Applying Software Patterns in the Design of a Table Framework

H. Conrad Cunningham
Department of Computer Science
University of Mississippi
237 Kinard Hall
University, MS 38677 USA
(662) 915-5358
cunningham@cs.olemiss.edu

Jingyi Wang
Data Products Division
Acxiom Corporation
1001 Technology Drive
Little Rock, AR 72223 USA
(501) 252-5781
jwang@acxiom.com

ABSTRACT
This paper describes how software design patterns are applied to advantage in the design of a small application framework for building implementations of the Table Abstract Data Type (ADT). The framework consists of a group of Java interfaces that collaborate to define the structure and high-level interactions among components of the Table implementations. The key feature of the design is the separation of the Table's key-based record access mechanisms from the physical storage mechanisms. The systematic application of the Layered Architecture, Interface, Bridge, and Proxy patterns lead to a design that is sufficiently flexible to support a wide range of client-defined records and keys, indexing structures, and storage media. The use of the Template Method, Strategy, and Decorator patterns also enables variant components to be easily plugged into the framework. The Evolving Frameworks patterns give guidance on how to modify the framework as more is learned about the family of applications. The conscious use of these software design patterns increases the understandability and consistency of the framework's design.

Keywords
Table ADT, application framework, software design pattern, layered architecture.

1. INTRODUCTION
A central idea in contempborary software design is the concept of a design pattern [Buschmann 1996]. When experts need to solve a problem, they seldom invent a totally new solution. More often they will recall a similar problem they have solved previously and reuse the essential aspects of the old solution to solve the new problem. They tend to think in problem-solution pairs. Identifying the essential aspects of specific problem-solution pairs leads to descriptions of general problem-solving patterns that can be collected and reused. A *design pattern* thus documents "a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution" [Buschmann 1996].

This paper shows how software design patterns can be applied in the design of a small application framework for building implementations of the Table Abstract Data Type (ADT). In general, a framework expresses a reusable design for a system as a collection of abstract classes and the way that their instances interact with one another. [Fayad 1999]. It represents a skeleton of a system that can be customized for a particular purpose. To customize the framework, a developer provides concrete implementations of the abstract classes.

The Table ADT represents a collection of records that can be accessed by the unique keys of the records. The design of the Table framework consists of a group of Java interfaces that work together in well-defined ways. The design encompasses a wide range of possible implementations of the Table ADT—simple array-based data structures in memory, B-tree file structures on disk, perhaps even structures distributed across a network. The key concept in the framework design is the separation of the key-based record access mechanisms from the physical storage mechanisms for the records.

The design takes advantage of several well-known software design patterns. The need to decouple the access mechanism from the storage mechanism suggests a hierarchical structure based on the Layered Architecture [Buschmann 1996][Shaw 1996] and Interface [Grand 1998] design patterns. Given the layered architecture, the Bridge and Proxy patterns [Gamma 1995] [Grand 1998] then suggest how to organize the interactions among the various layers. The Iterator pattern [Gamma 1995] [Grand 1998] is also helpful; it provides a systematic mechanism for accessing groups of records. The Template Method, Strategy, and Decorator patterns [Gamma 1995] [Grand 1998] provide standard structures for plugging variable components into the framework. Furthermore, as the framework evolves, it follows the general development path documented by the Evolving Frameworks system of patterns [Roberts 1998].

2. TABLE ADT
The 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 of a totally ordered set. The operations provided by the Table ADT allow a

record to be stored and retrieved using its key to identify it within the collection.

The Table framework has the following requirements:

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.
4. All interactions among its components should only be through well-defined interfaces that represent coherent abstractions.
5. Its design should use appropriate software design patterns to increase reliability, understandability, and consistency.

3. LAYERED ARCHITECTURE

The most significant aspect of this design problem is the separation of the table's high-level, key-based access mechanisms from the lower-level storage mechanisms for physical records. This mix of high- and low-level issues suggests a hierarchical architecture based on the *Layered Architecture* design pattern [Buschmann 1996] [Shaw 1996]. When there are several distinct groups of services that can be arranged hierarchically, this pattern assigns each group to a layer. Each layer can then be developed independently. A layer is implemented using the services of the layer below and, in turn, provides services to the layer above.

