Compressed Collection: A design pattern for representing data collections in a space-efficient manner

Henry Rabinowitz, Gary Sevitsky, John Martin
NYNEX Science & Technology, Inc.

***Intent*: To represent and maintain a collection of data containing an ordered attribute in a space-efficient manner.**

Motivation

Many companies collect and make projections of data on customer demand for various services. The data collections for this historical and projected data are commonly represented with much redundancy in table format year by year:

Demand for services

Time	**Service**	**Quantity**
1994	A	10
1994	B	25
1995	A	10
1995	B	25
1996	A	12
1996	B	25
1997	A	12
1997	B	20

To represent such a table in an object-oriented format, say for an object database, a conventional approach would be to create a collection of objects, where each line in the table becomes a single object:

Demand for services (a collection)

1994 A 10	1994 B 25	1995 A 10	1995 B 25	1996 A 12	1996 B 25	1997 A 12	1997 B 20

We would like to represent and manipulate this data using fewer objects, in a more space-efficient, less redundant manner. The question is how can we "compress" the representation of the data, while still retaining access to the data in the form of objects.

Idea

Represent the demand for services as a time-ordered collection of demand objects, each of which has an associated time *interval*, i.e., a range of time *[s, e)* which includes a start date *s* and extends up to but not including an end date *e*. Think of the interval as including the start date and ranging up to but not including the end date.

Demand for services (a collection)

[1994-1996) A 10	[1996-1998) A 12	[1994-1997) B 25	[1997-1998) B 20

In the above representation, "Demand for services" is a time-ordered, time-disjoint collection of Demand objects representing demands for Service A, followed by a similar collection of demands for service B. Thus the collection "Demand for services" is actually two "sequences" of time-ordered demands, the first sequence for service A, and the second sequence for service B.

Suppose we now need to add in new demand, represented in the traditional tabular fashion as

1995	A	12

The semantics of inserting this demand into the above representation depend on whether this new demand is intended to replace the previous value or to augment its quantity. We would like to allow for both kinds of semantics. (We assume that the quantity of demand is a single-valued function of date and service-type.)

If the new demand *replaces* the previous value, then the resulting collection looks like:

[1994-1995)	[1995-1998)	[1994-1997)	[1997-1998)
A	A	B	B
10	12	25	20

If the new demand is *added* to the previous values, the resulting collection looks like:

[1994-1995)	[1995-1996)	[1996-1998)	[1994-1997)	[1997-1998)
A	A	A	B	B
10	24	12	25	20

Solution

Represent the data as a collection of objects, each possessing a time interval. The collection is time-sequenced, with possibly several independent time sequences depending on attributes of the data. (In the example above, we had a time sequence for service A, followed by a separate time sequence for service B). The time intervals for objects within each sequence are disjoint (no overlap).

The collection is a subclass of an abstract type which we will call MultiSequence, which signifies that it consists of a number of time-ordered sequences of objects. The objects in the collection are all of a subclass of an abstract type which we will call Temporal, which signifies that they each contain a time interval.

When inserting a new object into a MultiSequence, there are 3 issues:

1. Under certain conditions, objects with overlapping or adjacent time intervals can be coalesced into a single object. This is true if they are identical in all relevant ways. In the example above, if service and quantity are identical in two adjacent or overlapping intervals, the objects can be coalesced. This criterion is represented by the virtual method

```
boolean Temporal::coalescable(Temporal* other)
```

which must be defined in the subclass. This is true if other can be coalesced with this.

2. Since a given time-ordered collection can have multiple sequences depending on attributes of the data, each subclass of Temporal must provide a method

```
boolean Temporal::sameSequence(Temporal* other)
```

   which is true if this and other belong to the same time-ordered sequence within a collection. In the example above, sameSequence() is true if two objects have the same value for service.

3. When objects with overlapping time intervals are combined, it may be necessary to add certain quantities, or otherwise aggregate data in the two constituent objects in order to create a new object. The virtual method

```
Temporal* Temporal::merge(Temporal* other)
```

   which must be defined in the subclass performs this role. merge() merges data from other into this.

