12 Designing a Flexible Framework for a Table Abstraction

H. Conrad Cunningham [1], Yi Liu [2], Jingyi Wang [3]

[1] Department of Computer and Information Science, University of Mississippi, University, MS 38677 USA

[2] Department of Electrical Engineering and Computer Science, South Dakota State University, Brookings, SD 57007 USA

[3] Acxiom Corporation, 1001 Technology Drive, Little Rock, AR 72223 USA

ABSTRACT

This chapter examines how commonality/variability analysis, software design patterns, and formal design contracts can be used effectively to design a software framework for a Table abstraction. 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 commonality/variability analysis and the Layered Architecture, Interface, Bridge, and Proxy design 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. Formal design contracts enable the expected behaviors to be expressed precisely. The use of the Template Method, Strategy, Decorator, and Composite patterns also enables variant compo-

nents 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.

12.1 INTRODUCTION

In a provocative essay from the mid-1980s, 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.

The concept of software family (Parnas 1976) is one of the responses to the need for software reuse. Parnas (Parnas 1976) defines a *software family* as “a set of programs with so many common properties that it is worthwhile to study the set as a group”. Thus, by developers analyzing and exploiting the “common aspects and predicted variabilities” (Weiss and Lai 1999) among the members of a software family, the resulting software system can be constructed to reuse code for the common parts and to enable convenient adaptation of the variable parts (Cunningham et al. 2006a). Some writers use the terms *frozen spot* to denote a common aspect of the family and *hot spot* to denote a variable aspect of the family (Pree 1995; Schmid 1996).

A *software framework* (Johnson and Foote 1988) is a form of software family. A framework is “a generic application that allows different applications to be created from a family of applications” (Schmid 1999). In general, a framework represents the skeleton of a system that can be customized for a particular purpose. The frozen spots embody the overall structure of the framework (that is, the overall design) and are reused by the entire family of applications. In the context of an object-oriented language, frozen spots are expressed as a set of abstract and concrete classes that collaborate to embody the solutions to problems in the application domain. The hot spots are represented by the abstract classes, which can be extended to provide customized implementations of the variable aspects of a family. A specific set of implementations of the hot spots yields a member of the software family.

A framework is a system that is designed with generality and reuse in mind. *Software design patterns* (Gamma et al. 1995; Buschmann et al. 1996), which are well-established solutions to program design problems that commonly occur in practice, are intellectual tools for achieving the desired level of generality and reuse (Cunningham et al. 2006a). They are the building blocks for reusing designs. Building a software framework for a family is more costly than building a single application, but a well-designed framework can yield considerable benefit if many members of the family eventually need to be constructed.

In software design it is always important to specify precisely what a software artifact is to do. This is especially important in software frameworks, where the implementations of the hot spots vary from one application to another and are not usually developed at the same time nor by the same team as the framework itself. Framework designers must specify interfaces that do not change regardless of which implementation is “plugged in” to a hot spot. The specification should guide the users of the framework to provide appropriate implementations of the hot spots. Parnas and his colleagues (Parnas 1978; Britton et al. 1981) call this an *abstract interface* because it gives the assumptions that are common to all implementations. Meyer’s Design by Contract (Meyer 1992, 1997; Mitchell and McKim 2002) method provides an effective formal technique for specifying the expected behaviors of abstract interfaces.

This chapter shows how commonality and variability analysis, software design patterns, and Meyer-like formal design contracts can be applied in the design of a small Java software framework for building implementations of the Table Abstract Data Type (ADT). A previous paper (Cunningham and Wang 2001) presents an earlier version of the framework design developed in a careful, but ad hoc manner. This chapter expands on that work by revisiting the design from the perspective of commonality and variability analysis, improving the formal specifications, specifying additional framework features, and examining how the framework can evolve.

The Table ADT represents a collection of records that can be accessed by the unique keys of the records. The framework design should encompass 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. By approaching this as a family, the goal is to be able to assemble a Table implementation by selecting the combination of record access structures and storage structures to meet a specific application need.

The design process first analyzes the Table ADT as a family and then takes advantage of several well-known software design patterns to structure the framework. The commonality/variability analysis (in particular, the desire to decouple the record access mechanism from the storage mechanism) suggests a hierarchical structure based on the Layered Architecture (Buschmann et al. 1996; Shaw 1996) and Interface (Grand 1998) design patterns. Given the layered architecture, the Bridge and Proxy patterns (Gamma et al. 1995; Grand 1998) then suggest how to organize the interactions among the various layers. The Iterator pattern (Gamma et al. 1995; Grand 1998) is also helpful; it provides a systematic mechanism for accessing groups of records. The Template Method, Strategy, Decorator, and Composite patterns (Gamma et al. 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 and Johnson 1998).

The rest of the chapter is organized as follows. Section 2 briefly describes the requirements of the Table ADT and applies commonality and variability analysis to recognize the frozen spots and hot spots of the Table ADT framework. Section 3 briefly introduces the technique of using formal design contracts, which is applied in the specification of the interface design in the sections that follow. Section 4 applies Layered Architecture design pattern to build the top-level framework architecture. Sections 5, 6 and 7 apply several patterns to the design of interfaces among the different layers. Section 8 describes a utility module needed by the lower levels of the architecture. Section 9 applies the Iterator pattern to enhance the framework design. Section 10 illustrates the patterns of evolving frameworks that can be adopted into the Table framework design. Section 11 discusses the related work and Section 12 gives a conclusion.

12.2 ANALYSIS OF THE 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.

In (Cunningham and Wang 2001), Cunningham and Wang consider the design of the Table framework to have 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.

In building a framework, it is important to separate the concerns. The designers must separate the frozen spots, the aspects common to the entire family members, from the hot spots, the aspects specific to one family member. Furthermore, they must separate the various common and variable aspects from each other and consider them somewhat independently (Cunningham et al. 2006a). Commonality and variability analysis (Coplien et al. 1998; Weiss and Lai 1999) is a means of identifying the frozen spots and hot spots. The analysis produces commonalities, a list of assumptions that are true to all the members of the family, and variabilities, a list of assumptions that are true for only some members of the family. Thus, frozen spots and hot spots are chosen on the basis of commonalities and variabilities, respectively. In this chapter, the commonalities and variabilities of the Table ADT are examined based on the requirements of the Table ADT and the prototype implementations (Wang 2000).

The requirements stated above mix concerns in the framework design—commonalities, variabilities, and non-functional aspects of the design and code. These need to be more cleanly separated than is done in (Cunningham and Wang 2001). Requirements 1 and 2 describe functional requirements of the family, which are our primary concerns here. Requirements 3 and 4 express desired characteristics of the framework. Requirement 5 suggests characteristics of the design process. By analyzing the functional requirements, we identify one primary commonality, i.e., frozen spot, as follows:

1. All clients of the framework use the Table ADT's key-based access methods to the collections of records stored in table. (Requirement #1)

We also identify five variabilities, i.e., hot spots, as follows:

1. *Variability in the keys*. Clients of the Table framework can define the keys using many different data structures. (Requirement #1)
2. *Variability in the records.* Clients of the Table framework can define the records using many different data structures. (Requirement #1)
3. *Variability in the external representation of the record state.* For tables stored on external devices, it must be possible to store the state of a record accurately on the external device and restore it to memory when needed. This process may vary somewhat depending upon the nature of the record and the external device. (Requirements #1 and #2)
4. *Variability in the indexing mechanisms.* Different customizations of the Table framework can use different algorithms for indexing the records. (Requirement #2)
5. *Variability in the storage mechanisms*. Different customizations of the framework can use different mechanisms for storing the records. (Requirement #2)

The hot spots #1 and #2 are not completely independent of each other. However, to separate the concerns, we choose to separate the variabilities of keys and records into two different hot spots. Hot spot #3 is a bit subtle, but the need for this variability should be clear as we proceed with the design.

