Teaching Software Architecture Principles in CS1/CS2

Paolo Bucci
Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
+1-614-292-0066
bucci@cis.ohio-state.edu

Timothy J. Long
Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
+1-614-292-1408
long@cis.ohio-state.edu

Bruce W. Weide
Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
+1-614-292-1517
weide@cis.ohio-state.edu

1. ABSTRACT

It is possible to teach architectural-level issues as early as CS1/CS2. But the ultimate success of this approach hinges in part on the availability of appropriate tools to assist students in building the right *mental models* of the development and composition of software systems.

1.1 Keywords

Software engineering education and training, software engineering tools and environments, software architecture

2. INTRODUCTION

Early work on software architecture was largely descriptive (e.g., [16]). It involved selecting samples of "real" software systems and producing taxonomies of the architectural styles observed in them. Recently, material that is more prescriptive has become available which advises software engineers regarding when they might use different architectural styles [13]. But in both kinds of work, authors have generally assumed (quite correctly) that their readers are experienced software engineers who harbor a variety of mental models of what software is and how it is constructed. These mental models are firmly entrenched in the readers' minds, sometimes dating from the time they took their first CS courses. So, a major part of an author's task when writing about software architecture is to augment or even to replace people's existing mental models with ones based on the new terminology and concepts.

Should we perpetuate the need to change people's mental models of software by continuing to teach introductory CS courses in a traditional manner? Or should we try to help new CS students develop a higher-level *initial* mental model of what software is and how it is constructed, by explicitly discussing and being prescriptive about software architecture ideas as students first learn about software? In this position paper we outline some observations from our experience in attempting to do the latter, concentrating attention on tool support — one factor which seems to be very important in helping a student form a rich mental model of software that is consistent with the ideas of software architecture.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISAW3 Orlando Florida USA
Copyright ACM 1998 1-58113-081-3/98/11...$5.00

3. BACKGROUND

For over a decade, the Reusable Software Research Group (RSRG) at The Ohio State University (OSU) has been exploring component-based software engineering technology. This work has led to the RESOLVE framework, language, and discipline [17]. Stated in software architecture terms, the RESOLVE framework and discipline provide a detailed prescription for how to:

- *think* about component-based software systems, and
- *design and build* software having a particular (yet quite general) architectural style.

The particular style we advocate is based on design of, and with, abstract and concrete templates, and the primary composition mechanism is template instantiation. The discipline is quite detailed; it mandates particular ways of addressing not just "macro-architectural" but many "micro-architectural" issues commonly faced by software designers [8]. The RESOLVE language is usually listed (e.g., see the on-line "Software Architecture Technology Guide" at http://www-ast.tds-gn.lmco.com/arch/guide.html) as one of several "architecture description languages", or ADLs. The RSRG home page is at http://www.cis.ohio-state.edu/rsrg.

Since late 1996 we have been carefully weaving many of the key ideas developed by the group into the curriculum of first-year CS courses for majors at OSU and elsewhere [9, 19]. Others have taught software architecture principles in advanced courses (e.g., [6]), but to our knowledge, we are the first to base introductory CS courses on such a view of software. These curriculum innovations reflect our opinion that we need to start immediately building in students'

minds a modern integrated component-based view of software in which architectural issues play a central role.

4. POSITION

We maintain the following position:

> It is possible to teach architectural-level issues as early as CS1/CS2. But the ultimate success of this approach hinges in part on the availability of appropriate tools to assist students in building the right *mental models* of the development and composition of software systems.

The basis for the first part of the position is outlined in [9], which we do not further discuss here. Rather, we concentrate on second part of the position, which is supported both by the previous research on interactive systems in general (e.g., [10, 11]), and our own experience in these new courses. A typical student forms early — and thereafter rather vigorously clings to — some initial mental model of what "software" is. The student does not have the luxury of opting out and forming no such mental model whatever. Even in the absence of explicit instructions to form a particular mental model as a way of thinking about software and software engineering, the student must and does form one just to deal with the program editor[1] which is used to do course assignments.

