A Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages

Jörg Kienzle
Software Engineering Laboratory
Swiss Federal Institute of Technology
CH—1015 Lausanne, Switzerland
Joerg.Kienzle@epfl.ch

Alexander Romanovsky
Department of Computing Science
University of Newcastle
NE1 7RU, Newcastle upon Tyne, UK
Alexander.Romanovsky@newcastle.ac.uk

ABSTRACT

This paper describes an approach to providing object persistence in object-oriented programming languages without modifying the run-time system or the language itself. By successively applying design patterns, such as the *Serializer*, *Factory Method*, and *Strategy*, we develop an object-oriented framework for providing object persistence. The advantages of object-orientation are highlighted: structured classification through class-hierarchies, extensibility and promotion of reuse. The framework clearly separates persistence control from storage control. A hierarchy of different storage types, useful in different application domains, is introduced. The framework does not rely on any kind of special programming language features. It only uses basic object-oriented programming techniques, and is therefore implementable in any object-oriented programming language. An experimental implementation in Ada 95 is presented.

Keywords
Persistence, Stable Storage, Object-Oriented Framework, Design Patterns, Streams.

INTRODUCTION

Research into persistent programming languages and systems in recent years has shown that this technology is useful for developing complex software in many problem domains. Persistence is used whenever data values from a program execution are saved so that they can be used in a later execution. Software fault tolerance mechanisms based on backward error recovery use persistence to provide state restoration in case of computer crashes or errors caused by software design faults [1]. Transaction durability [2] is often achieved using persistence techniques. How the data is saved and what kind of storage medium is used for that purpose depends on the applications demands and can vary considerably from one application to another. Unfortunately, widely used object-oriented programming languages still do not offer support for persistence.

This paper describes the design of a framework for providing object persistence in object-oriented programming languages. The outline of the paper is as follows: the next section introduces persistence and gives a brief overview of some programming languages that address it; section 3 shows how the ultimate goal of ensuring persistence is gradually approached, step by step, by applying design patterns; section 4 presents the full framework, obtained by putting together the partial solutions of section 3; the following section describes an experimental implementation of the framework using the object-oriented programming language Ada 95; section 6 looks at other related work in this area, and the last section draws some conclusions.

PERSISTENCE AND PROGRAMMING LANGUAGES

Persistence is a general term for mechanisms that allow application data from a program execution space to somehow survive the execution of the program, so that in a later execution it can be used again. There are many schemes that can be used for supporting persistence. For a complete survey, the reader is referred to [3].

The most sophisticated and desired form of persistence is the *orthogonal persistence* [4]. It is the provision of persistence for *all* data irrespective of their type. In a programming language providing orthogonal persistence, persistent data is created and used in the same way as non-persistent data; loading and saving of values does not alter their semantics; and the process is transparent to the application program. Whether or not data should be made persistent is often determined using a technique *called persistence by reachability*. The persistence support designates an object a *persistent root* and provides applications with a built-in function for locating it. Any object that is "reachable" from the persistent root, for instance by following pointers, is automatically made persistent.

The first language to provide orthogonal persistence, PS-Algol [5], was conceived in order to add persistence to an existing language with minimal perturbation of its initial semantics and implementation. There are persistent versions of functional programming languages, such *as Persistent Poly* [6] and *Poly ML* [7]. There has also been research done on adding orthogonal persistence to widely used programming languages. Probably the most interesting project at the moment is *PJava* [8], a project that aims at providing orthogonal persistence to the *Java* [9] programming language.

As orthogonal persistence is extremely difficult to achieve, all these implementations had to slightly modify the programming language and / or modify the run-time system. The papers [10, 11] for instance investigate adding orthogonal persistence to the *Ada 95* [12] language. The authors identify the following problems:

- Orthogonal persistence requires that both data and types can have indefinite lifetimes. If a persistent application is to evolve, structural equivalence and dynamic type checking are necessary when a program binds to an object from the persistent store. When introducing orthogonal persistence, type compatibility within an execution extends to type compatibility across different executions. This may conflict with the typing rules of the programming language.
- Often programming languages allow the use of static variables inside classes or even as standalone global variables. It is possible that a programmer uses such static variables to link objects together, such as for instance a static table that links *key* values to some other data. Now if the *key* values are made persistent, the table should also persist, or else the *key* values are useless. It might be tricky to provide automatic persistence for such static variables without breaking orthogonal persistence.
- Orthogonal persistence also requires that elaborate types such as task types / threads and subprogram pointers values persist. This can raise severe implementation problems.
- A program might evolve and change the definition of types and classes, but still try and work with values saved in previous executions. To make this work, some form of version control must be provided, and additional dynamic checking is required. The problem can be even more complicated when considering inheritance.
- Another important problem when providing orthogonal persistence is storage management. Persistent data that will not be used anymore must be deleted, for storage leaks will result in permanent loss of storage capacity. This basically requires some form of automatic garbage collection, at least for all persistent data.