Following the design method outlined above, the framework should allow the five variabilities to be realized independently from each other, which has an implication for the architecture of the Table framework. Before we proceed further, let's look a bit more at the use of formal design contracts for specifying software behaviors.

12.3 FORMAL DESIGN CONTRACTS

Design by Contract is a design approach developed by Meyer (Meyer 1992, 1997). It is motivated by an analogy with a contract in business. In the business setting a contract defines an agreement between a supplier and a client:

1. The supplier must satisfy certain obligations, such as providing the product the client ordered, and expects certain benefits, such as the client paying the established price for the product.
2. The client must satisfy certain obligations, such as paying the supplier the established price for the product, and expects the benefits, such as getting the product.
3. Both the supplier and the client must satisfy certain obligations that apply to all contracts, such as laws and regulations.

Meyer (Meyer 1992, 1997) adopts the concepts of “client”, “supplier” and “contract” into object-oriented design. Building upon earlier work on program verification (Hoare 1969), information hiding (Parnas 1972), data abstraction (Hoare 1972), and abstract data types (Guttag 1977), Meyer introduces logical assertions to describe the contract between the clients (users) of an abstract data type (ADT) and the suppliers (i.e., developers) of the ADT. In Meyer’s approach to object-oriented design and programming, an ADT is normally represented by a class. The key assertions are of three types: preconditions, postconditions, and invariants.

Preconditions and postconditions are assertions attached to each operation of an ADT. A *precondition* expresses requirements that any call of the operation must satisfy if it is to be correct. A *postcondition* expresses properties that are ensured in return by the execution of the call. If the precondition is not satisfied, the operation is not guaranteed to return a correct value or to even return at all. For example, an operation to delete a record from a collection might have a precondition requiring that a record with that key exists and a postcondition requiring that it no longer be an element of the collection.

An *invariant* is a constraint attached to an ADT that must hold true for each instance of the ADT whenever an operation is not being performed on that instance. In object-oriented design, this type of invariant is often called a *class invariant*. For example, in the Table ADT, an invariant might state that the table must *not* have more than one record with a particular key. The invariant gives a condition that must be satisfied to maintain the integrity of the table.

In the client-supplier context,

- a client must satisfy the obligation (the precondition) of an operation to expect to receive the benefit (the postcondition) of getting a correct result from the operation,

- a supplier must satisfy the obligation to make the postcondition of the operation hold upon return whenever the precondition of the operations is satisfied by the call,
- both the client and the supplier must maintain certain properties, the invariants.

In specifying the design of the interfaces of the Table framework, we not only need to give the method signatures (i.e., parameters and return type) but also to express their semantics (i.e. behaviors), using preconditions and postconditions for each method and invariants for the ADT as a whole (Cunningham and Wang 2001).

The simple application of Design by Contract is not by itself sufficient for formal proofs of correctness of the desired properties of framework applications. The concrete classes that implement hot spots in a framework must, of course, preserve the general expectations of the framework specification, that is, they should be behavioral subtypes (Liskov and Wing 1994) of the abstract classes they extend. However, the concrete implementations exhibit richer behaviors than the minimum required by the framework specification. Thus extended techniques are needed to handle these richer behaviors (Soundarajan and Fridella 2000; Hallstrom and Soundarajan 2002). Nevertheless, simple design contract techniques are still quite useful in helping designers explore and refine the requirements and framework designs.

12.4 LAYERED ARCHITECTURE

The overall architecture of the Table framework should embody the frozen spot and, as much as possible, separate the concerns related to each hot spot into an independent component. That is, it should hide the implementation of each hot spot within a separate component, behind a well-defined interface. To use the terminology from Parnas' information-hiding approach to modular software design, the implementation details for a hot spot should be a "secret" of the component that is hidden behind an appropriate "abstract interface" (Parnas 1972; Britton et al. 1981; Cunningham et al. 2004).

Clearly, there is a mix of high- and low-level issues among the hot spots. Clients can define their own key (hot spot #1) and record (hot spot #2) structures and then call the table (frozen spot) to store the records. The table implementation may use some key-based record access mechanism (hot spot #4) paired with some storage structure (hot spot #5).

This mix of high- and low-level issues suggests a hierarchical architecture based on the *Layered Architecture* pattern (Buschmann et al. 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. In the simplest version of this pattern, services in a layer cannot directly call upon services defined more than one layer down. It cannot directly call services defined in a layer above except using specific *call-backs* that it is supplied in calls from the higher level.

As shown in Fig. 12.1, we can define three layers in the Table framework design. From the top to the bottom these include:

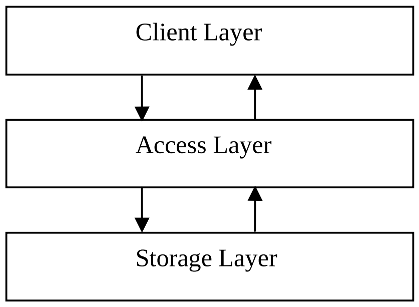

Fig. 12.1. Applying the *Layered Architecture* pattern

Client Layer. This layer consists of the client-level programs that use the table implementation in the layer below to store and retrieve records. Clients of the Table framework implement the user-defined data types for keys and records, which are the variabilities expressed by hot spots #1 and #2.

Access Layer. This layer must provide client programs key-based access to the records in the table. It uses the layer below to store the records physically. Implementations of this layer provide the data structures and algorithms for indexing the records, which is hot spot #4. The interface to this layer represents the frozen spot.

Storage Layer. This layer must provide facilities to store and retrieve the records from the chosen physical storage medium. Implementations of this layer provide the data structures and algorithms for storing the records, for example, a structure in the computer's main memory or a random-access file on disk. The layer expresses hot spot #5.

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 et al. 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.

What about hot spot #3? This hot spot involves the ability to represent a “record” in an external form suitable for storage on some physical storage medium (e.g., rendering it as a sequence of bytes). So, on the surface, it would seem that this would be a structure defined by the Client Layer that is passed through the Access Layer to the Storage Layer, where a call-back to the implementation of the structure in the client may take place. However, a closer examination reveals a more complicated situation. The client's keyed-record may itself consist of a hierarchy of structures, each of which needs to be converted to the external form independently. For some implementations of the Access Layer, a physical record to be stored by the Storage Layer might consist of a group of client keyed-records (e.g., a B-tree node or a hash-table bucket) or it might consist of auxiliary information about the access structure that needs to be made persistent. Because hot spot #3 does not fit cleanly into any of the layers, we place the needed abstraction in a utility module called the Externalization Module.

The various layers and modules need to be kept independent from 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 only the provided interfaces. Next, let us examine the design of the each layer and its interfaces.

12.5 CLIENT LAYER

The design of the *Client Layer* must enable the Access Layer to access client-defined keys and records and should avoid requiring unnecessary programming to use common data types.

