# Teaching Software Engineering Bottom Up

R. E. K. Stirewalt

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48840
e-mail: `stire@cse.msu.edu`

Abstract

A typical CS curriculum contains a course on software engineering, which introduces principles and heuristic methods for designing large software systems subject to desirable properties, such as maintainability and extensibility. The nature of this body of knowledge suggests that the best method for teaching it is to use the elaboration theory of instruction. Applying this theory to software engineering requires a complete inversion in the traditional coverage of topics. We developed a new course, CSE 370, which incorporates this "bottom up" coverage. Using this method, we are able to instill a higher level of cognitive ability in software-engineering methods than we were able to achieve using the old method.

Figure 1. Traditional (waterfall) model of software development.
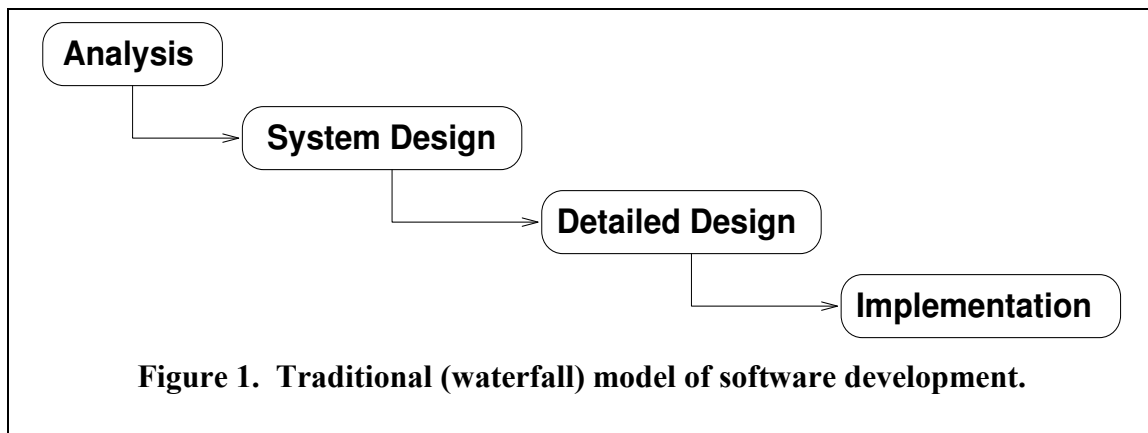
1 Introduction

A typical computer science curriculum contains a junior-level course on *software engineering*, which develops principles and heuristic methods for designing large software systems. At many universities, this course is organized around an idealized model of the software lifecycle[1], which comprises a linear sequence of discrete phases (**Figure 1**). This paper argues that such an organization is pedagogically flawed if the

goal is to develop skill in software design. We propose a "bottom-up" approach, which draws from the elaboration theory of instruction[2], and have developed a new course around this approach.

Problems with the traditional organization stem from the nature of knowledge in software design, which is organized around a small set of fundamental principles of software construction. An example is the principle of *information hiding*, which requires module interfaces to hide implementation details that might change over the lifetime of a system, thereby promoting extensibility and easing long-term maintenance.[3] Such principles are supported by best practices and heuristic methods, which provide more concrete guidance in their application. For example, several *design patterns*[4] depict how to combine class inheritance and delegation to construct interfaces whose implementations can be varied without affecting the module's clients. These principles and methods are not based on an analytical science of design. Rather, they are statements of collective experience and are thus *metaphorical* in nature.[*]

Fundamentally, metaphorical knowledge only has meaning in the context of experience.[5] For example, a design pattern is an expression of experience, and a developer with experience in object-oriented design will recognize it as such and be able to apply the pattern in a new context. However, to an inexperienced designer, the pattern represents nothing but words and pictures, which can be memorized and regurgitated, but which will likely never spring to mind in a design context that could exploit it. To really learn this material, students must first acquire the experience necessary to recognize these metaphors as statements of experience. Only then will they be able to retain and apply the knowledge.

