Abstract Manager

John Liebenau

lieb@itgssi.com

Copyright © 2001, John Liebenau. Permission is granted to copy
for the PLoP 2001 conference. All other rights reserved.

1. Intent

Separate an object family's lifecycle interfaces from its domain interfaces to encapsulate domain object creation, destruction, and selection. Let concrete subclasses adapt specific lifecycle APIs to conform to the standard lifecycle interfaces. Abstract Manager enables the clients of domain objects to be independent of specific component containers and/or persistence repositories.

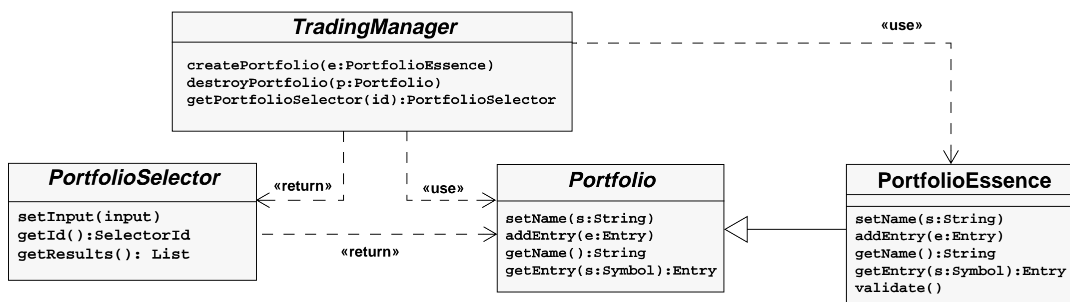
2. Motivation

Consider a business application, like a portfolio trading server, that will be deployed in a variety of operating environments. In this context an operating environment is a combination of operating system, component container and/or persistence repository. The application contains sets of business objects, such as Portfolios, Orders, and Reports, that are created, selected, and destroyed in the course of processing user requests. A business object has a typical lifecycle in which:

- data is gathered into a temporary business object,
- the temporary business object is validated,
- the temporary business object is used to create a more permanent business object that is managed by a component container or is placed in a persistence repository,
- the managed business object is retrieved for use by some kind of selection mechanism, and eventually
- the managed business object is destroyed by removal from the component container or by deletion from the persistence repository.

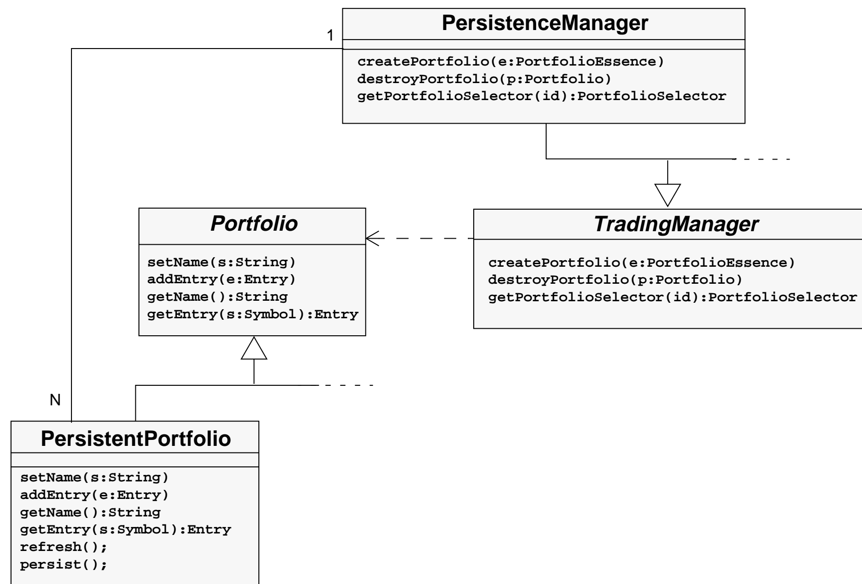
Specific component containers or persistence repositories have differing lifecycle APIs. These differences can complicate the code responsible for creating, selecting, and destroying Portfolio, Order, and Report objects because it will have to contain special cases for each API. What is needed is a way to insulate the application from specific lifecycle APIs so that the application can be free to select the operating environment that best fits each platform or deployment requirements.

We can solve this problem by declaring the following: interfaces representing Portfolio, Order, and Report business objects, Selector interfaces (e.g. PortfolioSelector) for specifying and executing the selection process for each business object type, and a TradingManager interface for creating and destroying business objects. The TradingManager interface is also used as the access point for obtaining Selectors in order to retrieve business objects. Each business object interface has an implementation called an Essence (e.g. PortfolioEssence) which provides the business object's in-memory state and behavior. An Essence is used for validation and as input for creating more permanent, managed business objects. The business object subsystem of our application would have the following interface structure (Orders and Reports are not shown):