12.5.1 Abstract Predicates for Keys and Records

As much as possible, clients (i.e., users) of the table implementations should be able to define their own key (hot spot #1) and record structures (hot spot #2). The internal details of the different types of records and keys, which are implemented in the Client Layer, must be hidden from the Access and Storage Layers. However, the specification of the Access Layer depends upon certain assumptions about the nature of the records and keys. In specifying the operations for the interfaces in this and other layers, we express key features of the keys and records as abstract predicates (Meyer 1997) to make these assumptions more explicit. These are called *abstract* because they are used for specification only; they do not represent functions that are to be built as executable code. The precise definition of these predicates depends upon the particular implementations used in this layer. The abstract predicates associated with the Client Layer are

- `boolean isValidKey(Object key)` that is true if and only if `key` is an element of the set of meaningful keys supported by the client's key class.
- `boolean isValidRec(Object rec)` that is true if and only if `rec` is an element of the set of meaningful records supported by the client's keyed record class.

12.5.2 Keys and the Comparable Interface

As stated earlier, clients of the table implementations should be able to 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` is sufficient to define the functionality of the keys. Any class that implements this interface must provide a public method `compareTo`, which is defined to have the signature and semantics (design contract) as defined below.

To state logical and mathematical expressions in specifications, this chapter uses a Java-influenced notation. The symbol **&&** denotes logical conjunction ("and"), **||** denotes the logical disjunction (inclusive "or"), **!** denotes negation, $\Rightarrow$ denotes logical implication ("if-then"), and **==** denotes equality. The symbol $\forall$ denotes universal quantification ("for all") and $\exists$ denotes existential quantification ("there exists"). For mathematical sets, we use braces **{** and **}** to list the elements explicitly , $\cup$ to denote union, **–** to denote set subtraction, $\in$ to denote membership, and $\varnothing$ to denote the empty set. In appropriate contexts, pairs of parentheses **(** and **)** denote tuple formation. In postconditions, the variable **`result`** refers to the value returned by a function 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.

The description and design contract (pre- and postconditions) for the `compareTo` method are as follows:

- `int compareTo(Object key)` that compares the associated object (`this`) with argument `key` and returns -1 if `key` is greater, 0 if they are equal, and 1 if `key` is less.
  Pre: `isValidKey(this) && isValidKey(key)`
  Post:
  ```
  result == (if this < key then -1
             else if this == key then 0
             else 1)
  ```

Clients can use any existing `Comparable` class for their keys or implement their own.

12.5.3 Records and the Keyed Interface

To enable keys to be extracted from records, we introduce the Java interface `Keyed` to represent the type of objects that can be manipulated by a table (hot spot #2). We model the `Keyed` abstraction as having an abstract attribute `key`.

Any class that implements this interface must implement the method `getKey`, which has the following description and design contract:

- `Comparable getKey()` that extracts the key from the associated record (`this`).
  Pre: `isValidRec(this)`
  Post: `(result == this.key) && isValidKey(result)`

An alternative design for handling the keys and records might be to allow the client to use any Java objects and then to supply appropriate objects that encapsulate the key-extraction and key-comparison operations—developed in accordance with the *Strategy* design pattern (Gamma et al. 1995; Grand 1998). This alternative might enable changes to these operations to be done more dynamically but at the loss of some type safety and of the ability to use the classes in the API that implement the `Comparable` interface. With the approach taken in this section, clients can, if needed, construct wrapper classes that implement the `Comparable` and `Keyed` interfaces and encapsulate the actual key and record objects. This use the *Adapter* design pattern (Gamma et al. 1995; Grand 1998) enables clients to utilize a wide range of pre-defined objects as keys or records as needed.

12.5.4 Interactions among the Layers

The Client Layer thus consists of the `Comparable` and `Keyed` interfaces and the abstract predicates `isValidKey` and `isValidRec` (all of which are part of the framework) and the concrete classes that implement the interfaces (which are part of the customization of the framework for some specific application). The encapsulation of the key and record implementations in the `Comparable`- and `Keyed`-implementing classes, respectively, thus enable the Access Layer to use the client-defined keys and records without knowing the specifics of their implementation. A table implementation in the Access Layer can use the `getKey` method of the `Keyed` interface to extract keys from the client-defined records and can then use the `compareTo` method of the `Comparable` interface to compare the client-defined keys.

12.6 ACCESS LAYER

The design of the Access Layer must provide the Client Layer programs key-based access to a collection of records (frozen spot), enable diverse implementations of the indexing structures (hot spot #4), and support diverse storage structures in the Storage Layer. The primary abstraction of the Access Layer is the Table ADT.

12.6.1 Abstract Predicates for Tables

In the specifications in this section, we use the following abstract predicates to capture assumptions the Table ADT makes about the environment:

- `isValidKey(Object key)` and `isValidRec(Object rec)` which are defined in the Client Layer to identify valid keys and records.
- `isStorable(Object rec)` which is defined in the Storage Layer to identify records that can be stored.

The specifications of other interfaces may also depend upon assumptions about the integrity of a Table ADT instance. We thus introduce the abstract predicate:

- `boolean isValidTable(Table t)` that is true if and only if `t` is a valid instance of `Table` (i.e., satisfies all the design contracts below).

12.6.2 Table Interface

We model the collection of records by the variable `table`, which is a partial function from the set of keys defined by the type `Comparable` to the set of records defined by the type `Keyed`. For convenience, we use the variable `table` to denote either the function or the corresponding set of key-record pairs.

Now, we can define the Table ADT as a Java interface that includes the following ADT invariant and public methods. In English, the *invariant* can be stated:

> *All stored keys and records in the* `table` *are valid and capable of being stored on the chosen external device, and the records can be accessed by their keys.*

Stated more formally, the invariant is:

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

The Table ADT has mutator (i.e., command or setter) operations with the following descriptions and design contracts:

- `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)} )`

The `Table` ADT has accessor (i.e., query or getter) operations with the following descriptions and design contracts:

- `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: `result == #table(r.getKey())`
- `boolean containsKey(Comparable key)` searches the table for the argument `key`.
  Pre: `isValidKey(key)`
  Post: `result == defined(#table(key))`
- `boolean isEmpty()` checks whether the table is empty.
  Pre: `true`
  Post: `result == (#table == ∅)`
- `boolean isFull()` checks whether the table is full.
  Pre : `true`
  Post: `result == (#table` implementation has no free space to store a new record)
- `int getSize()` returns the size of the table.
  Pre: `true`
  Post: `result == cardinality(#table)`

Note that there are several tacit assumptions being made. Having `getSize` return an integer means that the size of the table must be finite, but it is not necessarily bounded. Of course, for unbounded tables `isFull` would always need to return the value `false`. The contracts for the methods other than `getSize` do not preclude the definition of an infinite size table (e.g., with some ranges of key values having records that are generated by a function as needed). However, the behavior of `getSize` would need to be defined for infinite tables. Also a class that implements an infinite table would need to provide a constructor or additional methods for setting up techniques for calculated records that are not explicitly inserted into the table.

12.6.3 Interactions among the Layers

The Access Layer thus consists of the `Table` interface and the `isValidTable` abstract predicate (which form part of the framework itself) and the concrete classes that implement `Table` (which are part of a customization of the framework to create a specific member of the family). Concrete classes that implement the `Comparable` and `Keyed` interfaces are part of the Client Layer. The interactions between the Client Layer and the Access Layer occur 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.

In the design of the Access Layer, the only constraint placed upon the storage mechanism is that the records inserted into the table are capable of being stored and retrieved reliably (i.e., satisfy `isStorable`). Thus the design of the Access Layer enables client-defined keys and records, diverse record access mechanisms, and diverse storage mechanisms. Next, let us examine the Storage Layer and its interface.

