CSci 555: Functional Programming
Modular Design

H. Conrad Cunningham

23 March 2019

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848

University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of March 2019 is a recent version of Firefox from Mozilla.

Modular Design

These lecture notes accompany the lecture notes on Data Abstraction [Cunningham 2019f].

Introduction

In the provocative 1986 essay "No Silver Bullet–Essence and Accidents in Software Engineering," software engineering pioneer Fred Brooks asserts that "building software will always be hard" because software systems are inherently *complex*, must *conform* to all sorts of physical, human, and software interfaces, must *change* as the system requirements evolve, and are inherently *invisible* entities [Brooks 1986]. A decade later Brooks again observes, "The best way to attack the essence of building software is not to build it at all" [Brooks 1995]. That is, software engineers should reuse both software and, more importantly, software designs.

What was true in the 1980s is still true today. Although software development tools and practices have evolved, removing some of the "accidental" properties of software development, the essential difficulties remain. Complexity continues to increase. The interfaces to which software must conform continue to change quickly and increase in number. The requirements on software continue to evolve, driven by inexorable changes in the environment and increasing penetration of computerized processes into new aspects of society. Globalization generates new requirements, which arise from both new opportunities and new competition.

We often develop software systems in multiperson teams. In many cases, the teams are geographically distributed, perhaps even across national boundaries. Communication among team members adds complexity to software development.

How should we approach software development in this contemporary context?

As a starting point, let us again look to a software engineering pioneer: David Parnas. Parnas focuses on how to decompose a system into modules to achieve robustness with respect to change and potential reuse of software. He stresses the clarity of thought more than the sophistication of languages and tools.

Although Parnas and his colleagues published their ideas on *modular specification* in the 1970s and 1980s [Parnas 1972, 1976, 1979, 1985; Britton 1981], the ideas are as relevant today as they were when first published.

Family of Table Implementations

These notes illustrate the approach using the development of a family of implementations of a Table Abstract Data Type (ADT).

A *Table ADT* is an abstraction of a widely used set of data and file structures. It 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*. For the purposes here, the values of the keys are assumed to be elements from a *totally ordered set* (i.e. each element is <, =, or >every element). The Table ADT's operations enable us to store and retrieve a record using its key to identify it within the collection.

By approaching the Table ADT design as a *software family*, we seek to exploit the *commonalities* (the features that are "the same" for all likely implementations) while limiting the negative effects of the *variabilities* (the features that are potentially "different" among implementations) [Coplien 1998] on the development and maintenance of the software.

As Cunningham does in several published papers [Cunningham 2001, 2004, 2010] on this topic, here we use object-oriented programming concepts and the Java language to describe the implementation. However, we can use similar concepts found in procedural or functional programming languages.

Information-Hiding Modules

When programmers approach the design of a software system, they tend to break the system into several processing steps, similar to those in a flowchart, and define each step to be a module. This often results in design that is difficult to change when the requirements for the software change.

Parnas advocates a more general view of modules and an approach to decomposing a system that yields a system that is easier to modify. It seeks to isolate the aspects most likely to change inside a module, instead of spreading them across several modules.

Parnas principles: Modules and information hiding

Parnas defines a *module* as "a work assignment given to a programmer or group of programmers" [Parnas 2001b]. It is desirable for programming environment and language features to support the programmers' work on modules, but it is not essential.

In Parnas's view, the goals of a modularization are to [Parnas 1972]:

1. shorten development time by minimizing the required communication among the groups (*independent development*)
2. make the system flexible by limiting the number of modules affected by significant changes (*changeability*)

3. enable programmers to understand the system by focusing on one module at a time (*comprehensibility*)

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 [Parnas 1972].

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).

Table family: Modularization

For our purposes here, we consider the design of the Table family to have the following requirements [Cunningham 2001, 2004, 2010].

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.

At the top level, the system seems to have four primary dimensions of likely change. We can make each of these the secret of an information-hiding module. The modules and their secrets are as follows [Cunningham 2010].

Client Record: This module provides the key and record data types for the client-defined records. The client (user) of the Table family must provide an implementation of the module appropriate for the particular application; the other modules use these types. 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.

Table Access: This module provides the client programs key-based access to the collection of (client-defined) records stored in the table. The secret of the module is the set of data structures and algorithms that 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 provides the table with facilities to store and retrieve the table's records using the chosen physical storage medium. The secret of the module is the nature of how the records are stored physically. For example, the physical storage might be a structure in the computer's main memory or a random-access file on disk. (This module uses the facilities of the Externalization module to do the byte-level reading and writing of the records.)

