Keeping Secrets within a Family: Rediscovering Parnas

H. Conrad Cunningham
Computer Science
University of Mississippi
University, MS, 38677

Cuihua Zhang
Computer & Information Systems
Northwest Vista College
San Antonio, TX 78251

Yi Liu
Computer Science
University of Mississippi
University, MS 38677

Abstract

David Parnas wrote several papers in the 1970's and 1980's that are now considered classics. The concepts he advocated such as information hiding and use of abstract interfaces are generally accepted as the appropriate way to design nontrivial software systems. However, not all of what he proposed has been fully appreciated and assimilated into our practices. Many of his simple, elegant ideas have been lost amongst the hype surrounding the technologies and methods that have arisen in the past two decades. This paper examines Parnas's ideas, especially his emphasis on program families, and proposes that college-level computing science and software engineering curricula should renew their attention to these very important principles and techniques and present them in the context of contemporary software development.

Keywords: program family, information hiding, abstract interface, module, software product line.

1. Introduction

Software product lines and frameworks are increasing in importance among software developers and researchers; however, many of the key ideas behind them were first published in the 1970's and 1980's in classic papers by David Parnas and his colleagues [8]. Parnas argues that a "software designer should be aware that he is not designing a single program but a family of programs." [10] Computing science and software engineering students should develop such awareness early in their studies.

A *software product line* is often defined as "a family of products that share common features to meet the needs of a market area." [1] A software framework is a special case of a software product line. 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 set of related programs. The motivation for product lines and frameworks is to take advantage of the commonalities among the members of the product line to lower the overall cost of producing and maintaining a group of related software systems.

Since the foundation of software product lines and frameworks is what Parnas proposed in his papers, an examination of the concepts in these papers (collected in [5]) can still reveal much of value to current-day software developers and researchers. Many of the lessons taught in these works should also be incorporated into our college-level teaching.

This paper examines several of the lessons on the design of program families taught by Parnas that are still important for contemporary students to learn. Many of his simple, elegant ideas have been lost amongst the hype surrounding the technologies and methods that have arisen in the past two decades. The paper reviews Parnas and his colleagues' concepts of modularization, information hiding, abstract interfaces, and program families. An example Table framework from [3], designed and implemented with the Java programming language, is used to illustrate the concepts. After each topic is reviewed, a discussion of part of the example is provided to enhance the understanding of the topic.

2. Table framework example

The Table Abstract Data Type (ADT) represents a collection of records, each of which consists of a finite sequence of data fields. The value of one (or a composite of several) of these fields uniquely identifies a record within the collection; this field is called the key. The values of the keys are elements from a totally ordered set. The operations provided by the Table ADT allow a record to be inserted, retrieved, updated, and deleted using its key to identify it within the collection.

For the purposes of this paper, we consider the design of the Table framework to have the following requirements [3]:

1. It must provide the functionality of the Table ADT for a large domain of client-defined records and keys.
2. It must support many possible representations of the Table ADT, including both in-memory and on-disk structures and a variety of indexing mechanisms.
3. It must separate the key-based record access mechanisms from the mechanisms for storing records physically.

We approach the design of the Table framework using Parnas's modular specification techniques.

3. Information-hiding modules

Students commonly consider a module to be a unit of code such as a subroutine or a class. When they approach the design of a system, a common tendency is to break the system into several processing steps like steps in a flowchart and to define each step to be a module. Parnas presents a more general view of modules and a different approach to decomposing a system into modules.

3.1 Parnas principles

Parnas defines a *module* as "a work assignment given to a programmer or group of programmers" due to the nature of software engineering, which is multi-person, multi-version [9]. It is desirable for programming environments and languages to support the programmers' work on modules, but it is not essential.

In Parnas's view, the goals of a modularization are to [7]:

1. shorten development time by minimizing the required communication among the groups,
2. make the system flexible by limiting the number of modules affected by significant changes,
3. enable programmers to understand the system by focusing on one module at a time.

To accomplish these goals, it is important that modules be cohesive units of functionality that are independent of one another.

Parnas advocates the use of a principle called *information hiding* to guide decomposition of a system into appropriate modules, i.e. work assignments. He points out that the connections among the modules should have as few information requirements as possible [7].

