The Atomizer—
Efficiently Streaming Object Structures

Dirk Riehle (riehle@ubilab.ubs.ch)
UBILAB, Union Bank of Switzerland. Bahnhofstrasse 45, 8021 Zürich, Switzerland.

Wolf Siberski (wolf_siberski@rwg.e-mail.com)
RWG GmbH, Deutschland. Räpplenstraße 17, 70191 Stuttgart, Germany.

Dirk Bäumer (dirk_baeumer@rwg.e-mail.com)
RWG GmbH, Deutschland. Räpplenstraße 17, 70191 Stuttgart, Germany.

Daniel Megert (Daniel.Megert@ubs.com)
FKAM, Union Bank of Switzerland. Bahnhofstrasse 45, 8021 Zürich, Switzerland.

Heinz Züllighoven (Heinz.Zuellighoven@informatik.uni-hamburg.de)
Software Engineering Group, University of Hamburg. Vogt-Kölln-Strasse 30, 22527 Hamburg, Germany.

Abstract

The Atomizer pattern lets you efficiently stream objects into data structures of your choice as well as create objects from such data structures. The Atomizer pattern can be used whenever objects are written to or read from flat files, relational database tables, network transport buffers, etc.

The Reader part of the pattern builds an object structure by reading a data structure from a backend. The Writer part of the pattern writes an existing object structure as a data structure to a backend. Both parts together constitute the Atomizer pattern, which in turn is a specialization of the Visitor pattern. The Atomizer pattern itself keeps recurring as a pertinent design theme so that we chose to describe it as a pattern of its own.

The pattern can be found in more or less pure versions in probably every framework that provides support for object streaming. The CORBA externalization service represents are very clean implementation of the pattern. However, it develops its full potential only in the context of different streaming backends.

Intent

Read arbitrarily complex object structures from and write them to varying data structure based backends. The Atomizer pattern lets you efficiently store and retrieve objects from different backends, such as flat files, relational databases and RPC buffers.

Also Known As

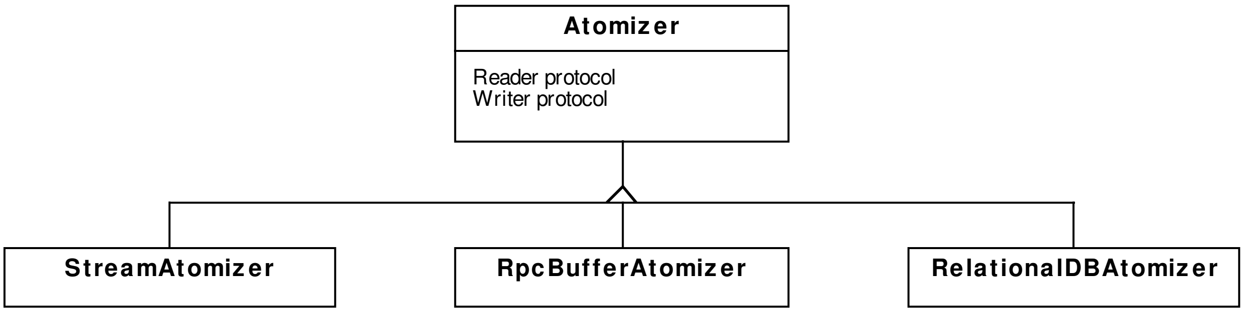
Stream, Reader/Writer

Motivation

Suppose you are modeling a Customer class in the banking domain. The Customer class will have several attributes, for example a name and a list of accounts, etc. You will want to make Customer and Account objects persistent, for example by storing them in a relational database. Sometimes you need to exchange customer data with other branch offices. This can be done by writing the objects to RPC buffers for transport via a network connection. Or, your employees often visit the customer at home, using a notebook computer when doing so. They need access to the customer data so that the objects have to be saved to a file on the notebook. Thus, every major application needs to read objects from and write them to a varying number of backends with different representation formats.

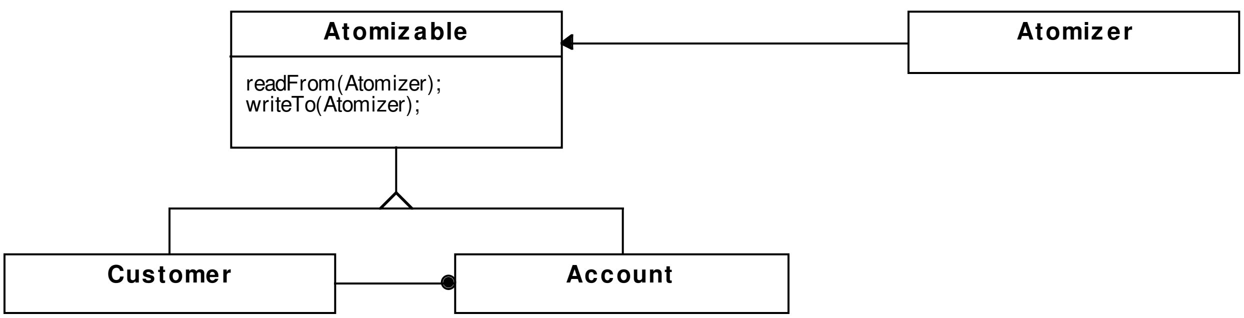
Application classes should have no knowledge about the external representation format which is used to represent their instances. Otherwise, introducing a new representation format or changing an old one requires to change almost every class in the whole system. These classes should contain no representation specific code for reading or writing their instances. It is much better to delegate the task of reading and writing to external and exchangeable classes which do the reading and writing, respectively.