12.7 STORAGE LAYER

The *Storage Layer* provides facilities to store records to and retrieve records from a physical storage medium. It encapsulates hot spot #5 and, hence, must enable a diverse range of physical media. Of course, this layer must also support client-defined records in the Client Layer and diverse record-access mechanisms in the Access Layer. It should also enable the access structures in the Access Layer to be stored on the physical media and decouple the implementations in the layers above from the physical media as much as possible.

12.7.1 Abstract Predicate for Storable Records

The specifications of the Access Layer and the Storage Layer interfaces depend upon certain assumptions about the nature of records that can be stored on the physical storage media. In specifying the operations, we express key features of the media in terms of an abstract predicate to make these assumptions more explicit. The predicate defined by the Storage Layer is:

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

12.7.2 Bridge Pattern

To define the interfaces between the Access and Storage layers, we adopt a structure motivated by the Bridge and Proxy design patterns (Gamma et al. 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 implementations of the `Table`'s indexing mechanisms.

The *Bridge* design pattern is useful when we wish to decouple the "interface" of an abstraction from its "implementation" so that the two can vary independently (Gamma et al. 1995; Grand 1998). In this design (as shown in Fig. 12.2), 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.

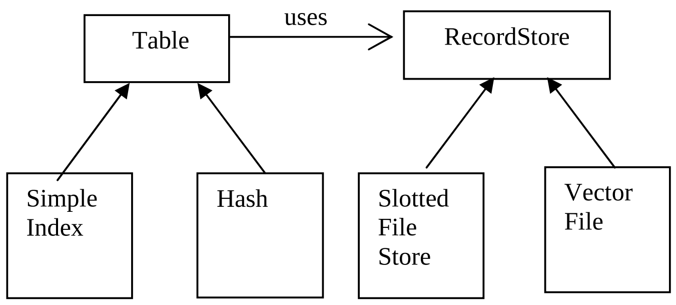

Fig. 12.2. Applying the *Bridge* pattern

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 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.

12.7.3 Proxy Pattern

Because we cannot 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* design pattern (Gamma et al. 1995; Grand 1998).

The idea of the Proxy design 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.

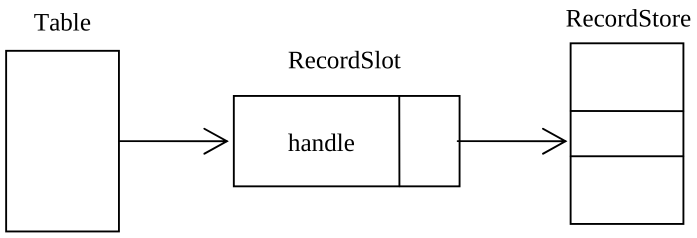

Fig. 12.3. Applying the *Proxy* pattern

In this design, we define the `RecordSlot` abstraction to represent the proxies for the slots within a `RecordStore`. As shown in Fig. 12.3, 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.

12.7.4 RecordStore Interface

We can now specify the `RecordStore` and `RecordSlo`t 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 finite, but unbounded in size.

We define the `RecordStore` ADT as a Java interface that includes the following ADT invariant and public methods. In English, the *invariant* can be stated:

> *All records in the* `store` *are capable of being stored on the selected medium and the stored records can be accessed by their handles.*

Stated more formally in logic, the invariant is:

```
(∀ h, r : r == store(h) : isStorable(r)) &&
(∀ h :: h ∈ alloc == defined(store(h)))
```

The `RecordStore` ADT has operations with the following descriptions and design contracts:

- `RecordSlot newSlot()` allocates a new record slot and returns the `RecordSlot` object.
  Pre: `true`
  Post:
  ```
  result.getContainer() == this &&
   result.getRecord() == NULLRECORD &&
   result.getHandle() ∉ #alloc &&
   result.getHandle() ∈ alloc ∪ {NULLHANDLE}
  ```
- `RecordSlot getSlot(int handle)` reconstructs a record slot using the given `handle` and returns the `RecordSlot`.
  Pre: `handle ∈ alloc`
  Post:
  ```
  result.getContainer() == this &&
  result.getRecord() == #store(handle) &&
  result.getHandle() == handle
  ```
- `void releaseSlot(RecordSlot slot)` deallocates the allocated record `slot`.
  Pre:
  ```
  slot.getHandle() ∈ alloc ∪ {NULLHANDLE} &&
  slot.getContainer() == this
  ```
  Post:
  ```
  alloc == #alloc - {slot.getHandle()} &&
  store == #store –
            {(slot.getHandle(),slot.getRecord())}
  ```

Note that, to support a wide domain of variability in implementation, the parameterless `newSlot` method 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`. For this method, we set the value of a new slot to be `NULLRECORD.` This constant denotes an inert, empty record implemented according to the *Null Object* design pattern (Woolf 1998; Grand 1998). That is, `NULLRECORD` has the same interface as the other records returned by `getRecord` (below) except that it has no data associated with it and the operations have no effect. According to Woolf, "the Null Object encapsulates the implementation decision to do nothing and hides those details from its collaborators" (Woolf 1998). It sometimes avoids a situation where a caller must take a special action to capture error returns from operations.

12.7.5 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. We model the `RecordSlot` ADT as having two abstract attributes, the `container` which is a reference to the associated `RecordStore` and the integer `handle.`

We thus define the `RecordSlot` ADT as a Java interface that includes the following ADT invariant and public methods. In English, the *invariant* can be stated:

> *The handle of a `RecordSlot` object denotes a slot of the `store` that has been allocated, unless it has the value `NULLHANDLE`.*

Stated more formally in logic, the invariant is:

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

The `RecordSlot` ADT has operations with the following descriptions and design contracts:

- `void setRecord(Object rec)` stores the argument object `rec` into this `RecordSlot`.
  Pre: `isStorable(rec)`
  Post: Let `h == getHandle():`
  ```
  (h ∈ #alloc ⇒ store == (#store –
                {(h,#store(h))}) ∪ (h,rec)} )
   && (h == NULLHANDLE  ⇒
      (∃ g : g ∈ #unalloc :
            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 `newSlot` method of `RecordStore`.

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

Post: Let `h == getHandle()`:
`(h ∈ #alloc ⇒ result == #store(h)) &&`
`(h == NULLHANDLE ⇒ result == NULLRECORD)`

- `int getHandle()` returns the handle of this `RecordSlot`.
  Pre: `true`
  Post: `result == this.handle`

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

- `boolean isEmpty()` determines whether the `RecordSlot` is empty (i.e., does not hold a record).
  Pre: `true`
  Post: `result == (getHandle() == NULLHANDLE ||`
  `getRecord() == NULLRECORD)`

Note that `getRecord` returns the inert `NULLRECORD` object if no record has been stored in the slot. Also note that `isEmpty` returns true for either an unallocated handle or the `NULLRECORD` being stored in the slot.

12.7.6 Interactions among the Layers

The Storage Layer consists of the `RecordStore` and `RecordSlot` interfaces and the abstract predicate `isStorable` (all of which are part of the framework) and the concrete classes that implement the interfaces (which are part of an application of the framework). A `Table` implementation in the Access Layer calls a `RecordStore` implementation in the Storage Layer to get `RecordSlot` object. The Access Layer code then calls `RecordSlot` to store and retrieve its records. If needed, a `RecordSlot` object calls back to a `Record` implementation in Access Layer. The `Record` interface is part of the Externalization Module, which we examine in the next section.

The design of the `RecordStore` and `RecordSlot` abstractions and the use of slot handles give the Storage Layer the capability to be implemented using a diverse group of physical media, including both main memory and on-disk structures. These interfaces provide operations with sufficient functionality and make the functionality available in manner that is independent from the actual physical medium used. The combination of these interfaces and the `Record` interface in the Externalization Module (defined in the next section) enable the Storage Layer to be decoupled from the layers above and for the Access Layer to store a wide range of information in the Storage Layer.

12.8 EXTERNALIZATION MODULE

How can the `RecordSlot` mechanism 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 layers above 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. So, once we allow diverse physical media, we have to handle the external byte presentation of record state (hot spot #3).

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

- `void writeRecord(DataOutput out)` writes `this` record to stream `out`.
  Pre: `true`
  Post: suffix of stream `out ==` `this` record's state encoded as byte sequence
- `void readRecord(DataInput in)` reads `this` record from stream `in`.
  Pre: `true`
  Post: `this` record's state `==` prefix of stream `in` decoded from byte sequence
- `int getLength()` returns the number of bytes in the external representation of `this` record (e.g., that will be written by `writeRecord`).
  Pre: `true`
  Post: `result ==` number of bytes in external representation of `this` record

The `Record` interface must also satisfy a *State Restoration Property*, defined as follows:

If, for some `Record` object, a `writeRecord` call is followed by a `readRecord` call with the same byte sequence, the observable state of the `Record` object will be unchanged.

The concrete implementations of the `Record` interface appear in either the Client Layer for client-defined records or in the Access Layer for “records” used internally within a `Table` implementation. The Storage Layer 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` and `RecordSlot` implementations are responsible for routing the stream of bytes to and from the physical storage medium.

The framework design using the `Record` interface takes a low-level approach to handling the conversion of user-defined records to the desired external form. It requires that the users provide facilities for translating their records to/from a sequence of bytes by having the records themselves implement the `Record` interface. An alternative would be to encapsulate this functionality within an externalization object developed in accordance with the Strategy pattern (Gamma et al. 1994). Methods of the externalization object could access the fields of the user's record to create the needed external form and vice versa. This access might be direct using accessor methods the user record provides or it might be indirect using reflection. Taking this approach further, the Storage Layer might be parameterized with other Strategy objects that convert from a device-independent form coming from the externalization object to the form actually stored on the physical device. Given that strict typing is not maintained when a record is externalized, this would be an acceptable, possibly more dynamic alternative to the approach using the `Record` interface. It would also better support external forms such as an XML representation. However, we opt for the simpler, low-level approach for the framework design in this chapter.

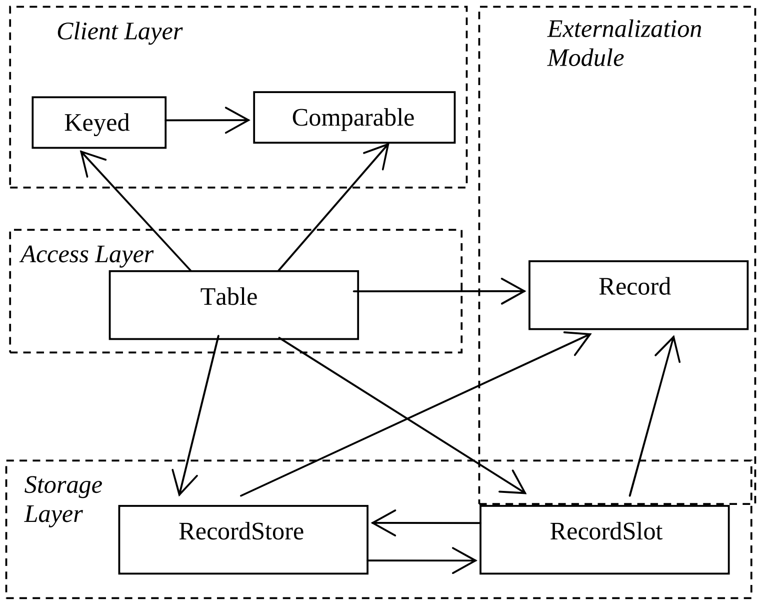

Fig. 12.4. Abstraction Usage Relationships

In summary, Fig. 12.4 shows the *use* relationships among the Client, Access, and Storage Layer and Externalization Module abstractions. The user program in the upper-level Client layer calls the Table ADT directly and the lower layers have callbacks to implementations of the `Keyed`, `Comparable`, and `Record` abstractions defined in the layers above.

12.9 ITERATORS

So far, we have specified the basic structure of the Table framework. More design patterns could be applied to enhance the design of the framework. This section illustrates how to apply the *Iterator* design pattern (Gamma et al. 1995; Grand 1998) in the Table framework. This design pattern enables the client code to access all the records in the table in some order without exposing the internal details of the table implementations. 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 method `next` to return the next element. We can add several useful iterators and iterator-manipulating methods to the framework design.

12.9.1 Table Iterator Methods

As a convenience for clients of the table implementations, we add two iterator accessor methods, `getKeys()` and `getRecords()`, to the `Table` interface (defined in the Access Layer). Remember that the ADT invariant for `Table` must also hold as a precondition and postcondition for these operations.

Here we introduce new notation for describing the semantics of iterators. The abstract attribute `seq` of an `Iterator` denotes the sequence of elements that the iterator yields on any subsequent calls of the `next()` method. The suffix predicate `nodups` operates on sequences and returns `true` if and only if the sequence contains no repeated elements. We also overload the ∈ and ∉ operators to work with sequences as well as sets. The utility 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 one by one.
  Pre: `true`
  Post:
  ```
  result.seq.nodups &&
       (∀ k ::
          k ∈ result.seq == defined(#table(k))
  ```
- `Iterator getRecords()` returns an iterator that enables the client to access all the records in the table one by one.
  Pre: `true`

```
Post: result.seq.nodups &&
      (∀ r ::
         r ∈ result.seq == (∃ k :: r = #table(k)))
```

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: iter.seq.nodups &&
     (∀ r : r ∈ iter.seq : isValidRec(r) &&
        isStorable(r) && !containsKey(r.getKey()))
  Post: table == #table ∪
               {(r.getKey(),r) : r ∈ iter.seq }
  ```

- `void delete(Iterator iter)` deletes the objects from the table whose keys match those returned by iterator `iter`.
  ```
  Pre: iter.seq.nodups &&
     (∀ k: k ∈ iter.seq :
        isValidKey(k) && containsKey(k))
  Post: table == #table –
               {(k,#table(k)) : k ∈ iter.seq }
  ```

We note that the precondition of the `insert(Iterator)`method requires all elements yielded by the iterator to be absent from the table. In practice, this may be difficult to ensure for all calls. Alternative specifications might be to require that an insert of an existing key to either be ignored or result in an update operation, but these would make the iterator version behave differently than the non-iterator version of `insert`. A similar situation arises for `delete(Iterator)` because its precondition requires the presence of every key.

12.9.2 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 external files. The design of this class takes advantage of the Template Method design pattern.

The *Template Method* design pattern (Gamma et al. 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 are concrete methods that implement the shared functionality of the class hierarchy. They are not intended to be overridden by subclasses.

Hook methods are (often 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 includes 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.

