EPOM

Eiffel Persistent Object Management

Paul G. Crismer, Eric Fafchamps and others

Revision: 1.0

Date : December 1, 2003

Abstract

EPOM is a framework that allows Eiffel Applications to load/store/update persistent state of objects in some datastore.

The originality of this framework is that it acknowledges the fact that it *has to* be an interface between two possibly *different* worlds.

Both worlds address different needs. The “Object” world manages abstractions, business rules and processes . The “Datastore” world manages efficient, shared and secure data storage and retrieval.

EPOM provides an abstract, object-oriented friendly, way of storing and retrieving objects that is totally independent from any storage method (files, RDBMS, OODBMS, ...).

Table of Contents

1. Introduction

There always have been a semantic gap between object-oriented models and data storage models.

“Object” world and “Datastore” world address different needs. The “Object” world manages abstractions, business rules and processes . The “Datastore” world manages efficient, shared and secure data storage and retrieval.

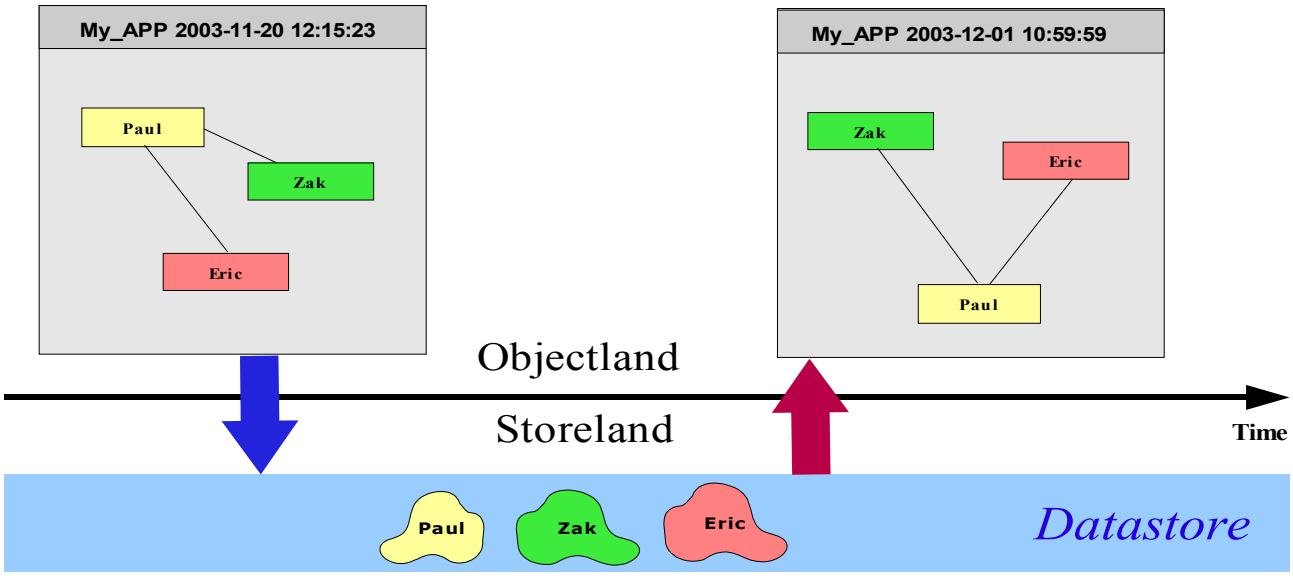
Object Oriented development mainly focuses on finding abstractions that fit some business domain. It also focuses on Software Engineering best practices : produce reusable software, reduce code duplication, design application architectures that last.

Applications are made up of interconnected objects that dialog together to perform some business activity.

Objects live in the operational space of an application session, personified by a process or a thread. Once the process is terminated, the memory space is reclaimed and objects die.

At a later time, in another application session it could be desirable to continue working with objects that were created or used in previous sessions.

This is where “persistence” appears. Before an application session ends, objects are sent to “storeland” where they rest until the next application session. They persist in “storeland” in some frozen state.



Drawing 1 Living objects and persistent state

A persistence framework eases this migration from objectland to storeland and vice-versa.

2. Quality factors

2.1. Related works

In a well known paper [AMBLER97] Scott Ambler describes very complete specifications of a robust persistence layer *dedicated to relational databases*. This document presents a check-list of the mechanisms a persistence layer should offer. This checklist shall be used to give a quick overview of what EPOM does.

