignment and "H" represents assignments requiring writing and/or UML models. The project incorporates writing, UML models, and programming.

The course starts with a dialog about building architecture, design and construction and about the education for architects and architectural engineers (which are both majors at Kansas State University). We discuss differences between building a dog house, a garage, a home, and a skyscraper. An important aspect of a dialog is that students contribute many of the essential points. The important result of this dialog has been that students accept the many concepts that carry over to software architecture. The primary concepts are models and specification diagrams as the artifacts of architectural design and the distinction of abstract models vs. implementation models. Numerous other concepts include: user-centered design, user requirements vs. standard codified requirements, validation and approval of design models, the importance of readability of design diagrams, understanding of common kinds of architecture (as a ranch home vs. a cape Code), use of common patterns within architectures, use of standard building components, frameworks for building (as in modular construction), inspection and acceptance of implemented work, and others.

LEARNING UNIT	HRS		TOPIC	Hours	joint with:	Assignments
1. CONCEPTS	11.5		general concepts	1.5		
1a			overview	0.5		
			UML overview	1.0		
1b			class model		with generic model, (2)	P2
			sequence models	1.5	(2)	
			GUI prototype	1.0	with POS, (2)	
			state model		reactive system, (2)	
			deployment model		web system, (2)	
1c			OCL	2.5	rental system, (2)	
			UML extensions	1.0		
			complexity	1.0		
			current papers	1.5		
2. ARCH TYPES	9.5		generic app w collection	1.0	JBuilder, (5)	P2
			persistent collection	0.5		
			Point of Sales	0.5		
			rental system	1.0		
			appointment system	1.0		
			reactive systems	2.0		
			card game	1.0		P4
			graphic manipulation	1.0		
			UML tool	0.5		P4
			collection web service	1.0		
3. PARTS	3.0		PARTS			
3a			wrapper / adaptor	0.5		
			collections	0.5		
3b			patterns	1.0		
			RMI	0.5		P3
			beans	0.5	(5)	
4. CREATING MODELS	3.5		in-class exercises	3.5	(2)	in-class
5. IMPLEMENTATION	4.0		JBuilder	1.0		P1
5a			app wizard	0.5		
			web API's	0.5		
			iterator	1.0		
5b			UML tool	1.0		
6. PR0JECT	5.5		definition	1.0		Project
6b			in-class reviews	4.5		
7. OTHER	8.0		exams	3.0		
			homework		(other)	
7b			writing	0.5		H5
			C++	2.0		P6
			pointers	1.0		
			school holiday	1.5		
TOTAL	45.0			45.0		

Table 1. Sample Topics per Learning Unit

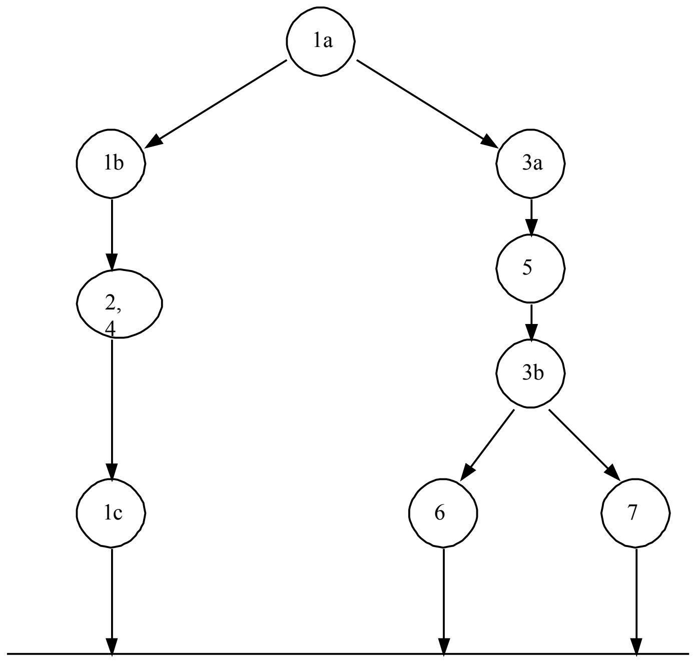

Figure 1. Order of Learning Units

After establishing the concepts of architecture, the second step in the course is an overview of the artifacts of software architecture, with comparison of their relation to building architecture. We survey the topics of the learning units and give brief examples of the items. Comments about each of the units follows.