As shown in Figure 1, we choose three layers in this design. From the top to the bottom these include:

Client Layer. This layer consists of the client-level programs that use the table implementation in the layer below to store and retrieve records.

Access Layer. This layer provides client programs key-based access to the records in the table. It uses the layer below to store the records physically.

Storage Layer. This layer provides facilities to store and retrieve the records from the chosen physical storage medium.

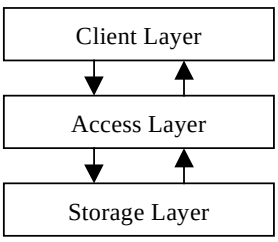

Figure 1. Layered Architecture

For example, suppose we want a simple indexed file structure with an in-memory index that uses an array-like relative file to store the records on disk [Folk 1998]. The implementation of the index would be part of the Access Layer; the implementation of the relative file would be in the Storage Layer. A program that uses the simple indexed file structure would be in the Client Layer.

The various layers need to be kept independent of one another. Thus, following the fundamental *Interface* design pattern [Grand 1998], we define each layer in terms of a set of related Java interfaces and require that interactions among the layers use these interfaces.

4. ACCESS LAYER

The Access Layer provides the Client programs key-based access to a collection of records. Its primary abstraction is the Table ADT. Before we can define the `Table` interface, however, we need to consider the characteristics of the records and their keys.

4.1 Keys and the Comparable Interface

As much as possible, we want to let clients (users) of the table implementations define their own record and key structures. However, any implementation of the Table ADT must be able to extract the keys from the records and compare them with each other. Thus we restrict the records to objects from which keys can be extracted and compared using some client-defined total ordering.

The built-in Java interface `Comparable` gives us what we need for the keys. Any class that implements this interface must provide the method:

- `int compareTo(Object key)` that 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.

Table implementations can use this method to compare keys. Clients can use any existing `Comparable` class for their keys or implement their own.

4.2 Records and the Keyed Interface

We introduce the Java interface `Keyed` to represent the type of objects that can be manipulated by a table. Any class that implements this interface must implement the method:

- `Comparable getKey()` that extracts the key from the associated record.

A table implementation can use this method to extract a key and then use the key's `compareTo` method to do the comparison.

4.3 Table Interface

Now, given the above types for keys and records, we introduce the Java interface `Table` to define the methods associated with the Table ADT. The interface must enable the client to access the table in the expected ways, e.g., to insert a new record or to delete a record with a given key.

We define the Table ADT as a Java interface that includes the following methods. A companion paper gives the formal semantics of these operations [Cunningham 2001].

- `void insert(Keyed rec)` inserts the `Keyed` object `rec` into the table.

- `void delete(Comparable key)` deletes the `Keyed` object with the given `key` from the table.
- `void update(Keyed rec)` updates the table by replacing the existing entry having the same key as argument `rec` with the argument object.
- `Keyed retrieve(Comparable key)` searches the table for the argument `key` and returns the `Keyed` object that contains this key.
- `boolean containsKey(Comparable key)` searches the table for the argument `key`.
- `boolean isEmpty()` checks whether the table is empty.
- `boolean isFull()` checks whether the table is full. (For unbounded tables, this method always returns false.)
- `int getSize()` returns the size of the table.

The Access Layer thus consists of the `Table` and `Keyed` interfaces and concrete classes that implement `Table`. Concrete classes that implement the `Comparable` and `Keyed` interfaces are part of the `Client` Layer. The interactions between the Client and Access Layer occurs as follows:

- The Client Layer calls the Access Layer using the `Table` interface.
- The Access Layer calls back to the Client classes that implement the `Keyed` and `Comparable` interfaces to do part of its work.

5. STORAGE LAYER

The Storage Layer provides facilities to store and retrieve records on a physical storage medium. To define the interfaces between the Access and Storage layers, we adopt a structure motivated by the Bridge and Proxy design patterns [Gamma 1995][Grand 1998] to achieve the desired degree of decoupling and collaboration. We also take into account both the expected characteristics of the storage media and the expected needs of the `Table` implementations.

5.1 Bridge Pattern

The *Bridge* pattern is useful when we wish to decouple the "interface" of an abstraction from its "implementation" so that the two can vary independently [Gamma 1995] [Grand 1998]. In this design (as shown in Figure 2), the "interface" is the `Table` abstraction in the Access Layer, which provides key-based access to a collection of records; the "implementation" is the `RecordStore` abstraction in the Storage Layer, which provides a physical storage mechanism for records. These two hierarchies of abstractions collaborate to provide the table functionality. At the time a table is created, any concrete `Table`-implementing class can be combined with any concrete `RecordStore`-implementing class.

