Using Function Generalization with Java to Design a Cosequential Framework

Pallavi Tadepalli and H. Conrad Cunningham
Computer and Information Science, University of Mississippi, University, MS 38677
(662) 915-5358
{pallavi, cunningham}@cs.olemiss.edu

Abstract

Most approaches to building frameworks involve incremental evolution of designs. One such incremental methodology involves the systematic transformation of an object-oriented design to produce the needed generic structure. An alternative approach is function generalization where an application is expressed as a set of functions in a functional programming language and is then transformed in a series of steps to produce a generalized application. The resulting set of generalized functions can be converted to Java code using design patterns to guide the framework construction. In this paper, the function generalization approach is applied directly using Java, where the set of functions are the methods in a class. Several transformation steps produce additional classes and methods that collectively form a framework. The example family used in this paper is the cosequential processing framework.

1. Introduction

An object-oriented software *framework* "is a reusable design expressed as a set of abstract classes and the way their instances collaborate" [8]. Framework design is considered to be a very difficult task where the difficulty is identifying the abstractions that can be tailored to specific applications within the family. These abstractions, which are called *hot spots*, are variable and application specific while the common aspects that are reusable are known as *frozen spots*.

The frozen spots are present within the collaborative structure of the classes in the framework and the concrete implementations of various methods and classes. The hot spots are represented as abstract base classes (or interfaces) in the framework. A particular custom application can be built by providing appropriate implementations of the relevant hot spot abstractions. Frameworks are generally built using design patterns that are structured to fit with the concepts of hot spots and frozen spots [9].

Framework design involves incrementally evolving a design rather than discovering it in one single step. Schmid's *systematic generalization* methodology is one such technique for framework design [9]. In this methodology, framework designers take an object-oriented design for a specific application within the family and convert it into a framework design by a sequence of generalizing transformations. Each transformation corresponds to the introduction of a hot spot abstraction into the structure. The methodology proposes techniques for analyzing the hot spot and constructing an appropriate design for the hot spot subsystem.

Another systematic and incremental methodology is the *function generalization* approach [1]. While Schmid's methodology generalizes the class structure of a prototype application, the function generalization approach generalizes the functional structure of a prototype application. It introduces the hot spot abstractions into the design by replacing concrete operations by more general abstract operations. The abstract operations become parameters of the generalized functions. That is, the generalized functions are higher order, having parameters that are themselves functions. A feature of the work reported in [1] is that it uses the purely functional programming language Haskell [10] to express the functional prototype. After generalizing the various hot spots of the application, the resulting generalized functions are used to generate a framework in an object-oriented language such as Java.

Instead of using a functional language unfamiliar to most programmers, the work reported in this paper applies function generalization directly to Java programs. The prototype application is implemented by Java classes. The set of methods in the prototype represent the functional structure that is generalized by various transformations. The generalization process may necessitate the creation of associated helper classes, whose objects become parameters in the methods of the application class. The resulting set of helper classes and generalized methods in the prototype classes form the framework.

The example family developed in this paper is a cosequential processing family, which is described in the next section. Section 3 presents the generalization of the functional structure of the prototype application, generalizing the hot spots individually. Section 4 describes the framework construction after the various transformations. Then, Section 5 summarizes the paper.

2. Cosequential Processing

Cosequential processing concerns the coordinated processing of two ordered sequences to produce some result, often a third ordered sequence [4]. An important

```
public class ArrayCosequential
{ public ArrayCosequential(int[] a, int[] b)
  { this.a = (a != null) ? a :(new int[0]);
    this.b = (b != null) ? b :(new int[0]);
    i = 0;  j = 0;  k = 0;
    c = new int[a.length+b.length];
  }
  public void merge()
  { while ((i < a.length) && (j < b.length))
    { if(a[i] < b[j])
      { c[k] = a[i];  k++;  i++;  }
      else if(a[i] == b[j])
      { c[k] = a[i];  k++;  i++;  j++; }
      else  // a[i] > b[j]
      { c[k] = b[j];  k++;  j++; }
    } // end while
    while (i < a.length)
    { c[k] = a[i];  k++;  i++; }
    while (j < b.length)
    { c[k] = b[j];  k++;  j++; }
  } // end merge
  public int[] getResult() { return c; }

  private int[] a, b, c;
  private int   i, j, k;
}
```