Finally the authors conclude that adding orthogonal persistence to the Ada 95 language would require major changes, making the new language backwards incompatible. It is interesting to note here that even in the case of the Java language, a modern object-oriented language that already provides automatic garbage collection and a powerful reflection mechanism, the virtual machine executing the Java byte code had to be modified in order to support orthogonal persistence [13].

As soon as we do not require orthogonal persistence, persistence support for conventional programming languages can be provided in multiple ways. Many languages have been extended or provide standard libraries that allow data to be made persistent for instance by saving it to disk. Avalon [14] for instance is an extension to C++ that provides persistence and transactions. The authors have extended the C++ language by adding new reserved words. *Stable* for instance is used to designate class attributes that are to be made persistent.

Persistence support in object-oriented programming languages must provide a mechanism that allows the state of an object to persist between different executions of an application. It can be quite challenging to find means for taking the in-memory representation of the objects state and writing it to some storage device. Fortunately, object-oriented programming languages often provide some kind of streaming functionality [12, 9] that allows transforming the state of an object into a flat stream of bytes. Some languages go even further and provide streams that allow a user to write objects into files or other storage devices (Ada `Stream_IO` [12, A.12.1], Java [9] `FileOutputStreams`). Unfortunately, the facilities provided by the programming language are not always sufficient, or they lack modularity and extensibility, making the definition of new persistent objects or the addition of new storage devices difficult or even impossible.

We believe that when designing persistence support for object-oriented programming languages one should strive to achieve the following:

- *Clear separation of concerns*: The persistent object should not know about storage devices or about the data format that is used when writing the state of the object onto the storage device and vice versa.
- *Modularity and extensibility*: It should be straightforward to define new persistent objects or add new storage devices.
- *Safe storage management*: Storage leaks leading to wasted space on the storage device must be prevented.

In order to help the programmer we propose a general framework for providing object persistence that maximizes these goals. It can be used by two different types of programmers: *persistence support programmers* and *application programmers*.

Persistence support programmers will add support for new storage devices to the framework using object-oriented programming techniques.

Application programmers will use the framework to declare persistent objects. When instantiating a new persistent object, the application programmer specifies where the state of the object will be saved by choosing among the existing implementation of storage devices. The operations defined for persistent objects allow the application programmer to save or restore the state of the object at any time.

The framework does not rely on any kind of special programming language features. It only uses basic object-oriented programming techniques and is therefore implementable in any object-oriented programming language. Most of its structure is based on well-known design patterns. The remainder of this paper documents the construction of the framework.

APPLYING DESIGN PATTERNS SUCCESSIVELY

Classification of Storage Types

At some point, persistent objects must save their state to some store (a storage), so that it can be retrieved again in a later execution. The term storage is used in a wider sense here. Sending the state of the object over a network and storing it in the memory of some other computer would for instance also make sense, as long as the data survives program termination.

Nevertheless, all storage types do not have the same properties, and therefore must not all provide the same set of operations. An abstract class hierarchy is the most natural way to represent the structure of such storage types. A concrete implementation of a storage type must derive from one of the storage classes and implement the required operations.

We propose to classify the different storage types in the way presented in figure 1.

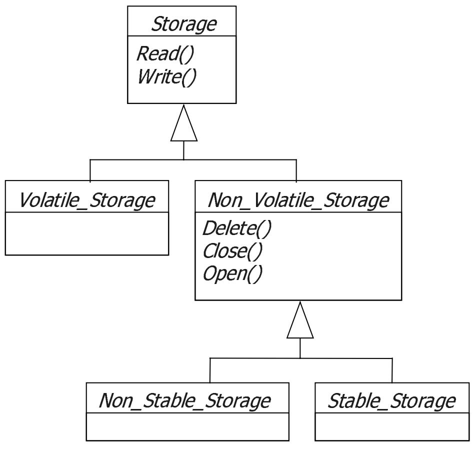

Figure 1: Storage Class Diagram

The `Storage` class represents the interface common to all storage types. The operations `Read` and `Write` represent the operations that allow reading and writing data from and to the storage device. What kind of value types they must support will be discussed in more detail in the next subsection.

The storage hierarchy is split into *volatile* storage and *non-volatile* storage. Data stored in volatile storage will not survive program termination. An example of a volatile storage is conventional computer memory. Once an application terminates, its memory is usually freed by the operating system, and therefore any data still remaining in it is lost. Data stored in non-volatile storage on the other hand remains on the storage device even when a program terminates. Databases, disk storage, or even remote memory are common examples of non-volatile storage. Since the data will not be lost when the program terminates, additional housekeeping operations are needed to establish connections between the object and the actual storage device, to cut off existing connections, and to delete data that will not be used anymore. These operations are `Open`, `Close` and `Delete`.

The kind of storage to be used for saving application data depends heavily on the application requirements. Properties such as performance, capacity of the storage media and particularities of usage (for instance *write-once* devices like CD writers) may affect the choice. Persistence can be implemented in a stronger form to support fault tolerance of different sorts, including tolerating software design faults (bugs), for instance by using the recovery block scheme [15], or tolerating faults of the underlying hardware, for instance by using checkpoints or recovery points [1]. To apply persistence properly and to choose the suitable storage type, the application programmer has to identify the fault assumptions and to know the reliability of the storage devices which can be used.