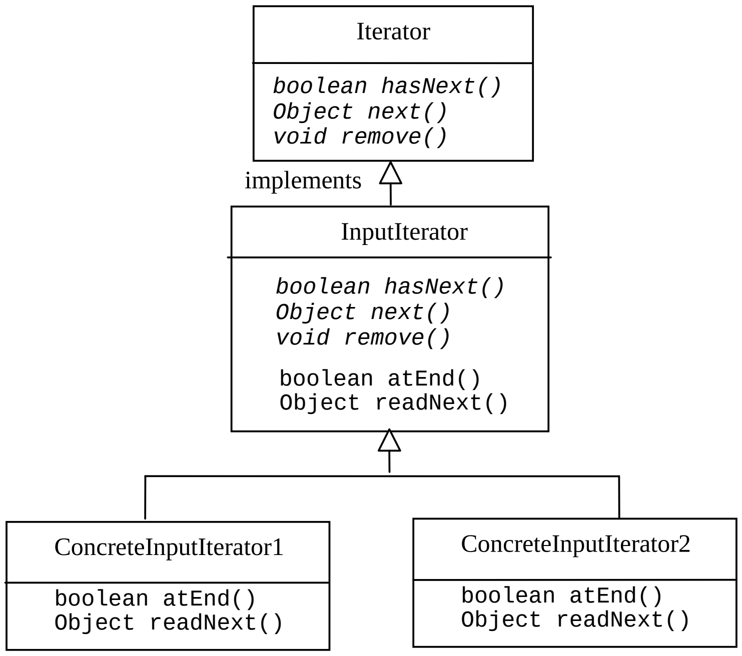

Fig. 12.5. Applying the *Template Method* pattern

A client who wishes to use this class must extend the `InputIterator` class, providing appropriate concrete definitions for the abstract methods. As shown in Fig. 12.5 the `InputIterator` is itself a small framework, with a hot spot concerning that nature of the source from which data objects are being read.

12.9.3 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. Its constructor takes three arguments: an iterator, a selector, and a converter. Its implementation takes advantage of the Decorator and Strategy design patterns as shown in Fig. 12.6.

The *Decorator* design pattern extends the functionality of an object in a way that is transparent to the users of that object (Gamma et al. 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 it 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 design pattern thus allows a `FilterIterator` to provide enhanced functionality at any place that an `Iterator` is used.

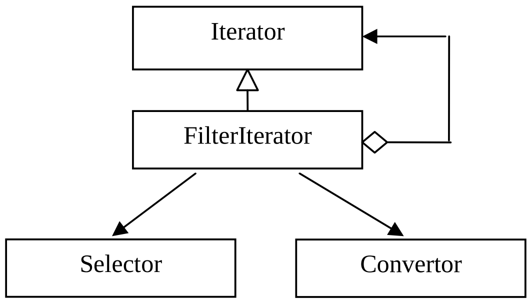

Fig. 12.6. Applying the *Strategy* and *Decorator* patterns

The *Strategy* design pattern abstracts a family of related algorithms behind an interface (Gamma et al. 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 design pattern enables the same `FilterIterator` object to be configured flexibly to have different behaviors as needed.

12.9.4 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 Client Layer that extends the `Keyed` interface with additional methods:

- `int getNumOfKeys()` that returns the number of keys supported by the associated record implementation
- `Comparable getKey(int k)` that extracts the key `k` from the record, where 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. Therefore, we define a new `QueryTable` interface that extends the `Table` interface and includes several new iterator-returning methods. These include:

- `Iterator selectKeys(int k, Selector sel)` that returns the sequence of *primary keys* for the records whose key `k` satisfies the selector `sel`
- `Iterator selectRecords(int k, Selector sel)` that returns the sequence of records whose key `k` satisfies selector `sel`

As a convenience, it is also useful to allow a query to be done using a combination of various primary and secondary key values. We can thus define two additional iterator methods:

- `Iterator selectKeys(Query q)` that evaluates the query `q` and returns the sequence of *primary keys* of all records that satisfy the query
- `Iterator selectRecords(Query q)`that evaluates the query `q` and returns the sequence of all records that satisfy the query

In this design, `Query` is the abstract base of a class hierarchy constructed according to the *Composite* design pattern (Gamma et al. 1995; Grand 1998). This hierarchy, shown in Fig. 12.7, represents the abstract syntax tree of the query commands. The primary operation of the `Query` classes is the method:

- `Iterator eval(QueryTable t)` that evaluates the query in the context of the `QueryTable` argument `t` and returns the primary keys from the table for records that satisfy the query

The concrete class `FieldSelector` is a leaf subclass of the `Query` Composite hierarchy. The class has two attributes, an integer to identify which key field of the multikeyed table is to be considered and a `Selector` Strategy object to determine what values of that (secondary or primary) key field are to be selected for inclusion in the result. When a simple query of this nature is evaluated (e.g., by the `selectKeys` method), the set associated with the resulting iterator consists of all the primary keys from the table for the records that satisfy the `FieldSelector`.

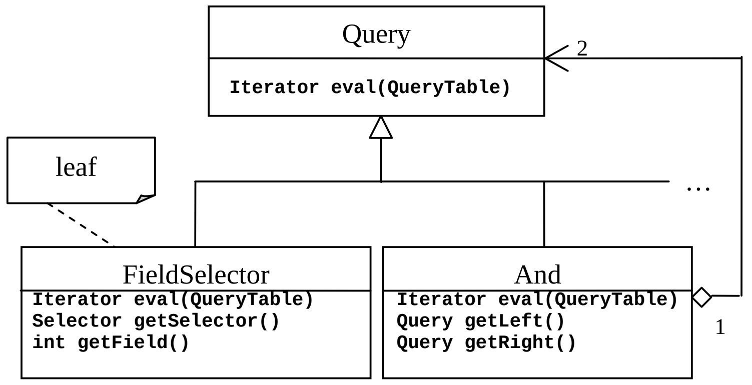

Fig. 12.7. Applying the *Composite* pattern

`Query` also has several composite subclasses denoting operations to be performed. For example, the subclass `And` has two attributes, a left and a right child query. When an `And` query is evaluated by the `selectKeys` method, first the two sub-queries are evaluated recursively to get two sets of primary keys and then the intersection of the two sets is returned. Similarly, `Or` performs a union of the sets, `Diff` subtracts the set represented by the second argment from the first, and `Xor` constructs the symmetric difference of the two sets (i.e., elements in only one of the two sets).

The prototype implementation of the Table framework (Wang 2000) implements a flat query syntax with the same general semantics as described above. The `QueryTable` it implements has an index for each primary and secondary key.

12.10 EVOLVING FRAMEWORKS

The framework design described informally in this chapter is presented from the perspective of an *a priori* design approach for frameworks. Such an approach seeks to derive the framework using systematic analysis (Coplien et al. 1998; Weiss and Lai 1999) and generalization (Schmid 1999; Cunningham and Tadepalli 2006) techniques. In a more traditional approach, 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 and Johnson 1998). The evolution of the different versions of the Table Framework also exhibits several of these *process patterns.*

12.10.1 Three Examples

In most nontrivial frameworks, it is not easy 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 and Johnson 1998).

Often, three implementation cycles are needed to develop a sufficient understanding of the application to construct good abstractions (Roberts and Johnson 1998). The original design of the Table framework was no different despite the simplicity of the problem (Cunningham and Wang 2001). In the exploration of the design, Wang constructed three prototype implementations of `RecordStore` and two implementations of `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.

12.10.2 Whitebox Frameworks

As this framework is defined so far, the Table framework is a pure *whitebox framework* (Johnson and Foote 1988; Fayad et al. 1998). 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 (Johnson and Foote 1988; Fayad et al. 1998). 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.