Figure 1. ArrayCosequential

feature of cosequential processing is that both input sequences must be ordered according to the same total ordering. Another required trait is that the processing should, in general, be incremental. That is, only a few elements of each sequence (perhaps just one) are examined at a time. This important family includes set and bag operations [7] and sequential file update applications [2, 3]. These are practical problems for which presence of a framework can be beneficial in providing good solutions.

As a baseline example, consider a program that takes two ascending sequences of integers and merges them together to form a third ascending sequence (i.e., a bag union operation). This program appears as a Java class `ArrayCosequential` in Figure 1 where the sequences are implemented as arrays. The ordering of the input and output sequences are required as preconditions and postconditions, respectively. The size of the output array is set to be equal to the sum of the sizes of both input arrays. This is because an accurate assumption of the size of the output array cannot be made in advance.

Suppose that variables `i` and `j` represent indices of the "current" elements of the first and second arrays (i.e., arrays `a` and `b`), respectively. The method `merge()` compares `a[i]` to `b[j]` and considers the three cases "less than", "greater than" and "equal". If `a[i]` is less than `b[j]`, then the value `a[i]` is added to the output array and array `a` is advanced to the next entry. If `a[i]` is greater than `b[j]` then, the value `b[j]` is added to the output array and array `b` is advanced to the next entry. When `a[i]` and `b[j]` are equal, then one of the two elements is added to the output array and both arrays are advanced to the next entry.

The method `getResult()` of `ArrayCosequential` is an accessor. It returns a reference to the output sequence to the client.

Building a framework for cosequential processing involves the general principles of framework design. We discuss those in the next section.

3. Framework Design

Schmid's systematic generalization methodology identifies the following steps for construction of a framework [9]:

- creation of a fixed application model
- hot spot analysis and specification
- hot spot high-level design
- generalization transformation

Schmid's methodology begins by creating a class model for the fixed application. The designers then analyze each hot spot in turn and generalize the class model by introducing an appropriate hot spot subsystem.

The function generalization approach begins with a prototype application consisting of a specific set of functions. In this case, the functions are a set of methods in a Java class that may use associated helper classes. A generalizing transformation may replace a data type at a hot spot by one that is more general, thus making it a polymorphic parameter of the methods in the program. As an alternative, a generalizing transformation may replace a fixed specialized operation at a hot spot by an abstract operation. The abstract operation in this case becomes a method that is declared *abstract*. These abstract operations represent variable aspects of the framework. The frozen spots are implemented by methods that call the abstract operations, which represent the common aspects of the framework. After each generalizing transformation, the result is a valid Java class along with its helper classes that can be compiled and executed with appropriate implementations for the abstract methods. Eventually the specific Java class is replaced by an abstract class. The framework then consists of the abstract class along with the helper classes.

Two common principles for framework construction are *unification* and *separation* [5]. The *unification principle* uses inheritance to implement the hot spot subsystem. The common aspects (frozen spots) are implemented by concrete *template methods* in a base class. The variable aspects (hot spots) are represented by a group of abstract *hook methods*. The hook methods are realized by concrete methods in a *hot spot subsystem* in an application of the family.

The *separation principle* uses delegation to implement the hot spot subsystem. The hook methods appear in a separate class that is used by the template methods. The approach taken in this paper is to follow the unification principle where the framework is defined to consist of an abstract base class that defines concrete template methods and abstract hook methods, along with supporting classes.

For the cosequential processing example, we begin with the prototype application represented by the class **`ArrayCosequential`** shown in Figure 1. First, we consider the domain of the family and identify potential hot spots. Then, we design the hot spot subsystem for each and carry out the appropriate transformations to generalize the Java class.

