Serializer

Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert and Heinz Züllighoven

ABSTRACT[1]

The Serializer pattern lets you efficiently stream objects into data structures of your choice as well as create objects from such data structures. The Serializer 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 Serializer pattern.

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 and the JAVA Serialization package are a clean applications 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 Serializer pattern lets you efficiently store and retrieve objects from different backends, such as flat files, relational databases and RPC buffers.

ALSO KNOWN AS

Atomizer, Streamer, 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. 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, bank representatives visit the customer at home, using a notebook computer when doing so. They need access to the customer data. Therefore 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.

---

[1] In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle and Frank Buschmann. Reading, MA: Addison-Wesley, 1997. Chapter 17.

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 would require 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.

To seperate reponsibilities for reading and writing we introduce a Reader/Writer class pair for each backend. These classes decouple the application classes them from the backends. The Reader protocol is used for reading (activating) object structures, and the Writer protocol is used for writing (passivating) them. Different Reader/Writer pairs represent different external representation formats and interact with different reading and writing backends. Figure 1 shows an example of a Reader/Writer class hierarchy.

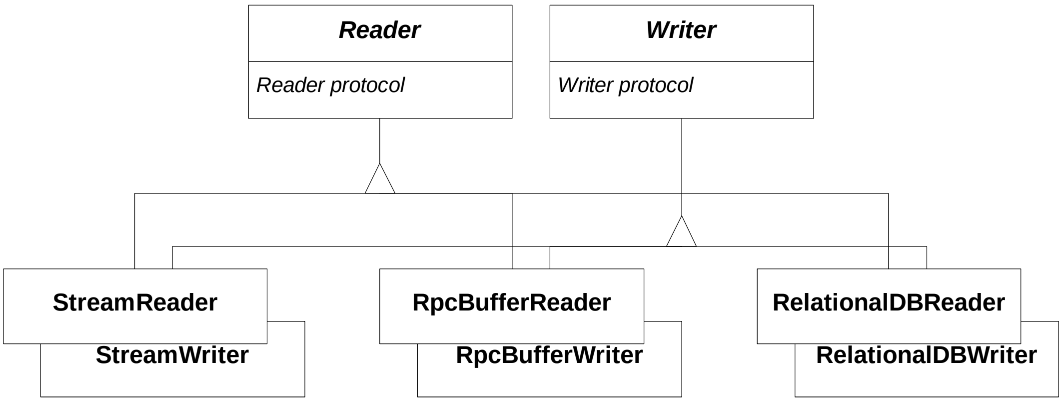

Figure 1: Example of a Reader/Writer class hierarchy

In turn, the Reader and Writer classes shouldn’t know the concrete application classes, because they 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 provides an interface called Serializable. This interface consists of two methods, one for reading and one for writing the object. The readFrom method accepts a Reader object for reading, and the writeTo method accepts a Writer for writing. Subclasses of Serializable implement this interface by accepting Reader/Writer objects and by reading from or writing their attributes to them. In Figure 2 the pattern is applied to the example.

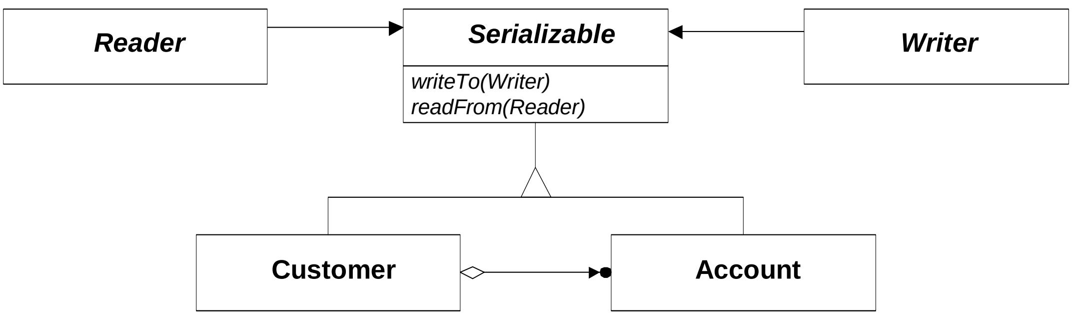

Figure 2: the Serializable interface

The Reader and Writer protocols offer every serializable object the possibility to read or write primitive value types, including object references. A Reader or a Writer can follow object references to traverse a whole object structure and to either create it rsp. write it to a specific backend.