To insert a Temporal object into a MultiSequence, we use

```
void MultiSequence::insert(Temporal* tb, Boolean replace)
```

where the boolean value replace indicates whether the inserted object replaces existing objects for the time period of tb, or is to be merged in with the existing objects.

To retrieve the Temporal objects in a collection that are valid at a particular date, use

```
Collection<Temporal*> MultiSequence::at(const Date& date)
```

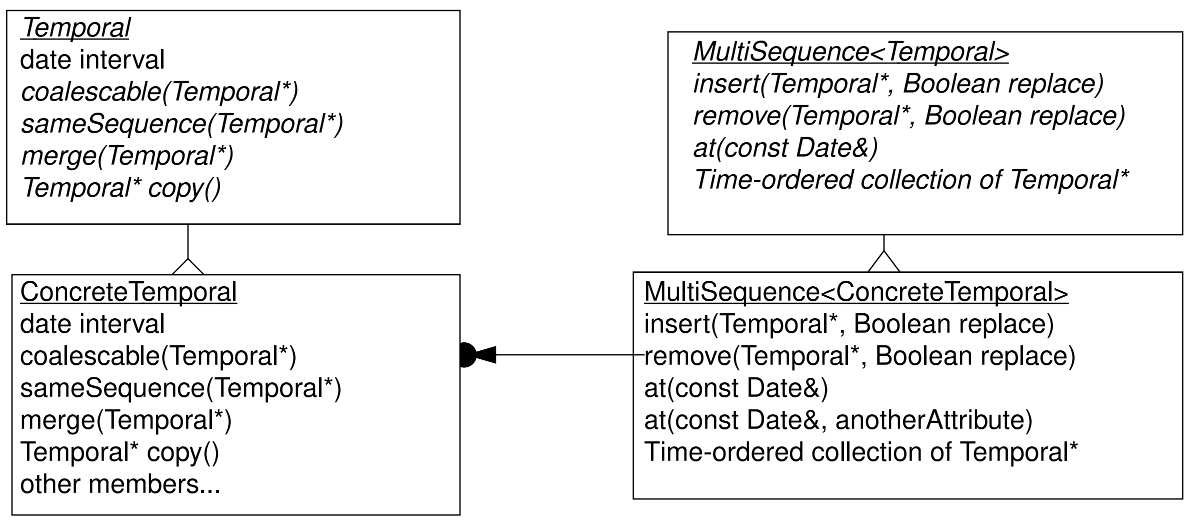
the at() method on the MultiSequence does the retrieval, effectively decompressing the data. It returns a collection since it may retrieve at most one object for each sequence within the original collection. It would be nice to be able to specify exactly which sequence to retrieve from: then we could retrieve a single element of the original collection. But such a method must be specified by the creator of the subclass ConcreteTemporal since it requires semantic knowledge of the criteria by which sequences are formed within the MultiSequence. For example, for the specific collection used in the example above, we could have a method

```
ConcreteTemporal* MultiSequence::at(const Date& date, const Service& service)
```

This method retrieves the single object representing the value of the collection at a given date for a given service, or NULL if there is none.