By analyzing the domain of the cosequential processing family, we determine that it must support a variety of built-in and user-defined types of data (not just integers) in the input and output sequences. The data in the sequences might be arranged according to some user-specified total ordering. The family should enable the data to come from a wide range of possible sources and go to a wide range of possible destinations. The output should be a sequence of values that are computed incrementally by examining pairs of values drawn in order from the two input sequences. Domain analysis also establishes the common aspect of the family which is the merging process of two input sequences to produce an output sequence. This common aspect is the frozen spot of the framework and is present in the application as the **`merge()`** method.

Considering the above domain characteristics and examining the prototype application, we can identify the following hot spots:

1. Variability in the total ordering used for the input and output sequences, i.e., of the comparison operators and input sequence type.
2. The ability to have more complex data entities in the input and output sequences, i.e., variability in record format.
3. The ability to vary the input and output sequences independently of each other.
4. Variability in the transformations applied to the data as it passes into the output.
5. Variability in the sources of the input sequences and destination of the output sequence.

We examine these in the subsections that follow.

3.1 Variability in total ordering

In the prototype application represented by the class **`ArrayCosequential`**, the input and output arrays are restricted to be of the primitive type **`int`** and the comparison operations, hence, to integer comparisons. In hot spot #1, the variability in total ordering is incorporated by choosing the interface **`Comparable`** from the Java API as the array type. The resulting class **`CompCosequential`** is shown in Figure 2. The **`Comparable`** interface has the accessor operation **`compareTo(Object)`** that compares objects according to some total ordering. It returns -1, 0 or 1 depending on whether the implicit argument of operation is less-than, equal-to or greater-than the explicit argument. The operation is supported by almost all Java core classes like **`String`** and **`Integer`**. User classes can implement **`Comparable`** and provide the needed operation.

```
public class CompCosequential
{ public CompCosequential
              (Comparable[] a,  Comparable[] b)
  {this.a = (a!=null) ? a :(new Comparable[0]);
   this.b = (b!=null) ? b :(new Comparable[0]);
   i = 0; j = 0;  k = 0;
   c = new Comparable[a.length+b.length];
  }
  public void merge()
  { while ((i < a.length) && (j < b.length))
    { if(a[i].compareTo(b[j]) < 0)
      { c[k] = a[i]; k++;  i++; }
      else if (a[i].compareTo(b[j]) == 0)
      { c[k] = a[i];  k++;  i++;  j++; }
      else   //(a[i].compareTo(b[j]) > 0)
      { c[k] = b[j]; k++;  j++; }
    }   //end while
    while i < a.length)
    { c[k] = a[i];   k++;  i++; }
    while (j < b.length)
    { c[k] = b[j];  k++;  j++;  }
  }   //end merge
  public Comparable[] getResult() { return c; }

  private Comparable[] a, b, c;
  private int          i, j, k;
}
```

Figure 2. CompCosequential

```
public interface Keyed
{  /*returns the key of the element.*/
   public Comparable getKey();
}
```

Figure 3. Keyed

3.2 Variability in record format

The input and output sequences in **`CompCosequential`** all contain elements of the same class type. The type may be any built-in or user-defined class that implements **`Comparable`**. This is because the **`compareTo`** method is defined by the various built-in classes to compare only elements of identical types. Nevertheless, the element of type **`Comparable`** is still a simple structure. Normally, however, applications in the cosequential processing family work with complex structures.