For UML models, we use the Schaum's Outline Text [7]. We cover model components in the following order: use-case model, narrative about the use-case model, abstract class model, sequence models, statechart model, and GUI prototype model. Later, component and deployment models (which unfortunately are called implementation models), and OCL (the object constraint language, which is not supported by commercial UML tools) are added. Each of the model components is illustrated in the context of one or more architecture domains, which are discussed in the next paragraph. Some points of warning should be noted:
(i) Supporting narrative for the use-case model is not explicitly noted as part of UML, but surely it is necessary.
(ii) While the text does present correct model diagrams, we add discussion of correctness and good style for such diagrams. Issues of style include the selection of names, size and layout of models, consistency of the component models, layering, use of packages, and more.
(iii) The text makes no distinction between abstract models and what I call implementation models. For this course, a class model is abstract if it hides or ignores "obvious" methods and all classes related to user input and data storage. A class model is an implementation model if it faithfully represents the implementation code. For abstract models, we often do not show explicit collection classes whereas for implementation models we always show an explicit collection class for each composition collection.

(iv) For use-case models we stress actors as nouns and cases as “essential business” methods with verb names. Since we only implement operations with a single user interface, we never allow cases with interaction of multiple actors.
(v) The style of sequence diagrams in the text differs from that of the UML tool. We prefer to show a separate activation rectangle for each method invocation, as does the TogetherJ tool.
(vi) The UML text shows many fragments of models, but little development of full software design. That is treated in our second learning unit.
(vii) One addition to UML is the GUI prototype. Initially, this is just a collection of sketches of user interfaces together with a state model of the transitions between various input states. Later, we implement the state model using a rapid development GUI builder (as in JBuilder or TogertherJ tool) and just state transition logic as the “code” for the model.

In learning the practice of design, there is little substitute for practice and feedback. The second learning unit consists of models of several kinds of common architectures. Note that model components are of different kinds (use-case, class, etc.). Overall models are of different kinds. We cover such “kinds” as a generic application, a generic application which manages a collection of items (a kind of mini-database application), a document editor, a point of sales system, an appointments system, rental systems, a real-time controller, card game applications (bridge and/or poker), direct graphic manipulation (with specific instances as a state machine simulator and a UML tool builder), and others. The core of introductory software architecture is that students should have experience with a wide variety of such models and with the discovery / creation process by which such models are developed. There is not a required text for this learning unit. Several of the applications and models take from or influenced by material in older books by Coad [8,9] . Guidelines for GUI design are taken from Coad [9] and Schneiderman [17].

Our pattern of presentation is to engage the students in discussion of an initial model, then show and explain an instructor’s solution, and then work through an in-class exercise. The in-class exercises are learning unit number four. The typical in-class exercise is that (except for the first use-case model) the class has some prior component models. Then students work in pairs (a “turn to your partner” mode) to develop a single model component. Students may ask questions, but general guideline answers are announced to the whole class. The instructor and a graduate teaching assistant are available to consult with groups during the exercise. When most pairs have a draft, pairs are merged into groups of four (or three if necessary) to compare solutions and select a single solution. Sometimes the instructor’s solution and sample solutions from the class are presented (anonymously) to the class at the end of the exercise. Sometimes, the solutions are collected, evaluated, and evaluation comments and examples are presented to the class at the next meeting. Sometimes these group exercises are “graded” for attendance, but never for an evaluation grade. Evaluation grading is done for examination questions.

