SECRETS, HOT SPOTS, AND GENERALIZATION: PREPARING STUDENTS TO DESIGN SOFTWARE FAMILIES

H. Conrad Cunningham, Pallavi Tadepalli, and Yi Liu
Department of Computer and Information Science
University of Mississippi
University, MS 38677 USA
{cunningham,pallavi,liuyi}@cs.olemiss.,edu

ABSTRACT

An important technique for coping with the increasing size, variability, and complexity of software systems is the construction of software families. This technique exploits the common properties of a group of related software systems to achieve reuse of design or code. Computing science students should be introduced to the concept of software families and the methods for constructing them. This paper describes a course on the principles and practice of developing software families. It outlines the structure of the course and evaluates it from the perspective of the students and the instructor.

1. INTRODUCTION

Software systems are increasing in complexity. This results from at least two trends. First, software systems are growing in size. An internal study at Philips Electronics estimates that the workload associated with the development of a typical product is increasing by a factor of ten every seven-to-eight years [2]. Size increases lead to increased complexity in development and in management of product evolution. Second, because system designers view software as the more malleable medium, there is a growing tendency to move functionality from hardware to software. In particular, designers increasingly look to software to provide the needed variability in functionality within a group of related products [2].

One response to these trends is to treat a group of related software products as a *software family*. This is an old idea whose currency has increased in recent years, as is evidenced by the emergence of major series of conferences on the topic [2,13]. In a 1976 paper [15], Parnas defines a program family as a set of programs "whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members." If programmers can exploit these commonalities and enable extensive reuse of software, then the long-term development costs can be decreased. In a 2001 article [19], Parnas observes that there is "growing academic interest and some evidence of real industrial success in applying this idea," yet "the majority of industrial programmers seem to ignore it in their rush to produce code." He warns, "If you are developing a family of programs, you must do so consciously, or you will incur unnecessary long-term costs" [19]. Therefore, teaching future industrial programmers (i.e., students) how to develop software families is an important challenge for computing science curricula.

How can we respond to this challenge within a college course? The general form of software family, called a *software product line*, tends to be quite complex, involving

considerable knowledge of the application domain and the use of special-purpose translators and configuration tools [1]. That makes the concepts and techniques difficult to teach. However, the form of software family called a *framework* is more accessible. In the context of an object-oriented language, a framework is a reusable design captured by a set of interrelated abstract classes that define the shared features of a software family. Frameworks consist of design specifications and code and build upon standard object-oriented concepts that students are taught in undergraduate classes. Simple examples can be used to illustrate the concept of frameworks [5].

This paper describes a course on the principles and practice of developing software families. It is oriented to the needs of advanced undergraduate or beginning graduate students and has two goals:

- The students should learn concepts and methods relevant for the design of software families. For most students, this involves learning how to think at higher levels of abstraction than they were required to do in lower-level programming classes.
- The students should learn appropriate technologies for designing and implementing a software family as a framework in an object-oriented language.

The course is organized in a bottom-up fashion. It explains the basic concepts, shows how they are used, gives students experience in using them, and then moves on to higher level concepts. The course has three parts: basic concepts and methods, framework construction, and generalization techniques. Sections 2 through 4 discuss these three parts of the course. Section 5 discusses a prototype offering of the course from the perspectives of the students and the instructor. Section 6 concludes the paper.

2. ENCAPSULATING SECRETS

The objective of the first part of the course is to lay a foundation for software families by reviewing the key concepts and techniques that underlie their design and implementation as frameworks. These concepts include modules, information hiding, abstract interfaces, program families, and object-oriented programming techniques.

What prerequisites can we assume the students will have? It is reasonable to assume that the students have object-oriented programming (OOP) expertise at the level expected of third-year undergraduates in a computing science program. The students should have familiarity with basic concepts such as classes, inheritance, delegation, and polymorphism and reasonable skill in applying these concepts using Java or C++. This course does not assume that the students have taken a formal course in "software engineering" previously. A good approach to the course is to adapt the ideas from classic papers by Parnas and his colleagues to the contemporary scene.

This part of the course uses the papers [14], [16], and [3] to introduce students to the fundamental concepts of module, information hiding and abstract interface. It also uses the papers [15], [17], and [18] to introduce the idea of program families. These papers are collected in [11].

Students commonly consider a module to be a unit of code such as a subroutine or a class. However, Parnas defines a *module* as "a work assignment given to a programmer or group of programmers" as a part of a larger effort to construct a software system [16]. His goals are to have modules that can be developed independently, that can be changed

with minimal effects on other modules, and that enable the overall system to be comprehended by examining one module at a time.