Externalization: This module provides facilities to convert "records" in memory to and from a sequence of bytes on the physical storage medium. The secret of this module is the byte-level nature of the physical data representation on the storage medium.

If we elaborate this design as a library–or API (Application Program Interface)–then the Table Access and Record Storage modules likely would be represented by various program units in the library. The Client Record and Externalization modules would be represented by specifications that the user of the library must implement.

Note: Papers [Cunningham 2001] and [Cunningham 2004] combine the Client Record and Externalization modules into one module. The presentation here follows the presentation in [Cunningham 2010], which carried out a more refined analysis of the commonalities and variabilities present in this problem.

Perspective

In object-oriented languages, classes and modules can be easily confused because 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 (i.e. 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 [Weiss 2001]. 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 [Parnas 1972, 1979, 1985]. 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. As software developers, we should learn the principle and how to apply it effectively in a variety of circumstances. We also need to learn to design modules that are coherent abstractions with well-defined interfaces.

Abstract Interfaces

When programmers 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, programmers also need to describe the *semantics*, or expected behaviors, of the operations explicitly.

Parnas principles: Interface specifications

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" (italics added) [Britton 1981]. 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 [Britton 1981; Parnas 2001]. 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 specification and design of abstract interfaces [Britton 1981], 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 programming 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. (This second specification is like the specifications given in the notes on Data Abstraction.)

Table family: Module interfaces

Client Record

Consider the abstract interface for the Client Record module in the Table family example. We want, as much as possible, to let clients (users) of the Table family 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, the Client Record module must implement records that satisfy the *Client Record Assumptions*:

1. A record is an object from which a key can be extracted.
2. Two keys can be compared using a total ordering.

In a Java implementation of the Table family, the programming interface for the Client Record module consists of two Java interfaces, each with one method as shown below.

```
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
```

Note: In these notes, we follow Cunningham's papers [Cunningham 2001, 2004, 2010], which use an older, non-generic version of Java. A design using the features of Java 5 and later would be similar but could take advantage of generics, thus increasing type safety.

The built-in Java interface `Comparable` satisfies the requirement for the keys [Cunningham 2001, 2010]. Any class that implements this interface must provide

the *callback* method `compareTo()` that compares the associated object with its argument according to total ordering. 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 [Cunningham 2001, 2010]. Any class that implements this interface must implement the *callback* 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.

An implementation of the Client Record module would normally consist of a class that implements the Java interfaces `Keyed` 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`.

Externalization

Now consider the abstract interface to the Externalization module. The Record Storage module must be able to write records to and read records from the physical locations on the chosen storage medium in an appropriate format. 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 this design, we take a low-level approach. The Record Storage module may need to convert the client's record to and from a sequence of bytes. Thus the Externalization module must implement records that satisfy the *Externalization Assumptions*:

1. A record can be converted to a finite sequence of bytes.
2. A record can be reconstructed from a finite sequence of bytes. The process of converting a record to bytes and back results in an equivalent record.
3. It is possible to determine the number of bytes in a sequence corresponding to a record.

In a Java implementation of the Table family, the programming interface for the Externalization module consists of a Java interface with three (callback) methods as shown below.

```
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()
```

We introduce the Java interface `Record` to represent the type of objects that can, if needed, be converted to and from a sequence of bytes [Cunningham 2001, 2010]. This interface has the three methods `writeRecord()`, `readRecord()`, and `getLength()` to write the physical record, read a 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 Externalization 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 Externalization module includes a class that implements the Java interface `Record`. In most situations, the same client record classes will need to implement both the `Keyed` and `Record` interfaces, effectively merging implementations of the Client Record and Externalization modules. As we see below, in some cases a second instance of the Externalization module classes will be needed for the physical records passed between the Table Access and Record Storage modules.

Record Storage

Now consider the abstract interface to the Record Storage module, which depends upon the Externalization Module. This module requires the records it stores and retrieves to satisfy the Externalization Assumptions above. The module must also satisfy the *Record Storage Assumptions*:

1. A record can be stored on the physical storage medium as a sequence of bytes at some location if the sequence is shorter than the specified maximum length.
2. A record can be retrieved from the physical storage medium as a finite sequence of bytes from some location. The record retrieved is equivalent to the one stored at that location.
3. It is possible to allocate and deallocate physical record locations on the physical storage medium.

The programming interface for the Record Storage module consists of a pair of closely related abstractions represented by the Java interfaces `RecordStore` and `RecordSlot`. These abstractions manage the physical storage facility; collectively, they have eight operations as described in paper [Cunningham 2010].