Applying the Serializer 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 Serializer 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.

Don't use the Serializer, if

- the application objects have to provide backend specific information to the format conversion algorithm.

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 that are used for transporting objects between processes. An Serializer can also be used as a Copier to copy an object structure; it is even useful for building an object browser like the Smalltalk Inspector which displays objects at runtime.

STRUCTURE

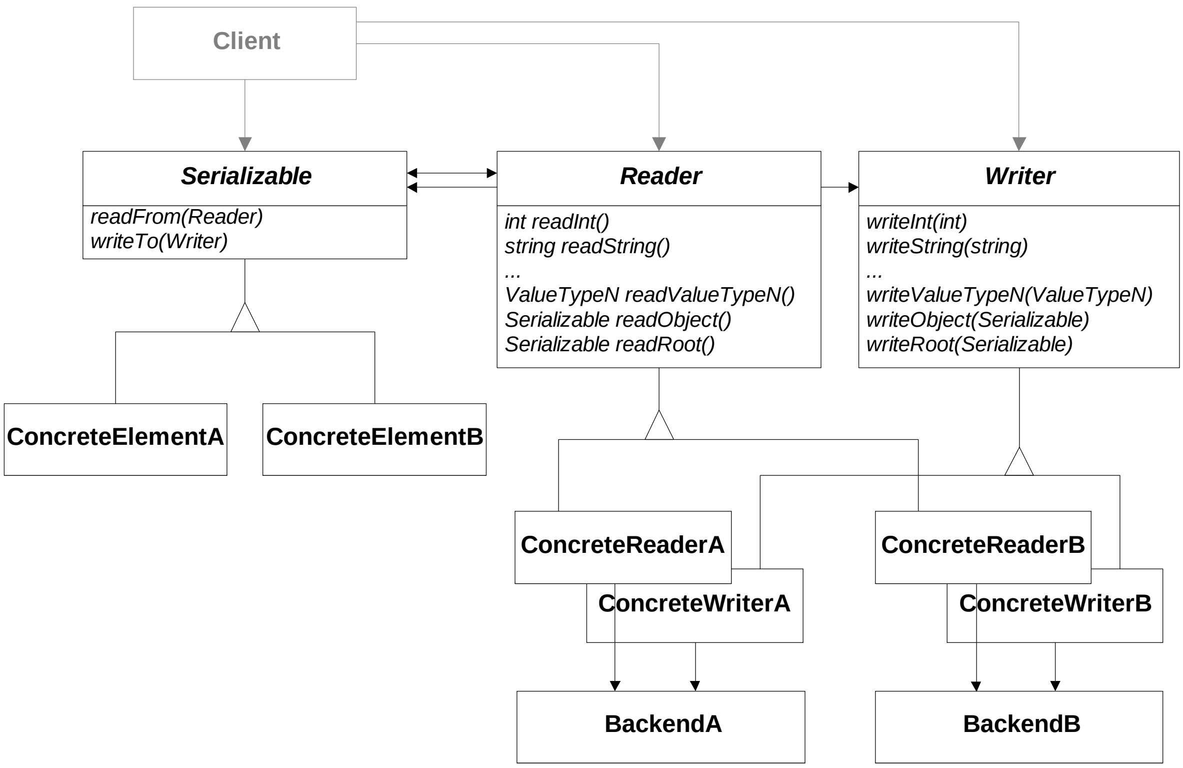

Figure 3: Structure of the Serializer pattern

PARTICIPANTS

- **Reader/Writer**
  - declare 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 serializable objects.
- **ConcreteReader/ConcreteWriter (StreamReader/Writer, RpcBufferReader/Writer)**
  - implement the Reader and Writer protocols for a particular Backend and external representation format.
- **Serializable**
  - is an interface class which defines operations to accept a Reader for reading and a Writer for writing. These operations have to provide the attributes to the Reader/Writer.
  - provides a Create operation which takes a class id and creates an object of the denoted class.
- **ConcreteElement (Customer, Account)**
  - implements the Serializable interface to read or write its attributes.
- **Backend (Stream, RpcBuffer)**
  - is a particular backend like a stream or a relational database frontend.
  - is used by the ConcreteReader/ConcreteWriter which shields it from the application classes.
  - the backend has not to be encapsulated in a class; its interface may also be procedural.

COLLABORATIONS

A Reader rsp. Writer collaborates with the Serializable protocol class to read rsp. write serializable objects. The Reader/Writer hands itself over to the serializable objects, while the serializable objects make use of its protocol to read rsp. write their attributes. The reading and writing processes are nearly identical. They result in a recursive back and forth interplay between serializable objects and the Reader/Writer.

During the writing process, each object writes its attributes by calling the appropriate write method of the Writer. The Writer handles attributes that are object references according to some predefined specification (see discussion on streaming policies in the implementation section). If the referenced objects are to be written, the Writer asks them to write them onto itself.

When reading an object the Reader first creates a new instance of the appropriate class and then hands itself over to it. The new serializable object reads its attributes by calling the respective read methods for each attribute. During the reading process the Reader creates all objects which are requested by already existing objects.

A ConcreteReader rsp. ConcreteWriter reads from rsp. writes to its backend using a backend specific interface which need not be object-oriented.

Figure 4 is an interaction diagram for a sample collaboration. The client calls *writeRoot* with *aCustomer* which has the attributes *name* and *accounts*. The dotted lines indicate that *aConcreteWriter* remains active while calling *writeTo*. The diagram only shows the start of the write process:

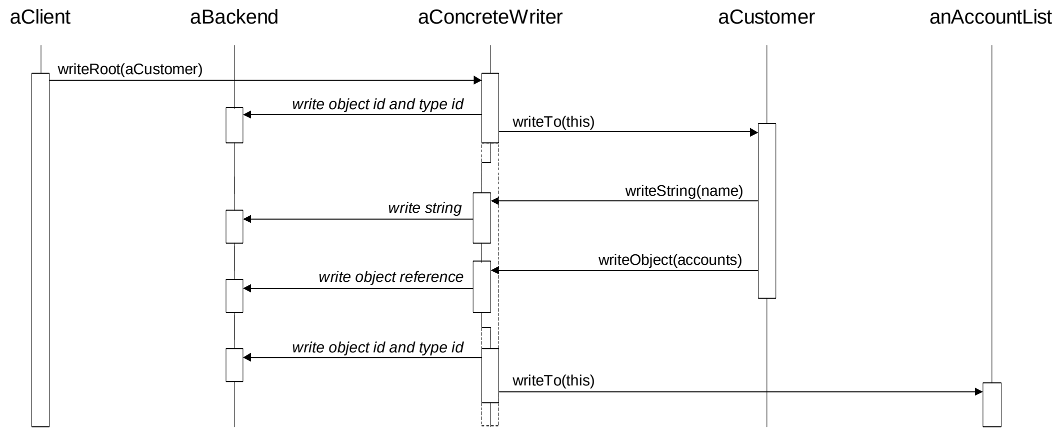

Figure 4: interaction diagram for a sample write process

CONSEQUENCES

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

1. *Using the Serializer 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 Reader/Writer pair. Often, it suffices to parameterize some standard Reader/Writer implementation with a storage backend, thereby easing the introduction of new data representation formats even more.
2. *Using the Serializer takes knowledge about external data representation formats out of the objects to be streamed.* By using the Reader/Writer interface of simple read and write operations, the objects are effectively shielded from any data format of their external representation.
3. *Using the Serializer pattern requires new classes to support the Serializable protocol.* Classes of streamable objects must implement the Serializable protocol. This requires reading and writing every relevant object attribute.
4. *Using the Serializer patterns weakens encapsulation.* It is at the heart of the Serializer pattern to allow the access to an object's internal state. This weakens encapsulation to some extent. It is very inconvenient, however, to break encapsulation by misusing the Serializable protocol.
5. *The set of value types supported by the Readers/Writers 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 types), and to add special methods to the Reader and Writer 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 a Serializer:

- *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 databases, deep streaming is unsuitable so that you will 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. But 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. Such an identification, an id, only has to be unambiguous within the type's name space, not necessarily for all types' name spaces, since the object's type is always stored together with its id. There are several possibilities to implement object identification schemes:

  – *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 eventually, most implementations we know of use 8 byte counters.

  – *Relying on externally generated ids.* Very often, specific backends offer id generation mechanisms, for example database systems. If possible, these facilities should be used.

- *Writing additional information.* The signatures of the Reader's and Writer's 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 sequence of the written attributes. Therefore, consider writing more information about the attributes:

  – *Writing the attribute name.* It is advantageous also to write the attribute name. This makes the read operations independent of the sequence of the written object attributes. Some backends need the attribute name to write rsp. read the attributes correctly. For example, a relational database backend might have to interpret the data it receives in terms of the column names into which they are written. Then, it is necessary to associate attribute names with the corresponding column names.