There are features of the classroom that are essential for the in-class exercises. First, the classroom is a lecture auditorium. Each row is a long curved table and students have chairs with rollers. This arrangement allows students to work in small group in-class exercises (two, three, or four students in a group). The room has large center aisle and adequate passage behind students’ chairs, so that the instructor can easily move to student groups to answer questions or to provide feedback about in-class exercises. The instructor’s podium has controls for a large screen projection of either the networked console computer, the instructor's notebook computer (for display of demonstration software), or 8.5" x 11" opaque sheets. The last feature showing hand drawn diagrams, either from the instructor or samples of student work as part of in-class exercises.

Learning unit five covers use of software development tools. One reason these are coupled to software architecture is that the tools impose an architecture style on the developed code. Another reason is that the tools support use of standard "parts" which is learning unit three. The course requires use of JBuilder [16] for program development and TogetherCC [18] for UML model development and for code development. We also show Visual Studio and Microsoft VISIO [20] as points of comparison. Newer versions of JBuilder / TogetheCC will likely fully merge the development of models and code. One pragmatic issue is that students were able to use the free evaluation version of JBuilder on their own computers, whereas many students did not have large enough machines to run TogetherJ. Initially, many students felt that these tools were an impediment to their progress in developing tiny programs; they could code easier with a text editor and compiler. However, as more tool features were presented that complaint subsided. One issue in using JBuilder (or other development environments) is that the tool produced a different generic architecture than many of the students had used in prior courses. An obvious strength of development tools is in rapid GUI development. This was useful in building the first GUI prototype, which had many more frames and controls than programs in earlier courses. An important feature of TogetherCC is that code and class models can be consistent (which was not true in previous versions of the tool and of the course). At this point, the tool constrains that sequence models are consistent with the class model, but there is no constraint that the sequence models are consistent with the actual code. As with the previous unit, there is no text for using the development tools. Fortunately, both JBuilder and TogetherCC have extensive help, tutorial, and example files included in their full installation.

Some of the standard "parts" that are covered include the following:
(i) The application model for the course must have standard menu and frame controls. It must have an "About" frame with pictures of the students who submits the program. It must have a "Help" frame with scrolling text. These features are partially generated by the development tools. The templates used by the generation wizard can be further customized to the specific requirements.
(ii) Data management applications must use the Java collection classes with an adaptor wrapper class to provide putting and getting specific type objects rather than just "object" items. The first data application must use serialization for persistent storage.
(iii) Later programs must use some beans beyond the common Swing components. We show use of a calendar bean (such as [1])and locally developed database access beans. .
(iv) Later we use a pattern wizard for the adaptor class to wrap the standard collection classes.
(v) The second version of the data management application uses the RMI features to access items on a remote server. For this case, the instructor provides the remove service and student merely use the RMI service.
(vi) We present (but do not use) some of the GOF patterns [11 and 15] and some of the business patterns developed by Coad [10].
Students had not used most of these "standard" items in previous courses. Program examples for

most of the items in this unit are illustrated in the text by Gittleman [15]; however, that text does not use UML models or development tools. Likely, the components focus of the course will increase in future semesters.

The project component of the course combines model development and program implementation. Project work starts at the mid point of the semester. Students were encouraged to work in pairs. The project builds on either exercises in the first half of the semester or upon project work from previous semesters. Most recently, the assignment was to add features to one of several versions of the project from the last semester. Students were required to build the models for the modified project. They were to keep a progress log with sign-off lines for the drafts of each model component. They were able to automatically generate class models from the previous code. We did allow a few students to use Visual Age and Visual Studio J#. For several weeks, one session per week (often running past the allocated class period) was allocated to review of project models by the instructor and the graduate assistant. This was the most effective unit of the course in providing feedback about student's work, in part because students were well engaged in the project. However, there is an instructional burden associated with the project; it takes almost a week to grade the final submission of program models and code for all of forty-plus projects. Example models for one of the projects can be found at the course archive site [22].