This is why among the different non-volatile storage devices, we distinguish *stable* and *non-stable* devices. Data written to non-stable storage may get corrupted, if the system fails in some way, for instance by crashing during the write operation. Stable storage ensures that stored data will never be corrupted, even in the presence of application crashes and other failures.

Stable storage has been first introduced in [16]. The paper describes how conventional disk storage that shows imperfections such as bad writes and decay can be transformed into stable storage, an ideal disk storage with no failures, using a technique called *mirroring*. When using this technique, data is stored *twice* on the disk[1]. If a crash occurs during the write operation of the first copy, the previously valid state can still be retrieved using the second copy. If the crash happens during the write operation of the second copy, the new state has already been saved in the first copy. When the system restarts later on, the state stored in the first copy is duplicated and saved over the second copy.

Using this mirroring technique, any non-volatile, non-stable storage can be transformed into stable storage. It is therefore possible to write an implementation of the mirroring algorithm that is independent of the actual non-volatile storage that will effectively be used to store the data. To

[1] Often two different physical disks are used to store the two copies of the data to increase reliability even more.

achieve this decoupling, the *Strategy* design pattern described in [17] has been used. The *Strategy* design pattern has three types of participants: the *Strategy*, the *Concrete Strategy* and the *Context*.

The *Strategy*, in our case the non-volatile, non-stable storage class, declares the common interface to all concrete strategies. The *Context*, in our case the mirroring class, uses this interface to make calls to a storage implementation defined by a *Concrete Strategy*. This could be for instance a file storage class that implements storage based on the local file system. The structure of the collaboration is shown in figure 2:

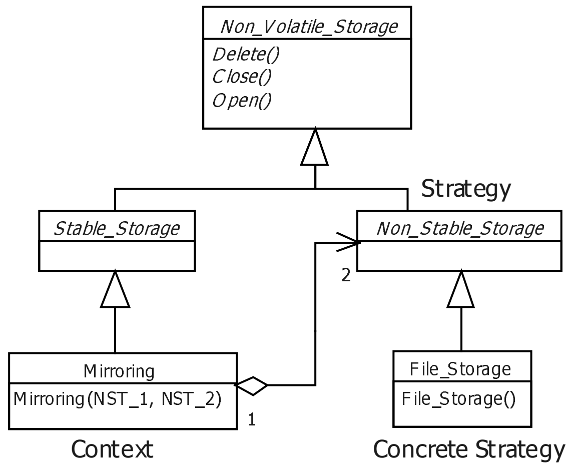

Figure 2: Stable Storage using Mirroring

At instantiation time, two non-volatile storage objects (`NST_1` and `NST_2`) must be passed as a parameter to the constructor of the mirroring class. This is how a variety of stable storages can be created reusing concrete implementations of non-volatile storage. What kind of non-volatile storage the application programmer will choose depends on the needs of the application. To help him make his choice, a concrete non-volatile storage must document the assumptions under which the storage is considered non-volatile and other information that might be useful for the application programmer such as for instance performance.

The mirroring technique is not the only one that can be used to create stable storage. Database systems for instance have their own mechanism to guarantee atomic updates of data. Typically this is done by structuring updates of data as transactions [2]. A transaction can be committed, in which case the updates will be made permanent, or aborted, in which case the system remains unchanged. If any kind of failure occurs during the transaction, the data also remains unchanged. It is possible to write a concrete stable storage class that provides a bridge between the object-oriented programming language and a database.

Yet another form of providing stable storage is replication. The state of a persistent object can be broadcasted over the network and stored for instance in remote memory. Although memory is usually not seen as non-volatile, from the application point of view it is, since it survives program termination. The group of replicas as a whole can be considered stable, for as long as at least one of the remote machines remains accessible, the data can always be retrieved during a later execution.

Just as in the mirroring example, the replicated solution can be implemented in a generic way using the *Strategy* pattern. This time, the relationship between the context and the strategy is one to many as depicted in figure 3:

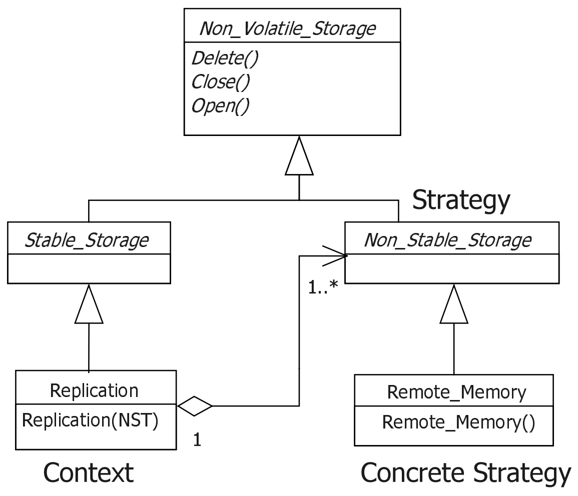

Figure 3: Stable Storage using Replication

The replication class implements broadcasting and other replica management algorithms that handle failures of replicas during program execution. As long as the set of replicas is a static one, the operations provided by the non-volatile non-stable class will do just fine. If dynamic reconfiguration of the set of replicas was to be supported, additional communication must be done between the replication class and the concrete storage during start-up of the replica. In that case, the interface provided by the non-stable class would not be enough. A new subclass must be introduced that offers additional operations.

