On Persistent and Reliable Streaming in Ada

Jörg Kienzle

Software Engineering Laboratory
Swiss Federal Institute of Technology
CH - 1015 Lausanne Ecublens
Switzerland

email: Joerg.Kienzle@epfl.ch

Alexander Romanovsky

Department of Computing Science
University of Newcastle
NE1 7RU, Newcastle upon Tyne
United Kingdom

email: Alexander.Romanovsky@newcastle.ac.uk

Abstract. Saving internal program data for further use is one of the most useful ideas in programming. Developing general features to provide such data saving/ restoring is a very active research area. There are two application areas for such features we believe to be crucial: system fault tolerance and data persistence. Our analysis shows that the features used in these areas have a lot in common: they are to flatten data of different types and save them in a store which can be used later on. The recent revision of the Ada language standard, Ada 95, introduces a new mechanism called streams that allows structured data to be flattened. Streams are sequences of elements comprising values from possibly different types. Ada 95 allows programmers to develop their streams following the standard abstract class interface. In this paper we show how to use the stream concept for developing new features to provide internal program data saving suitable for fault tolerance and persistence. A hierarchy of different storage types, useful in different application domains, is introduced. The standard stream interface is extended, making it possible for programmers to have a better control of the way streams work by separating storage medium control from the actual stream type using the design patterns. The convenience of this new interface is demonstrated by developing a generic package allowing any non-limited object to be written into a storage device. It can be used for providing data persistence and as a state restoration feature in schemes used for tolerating software design faults.

Keywords. Streams, Persistence, Stable Storage, Design Patterns, Ada 95, Object-Oriented Programming, Fault Tolerance.

1 Introduction

Data are often kept in a secondary memory medium to be used in further program execution. It is not difficult to see that many modern services rely on saving data. Starting with databases and sequential files programmers have been trying to develop useful and general concepts in this area. How data are saved, what sort of API is provided, what assumptions are made (e.g. fault assumptions), etc., depends on the characteristics of the feature and on the application. In this paper we will concentrate on features which are used for saving and restoring values of internal program data. There are two main areas which require such features: developing fault tolerant systems and persistent systems.

Two general types of recovery are used in building fault tolerant systems [1]: forward and backward error recovery. When backward error recovery is used, the internal program data are saved in a memory which will not be affected by the faults assumed.

Later on, should an error be detected, the program is returned into a previous correct state by restoring its internal data. Depending on the fault assumptions and on the recovery scheme used, the program can be either re-started (if we are dealing with hardware crashes) or a diversely-designed program (alternate) can be tried (if a recovery block scheme [2] is used to tolerate software design faults). The former approach is often referred to as checkpointing. The features which are used for data saving and restoring in the latter are often called state restoration features.

Data persistence [3, 4] relies on saving values of data from a program execution space so that they can be used in a later execution: that is the values "persist" from one execution to another. There are many possible schemes for supporting persistence; for a complete survey, the reader is referred to [5].

In our opinion fault tolerance and persistence are quite distinct program properties and there are important differences in the way data saving is used in these two areas. Persistence relies on saving data values to allow them to be used in a later execution. In fault-tolerant systems the state of the whole program at some moments of time is saved and stored in such a way that the same program can continue execution from one of these states. This means in particular that these states must be consistent. Although very often the designers of the persistence services cannot help extending them to allow some simplified forms of error recovery, in our opinion it is important not to mix them and separate them properly while building, for example, persistence services for fault tolerant systems.

While implementing a persistence service, designers do not take into consideration fault assumptions as this is not relevant. But when we save data for fault tolerance, we should make sure that they will survive all assumed faults; this means for example that for the recovery block scheme we can use main memory for data saving if we assume only design faults. While developing error recovery features one should often take into account that errors can be detected at any time: depending on the failure assumptions, it may be the case that the program crashes and that it is not possible to do any data saving after an error has been detected. Sometimes it should be possible to tolerate media failures as well. When developing a persistence service one can basically assume that the program works/finishes correctly and it can perform all actions required for persistence any time it wants.

The definition of persistence is not specific about how the program finishes. This is why several persistence services have been extended to provide some forms of fault tolerance. Although, this is a reasonable approach for many practical reasons (e.g. performance), generally speaking, these two services can be provided separately and we believe that it is important to view them as such.

The remainder of this paper is organized as follows: the next section explains briefly how Ada 95 streams work. Section 3 discusses our reasons for choosing them to implement persistence and fault tolerance. In the following section a flexible, stream-based approach which implements these properties is described. It allows, in particular, the designers of stable storage to introduce new types of storage for keeping data. Section 5 shows by an example how the persistent and reliable streams are to be employed by the users. Section 6 looks at shared passive packages and the last section outlines our plans for future research.