In their book on BON (Business Object Notation), Nerson and Walden [WALDEN95] also propose the design of a persistence layer dedicated to relational databases.

CASTOR provides a persistence framework and associated tools for Java. The main drawback of this product is that the Relational database schema must be derived from the object model. Both models cannot evolve on their own. It is a major drawback because it forgets the fact that OO applications often must share legacy data with legacy applications.

2.2. Status

EPOM has been used in production for about 1 year at Groupe S, a major Payroll outsourcer and legal-related Human Resource services provider in Belgium. At this company, EPOM replaces an older framework which proved to be difficult to implement and to maintain. EPOM provides more flexibility and is more OO : client code is lighter.

While transaction management is possible, there is not *transaction abstraction*. Future extensions should provide such an abstraction.

2.3. Performance

EPOM does not “cost” anything in performance.

Performance of the framework depends on the performance of datastore accesses. In a relational database environments, it has been possible to speed up an application by a factor of 3 simply through carefully defining or restructuring table indexes. Applications that heavily rely on datastore accesses spend their time waiting for answers coming from the datastore server.

2.4. Constraints

-

2.5. Extendibility

EPOM has been defined so that extending it is easy and modular : i.e adding new persistant classes or new accesses.

3. EPOM Requirements

3.1. Simplicity

Given some persistent object ***p***, the persistent features should be called in a "natural" way :

```
p.write    -- to write object p into storeland
p.update   -- to update object p into storeland
p.refresh  -- to refresh object p from its image in storeland
```

In order to ensure object invariants at creation through reading, it is wise to use the factory pattern for read accesses :

```
persistent_factory.read_p (some identifying parameters)
p := persistent_factory.last_read_object
```

3.2. Persistent identifiers

Within the lifetime of a process, objects are identified by their *reference* or memory address.

Persistent objects must be identified by something that *persist* from an application session to another : persistent identifiers.

The content of persistent identifiers highly depends on the underlying datastore. PO_PID instances hold the secret about how objects are identified with respect to a datastore. That is why persistent identifiers are opaque to the clients of persistent objects and to persistent objects themselves.

Persistent Identifiers are created by *adapter* objects (see hereunder).

3.3. Persistent classes

Properties of persistent objects are abstracted in class PO_PERSISTENT.

CLASS	**PO_PERSISTENT**
Description	**Objects that can persist on some datastore**
Queries	
is_persistent	does Current have a persistent image on datastore ?
is_volatile	does Current have no persistent image on datastore ?
is_deleted	has Current's persistent image been deleted from datastore ?
is_modified	has Current been modified since last persistence operation ?
is_persistence_error	has last persistence operation given an error ?
exists	does current object exist in datastore ? Does it have an image in storeland ?
Commands	
write	write current object state to data store
update	update data store from current object state
delete	delete current object state from data store
refresh	delete current object state from data store

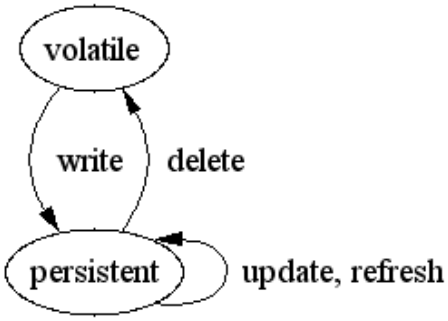
Classes whose instances must persist have to inherit from PO_PERSISTENT

There are two ways of creating instances of persistent object :

- with a creation instruction or
- by reading the datastore.

At creation, persistent objects are in *volatile* state, i.e there is no persistent image of it in datastore. The only way to create a persistent object in *persistent* state is through its *Adapter*.

The state transition diagram of persistent objects is presented in drawing 2 :



Drawing 2Persistent state diagram

3.4. Weak references

It should be possible to keep references to persistent objects without them being in memory.

The class PO_REFERENCE offers weak reference services. It is a “must have” in order to avoid reading a whole datastore when reading a single object related to other persistent objects.

3.5. Collections of persistent objects

It should be possible to query multiple instances from a datastore. Those instances are available through PO_CURSOR instances. This class gives a linear access to collections of persistent objects possibly through PO_REFERENCE instances.