The course has been required to include a component of introduction to C++ and a component of technical writing even though those are not specifically related to software architecture. The technical writing assignment was to review a current article or technology related to software architecture. Some of the articles are presented to the class; most recently, the concept of model checking was shown.

Conclusion

Once students learn the core features of a programming language, the major "programming" task for development of larger software applications really deals with aspects of software architecture. Yet, the component of software architecture is not given adequate emphasis in computing curricula and certainly not adequate emphasis in programming courses for majors in other disciplines. This paper has presented the outline of a course on software architecture and design.

References

[1] ACM Curriculum Committee on Computer Science. Curriculum '68: "Recommendations for the undergraduate program in computer science." Com. of the ACM, 11(3):151-197, March 1968.

[2] ACM Curriculum Committee on Computer Science. Curriculum '78: "Recommendations for the undergraduate program in computer science." Com. of the ACM, 22(3):147-166, March 1979.

[3] Computing Curricula 1991, Sample Curricula, ACM, http://www.acm.org/education/curr91/eab4.html

[4] Computing Curricula 2001, Draft, http://www.computer.org/education/cc2001/final/index.htm

[5] Chapter 7. Introductory Courses, Computing Curricula 2001, http://www.computer.org/education/cc2001/final/chapter07.htm

[6] CS290T. Software Development, Computing Curricula 2001, http://www.computer.org/education/cc2001/final/cs290t.htm

[7] S. Bennett, J. Skelton, K. Lunn, UML, Schaum's Outlines, McGraw Hill, 2001, paperback.

[8] P. Coad, D. North, M. Mayfield, Object Models: Strategies, Patterns, and Applications, Yourdon Press, 1997.

[9] P. Coad, M. Mayfield, J. Kern, Java Design, Prentice Hall, paperback, 1999.

[10] P. Coad, E. Lefebvre, J. DeLuca, Java Modeling in Color with UML, Prentice Hall, 1999.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[12] D. Garland, et al, "Experience with a Course on Architectures for Software Systems" , Lecture Notes in Computer Science Vol. 376, Springer Verlag, 1992. http://www-2.cs.cmu.edu/~able/publications/sacourse-csee92/

[13] D. Garlan and M. Shaw, "An Introduction to Software Architecture", in V. Ambriola and G. Tortora (ed.), Advances in Software Engineering and Knowledge Engineering, Vol 1, World Scientific Publishing Company, Singapore, pp. 1-39, 1993. http://shaw-weil.com/marian/DisplayPaper.asp?paper_id=49

[14] D. Garlan, "Software Architecture", Wiley Encyclopedia of Software Engineering, J. Marciniak (Ed.), John Wiley & Sons, 2001. http://www-2.cs.cmu.edu/~able/publications/encycSE2001/

[15] A. Gittleman, Advanced Java: Internet Applications, 2 ed., Scott Jones, paperback, 2002.

[16] JBuilder, Borland Corp., http://www.borland.com/jbuilder/ , 2003

[17] B. Schneiderman, Designing the User Interface, Addison Wesley, 1998.

[18] Together Control Center, Together Soft Corp., http://www.togethersoft.com/products/index.jsp, 2002

[19] Unified Modeling Language, Object Modeling Group, http://www.uml.org/ , 2003

[20] VISIO, Microsoft Corp., http://www.microsoft.com/office/visio/default.asp, 2003

[21] alphaBeans, http://java.iba.com.by/javaweb/alphabeans.nsf, 2002

[22] CIS501 Software Architecture Archive, Kansas State University, http://www.cis.ksu.edu/~hankley/d501/Models/Memorando/ , 2002

Biography

WILLIAM HANKLEY is Professor of Computing and Information Sciences at Kansas State University in Manhattan, KS. He received his B.S.E.E. and M.S. from Northwestern University and the Ph.D. in Electrical Engineering from the Ohio State University in 1967. He has developed undergraduate courses on programming, software architecture, database applications, and graduate courses on software specification and verification. He has eight grandchildren.