Building a Layered Framework for the Table Abstraction

H. Conrad Cunningham
Department of Computer Science
University of Mississippi
302 Weir 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 the design of a small application framework for building implementations of the Table Abstract Data Type (ADT). The framework is built upon a group of Java interfaces that collaborate to define the structure and high-level interactions among components of the Table implementations. The key concept in the design is the separation of the Table's key-based record access mechanisms from the physical storage mechanisms. The design is sufficiently flexible to support a wide range of client-defined records and keys, indexing structures, and storage media. The design also takes advantage of several well-known software design patterns and of formal design contracts to increase reliability, understandability, and consistency.

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

1. INTRODUCTION
The Table Abstract Data Type (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.

Many different concrete data and file structures can be used to implement the Table ADT. Simple sorted arrays may suffice for small in-memory tables. Larger in-memory tables can be implemented using data structures such as hash tables or balanced binary trees to provide fast access. For collections too large to fit in memory, the implementation can maintain a key-based index structure in memory but keep the full collection of records on disk. Even larger collections might use balanced, multiway tree structures such as the B-tree to organize the records on disk.

This paper describes the design of a small application framework for building implementations of the Table ADT. 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 [2]. 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 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—from data structures in memory to file structures distributed across a network. The key concept in the 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. According to Buschmann, a design pattern "describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution" [1]. For example, the need to decouple the access mechanism from the storage mechanism suggests a hierarchical structure based on the Layered Architecture pattern [1][12]. Given the layered architecture, the Bridge and Proxy patterns [4][6] then suggest how to organize the interactions among the various layers.

2. REQUIREMENTS
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 design contracts and appropriate design patterns to increase reliability, understandability, and consistency.

3. TABLE ABSTRACTION

The first issues that we need to address concern the characteristics of the records and keys. 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 a `compareTo(Object)` method. Table implementations can use this method to compare keys. Clients can use any existing `Comparable` class for their keys or implement their own.

We introduce the Java interface `Keyed` to represent the type of objects that can be stored in a table. Any class that implements this interface must implement a `getKey()` method that extracts the key from the record. A table implementation can use this method to extract a key and then use the key's `compareTo` method to do the comparison.

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. In specifying the methods, we give both their signatures (i.e., parameters and return types) and semantics (i.e., behaviors). A useful approach for expressing semantics is to give design contracts [8] using preconditions and postconditions for each method and invariants for the ADT as a whole.

Because a `Table` implementation must work with many different types of records, keys, and storage media, the internal details of these must be hidden. In specifying the operations, we can express key features of records, keys, and storage media as abstract predicates. The precise definition of these predicates depends upon the particular implementations used. Three predicates prove to be useful:

- `isValidKey(Comparable)` is true if and only if the argument is an element of the set of meaningful keys supported by the client's key class, that is, by a class that implements the `Comparable` interface.

- `isValidRec(Keyed)` is true if and only if the argument is an element of the set of meaningful records supported by the client's keyed record class, that is, by a class that implements the `Keyed` interface.

- `isStorable(Keyed)` is true if and only if the argument can be stored on the storage medium being used with the implementation of the table.

To state the semantics, we need to introduce an appropriate model and notation. We model the collection of records by the variable `table`, which is a partial function from the set of valid keys to the set of valid and storable records. For convenience, we use `table` as either a function or a set. In postconditions, the variable `retVal` refers to the value returned by the method call and the prefix `#` attached to a variable denotes the value at the time the method was called. Unless a new value is explicitly assigned to a variable in the postcondition, its value must not be changed by the method call.

Now, we can define the Table ADT as a Java interface that includes the following ADT invariant and public methods:

Inv: `(∀k, r: r = table(k) : isValidKey(k) && isValidRec(r) && isStorable(r) && k = r.getKey()).`

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

  Pre: `isValidRec(r) && isStorable(r) && !containsKey(r.getKey()) && !isFull().`
  Post: `table = #table ∪ {(r.getKey(),r)}.`

- `void delete(Comparable key)` deletes the `Keyed` object with the given `key` from the table.

  Pre: `isValidKey(key) && containsKey(key).`
  Post: `table = #table -{(key,#table(key))}.`

- `void update(Keyed r)` updates the table by replacing the existing entry having the same key as argument `r` with the argument object.

  Pre: `isValidRec(r) && isStorable(r) && containsKey(r.getKey()).`
  Post: `table = ( #table - {(r.getKey(),#table(r.getKey()))} ) ∪ {(r.getKey(),r)}.`

- `Keyed retrieve(Comparable key)` searches the table for the argument `key` and returns the `Keyed` object that contains this key.