Object Serialization

Data representing the state of a persistent object must be transformed from its representation in the main memory into a form suitable for keeping on a storage device. Most of the time the most convenient form will be a flat stream of bytes e.g. for storing data in flat files or sending data though network transport buffers. Interfaces to ODBMs can be more elaborate.

The *Serializer* design pattern described in [18] is an ideal solution for this kind of problem. It provides a mechanism to efficiently stream objects into data structures of any form as well as create objects from such data structures. The participants of the *Serializer* pattern are the *Reader / Writer*, the *Concrete Reader / Concrete Writer*, the *Serializable* interface, *Concrete Elements* that implement the *Serializable* interface and different *Backends*. The structure of the *Serializer* pattern is shown in figure 4:

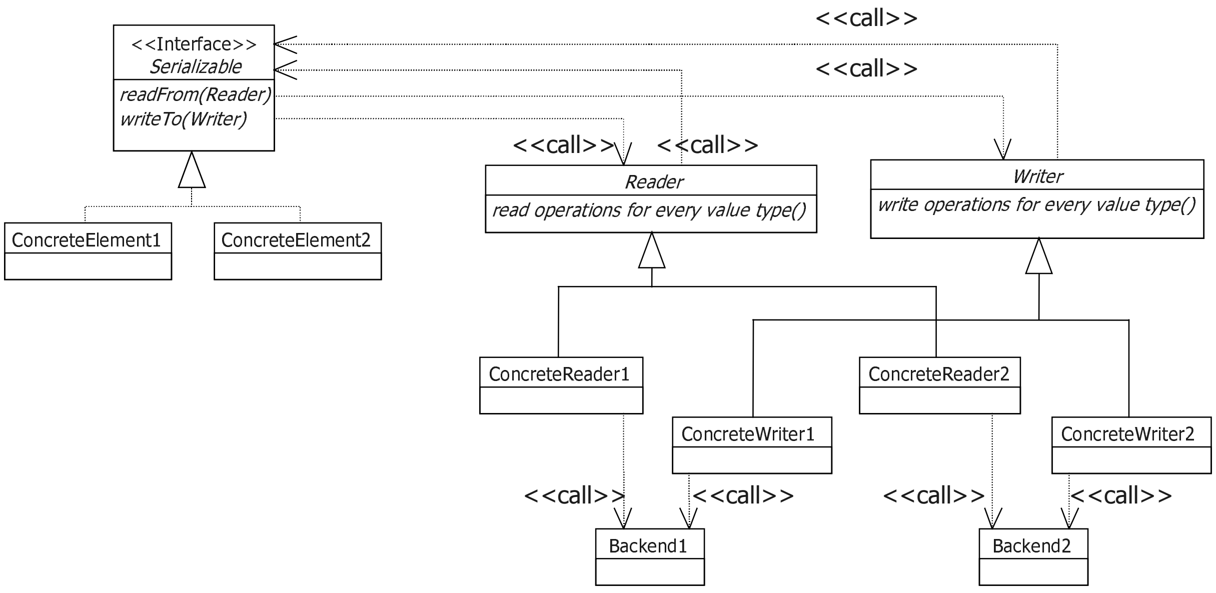

Figure 4: The Serializer Pattern

The *Reader* and *Writer* parts 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. *Concrete Element* is an object implementing the *Serializable* interface, which allows it to read and write its attributes to a *Concrete Reader* / *Concrete Writer*.

The *Backend* is a particular backend, and corresponds to our storage class shown in the previous subsection. A *ConcreteReader*/*ConcreteWriter* reads from/writes to its backend using a backend specific interface. Relational database front-ends, flat files or network buffers are examples of concrete backends.

When invoked by a client, the *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*. For certain value types such as composite types, the *Reader* / *Writer* might call back to the serializable object or forward the call to other objects that implement the *Serializable* interface. This results in a recursive back-and-forth interplay between the two parties.

The bigger the set of supported value types of the *Reader* / *Writer* interface is, the more type information can be used by the *Concrete Reader* / *Concrete Writer* to efficiently store the data on the backend. On the other hand, there are backends that support only a small set of value types. Flat files for instance only support byte transfer. For these kinds of backends the *Concrete Reader* / *Concrete Writer* must contain implementation code that maps the `read` / `write` operations of unsupported value types to the ones that are supported.

The big advantage of the *Serializer* pattern is that the application class itself has no knowledge about the external representation format which is used to represent their instances. If this were not the case, introducing a new representation format or changing an old one would require to change almost every class in the system.