We assume that a storage medium abstracted into the `RecordStore` ADT consists of a set of physical "slots". Each slot has a unique "address", the exact nature of which is dependent upon the medium. A program may allocate slots from this set and release allocated slots for reuse. There may, however, be restrictions upon the characteristics of the records acceptable to the storage medium. For example, if a random-access disk file is used, it may be necessary to restrict the record to data that can be written into a fixed-length block of bytes.

There are many possible implementations of `Table` in the Access Layer--such as simple indexes, balanced trees, and hash tables. Any `Table` implementation must be able to allocate a new slot, store a record into it, retrieve the record from it, and then deallocate the slot when it is no longer needed. The `Table` must be able to refer to slots in a medium-independent manner. Moreover, most implementations will need to treat these slot references as data that can be stored in records and written to a slot. For example, the nodes of a tree-structured table are "records" that may be stored in a `RecordStore`; these nodes must include "pointers" to other nodes, that is, references to other slots.

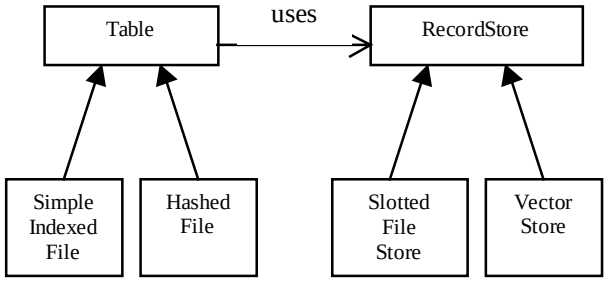

Figure 2. Bridge Pattern

5.2 Proxy Pattern

Since we do not wish to expose the internal details of the `RecordStore` to the Access Layer, we need a medium-independent means for addressing the records in the `RecordStore`. The approach we take is a variation of the *Proxy* pattern [Gamma 1995] [Grand 1998].

The idea of the Proxy pattern is to use a proxy object that acts as a surrogate for a target object. When a client wants to access the target object, it does so indirectly via the proxy object. Since the target object is not accessed directly by the client, the exact nature and location, even the existence, of the target object is not directly visible to the client. The proxy object serves as a "smart pointer" to the target object, allowing the target's location and access method to vary.

In this design, we define the `RecordSlot` abstraction to represent the proxies for the slots within a `RecordStore`. As shown in Figure 3, these two abstractions collaborate to enable the Access Layer to store and retrieve records in a uniform way, no matter which storage medium is used. Because of the need to write the slot references themselves into records as data, we also assign an integer "handle" to uniquely identify each physical slot in a `RecordStore`. Since multiple `RecordStore` instances may be in use at a time, each `RecordSlot` also needs a reference to the `RecordStore` instance to which it refers.

5.3 RecordStore Interface

Now we can specify the `RecordStore` and `RecordSlot` interfaces. We define the *`RecordStore`* ADT as a Java interface that includes the following methods:

- `RecordSlot getSlot()` allocates a new record slot and returns the `RecordSlot` object.
- `RecordSlot getSlot(int handle)` rebuilds a record slot using the given `handle` and returns the `RecordSlot`.
- `void releaseSlot(RecordSlot slot)` deallocates the allocated record `slot`.

In the `getSlot()` method the framework design allows lazy allocation of the handle and, hence, of the associated physical slot. That is, the handle may be allocated by `getSlot()` or later upon its first use to store a record in the `RecordStore`.

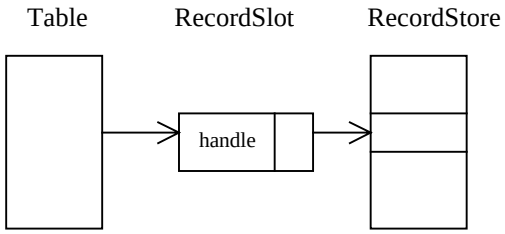

Figure 3. Proxy Pattern

5.4 RecordSlot Interface

The `RecordSlot` interface represents a proxy for the physical record "slots" within a `RecordStore`. The `RecordSlot` interface includes the following methods:

- `void setRecord(Object rec)` stores the argument object `rec` into this `RecordSlot`.