3.6. Adapters

Persistence adapters bridge the gap between objectland and storeland.

Adapters manage all accesses related to one specific persistent object type. They hold the secret about how are objects stored and how are datastore entities combined to create objects.

They at least provide *basic accesses*. They can provide extended accesses.

Adapters are factories : read operations create the objects. An object that has just been read ensures its invariant.

Adapters also create PO_PID instances. They are the only objects that really know the semantics of pid contents.

3.6.1. PID factory

Adapters provide pid factory features :

- pid_for_object exported to clients,
- create_pid_from_object exported to PO_ADAPTER descendants.

3.6.2. Basic accesses

Adapters implement the basic accesses for individual objects :

Accesses	**Description**
exist	Does a persistent state exist in storeland for some persistent identifier ?
read	Create an object from a persistent identifier, and initialize it from the state in storeland
write	Create an image or persistent state of an object in storeland
update	Update the persistent state of an object in storeland
refresh	Forget the current state of an object and reload its persistent state from storeland
delete	Delete an object from storeland and mark it as logically deleted

Write, update, refresh and delete accesses operate on PO_PERSISTENT instances.

Exist and read accesses use a Persistence Identifier (*pid)*.

The problem is “where does the pid come from”? As said earlier, adapters are the *only* objects that do know the semantics of a *pid*. They should provide factory features for pids.

For example, a *book* adapter provides a feature called *create_pid_from_object*.

3.6.3. Extended accesses

Other accesses, and especially *queries* that retrieve collections of objects are supported as an extension to the basic accesses on individual objects.

For example a *book* adapter could propose the following extended accesses :

- read_by_isbn (isbn : ISBN)
- read_by_title(title : STRING)
- read_by_author_like (pattern : STRING)

3.7. Manager

The persistence manager is a broker of adapters. Adapters must be singletons in a system. An object that want to be client of an adapter must ask it to the persistence manager.

3.8. Launcher

The persistence launcher PO_LAUNCHER class have access to features of PO_MANAGER that allow registering new instances of PO_MANAGER.

3.9. Ambler's checklist

Ambler [AMBLER97] provides a list of requirements for a robust persistence framework.

We do not agree on some of Ambler's requirements, especially number 14 (direct SQL into client code "for performance reasons") which, in our eyes, destroys the whole design and make it fragile. Our experience shows that "performance" is not a good reason enough : it has always been possible to get better performance while keeping datastore-coupled considerations in adapter classes. Client code should be OO, nothing more, nothing less.

EPOM features are checked through the list provided hereunder. Ambler's requirements are literally taken from his paper. We've put ellipses (...) when some explanations have been deleted.

No	Ambler's requirements	EPOM
1	**Several types of persistence mechanism.** A persistence mechanism is any technology that can be used to permanently store objects for later update, retrieval, and/or deletion. Possible persistence mechanisms include flat files, relational databases, object-relational databases, hierarchical databases, network databases, and objectbases.	YES[1]
2	**Full encapsulation of the persistence mechanism(s).** Ideally you should only have to send the messages **save**, **delete**, and **retrieve** to an object to save it, delete it, or retrieve it respectively. That's it, the persistence layer takes care of the rest. Furthermore, except for well-justified exceptions, you shouldn't have to write any special persistence code other than that of the persistence layer itself.	YES[2]
3	**Multi-object actions.** Because it is common to retrieve several objects at once, perhaps for a report or as the result of a customized search, a robust persistence layer must be able to support the retrieval of many objects simultaneously. The same can be said of deleting objects from the persistence mechanism that meet specific criteria.	Yes[3]
4	**Transactions.** Related to requirement #3 is the support for transactions, a collection of actions on several objects. A transaction could be made up of any combination of saving, retrieving, and/or deleting of objects. Transactions may be flat, an "all-or-nothing" approach where all the actions must either succeed or be rolled back (canceled), or they may be nested, an approach where a transaction is made up of other transactions which are committed and not rolled back if the large transaction fails. Transactions may also be short-lived, running in thousandths of a second, or long-lived, taking hours, days, weeks, or even months to complete.	N/A[4] as first class objects.
5	**Extensibility.** You should be able to add new classes to your object applications and be able to change persistence mechanisms easily (you can count on at least upgrading your persistence mechanism over time, if not port to one from a different vendor). In other words your persistence layer must be flexible enough to allow your application programmers and persistence mechanism administrators to each do what they need to do.	Yes
6	**Object identifiers.** An object identifier (Ambler, 1998c), is a *feature*, that uniquely identifies an object.	Yes (PO_PID)

1 EPOM is not tight to any persistence mechanism. It has been designed with at least 2 mechanisms in mind : relational database and flat files.