Often, students approach the design of a system by breaking it into several processing steps and defining each step to be a module. Parnas takes a different approach. He advocates the use of a principle called *information hiding* to guide decomposition of a system into a modular structure satisfying his goals [14]. This principle means that each module should hide a design decision (i.e., its *secret*) from the other modules. In particular, the designer should choose to hide an aspect of the system that is likely to change from one version to another. If two aspects are likely to change independently, they should be secrets of separate modules. The aspects that are unlikely to change can be represented by the overall modular structure and the interactions (i.e. connections) among the modules. The basics of information hiding can be explained in one lecture in a typical college class. However, the principle "is actually quite subtle" and usually "takes at least a semester of practice to learn how to use it" effectively [19].

Students commonly think an interface is just a set of operation signatures. Britton takes a more general view, defining the "interface between two programs" to consist "of the set of assumptions that each programmer needs to make about the other program in order to demonstrate the correctness of his own program." [3] In addition to an operation's signature, this list of assumptions must also include information about the meaning of an operation and of the data exchanged, about restrictions on the operation, and about exceptions to the normal processing that may arise. In this context, students should learn to be aware of the semantics of operations as well as their syntax.

To be useful, an information-hiding module must be described by an interface (i.e. set of assumptions) that does not change when one module implementation is substituted for another. This is called an *abstract interface* because it is an interface that represents the assumptions that are common to all implementations of the module [3,16]. As an abstraction, it concentrates on the essential nature of the module and obscures the incidental aspects that vary among implementations.

Information hiding modules with abstract interfaces are appropriate units for construction of software families. Using these principles, the technique is to define a software family by giving the "specifications of the externally visible collective behaviors" [15] of the modules instead of the internal implementation details. The technique works by identifying "the design decisions which *cannot* be common properties of the family" [15] and hiding each as a secret of a module.

The programming project in this part of the course requires the students to construct a small system consisting of information-hiding modules with appropriately defined abstract interfaces. The goal is to develop a system that enables an implementation of a module to be replaced by another without rippling changes to other modules. The classic KWIC (KeyWord In Context) problem used in [14] is an appropriate choice for the project. As Shaw notes in [23], "the problem is appealing because we can use it to illustrate the effects of changes on software design." That is exactly what students must learn to confront—to build a system and then be faced with the task of changing the system in many, often unanticipated, ways. These modified systems are different members of a software family.

3. EXTENDING HOT SPOTS

The objective of the second part of the course is to introduce the concepts and techniques for constructing and using software frameworks. The source materials include papers or book excerpts on OOP reuse techniques [12], design patterns [9], and framework case studies [4]. It uses small concrete examples [5] and exercises to make it easier for students to develop an understanding of the framework techniques. This concrete experience provides a good basis for the students to work with the more abstract ideas in the third part.

A *software framework* is a generic application that provides the skeleton upon which various customized applications can be constructed. Each framework consists of some common aspects that are reusable, called *frozen spots,* and certain variable aspects that are application specific, called *hot spots* [22]. The frozen spots are embodied in the overall collaborative structure of the classes in the framework and concrete implementations of various methods and classes. The hot spots are represented as abstract base classes (or interfaces). A particular application can be built by providing appropriate implementations of the relevant hot spot abstractions. The common behaviors of the application family are represented by concrete *template methods* in the base classes that invoke the variable behaviors by calling *hook methods* in the hot spot abstractions.

This part of the course presents two principles for framework construction—unification and separation [8]. The *unification principle* uses inheritance to implement the hot spot subsystem. Both the template and hook methods are defined in the same abstract base class. The hook methods are implemented in subclasses. The *separation principle* uses delegation to implement the hot spot subsystem. The template methods are implemented in a concrete client class; the hook methods are defined in a separate abstract class and implemented in its subclasses. The template methods thus delegate work to an instance of the subclass that implements the hook methods.

Frameworks are generally built using design patterns [9] that are structured to fit with the concept of hot spots and frozen spots [22]. In particular, this course presents two patterns, corresponding to the two framework construction principles, as the primary techniques for implementing the basic structure of frameworks; the Template Method pattern and the Strategy pattern.

The *Template Method pattern* uses the unification principle. In using this pattern, a designer should "define the skeleton of an algorithm in an operation, deferring some steps to a subclass," to allow a programmer to "redefine the steps in an algorithm without changing the algorithm's structure." [9] It captures the commonalities in the template method in a base class while encapsulating the differences as implementations of hook methods in subclasses, thus ensuring that the basic structure of the algorithm remains the same [8].