We recently redesigned our software-engineering course to better develop in students the ability to retain and apply software-engineering knowledge. The new course draws from an established pedagogical theory called the *elaboration theory of instruction*, which uses *epitomizing,* rather than *summarizing*, to select and organize the content of a course.[6] Briefly, summarizing entails teaching a large number of ideas, usually at a superficial, abstract, memorization level. By contrast, epitomizing entails teaching a small number of fundamental ideas and presenting them at a concrete level that is immediately meaningful to the students. To apply this theory, we had to radically change the order in which we covered the topics. In fact, we now cover topics in exactly the reverse order in which they were covered before.

This new "bottom up" approach utilizes an intensive regimen of laboratory exercises, which require students to add a new feature to an existing corpus of software. To add the new feature, a student must first understand and then redesign (or refactor) the existing code, which we designed for the purpose of illustrating the complexities that arise in large software. Each of these lab exercises gives students a basis in experience for understanding the lecture material, which presents a pattern or solution to address the

---

[*]Contrast the subject of this metaphorical knowledge with other concepts in computer science—e.g., fundamental data structures and algorithms—which enjoy a formal (mathematical) meaning irrespective of design context.

complexity that was motivated in the lab. Thus, unlike in traditional courses, which begin with a requirements specification followed by design and implementation, CSE 370 begins with directed implementation and maintenance tasks, which motivate design strategies and techniques.

2 Background

We developed CSE 370 to address issues that were uncovered when we began preparing for ABET accreditation. To be accredited, a curriculum must be subjected to a process of *continuous quality improvement* (CQI), which we had to develop *ex nihil*. The effort required us to identify and precisely specify fine-grain, measurable objectives for each course and to assess student performance against these objectives. Because these activities influenced the design of CSE 370, and because we use fine-grain objectives to validate our work, we now briefly introduce these concepts.

2.1 History of the problem

CQI is a general framework for systematically improving the products that flow out of a production process. To successfully apply CQI to a curriculum requires:

1. specifying learning outcomes at a granularity that is sufficient for improving specific components of instruction,

2. developing assessment methods to measure these fine-grain learning outcomes, and

3. improving the transparency of instruction, so that variations in outcomes can be traced back to events within the course or its prerequisites.

Stice describes how to fulfill the first two requirements by specifying learning outcomes in terms of *concrete cognitive objectives,*[7] each of which must specify (1) what the learner is to do after instruction, (2) under what conditions, and (3) the acceptable performance.

1. **Knowledge**: List, recite

2. **Comprehension**: Explain, paraphrase

3. **Application**: Calculate, solve, determine, apply

4. **Analysis**: Compare, contrast, classify

5. **Synthesis**: Create, construct, predict, design

6. **Evaluation**: Judge, decide, critique, assess, argue

Figure 2: Bloom's taxonomy of cognitive objectives

Concrete cognitive objectives afford several benefits. First, because each objective specifies an action on the part of the learner, one can use the Bloom Taxonomy[8] (**Figure 2**) to infer the level of cognitive ability that is being exercised by a particular objective. Second, because each objective indicates the acceptable performance, it can be measured.

Finally, by linking courses through prerequisites and outcomes, one can make adjustments in the content/evaluation metrics of one course to accommodate the needs of another.

Despite these benefits, concrete cognitive objectives are difficult to specify, and they often require the instructor to make major changes to the contents and assessment methods of his or her course. We experienced both of these difficulties when we began to specify the objectives for our old software-engineering course, CSE 470. For example, we discovered our assessment methods rarely exercised cognitive abilities beyond Bloom level 2, which is significant because an upper-division engineering course should develop abilities at levels 3 and higher. As we began to improve our methods to assess these higher-level objectives, we discovered students were unable to perform at this level. This told us that something was wrong with how we were teaching. To fully understand the problem requires some background in the nature of software-engineering concepts, which we now describe.