  – *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 on 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 an attribute is 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 Reader or the streamed object's implementation itself might then provide the functionality for backward 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 Serializer 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.* If your runtime system provides a metaobject protocol which allows access to an object's attributes, it is possible to implement the read and write operations once, directly in Serializable. The disadvantage is that you usually can't mark attributes as transient or persistent anymore. An interesting exception is Java, which provides both field names and transient or persistent flags, so that the readFrom and writeTo operations can fully be written on a meta-level.
- *Using a data buffer as a backend.* Sometimes, you will want to decouple concrete services representing backends from the Serializer. You can do so by making the Serializer work on a generic data buffer instead of a specific backend. This allows for a generic implementation of large parts of the Serializer 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 Serializer 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 (e.g. initialization of non-persistent attributes), 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 serializable classes by folding the Serializable protocol into a single operation, for example named attributes(). The Reader and Writer protocol is also folded into one interface Serializer with the operations serializeInt, serializeString, etc. The attributes() implementations call the serializeXXX methods with references to their attributes as parameters (instead of their actual values).

  If the Serializer object is actually a concrete reader, its serialize methods will use the reference to replace the value of the referenced attribute. If the Serializer object is a writer, it will use the reference to retrieve and write the attribute. The client chooses whether to read or to write by giving the serializable object to the respective Serializer. This approach works particularly well in C++.
- *Creating objects during the read process*. During the read process 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 Serializable 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 "Product Trader" pattern described in [Bäumer+96].
- *Taking care of diamond inheritance structures.* When writing an object, care has to be taken that attributes are not written twice due to a diamond in the inheritance structure. Either the Writer must check that no attributes are written twice, or the object itself must

flag the attributes as already been written. The last approach prevents that reaching a specific class' writeTo operation for a second time causes its attributes being written again. The same applies to the reading process.

- *Aborting the reading or writing process.* If an unsalvageable failure occurs, 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 occurs.
- *Treating class attributes separately.* Class attributes should be read only at program or image initialization time, and written only at program finalization time. Instead of reading or writing them in the readFrom and writeTo operations, their initialization and finalization should be handled separately.