  The allocation of the handle can be done here or already done by the `getSlot()` method of `RecordStore`.
- `Object getRecord()` returns the record stored in this `RecordSlot`.
- `int getHandle()` returns the handle of this `RecordSlot`.
- `RecordStore getContainer()` returns a reference to the `RecordStore` with which this `RecordSlot` is associated.
- `boolean isEmpty()` determines whether the `RecordSlot` is empty (i.e., does not hold a record).

`Method getRecord() returns the NULLRECORD` object if no record has been stored in the slot. This denotes an inert, empty record implemented according to the *Null Object* design pattern [Grand 1998].

5.5 Record Interface

One issue we have not addressed is how the `RecordSlot` mechanism can store the records on and retrieve them from the physical slots on the storage medium. This is an issue because the records themselves are defined in the Client Layer and their internal details are, hence, hidden from the `RecordStore`. For in-memory implementations of `RecordStore` this is not a problem; the `RecordStore` 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.

The solution taken here is similar to what is done with the `Keyed` interface. We introduce a `Record` interface with three user-defined methods:

- `writeRecord(DataOutput)` that writes the record to a `DataOutput` stream,
- `readRecord(DataInput)` that reads the record from a `DataInput` stream,
- `getLength()` that returns the number of bytes that will be written by `writeRecord`.

The `Record` interface is defined in the Storage Layer. However, the concrete implementations of the interface appear in either the Client Layer for client-defined records or the Access Layer for "records" used internally within a `Table` implementation The `RecordStore` calls the `Record` methods when it needs to read or write the physical record. The code in the `Record`-implementing class does the conversion of the internal record data to and from a stream of bytes. The `RecordStore` implementation is responsible for routing the stream of bytes to and from the physical storage medium.

5.6 Discussion

In summary, the Storage Layer consists of the `RecordStore`, `RecordSlot`, and `Record` interfaces and concrete classes that implement `RecordStore` and `RecordSlot`. Figure 4 shows the use relationships among the Access and Storage layer abstractions.

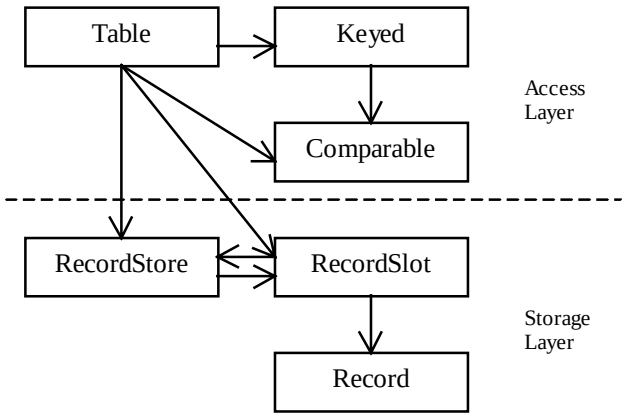

Figure 4. Abstraction Usage Relationships

The key goal in the framework design is the separation of the key-based access mechanisms, represented by the `Table` interface,

from the physical storage mechanisms for the records, represented by the `RecordStore` interface. This approach is inspired, in part, by Sridhar's YACL C++ library's approach to B-trees [Sridhar 1996], which separates the B-tree implementation from the NodeSpace that supports storage for the B-tree nodes. The design extends Sridhar's concept with the `RecordSlot` abstraction, which is inspired, in part, by Goodrich and Tamassia's Position ADT [Goodrich 1998]. The Position ADT abstracts the concept of "place" within a sequence so that the element at that place can be accessed uniformly regardless of the actual implementation of the sequence.

This paper's approach generalizes the NodeSpace and Position concepts and systematizes their design by using standard design patterns. The Layered Architecture and Bridge patterns motivate the design of the `RecordStore` abstraction and the Proxy pattern motivates the design of the `RecordSlot` mechanism. The result is a clean structure that can be described and understood in terms of standard patterns concepts and terminology. Careful attention to the semantics of the abstract methods in the various interfaces helps us allocate responsibility among the various abstractions in the framework and helps us decide what functionality can be supported across many possible implementations [Cunningham 2001].