Aside: Paper [Cunningham 2010]] uses an abstract predicate `isStorable(Record)` to help formalize the semantics of the Record Storage module. This abstract predicate would be false when (a) a record cannot be stored at an allocated location for some reason and (b) there are no unused locations to allocate. In these notes, we simplify this condition as follows.

- We express (a) with the condition "shorter than the specified maximum length." However, this might not characterize all situations that could result in failure.
- For (b) we assume the size of the physical storage is unbounded. For example, consider a file. The maximum size of a file might be very large, but it is usually bounded by the size of the disk or limits set by the file system. However, unboundedness does correspond to the way programs normally write to files. They attempt to write and generate an exception if the write fails because of a full disk. Having a full disk is something that cannot easily be checked in advance.

Table Access

Finally, consider the abstract interface to the Table Access module, which depends upon all three other modules. This module requires client records that satisfy the Client Record Assumptions and physically storable records that satisfy the Externalization Assumptions and are shorter than the maximum length supported by the Record Storage module (Record Storage Assumption 1). The Table Access module must also satisfy the *Table Access Assumptions*:

1. A key occurs at most once in the table.
2. A record can be stored in the table using the record's key if no other record with that key occurs in the table.
3. A record stored in the table can be retrieved using its key. The record retrieved is equivalent to the one stored most recently with that key.
4. A record stored in the table can be deleted from the table using its key.
5. It is possible to determine whether there is record with a given key in the table, whether the table is empty or is full, and how many records are stored in the table.

The programming interface to the Table Access module consists of a Java interface `Table` that represents the Table ADT. This interface has eight operations as shown in the paper [Cunningham 2010]. The paper also extends this interface with several different iterator operations.

In most cases, the client records supplied to the Table Access module must satisfy the Externalization Assumptions as well as the Client Record Assumptions.

- In some implementations, the client record is passed through the Table Access module unmodified to the Record Storage module, which must externalize it as a sequence of bytes.
- In other cases (e.g. B-tree indexes), the Table Access module may aggregate several client records into one physical record. This requires the Table Access module to provide the Record Storage module a physical record that satisfies both the Externalization and Record Storage interfaces. This new physical record implementation likely needs the client records to satisfy the Externalization Assumptions.

Perspective

Parnas's ideas on abstract interfaces [Britton 1981] have been refined by others and incorporated into methods such as Meyer's *design by contract* [Meyer 1997]. Meyer's approach involves specification of invariants, preconditions, and postconditions similar to the approach used in the Data Abstraction notes [Cunningham 2018f].

Papers [Cunningham 2001] and [Cunningham 2010] give the semantics of the operations in these Table ADT modules in terms of formal design contracts and information models. The key contribution of this work 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.

Parnas 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 [Waldo 2001].

Like information hiding, design of elegant and effective module interfaces is an important skill that expert software developers should learn. Abstract interfaces and information hiding are the key concepts enabling the construction of software families.

Software Families

We often practice code reuse in an informal manner. When given a new problem to solve, we 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 use a disciplined technique from the beginning.

Information hiding modules and abstract interfaces are the basic concepts needed to design multi-version 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 software or program family to use Parnas's terminology.

Parnas principles: Program families

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" [Parnas 1976]. 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 [Parnas 1976], 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.

Table family: Encapsulating variabilities

Again consider the Table family. The analysis of the problem domain led to a design in which the primary expected sources of change (i.e. the variabilities) are encapsulated within four 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 Client Record, Table Access, Record Storage, and Externalization module implementations.

The members of the family discussed in [Cunningham 2001] 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 and Externalization modules with appropriate definitions of the records and keys.

Perspective

Since Parnas's paper [Parnas 1976] on the concept of program family first appeared, considerable interest has grown in what is now called a *software product line* [Ardis 2000; Weiss 1999] or *software family* (which your instructor prefers). 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" [Parnas 2001]. He warns, "If you are developing a family of programs, you must do so consciously, or you will incur unnecessary long-term costs" [Parnas 2001]. The importance of this issue has not diminished since Parnas's papers on information hiding and program families four decades ago.

Discussion

Three decades after Parnas first articulated the principle, he argued that information hiding is still "the most important and basic software design principle" [Parnas 2001a]. Yet, he observed 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" [Parnas 2001a].

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" [Parnas 2001a].

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" [Parnas 2001b]. This is not an easy process. The design of an appropriate abstract interface "requires both careful investigation and some creativity" [Parnas 2001b] 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.