2 One of the main design decision of EPOM is its Object-orientedness and its abstract interface : objects are created by factory features of persistence adapters, while object provide *write, refresh, delete, update* features.

3 Those are "extended accesses" and should be handled by object adapters.

4 Limited by the underlying persistencemechanism

No	Ambler's requirements	EPOM
7	**Cursors.** A persistence layer that supports the ability to retrieve many objects with a single command should also support the ability to retrieve more than just objects (...). A cursor is a logical connection to the persistence mechanism from which you can retrieve objects using a controlled approach (...).	Yes[5]
8	**Proxies.** A complementary approach to cursors is that of a "proxy." A proxy is an object that represents another object but does not incur the same overhead as the object that it represents. A proxy contains enough information for both the computer and the user to identify it and no more (...). By using proxies you don't need to bring all of this information across the network for every person in the list, only the information that the users actually want.	Yes[6]
9	**Records.** The vast majority of reporting tools available in the industry today expect to take collections of database records as input, not collections of objects. If your organization is using such a tool for creating reports within an object-oriented application your persistence layer should support the ability to simply return records as the result of retrieval requests in order to avoid the overhead of converting the database records to objects and then back to records.	N/A[7].
10	**Multiple architectures.** As organizations move from centralized mainframe architectures to 2-tier client/server architectures to n-tier architectures to distributed objects your persistence layer should be able to support these various approaches. The point to be made is that you must assume that at some point your persistence layer will need to exist in a range of potentially complex environments.	N/A
11	**Various database versions and/or vendors.** Upgrades happen, as do ports to other persistence mechanisms. A persistence layer should support the ability to easily change persistence mechanisms without affecting the applications that access them, therefore a wide variety of database versions and vendors should be supported by the persistence layer.	Yes
12	**Multiple connections.** Most organizations have more than one persistence mechanism, often from different vendors, that need to be accessed by a single object application. The implication is that a persistence layer should be able to support multiple, simultaneous connections to each applicable persistence mechanism. Even something as simple as copying an object from one persistence mechanism to another, perhaps from a centralized relational database to a local relational database, requires at least two simultaneous connections, one to each database.	NA but possible
13	**Native and non-native drivers.** (...) *This requirement is related to requirement 1.*	Yes
14	**Structured query language (SQL) queries.** Writing SQL queries in your object-oriented code is a flagrant violation of encapsulation – you've coupled your application directly to the database schema. However, for performance reasons you sometimes need to do so. Hard-coded SQL in your code should be the exception, not the norm, an exception that should be well-justified before being allowed to occur. Anyway, your persistence layer will need to support the ability to directly submit SQL code to a relational database.	N/A[8]

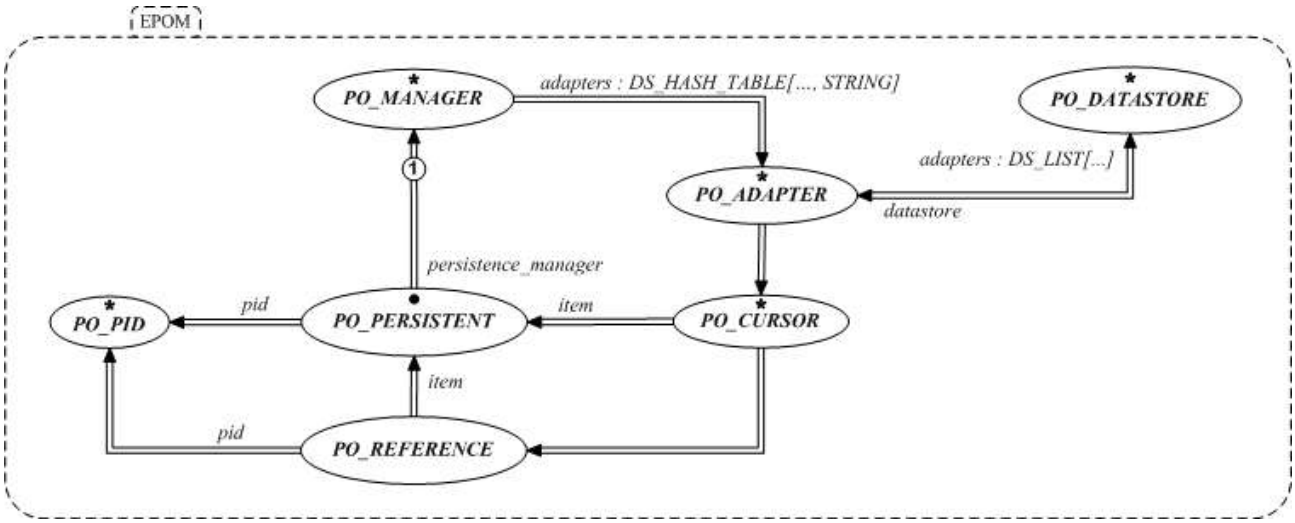
5 EPOM makes it possible to sweep through "cursors", which are collection of objets or object references.