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+96]:
  - *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 [Lopes96] 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 for some reason, see [GOF95]. 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.* 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 section. First, we will describe the writing process, and then the reading process.

The general class `Serializable` offers an operation for accepting a Reader for reading, an operation for accepting a Writer 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 Serializable
{
public:
  virtual void readFrom(Reader*) =0;
  virtual void writeTo(Writer*) const =0;
  static Serializable* newByName(char*);
};
```

Classes to be streamed via a Reader or Writer must inherit from `Serializable`, as discussed. This holds true for the `Customer` and `List<Account>` classes from the motivation section as well. Their interfaces might look as follows:

```
class Customer : public Serializable
{
public:
  virtual void readFrom(Reader*);
  virtual void writeTo(Writer*) const;
  ...
private:
    // attributes
  string name;
  List<Account*>* accounts;
};

class Account : public Serializable
{
  ... // like Customer
};

template<class T> List : public Serializable
{
public:
  virtual void readFrom(Reader*);
  virtual void writeTo(Writer*) 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 do so, both classes overwrite the `readFrom` and `writeTo` operations. These operations make use of the Writer interface which looks like:

```
class Writer
{
public:
    // primitive "built-in" value types
  virtual void writeChar(const string& name, char value) =0;
  virtual void writeInt(const string& name, int value) =0;
  ...
    // non-primitive value types
  virtual void writeString(const string& name, const string& value) =0;
  ...
    // references to objects
  virtual void writeObject(const string& name, const Serializable*) =0;
  virtual void writeRoot(const Serializable*) =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(Writer* writer) const
{
    // simply write the two attributes
  writer->writeString("name", name);
  writer->writeObject("accounts", accounts);
}

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

The Writer can write all value types 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 the Writer must decide whether to write the full object and not just the reference and must keep track of which objects already have been streamed.

Let’s pick a concrete example: An `ASCIIStreamWriter` 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 Writer interface defined above, it only introduces some additional operations for receiving the input and output streams. We assume a deep streaming policy.

`ASCIIStreamWriter` 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:

```
void ASCIIStreamWriter::writeLong(const string& name, long value)
{
  buffer << "long " << name << " = " << value << endl;
}
```

Writing a general reference of type `Serializable` is slightly more complicated. First, the Writer writes the object id to the buffer. Then it has to check whether the referenced object is already written. If not, the Writer pushes it on a stack and writes it later. `wasHandled` is a list which collects all objects that have already been written out, and `toHandle` is the stack which receives all objects that still must be written. Both are attributes of the Writer.

```
void ASCIIStreamWriter::writeObject(const string& name, Object* object)
{
    // first write id for object reference
  buffer << typeid(object) << " " << name << " = ";
  buffer << object->objectId() << endl;
    // check whether object was already handled
  if (!wasHandled->contains(object))
    toHandle->push(object);
}
```

The writing process is started by a client with a call to `writeRoot` with the object serving as the root as parameter. `writeRoot` contains the main loop which is continued until all referenced objects are written. During one iteration in the main loop, a single object's attributes are received by the Writer and written to the backend.

```
void ASCIIStreamWriter::writeRoot(Serializable* root)
{
  wasHandled->clear();
  toHandle->clear();

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

    // loop until all referenced objects are written
  while (!toHandle->isEmpty())
  {
      // pop this iteration's object
    Serializable* object = toHandle->pop();
      // write type id and object id
    buffer << typeid(object) << " " << object->objectId() << " = " << endl;
    buffer << "{" << endl;
      // note object as already handled
    wasHandled->append(object);
      // finally ask object to write its attributes into the Writer
    object->writeTo(this);
      // some more delimiters and pretty printing
    buffer << "}" << endl << endl;
  }
}
```

We have seen all relevant aspects of the writing process now: A client first instantiates or reuses an existing Writer 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, the Writer pushes the reference on a stack to write the object later.

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