Thus, we introduce the Atomizer class which decouples the application classes from a specific backend. The Atomizer interface is divided into two protocols, one for reading and one for writing. These are the Reader and Writer protocols of an Atomizer. The Reader protocol is used for reading (activating) object structures, and the Writer interface is used for writing (passivating) them. Different Atomizers represent different external representation formats and interact with different reading and writing backends. The following figure shows an exemplary hierarchy of Atomizer classes.



In turn, an Atomizer class shouldn't know the concrete application classes, because it would have to be modified whenever an application class is added or changed. To achieve this, the application classes have to provide a generic access interface to their internal state.

Therefore, every application class has to provide an interface called Atomizable which accepts an Atomizer object for reading and writing. Subclasses of Atomizable implement this interface by accepting Atomizer objects and by reading from or writing their attributes to them. The interplay between Atomizable and Atomizer conforms to the Visitor pattern (GOF, 1995) and will be explained in more detail below. The following figure applies the pattern to the example:



The Reader and Writer protocols offer every atomizable object the possibility to read or write basic value types, including object references. An Atomizer can follow object references to traverse a whole object structure and to either create it (Reader part) or write it (Writer part) to a specific backend.

Applying the Atomizer pattern lets you traverse object structures on an attribute level, and while doing so convert the object structure into any required external representation format. Application objects are freed from having to care about how to read from or write to external media so that it becomes easy to introduce new or change old input and output formats and backends.

Applicability

Use the Atomizer if

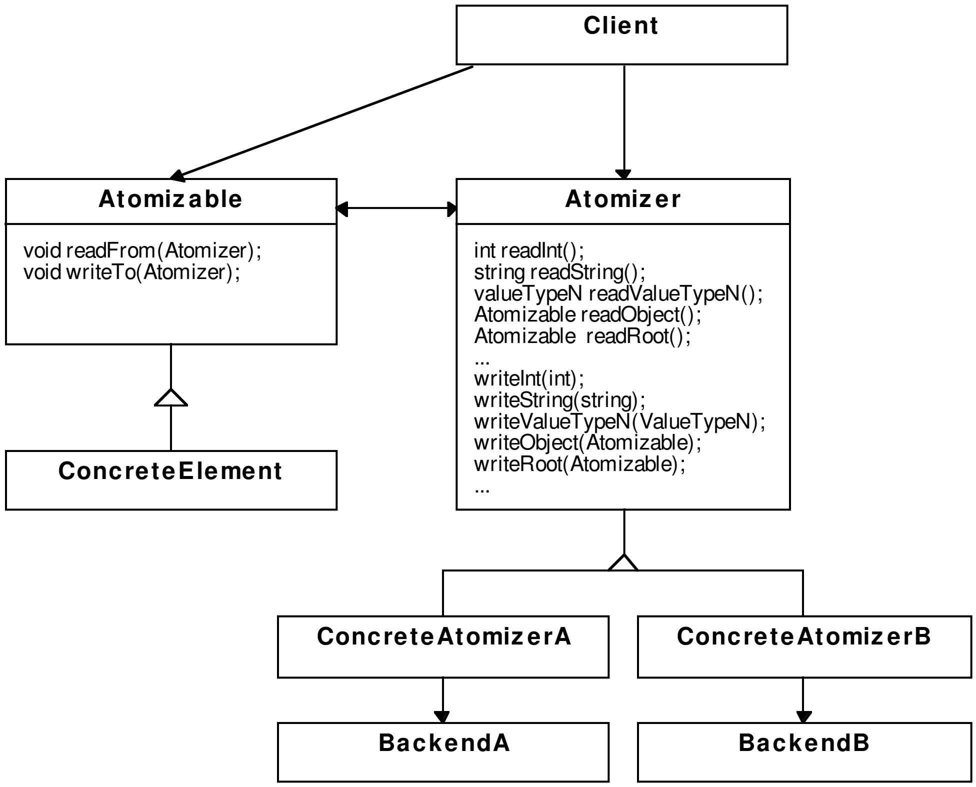
- you have to convert arbitrarily complex object structures into different data representation formats and back, and you don't want to put knowledge about the representation formats into the objects to be read or written.

The Atomizer is unnecessary overhead, if

- only one external data representation format will ever be needed.

The pattern cannot only be used to store objects in any kind of data stream like ordinary files or debugging dumps; it is also useful for storing them in relational databases or data buffers used for transporting objects between processes. An Atomizer can also be used as a Copier to copy an object structure; it can even be used to realize object browsers like the Smalltalk Inspector which display object structures at runtime.