The aim of hot spot #2 is to allow elements of the sequences to be more complex structures such as *records*. A record consists of multiple fields where a subset defines the *key*. The key value controls the ordering of the record within a sequence. In this transformation that produces the class **`KeyedCosequential`**, the three arrays will continue to have elements of a single type. The ability to have complex records is supported by defining an interface **`Keyed`** as shown in Figure 3, which must be implemented by the elements of the sequences. The elements are compared by comparing the key values of the **`Keyed`** elements. The key values are of type **`Comparable`** as shown in Figure 4. They are extracted with the **`getKey()`**

method and stored in two variables **currA** and **currB** respectively.

```
public class KeyedCosequential
{public KeyedCosequential(Keyed[] a, Keyed[] b)
 { this.a = (a!=null) ? a :(new Keyed[0]);
   this.b = (b!=null) ? b :(new Keyed[0]);
   i = 0; j = 0; k = 0;
   c = new Keyed[a.length+b.length];
 }   // end constructor
 public void merge()
 { while ((i < a.length) && (j < b.length))
   { currA = ((Keyed)a[i]).getKey();
     currB  = ((Keyed)b[j]).getKey();
     if (currA.compareTo(currB) < 0)
     { c[k] = a[i]; k++; i++; }
     else if (currA.compareTo(currB) == 0)
     { c[k] = a[i]; k++; i++; j++; }
     else  // (currA.compareTo(currB) > 0)
     { c[k] = b[j]; k++; j++; }
   }   // end while
   while (i < a.length)
   { c[k] = a[i]; k++; i++; }
   while (j < b.length)
   { c[k] = b[j]; k++; j++; }
 }   // end merge
 public Keyed[] getResult() { return c; }

 private Keyed[]       a, b, c;
 private int           i, j, k;
 private Comparable    currA,currB;
}
```

Figure 4. KeyedCosequential

3.3. Independent variability of sequences

In the **KeyedCosequential** class, the three arrays have elements of the same record type. However, some cosequential applications require the record structures to vary among the sequences. For example, the sequential file update application applies transactions stored in a transaction file against master data records stored in a master file. The master and transaction records normally carry different information in different record structures. The master and transaction files are inputs while a new master file is the output. For any master record, there may be several matching transaction records with the same key. In other applications, the output may be a simple record rather than a long sequence. For example, an application to sum the values that appear in the first input sequence but not in the second would be a reasonable application of the family. Thus, in hot spot #3, the independent variability of sequences is handled by the **Keyed** interface.

There can be several classes that implement the interface and support different record structures of the various sequences. The key extraction function **getKey()** for each implementation should return the keys according to same total ordering. The comparison of keys continues as earlier. However, certain transformations are required to map the input records to output record types. Thus, we introduce two new transformation functions that appear as two abstract methods (**transA** and **transB**). These two methods convert the input record element to the output record type. In the new class **TKeyedCosequential**, the previous statements

c[k] = a[i]; c[k] = b[j]; are replaced with

```
c[k] = transA(a[i]);
c[k] = transB(b[j]);
```

respectively. The new **TKeyedCosequential** class is now declared to be **abstract** as it contains two abstract methods and one final method. The merge method is declared to be **final**. An application can now be built by subclassing the abstract **TKeyedCosequential** class and providing appropriate implementations for the **transA** and **transB** methods. To facilitate easy access to the output sequence in the application, a helper method **getK()** whose signature is **protected int getK(),** is introduced. This method returns the current index of the output sequence that is otherwise manipulated by the application. Everything else remains the same as in **KeyedCosequential.**

3.4. Variability of transformations

The **TKeyedCosequential** class allows simple mapping transformations between input and output records. Practical scenarios, however, require more complex transformations as is evident in the sequential file update application. It consists of a master file of records and a transaction file of updates to be performed on the master file to produce a new master file as output. Each key may be associated with no more than one record in the master file. However, there may be any number of update transactions that must be performed against a master record before the new master record can be output. Thus, there needs to be some local state maintained throughout the processing of all the transaction records associated with one master record. In hot spot #4, the variability in transformation functions to convert input sequences to the output sequence appear as abstract hook methods **transLt**, **transGt** and **transEq**. These replace the previously introduced methods **transA** and **transB.** The resulting class **TransKeyedCosequential** is shown in Figure 5. Other changes made in this class include the introduction of two new helper methods, **getA** and **getB.** These methods return the "current" record values in each sequence.