In some object-oriented programming languages, such a serialization mechanism is already provided, which means that the `readFrom` / `writeTo` operations defined in the *Serializable* interface have predefined implementations for all value types of the programming language that are not covered by the *Reader* / *Writer* interface. The Java *Serialization* package [19] or Ada *Streams* [12, 13.13] are examples of such predefined language support. If no language support is available, the `readFrom` / `writeTo` operations of the *Serializable* interface must be implemented for every *Concrete Element*.

Creation of Persistent Objects

When creating an instance of a persistent object, the application programmer must be able to specify on what kind of storage he wants the state of the object to be saved. The object can then create an instance of the corresponding storage class and thus establish a connection to the storage device.

The information needed to create an instance of a concrete storage class is device dependent. To create a new file, a user must typically provide a file name that follows certain conventions, and maybe also a path that specifies in which directory the file should be created. To access remote memory, an IP number or machine name must be provided. To solve this problem, a parallel hierarchy of storage parameters has been introduced. It has the same structure as the storage hierarchy (see figure 5). This allows each storage class to define its own storage parameter type containing all the information it needs to uniquely identify data stored on the device.

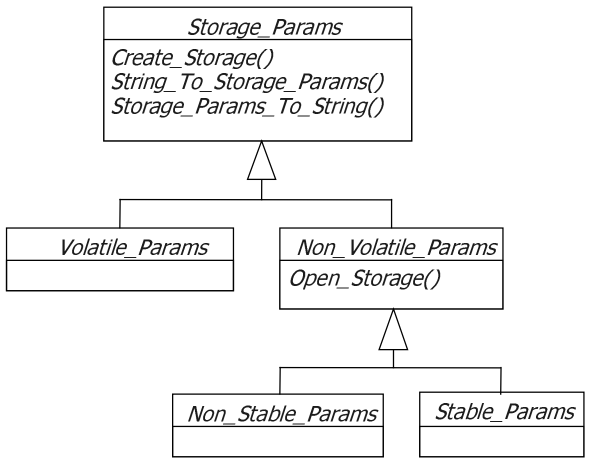

Figure 5: Storage Parameter Class Diagram

At the same time, the storage parameter class allows a user to create instances of storage classes. This is done using the well-known *Factory Method* pattern described in [17]. The participants of this design pattern are the *Product*, the *Concrete Product*, the *Creator* and the *Concrete Creator*.

The *Product* and *Concrete Product* are in our case the storage class and its descendants, as they define and implement the interface of the objects the factory will create.

The *Creator* is the storage parameter class, for it declares the abstract factory method `Create_Storage`. A *Concrete Creator*, in our case a concrete storage parameter class, must provide an implementation for this method: the corresponding creator function of the storage class must be called, passing as a parameter the information stored inside the concrete storage parameter instance. Non-volatile storage needs a second creator function, `Open_Storage`, that will instantiate the non-volatile class without creating a new storage on the device. Instead a connection between already existing data and the storage object will be established.

The `Create_Storage` and the `Open_Storage` operations define the connection between the two parallel class hierarchies.

Identification of Persistent Objects

Since the state of a persistent object survives program termination, there must be a unique way to identify a persistent object that remains valid over several executions of the same program. The storage parameter that has been introduced in the previous subsection uniquely identifies a location on the storage device, and can therefore also be used as a means for object identification.

Sometimes it can be convenient for an application programmer to treat persistent objects in a uniform way. An object name in the form of a string has proven to be an elegant solution for uniform object identification [8]. The two functions `Storage_Params_To_String` and `String_To_Storage_Params` provide a mapping between the two identification means.

Storage Management

Once a persistent object has been created and its state saved to a non-volatile storage, the data will theoretically remain on the storage forever. The only way to remove the data and free the associated storage space is to explicitly delete the object. This situation can lead to permanent storage leaks, if the application programmer forgets to store the parameters that allow him to identify the object on subsequent application runs.

A simple solution to this problem is to provide some sort of reliable persistent directory. The parameters of every persistent object created so far are automatically stored in it. At any time, the application programmer can consult the list of existing persistent objects to determine which of them he still needs and which of them he wants to delete.

Since the objects persist, the state of the directory should also survive program termination. The directory itself therefore is just another persistent object. When writing the state of the directory to the storage, the storage parameters of all persistent objects that have been created in the system must be written to the storage. It is therefore important that the storage parameter class also implements the *Serializable* interface.

The directory must be reliable. Even a crash during the update of the directory should not corrupt the data. This can be achieved by storing the directory on a stable storage. But this is not enough. Storage leaks can occur if a crash occurs after a new persistent object has been created, but before the creation has been registered in the directory. To prevent this problem, the creation and deletion of objects and the updating of the directory to reflect the change must be executed atomically.

PUTTING EVERYTHING TOGETHER

With the previous solutions in mind, we can now put together the overall system. Its structure is shown in figure 6.

For simplicity, the *Reader* and *Writer* parts of the *Serializer* pattern are shown as one class.