Structure



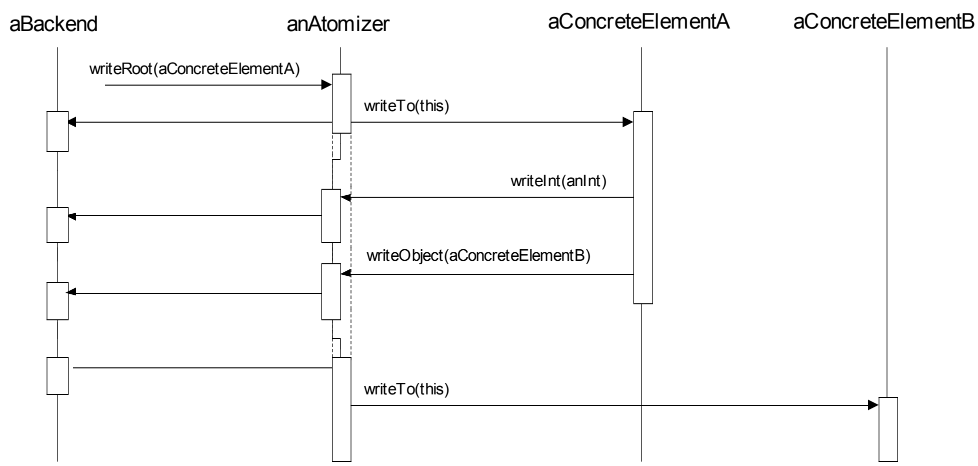
Participants

- **Atomizer**
  - declares a Reader protocol for reading objects and a Writer protocol for writing objects. These protocols consist of read respectively write operations for every value type, including object references;
  - hides the Backend and external representation format from atomizable objects.
- **ConcreteAtomizer**
  - implements the Reader and Writer protocols for a particular Backend and external representation format.
- **Atomizable**
  - is an abstract class which defines operations to accept an Atomizer for reading or writing.
  - provides a Create operation which takes a class id and creates an object of the denoted class.
- **ConcreteElement**
  - implements the Atomizable interface to read or write its attributes into an Atomizer.
- **Backend**
  - is a particular backend like a stream or a relational database frontend.
  - is used by the Atomizer which shields it from the application classes.

Collaborations

- The Atomizer collaborates with the Atomizable protocol class to read or write objects the classes of which are derived from Atomizable. The Atomizer hands itself over to the atomizable objects, while the atomizable objects make use of the Atomizer's Reader and Writer protocol to read and write their attributes. The reading and writing processes are nearly identical. They follow the Visitor pattern and thus results in a recursive back and forth interplay between atomizable objects and the Atomizer.
- During the writing process, each object writes its attributes by calling the appropriate write method of the Atomizer. The Atomizer handles attributes that are object references according to some predefined specification. If they are to be dereferenced, the Atomizer asks them to write them onto itself.
- When reading an object the Atomizer first creates a new instance of the correct class and then hands itself over to it. The new atomizable object reads its attributes by calling the appropriate read methods for each attribute. During the reading process the Atomizer creates all objects which are requested by already existing objects.
- An Atomizer reads from and writes to its backend using a backend specific interface.

The following interaction diagram shows an exemplary collaboration. The client calls *writeRoot* with *aConcreteElementA* which has the attributes *anInt* and *aConcreteElementB*. The dotted lines indicate that *anAtomizer* remains active while calling *writeTo*. The diagram only shows the start of the write process.



Consequences

Take the following consequences into account, when considering to apply the Atomizer pattern:

1. *Using the Atomizer makes adding new data representation formats for objects easy.* Object structures can be written to and read from new and unforeseen backends simply by introducing a new Atomizer. Often, it suffices to parameterize some standard Atomizer implementation with a storage backend, thereby easing the introduction of new data representation formats even more.
2. *Using the Atomizer takes knowledge about data representation formats out of the objects to be streamed.* By using the Atomizer interface of simple read and write operations, the objects are effectively shielded from any data format used for their external representation.
3. *Using the Atomizer pattern requires new classes to support the Atomizable protocol.* Classes which allow their objects to be streamed must inherit from Atomizable. The implementation of the Atomizable protocol requires reading and writing every relevant object attribute.
4. *Using the Atomizer patterns weakens encapsulation.* It is at the heart of the Atomizer pattern to allow the access to an object's implementation state. This weakens encapsulation to some extent. It is very inconvenient, however, to break encapsulation by misusing the Atomizable protocol.
5. *The set of value types supported by an Atomizer has to be considered well.* At first glance, one might consider supporting only object references and the programming language's "built-in" value types like integer, float, etc. However, for some types it is appropriate to treat them like built-in value types (for example, String or Date), and to add special methods to the Atomizer interface to handle them. This may be the case with general as well as domain specific value types.