The *Strategy pattern* uses the separation principle. In using this pattern, a designer should "define a family of algorithms, encapsulate each one, and make them interchangeable" to allow a programmer to let "the algorithm vary independently from the clients that use it." [9] The Strategy pattern employs delegation. That is, the pattern extends the behavior of a client class by calling methods in another class instead of creating a subclass of the client. The common aspects (template methods) are captured in

the concrete methods of the client; the variable aspects (hook methods) are declared in the abstract Strategy class and implemented by its subclasses. The behavior of the client class can thus be changed by supplying it with instances of different Strategy subclasses.

This part of the course uses a small example framework to illustrate the techniques and to serve as the basis for a programming project. The divide and conquer family of algorithms is a simple example that can be used to illustrate both approaches to framework design. It consists of a set of algorithms that should be known to the students. Hence, the application domain should be easy to explain [5]. In the associated project, students can be given the framework and asked to construct applications. This requires an understanding of the framework's design at a level sufficient to use it, without requiring the students to develop their own framework abstractions.

4. EXPLORING GENERALIZATIONS

The objective of the third part of the course is to introduce students to concepts and techniques for creating abstractions appropriate for implementation as a software framework. Source materials for this part include papers on the systematic generalization approach to framework design [22], on commonality and variability analysis [1], and on framework case studies [21].

In most nontrivial frameworks, it is not possible to come up with the right hot spot abstractions just by thinking about the problem informally. Using ad hoc methods, three implementation cycles are often needed to develop a sufficient understanding of the domain to construct good abstractions [20]. However, Schmid advocates an approach to object-oriented framework design that systematizes the process of constructing frameworks [22]. In his *systematic generalization* methodology, framework designers take the design for a specific application within the family and convert it into a framework design by a sequence of generalizing transformations. Each transformation corresponds to the introduction of a hot spot abstraction into the structure. The methodology proposes techniques for analyzing the hot spot and constructing an appropriate design for the hot spot subsystem. Schmid's approach has much in common with the commonality/variability analysis techniques for developing software product lines, another form of software family [1].

This part of the course adopts the systematic generalization approach of Schmid, at least informally. One exercise is for the students to carry out the hot spot analyses for a number of problems. This gives them experience in discovering possible points of variability, determining which ones are relevant, and characterizing their natures.

As in the previous section, the programming project should concern design of a small framework for some problem domain that is easy to explain to students. However, unlike the previous section, this problem should require the students to carry out a careful analysis to discover the hot spots and to design appropriate hot spot subsystems. One appropriate problem is *cosequential processing,* which concerns the coordinated processing of two ordered sequences to produce some result, often a third ordered sequence [7]. One key aspect of cosequential processing is that both input sequences must be ordered according to the same total ordering. Another key aspect is that the processing should, in general, be incremental. That is, only a few elements of each

sequence (perhaps just one) are examined at a time. This important family includes set operations [10] and sequential file update applications [6].

5. DISCUSSION

In the Fall 2003 semester, the first author taught a prototype of this course to a class of 32 graduate students, which included the second and third authors. In general, the instructor and students were pleased with the course content and approach. This section relates a few additional observations the authors have about the course based on their experiences as students and instructor for the prototype offering.

Students are generally adept in writing programs that satisfy the requirements of a specific application. The design process in such cases is aimed more toward satisfying current requirements than to providing for any future changes. For students with such a mindset, the jump to thinking in terms of future changes with respect to creating families is huge. Not only does it involve advanced programming skills but also a more comprehensive approach toward analysis and design. Students are forced to think at a higher level of abstraction than they have had to do in previous work.

While the ideas of creating families seem reasonable in theory, the actual approach to design, analysis, and implementation needs practice for the ideas to become ingrained in the students' minds. This course provides the necessary introduction to the various ideas associated with software families and encourages the students to think abstractly and "for future change." These abstract ideas are made more concrete by providing the students practice by means of relevant assignments.

Students considered the class to be challenging. Some found the change in thought process to be difficult and the concepts "too abstract." This seemed to be a reflection of those students' programming skills as they seemed to understand the ideas in general but were unable to implement them easily. Some skilled programmers underestimated the time needed to complete the programming projects. The assignments required more design effort than the simpler assignments in previous classes. Multiple versions of the programs were also required for a reasonable solution to the assignments.

This course provided a sufficient introduction to the ideas of software product lines and frameworks, but it did not develop the level of expertise that some students desired. The students had a good opportunity to think in terms of larger software systems than most had previously. However, only three programming projects were given. That was an insufficient number to ensure that the students fully grasped the concepts. If possible, a subsequent offering of the course should provide the students with more design and programming practice so they can improve their skills. Maybe a follow-up projects course can be introduced to enable students to refine their skills on larger, more realistic problems. Such students would be better prepared to design and implement software families in an industrial setting.