Information hiding means that each module should hide a design decision from the rest of the modules. This is often called the *secret* of the module. In particular, the designer should choose to hide within a module an aspect of the system that is likely to change as the program evolves. If two aspects are likely to change independently, they should be secrets of separate modules. The aspects that are unlikely to change are represented in the design of the interactions (i.e. connections) among the modules. This approach supports the goal of changeability (goal 2). When care is taken to design the modules as clean abstractions with well-defined and documented interfaces, the approach also supports the goals of independent development (goal 1) and comprehensibility (goal 3).

3.2 Table framework design

Consider the Table framework example. At the top level, there seems to be three primary dimensions of change in the system. We can make each of these the secret of an information-hiding module. The modules and their secrets are as follows:

Table Access. This module provides key-based access to the collection of records stored in the table. The secret of the module is the set of data structures and algorithms used to provide the index for access to the records. For example, this might be a simple index maintained in a sorted array, a hash table, or a tree-structured index.

Record Storage. This module manages the physical storage for the records in the table. The secret of the module is the detailed nature of the storage medium. For example, the storage medium might be a structure in the computer's main memory or a random-access file on disk.

Client Record. This module provides the key and record data types needed by the other modules; the client of the Table framework must provide an implementation of the module appropriate for the particular application. The secret of the module is the structure of the client's record, including the identification of the key field, its data type and ordering relation and identification of the non-key fields and their data types.

3.3 Perspective

Classes and modules can be easily confused since they are both self-contained units, and they do share some of the same goals and characteristics. The difference is, however, that a typical work assignment, or a module, that needs to support change is often larger than a single class; it may contain several related classes, and these classes should be designed and maintained as a unit.

Information hiding has, of course, been absorbed into the dogma of object-oriented programming. However, information hiding is often oversimplified as merely hiding the data and their representations [15]. The secret of a well-designed module may be much more than that. It may include such knowledge as a specific functional requirement stated in the requirements document, the processing algorithm used, the nature of external devices accessed, or even the presence or absence of other modules or programs in the system [7, 10, 11]. These are important aspects that may change as the system evolves.

Information hiding is one of the most important principles in software engineering. At first glance, it seems to be an obvious technique. However, further study reveals it to be a subtle principle that takes considerable practice to apply well in software design. Students in computing science programs should learn the principle and how to apply it in a variety of circumstances. They also need to learn to design modules that are coherent abstractions with well-defined interfaces.

4. Abstract interfaces

When students specify the interface for a class or other program unit, they typically identify the set of operations (procedures and functions) that can be called from outside the unit. That is, they consider the return type of each operation and its signature—the name and the number, order, and types of its parameters. This describes the syntax, or structure, of the interface. However, the students also need to be taught to describe the semantics, or expected behaviors, of the operations explicitly.

4.1 Parnas principles

Parnas and his colleagues advocate that the "interface between two programs consists 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." [2] 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 arise in response to undesired events.

In Parnas's information-hiding approach, each module must hide its secret from the other modules of the system. The module's secret is a design decision that changes from one implementation of the module to another. To be useful, the module must be described by an interface (i.e. set of assumptions) that does not change when one module implementation is substituted for another. Parnas and his colleagues call this an *abstract interface* because it is an interface that represents the assumptions that are common to all implementations of the module [2,9]. As an abstraction, it concentrates on the essential nature of the module and obscures the incidental aspects that vary among implementations.

Parnas and his colleagues take an interesting two-phase approach to the design of abstract interfaces, one that they argue is especially important in the design of interfaces to "devices" in the environment. The method constructs two partially redundant descriptions of an abstract interface. They are redundant because they describe the same assumptions.

First, the designer carefully studies the possible capabilities of the types of devices that might be used (or module implementations that might be needed) and then explicitly states in plain English the list of assumptions that can be made about all the devices (module implementations) in the set. This list is meant for people who are experts in the application domain, but who might not be skilled programmers. This plain English list makes invalid assumptions easier to detect.

Second, the designer constructs a list of the specific operations in the interface and describes the signature and semantics of each operation. Every capability implied in the specifications of the operations must be explicitly stated in the list of assumptions. These programming constructs can be later used in programs.

4.2 Table framework design

Consider the abstract interface for the Client Record module in the Table framework example. We want, as much as possible, to let clients (users) of the Table framework define their own record and key structures. However, the Table Access module must be able to extract the keys from the records and compare them with each other. Thus we require that the Client Record module be implemented so that assumption 1 given in Figure 1 holds.