  Pre: `isValidKey(key) && containsKey(key).`
  Post: `retVal = #table(r.getKey()).`

- `boolean containsKey(Comparable key)` searches the table for the argument `key`.

  Pre: `isValidKey(key).`
  Post: `retVal = defined(#table(key)).`

- `boolean isEmpty()` checks whether the table is empty.

  Pre: `true.`
  Post: `retVal = (#table = ∅).`

- `boolean isFull()` checks whether the table is full. (For unbounded tables, this method returns false.)

  Pre : `true`
  Post: `retVal =` (`#table` has no free space to store a new record).

- `int getSize()` returns the size of the table.

  Pre: `true.`
  Post: `retVal = cardinality(#table).`

4. 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 [1] [12]. 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.

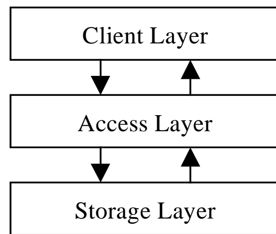

Figure 1. Layered Architecture

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 [3]. 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.

4.1 Access Layer

The Access Layer consists of the `Table` and `Keyed` interfaces (as described in section 3) and concrete classes that implement `Table`. 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.

4.2 Storage Layer

To define the interfaces between the Access and Storage layers, we adopt a structure motivated by the Bridge and Proxy design patterns 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.

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 [4][6]. In this design, 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. As shown in Figure 2, a `Table`–implementing class *uses* a `RecordStore`–implementing class in providing the table functionality. At the time a table is created, any concrete `Table`-implementing class can be combined with any concrete `RecordStore`-implementing class.

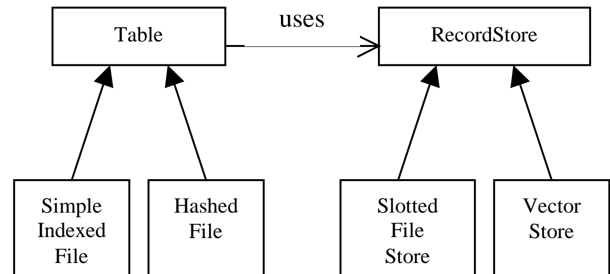

Figure 2. Bridge Pattern

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 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 slots. 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.

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 [4][6].

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.

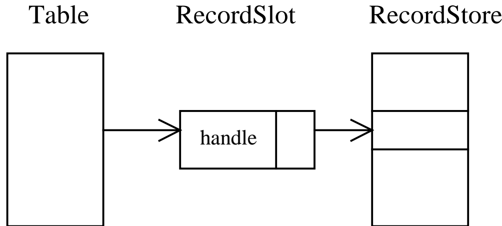

Figure 3. Proxy Pattern

4.3 RecordStore Interface

Now we can specify the `RecordStore` and `RecordSlot` interfaces. The model for the semantics of these ADTs includes two sets. The set `alloc` denotes the set of slot handles that have been assigned to `RecordSlot` instances. The set `store` is a partial function from the set of valid handles to the set of storable objects. For convenience, the set `unalloc` is used to denote the set of valid but unallocated handles, that is, the complement of the set `alloc`. The constant `NULLHANDLE` represents a special integer code that cannot be assigned as a valid slot handle; it is neither in `alloc` nor `unalloc`. Here we assume that `RecordStore` is unbounded in size.

We define the `RecordStore` ADT as a Java interface that includes the following methods and ADT invariant:

Inv. (∀ `h:: h ∈ alloc ≡ defined(store(h))`).

- `RecordSlot getSlot()` allocates a new record slot and returns the `RecordSlot` object.

  Pre: `true.`
  Post: `retVal.getContainer() =`
  `this_RecordStore &&`
  `retVal.getRecord() = NULLRECORD &&`
  `retVal.getHandle() ∉ #alloc &&`
  `retVal.getHandle() ∈ alloc ∪`
  `{NULLHANDLE}.`

  Note that the above allows lazy allocation of the handle and, hence, of the associated physical slot. That is, the handle may be allocated here or later upon its first use to store a record in the `RecordStore`. We choose the value `NULLRECORD` to denote an empty record implemented according to the Null Object design pattern [6].
- `RecordSlot getSlot(int handle)` reconstructs a record slot using the given `handle` and returns the `RecordSlot`.

  Pre: `handle ∈ alloc.`
  Post: `retVal.getContainer() =`
  `this_RecordStore &&`
  `retVal.getRecord() = #store(handle)&&`
  `retVal.getHandle() = handle.`
- `void releaseSlot(RecordSlot slot)` deallocates the allocated record `slot`.