Additional methods **getNextA** and **getNextB** access the elements and their respective key values at the current indices in each of the arrays and then increment the index to the next position in the array. The current elements (or records) in each array are stored in the variables **currRecA** and **currRecB** while their corresponding keys are stored in **currA** and **currB**, respectively.

```
abstract public class TransKeyedCosequential
{ public TransKeyedCosequential
                    (Keyed[] a, Keyed[] b)
  { this.a = (a!=null) ? a :(new Keyed[0]);
    this.b = (b!=null) ? b :(new Keyed[0]);
    i = 0; j = 0; k = 0;
    c = new Keyed[a.length+b.length];
    aNotEmpty=true; bNotEmpty=true;
  }
  final public void merge()
  { getNextA(); getNextB();
    while (aNotEmpty && bNotEmpty)
    { if (currA.compareTo(currB) < 0)
      { transLt(); getNextA(); }
      else if (currA.compareTo(currB) == 0)
      { transEq(); getNextAEquals();
        getNextBEquals(); }
      else // (currA.compareTo(currB) > 0)
      { transGt();  getNextB(); }
    } // end while
    while (aNotEmpty && !bNotEmpty)
    { transBEmpty(); getNextA();    }
    while (!aNotEmpty && bNotEmpty)
    { transAEmpty(); getNextB();     }
    finish();
  } // end merge
  protected int getK() {return k;}
  protected final Keyed getA()
  { return currRecA; }
  protected final Keyed getB()
  { return currRecB; }
  private void getNextA()
  { if (i < a.length)
    { currRecA = (Keyed)a[i];
      currA = currRecA.getKey();
      i++;  aNotEmpty=true;
    }
    else
    { currA = null;currRecA = null;
      aNotEmpty=false; }
  } // end getNextA
  private void getNextB()
  { if (j<b.length)
    { currRecB = (Keyed)b[j];
      currB = currRecB.getKey();
      j++; bNotEmpty=true;
    }
    else
    { currB = null; currRecB = null;
      bNotEmpty=false; }
  } // end getNextB
  public Keyed[] getResult() { return c; }
  //hook methods
  abstract protected void transLt();
  abstract protected void transGt();
  abstract protected void transEq();
  protected void finish() {}
  protected void transAEmpty(){ transGt(); }
  protected void transBEmpty(){ transLt(); }
  protected void getNextAEquals()
  { getNextA(); }
  protected void getNextBEquals()
  { getNextB(); }

  private Keyed[]     a, b, c;
  private int         i,  j, k;
  private Comparable  currA,     currB;
  private Keyed       currRecA,  currRecB;
  private boolean     aNotEmpty, bNotEmpty;
}
```

Figure 5. TransKeyedCosequential

To provide appropriate behavior when the keys are equal, other hook methods have been added to the `TransKeyedCosequential` class. These new methods are `getNextAEquals` and `getNextBEquals`, which access the subsequent elements from the individual sequences as required. The methods `transAEmpty` and `transBEmpty` were also added to provide variability in transformation when one of the two sequences is empty. Default behavior has been provided for the above four methods. One other hook method `finish` is added to take the final state of the computation and complete the output sequence.

3.5. Variability of source/destination

The fifth hot spot allows independent sources for inputs along with variable output destinations. To accomplish this, different abstractions serve as wrappers for the variable sources and destinations for the values in the sequences. The `Iterator` interface in the Java API serves as a useful abstraction for input sequences. Its methods `hasNext()` and `next()` check for the existence of an additional element and returns that element, respectively. Objects that implement the `Iterator` interface are returned by a number of built-in collection classes in the Java API. Programmers can also develop their own classes that implement this interface.