6 This is not formally supported *as is* in EPOM. This can be provided by combining several abstractions and features and by extending the state machine associated to each persistent class: for example persistent references, read (partial persistent state) and refresh (full persistent state).

7 While this is not applicable to EPOM, this is not impossible : *extended accesses* of adapter classes are left to the creativity of developers; such accesses could retrieve collections of tuples related to a specific persistent class.

8 We totally disagree with putting direct SQL or any query language into the client code. **Datastore-dependent code should be located in the adapter classes only**.

3.10. Structure diagram



Drawing 3Structure Diagram of EPOM

The Structure diagram of EPOM shows the different classess and some of the relations between them :

- The singleton PO_MANAGER instance knows all PO_ADAPTER instances.
- The PO_ADAPTER instances bridge PO_PERSISTENT instances to some PO_DATASTORE instance, thus bridging *object-land* and *store-land*.
- PO_CURSOR uses the services of PO_REFERENCE to reference not-yet-retrieved objects.

This diagram does not show that PO_PID classes are *tightly coupled* to PO_ADAPTER classes : PO_PID instances are created by PO_ADAPTER factory features, PO_PID classes export features to PO_ADAPTER classes that are not visible to other classes.

It also does not show that PO_ADAPTER classes are *factories* for instances of PO_PERSISTENT that are retrieved from a datastore.

It also does not presents the PO_LAUNCHER class that has privileged access to the singleton PO_MANAGER instance and that creates it and is able to register new PO_DATASTORE instances to it.

3.11. Glossary

Class	**Description**
PO_ADAPTER	Objects that handle all accesses to persistent objects of type G on a datastore. The persistent objects all have class_name as persistence name. Adapters are factories : they create read object instances and let clients access them through last_cursor.
PO_CURSOR	Objects that iterate over a collection of persistent objects.
PO_DATASTORE	Objects that control datastore access

Class	Description
PO_LAUNCHER	Objects launch the persistence system. Give access to selectively exported features of PO_MANAGER_SINGLETON_ACCESS and PO_MANAGER.
PO_MANAGER	Objects that are brokers for persistence adapters.
PO_PERSISTENT	Objects that can persist on some datastore.
PO_PID	Persistent Identifiers. Identify the persistent state of persistent objects. PID instances are opaque to client applications.
PO_REFERENCE	Weak references to persistent objects.
PO_SHARED_MANAGER	Objects that share a single PO_MANAGER.
PO_STATUS	Objects that give status information about latest persistance operation.
PO_STATUS_MANAGEMENT	Objects that are allowed to manage PO_STATUS objects
PO_STATUS_USE	Objects that use a Persistence Operations Status

3.12. Creation chart

Creation chart highlights what is not shown in the structure diagram.

Class	create instances of
PO_ADAPTER	PO_PERSISTENT, PO_PID, PO_CURSOR, PO_REFERENCE
PO_LAUNCHER	PO_MANAGER, PO_DATASTORE

4. Scenarios

This section presents some important objects scenarios that highlight basic operations of the framework.