Implementation

Consider the following issues when implementing an Atomizer:

- *Deciding between deep and non-deep streaming.* Deep streaming an object structure means streaming every referenced object. This is typically done when reading from a file or writing to it. With other kinds of backends, for example data bases, deep streaming is unsuitable so that you might choose a different streaming policy, for example a policy which streams only changed objects. Implementing non-deep streaming is more complex than implementing deep streaming. On the other hand, using deep streaming is potentially very costly since it might require transporting large amounts of data due to the highly interconnected nature of object structures.
- *Identifying objects.* Objects usually reference other objects. In a passive data format, these references must be represented by an unambiguous identification (references as main memory addresses are not suitable for this). Such an identification, an id, only has to be unambiguous within the type's name space, not for all types' name spaces, since the object's type will always be available. There are several possibilities to implement object identification schemes:
  - *Creating object ids on the fly.* When doing deep streaming it is sufficient to create them on the fly. This can be done easily by converting the main memory address into an integer when writing the object. When reading the object, the integer can be used as an unambiguous identification within the data structure for the denoted object.
  - *Using a global counter.* Many implementations use a global counter to create object ids. When an object is created it receives the counter value as an id with the counter being incremented. To avoid running out of ids in the long run, most implementations use 8 byte counters.
  - *Relying on externally generated ids.* Very often, particular backends offer id generation mechanisms, for example database systems. If possible, these facilities should be used.
- *Writing additional information.* The signatures of the Atomizer's read and write operations depend on the purposes you are using object streaming for. The minimal information that must be written is the value to be streamed. But then reading depends on the ordering with which the attributes are written. Therefore, consider writing more information about the attributes:
  - *Writing the attribute name.* It is advantageous also to write the attribute name of an attribute. This helps to make the read operations independent of the ordering with which the object's values were written. In addition, it provides valuable, sometimes necessary, hints to elaborate backends. For example, a relational database backend might have to interpret the data it receives in terms of the tables into which they are written. Then, it is necessary to identify each value by name.
  - *Distinguishing between transient and persistent data.* You might be tempted to write out and read back only the primary attributes of the object and omit the functionally dependent ones, because they can be reconstructed from the primary ones. Then, however, you are focusing the application of this pattern to object streaming only, and miss possible other uses like object browsers. You might therefore consider writing out all attributes and enhancing the (read and) write operations with a tag indicating whether the attribute is to be considered as transient or persistent.
  - *Writing a version number.* Each object might write a version number identifying the version of its implementation. This provides some (though minimalistic) support for evolution. Conversion functions in the Atomizer or the streamed object's implementation itself might then provide the functionality for backward and forward compatibility.
- *Providing an object manager.* When writing or reading, you need to keep track of objects which have already been read or written in order to avoid an endless loop when dealing with circular references. A possible solution is to introduce an object manager used for managing object tables that provide the needed information. The manager object can be asked whether an object of a certain id has already been read or written. It can also map object ids to object references.

  The object manager keeps track of objects on a global, system-wide level. It is to be distinguished from the object management facilities of an Atomizer which has to keep track of the objects read or written within a specific reading or writing process.
- *Implementing the reading and writing operations using a metaobject protocol.* Instead of extending the read and write operations in every Atomizable subclass, they might be implemented once, directly in Atomizable. To do so, a metaobject protocol is required which allows simple access to the an object's attributes.

Despite such a general solution, it is still a viable means to implement those operations by hand, because it lets you control the reading and writing process in more detail, for example by flagging certain attributes as transient or persistent. A general metalevel approach would require further specification mechanisms here which go beyond most standard programming languages.

- *Using a data buffer as a backend.* Sometimes, you will want to decouple concrete services representing backends from the Atomizer. You can do so by making the Atomizer work on a generic data buffer instead of a specific backend. This allows for a generic implementation of large parts of the Atomizer which simply stream the objects into the data buffer. The client can then provide specific backends with that buffer. Doing so, you effectively decouple the Atomizer from a specific backend and allow its generic implementation. Furthermore, it becomes easy to define context boundaries of when a read or write begins and ends.

- *Providing additional initialization operations.* If some additional object initialization has to be carried out after reading an object, consider providing an additional initialization operation. It is not advisable to do this while still reading the attributes for the same reasons you separate initialization procedures from the basic object creation procedure: Objects at this early point of initialization may only be half baked, and initializing functionally dependent attributes from a potentially inconsistent object state might cause unforeseen and unwanted side-effects.

- *Folding the read and write method pairs into single methods.* It is possible to halve the coding effort for atomizable classes by folding the Atomizable protocol into a single operation, for example named acceptAtomizer(Atomizer). The read and write operation pairs of the Atomizer's Reader and Writer protocol then have to be folded into a single atomize method for a specific value type. The acceptAtomizer implementations call the atomize methods with references to their attributes as parameters (instead of their actual values).

  If the Atomizer object is actually reading an object, its atomize methods will use the reference to replace the value of the referenced attribute. If the Atomizer object is actually writing an object, it will use the reference to retrieve and write the attribute. Rather than providing two different modes, a reading and a writing mode, you might consider implementing separate Reader and Writer classes. This approach works particularly well in C++.

- *Creating objects during the read process*. During the read you have to be able to create a new object, given its class id. For this task you should use a creational pattern, for example a Factory Method. The simplest solution is to make Atomizable provide a static function (in C++) or a class method (in Smalltalk) which receives the class name or class id which it maps on a class object or prototype using a dictionary or similar kind of mapping. A flexible pattern for this purpose is the "Late Creation" pattern described in (Bäumer and Riehle, 1996).

- *Taking care of diamond inheritance structures.* When streaming an object, care has to be taken that attributes are not streamed twice due to a diamond in the inheritance structure. If a class implements the writeTo operation generically based on meta-information, this problem can be easily avoided. If writeTo is implemented by every new subclass, either the Atomizer must check that no attributes are written twice, or the object itself must flag the attributes as already having been written. The last approach prevents that reaching a specific class' writeTo operation for a second time causes its attributes being written again.