2 Streams in Ada 95

Ada 95 [6], the recent revision of the Ada standard, does not have elaborate features for backward error recovery or data persistence. This is why many attempts have been made to extend the language, for example, a recovery block scheme in [7], and two approaches for persistent Ada [8, 9]. We believe that extending the language is for many reasons not practical; in this paper we rely on standard Ada 95 only.

Among many other new features, Ada 95 introduces a new concept called streams. A stream is a sequence of elements comprising values from possibly different types. The values stored in a stream can only be accessed sequentially. Ada streams can be seen as one of the first incarnations of the *Serializer* design pattern described in [10]. The CORBA *externalization* service [11] and the Java *Serialization* package [12] are other examples that implement the *Serializer* pattern.

This pattern allows programmers to efficiently stream objects into data structures of their choice, as well as create objects from such data structures. The pattern can be used whenever objects are written to or read from flat files, relational database tables, network transport buffers, etc. The participants of the pattern are: *Reader*/*Writer* and *ConcreteReader*/*ConcreteWriter*, the *Serializable* interface, *ConcreteElement* and *Backend*.

The *Reader* and *Writer* part declare protocols for reading and writing objects. These protocols consist of read respectively write operations for every value type, including composite types, array types and object references. The *Reader* and *Writer* hide the *Backend* and the external representation format from the serializable objects. *ConcreteReader* and *ConcreteWriter* implement the *Reader* and *Writer* protocols for a particular backend and external representation format. The *Serializable* interface defines operations that accept a *Reader* for reading and a *Writer* for writing. It also should provide a `Create` operation that takes a class identifier as an argument and creates an instance of the denoted class. *ConcreteElement* is an object implementing the *Serializable* interface, which allows it to read and write its attributes. The *Backend* is a particular backend, such as a storage device, a relational database front-end or a network buffer. A *ConcreteReader*/*ConcreteWriter* reads from/writes to its backend using a backend specific interface.

The structure of the *Serializer* pattern is shown in the following UML class diagram:

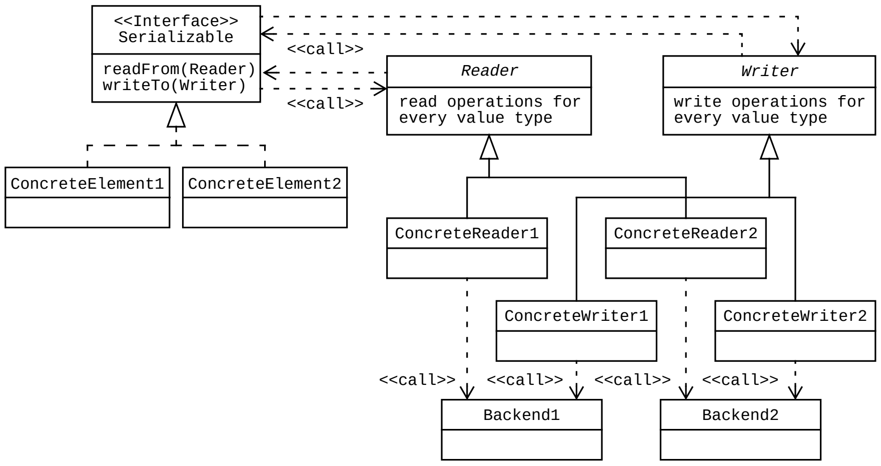

Figure 1: The *Serializer* Pattern Structure

When invoked by a client, a *Reader*/*Writer* hands itself over to the serializable object. The serializable object makes use of its protocol to read/write its attributes by calling the read/write operations provided by the *Reader*/*Writer*. This results in a recursive back-and-forth interplay between the two parties.

We will now show how Ada 95 streams implement the *Serializer* pattern. The standard package `Ada.Streams` defines the interface for streams in Ada 95 [6, 13.13.1]. It declares an abstract type `Root_Stream_Type`, from which all other stream types must derive.

Every concrete stream type must override the Read and Write operations, and may optionally define additional primitive subprograms according to the functionality of the particular stream. Obviously, the root stream type plays the *Reader*/*Writer* role in the *Serializer* pattern. Derivations of the root stream type incarnate the *ConcreteReader*/*ConcreteWriter* and the backend interface.

In Ada 95, the pre-defined attributes 'Write and 'Output are used to write values to a stream, thus converting them into a flat sequence of stream elements. Reconstructing the values from a stream is done with the pre-defined attributes 'Read and 'Input. They make dispatching calls on the Read and Write procedures of the Root_Stream_Type. When using 'Write and 'Read, neither array bounds nor tags of tagged types are written to or read from the stream. 'Output and 'Input must be used for that purpose.