  Pre: `slot.getHandle() ∈ alloc.`
  Post: `alloc = #alloc - {slot.getHandle()}`
  `&& store = #store –`
  `{(slot.getHandle(),slot.getRecord())}.`

4.4 RecordSlot Interface

The `RecordSlot` interface represents a proxy for the physical record "slots" within a `RecordStore`. The semantics of its operations are, hence, stated in terms of the effects upon the associated `RecordStore` instance. The `RecordSlot` interface includes the following methods and ADT invariant:

Inv. `getHandle() ∈ alloc ∪ {NULLHANDLE}.`

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

  Pre: `isStorable(rec).`
  Post: Let `h = getHandle() && g ∈ #unalloc:`
  `(h ∈ #alloc ⇒ store = (#store -`
  `{(h,#store(h))}) ∪ {(h,rec)})`
  `&&`
  `(h = NULLHANDLE ⇒ alloc = #alloc ∪ {g}`
  `&& store = #store ∪ {(g,rec)}).`

  Note that this allows the allocation of the handle to be done here or already done by the `getSlot()` method of `RecordStore`.

- `Object getRecord()` returns the record stored in this `RecordSlot`.

  Pre: `true.`
  Post: Let `h = getHandle():`
  `(h ∈ #alloc ⇒ retVal = #store(h)) &&`
  `(h = NULLHANDLE ⇒ retVal = NULLRECORD).`
- `int getHandle()` returns the handle of this `RecordSlot`.

  Pre: `true.`
  Post: `retVal =` handle associated with this slot.
- `RecordStore getContainer()` returns a reference to the `RecordStore` with which this `RecordSlot` is associated.

  Pre: `true.`
  Post: `retVal =`
  `RecordStore` associated with this slot.
- `boolean isEmpty()` determines whether the `RecordSlot` is empty (i.e., does not hold a record).

Pre: `true`.
Post: `retVal =`
`(getHandle() = NULLHANDLE ||`
record associated with slot is `NULLRECORD)`.

4.5 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 records (or perhaps copy references to them). However, disk-based implementations must write the records to a (random-access) file and reconstruct the records when they are 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

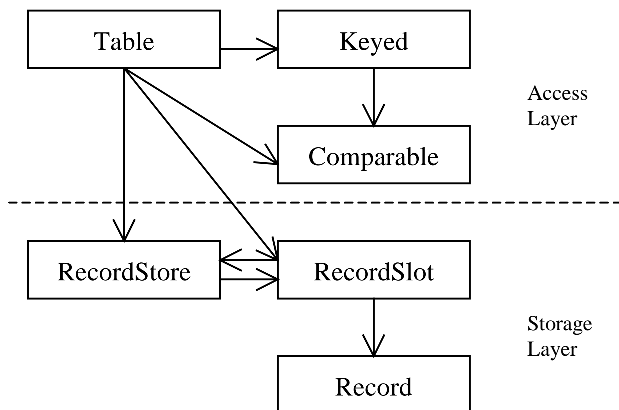

Figure 4. Abstraction Usage Relationships

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.

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 intended *usage* relationships among the Access and Storage layer abstractions. The abstraction at the base of the arrow uses the abstraction at the head.

5. FRAMEWORK ENHANCEMENTS

The framework presented is relatively minimal. It is useful to augment the framework in various ways.

5.1 Adding Iterators

Iterators, which enable the client code to examine the table's entries sequentially (as documented in the Iterator pattern [4][6]), are convenient features for users. For example, we might add two methods to the `Table` interface that return objects that implement the built-in Java interface `Iterator`. The contracts for these new accessor methods are show below.

Here we introduce some new notation. Function `seqFromIter` takes an iterator and returns the sequence of all objects returned by successive calls of the iterator's `next()` method. We also overload the ∈ and ∉ operators to work with sequences as well as sets. The function `occurs(e,s)` returns the number of occurrences of element `e` in sequence `s`.

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

  Pre: `true`.
  Post: `(∀ k,r: (k,r) ∈ #table :`
  `occurs(k,seqFromIter(retVal)) = 1)`
  `&& (∀ k: k ∈ seqFromIter(retVal):`
  `defined(#table(k))`.

- `Iterator getRecords()` returns an iterator that enables the client to access all the records in the table.
  Pre: `true`.
  Post: `(∀ k, r : (k,r) ∈ #table :`
  `occurs(r,seqFromIter(retVal)) = 1)`
  `&& (∀ r: r ∈ seqFromIter(retVal):`
  `#table(r.getKey()) = r)`.