- *Aborting the reading or writing process.* If an unsalvageable failure ocurrs, the system should abort the reading or writing process and rollback all changes that have been carried out. Conceptually, the reading or writing of an object structure should be a transaction, that is it provides its own execution context and only upon commit makes these changes visible to the environment. Implementing fail-safe object space transactions in a generic way is hard, though. Therefore, a specialized transaction manager for reading and writing should keep track of the objects and their embedding into the environment and be able to perform a rollback in case an exception ocurrs.

- *Treating Class variables separately.* Usually, class variables are read only at program or image initialization time, and they are written only at program finalization time. This does not pose problems, if the initialization and finalization is handled separate from the regular instance streaming process.

The following sections have only to be considered when doing incomplete streaming:

- *Selecting a streaming policy.* Object structures can become arbitrarily deep; when doing non-deep streaming you must make a decision to which extent you want to stream object structures. We distinguish between the following streaming policies (Bischofberger et al., 1996):

  - *Shallow streaming.* An object is streamed only to the first level of attributes. No references within the object are followed. This solution should be applied if nothing can be said about streaming requirements except that it can be very costly to ask for more than a shallow object.

  - *Fixed level streaming.* Streaming is performed to a predefined depth. Starting with a root object, every reference is followed until a nesting count reaches a predefined value. This is a general solution applicable if deep streaming is too costly but there is no information about the object structure which would make it possible to specify a better strategy.

- *Partial streaming.* In this case, streaming is performed according to some predefined graph specification which defines which object references are to be followed and which are to be left dangling, for example as proxies. This is the best solution since it lets developers map domain specific requirements on streaming behavior. An interesting treatment of this has been presented by Lopez (1996) who calls partial streaming "adaptive streaming," which unfortunately interferes with the naming of our next policy.

- *Adaptive streaming.* Adaptive streaming is a specialization of partial streaming. Instead of deriving a streaming specification from the business requirements, adaptive streaming derives a streaming specification dynamically from the actual client usage of an object structure. Starting out with shallow streaming, a streaming service starts to gather data about the frequency of streaming and dereferencing requests, and derives a dynamic partial streaming specification from this.

When writing an object structure, it makes sense only to write those objects which have actually changed since they last have been read or written. This is particularly appropriate when writing to databases. You might have to provide "dirty flags" or some other technique to indicate that an object has changed.

- *Handling dangling references.* When partially reading an object structure not all references will be resolved. The unresolved references are left dangling. There are two major ways of dealing with these dangling references: Proxies and replacing or modifying the reference interpretation mechanism of the runtime system.

  - *Using proxies.* A proxy is a substitute for a real object which is not fully available at hand for some reasons, see (GOF, 1995). A dangling reference can be realized as a proxy. It can be an object of the correct type but without initialized attributes, or it can be an object of a special proxy type. Both variants must be capable of catching operation calls and dispatching them to some reading facility for the real object before executing the real operation.

  - *Changing the reference interpretation mechanism.* Since references are values, you might replace or modify the runtime system or the compiler to interpret references in an enhanced way. Such an enhanced interpretation might include checking a flag in the reference value which indicates whether the reference points to a valid main memory object or not. If not, the value could further be interpreted to provide a database id or the like for the real object in question. Such a modification is almost always system dependent. It should be done only if proxies are considered unsuitable for reasons of performance.

Sample Code

We will now review the example from the motivation pattern. We start by describing the writing process, and afterwards turn to the reading process.

The general `Atomizable` offers an operation for accepting an Atomizer for reading, an operation for accepting an Atomizer for writing, and an operation for creating instances of its subclasses known only by the class name at runtime. The class interface looks like this:

```
class Atomizable
{
public:
   virtual void readFrom(Atomizer*) =0;
   virtual void writeTo(Atomizer*) const =0;
   static Atomizable* newByName(char*);
};
```

Classes to be streamed via an Atomizer must inherit from `Atomizable`, as discussed. This holds true for the `Customer` and `List` classes from the motivation section as well. Their interfaces might look like the following (we have omitted any behavior that is not relevant for the example):

```
class Customer : public Atomizable
{
public:
   virtual void readFrom(Atomizer*);
   virtual void writeTo(Atomizer*) const;
   ...
private:
      // implementation state
   String name;
   List<Account>* accounts;
```

```
};

template<class T> List : public Atomizable
{
public:
   virtual void readFrom(Atomizer*);
   virtual void writeTo(Atomizer*) const;
   ...
private:
      // implementation state
   long count;
   T* list; // C++ native array implementation
};
```

Both classes define an implementation state that must be considered for reading and writing to an Atomizer. To do so, both classes overwrite the `readFrom` and `writeTo` operations. These operations make use of the `Atomizer` class interface the Writer protocol part of which looks like:

```
class Atomizer
{
public:
      // primitive "built-in" value types
   virtual void writeChar(const char) =0;
   virtual void writeUChar(unsigned char) =0;
   virtual void writeCharPtr(const char*) =0;
   ...
      // non-primitive value types
   virtual void writeString(const String*) =0;
   virtual void writeString(const String&) =0;
   ...
      // references to objects
   virtual void writeObject(const Object*) =0;
   virtual void writeObject(const Object&) =0;
   virtual void writeRoot(const Object*) =0;
   virtual void writeRoot(const Object&) =0;
   ...
}
```