2.2 Software-engineering knowledge

Software engineering is concerned with principles and methods for specifying, designing, implementing, and maintaining large software systems. Knowledge in this discipline is organized around an idealized model of software development, which identifies distinct phases that are performed sequentially (**Figure 1**). Briefly, *analysis* produces a system specification; *system design* produces a system architecture that allocates responsibilities to major hardware and software components; *detailed design* develops (or reuses) interfaces for individual software modules; and *implementation* implements these modules.

Each phase produces a tangible artifact; however the only executable artifact is the final implementation, and in real projects, software development may proceed for several years before a fully developed executable appears. Thus, much of the software-engineering body of knowledge includes principles, notations, and heuristic methods that support these early phases and (to some extent) the evaluation of the early artifacts.

An example notation is a UML class diagram, which graphically depicts the static structure and inter-relationships of the classes in a system. Such diagrams are useful in the early phases of development because they describe an important aspect of the final system—namely the structural dependencies of the data—and yet they are much more abstract than the final code. Practicing engineers leverage this abstraction to evaluate and compare candidate designs prior to implementing them.

An example principle is *information hiding*, which requires module interfaces to hide implementation details that might change over the lifetime of a system, thereby promoting extensibility and easing long-term maintenance. Such principles are supported by best practices and heuristic methods, which provide more concrete guidance in their application. For example, several *design patterns*[9] depict how to combine class inheritance and delegation to construct interfaces whose implementations can be varied without affecting the module's clients. Such principles and design patterns derive from

years of collective experience of many software developers on many different kinds of projects. As with UML class diagrams, their virtue lies in their abstraction, which permits an experienced designer to apply them in many new contexts.

2.3 The problem

Unfortunately, the abstraction that permeates the software-engineering body of knowledge, so useful to the experienced designer, appears to be a barrier to understanding for the inexperienced student. We observed this phenomenon when we began to specify and assess concrete cognitive objectives for our old course. To be concrete, consider the following example objective:

> Given source code for three or more classes whose instances collaborate with one another to implement an invariant, refactor these classes using the *mediator pattern* to encapsulate, within a single class, the logic for maintaining the invariant.

This objective describes a cognitive ability at level 3 in the Bloom taxonomy.

Despite spending considerable lecture time describing design patterns and the notations and conventions that go along with them, students were consistently unable to achieve objectives at this level. On the other hand, these same students demonstrated level 2 abilities, such as being able to recall from memory all of the essential characteristics of the mediator pattern. Reigeluth and Stein suggest that this phenomenon is inevitable under the *summarization model* of instruction, in which lots of very abstract concepts are introduced at a shallow level of detail. They suggest an alternative, called the *elaboration model*, in which fewer concepts are covered more deeply via a process known as *epitomizing*, whereby a concept is introduced using concrete motivating examples and then abstracted over time.

We began to restructure CSE 470 around the elaboration model and quickly discovered topics that were almost impossible to epitomize. A good example is coverage of end-to-end software development methodologies, which introduce notations and heuristics for engineering a product from analysis through to implementation. When properly applied, these heuristics will lead to implementations with certain desirable properties. For example, methodologies such as OMT[10] and the Rational Unified Process[11] aim to produce implementations that are amenable to extension and reuse. In fact, many of the notations and methods that are used during analysis were created to forestall problems during design and implementation. Thus, to motivate the methods used in the early phases requires a deep appreciation of the issues that arise in the later phases. Herein lies the lesson: To apply the elaboration theory to teaching software engineering, one must cover the different phases of development from the bottom up rather than from the top down.

3 Solution: CSE 370

To recap: The elaboration theory of instruction seems the most appropriate for developing sufficiently high cognitive ability in the use of abstraction, which is so central

to software-engineering knowledge. However, to apply the theory correctly requires introducing the phases of the software-development cycle bottom up, i.e., from the most concrete phase (implementation) to the most abstract (analysis). Unfortunately, the popular software engineering texts are not organized in this manner, and we know of no resources or teaching aids for running a course in this manner.