All non-limited types have default implementations of the stream attributes, hence all non-limited types implement the *Serializable* interface and are therefore *Concrete Element*. It is possible to replace the default implementation of the stream attributes for any type via an attribute definition clause. In order to write a value of a limited type to a stream, such an attribute definition clause is even mandatory. Any procedure having one of the predefined signature shown in [6, 13.13.2] can replace the default implementation. The following example shows how to replace the predefined implementation of 'Write for an integer type:

```
type My_Integer is new Integer;

procedure My_Write  (Stream   : access Ada.Streams.Root_Stream_Type'Class;
                     Item     : My_Integer);

for My_Integer'Write use My_Write;
```

The only concrete stream implementation that is defined in the language standard is Stream_IO [6, A.12], a child package of Ada.Streams. It provides stream-based access to files. Stream_IO offers also file manipulation operations such as Create, Open, Close, Delete, etc. The following example shows how to write values of elementary types, array types and tagged types to a stream and how to reconstruct them again:

```
with Ada.Streams.Stream_IO; use
Ada.Streams.Stream_IO;

--  writing
declare
  My_File: File_Type;
  S : Stream_Access;
  I : Integer;
  My_String : String := “Hello”;
  T : A_Tagged_Type'Class := … ;
begin
  Create (My_File “file_name”);
  S := Stream (My_File);
  --  do some work
  Integer'Write (S, I);
  String'Output (S, My_String);
  A_Tagged_Type'Class'Output (S, T);
  Close (My_File);
end;
```

```
--  reading
declare
  My_File : File_Type;
  S : Stream_Access;
  I : Integer;
begin
  Open (My_File, “file_name”);
  S := Stream (My_File);
  Integer'Read (S, I);
  declare
    My_String: String :=
      String'Input (S);
    T : A_Tagged_Type :=
      A_Tagged_Type'Class'Input (S);
  begin
    --  do some work
  end;
  Close (My_File);
end;
```

3 Our Intentions

Ada streams are a very powerful and universal object-oriented mechanism; our intention is to use them for developing fault tolerance and persistence features. This fits exactly the underlying idea behind Ada streams, which is that programmers can develop their own stream subclasses by inheriting from the given abstract class. These streams can be suitable for different purposes, media, data, applications, assumptions, etc. To the best of our knowledge there has been no research reported along this line.

This approach has many advantages. It allows us to stay within the standard Ada language, which makes our approach useful for any settings and platforms which have standard Ada compilers and run-times. Although proposals discussing various Ada extensions are of great importance for the future language standards, there are useless from the point of view of practitioners designing systems now.

We perfectly realize that the features we intend to develop do not meet all requirements of the *orthogonal persistence* [4], but paper [9] clearly demonstrates that it is impossible to develop it within standard Ada 95. Our intention is to stay within the standard and develop data saving mechanisms as elaborate as Ada allows.

Although, as we have explained before, we treat backward error recovery and persistence as different properties, our analysis shows that a general approach suitable for both areas can be developed, as they share common demands. Our approach will incorporate a class hierarchy of different streams which are intended for saving data so that it can be used for both purposes.

Streams only develop their full potential in the context of different streaming backends such as flat files, relational database tables or network transport buffers. We have found that in spite of the fact that Ada streams are a very general and powerful concept, the `Ada.Streams` package does not well separate different forms of streams, e.g. buffered streams, from different streaming backends. This separation and the provision of additional backend control are vital for applying streams for developing backward error recovery and persistence features in Ada.

In the following part of the paper we will discuss a general extensible object-oriented data saving mechanism suitable for developing reliable and persistent systems. This mechanism will be flexible enough to allow transparent changes of the media and will rely on standard Ada features only.

4 Ada Streams Revisited

This section presents a flexible approach to streaming which can be used for developing both backward error recovery and persistence features.

First we introduce a separation of buffered and non-buffered streams. We believe that these are essentially different and that it is important to introduce this difference on an abstract level. The two main reasons for this decision are:

- the stream control is different for buffered/non-buffered streams
- very often programmers can make performance optimizations because they know the peculiarities of the application with respect to buffering, size of data, phases of the program execution, characteristics of the media which stores the data, etc.

In the first subsection, an extended stream interface is proposed to allow an additional control related to buffered streaming. Secondly, we develop a type hierarchy which includes different storage types: *volatile*, *non-volatile*, *stable* and *non-stable*.

4.1 Buffered Streams

The Ada Reference Manual states that streams can be implemented in various ways, providing access to external sequential files, internal buffers or even network channels [6, 13.13]. The language manual provides an interface for streams by defining an abstract root type in the package `Ada.Streams` from which concrete implementations must derive. The only concrete implementation that is defined in the language standard is the stream type that provides sequential file access mentioned in the previous section. We have seen already that in addition to the operations defined for all stream types, the streams in the package `Stream_IO` provide file manipulation operations such as `Create`, `Open`, `Close`, `Delete`, and stream-related operations such as `Flush`. Calling `Flush` will actually write the data that has been previously written to the stream out to the file. `Flush` is an operation that takes a `File` as a parameter, but from our point of view, `Flush` should be an operation of the stream itself. Whenever streams are used to access storage devices, it is not always a good idea to write the data to the device on every call to `'Write` or `'Output`. At what time the data should be written to the device is largely device dependent. Disk devices for example are usually accessed in fixed-sized chunks of data called blocks. In this case, too many individual write accesses can result in considerable performance loss. It is much more efficient to buffer the data.