Similarly, the Record Storage module must be able to store the records on and retrieve them from the physical slots on the storage medium. For in-memory implementations of the Record Storage module, this is not a problem; they can simply clone the record (or perhaps copy a reference to it). However, disk-based implementations must write the record to a (random-access) file and reconstruct the record when it is read. In general, the Record Storage module may need to convert the client's record to and from a sequence of bytes. Thus we specify that assumption 2 shown in Figure 1 must hold.

The programming interface for the Client Record module is shown in Figure 2. It consists of three Java interfaces with a total of five methods.

1. Records are objects from which the keys can be extracted and compared using a total ordering.
2. As needed, records can be converted to and from a sequence of bytes. It is possible to determine the number of bytes in the record.

Figure 1. Assumption list for Client Record

The built-in Java interface `Comparable` satisfies the requirement for the keys [3]. Any class that implements this interface must provide the method `compareTo()` that compares the associated object with its argument. Clients can use any existing `Comparable` class for their keys or implement their own in the Client Record module.

We introduce the Java interface `Keyed` to represent the type of objects that can be stored and retrieved by the Table Access module [3]. Any class that implements this interface must implement the method `getKey()` that extracts the key from the associated record. Clients must supply a class in the Client Record module that provides an appropriate implementation of the `Keyed` interface. The Table Access module can use this method to extract a key and then use the key's `compareTo` method to do the comparison. The details of the record structure are otherwise hidden in the Client Record module.

We also introduce the Java interface `Record` to represent the type of objects that can, if needed, be converted to and from a sequence of bytes [3]. This interface has the three methods `writeRecord()`, `readRecord()`, and `getLength()` to write the record, read the record, and return the size of the record, respectively. The Record Storage module calls the `Record` methods when it needs to read or write the physical record. The code in the `Record`-implementing class (e.g., defined in the Client Record module) converts the internal record data to and from a stream of bytes. The Record Storage module is responsible for routing the stream of bytes to and from the physical storage medium.

An implementation of the Client Record module would thus normally consist of a class that implements the Java interfaces `Keyed` and `Record` and a decision on how to represent the record's keys. The latter decision might be to construct some class that implements the built-in Java interface `Comparable` or it might be to choose an existing built-in class that already implements `Comparable`.

```
interface Comparable
   int compareTo(Object key)
   // compares the associated object with argument key and
   // returns -1 if key is greater, 0 if they are equal, and 1 if
   // key is less.
interface Keyed
   Comparable getKey()
   // extracts the key from the associated record
interface Record
   void writeRecord(DataOutput)
   // writes the record to a DataOutput stream
   void readRecord(DataInput)
   // reads the record from a DataInput stream
   int getLength()
   // returns the number of bytes that will be written by
   // writeRecord()
```

Figure 2. Programming interface for Client Record

The abstract interfaces of the Table Access and Record Storage modules are described in [3]. The Table Access module has a Java interface `Table` that represents the Table ADT as described in Section 2; this interface has (at least) seven operations. The Record Access module has a pair of closely related abstractions represented by the Java interfaces `RecordStore` and `RecordSlot`. These abstractions manage the physical storage facility; collectively, they have six operations. The semantics of the operations in these modules are given in terms of formal design contracts and information models. The key contribution of [3] is the specification of modules with abstract interfaces that enabled the separation of the key-based access mechanism in the Table Access module from the physical storage mechanism in the Record Storage module.

4.3 Perspective

Parnas's ideas on abstract interfaces [2] have been refined by others and incorporated into various methods such as Meyer's design by contract [6]. However, the method of using two partially redundant descriptions has not been used extensively. It deserves more attention in a world where a program may require services from the interfaces of many other programs and, in turn, provide other programs interfaces to its services [14].

Like information hiding, design of elegant and effective module interfaces is an important skill that computing science students should learn. Computing science programs should present principles for effective design of abstract interfaces and help students learn the subtleties of their application. Abstract interfaces and information hiding are the key concepts enabling the construction of program families.