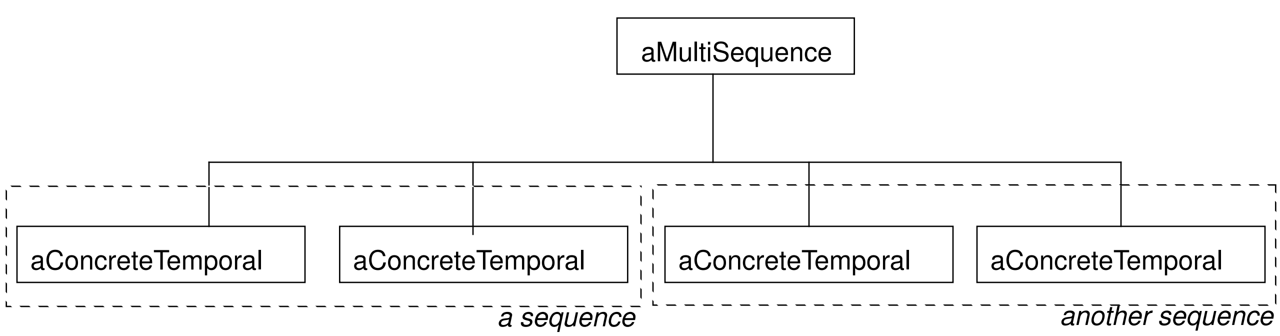
Structure

The class diagram follows the conventions established in [Gamma, et. al.]. Note, italics mean abstract class or virtual method.



Object Structure:

The object diagram shows the object structure of a typical MultiSequence object. It contains a collection of ConcreteTemporal objects. This



collection is ordered as a series of sequences, where each sequence is a time-ordered collection of ConcreteTemporals that share certain characteristics. Note that *sequence* is not an explicit class in this model.

Consequences:

The "compressed collection" design pattern changes the way data is represented as objects from object = data-point to object = range of data, where the range can be of varying size. This shift of representation is not difficult to explain; the complexity comes in implementing some of the methods needed to make it work.

Advantages

+ Compact representation. If the data doesn't vary much over time, considerable space compression can be achieved. The degree of compression cannot be quantified in a general way, since it depends on properties of the data itself.

In a demand forecast used by the NYNEX Network Exploratorium, this pattern resulted in a 72% reduction in the number of Demand objects. Since the space overhead of inheriting from Temporal is just the storage of one extra date (to give a date range), the space savings in this example is essentially 72%.

+ The external interface is natural and simple.

+ The compressed collection representation exposes regularities in the data that can be hidden in the mass of data from a table view.

+ The compressed collection representation can also cut down on the amount of data that must be manipulated by programs that manipulate time-based data.

+ Algorithms to handle this data may end up with fewer iteration steps, since they do not need to iterate once for each year.

+ This pattern, with some extensions, can be used to implement a 1-n or m-n time-ordered relation between two classes. For example, suppose an Arc in a network graph is associated with a collection of demands for service over that Arc of the kind we have shown. The relation between Arc and Demand can be modeled using this pattern.

Disadvantages

- More calculation is required to create and insert into a MultiSequence than with the traditional representation. (However, the complex algorithms for this calculation are all safely encapsulated in the superclass MultiSequence. So the "user" of subclasses need not confront this complexity.)

- More calculation is required to find the value at a given date. It is harder to index directly into the data based on a given date, e.g., to find the quantity of Demand for service B in 1994. It is necessary to do either a sequential search, or to create some sophisticated time-based indexing structure.

- Once we define Temporal as an interval-containing type, then we must implement full interval arithmetic algorithms for all functions involving Temporal objects, even though our originating data is always coming in in year-by-year chunks. This adds algorithmic complexity internally to the objects in this pattern, though the external interface is not complex.

- The insert() method can cause the object being inserted to no longer exist (e.g., if it is merged into a larger object, or split into smaller objects). this can be disconcerting to the unwary employer of this method. E.g.,