A typical student in a traditional CS1 course today starts out writing small programs, from scratch in a language such as C or C++, using a text editor that gives complete freedom and little or no guidance during the process of software development. Many other approaches to editors have been explored with the goal of providing additional support for the challenging and demanding task of building software (e.g., [12, 14]), and some have even been used in introductory courses (e.g., [5, 15]). Every editor has one intrinsic feature, though: Through its *system image*, i.e., its user interface and documentation, it conveys to the user some conceptual model of what a software system is. But in no tool we know of — and certainly in no tool used in introductory courses — is this model adequate for discussing architectural-level issues.

The conceptual model of software that an editor projects depends essentially on two things: the *representation* used to display software components, and the *operations* the editor provides to manipulate, modify, and combine components. A text editor that displays programs as simple text, and that provides text-editing operations such as cut-and-paste, suggests a model of a software system as some files containing text which has little or no structure beyond what is required to get the compiler to accept that text. So, what do most students think a software system is? Not surprisingly, a typical student whose experience has been with a traditional CS1 course thinks that it is one or more files containing text written in a programming language.

Many software professionals retain this mental model even after many years on the job; hence, it is often difficult to convey the view that software systems have "architectures" and that the "components" of software systems are not necessarily files and lines of code. Other experienced software engineers have managed to overcome their initial, simplistic mental models and have formed their own new mental models of what software is and how it is constructed from language-oriented building blocks. Here each building block (perhaps a procedure, or a class, or a template, or a "filter") has a meaning, a role, and a purpose that go beyond the concrete syntactic representation the language assigns to the constructs involved in describing it. Developing an even more sophisticated view requires a higher-level notion of what software is: A software system is not a string of characters or any other concrete representation; it is a set of conceptual entities built from other conceptual entities to achieve an intended behavior.

5. TOOL DESIGN ISSUES

What should a program editor be like in order to help students form an initial mental model of software which is sophisticated enough to involve the higher-level notions related to software architecture, and which therefore is not based merely on files containing programming language code? We use the term *conceptual editor* to describe a program editor designed to support and promote a specific high-level conceptual model of the development and composition of software systems. This section outlines some requirements that we consider fundamental to the design of such a tool. In context we briefly explain the status of the *Software Composition Workbench* (*SCW*) tool we are developing and currently deploying in our CS1/CS2 classes. It is designed to meet these requirements. Additional details and comparisons of the SCW with syntax-directed editors, structure editors, and visual environments, are provided in [1, 2, 3].

5.1 Conceptual model integrity

A high-level conceptual model of software and how it should be developed is the necessary starting point in the design of a conceptual editor. Without such a model guiding the design of the editor, it is unlikely, if not impossible, that the resulting tool will convey the appropriate mental model to students. Most existing editors typically are designed around one or more programming languages and/or the demands of traditional programmers, and they fail to provide a clear, consistent, high-level view of what software is and how it is constructed.

The SCW is based on the RESOLVE framework in general, more specifically on the *ACTI* ("abstract and concrete templates and instances") model of component-based software

[1] Throughout this paper we use "program editor", or simply "editor", to refer to the tool used to enter, modify, and view software.

systems. ACTI is a conceptual model of software systems and subsystems that was proposed and formalized by Edwards [4]. We introduce ACTI in our CS1/CS2 courses [9, 19] as the basis for explaining what a software system is and how it should be designed.

5.2 Programming language independence

A conceptual editor is not influenced by language mechanisms or by the hardware or operating system — things that usually have a big impact on the design of programming languages. However, it is strongly influenced by the architectural style(s) it is supposed to support, by the software development discipline, and by the programming paradigm. For instance, if we want to convey a "pipes-and-filters" architectural style, then we might treat a software system *conceptually* as a collection of components (filters) and connectors (pipes). On the other hand, if we choose to convey an object-oriented architectural style, then we might treat a software system *conceptually* as a collection of classes and objects, where objects are instances of classes, and classes have methods and attributes. Even for a given style there might be more than one model (or more than one way to look at things) that applies to the same entity, e.g., a data flow model vs. a control flow model of algorithms.