6. ITERATORS

The *Iterator* design pattern documents a systematic way for client code to access the elements of a collection sequentially without exposing the internal details of the collection's implementation [Gamma 1995] [Grand 1998]. The interface `Iterator`, defined in the Java API, provides a standard means for Java programs to support iterators. It includes method `hasNext()` to check for the existence of another element and `next()` to return the next element. We can add several useful iterators and iterator-manipulating methods to the framework design.

6.1 Table Iterator Methods

As a convenience for clients of the table implementations, we add two iterator methods to the `Table` interface:

- `Iterator getKeys()` returns an iterator that enables the client to access all the keys in the table one by one.
- `Iterator getRecords()` returns an iterator that enables the client to access all the records in the table one by one.

Similarly, we can add overloaded versions of the `insert` and `delete` methods that take appropriate iterators as arguments.

- `void insert(Iterator iter)` inserts the `Keyed` objects denoted by the iterator `iter` into the table.
- `void delete(Iterator iter)` deletes the objects from the table whose keys match those returned by iterator `iter`.

6.2 Query Iterator Methods

The `Table` abstraction defined in a previous section only provides access based on the unique, primary key of the record. Sometimes a client may want to access records based on the values of other fields. Unlike the primary key, these secondary key fields may not uniquely identify the record within the collection.

The framework can be readily extended to accommodate access on secondary keys as well as the primary key. We can, for example, define a `MultiKeyed` interface in the Access Layer that extends the `Keyed` interface with additional methods:

- `int getNumOfKeys()` that returns the number keys supported by the associated record implementation,
- `Comparable getKey(int k)` that extracts a secondary key `k` from the record. Key 0 is the primary key.

While it is sufficient for the basic `Table` mechanism to have a simple method `retrieve(Comparable)`, a table that supports access on multiple keys needs to allow a variable number of items to be retrieved for each secondary key value. As a convenience, it is also useful to allow a query to be done with a combination of various primary and secondary key values. We can define a `QueryTable` interface that extends `Table` and adds two new iterator methods:

- `Iterator selectKeys(query)` evaluates the `query` and returns the sequence of keys of all records that satisfy the `query`.
- `Iterator selectRecords(query)` evaluates the `query` and returns the sequence of all records that satisfy the `query`.

6.3 Input Iterators

The method `insert(Iterator)` is a convenient mechanism for loading a table with a sequence of items that come from a different format. We add the abstract base class `InputIterator` to enable users to conveniently create a class to read records from from external files. The design of this class takes advantage of the Template Method design pattern.

The *Template Method* pattern [Gamma 1995] [Grand 1998] is a quite useful pattern for building frameworks. Central to this pattern is an abstract class that provides a skeleton of the needed behaviors. The class consists of two kinds of methods:

Template methods. These are concrete classes that implement the shared functionality of the class hierarchy. They are not intended to be overridden by subclasses

Hook methods. These are abstract methods that provide "hooks" for attaching the functionality that varies among applications. Although hook methods may have a default definition in the abstract class, in general they are intended to be overridden by subclasses. A template method calls a hook method to carry out application-dependent operations.

The `InputIterator` class implements the Java `Iterator` interface, providing the required `Iterator` methods as template methods. It also include two abstract hook methods that are called by the template methods:

- `boolean atEnd()` that returns `true` when the end of the input has been reached.

- `Object readNext()` that returns the next object in the input stream.

A client who wishes to use this class must extend the `InputIterator` class, providing appropriate concrete definitions for the abstract methods.

6.4 Filtering Iterators

Sometimes users need to transform the elements of one sequence into another. Some elements may need to be deleted and others kept. Sometimes a conversion operation needs to be applied to every element of a sequence. We can support these operations on iterators by introducing the `FilterIterator` class.

The `FilterIterator` class is a concrete class that implements the `Iterator` interface. It's constructor takes three arguments: an iterator, a selector, and a converter. Its implementation takes advantage of the Decorator and Strategy design patterns.

The *Decorator* pattern extends the functionality of an object in a way that is transparent to the users of that object [Gamma 1995][Grand 1998]. A Decorator object is of the same type as the original object. It serves as a wrapper around the original object that provides enhanced functionality but delegates part of its work to the original object. The `FilterIterator` is an iterator whose constructor takes another iterator as an argument; it uses the argument iterator as its source of data but selects and transforms the data that is returned by its `next()` method. The use of the Decorator pattern thus allows a `FilterIterator` to provide enhanced functionality at any place that an `Iterator` is used.