From the instructor's perspective, the use of original source materials had mixed results. On the one hand, the papers provided the simple and complete explanations that original materials often do. On the other hand, discussing key concepts in terms of technologies that are two or three decades old (in the case of the Parnas papers) made the ideas seem confusing and out-of-date to contemporary students accustomed to object-

oriented programming languages. Covering the concepts in the Parnas papers took longer than was probably necessary to convey the desired concepts and techniques. In a future offering of the course, lecture notes should be created that cover at least some of the important concepts and that use programming examples in an OOP language.

6. CONCLUSION

The growth of academic and industrial interest in software product lines [2,13] indicates that more and more organizations are choosing to approach significant software development tasks using software family methods. Programmers and software engineers thus should seek to acquire the intellectual and practical skills needed for the design and implementation of software product lines and frameworks. To prepare students for tomorrow's programming and software engineering positions, computing science and software engineering curricula should include the relevant concepts, methods, and experiences in their courses.

This paper describes the structure and content of one such course. This course is built around the concepts of encapsulated secrets (i.e., information-hiding and abstraction), extensible hot spots (i.e., abstracted points of variation), and systematic generalization of applications to construct software frameworks. Mastery of these concepts and techniques should prepare students to begin to design software families. With sufficient practice, the students can develop the skills needed to meet the challenges of tomorrow's software applications.

7. ACKNOWLEDGEMENTS

This work was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)."

8. REFERENCES

[1] M. Ardis, N. Daley, D. Hoffman, H. Siy, and D. Weiss. "Software Product Lines: A Case Study," *Software—Practice and Experience*, Vol. 30, pp. 825-847, 2000.

[2] J. Bosch, H. Obbink, and A. Maccari. "Research Themes and Future Trends," In F. van der Linden, editor, *Software Product Family-Engineering*, LNCS 3014, Springer 2004.

[3] K. H. Britton, R. A. Parker, and D. L. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules," In *Proceedings of the 5th International Conference on Software Engineering*, pp. 95-204, March 1981.

[4] H. C. Cunningham and J. Wang. "Building a Layered Framework for the Table Abstraction," In *Proceedings of the ACM Symposium on Applied Computing*, pp. 668-674, March 2001.

[5] H. C. Cunningham, Y. Liu, and C. Zhang. "Using the Divide and Conquer Strategy to Teach Java Framework Design," In *Proceedings of the International Conference on the Principles and Practice of Programming in Java (PPPJ) Conference*, pp. 40-45, June 2004.

[6] E. W. Dijkstra. "Updating a Sequential File," Chapter 15, In *A Discipline of Programming*, pp. 117-122, Prentice Hall, 1976.

[7] M. J. Folk, B. Zoellick and G. Riccardi. *File Structures: An Object-Oriented Approach with C++*, Third edition, Addison-Wesley, 1998.

[8] M. Fontoura, W. Pree, and B. Rumpe. *The UML Profile for Framework Architectures.* Addison-Wesley, 2002.

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

[10] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*, Wiley, Third Edition, 2004.

[11] D. M. Hoffman and D. M. Weiss, editors. *Software Fundamentals: Collected Papers by David L. Parnas,* Addison-Wesley, 2001.

[12] R. E. Johnson and B. Foote. "Designing Reusable Classes," *Journal of Object-Oriented Programming*, Vol. 1, No. 2, pages 22-35, June/July 1988.

[13] R. L. Nord, editor. *Software Product Lines*, LNCS 3154, Springer-Verlag, 2004.

[14] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, pp.1053-1058, 1972.

[15] D. L. Parnas. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering,* Vol. SE-2, No. 1, pp. 1-9, March 1976.

[16] D. L. Parnas. "Some Software Engineering Principles," *Infotech State of the Art Report on Structured Analysis and Design, Infotech International*, 10 pages, 1978. Reprinted in *Software Fundamentals: Collected Papers by David L. Parnas*, D. M. Hoffman and D. M. Weiss, editors, Addison-Wesley, 2001.

[17] D. L. Parnas. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 1, pp. 128-138, March 1979.

[18] D. L. Parnas, P. C. Clements, and D. M. Weiss. "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 259-266, March 1985.

[19] D. L. Parnas. "Software Design," In D. M. Hoffman and D. M. Weiss, editors. *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, 2001.

[20] D. Roberts and R. Johnson. "Patterns for Evolving Frameworks," In R.Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design* 3, pp. 471-486, Addison-Wesley, 1998.

[21] H. A. Schmid. "Creating Applications from Components: A Manufacturing Framework Design," *IEEE Software*, Vol. 13, No. 6, November 1997.

[22] H. A. Schmid. "Framework Design by Systematic Generalization," In M. E. Fayad , D. C. Schmidt, and R. E. Johnson, editors, *Building Application Frameworks*, pp. 353-378, Wiley, 1999.

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