The `Cosequential` class shown in Figure 8 encapsulates the use of the `Iterator` methods inside helper methods `getNextA()` and `getNextB()` to keep subclasses from using the iterator inappropriately. The methods `getA()` and `getB()` give access to the current record on the input sequences. We can introduce a Java interface `OutSeq` as shown in Figure 6 to abstract the output sequence. It has two methods `put(Object)` and `close()` that append an element at the end of the output and terminate the sequence when finished. As a result, the methods `getK()` and `getResult()` can be removed. The classes that implement the `Iterator` and `OutSeq` interfaces provide the encapsulations for sources and destinations, respectively. The `Cosequential` class in Figure 8 encapsulates the use of the `OutSeq put()` method inside the helper method `put()`; method `close()` is only used by the client of the framework.

```
public interface OutSeq
{   public void put(Object r);
    public void close();
}
```

Figure 6. OutSeq

4. Framework Construction

The unification principle of framework construction uses inheritance to implement the hot spot subsystem. The `Cosequential` class in Figure 8 on the next page is the abstract base class. It consists of one *template method* `merge()`, shown in Figure 7, that implements the common behaviors (frozen spots) and several abstract hook methods that implement the variable aspects(hot spots) of the system. This class uses the Template Method design pattern [6] to structure the framework.

A specific application of the framework must subclass **`Cosequential`** to provide appropriate definitions of the hook methods. It must also provide appropriate implementations of the **`Keyed`** and **`OutSeq`** interfaces. Depending upon the nature of the keys, a new implementation of the **`Comparable`** interface may also be needed. Similarly, a new implementation of the **`Iterator`** interface may be needed to provide the input sequences to the **`merge()`** procedure. A class diagram that depicts the relationships among the various classes and interfaces that form the cosequential processing framework is shown in Figure 9. An application of the cosequential processing framework to the master-transaction update problem is given in [1].

```
final public void merge()
{ getNextA(); getNextB();
  while (aNotEmpty && bNotEmpty)
  { if (currA.compareTo(currB) < 0)
    { transLt();
      getNextA();
    }
    else if (currA.compareTo(currB) == 0)
    { transEq();
      getNextAEquals();
      getNextBEquals();
    }
    else  // (currA.compareTo(currB) > 0)
    { transGt();
      getNextB();
    }
  }   //end while
  while (aNotEmpty)
  { transBEmpty();
    getNextA();
  }
  while (bNotEmpty)
  { transAEmpty();
    getNextB();
  }
}
```

Figure 7. merge Method for class Cosequential

```
import java.util.*;
abstract public class Cosequential
{ public Cosequential
        (Iterator _a, Iterator _b, OutSeq _c)
  { a=_a; b=_b; c =_c;
    aNotEmpty = (a != null) && a.hasNext();
    bNotEmpty = (b != null) && b.hasNext();
    if (c == null)
    { throw new RuntimeException
         ("Output sequence not initialized");
    }
  }  // end constructor
  //hook methods
  abstract protected void transLt();
  abstract protected void transGt();
  abstract protected void transEq();
  protected void transAEmpty() { transGt(); }
  protected void transBEmpty() { transLt(); }
  protected void getNextAEquals()
  { getNextA(); }
  protected void getNextBEquals()
  { getNextB(); }
  protected void finish() {}
  //template method
  final public void merge()  {...} //Figure 7
  //helper methods
  protected final Keyed getA()
  { return currRecA; }
  protected final Keyed getB()
  { return currRecB; }
  protected void put(Object r) { c.put(r); }
  private void getNextA()
  { aNotEmpty = a.hasNext();
    if (aNotEmpty)
    { currRecA = (Keyed)a.next();
      currA = currRecA.getKey();
    }
    else  { currA = null;  currRecA = null; }
  }//end getNextA
  private void getNextB()
  { bNotEmpty = b.hasNext();
    if (bNotEmpty)
    { currRecB = (Keyed)b.next();
      currB = currRecB.getKey();
    }
    else  { currB = null;  currRecB = null; }
  }//end getNextB

  private Iterator a,b;
  private OutSeq c;
  private Comparable currA,currB;   //keys
  private Keyed currRecA,currRecB;  //records
  private boolean aNotEmpty, bNotEmpty;
}
```

Figure 8. Cosequential

6. Conclusion

The cosequential processing framework can be used to build a number of real world applications such as set and bag operations and sequential file update applications. Designing the framework to suit a variety of applications involved understanding the underlying applications and creating appropriate abstractions. The structuring of the framework involves use of design patterns to lay out the hot spots and frozen spots of the system. The framework construction follows the function generalization approach where the functional structure of the program is generalized. It is similar to Schmid's systematic generalization approach in its systematic analysis of the application and employing domain knowledge to determine the various points of variability. Each point of variability (hot spot) compels separate and appropriate generalization transformations to the design.