This interface offers operations to write all value types considered to be important, including all built-in value types like int and float, non-primitive value types like String, and finally object references. The operation `writeTo` of class `Customer` and `List` might now be implemented like this:

```
void Customer::writeTo(Atomizer* atomizer) const
{
      // simply write the two attributes
   atomizer->writeString("name", name);
   atomizer->writeObject("accounts", accounts);
}

template<class T> void List::writeTo(Atomizer* atomizer) const
{
      // first write the count attribute
   atomizer->writeLong("count", count);
      // then write the array as a succession of object references
   for(long i=0; i<count; i++) {
      atomizer->writeObject("list[" + String(i) + "]", list[i]);
   }
}
```

All value types can be written directly to a backend, using whatever physical representation seems suitable and fits the backend. Of interest, however, is the handling of object references. Writing them is simple (they just have to be converted into an id), but since they represent objects it must be decided whether to write the full object and not just the reference , and it must be kept track of which object have already been streamed.

Let's pick a concrete example: An `ASCIIStreamAtomizer` uses the standard iostream classes as the output medium for the basic value type representations. In addition, it uses ASCII based formatting to make the output both human and machine readable. Its interface looks like the Atomizer interface defined above, it only introduces some additional operations for receiving the input and output streams.

`ASCIIStreamAtomizer` uses an instance variable named `buffer` to hold the `ostream` instance to which the output data is written. Writing a primitive value like a long integer is simple. It looks like the following code:

```
void ASCIIStreamAtomizer::writeLong(const String& key, long value)
{
   buffer << indent() << "long " << key << " = " << value << endl;
}
```

Writing a general reference of type `Atomizable` is slightly more complicated. In addition to writing the reference in form of an id, the Atomizer has to make a note to check with the streaming policy whether to write the full object. It suffices to push the object possibly to be written out onto a stack and take care of it later. `wasHandled` is a list which collects all objects that have already been written out, and `toHandle` is a stack which receives all objects that still must be written. Both are attributes of the Atomizer.

```
void ASCIIStreamAtomizer::writeObject(const String& key, Object* object)
{
      // first write id for object reference
   buffer << typeid(object) << " " << key << " = " << object->objectId() << endl;
      // then check whether object itself has to be written
   if (!mustHandle(key, object)) return;
      // now check whether object was already handled
   if (!wasHandled->contains(object)) toHandle->push(object);
}
```

The operation `mustHandle` checks whether the given object has to be written or not. It can either be directly implemented by the Atomizer, or it can delegate its implementation to a Strategy object implementing more elaborate streaming policies. The information required by the streaming policy is the attribute path from the root object of the object graph to be streamed.

In case of a shallow streaming policy, `mustHandle` always returns false, in case of a deep streaming policy, it always returns true. We assume a deep streaming policy for the `ASCIIStreamAtomizer`.

The writing process is started by a client with a call to `writeRoot`. It contains the writing process' main loop. The client provides it with the object serving as the root for the object graph to be written.

```
void ASCIIStreamAtomizer::writeRoot(Atomizable* root)
{
   wasHandled->clear();
   toHandle->clear();

      // push first object to be written
   toHandle->push(root);

      // loop until no more objects are available
   while (!toHandle->isEmpty())
   {
        // pop this iteration's object
      Atomizable* object = toHandle->pop();
         // write type id, object id (=reference)
      buffer << typeid(object) << " " << object->objectId() << " = " << endl;
      buffer << indent() << "{" << endl;
         // note object as already handled
      wasHandled->append(object);
         // finally ask object to write its attributes into the atomizer
      object->writeTo(this);
         // some more delimiters and pretty printing
      buffer << indent() << "}" << endl << endl;
   }
}
```

We have seen all relevant aspects of the writing process now: A client first instantiates or reuses an existing Atomizer and hands over the root object of the object structure to be written using `writeRoot`. `writeRoot` calls `writeTo` on this root object and waits to receive the object's attributes. Some of these attributes are primitive value types which directly can be written to the output buffer. Some of these attributes are references to other objects. After writing the id representing the referenced object and checking with the streaming policy, the Atomizer pushes the reference on a stack to write the object behind the reference later.

During one iteration in the main loop, a single object's attributes are received by the Atomizer and written to the backend. In addition, the Atomizer checks whether the object references among the object's attributes require further treatment, that is they have to be streamed themselves. The loop continues unless all objects requiring treatment have been written.