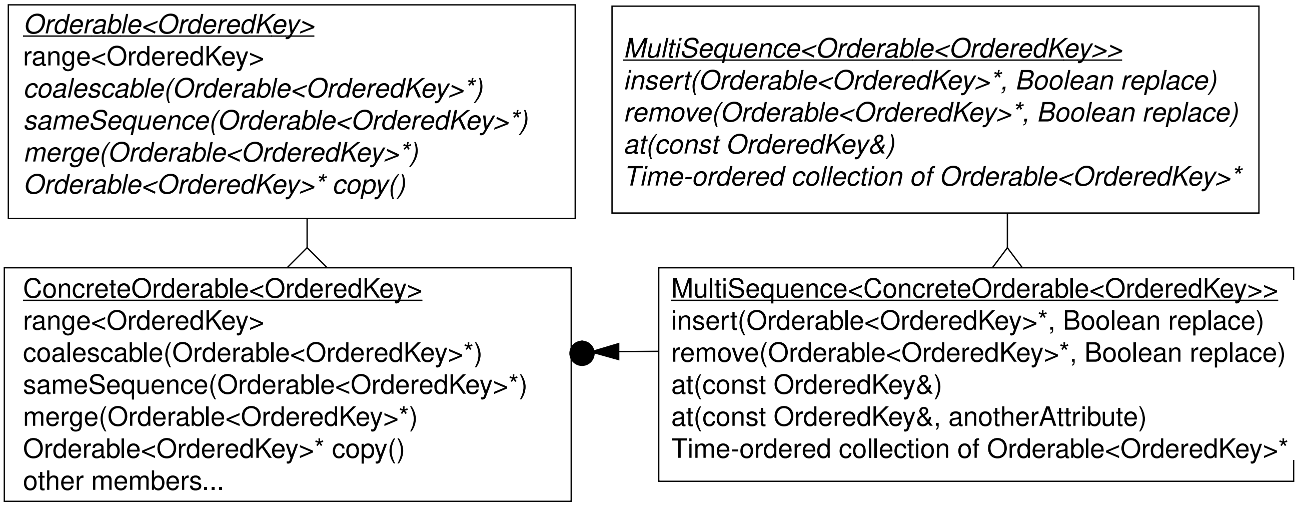
```
Demand* d;
AZDemands ds; // a collection of demands between A and Z
...
ds.insert(d);
d->setPlannable(true);// this is an error. d may no longer be a valid pointer
```

- The at() method for the collection needs to be specified in a way that depends on the attributes of ConcreteTemporal in order for it to return a unique result.

- The algorithmic complexity introduced by using this pattern may not be worth the space savings achieved unless the space savings are substantial or memory is at a premium.

Generalization

The motivating example given above shows compression of time-based data sequences. This is a frequently occurring subset of a larger class of data sequences that contain some kind of ordered field. Given that a sequence of data objects of the same type each contains a field that is orderable, it is possible to compress the sequence in the same manner as we did for time-based data sequences.



Implementation:

This pattern is a special case of the Template Method pattern[Gamma, et. al., p.325], since the code for MultiSequence::insert(Temporal*) can be written once in the base class and used as is. It does not require further modification. The “user” of this pattern need only specify the coalescable(), sameSequence(), and merge() methods for the subclass.

One implementation alternative is to elevate sequence to a full-fledged class participant in this pattern. Then a MultiSequence would be a collection of Sequences. An advantage of this approach is that the method Sequence::at(OrderedKey&) returns a single Orderable object. The problem is that the specification of which Sequence we want to deal with requires information that is not available in the base class Orderable or MultiSequence<Ordereable>.

The most complex part of the pattern to implement is the code for Multisequence::insert(), because it must potentially coalesce or delete objects. Fortunately, the complexity of this algorithm is encapsulated in the Multisequence class, and is hidden from the creators of subclasses.

Uses

The Compressed Collection pattern has been used in the NYNEX Network Exploratorium project, a toolkit for understanding and optimizing the planning of telephone facilities networks. The pattern is used to implement time-ordered relationships in the data repository for the large volume of data stored in the Network Exploratorium. Since the data repository is implemented as an object database, and the data is interconnected in a manner difficult to partition, it is desirable to keep the entire database as a single working set. Even with the cheap RAM available today, it is necessary to find a way to reduce the large volume of objects in the database. The Compressed Collection pattern allows a larger volume of data to fit into the working set.

Acknowledgments

The original idea for this manner of compressing the representation of collections comes from John Martin. Gary Sevitsky provided the design and implementation in C++ for time-based collections used in the NYNEX Network Exploratorium. The WICS-95 Design Patterns class provided initial critique of this pattern. Rick Garrison provided the Framemaker format template.

References

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