We have therefore defined a package `Streams` that provides two stream types, `Stream_Type` and `Buffered_Stream_Type`, both descendants of `Ada.Streams.Root_Stream_Type`.

```
with Ada.Streams; use Ada.Streams;
with Buffer_Types; use Buffer_Types;
package Streams is
   type Stream_Type (Storage : access Storage_Type'Class)
      is new Ada.Streams.Root_Stream_Type with private;
   procedure Read   (Stream   : in out Stream_Type;
                     Item     : out Stream_Element_Array;
                     Last     : out Stream_Element_Offset);
   procedure Write  (Stream   : in out Stream_Type;
                     Item     : in Stream_Element_Array);
   type Buffered_Stream_Type (Buffer : access Buffer_Type'Class)
      is new Ada.Streams.Root_Stream_Type with private;
   procedure Read   (Stream   : in out Buffered_Stream_Type;
                     Item     : out Stream_Element_Array;
                     Last     : out Stream_Element_Offset);
   procedure Write  (Stream   : in out Buffered_Stream_Type;
                     Item     : in Stream_Element_Array);
   procedure Flush  (Stream   : in out Buffered_Stream_Type);
end Streams;
```

This allows the user to choose between a normal stream (one that writes the data to the storage medium on every `'Write`) and a buffered stream (one that buffers the data until the user calls `Flush`). The type of storage that will be used for the stream must be chosen at instantiation time through an access discriminant (see section 4.2). This technique is described in [13] as the *Strategy* pattern.

The participants of the *Strategy* pattern are the *Strategy*, the *ConcreteStrategy* and the *Context*. The pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it. The most important participant is the *Strategy*, which declares an interface common to all supported algorithms (in our case the storage devices). *ConcreteStrategy* implements a concrete algorithm using the *Strategy* interface. Finally, *Context* is configured with a *ConcreteStrategy* object, and uses the interface defined by *Strategy* to call the algorithm.

An application programmer can instantiate a stream by passing the desired storage type as a parameter:

```
S : Stream_Ref := new Stream_Type (Instance_Of_Storage_Type);
```

4.2 The Storage Hierarchy

As shown in the previous subsection, a user starts by creating an instance of a storage type in order to instantiate a stream. The UML class diagram shown in figure 2 illustrates the hierarchy of storage types and the role they play in the *Strategy* pattern.

We split the storage hierarchy into *volatile* storage and *non-volatile* storage. Data stored in the volatile storage do not survive program termination, hardware crashes or transient errors. A volatile storage can for example be implemented using conventional computer memory. Once an application terminates or crashes, its memory is usually freed by the operating system, and therefore all internal program data are lost. On the other hand, data stored in non-volatile storage remain intact even when the program terminates. Databases or disk storage are commonly used for implementing non-volatile storage. Among the different types of non-volatile storage, we distinguish *stable* and *non-stable* ones. Data written into non-stable storage may get corrupted when the system fails (for instance, during the write operation). Stable storage ensures that the data that has been written on it will never be corrupted, even in the presence of application crashes and other failures [14]. If a crash occurs during the write operation, the previously valid state can still be retrieved. Features of this type are used in atomic transactions [15] to guarantee the durability of the database systems.