12.10.3 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 and Johnson 1998). 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 et al. 1998)
- `HashedFileClass`, an implementation that uses a hash table to support the key-based access

In the prototype component library (Wang 2000), the `SimpleIndexedFile` component actually implements the `QueryTable` interface, the extended version supporting more complex queries.

12.10.4 Hot Spots

Even if one does attempt to identify some of the frozen and hot spots beforehand, experience in developing applications with a framework helps to identify more points of shared functionality and more points of variability. Once identified, the shared functionality (new frozen spots) can be incorporated into the framework as concrete classes or as concrete methods of abstract classes. The points of variability (new hot spots) can be incorporated into the framework as abstract hook methods that are refined via inheritance (e.g., using the Template Method pattern). Alternatively, hot spots can be implemented by delegation to classes that encapsulate the required functionality (e.g., using the Strategy and Decorator patterns).

In the Table framework, the input iterator extension is an example of new functionality that might be added to the framework as a result of user experience. Users of the framework discover that they are frequently writing new iterator classes to wrap different data sources. This suggests that a new frozen spot, the `InputIterator` Template Method class, be added to the framework. The hook methods of this class represent a hot spot that can be defined by subclassing the `InputIterator` class.

12.10.5 Pluggable Objects

In early versions of an evolving framework, there is the tendency to have large-grained hot spots implemented in a whitebox fashion using inheritance. As the framework is used, it is sometimes discovered that almost the same subclass is being repeatedly implemented. The solution is to implement the common parts of these subclasses as a concrete class and parameterize it so that the variable aspects can be "plugged in" as an argument to the constructor or some setter method.

In the Table framework, the filtering iterator extension is another example of new functionality that might be added to the framework as a result of user experience. Users of the framework discover that they are frequently selecting a subset of the items in the table using a standard iterator and then performing some transformation on each selected item. This suggests a new frozen spot, the `FilterIterator` concrete decorator class, be added to the framework, with two new hot spots for the selection and conversion functions. The hot spots are implemented as Strategy objects passed as arguments to the constructor.

The Evolving Frameworks patterns include several other steps that the development of long-lived frameworks may take: the gradual inclusion of many useful, fine-grained objects to eventually enable a fully blackbox framework to be constructed and the development of visual builders and language-oriented tools to assist clients to use the framework to develop and test new applications. The Table framework has not yet evolved to the point where these patterns have been used.

12.11 DISCUSSION

A key requirement in the framework design presented in this chapter 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 idea 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 and Tomassia 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 chapter’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 design 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.

Framework design involves incrementally evolving a design rather than discovering it in one single step. Historically, this evolution is a process of examining existing designs for family members, identifying the frozen spots and hot spots of family, and generalizing the program structure to enable reuse of the code for frozen spots and use of different implementations for each hot spot. This generalization may be done in an informal, organic manner as the Patterns for Evolving Frameworks (Roberts and Johnson 1998) or it may be done using systematic techniques such as *systematic generalization* (Schmid 1997, 1999) and *function generalization* (Cunningham and Tadepalli 2006; Cunningham et al. 2006b).

Schmid's methodology seeks a way to identify the hot spots a priori and construct a framework systematically. It identifies four steps for construction of a framework: (1) creation of a fixed application model, (2) hot spot analysis and specification, (3) hot spot high-level design, and (4) generalization transformation. In Schmid's approach, the fixed application model is an object-oriented design for a specific application within the family. Once a complete model exists, the framework designer analyzes the model and the domain to discover and specify the hot spots. The hot spot's features are accessed through the common interface of the abstract class. However, the design of the hot spot subsystem enables different concrete subclasses of the base class to be used to provide the variant behaviors.

Function generalization (Cunningham and Tadepalli 2006; Cunningham et al. 2006b) is another systematic approach. Instead of generalizing the class structure for an application design as Schmid's methodology does, the function generalization approach generalizes the functional structure of an executable specification to produce a generic application. It introduces the hot spot abstractions into the design by replacing concrete operations by more general abstract operations. These abstract operations become parameters of the generalized functions. That is, the generalized functions are higher-order, having parameters that are themselves functions. Such functions can be expressed in functional programming languages, such as Haskell (Peyton Jones 2003), and also in newer, multiparadigm languages such as Scala (Odersky et al. 2006) and application languages such as Ruby (Thomas et al. 2005). After generalizing the various hot spots of the family, the designers can use the resulting generalized functions to define a framework in an object-oriented language such as Java.

The Table framework presented here was originally developed in a somewhat organic fashion but did utilize software design patterns systematically (Cunningham and Wang 2001). This chapter revisits that work from the standpoint of more careful commonality/variability analysis. Future work should examine the framework design using a more formally systematic technique such as function generalization and seek to evolve the framework design more toward a blackbox design.

12.12 CONCLUSION

This chapter describes how commonality/variability analysis, software design patterns, and formal design contracts 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 commonality/variability analysis and the Layered Architecture, Interface, Bridge, and Proxy design 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, Decorator, and Composite design 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.

12.13 EXERCISES

1. Suppose you wish to modify the Client Layer design to use comparison and extraction Strategy objects as described in Section 12.4.3. Discuss the impacts of these changes upon the Client and Access Layer designs.

2. Suppose you wish to modify the `Table` ADT to allow a (conceptually) infinite number of key-value pairs to be held in the table. How would you modify the specification? What new operations, if any, would you add? Suggest an implementation of such a table.

3. Suppose you wish to develop a new `map` operation (similar to what might be found in a functional programming language like Haskell or Lisp) in the `Table` ADT. A `map` operation takes a function and applies the function to every element of some data structure, leaving the modified element in the place of the previous element. Define the method `map` and give its design contract. What restrictions, if any, on the function must be made to ensure the integrity of the `Table`?

4. Suppose you wish to use a more general approach to externalization of the record's internal state than the low-level, byte-stream approach used in this chapter. (See the discussion in Section 12.8.) Give an alternative design and identify the impacts of this change upon the Externalization Module and other aspects of the framework.

5. Characterize the new hot spot(s) introduced into the `FilterIterator` abstraction. What are the variabilities? What design pattern is used to realize each variability?

6. The `InputIterator` uses the Template Method design pattern and the `FilterIterator` uses the Strategy design pattern. Investigate the literature on these patterns (Gamma et al 1995; Grand 1998). What are the relative advantages and disadvantages of these two patterns as means for implementing variability for a hot spot?

7. Using the logical notation of this chapter, state the needed preconditions for the methods `int getNumOfKeys()` and `Comparable getKey(int)` of the `MultiKeyed` abstraction defined in Section 12.9.4.

8. Using the logical notation of this chapter, state appropriate design contracts for the `Iterator`-returning methods `selectKeys(int,Selector)` and `selectRecords(int,Selector)` of the `QueryTable` abstraction defined in Section 12.9.4.

9. Using the logical notation of this chapter, state appropriate design contracts for the `Iterator`-returning query methods `selectKeys(Query)` and `selectRecords(Query)` defined in Section 12.9.4. These can use the `eval(QueryTable)` method of the `Query` class hierarchy.

10. Using the logical notation of this chapter, state an appropriate design contract for the method `eval(QueryTable)` of the `Query` class hierarchy defined in Section 12.9.4.