The *Strategy* pattern abstracts a family of related algorithms behind an interface [Gamma 1995][Grand 1998]. The desired algorithm can be selected at runtime and plugged into the object that uses the algorithm. The selector and converter arguments of the `FilterIterator` are Strategy objects that encapsulate the selection and conversion algorithms, respectively. For example, the selector is an object of a class that implements the `Selector` interface. This interface requires that the class implement the method:

- `boolean selects(Object obj)` that returns `true` if and only if `obj` satisfies the chosen criteria.

The `FilterIterator` delegates the choice of which objects from its input sequence to keep to the `selects()` method of the selector object. The use of the Strategy pattern enables the same `FilterIterator` object to be configured flexibly to have different behaviors as needed.

7. EVOLVING FRAMEWORKS

Framework designs tend to evolve as their usage grows and the developers learn more about the application domain. This evolution often follows the steps documented in the *Evolving Frameworks* system of patterns [Roberts 1998]. The Table Framework is being developed according to several of these patterns.

7.1 Generalizing from Three Examples

In most nontrivial frameworks, it is not possible to come up with the right abstractions just by thinking about the problem. Domain experts typically do not know how to express the abstractions in their heads in ways that can be turned into designs for abstract classes; programmers typically do not have a sufficient understanding of the domain to derive the proper abstractions immediately [Roberts 1998].

Typically, three implementation cycles are needed to develop a sufficient understanding of the application to construct good abstractions [Roberts 1998]. Design of the Table framework was no different despite the simplicity of the problem. In the exploration of the design, we constructed three prototype implementations of the `RecordStore` and two implementations of the `Table` [Wang 2000]. Earlier work designing similar Table libraries also yielded insight. Each implementation effort gave new insights into what an appropriate set of abstractions were and uncovered potential problems.

7.2 Whitebox and Blackbox Frameworks

As this framework is defined so far, it is a pure *whitebox framework* [Fayad 1999]. In general, a whitebox framework consists of a set of interrelated abstract base classes. Developers implement new applications by extending these base classes and overriding methods to achieve the desired new functionality. The implementers must understand the intended functionality and interactions of the various classes and methods. Such frameworks are flexible, extensible and easy to build, but they are difficult to learn and use.

While whitebox frameworks rely upon inheritance to achieve extensibility, *blackbox frameworks* use object composition to support extensible systems [Fayad 1999]. Such frameworks define interfaces for components and allow existing components to be plugged into these interfaces. Appropriate components that conform to these interfaces are collected in a component library for ready reuse. Such frameworks can be easy to use and extend. However, they tend to be difficult to develop because they require the developers to provide appropriate interfaces for a wide range of potential uses.

7.3 Component Library

Once a basic whitebox framework is in place, the design usually evolves toward a blackbox framework by the addition of useful concrete classes to a *component library* [Roberts 1997]. The addition of concrete implementations of the `Table` and `RecordStore` abstractions thus is a natural next step in the evolution of the Table framework.

A prototype component library has been developed for an earlier version of Table framework design [Wang 2000]. This component library provides three different implementations of the Storage Layer, in particular of the `RecordStore` interface:

- `VectorStore`, an implementation that stores the records in a Java `Vector`;
- `LinkedMemoryStore`, an implementation that stores the records in a linked list;
- `SlottedFileStore`, an implementation that stores the records in a relative file of fixed length blocks on disk and uses a bit-map to manage the blocks.

The component library also provides two implementations of the

Access Layer, in particular of the `Table` interface:

- `SimpleIndexedFile`, an implementation that uses a simple sorted index in memory to support the location of records using keys [Folk 1998];
- `HashedFileClass`, an implementation that uses a hash table to support the key-based access.

In the prototype component library, the component `SimpleIndexedFile` actually implements the `QueryTable` interface.