5. Program families

Students (and many professionals) often practice code reuse in an informal manner. When given a new problem to solve, they may find a program for a similar problem and, using a text editor, modify the program to get a solution to the new problem. This may be a reasonable approach for small, simple programs in a situation in which it is legitimate to adapt the existing code, a solution is needed quickly, there is little concern about the efficiency or elegance of the program, and the program will only be used for a short period of time. However, if these conditions do not hold, this undisciplined technique can lead to a chaotic situation where many versions of a whole program must be maintained simultaneously in source code. Changes and error corrections cannot be conveniently and reliably moved among the different versions as needed. To overcome these problems, we should teach students a disciplined technique from the beginning.

Students should thus be taught more systematic methods to design and implement multi-version programs. Information hiding modules and abstract interfaces are the basic concepts needed to design such programs. The information hiding approach seeks to identify aspects of a software design that might change from one version to another and to hide them within independent modules behind well-defined abstract interfaces. Because one implementation can be easily substituted for another, this type of design can be considered as defining a program family.

5.1 Parnas principles

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." [8] In his view, a family member is developed by incrementally identifying the common aspects of the family and representing the intermediate forms of the program as they evolve. These intermediate forms should be documented fully and saved for development of future family members. Instead of developing a new family member by modifying a previous member, the designer finds the appropriate intermediate representation and restarts the design from that point.

In Parnas's *module specification* approach [8], which is based on the principles of information hiding and abstract interfaces, the technique is to define a software system by giving the "specifications of the externally visible collective behaviors" of the modules instead of the internal implementation details. It works by identifying "the design decisions which *cannot* be common properties of the family" and hiding each as a secret of a module.

5.2 Table framework design

Again consider the Table framework. The analysis of the problem domain led to a design in which the primary expected sources of change are encapsulated within three information-hiding modules with carefully defined abstract interfaces. This generated a program family in which the different members vary according to their selections for the Table Access, Record Storage, and Client Record module implementations. The members of the family discussed in [3] include two different Table Access module implementations; one is a simple in-memory index that uses sorted arrays of keys and binary search, and the other is an in-memory hashed index. Similarly, there are three implementations of the Record Storage module; two of these use in-memory data structures, and the third uses a random-access file on disk. A client can configure a system by combining implementations of the Table Access and Record Storage modules with an implementation of the Client Record module with appropriate definitions of the records and keys.

5.3 Perspective

Since Parnas's paper [8] on the concept of program families first appeared, considerable interest has grown in what are now usually called *software product lines* [1]. 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." [12] He warns, "If you are developing a family of programs, you must do so consciously, or you will incur unnecessary long-term costs." [12] This issue should be addressed in computing science and software engineering curricula.

6. Teaching the Parnas principles

Three decades after Parnas first articulated the principle, he argues that *information hiding* is still "the most important and basic software design principle." [12] Yet, he observes that "it is often not understood and applied" despite being the intellectual underpinning of recent ideas such as object-oriented and component-based programming. He laments that he commonly sees "programs in both academia and industry in which arbitrary design decisions are implicit in intermodular interfaces making the software unnecessarily hard to inspect or change." [12]

Computing science and software engineering educators must assume part of the blame for this situation and accept much of the responsibility for remedying it. The basics of information hiding can be explained in one lecture in a typical college-level class. However, the principle "is actually quite subtle" and usually "takes at least a semester of practice to learn how to use it." [12] Educators should go beyond the superficial attention given in textbooks and incorporate application of the principle into most aspects of software design courses. The examples presented should be designed according to the principle and the secrets of the modules should be articulated during design and explicitly documented. Student work should be evaluated on how well it applies the principle.

Information-hiding modules must, of course, have interfaces that hide the secrets of the modules. The interfaces must be "less likely to change than the 'secrets' that they hide." [9] This is not an easy process. The design of an appropriate *abstract interface* "requires both careful investigation and some creativity" [9] on the part of the software designer. As with information hiding, the concept of abstract interfaces is not difficult to explain. It is, however, a subtle concept that takes considerable practice to be able to apply well.