To address this deficiency, we developed CSE 370, which covers the SE body of knowledge starting at implementation and working backward toward analysis and which uses a carefully orchestrated regime of laboratory exercises followed by lectures to epitomize the various topics. CSE 370 is a four semester-hour course, with three hours of lecture and two hours of scheduled laboratory time per week. Currently, we offer three lecture sections per year, and each lecture section comprises three lab sections, each of which accommodates up to 16 students. Each lab section is administered by a graduate teaching assistant According to our bottom-up approach, each lecture addresses a problem, which is motivated in the lab, and culminates in a design pattern, strategy, or method that allows the problem to be solved systematically prior to implementation.

3.1 Course content

We now briefly describe the contents of the first two (of our four) course modules, which correspond to the phases of software development (**Figure 1**). For each module, we provide a description of the motivational laboratory exercises and an example objective to demonstrate the level of cognitive ability developed.

3.1.1 Implementation

The first module in the course is concerned with implementation. Because we are using object-oriented methodologies, this module aims to develop a high level of skill in the use of programming features that are fundamental to object-oriented programming, namely class inheritance and polymorphism. The laboratory exercises provide implementations of classes and then ask the students to modify those classes to introduce inheritance relationships and polymorphic operations.[†]

Lectures that follow these labs open with implementation problems whose solutions require the use of the mechanisms exercised in the lab. Students work in pairs to develop paper solutions, which the instructor surveys and then presents to the class. Lectures in this module are also concerned with developing and enforcing student use of precise terminology, all in the context of these concrete problems.

The entire module requires approximately two weeks to cover. At the end, students are able to apply themselves to objectives such as:

> Given a hierarchy of classes (e.g., in C++), extend this hierarchy with a *polymorphic* operation for which each class in the hierarchy provides a different method.

[†] A.k.a. virtual functions in C++.

Having achieved a mastery of these implementation concepts, the students are then ready for the first big abstraction step, in which they begin to use design notations and modeling languages rather than code.

3.1.2 Detailed design

The second module, concerned with detailed design, is more the meat of the course. This module aims to develop a familiarity with general principles of software design, such as information hiding, design for reuse, and design for extension and contraction. Laboratory exercises at the beginning of this module ask students to write programs that use or otherwise extend source code that was *not* designed according to these principles. The idea is that if students can see, first hand, the effects of poor design decisions, they will be ready to receive the abstract principles and heuristics that will help them produce good designs. Lectures that follow these labs pick up on the lab exercises and introduce notations, such as UML class and sequence diagrams, for representing the code in the labs in a way that allows students to visually perceive the design flaws.

As we progress through this module, we introduce design concepts and heuristics that prescribe how to formulate a principled design with the aid of the modeling notations that were introduced to demonstrate the flaws in the motivational problems. An example is the use of role-based design[12] to develop reusable classes for building interactive systems. Laboratory exercises at this level ask students to take existing software and modify it to make the classes reusable in some new (concrete) context.

In all, this module takes nearly five weeks to cover, but we have found at the end, students have a deep understanding of these (very abstract) concepts. An example objective is:[‡]

> Let C_1 and C_2 be familiar collaborations, where C_1 comprises n roles, implemented by classes $c_{1,1} .. c_{1,n}$, and C_2 comprises m roles, implemented by classes $c_{2,1} .. c_{2,m}$. Given a problem that calls for the synthesis of these two collaborations, develop adaptor classes for the objects that play roles in both collaborations, and write the appropriate configuration code.

Prior to the changes incorporated into CSE 370, we could not have expected students to master objectives at this level. Now, we routinely assess this objective on timed examinations, and students perform well.