```
void Customer::readFrom(Reader* reader)
{
  name = reader->readString("name");
  accounts = (List<Account*>*) reader->readObject("accounts");
}

template<class T> void List::readFrom(Reader* reader)
{
  count = reader->readLong("count");
```

```
  list = new T[count];
  for(long i=0; i<count; i++) {
    list[i] = (Account*) reader->readObject("list[" + string(i) + "]");
  }
}
```

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

```
class Reader
{
public:
    // primitive "built-in" value types
  virtual char readChar() =0;
  virtual int readInt() =0;
  ...
    // non-primitive value types
  virtual string readString() =0;
  ...
    // references to objects
  virtual Object* readObject() =0;
  virtual Object* readRoot() =0;
  ...
}
```

While writing a Writer can write attribute after attribute to the output buffer; a Reader, however, has to read all attributes in advance, because the object to be instantiated might ask for its attributes in a different order in which they were written. We could require serializable objects 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 maintained in a dictionary which maps the attribute names on pairs of strings representing the attribute's type and value. The `readChar` operation look like this:

```
char ASCIIStreamReader::readChar(const string& name)
{
    // retrieve attribute with key name, convert it to char and return it
  return attributes->at(name)->value().asChar();
}
```

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

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

    // check whether object was already instantiated
  if (!wasHandled->containsKey(id))
  {
    string type = attributes->at(key)->type();
      // create new object of given type to be returned
    object = Serializable::newByName(type);
      // note as already handled
    wasHandled->putAt(id, object);
  }
    // return old object
  else object = wasHandled->at(id);

  return object;
}
```

In analogy to `writeRoot`, `readRoot` implements the main reading loop. It is called by a client to initiate the reading process. The loop continues until the end of the buffer is reached. `readRoot` returns the first object which was read:

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

    // loop until entire stream is parsed
  while (!buffer.eof())
  {
    Serializable* 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 = Serializable::newByName(type);
        // note as being created
      wasHandled->putAt(id, object);
        // the first object is the root object
      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 ASCIIStreamReader::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 reads from a buffer until it reaches its end. This is identical to a deep streaming policy, assuming that the buffer contains a complete object graph. `readRoot` creates the object to be read, then reads all its attributes using `initAttributes`, and at last calls `readFrom` on the serializable object. The object requests its attributes from the Reader 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.

KNOWN USES

Object streaming is supported by almost every mature (application) framework such as ET++, InterViews/Unidraw, and Smalltalk. Let’s take a look at ET++’s implementation of object streaming [Weinand+94]. It is realized by an interplay between Object and Stream of which two subclasses IStream and OStream exist (for reading and writing respectively). The main difference between the ET++ implementation and Serializer 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 Serializer 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.

Although strictly speaking the CORBA externalization service [OMG96] is only a specification, it uses the same interface and separation of responsibilities as the Serializer 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 Serializable. 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. describe "Object Pickling in the Java System" which corresponds to the Serializer pattern [Riggs+96]. The interface Serializable, in contrast to our definition, has no operations but serves as an indicator of serializability to a Reader/Writer only. Concrete objects may implement operations writeObject and readObject if they wish to specialized the default implementation. The Serializer is separated into two distinct interfaces, a Reader interface (ObjectInput) and a Writer interface (ObjectOutput). Standard implementations for ObjectInput and ObjectOutput are the Java library classes ObjectInputStream and ObjectOutputStream. 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 opera-tions. 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 Serializer 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+96].

The Geo project pursued at Ubilab uses the Serializer for network transport of objects, copying, object inspection, and file streaming. A number of different Reader and Writer classes implement the Reader and Writer interfaces [Riehle+97].

The Beyond-Sniff project pursued at Ubilab is a distributed software development environment which uses the Serializer 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+96].

Parrington uses the Serializer pattern in the context of the distributed programming system Arjuna to marshal and unmarshal the parameters of remote procedure calls [Parrington95].