11. Implement the framework and design an application.
    a. Develop a version of the Access Layer (i.e., `Table`) that uses an array in memory (or `Vector` or `ArrayList`) to create a sorted index of the keys.
    b. Develop a version of the Storage Layer that uses a Java `Vector` (or `ArrayList`) as the storage medium for the records.
    c. Pair the two programs developed in the previous two problems.
    d. Test the application with various kinds of keys and records.
12. Continue the programming exercise above and develop new components. Develop a version of the Access Layer that uses a hash table and pair it with the Storage Layer developed above.
13. Complete the design and implement an Access Layer based on a multikeyed table as defined by the `QueryTable` abstraction in Section 12.9.4.
14. The framework presented in this chapter mostly consists of large-grained components. Examine one of the detailed designs and implementations of the Access Layer from the previous three exercises. Suggest additional frozen spots and hot spots in your design that will allow a useful finer-grained framework to be constructed by using more "pluggable objects".
15. Examine the Java API for stream and file input/output. Identify the hot spots in this framework. How are the hot spots implemented? What design patterns are used to structure the designs?

ACKNOWLEDGEMENTS

The preparation of an earlier version of this chapter was supported, in part, by a grant from Acxiom Corporation titled "An Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)". The authors thank Robert Cook and "Jennifer" Jie Xu for their suggestions for improvements to the paper (Cunningham and Wang 2001). We also thank the two anonymous reviewers, the editors, Chuck Jenkins, and Pallavi Tadepalli for their useful comments on this chapter. As this chapter was being revised, the first author benefited from discussions about various aspects of the framework design with Jenkins. Pallavi Tadepalli is a collaborator on the related function generalization research (Cunningham and Tadepalli 2006; Cunningham et al. 2006b) and Cuihua Zhang is involved with work on the educational aspects of software patterns and framework design (Cunningham et al. 2004, 2006a). This research 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.

REFERENCES

Britton KH, Parker RA, Parnas DL (1981) A procedure for designing abstract interfaces for device interface modules, In: Proceedings of the 5th International Conference on Software Engineering, pp 95-204.

Brooks FP Jr (1986) No silver bullet—Essence and accidents in software engineering, In: Information Processing, Elsevier Science, pp 1069-1076.

Brooks FP Jr (1995) “No Silver Bullet” refired, Chapter 17, In: The mythical man-month, Anniversary edn, Addison-Wesley.

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented software architecture: A system of patterns, Wiley.

Coplien J, Hoffman D, Weiss D (1998) Commonality and variability in software engineering, IEEE Software, vol 15, no 6, pp 37-45.

Cunningham HC, Wang J (2001) Building a layered framework for the table abstraction, In: Proceedings of the ACM Symposium on Applied Computing, pp 668-674.

Cunningham HC, Tadepalli P (2006) Using function generalization to design a cosequential processing framework, In: Proceedings of the 39th Hawaii International Conference on System Sciences, IEEE, 10 pages.

Cunningham HC, Zhang C, Liu Y (2004) Keeping secrets within a family: Rediscovering Parnas. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP), CSREA Press, pp 712-718.

Cunningham HC, Liu Y, Zhang C (2006a) Using classic problems to teach Java framework design. Science of Computer Programming, vol 59, pp 147-169.

Cunningham HC, Liu Y, Tadepalli P (2006b) Framework design using function generalization: A binary tree traversal case study. In: Proceedings of the ACM SouthEast Conference, pp 312-318.

Fayad ME, Schmidt DC, Johnson RE (1999) Application frameworks, In: Fayad ME, Schmidt DC, Johnson RE (eds) Building application frameworks: Object-oriented foundations of framework design, Wiley, pp 3-27.

Folk MJ, Zoellick B, Riccardi G (1998) File structures: An object-oriented approach with C++, Addison Wesley.

Gamma R, Helm R, Johnson R, Vlissides J (1995) Design patterns: Elements of reusable object-oriented software, Addison Wesley.

Goodrich MT, Tomassia R (1998) Data structures and algorithms in Java, Wiley.

Grand M (1998) Patterns in Java, vol 1, Wiley.

Guttag JV (1977) Abstract data types and the development of data structures, Communications of the ACM, vol 20, no 6, pp 396-404.

Hallstrom J, Soundarajan N (2002) Incremental development using object-oriented frameworks: A case study, Journal of Object Technology, Special issue TOOLS USA 2002, vol 1, no 3, pp 189-205.

Hoare CAR (1969) An axiomatic basis for computer programming, Communications of the ACM, vol 12, no 10, pp 45-58,.

Hoare CAR (1992) Proofs of correctness of data representations, Acta Informatica, vol 1, pp 271-281.

Johnson RE, Foote B (1998) Designing reusable classes, Journal of Object-Oriented Programming, vol 1, no 2, pp 22-35.

Liskov B, Wing J (1994) A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems, vol 16, pp 1811-1840.

Mitchell B, McKim J (2002) Design by contract, by example. Addison-Wesley.

Meyer B (1992) Applying design by contract. IEEE Computer, pp 40- 51.

Meyer B (1997) Object-oriented software construction, second edn, Prentice Hall PTR.

Odersky M, Altherr P, Cremet V, Dragos I, Dubochet G. Emir B, McDirmid S, Micheloud S, Mihaylov N, Schinz M,. Stenman E, Spoon L, Zenger M (2006) An overview of the Scala programming language, second edn, LAMP-REPORT-2006-001, Ecole Polytechnique Federale De Lausanne (EPFL), 20 pages.

Parnas DL (1972) On the criteria to be used in decomposing systems into modules, Communications of the ACM, vol 15, no 12, pp 1053-1058.

Parnas DL (1976) On the design and development of program families, IEEE Transaction on Software Engineering, vol SE-2, pp 1-9.

Parnas DL (1978) Some software engineering principles. Infotech State of the Art Report on Structured Analysis and Design, Infotech International, 10 pages, 1978. Reprinted in: Hoffman DM, Weiss DM (eds) (2000) Software fundamentals: Collected papers by David L. Parnas, Addison-Wesley.

Peyton Jones S (2003) Haskell 98 language and libraries: The revised report, Cambridge University Press.

Pree W (1995) Design patterns for object-oriented software development, Addison-Wesley.

Roberts D, Johnson R (1998) Patterns for evolving frameworks, In: Martin R, Riehle D, Buschmann F (eds) Pattern languages of program design 3, Addison-Wesley, pp.471-486.

Schmid HA (1996) Creating applications from components: A manufacturing framework, IEEE Software, vol 13, no 6, pp 67-75.

Schmid HA (1999) Framework design by systematic generalization, In: Fayad ME, Schmidt DC, Johnson RE (eds) Building application frameworks: Object-oriented foundations of framework design, Wiley, pp 353–378.

Soundarajan N, Fridella S (2000) Framework-based applications: from incremental development to incremental reasoning, In: Proceedings of the 6th Interantional Confernce on Software Reuse (ICSR), LNCS 1844, Springer-Verlag, pp 100-116.

Shaw M (1996) Some patterns for software architecture, In: Vlissides JM, Coplien JO, Kerth NL (eds), Pattern languages of program design 2, Addison Wesley.

Sridhar MA (1996) Building portable C++ applications with YACL, Addison-Wesley.

Thomas D, Fowler C, Hunt A (2005) Programming Ruby: The pragmatic programmer's guide, second edition, The Pragmatic Bookshelf.

Wang J (2000) A flexible Java library for table data and file structures, Technical Report UMCIS-2000-07, Department of Computer and Information Science, University of Mississippi.

Weiss DM, Lai CTR (1999) Software product-line engineering: A family-based software development process, Addison-Wesley.

Woolf B (1998) Null object. In: Martin R, Riehle D, Buschmann F (eds), Pattern languages of program design 3, Addison-Wesley, pp. 5-18.