Using the Framework

Before using any part of the framework, the application programmer must initialize the persistence support. He has to choose where to store the persistent directory by instantiating the appropriate storage parameters and passing them to the `Initialize` operation of the persistence support class. In order to make the directory reliable a stable storage must be used. A good idea is to hard-code the storage parameters of the persistent directory in the application code, for on subsequent runs the same parameters must be used again. During the `Initialize` operation, the persistent support class will try and restore a previously valid directory. If this fails, a new, empty directory is created instead. Once the initialization has been performed, the application programmer can create, restore, save and delete persistent objects.

The root class of the framework is the persistent object class. It implements the *Serializable* interface described in the previous section and the operations `Create`, `Restore`, `Save` and `Delete`.

The registered object class derives from the persistent object class. It is responsible for registering any new persistent object instances with the persistent directory. In order to define a new persistent object, the application programmer must derive from the registered object class and add any application dependent state using new class attributes. He must also implement the `readFrom` and `writeTo` opera-

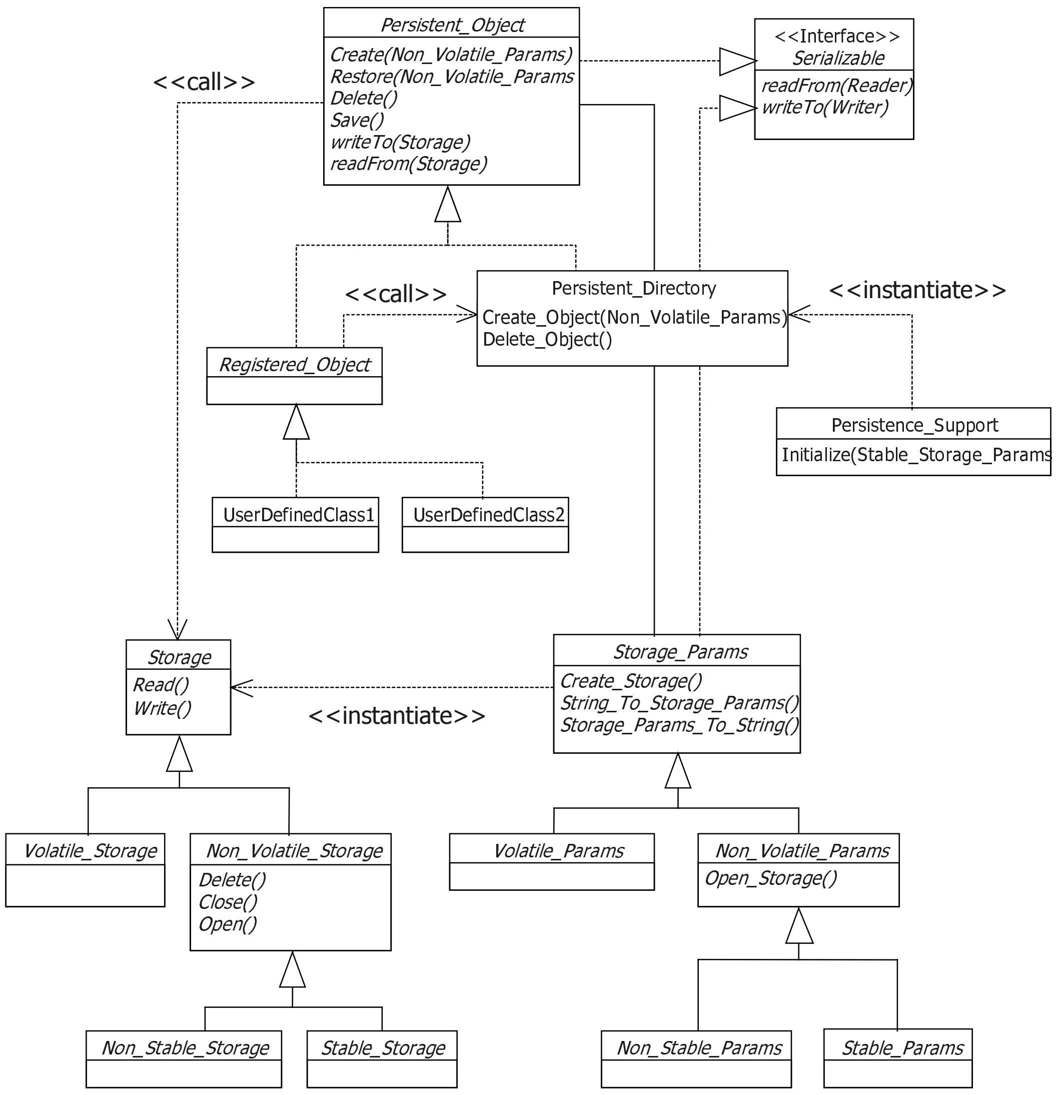

Figure 5: Framework Overview

tions of the *Serializable* interface, if the underlying programming language does not provided them automatically.

The user-defined class must also provide two constructors, `Create`, to create a new instance of a persistent object, and `Restore`, to restore the state of an already existing persistent object. They must perform initialization of the application dependent object state if needed and then up-call the corresponding constructor of the registered object class, which will perform the necessary operations for registering the object with the persistent directory.

