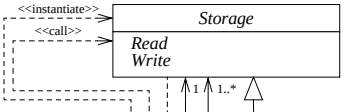
Object Persistence: A Framework Based On Design Patterns

Overview

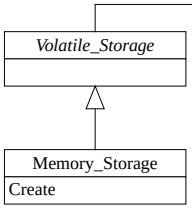
This poster presents a framework providing persistence support for object-oriented programming languages without modifying the run-time system or the language itself. It does only rely on basic object-oriented programming techniques, and can therefore be implemented in any object-oriented programming language. It is based on *design patterns*. Its strengths are:

- *Clear Separation of Concerns*: Persistent objects do not know about storage devices or external data formats.
- *Modularity and Extensibility*: It is straightforward to define new persistent objects or add support for new storage devices.
- *Safe Storage Management*: Storage leaks are prevented.

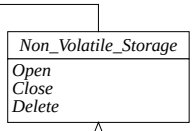
The *storage* class represents the interface common to all storage types. A persistence support programmer will extend this class hierarchy to add support for some new storage device. The operations `Read` and `Write` must support at least one kind of value type (e.g. bytes). The bigger the set of supported value types is, the more type information can be used to efficiently store the data on the storage.



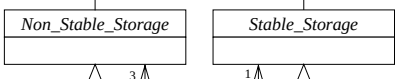
Data stored in *volatile storage* (such as main memory) will not survive program termination.



Data stored in *non-volatile storage* (such as files) will survive program termination, and therefore must offer additional housekeeping operations.

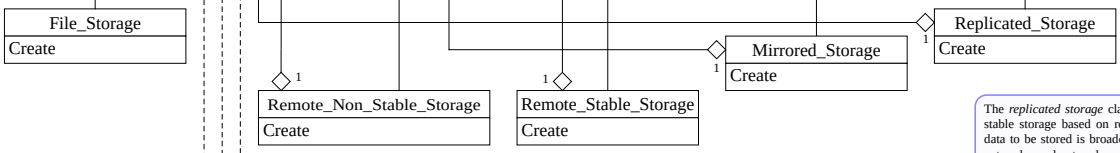


Data written to *non-stable storage* may get corrupted, if the system fails in some way, for instance by crashing during the write operation.



Stable storage ensures that stored data will not be corrupted[1], even in the presence of application crashes and other failures. This implies that the `Write` operation is *atomic*.

1.Assumptions must be well documented!

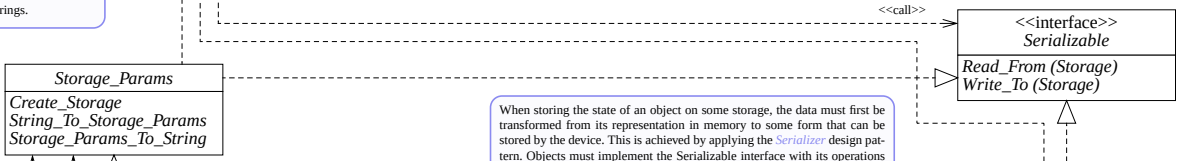


The *replicated storage* class implements stable storage based on replication. The data to be stored is broadcasted over the network and stored on the remote machine on any non-volatile, non-stable storage. Again, the *Strategy* design pattern allows to decouple broadcasting, group and replica management from the actual storage used to store the data.

The *remote* storage classes use the *Strategy* design pattern to make any storage remotely callable. In our implementation, these two classes have been implemented as a generic class (template). Depending on what kind of storage the application programmer needs, the class can be instantiated with a stable storage, creating a remote stable storage, or any other storage, creating a remote non-volatile, non-stable storage.

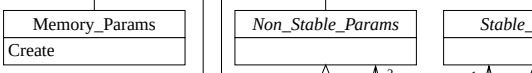
The *mirrored storage* class uses a technique called mirroring to create a stable storage based on any non-volatile, non-stable storage: two copies of the data are stored, and updates on that data are done sequentially. A log helps to perform recovery in case of failures. The decoupling of the mirroring algorithm and the actual storage classes used to store the data and the log is achieved using the *Strategy* design pattern.

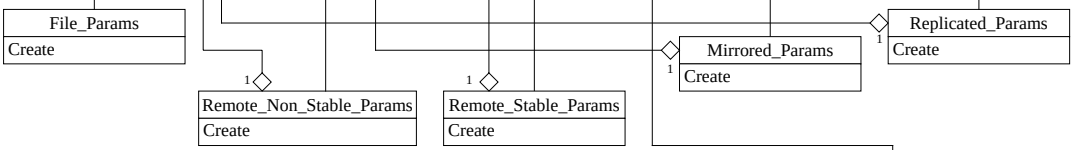
The parallel hierarchy of storage parameters allows each storage class to define its own *storage parameter* class containing all the information needed to uniquely identify data stored on the device. At the same time, it allows an application programmer to create instances of storage classes by means of the *Factory Methods* `Create_Storage` and `Open_Storage`. The functions `Storage_Parameter_To_String` and `String_To_Storage_Parameter` provide a means for uniform parameter handling by means of strings.

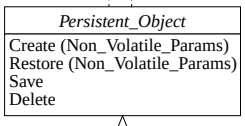


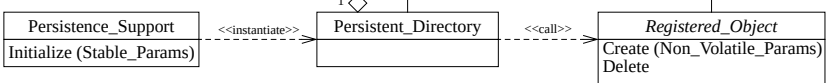
When storing the state of an object on some storage, the data must first be transformed from its representation in memory to some form that can be stored by the device. This is achieved by applying the *Serializer* design pattern. Objects must implement the Serializable interface with its operations `Read_From (Storage)` and `Write_To (Storage)`. Their implementation will call the `Read` and `Write` operations of the concrete storage to store the state of the object. Using this technique, the object itself has no knowledge about the external representation format. In some object-oriented programming languages such a serialization mechanism is already provided (Ada streams, Java serialization package).

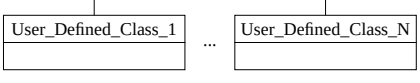
The *persistent* object class is the root class of the framework. It implements the Serializable interface. Depending on the application needs, an application programmer chooses a storage medium and passes the corresponding storage parameters to one of the constructors `Create` or `Restore` (again an application of the *Strategy* pattern). From then on. the state of the object can be saved by calling the operation `Save`. An application programmer must derive from this class to create user-defined persistent objects.











To prevent storage leaks in the presence of failures the storage parameters of any created persistent object can be stored in a stable *persistent directory*[1]. The application programmer will choose the kind of stable storage used to store the directory when initializing the persistence support. The operations of the *registered object* class will automatically update the directory. The creation / deletion of objects and the updating of the directory must be executed atomically.

1.For this purpose, the storage parameter class also implements the Serializable interface.

Implementation

This framework has been implemented using the object-oriented programming language Ada 95. Ada's elaborate features for concurrency control, serialization and distribution have been used extensively.

Additional Information

Additional information and complete papers describing the framework and the implementation can be obtained at the following URL:

`http://lglwww.epfl.ch/research/ongoing/persistence.html`

EPFL - Computer Science Department - Software Engineering Laboratory & University of Newcastle Upon Type - Computer Science Department

Jörg Kienzle & Alexander Romanovsky, April.2000