RELATED PATTERNS

The newByName operation of the protocol class Serializable is best implemented by using the Product Trader pattern [Bäumer+97]. Alternatively, as suggested in [GOF95], 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 [GOF95].

ACKNOWLEDGMENTS

Our Shepherd John Vlissides provided valuable feedback and helped us improve the paper significantly. We received very helpful comments from the members of the writers workshop “Distribution” at PLOP ’96. Ingrid Dörre and Frank Schneider shared their experience in implementing an Serializer subsystem with us and reviewed drafts of the paper. Erich Gamma gave us important hints, especially on the type safety advantage of the Reader/Writer pair compared to a single class Serializer which contains both protocols.

BIBLIOGRAPHY

[Bäumer+96] Dirk Bäumer, Rolf Knoll, Guido Gryczan and Heinz Züllighoven. “Large Scale Object-Oriented Software-Development in a Banking Environment—An Experience Report.” LNCS 1098, ECOOP ’96, *Conference Proceedings,* pp. 73-91.

[Bäumer+97] Dirk Bäumer and Dirk Riehle. “Product Trader ”. *This volume*.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. (1995). *Design Patterns: Elements of Reusable Design*. Reading, Massachusetts: Addison-Wesley.

[Lopes96] Cristina Videira Lopes. “Adaptive Parameter Passing.” LNCS 1049, ISOTAS ’96, *Conference Proceedings.* Edited by Kokichi Futatsugi and Satoshi Matsuoka. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 118-136.

[Mätzel+96] Kai-Uwe Mätzel and Walter R. Bischofberger. “The Any Framework: A Pragmatic Approach to Flexibility.” COOTS ’96, *Conference Proceedings,* pp. 179-190.

[OMG96] Object Management Group, Inc. *CORBAservices: Common Object Services Specification, volume 1*. Revised edition March 31, 1995. Updated March 28, 1996.

[Parrington95] Graham D. Parrington. “A Stub Generation System for C++.” *Computing Systems* 8, 2 (Spring 1995), pp. 135-167.

[Riggs+96] Roger Riggs, Jim Waldo, Ann Wolrath and Krishna Bharat. “Pickling State in the Java System.” COOTS-2, *Conference Proceedings,* pp. 241-250.

[Riehle+97] Dirk Riehle, Roger Brudermann and Walter Bischofberger. Request Handling in Geo. *Ubilab Technical Report 97.5.1*. Zurich, Switzerland: Union Bank of Switzerland, 1997.

[Weinand+94] 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, pp. 66-92.

Copyright 1997 by the authors.

Dirk Riehle works for Ubilab, the information technology research laboratory of Union Bank of Switzerland. He can be reached at Ubilab, Union Bank of Switzerland, Bahnhofstrasse 45, 8021 Zürich, Switzerland. He welcomes e-mail at Dirk.Riehle@ubs.com or riehle@acm.org.

Wolf Siberski works for RWG GmbH, Germany. He can be reached at RWG GmbH, Räpplenstraße 17, 70191 Stuttgart, Germany. He welcomes e-mail at daasb@rwg.de.

Dirk Bäumer works for RWG GmbH, Germany. He can be reached at RWG GmbH, Räpplenstraße 17, 70191 Stuttgart, Germany. He welcomes e-mail at Dirk_Baeumer@rwg.e-mail.com or baeumer@informatik.uni-hamburg.de.

Daniel Megert works at the information technology department for the corporate customer business of Union Bank of Switzerland. He can be reached at Union Bank of Switzerland, Bahnhofstrasse 45, 8021 Zürich, Switzerland. He welcomes e-mail at Daniel.Megert@ubs.com or Daniel.Megert@acm.org.

Heinz Züllighoven is a professor of computer science at University of Hamburg. He can be reached at Software Engineering Group, University of Hamburg, Vogt-Kölln-Strasse 30, 22527 Hamburg, Germany. He welcomes e-mail at Heinz.Zuellighoven@informatik.uni-hamburg.de.