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.

  Pre: `(∀ r : r ∈ seqFromIter(iter) :`
  `occurs(r,seqFromIter(iter)) = 1 &&`
  `isValidRec(r) && isStorable(r) &&`
  `!containsKey(r.getKey()))`.
  Post: `table = #table ∪ {(r.getKey(),r) :`
  `r ∈ seqFromIter(iter)}`.

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

  Pre: `(∀ k : k ∈ seqFromIter(iter) :`
  `occurs(k,seqFromIter(iter)) = 1 &&`
  `isValidKey(k) && containsKey(k))`.
  Post: `table = #table – {(k,#table(k)) :`
  `k ∈ seqFromIter(iter)}`.

5.2 Extending the Table Abstraction

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 easily 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 method `getNumOfKeys()` to enable the framework to determine how many keys are supported and method `getKey(int)` to extract a secondary 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 methods similar to the iterator methods discussed above:

- `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`.

5.3 Adding a Component Library

As this framework is defined so far, it is a pure whitebox framework [2]. That is, implementations are built by providing classes that implement (i.e., inherit from) the various Java interfaces. The implementors must understand the intended functionality and interactions of the various classes and methods.

In general, a problem with whitebox frameworks is that every application requires new concrete classes to be implemented for each of the abstract classes (and interfaces) [7]. A more useful way to organize a framework is to add a component library containing prebuilt classes that implement the needed application-specific behaviors [2][10]. Users of the framework can then plug together the desired components to form new systems; they only need to understand the functionality of the components at their external interfaces, not any of the implementation details.

A prototype component library has been developed for an earlier version of this framework design [14]. 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 [3];
- `HashedFile`, 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.

6. DISCUSSION

The key concept in the 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 [13], 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 [5]. 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 and formal design contracts. 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 (that is, the design contracts) helps us allocate responsibility among the various abstractions in the framework and helps us decide what functionality can be supported across many possible implementations. The formal design contracts for the methods also give a precise description of the expected behaviors of the concrete implementations of the framework's abstractions.

In most nontrivial frameworks, it is not possible to come up with the right abstractions just by thinking about the problem. Typically, three implementation cycles are needed to develop a sufficient understanding of the application to construct good abstractions [10]. 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` [14]. 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.

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 [10]. The addition of concrete implementations (based on the prototypes) of the `Table`

and RecordStore abstractions thus is a natural next step in the evolution of the Table framework.

The evolution toward a blackbox framework also typically involves moving from large-grained to finer-grained abstractions [10]. Experience in developing applications with a framework helps identify shared functionality and points of variability. The shared functionality, often called *frozen spots* [9][11], can be incorporated into the framework as concrete classes or as concrete methods of abstract classes. The Template Method pattern [4] [6] 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 [4][6]). This evolutionary step has not occurred in the simple Table framework, but should occur as additional implementations and analyses are done.

7. CONCLUSION

This paper describes the design of a small application framework for building implementations of the Table ADT. The framework is built upon a group of Java interfaces that collaborate to define the structure and high-level interactions among components of the Table implementations. The key concept is the separation of the Table's key-based record access mechanisms from the physical storage mechanisms. The design is sufficiently flexible to support a wide range of client-defined records and keys, indexing structures, and storage media. The design also takes advantage of several well-known software design patterns and of formal design contracts to increase reliability, understandability, and consistency.

8. ACKNOWLEDGMENTS

The authors thank Robert Cook and "Jennifer" Jie Xu for reading this paper carefully and making several useful suggestions for improvements. 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.

9. REFERENCES

[1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture, A System of Patterns*, Wiley, 1996.

[2] 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.*

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

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

[5] Goodrich, M. T. and Tomassia, R. *Data Structures and Algorithms in Java*, Wiley, 1998.

[6] Grand, M. *Patterns in Java, Volume 1*, Wiley, 1998.

[7] Johnson, R. E. and Foote, B. Designing reusable classes, *Journal of Object-Oriented Programming*, 1988.

[8] Meyer, B. *Object-Oriented Software Construction,* Second Edition, Prentice Hall PTR, 1997.

[9] Pree, W. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[10] 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.

[11] 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.

[12] 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.

[13] Sridhar, M. A. *Building Portable C++ Applications with YACL*, Addison-Wesley, 1996.

[14] 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.

10. BIOGRAPHIES

H. Conrad Cunningham is an Associate Professor of Computer 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.