The reading process is very similar to the writing process. All atomizable classes implement the `readFrom` operation to read their attributes from an Atomizer. The classes `Customer` and `List` implement it like this:

```
void Customer::readFrom(Atomizer* atomizer)
{
   name = atomizer->readString("name");
   accounts = atomizer->readObject("accounts");
}

template<class T> void List::readFrom(Atomizer* atomizer)
{
   count = atomizer->readLong("count");

   list = new T[count];
   for(long i=0; i<count; i++) {
      list[i] = atomizer->readObject("list[" + String(i) + "]");
   }
}
```

The Reader protocol part of the Atomizer interface simply mirrors the Writer protocol part. It consists of a long succession of read operations for all value types:

```
class Atomizer
{
public:
      // primitive "built-in" value types
   virtual char readChar() =0;
   virtual unsigned char readUChar() =0;
   virtual char* readCharPtr() =0;
   ...
      // non-primitive value types
   virtual String& readString() =0;
   ...
      // references to objects
   virtual Object* readObject() =0;
   virtual Object* readRoot() =0;
   ...
}
```

While writing the Atomizer can write attribute after attribute to the output buffer, however, while reading it has to read a full object as a block in advance, because the object to be instantiated might ask for its attributes in a random order. We could require atomizable object to always ask for their attributes in the same order in which they were written, but we prefer to avoid such ordering dependencies.

Attributes are simply maintained in a dictionary which maps the attribute names on pairs of strings representing the attribute's type and value. The `readLong` and `readObject` operations might now look like this:

```
long ASCIIStreamAtomizer::readLong(const char* key)
{
      // retrieve attribute "key", convert it to long and return it
   return attributes->at(key)->value().asLong();
}
```

```
Atomizable* ASCIIStreamAtomizer::readObject(const char* key)
{
   Atomizable* object = null;
      // interpret  value as long (representing ids)
   long id = attributes->at(key)->value().asLong();

      // check whether object was already instantiated
      // could be delegated to an object manager (see impl. section)
   if (!wasHandled->containsKey(id))
   {
      String& type = attributes->at(key)->type();
         // create new object of given type to be returned
      object = Atomizable::newByName(type);
         // note as already handled
      wasHandled->putAt(id, object);
   }
      // return old object
   else object = wasHandled->at(id);

   return object;
}
```

`readObject` first checks whether the object indicated by the id already exists, and, if not, creates it using the accompanying type information. All the relevant attribute information about an object, the values and their types, is maintained in a dictionary named `attributes`. `attributes` is built in `initAttributes` which is called by `readRoot` of `ASCIIStreamAtomizer` every time before the call to `readFrom`.

In analogy to `writeRoot`, `readRoot` represents the main reading loop. It is called by a client to initiate the reading process:

```
Atomizable* ASCIIStreamAtomizer::readRoot()
{
   Atomizable* root = null;
   wasHandled->clear();
   toHandle->clear();

      // loop until full stream is parsed
   while (!buffer.eof())
   {
      Atomizable* object = null;
      char type[32], equal[4], bracket[4], tmp[4];
      unsigned long id;

         // read type id and object id
      buffer >> type >> id >> equal >> bracket;
         // was object already created (but not initialized)?
      if (!wasHandled->containsKey(id))
      {
           // create object using type information
         object = Atomizable::newByName(type);
           // note as being created
         wasHandled->putAt(id, object);
           // note root object to be returned
         if (!root) root = object;
      }
         // get existing object
      else object = wasHandled->at(id);

         // read the object’s attributes en block
      initAttributes(object->getAttributeCount());
         // tell object to retrieve its attributes
      object->readFrom(this);
```

```
      buffer >> bracket >> tmp;
   }

   return root;
}
```

`initAttributes` simply reads a predefined number of values and puts them in the `attributes` dictionary.

```
void ASCIIStreamAtomizer::initAttributes(int no)
{
   attributes->clear();

   for ( int i = 0; i < no; i++ ) {
      char type[32], name[32], equal[4], value[32];
         // read attribute type id, attribute name and value
      buffer >> type >> name >> equal >> value;
         // put type/value pair into attributes dictionary
      attributes->putAt(name, StringPair(type, value));
   }
}
```

Now we have all pieces at hand to understand the reading process: A main loop (`readRoot`) reads from a buffer until it reaches its end. This is identical to a full streaming policy, assuming that the buffer contains a complete object graph. `readRoot` creates the object to be read, reads its attributes in advance using `initAttributes`, and calls `readFrom` on the atomizable object to be read. The object requests its attributes from the Atomizer which satisfies these requests by returning the values from the `attributes` dictionary. Attributes which are references are instantiated as shallow objects, that is without initializing their attributes. This is delayed until the object itself turns up in the stream.

We have now discussed how an Atomizer interacts with a stream based backend. Due to the complexity of different backends, for example a relational database, this interaction might become much more complicated. This was not discussed here, simply because it doesn't represent the essence of the Atomizer pattern. The Atomizer pattern demonstrates how objects can be assembled and disassembled in a canonical way in order to convert their representation and read or write them to varying backends.

Known Uses

Object streaming is supported by almost every mature (application) framework like ET++, InterViews/Unidraw, and Smalltalk. We take a look at ET++'s implementation of object streaming (Weinand and Gamma, 1994). It is realized by an interplay between Object and Stream of which two subclasses IStream and OStream exist (for reading and writing respectively). The stream classes only handle the "built-in" value types, but no object references. This is done by Object, which provides the functionality for writing object structures and for reconstructing them. Many of the issues are similar to what has been discussed in the implementation section.

The main difference between the ET++ implementation and Atomizer implementations is that ET++ only handles one output format for object streaming. That's why it doesn't bother to take this out of class Object as the Atomizer pattern suggests. However, for meta-level access to an object's data members, a different operation called AccessMembers is defined. The object streaming functionality could have been based on a rewritten and enhanced AccessMembers operation.