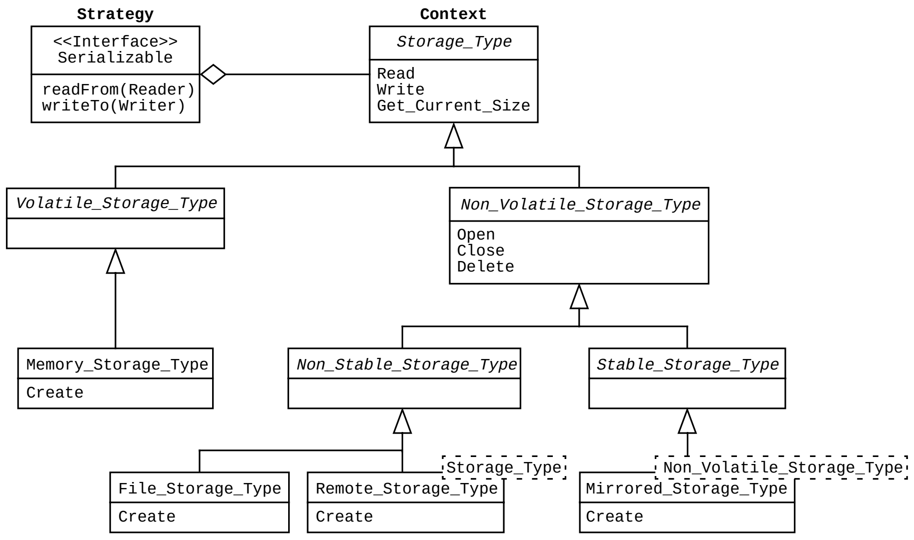

Figure 2: The Storage Type Hierarchy and the *Strategy* Pattern

The only two concrete storage types currently implemented are volatile memory and non-volatile, non-stable disk storage. The generic class `Remote_Storage_Type` allows any storage to be called remotely using the Ada Distributed Systems Annex, thus transforming the storage into a non-volatile storage. There are also two generic classes that allow to create stable storage based on non-stable storage, `Mirrored_Storage_Type` and `Replicated_Storage_Type` (not shown in the figure due to space reasons).

The interface of the top-level `Storage_Type` is given below:

```
with Ada.Streams; use Ada.Streams;
with Ada.Finalization; use Ada.Finalization;
package Storage_Types is
   type Storage_Type (<>) is abstract tagged limited private;
   type Storage_Ref is access all Storage_Type'Class;
   procedure Read   (Storage  : in out Storage_Type;
                     Item     : out Stream_Element_Array;
                     Last     : out Stream_Element_Offset) is abstract;
   procedure Write  (Storage  : in out Storage_Type;
                     Item     : in Stream_Element_Array) is abstract;
   function Get_Current_Size (Storage : in Storage_Type)
      return Stream_Element_Count is abstract;
private
   type Storage_Type is new Limited_Controlled with null record;
end Storage_Types;
```

Storage_Type is privately derived from Limited_Controlled in order to allow concrete storage implementations to perform automatic initialization and finalization, if necessary. Disk files for instance should always be closed, network ports should be freed, etc. Storage_Type is limited, so it can store, if necessary, other limited data, such as for example file descriptors. Finally, the public view of Storage_Type has unknown discriminants. That way the user of a storage type is forced to call one of the constructor functions of a concrete storage type; he can not just declare an instance of the type and thereby bypass correct initialization.

The operations provided by Storage_Type are Read, Write and Get_Current_Size. The Read and Write procedures are equivalent to the ones required for the stream type. Actually, the Read and Write procedures of the stream type are just call-though procedures to the associated storage device. The Get_Current_Size function returns the current length of the data associated with the storage in stream elements. This function has been introduced to simplify buffer management.

4.3 The Buffer Hierarchy

It is not difficult to see that to declare an instance of a buffered stream the user of the new Streams package (section 4.1) must first instantiate a buffer. Buffers here come in two flavors, unbounded and bounded.

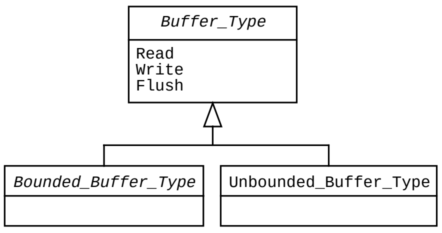

Figure 3: The Buffer Type Hierarchy

The package describing the abstract buffer type is shown below:

```
with Ada.Streams; use Ada.Streams;
with Ada.Finalization;
use Ada.Finalization;
with Storage_Types; use Storage_Types;
package Buffer_Types is
   type Buffer_Type (Storage : access Storage_Type'Class)
      is abstract new Limited_Controlled with private;
   type Buffer_Ref is access all Buffer_Type'Class;
   procedure Read   (Buffer   : in out Buffer_Type;
                     Item     : out Stream_Element_Array;
                     Last     : out Stream_Element_Offset) is abstract;
   procedure Write  (Buffer   : in out Buffer_Type;
                     Item     : in Stream_Element_Array) is abstract;
   procedure Flush (Buffer : in out Buffer_Type) is abstract;
private
   type Buffer_Type (Storage : access Storage_Type'Class)
      is abstract new Limited_Controlled with null record;
   procedure Finalize (Buffer : in out Buffer_Type);
end Buffer_Types;
```

When using buffered streams, the user must first decide what kind of buffer he wants to use, instantiate it and pass the reference to the buffered stream. When instantiating a buffer, a storage device must be passed as a discriminant.

The buffer type is derived from Limited_Controlled in order to perform proper finalization of the associated storage device. The Read and Write operations of the buffered stream will call the Read and Write operations of the buffer type. In the Write procedure, the data is first written into a memory buffer, and only when Flush is called, the data is written out to the corresponding storage. Read does the inverse, that is it will try and read all the data or as much data as fits from the storage device into the buffer upon the first call to read. Subsequent calls can then be served without accessing the storage.