7.4 Hot Spots

Experience in developing applications with a framework helps identify shared functionality and points of variability. The shared functionality, often called *frozen spots* [Pree 1995][Schmid 1999], can be incorporated into the framework as concrete classes or as concrete methods of abstract classes. The Template Method pattern is one common technique for implementing the frozen spots. The points of variability, often called *hot spots*, can be incorporated into the framework as abstract hook methods that are refined via inheritance. Alternatively, hot spots can be implemented by delegation to classes that encapsulate the required functionality (e.g., using the Strategy and Decorator patterns). This evolutionary step has not occurred in the simple Table framework, but should occur as additional implementations and analyses are done

Schmid has proposed an approach to framework design that promises to systematize the process of constructing frameworks [Schmid 1999]. In his Systematic Generalization methodology, framework designers take a specific application within the framework's domain and then convert it into a framework by a sequence of generalizing transformations. Each transformation introduces a hot spot into the structure. The methodology then proposes techniques for analyzing the hot spot and constructing an appropriate hot-spot subsystem to plug into that hot spot. Future work on the Table framework should use this approach to seek further generalizations.

8. CONCLUSION

This paper describes how software design patterns are applied advantageously in the design of a small application framework for building implementations of the Table ADT. The framework consists of a group of Java interfaces that collaborate to define the structure and high-level interactions among components of the Table implementations. The key feature of the design is the separation of the Table's key-based record access mechanisms from the physical storage mechanisms. The systematic application of the Layered Architecture, Interface, Bridge, and Proxy patterns lead to a design that is sufficiently flexible to support a wide range of client-defined records and keys, indexing structures, and storage media. The use of the Template Method, Strategy, and Decorator patterns also enables variant components to be easily plugged into the framework. The Evolving Frameworks patterns give guidance on how to modify the framework as more is learned about the family of applications. The conscious use of these software design patterns increases the understandability and consistency of the framework's design.

9. ACKNOWLEDGEMENTS

The authors thank Robert Cook and "Jennifer" Jie Xu for their suggestions concerning this work. This work also benefited from insights provided by projects completed by the first author's former students Wei Feng on relative files, Jian Hu on Table libraries, and Deep Sharma on B-tree libraries.

10. REFERENCES

1. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture, A System of Patterns*, Wiley, 1996.
2. Cunningham, H. C., and Wang, J. Building a layered framework for the table abstraction, In *Proceedings of the ACM Symposium on Applied Computing*, March 2001.
3. Fayad, M. E., Schmidt, D. C., and Johnson, R. E. Application frameworks, In. Fayad, M. E., Schmidt, D. C., and Johnson, R. E., editors, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, Wiley, *1999.*
4. Folk, M. J., Zoellick, B., and Riccardi, G. *File Structures: An Object-Oriented Approach with C++*, Addison Wesley, 1998.
5. Gamma, R., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
6. Goodrich, M. T. and Tomassia, R. *Data Structures and Algorithms in Java*, Wiley, 1998.
7. Grand, M. *Patterns in Java, Volume 1*, Wiley, 1998.
8. Pree, W. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.
9. Roberts, D. and Johnson, R. Patterns for evolving frameworks, In Martin, R., Riehle, D., and Buschmann, F., editors, *Pattern Languages of Program Design 3*, Addison-Wesley, 1998.
10. Schmid, H. A. Framework design by systematic generalization, In Fayad, M. E., Schmidt, D. C., and Johnson, R. E., editors, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, Wiley, 1999.
11. Shaw, M. Some patterns for software architecture, In Vlissides, J. M., Coplien, J. O., and Kerth, N. L., editors, *Pattern Languages of Program Design 2*, Addison Wesley, 1996.
12. Sridhar, M. A. *Building Portable C++ Applications with YACL*, Addison-Wesley, 1996.
13. Wang, J. *A Flexible Java Library for Table Data and File Structures*, Technical Report UMCIS-2000-07, Department of Computer and Information Science, University of Mississippi, May 2000.

11. BIOGRAPHIES

H. Conrad Cunningham is Associate Professor and Interim Chair of the Department of Computer and Information Science at the

University of Mississippi. His professional interests include concurrent and distributed computing, programming methodology, and software architecture. He has a BS degree in mathematics from Arkansas State University and MS and DSc degrees in computer science from Washington University in St. Louis, Missouri.

Jingyi Wang is a Software Developer at Acxiom Corporation in Little Rock, Arkansas. Her professional interests include the design of enterprise computing applications. She has a BA degree in economics from Fudan University in Shanghai, China, and an MA in economics and an MS in computer science from the University of Mississippi.