The `Create` operation of the registered object class will call the `Create_Object` operation of the persistent directory. Inside `Create_Object`, the operation `Create` of the persistent class is called. This will actually create the object on the specified store as explained below. At the same time, the storage parameters of the newly created class are stored inside the persistent directory. As mentioned before, this must be done atomically. A simple way of implementing this is to use a simple form of logging.

The implementation of the constructors `Create` and `Restore` in the persistent object class will create an instance of the storage type identified by the storage parameter using the `Create_Storage` or `Open_Storage` factory methods. If `Restore` has been called, the state of the object is read from the storage using the operations of the *Serializable* interface.

This is again an application of the *Strategy* design pattern. The persistent object, the *Context*, is configured with a storage, the *Strategy*, at instantiation time. The application programmer chooses the storage medium for his object by passing the corresponding storage parameter to the `Create` or `Restore` constructors. Once this association has been set up, the state of the persistent object can be written to the associated storage by using the `Save` operation. The implementation will then write an object identifier and successively all object attributes to the associated storage using the operations provided by the *Serializable* interface.

EXPERIMENTAL IMPLEMENTATION

An experimental implementation of the framework [20] has been realized using the object-oriented programming language Ada 95 [12]. The fact that Ada 95 supports streaming of objects has simplified the implementation, but also narrowed down the `read` / `write` operations of the storage class to support byte reads and writes only.

This fact is reflected in the specification of the abstract `Storage_Type`. There is only one pair of `read` / `write` operations, and it operates on stream element arrays (arrays of bytes):

```
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;

private

   type Storage_Type is abstract new
        Limited_Controlled with null record;

end Storage_Types;
```

A *persistence support* programmer writing a new interface for a storage device must derive from this class (or more precisely from a subclass such as `Non_Stable _Storage_Type` whose properties correspond to the properties of the device) and implement the `Read` and `Write` operations.

An *application programmer* does not have to worry about extending the storage class hierarchy. He can use the provided implementations such as storage based on the local file system.

The following sample code shows how an application programmer uses the framework. First, he must declare his own persistent type by deriving from the registered object class, here called `Registered_Object_Type`, adding additional attributes that will contain the application data. He must also provide a constructor that allocates the new object, performs initialization and then up-calls the constructor of the registered object class. The following example shows how to declare a persistent integer type:

```
with Persistent_Objects.Registered;
use Persistent_Objects.Registered;

package Persistent_Integers is

   type Persistent_Int_Type is new
      Registered_Object_Type with record
      Value : Integer;
   end record;

   type Persistent_Int_Ref is access all
      Persistent_Int_Type'Class;

   function Create (Storage_Params :
            Non_Volatile_Params_Type'Class)
      return Persistent_Int_Ref;

end Persistent_Integers;
```

The body of the constructor is shown below:

```
function Create (Storage_Params :
         Non_Volatile_Params_Type'Class)
                return Persistent_Int_Ref is
   Result : Persistent_Int_Ref := new
      Persistent_Int_Type;
begin
   -- Call the super class constructor
   Create (Registered_Object_Ref (Result),
           Storage_Params);
   return Result;
end Create;
```

With these declarations, instances of the persistent integer class can now be created and used.

```
--  include the necessary files
with File_Storage_Params;
with Persistent_Integers;
use Persistent_Integers;
--  Create a new persistent integer and
--  save its contents
declare
   I : Persistent_Integer_Ref :=
       Create (File_Storage_Params.
          String_To_Storage_Params ("foo"));
begin
   I.Value := 123;
   Save (I.all);
end;
--  Restore the contents of the previously
--  created persistent integer
declare
   I : Persistent_Integer_Ref :=
       Persistent_Integer_Ref (
       Restore (File_Storage_Params.
         String_To_Storage_Params ("foo")));
begin
    Put (I.Value);
end;
```

When creating the persistent integer, the application programmer chooses on which storage the state shall be stored by calling the `String_To_Storage_Params` function of the chosen storage class. In the example above, the persistent integer is stored in a file, since `String_To_Storage_Params` of the `File_Storage_Params` class is called.

RELATED WORK

PJava

The PJava project [8] aims to provide orthogonal persistence for the Java language without modifying the language. Roots of persistence have been defined, where individual objects can be registered during run-time. All objects reachable from a persistent root are made persistent (*persistence by reachability*). This is achieved by modifying the Java Virtual Machine.

CORBA: Persistent Object Service

The CORBA Persistent Object Service [20] is a standardized CORBA service that allows CORBA Objects to make all or part of their state persistent. Whether or not the client of such a persistent object is aware of the persistent state is a choice the object has. By supporting special interfaces, and describing the persistent data using an interface definition language, a persistent object can delegate the management of its persistent state to other objects.

The persistent data finally get stored in so-called data-stores. This is the CORBA way to abstract the real storage devices, such as file systems or databases. Each persistent object can dynamically be connected to a data-store. From then on it is possible to save and restore the persistent state.