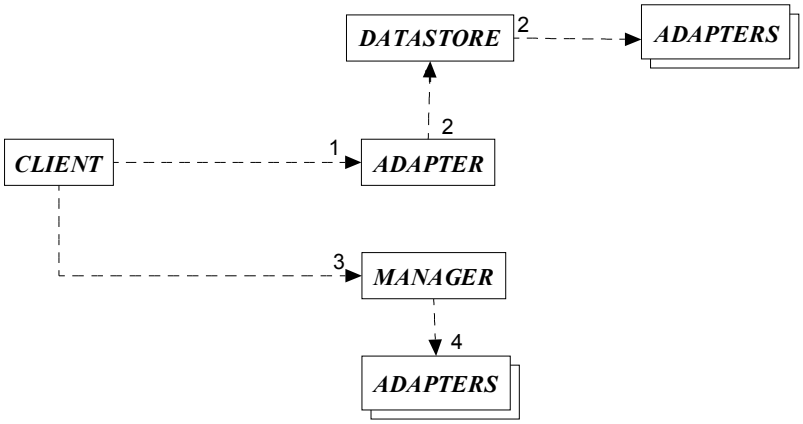
Those scenarios address (1) the management of the framework itself, and (2) object input/output.

4.1. Framework Management

Framework management include :

- initialization of the framework creating the manager singleton and registering it with adapters.
- connection of a datastore
- disconnection of a datastore
- cache control

4.1.1. Initialization of framework

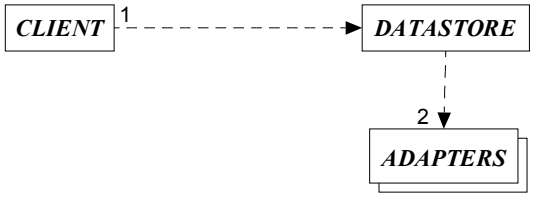


Scenario M-1: Initialization of persistance framework

1-2 Client creates an adapter, and associates it with a datastore; the adapter registers itself to the datastor

3-4 Client registers the datastore to the manager.

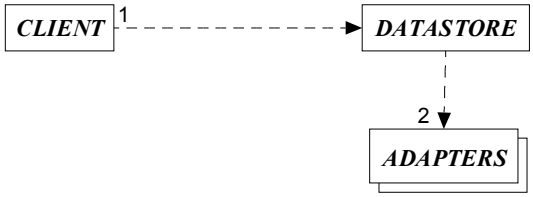
4.1.2. Datastore connection



Scenario M-2 : Datastore is connected to database

1-2 Client asks the datastore to connect; if connected the datastore calls the adapters back to let them know the connection has been established

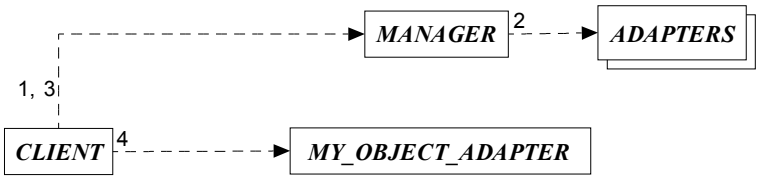
4.1.3. Datastore disconnection



Scenario M-3 : Datastore is disconnected from database

1-2 Client asks the datastore to disconnect; if disconnected the datastore calls the adapters back to let them know the connection has been disconnected

4.1.4. Cache control



Scenario M-4: Cache control

1-3 Client asks the manager for a specific adapter

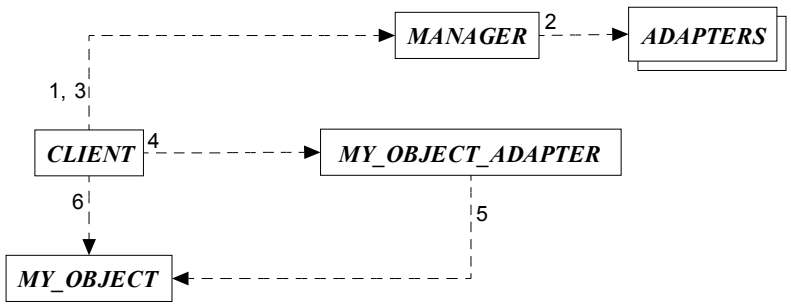
4 The adapter is asked by the client to clear its objects cache

4.2. Object I/O

4.2.1. Read

Reading objects involve asking an adapter to retrieve objects that comply with some criterias. Those *queries* either retrieve a single object or a collection.