Educators should seek to give students appropriate instruction on the concepts and techniques for building good abstract interfaces and provide experiences in building such interfaces. Although Parnas's two-phase procedure [2, 9] has not been used extensively, it is a good approach to use in education. The first phase focuses the students' attention on identifying explicitly the common properties of the set of all likely versions of a module. Since it uses English text, there are no new notations or technologies to learn. The second phase focuses the students' attention on designing specific interfaces that are consistent with the assumptions identified in the first phase. If more formality is desired, then the second phase can be augmented by an approach such as design by contract [6]. Instructors should discourage the common practice of diving immediately into the definition of the operations in the second phase, bypassing the first phase. The examples and exercises in a course should reflect the abstract interface approach. Student work should be evaluated based on how well it applies the approach and how effectively the interfaces hide the changeable design decisions of the module.

The principles of information hiding and abstract interface design are key underlying concepts for the construction of *program families*. However, design of a program family requires more. The designers must analyze the application domain and explicitly identify the common and the variable aspects of the family members [1]. The common aspects can be incorporated into the module structure and the variable aspects made secrets of modules. Techniques for identifying these commonalities and variabilities should be taught in software design courses.

The techniques and tools for building product lines can be quite complex, involving special-purpose translators and configuration tools [1]. Hence, general product line construction is difficult to teach within the confines of a college course. However, software frameworks are more accessible to students and professors. Frameworks consist of design specifications and program 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 and serve as a basis for programming exercises [4]. An interesting possible approach to teaching framework design is to generalize the design of a specific application from the family using Schmid's techniques for *hotspot analysis* and *systematic generalization* [13]. Construction of program families can be taught successfully if explicit attention is given to the underlying principles and these principles are consistently reinforced over time.

7. Conclusion

In 1979, David Parnas wrote that a "software designer should be aware that he is not designing a single program but a family of programs." [10] In a number of papers published in the 1970's and 1980's, Parnas and his colleagues codified the principles and practices for engineering such program families. They refined and demonstrated their software engineering approach in a difficult real-world setting, the reimplementation of the hard realtime Operational Flight Program (OFP) for the U.S. Navy's A-7E airplane [11].

The thesis of this paper is that contemporary students, once they have mastered the skills for development of individual programs, should be taught to approach program design as the development of a whole family of software products. Furthermore, it argues that the principles and practices laid down by Parnas a quarter century ago are still applicable today. Perhaps they are not fully appreciated, and sometimes they may get lost amongst all the hype surrounding the technologies and tools that have emerged in recent years. However, the principles from Parnas's classic papers are still valuable for current-day students and practitioners to study and apply in their software development activities.

The Parnas methods can be characterized by two key ideas, information hiding and abstract interfaces. The information hiding principle says that a system should be decomposed into modules where each module hides a single design decision (secret) that may change independently from other design decisions about the system. The abstract interface of a module is a listing of all the assumptions that a user of the module may make about the module. These assumptions must not reveal the secret of the module and must be well-defined and carefully documented.

The Parnas approach to the design of software families can be summarized as "keeping secrets within a family." It seeks to identify aspects of a design that might change from one version to another of an application and make them secrets of independent modules with well-defined abstract interfaces. The modules hide their secrets from each other. Because this approach enables one implementation of a module to be substituted for another easily, this type of design defines a program family.

8. Acknowledgements

The work of Cunningham and Liu was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)." The prototype implementations of the Table framework [3] were done by Jingyi Wang. Pallavi Tadepalli reviewed this paper and suggested several improvements.

9. 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] 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.

[3] 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.

[4] H. C. Cunningham, Y. Liu, and C. Zhang. "Using the Divide and Conquer Strategy to Teach Java Framework Design," To appear in the *Proceedings of the Principles and Practice of Programming in Java (PPPJ) Conference,* 6 pages, June 2004.

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

[6] B. Meyer. *Object-Oriented Program Construction*, Second edition, Prentice Hall, 1997.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] H. A. Schmid. "Framework Design by Systematic Generalization," In M. E. Fayad and R. E. Johnson, editors, *Domain-Specific Application Frameworks*, Wiley, 2000.

[14] J. Waldo. "Introduction: A Procedure for Designing Abstract Interfaces for Device Interface Modules," In *Software Fundamentals: Collected Papers by David L. Parnas,* D. M. Hoffman and D. M. Weiss, editors, Addison-Wesley, 2001.

[15] D. M. Weiss. "Introduction: On the Criteria to Be Used in Decomposing Systems into Modules," In *Software Fundamentals: Collected Papers by David L. Parnas,* D. M. Hoffman and D. M. Weiss, editors, Addison-Wesley, 2001.