PerDiS

The PerDiS project [22] takes a very different approach to persistence. They address the issue of providing support for distributed collaborative engineering applications such as CAD programs, where large volumes of fine-grain, complex objects must be shared across wide-area networks. They present the application programmer with some form of a persistent, distributed memory. The application accesses this memory transactionally. The memory is divided into clusters containing objects. Named roots provide the entry points. Objects are connected by pointers. Reachable objects are stored persistently in clusters on disk; unreachable objects are garbage-collected automatically.

CONCLUSIONS

We have described the construction of an object-oriented framework providing persistence support for object-oriented programming languages. It does only rely on basic object-oriented programming techniques, and is therefore implementable in any object-oriented programming language. The feasibility has been demonstrated by an experimental implementation in Ada 95.

The advantage of using object-oriented programming is obvious. Class hierarchies have allowed us to clearly classify the different storage devices and show the dependencies among them. Abstract classes provide the interface for the different types of storage.

The structure of the framework is based on well-known design patterns. The advantages of using design patterns in this context are substantial:

- People familiar with the used design patterns will be able to understand the structure of the framework faster.
- Design patterns are solutions to specific problems that have proven to be successful.
- Design patterns enhance the modularity and flexibility of object-oriented programming.

The concrete advantages of the design patterns used in the framework are:

- The *Serializer* pattern makes it easy to add new data representation formats for objects that must be written to new storage devices by introducing a new *Reader / Writer* pair. It also moves the knowledge about the external data representation format out of the persistent object itself.
- Encapsulating the storage devices in separate *Strategy* classes allows the application programmer to change or replace particular storage implementations and the persistence support programmer to extend the storage device hierarchy without modifying the persistent object class. It also promotes reuse when stable storage is implemented on top of non-volatile storage.
- Applying the *Factory Method* pattern in combination with a parallel class hierarchy representing storage parameters allows us to provide persistent objects with a uniform way of creating storage devices. At the same time the storage parameter provides unique identification of persistent objects.

ACKNOWLEDGMENTS

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

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. Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.
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.; Chisholm, K.; Cockshott, W.: "PS-Algol: An Algol with a Persistent Heap". *ACM SIGPLAN Notices* ***17****(7)*, pp. 24 – 31, July 1981.
6. Matthews, D. C. J.: "Poly Manual". *ACM SIGPLAN Notices* ***20****(9)*, pp. 52 – 76, September 1985.
7. Matthews, D. C. J.: "A persistent storage system for Poly and ML". *Technical Report TR-102*, Computer Laboratory, University of Cambridge, January 1987.
8. Atkinson, M. P.; Daynes, L.; Jordan, M. J.; Printezis, T.; Spence, S.: "An orthogonally persistent Java". *ACM SIGMOD Record* ***25****(4)*, pp. 68 – 75, December 1996.
9. Gosling, J.; Joy, B.; Steele, G. L.: *The Java Language Specification*. The Java Series, Addison Wesley, Reading, MA, USA, 1996.
10. 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.
11. 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.
12. ISO: *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*, Lecture Notes in Computer Science **1246**, Springer Verlag, 1997; ISO, 1995.
13. Atkinson, M. P.; Jordan, M. J.; Daynès, L.; Spence, S.: "Design Issues for Persistent Java: a Type-Safe, Object-Oriented, Orthogonally Persistent System". In *Proc. of the 6th Int. Workshop on Persistent Object Systems*, Cape May NJ (USA), May 1996.
14. Eppinger, J. L.; Mummert, L. B.; Spector, A. Z.: *Camelot and Avalon - A Distributed Transaction Facility*. Morgan Kaufmann Publishers, San Mateo, CA, 1991.
15. Randell, B.: "System structure for software fault tolerance". *IEEE* Transactions *on Software Engineering* ***1****(2)*, pp. 220 – 232, 1975.
16. Lampson, B. W.; Sturgis, H. E.: "Crash Recovery in a Distributed Data Storage System". *Technical report*, XEROX Research, Palo Alto, June 1979.
17. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns*. Addison Wesley, Reading, MA, 1995.
18. Riehle, D.; Siberski, W.; Bäumer, D.; Megert, D.; Züllighoven, H.: "Serializer". In *Pattern Languages of Program Design 3*, pp. 293 – 312. Addison Wesley, 1998.
19. Sun Microsystems: *Java Object Serialization Specification*, November 1998.
20. Kienzle, J.; Romanovsky, A.: "A Flexible Approach for Streaming". *Technical Report 99/323*, Swiss Federal Institute of Technology, November 1999.
21. Object Management Group, Inc.: *Externalization Service Specification*, December 1998.
22. Ferreira, P.; Shapiro, M.; Blondel, X.; Fambon, O.; Garcia, J.; Kloosterman, S.; Richer, N.; Roberts, M.; Sandakly, F.; Coulouris, G.; Dollimore, J.; Guedes, P.; Hagimont, D.; Krakowiak, S.: "PerDiS: design, implementation, and use of a PERsistent Distributed Store". *Technical report*, QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, October 1998.