The principles of information hiding and abstract interface design are key underlying concepts for the construction of software families. However, design of a software family requires more. The designers must analyze the application domain and explicitly identify the common and the variable aspects of the family members [Ardis 2000; Weiss 1999]. The common aspects can be incorporated into the module structure and the variable aspects made secrets of modules.

The techniques and tools for building software product lines can be quite complex, involving special-purpose translators and configuration tools [Ardis 2000; Weiss 1999]. Hence, general product line construction is difficult to teach within the confines of a college course.

However, the *software framework* approach [Johnson 1998] is often more accessible to those first learning to design program families. An object-oriented software framework consists of design specifications and program code and builds upon standard object-oriented programming concepts learned as an undergraduate. A framework consists of a library of concrete and abstract classes (also interfaces in the Java sense) that forms the skeleton of systems within the family. Programmers develop applications of the framework by defining the needed concrete classes for their specific application.

The Table ADT family outlined in these notes uses the software framework approach. Paper [Cunningham 2010] discusses the software framework design in more detail.

Still yet, developing a non-trivial framework is not easy. Software developers must practice the framework design principles consistently over time to master the methods.

Acknowledgements

In Spring 2015, I adapted these lecture notes from the papers [Cunningham 2004] and [Cunningham 2010]. Former graduate students Yi Liu, Jingyi Wang, and Cuihua Zhang were co-authors on one or both of those papers. Former colleague Robert Cook and former graduate student "Jennifer" Jie Xue made suggestions for improvement of paper [Cunningham 2001], which is an earlier version of [Cunningham 2010]. Former graduate students Chuck Jenkins and Pallavi Tadepalli also made useful suggestions on [Cunningham 2010]. Jingyi Wang implemented a version of the Table framework for her MS project. Some of the ideas reflected in the Table framework arose from earlier MS projects on related frameworks by graduate students Wei Feng, Jian Hu, and Deep Sharma.

In Spring 2017, I adapted the notes to use Pandoc. Subsequently, in 2017, 2018 and 2019, I contined to update the format of the document to be more compatible with my evolving document structures.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to

HTML, PDF, and other forms as needed.

References

[**Ardis 2000**]: 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.

[**Britton 1981**]: 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. 195-204, March 1981.

[**Brooks 1986**]: F. P. Brooks Jr. "No Silver Bullet—Essence and Accidents in Software Engineering," Information Processing, Elsevier Science, pp. 1068-1076, 1986.

[**Brooks 1995**]: F. P. Brooks Jr. " 'No Silver Bullet' Refired," Chapter 17, In The Mythical Man Month, Anniversary Edition, 1995.

[**Coplien 1998**]: J. Coplien, D. Hoffman, and D. Weiss. "Commonality and Variability in Software Engineering," IEEE Software, Vol. 15, No. 6, pp. 37-45, 1998.

[**Cunningham 2001**]: 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.

[**Cunningham 2004**]: H. C. Cunningham, C. Zhang, and Y. Liu. "Keeping Secrets within a Family: Rediscovering Parnas," In Proceedings of the International Conference on Software Engineering Research and Practice (SERP), pp. 712-718, CSREA Press, June 2004.

[**Cunningham 2010**]: H. C. Cunningham, Y. Liu, and J. Wang. "Designing a Flexible Framework for a Table Abstraction," In Y. Chan, J. Talburt, and T. Talley, editors, Data Engineering: Mining, Information, and Intelligence, pp. 279-314, Springer, 2010.

[**Cunningham 2019f**]: H. Conrad Cunningham. *Data Abstraction*, 2019.

[**Johnson 1998**]: R. E. Johnson and B. Foote. "Designing Reusable Classes," Journal of Object-Oriented Programming, Vol. 1, No. 2, pp. 22-35, 1998.

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

[**Parnas 1972**]: 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.

[**Parnas 1976**]: 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.

[**Parnas 1979**]: 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.

[**Parnas 1985**]: 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.

[Parnas 2001a]: 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.

[Parnas 2001b]: 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.

[Waldo 2001]: 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.

[Weiss 1999]: D. M. Weiss and C. T. R. Lai. Software Product-Line Engineering: A Family-Based Software Development Process, Addison Wesley, 1999.

[Weiss 2001]: 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.

Terms and Concepts

Module, modular specification and design, independent development, changeability, comprehensibility, commonality and variability, information hiding, secret, interface, signature, syntax and semantics, abstract interface, two-phase specification, callback, design by contract, software or program family, software product line, software framework, table ADT, key, total and partial orderings.