The CORBA externalization service (OMG, 1996) also applies the Atomizer pattern: A stream service (either Stream or StreamIO) takes over the combined role of Reader and Writer, and a Streamable interface represents our protocol class Atomizable. An object or a set of objects is streamed between a begin_context() and an end_context() call to Stream. All references within these bracketing calls are resolved without duplicating objects. When creating an object structure, the stream service uses the readonly attribute Key of every Streamable object: It serves as the specification to retrieve a new object from a Factory looked up via a FactoryFinder. CORBA distinguishes between write_object() and write_graph() operations: write_object() only writes the referenced object, while write_graph() writes out a full object graph specified via the CORBA relationship service.

Riggs et al. (1996) describe "Object Pickling in the Java System" which corresponds to the Atomizer pattern. The Atomizable interface is separated into two Java interfaces, one for reading (ObjectInput), one for writing (ObjectOutput). The Atomizer is separated into two distinct classes, a Reader class (ObjectInputStream) and a Writer class (ObjectOutputStream). Standard backends for ObjectInputStream and ObjectOutputStream are the Java library classes InputStream and OutputStream. Writing an object always performs a deep streaming, only via specials is it possible to do a shallow streaming as required, for example, for remote procedure calls. Clients can put more than one object graph into an ObjectOutputStream; they indicate the end of a section by calling flush() on the OutputStream. Riggs et al. provide a meta-information based implementation of the readFrom and writeTo operations. This works well, because the Java runtime meta-information not only allows access to an object's field but also provides information whether a field is to be considered as transient or persistent, and what the field's name is.

The GeBOS series of banking projects developed at RWG in Stuttgart, Germany, uses the Atomizer pattern to read and write arbitrary object structures from flat files and relational databases. Different formats for flat files like electronic logs, debugging dumps, etc. are supported (Bäumer et al., 1996).

The GBO specification for a reflexive distributed object-oriented software architecture currently developed by Genesis Development Corp. and UBILAB for the Union Bank of Switzerland uses the Atomizer pattern to do arbitrary object streaming, for example to and from backends like flat files, databases, and RPC buffers (Bischofberger et al., 1996).

The Beyond-Sniff project pursued at UBILAB is a distributed software development environment which uses the Atomizer pattern to store and transfer arbitrarily large data structures, for example for retrieval results from a symbol-table to a client's programming environment instantiation (Mätzel and Bischofberger, 1996).

Parrington uses the Atomizer pattern in the context of the distributed programming system Arjuna to marshal and unmarshal the parameters of remote procedure calls (Parrington, 1995).

Related Patterns

Both Reader and Writer are specializations of the Visitor pattern (GOF, 1995). The create operation of the protocol class Atomizable is best implemented by Late Creation (Bäumer and Riehle, 1996). Alternatively, as suggested in (GOF, 1995), page 111, it can be implemented using Factory Methods. The streaming policy used to decide whether to activate/passivate a certain object reference can be implemented as a Strategy.

Bibliography

Bäumer et al. (1996)
Dirk Bäumer, Rolf Knoll, Guido Gryczan and Heinz Züllighoven. "Large Scale Object-Oriented Software-Development in a Banking Environment—An Experience Report." LNCS 10??, ECOOP '96, *Conference Proceedings.*

Bäumer and Riehle (1995)
Dirk Bäumer and Dirk Riehle. "Late Creation—A Creational Pattern." *Submitted to PLoP '96.*

Bischofberger et al. (1996)
Walter Bischofberger, Michael Guttman, and Dirk Riehle. "Architecture Support for Global Business Objects: Requirements and Solutions." UBILAB '96, *Conference Proceedings.* To appear.

Gang-of-four (1995)
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. (1995). *Design Patterns: Elements of Reusable Design*. Reading, Massachusetts: Addison-Wesley.

Lopez (1996)
Cristina Videira Lopez. "Adaptive Parameter Passing." LNCS 1049, ISOTAS '96, *Conference Proceedings.* Edited by Kokichi Futatsugi and Satoshi Matsuoka. Berlin, Heidelberg: Springer-Verlag, 1996. 118-136.

Mätzel and Bischofberger (1996)
Kai-Uwe Mätzel and Walter R. Bischofberger. "The Any Framework: A Pragmatic Approach to Flexibility." COOTS '96, *Conference Proceedings.* 179-190.

OMG (1996)
Object Management Group, Inc. *CORBAservices: Common Object Services Specification, volume 1.* Revised edition March 31, 1995. Updated March 28, 1996.

Parrington (1995)
Graham D. Parrington. "A Stub Generation System for C++." *Computing Systems* 8, 2 (Spring 1995): 135-167.

Riggs et al. (1996)
Roger Riggs, Jim Waldo, Ann Wolrath and Krishna Bharat. "Pickling State in the Java System." COOTS-2, *Conference Proceedings.* Pages 241-250.

Weinand and Gamma (1994)
André Weinand and Erich Gamma. "ET++ — a Portable, Homogenous Class Library and Application Framework." *Computer Science Research at UBILAB*. Edited by Walter R. Bischofberger and Hans-Peter Frei. Konstanz: Universitätsverlag Konstanz, 1994. 66-92.