The SCW is designed to convey the ACTI model and the associated RESOLVE discipline for using it (which, as mentioned in Section 3, involves template instantiation as the primary architectural-level composition mechanism).

5.3 Distinctions among intellectual tasks

The conceptual model also must reflect the fact that building software involves many intellectually distinct activities, and conceptually distinct entities. So, for instance, composing components (system integration) is intellectually different from designing a new component interface, which is intellectually different from implementing it.

The SCW directly supports each of the above distinct activities as well as other separable tasks related to component extension, unit testing, modular reasoning, etc.

5.4 Constraints as guides

The editor can guide the student by limiting the space of possible alternatives to those that are meaningful and consistent with the conceptual model the tool is trying to convey. The use of appropriate syntactic and semantic constraints can be an effective way of leading the student to the recommended architectural and design choices. For example, a conceptual editor promoting a pipes-and-filters architectural style could enforce certain structural constraints on the configuration of components and connectors [18]. Of course, the constraints imposed by the editor must be chosen — and the editor must be designed — so the student perceives them as helpful and not restrictive.

The RESOLVE discipline is highly prescriptive, so imposing constraints with the SCW is no problem. Only experience with the SCW will let us determine whether students view these constraints as helpful or too restrictive.

5.5 Visibility

Important aspects of a system often are hidden among other details, forcing the student to make an additional effort to keep track of them, and possibly causing problems when the student ignores them. The important conceptual entities involved must all be made explicitly visible. For instance, the relationships (design-time dependencies or integration-time dependencies [7]) among components should be made visible and easily accessible to the student so the architecture of the system can be more easily understood and directly manipulated.

We routinely use two kinds of diagrams [9, 19] to visualize relationships involving various ACTI conceptual entities, but these diagrammatic views currently are not supported by the SCW. With some experience we hope to be able to determine whether such diagrams are as useful with tool support as they are when drawn as paper documentation.

5.6 Representation consistency

How the important aspects of a software system are presented to students is essential to their proper understanding and successful development of software. This representation is key to a successful conceptual editor. If it captures the nature of the conceptual model, it can promote it and thus provide support for the tasks to be done. On the other hand, the wrong representation can doom an editor to failure. Different representations for different aspects of a system may be necessary in an architectural style that involves conceptually distinct activities. So, for example, the use of boxes to represent components and directed lines to represent connectors may be appropriate in some circumstances (e.g., if a connector represents a fixed, design-time relationship between components). In other situations the connectors themselves might be viewed as first-class entities and might be represented with their own boxes (e.g., if a connector specifies a protocol for hooking up components [16] or another less traditional relationship [18]).

The SCW uses a form-based representational approach with the forms organization directly matching the ACTI conceptual model organization. Currently it uses no diagrams of the type mentioned above.

5.7 Appropriate operations

The operations supported by the editor for the construction and manipulation of a program must make sense in the realm of the conceptual model. To this end, they must not be chosen based on the actual displayed representation of the software system, but based only on the conceptual view being defined. For example, a tool using a "boxes-and-arrows" notation (boxes for components and arrows for connectors), should provide operations with conceptual integrity to create new components, to add/remove components to the current system, to connect the components in

meaningful ways [18], etc. It should not provide operations that directly manipulate the displayed representation, such as drawing a new box or drawing an arrow between two boxes.

The SCW allows students to browse through component catalogs (i.e., not files) on the basis of component families (i.e., not a file system) and via a small set of conceptual relationships [7] among components defined in the ACTI model (i.e., not by following "#include" statements). New components are created essentially by filling in forms that reflect the meaningful conceptual entities of the ACTI model, mostly by using menus to select from among a small set of conceptually appropriate operations which are determined by the context in which the user is working.