After each transformation, the resulting class is executable along with certain additional required implementations. This is unlike Schmid's work where no executable programs are created. By having each result executable, the designers were able to test each hot spot thoroughly which helped them to find flaws existing in the framework. To test the framework a master/transaction file update application was also successfully created.

The function generalization technique worked reasonably for the cosequential processing family. The technique as illustrated in this paper can be used in developing other examples.

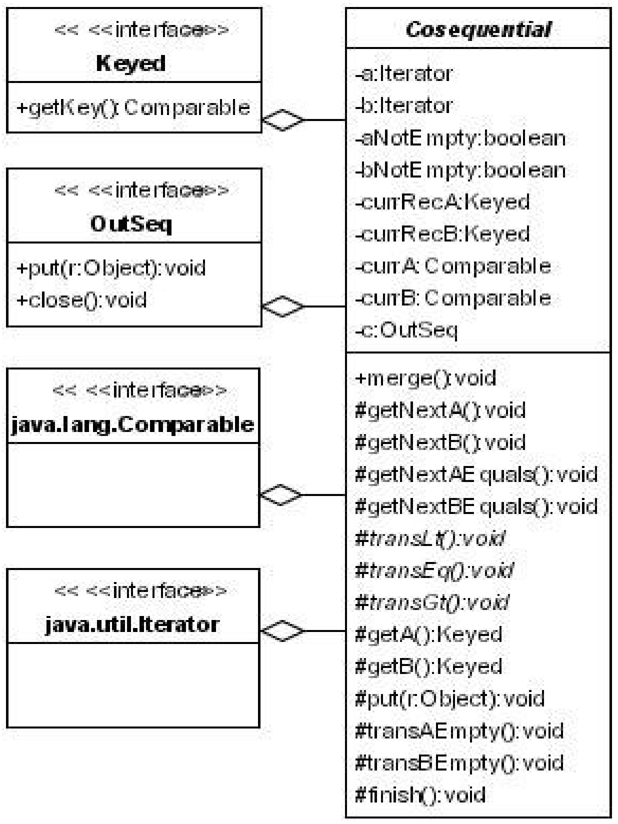

Figure 9. Cosequential Framework Classes

Acknowledgements

This work is supported in part by a grant from Acxiom Corporation titled “The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE).”

References

[1] H. C. Cunningham and P. Tadepalli. “Using Function Generalization to Design a Cosequential Processing Framework,” Manuscript, 21 pages, December 2004.

[2] E. W. Dijkstra. “Updating a Sequential File,” Chapter 15, In *A Discipline of Programming*, pp. 117-122, Prentice Hall, 1976.

[3] B. Dwyer. “One More Time—How to Update a Master File,” *Communications of the ACM*, Vol. 24, No.1, pp.3-8, January 1981.

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

[5] M. Fontoura, W. Pree, and B. Rumpe. *The UML Profile for Framework Architectures.* Addison-Wesley, 2002.

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

[7] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java,* Wiley, Third edition, 2004.

[8] R. Johnson. “Frameworks Home Page,” http://st-www.cs.uiuc.edu/users/johnson/frameworks.html. Last accessed: November 7, 2004.

[9] H. A. Schmid. “Framework Design by Systematic Generalization," In M. E. Fayad , D. C. Schmidt, and R. E. Johnson, editors, *Building Application Frameworks*, pp. 353-378, Wiley, 1999.

[10] S. Thompson. *Haskell: The Craft of Functional Programming*, Second edition, Addison-Wesley, 1999.