Single object



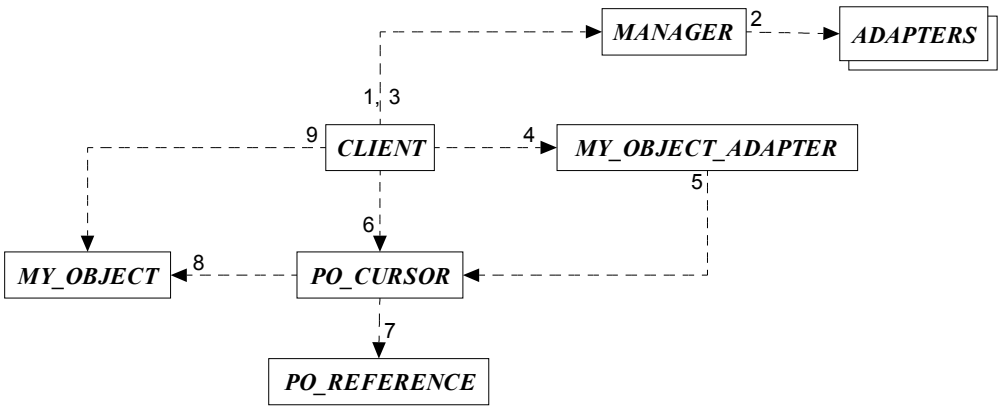
Scenario IO-1: Reading of a single object

1-3 client obtains an adapter from the manager

4-5 client ask the adapter to read an object using one of its proposed access methods; if something has been read the adapter instantiates the persistent object

6 client uses services of persistent object

Collection of objects



Scenario IO-2: Reading of a collection of objects

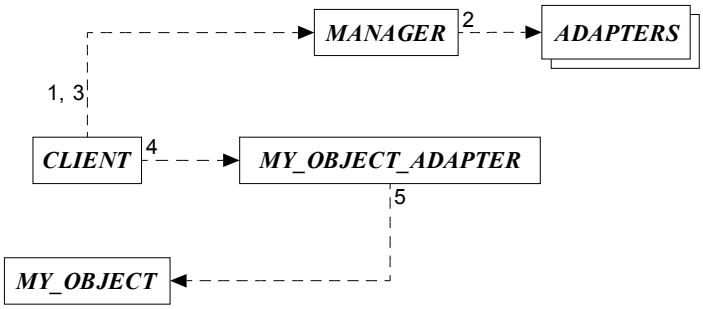
1-3 client obtains an adapter from the manager

4-5 client ask the adapter to read an object using one of its proposed access methods; if something has been read the adapter instantiates a cursor that gives access to the collection of persistent objects

6-9 client sweeps through cursor and uses services of each persistent object

4.2.2. Write

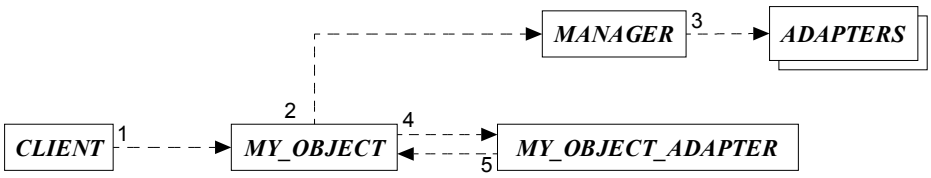
Persistent storage can be written either through the adapter, or by asking the object itself to store its persistent state.



Scenario IO-3: Modifying a single object

1-3 client obtains an adapter from the manager

4-5 client ask the adapter to write or update the persistent state of my_object



Scenario IO-4: Modifying a single object - simplified

1 client asks the object to write or update

2-3 object obtains its associated adapter through the manager

4-5 the adapter is asked to write or update the object

5. Bibliography

[AMBLER97] Scott W. Ambler. *"The design of a robust persistence layer for relational databases"*. http//www.ambysoft.com/persistenceLayer.pdf

[WALDEN95] Jean-Marc Nerson and Kim Waldén. *"Seamless Object Oriented software architecture"*. Http://www.bon-method.com/book.html.

Inspirations: Corba PSS; BON (Chapter 9).