6. ACKNOWLEDGMENTS

We gratefully acknowledge financial support from the National Science Foundation under grants DUE-9555062 and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, and from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, the U.S. Department of Education, or Microsoft. We also appreciate the assistance of the many other people who are involved in this project, especially Steve Atkinson, Steve Edwards, Steve Fridella, Joe Hollingsworth, Greg Kulczycki, Bill Ogden, Elley Quinlan, Scott Pike, Murali Sitaraman, and Stu Zweben.

7. REFERENCES

[1] Bucci, P. A Program Editor to Promote Reuse. In *Proc. 7th Annual Workshop on Software Reuse*, Univ. of Maine Publications, 1995.

[2] Bucci, P. Conceptual Program Editors. In *Proc. 8th Annual Workshop on Software Reuse*, Univ. of Maine Publications, 1997.

[3] Bucci, P. *Conceptual Program Editors: Design and Formal Specification.* Ph.D. diss., Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, OH, 1997.

[4] Edwards, S.H. *A Formal Model of Software Subsystems.* Ph.D. diss., Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, OH, 1995.

[5] Garlan, D.B., and Miller, P.L. GNOME: An Introductory Programming Environment Based on a Family of Structured Editors. *ACM SIGPLAN Notices 19*, 5 (May 1984), 65-72.

[6] Garlan, D., Shaw, M., Okasaki, C., Scott, C., and Swonger, R. Experience with a Course on Architectures for Software Systems. In *Proc. Sixth SEI Conf. on Software Eng. Education*, Springer-Verlag, LNCS 376, 1992.

[7] Gibson, D.S. *Behavioral Relationships Between Software Components.* Ph.D. diss., Dept. of Comp. and Inf. Sci., The Ohio State Univ., Columbus, OH, 1997.

[8] Hollingsworth, J.E., and Weide, B.W. One Architecture Does Not Fit All: Micro-Architecture is as Important as Macro-Architecture. In *Proc. 7th Annual Workshop on Software Reuse*, Univ. of Maine Publications, 1995.

[9] Long, T.J., et al. Providing Intellectual Focus to CS1/CS2. In *Proc. 1998 ACM SIGCSE Symp.*, ACM, February 1998, pp. 252-256.

[10] Meyrowitz, N., and van Dam, A. Interactive Editing Environments: Part I. *ACM Computing Surveys 14*, 3 (Sept. 1982), pp. 321-352.

[11] Norman, D.A. *Things That Make Us Smart.* Addison-Wesley, 1993.

[12] Notkin, D., et al. Special Issue on the GANDALF Project. *J. Syst. Software 5*, 2 (May 1985).

[13] Perry, D.E. An Overview of the State of the Art in Software Architecture. In *Proc. 19th Intl. Conf. on Software Eng.*, ACM, 1997, 590-591.

[14] Pictorius, Inc. *Prograph Classic Tutorial*, 1995.

[15] Reps, T., and Teitelbaum, T. The Synthesizer Generator. *ACM SIGPLAN Notices 19*, 5 (May 1984), 42-48.

[16] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[17] Sitaraman, M., and Weide, B.W., eds. Component-Based Software Using RESOLVE. *ACM Software Eng. Notes 19*, 4 (1994), 21-67.

[18] Stovsky, M.P, and Weide, B.W. Building Interprocess Communication Models Using STILE. *Proc. 21st Hawaii Intl. Conf. on Systems Sciences*, 1988, vol. 2, 639-647; reprinted in *Visual Programming Environments: Paradigms and Systems*, E.P. Glinert, ed., IEEE Press, 1990, 566-574.

[19] Weide, B.W. *Software Component Engineering.* OSU Reprographics, Columbus, 1998.