When implementing the unbounded buffer class, it was possible to use an instance of the volatile memory storage type to buffer the data. This illustrates the increased possibilities of reuse.

4.4 Non-Volatile Storage

Compared to volatile storage, data stored in non-volatile storage will survive program termination. It is therefore necessary to provide housekeeping operations similar to the ones provided by Ada.Streams.Stream_IO for files. These include above all operations for creation and destruction of such non-volatile data. The non-volatile storage type provides three new operations for this purpose:

```
procedure Open (Storage : in out Non_Volatile_Storage_Type) is abstract;
procedure Close (Storage : in out Non_Volatile_Storage_Type) is abstract;
procedure Delete (Storage : in out Non_Volatile_Storage_Type) is abstract;
```

Open allows the user to establish a connection between already existing data on the device and the storage type. This is for instance needed for files, but also for network sockets or databases. The Close operation severs the association again, leaving the data on the device. Delete is used to definitively remove the data from the storage device.

4.5 Identifying Non-Volatile Data

Since the actual data stored on non-volatile storage will survive the lifetime of the object instance that represents it during program execution, there must be some means to uniquely identify the data in order to be able to manipulate the data again on subsequent runs of the application. Files usually have file names associated with them, but other storage types may use different identification techniques. Data stored in persistent memory for instance can be identified using addresses. In order to provide correct identification for each storage type, a hierarchy of storage parameter objects has been introduced. The class diagram in figure 4 shows the structure of the storage parameter hierarchy. It is identical to the one for storage types.

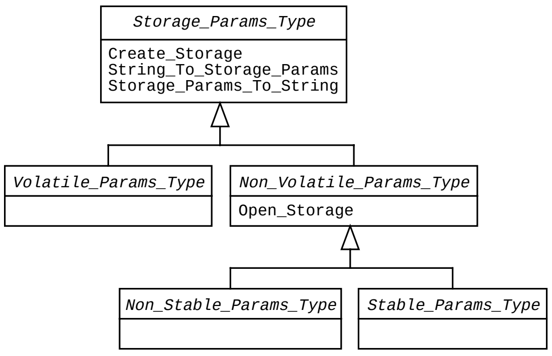

Figure 4: The Storage Parameter Hierarchy

The first function, Create_Storage, allows a user to create an instance of the storage type that corresponds to the supplied storage parameters. This technique is known as the *Factory Method* pattern. A concrete Create_Storage will call the appropriate Create function of the storage type[1]. The second function, String_To_Storage _Params, is provided to ease the creation of storage parameters. Strings can provide a

1. Remember that the storage type has unknown disciminants, and therefore the user can not declare an instance of the type without using this constructor function.

common way to identify data, regardless on what actual type of storage device the data is stored on. Using the String_to_Storage_Params function and its inverse function Storage_Params_To_String it is also possible to identify data that moves from one storage device to another one using the same string.

For the same reasons as the non-volatile storage type, non-volatile storage parameters offer a new function Open_Storage that looks for already existing data on the storage device, creates an instance of the corresponding storage type and establishes a connection between the device and the instance.

5 Example

In this section we demonstrate how the new stream interface proposed in section 4 can be used for developing a generic package which can be used to make any non-limited tagged type persistent. The specification of this package is as follows:

```
with Ada.Streams; use Ada.Streams;
with Ada.Finalization; use Ada.Finalization;
with Streams; use Streams;
with Storage_Types.Non_Volatile; use Storage_Types.Non_Volatile;
with Storage_Params.Non_Volatile; use Storage_Params.Non_Volatile;
generic
   type Base_Type is tagged private;
package Persistent_Object_G is
   type Persistent_Type (<>) is new Base_Type with private;
   type Persistent_Ref is access all Persistent_Type'Class;
   function Create (Storage_Params : in Non_Volatile_Params_Type'Class)
      return Persistent_Ref;
   function Restore (Storage_Params : in Non_Volatile_Params_Type'Class)
      return Persistent_Ref;
   procedure Save (Object : in out Persistent_Type'Class);
private
   type Persistent_Data_Type is new Controlled with record
      Storage_Stream : Stream_Ref;
   end record;
   procedure Finalize (S : in out Persistent_Data_Type);
   procedure My_Write  (Stream   : access Ada.Streams.Root_Stream_Type'Class;
                        Item     : in Persistent_Data_Type);
   for Persistent_Data_Type'Write use My_Write;
   procedure My_Read   (Stream   : access Ada.Streams.Root_Stream_Type'Class;
                        Item     : out Persistent_Data_Type);
   for Persistent_Data_Type'Read use My_Read;
   type Persistent_Type is new Base_Type with record
      Data : Persistent_Data_Type;
   end record;
end Persistent_Object_G;
```

As you can see, mix-in inheritance is used to add three new operations to the base type: Create, Restore and Save. Since the persistent object type has unknown discriminants, Create and Restore must be used to declare an instance of a persistent object. Create will create a new instance from scratch, whereas Restore will try and read the contents of the instance from the storage device identified by the storage parameters, assuming that the object has been previously saved to the device. Save is the operation that must be called to store the contents of the object onto the associated storage.