3.2 Discussion

Since its inception, we have run CSE 370 three times, and we track student performance in detail. These data suggest that the expected numbers of students are demonstrating Bloom-level 3 (and higher) proficiency in software-engineering methods. In addition to the bottom up treatment and the careful orchestration of labs and lectures to epitomize the abstract concepts, we have also found other teaching innovations to be helpful.

[‡] For the interested reader, definitions of the technical terms used in this objective can be found in the Reenskaug reference, *Working with Objects: The OORam Software Engineering Method.*

Our approach benefits greatly from collaborative learning during lectures, especially think–pair–share.[13] Many of the lectures introduce design methods that address problems motivated by lab exercises. Applying these methods takes practice and requires critical instructor feedback. This need for personalized feedback will quickly exhaust precious instructor time if it cannot be handled during class, but it is clearly not possible to give personalized feedback to 50 students in an hour and a half. Collaborative learning techniques allow us to provide personalized feedback in a way that reaches all of the students at once.

Following the introduction of each new pattern, strategy, or method, we assign an in-class exercise, which students must work on, first by themselves and then in pairs, to formulate a solution. Working in pairs has two benefits: First, each student is forced to actually apply the technique rather than just copying it down. Second, and more importantly, a student's partner can often provide sufficient feedback on simple misconceptions, e.g., issues of programming-language syntax or misunderstanding what the exercise asks for. Consequently, many common misunderstandings are clarified by the students themselves, leaving the instructor to deal with the fundamental misunderstandings, of which there tend to be a very small number. I have found that the use of this technique dramatically increases the bandwidth of discussion and the overall retention of these concepts.

Finally, to teach these concepts bottom up required us to develop new resources, specifically laboratory exercises and sample programs. A key insight was to draw our examples from the realm of graphical user-interface software. Poorly designed user-interface code can motivate a wealth of design problems, and well-designed user-interface code exploits many of the design patterns that we want the students to learn. Our software uses and extends the *fast light toolkit* (fltk)[14], which is a free XWindows-based user-interface toolkit written in the C++ language. Our software runs under Sun Solaris and the cygwin simulator under Windows. For more details, or to get a copy of the software and the lab exercises, please e-mail the author.

4 Acknowledgements

[1] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall Publishing, 1991

[2] C. M. Reigeluth and F. S. Stein, "The elaboration theory of instruction," appears in *Instructional-design theories and models: An overview of their current status*, C. M. Reigeluth, editor, Lawrence Erlbaum Associates, 1983

[3] D. L. Parnas, "On the design and development of program families," appears in *IEEE Transactions on Software Engineering* SE-2(1), March, 1976

[4] E. Gamma *et al*., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley Publishing, 1995

[5] D. Sayers, *The Mind of the Maker*, Harper Collins, 1941

[6] C. M. Reigeluth and F. S. Stein, "The elaboration theory of instruction," appears in *Instructional-design theories and models: An overview of their current status*, C. M. Reigeluth, editor, Lawrence Erlbaum Associates, 1983

[7] J. E. Stice, "A First Step Toward Improved Teaching," appears in *Engineering Education* 66(5), February, 1976

[8] B. Bloom, *Taxonomy of Educational Objectives. Handbook I: Cognitive Domain*, David MacKay Company, 1956

[9] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley Publishing, 1995

[10] J. Rumbaugh *et al*., *Object-Oriented Modeling and Design,* Prentice Hall Publishing, 1991

[11] P. Krutchen, *The Rational Unified Process: An Introduction*, Addison–Wesley, 2000

[12] T. Reenskaug, *Working with Objects: The OORam Software Engineering Method*, Manning, 1995

[13] W. C. Campbell and K. A. Smith, *New Paradigms for College Teaching*, Interaction Book Company, 1997

[14] The Fast Light Toolkit homepage: http://www.fltk.org

KURT STIREWALT received his Ph.D. in Computer Science from the Georgia Institute of Technology and is now an assistant professor at Michigan State University. His research is concerned with the practical use of formal and semi-formal graphical models in the design, verification, and maintenance of large software systems.