To create persistent objects, the generic package must be instantiated:

```
with My_Types;
with Persistent_Object_G;
package Persistent_Integer is
   new Persistent_Object_G (My_Types.My_Integer_Type);
```

The following lines of code illustrate how an instance of such a persistent integer type can be saved to a file on disk:

```
with Storage_Params.Non_Volatile.Non_Stable.File_Storage_Params;
use Storage_Params.Non_Volatile.Non_Stable.File_Storage_Params;
declare
   S : Persistent_Integer.Persistent_Ref;
   P : File_Storage_Params_Type := String_To_Storage_Params (“filename”);
begin
   S := Persistent_Integer.Create (P);
   S.I := …;
   Save (S.all);
end;
```

Let’s take a look at the implementation of this generic package. `Persistent_Type` adds a controlled component called `Persistent_Data_Type` to `Base_Type`. This `Persistent_Data_Type` contains a reference to a stream. The following lines of code show how this stream is allocated during a call to `Create`:

```
function Create (Storage_Params : in Non_Volatile_Params_Type'Class)
   return Persistent_Ref is
   Result : Persistent_Ref := new Persistent_Type;
begin
   Result.Data.Storage_Stream := new
   Stream_Type (Non_Volatile_Storage_Ref (Create_Storage (Storage_Params)));
   return Result;
end Create;
```

To create a stream, we need a storage object. To instantiate the storage we call the factory method `Create_Storage`, passing as an argument the given storage parameters.

Now we also understand why the persistent data type must be controlled. It is important to free the memory associated with the stream and release the storage device once the object no longer exists. The implementation of `Save` is also quite straightforward:

```
procedure Save (Object : in out Persistent_Type'Class) is
begin
   Persistent_Type'Class'Output (Object.Data.Storage_Stream, Object);
end Save;
```

The contents of the object are output to the stream using the `’Class’Output` attribute. The `Restore` function can then read the object back in using `’Class’Input`.

6 Shared Passive Partitions and Data Saving

Besides Ada streams, there is another standard Ada API that could be used for providing data persistence. The Distributed Systems Annex (Annex E) of the Ada 95 Reference Manual [6] defines so called *shared passive partitions* intended for providing access to global data shared between different partitions in a distributed system. During the configuration of a distributed Ada program, passive partitions are mapped to *processing nodes* or *storage nodes*. Any access of an active partition to a variable declared in a shared passive partition will then automatically be translated into an access to the designated processing node or storage node. A typical example of a shared passive partition is shared memory in a multiprocessor environment.

The Ada standard does not address the questions of whether the data kept in a shared passive partition survive program termination. If a shared passive partition is mapped to a non-volatile storage, such as files for example, the data stored in it may do so. The Ada standard does not require this as it does not impose any links between persistence or fault tolerance, on the one hand, and distribution in general, on the other.

Starting with version 3.12, the GNAT compiler [16] has allowed non-distributed Ada programs to use shared passive partitions. The compiler maps each variable declared in a shared passive partition to a file named after the expanded variable name. In subsequent application runs, the contents of these variables are automatically initialized with the contents stored in the files.

Although shared passive partitions providing automatic data persistence are easier for the application programmer to use, we have decided against using them for many reasons:

- Although shared passive partitions are defined in the Ada standard, they are part of the Distributed Systems Annex, and therefore a standard Ada compiler is not required to support them. Even if shared passive partitions are supported, no guarantees can be given regarding data persistence, since the Ada Reference Manual does not address persistence at all.
- Which kind of storage is to be used for a particular object is decided at configuration time, and is therefore compiler-dependent. It is also less flexible as it is not possible to change the storage of an object during run-time.
- Using shared passive partitions makes adding support of new storage media difficult as the interface becomes compiler-dependent.
- Storage control is less explicit because data saving will occur automatically during every assignment to a variable that has been declared in a shared passive partition.
- Using shared passive partitions can cause a decrease in performance when fault tolerance features are implemented on top of persistence because, to provide fault tolerance, only state that is considered to be *consistent* should be saved to storage. For example, in *transactions* data stored in transactional objects are written to stable storage only when a transaction commits.

Nevertheless, we have contacted the authors of GLADE [17], the implementation of the Distributed Systems Annex of the GNAT compiler, to evaluate the possibility of using shared passive partitions as an interface to our storage hierarchy. A standard interface between the compiler and the storage hierarchy must be defined and the configuration language will have to be extended in order to allow programmers to choose the desired storage.

7 Conclusions and Future Work

In this paper we propose a general approach to developing flexible features for reliable and persistence streaming in Ada. Fault tolerance (via backward error recovery) and persistence supports can be developed using this approach. Our approach uses standard Ada features only and can therefore be used with any standard Ada compiler and run-time system. The approach heavily relies on the peculiarities of object-oriented programming: we propose a class hierarchy of the storages of different types suitable for achieving fault tolerance and data persistence; the resulting approach promotes re-use and object-oriented programming. Our approach uses basic ideas of Ada streams for flattening data of different types and adds the ability to keep the flattened data on different storage devices depending on the application requirements.

We have found that the standard Ada 95 stream interface does not separate sufficiently the different streaming backends from the actual streams. For this reason, a new interface for streams based on the *Strategy* pattern has been designed and implemented. The example of a generic package providing object persistence demonstrates the usefulness of this new interface.

In the future, we intend to gain more experience by implementing different kinds of storages, e.g. interfaces to databases, and by using complex realistic case studies. We will use the new stream interface to add persistence to our shared recoverable objects described in [18], and provide an automatic restore capability after crash failures. Our plans are then to implement some kind of concurrent transactional service built upon these abstractions. Another promising directions of the research is to implement state restoration features which can be used in the Ada recovery block scheme (the challenging task here is to facilitate state restoration and make it transparent for the users as much as possible).

8 Acknowledgements

Alexander Romanovsky has been partially supported by EPSRC/UK DISCS ("Diversity in Safety Critical Software") Project.

9 References

[1] Lee, P. A.; Anderson, T.: “Fault Tolerance - Principles and Practice”. In *Dependable Computing and Fault-Tolerant Systems*, volume 3, Springer Verlag, 2nd ed., 1990.

[2] Randell, B.: “System structure for software fault tolerance”. *IEEE Transactions on Software Engineering* ***1****(2)*, pp. 220 – 232, 1975.

[3] Atkinson, M. P.; Buneman, O. P.: “Types and Persistence in Database Programming Languages”. *ACM Computing Surveys* ***19****(2)*, pp. 105 – 190, June 1987.

[4] Atkinson, M. P.; Bailey, P. J.; Chisholm, K. J.; Cockshott, W. P.; Morrison, R.: “An Approach to Persistent Programming”. *Computer Journal* ***26****(4)*, pp. 360 – 365, 1983.

[5] Atkinson, M. P.; Morrison, R.: “Orthogonally Persistent Object Systems”. *VLDB Journal* ***4****(3)*, pp. 319 – 401, 1995.

[6] ISO: *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*, Lecture Notes in Computer Science **1246**, Springer Verlag, 1997; ISO, 1995.

[7] Kermarrec, Y.; Nana, L.; Pautet, L.: “Providing fault-tolerant services to distributed Ada 95 applications”. In *TRI-Ada’96 conference*, pp. 39 – 47, ACM Press, December 1996.

[8] Crawley, S.; Oudshoorn, M.: “Orthogonal Persistence and Ada”. In *Proceedings of TRI-Ada’94, Baltimore, Maryland, USA, November 1994*, pp. 298 – 308, ACM Press, 1994.

[9] Oudshoorn, M. J.; Crawley, S. C.: “Beyond Ada 95: The Addition of Persistence and its Consequences”. In *Reliable Software Technologies - Ada-Europe’96*, volume 1088 of *Lecture Notes in Computer Science*, pp. 342 – 356, Springer Verlag, 1996.

[10] Riehle, D.; Siberski, W.; Bäumer, D.; Megert, D.; Züllighoven, H.: “Serializer”. In *Pattern Languages of Program Design 3*, pp. 293 – 312, Addison Wesley, 1998.

[11] Object Management Group, Inc.: *Externalization Service Specification*, December 1998.

[12] Sun Microsystems: *Java Object Serialization Specification*, November 1998.

[13] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[14] Lampson, B. W.; Sturgis, H. E.: “Crash Recovery in a Distributed Data Storage System”. *Technical report*, XEROX Research, Palo Alto, June 1979.

[15] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[16] Banner, B.; Schonberg, E.: “The Structure of the GNAT Compiler”. In *Proceedings of TRI-Ada’94, Baltimore, Maryland, USA, November 1994*, pp. 48 – 57, ACM Press, 1994.

[17] Pautet, L.; Tardieu, S.: “Inside the Distributed Systems Annex”. In *Reliable Software Technologies - Ada-Europe’98*, volume 1411 of *Lecture Notes in Computer Science*, pp. 65 – 77, 1998.

[18] Kienzle, J.; Strohmeier, A.: “Shared Recoverable Objects”. In Harbour, M. G.; de la Puente, J. A. (Eds.), *International Conference on Reliable Software Technologies - Ada-Europe’99, Santander, Spain, June 7-11 1999*, volume 1622 of *Lecture Notes in Computer Science*, pp. 397 – 411, 1999.