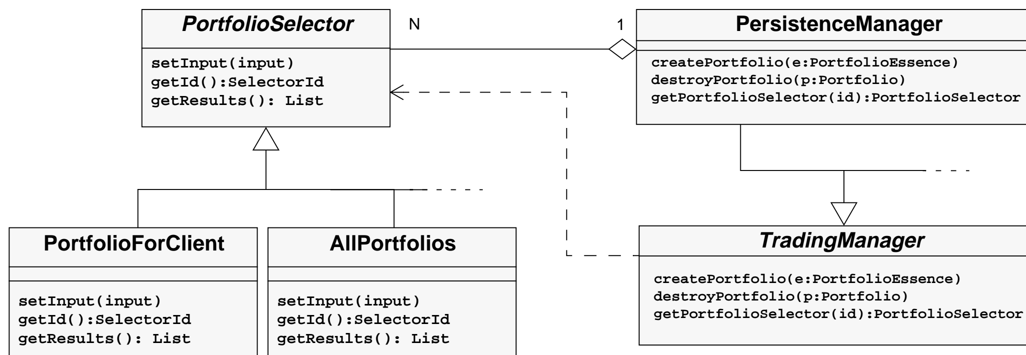


Application code will only depend on the interfaces provided by TradingManager, Portfolio, and PortfolioSelector. For each container system or persistence repository, the business object developer will have to provide concrete subclasses of these interfaces that adapt the specific lifecycle API to the common API. Each TradingManager subclass implements the operations for creating and destroying concrete subclasses of Portfolio using a specific lifecycle API.

For example, the application could use a relational database as its persistence repository and a home grown object-to-relational mapping to bring objects in and out of the database. PersistenceManager implements the TradingManager interface by providing create and destroy methods that insert and remove PersistentPortfolios from a relational database.



TradingManager declares methods for returning PortfolioSelector objects that are used to retrieve Portfolios from the database. PersistenceManager implements those methods by managing references to a set of concrete PortfolioSelector objects and returning the reference associated with a selector's identifier. This allows the set of selectors to be open to extension while keeping the TradingManager and PersistenceManager interfaces closed to changes [Meyers88][Martin96].



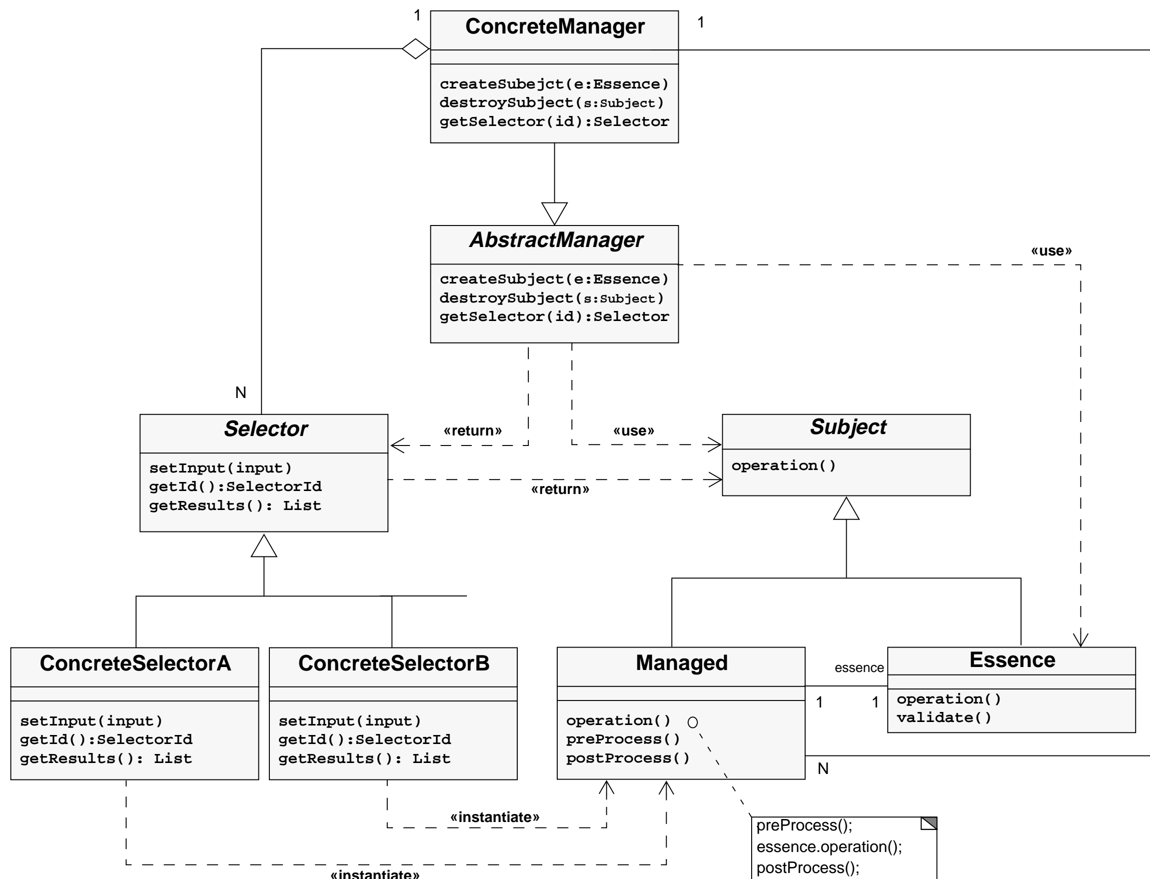
New concrete implementations of TradingManager, Portfolio, and PortfolioSelector can be provided to use COM, CORBA, or EJB as the component technology without disturbing the applications use of the business objects.

3. Applicability

Use the Abstract Manager pattern when:

- your application code must be independent of the operating environments on which it is deployed.
- your application must have the capability of choosing a specific component container or persistence repository from multiple choices, usually at deployment time but possibly at runtime or link-time as well
- you want to insulate your application against changes in the underlying component container or persistence repository
- your application uses a family of related business objects
- you want to implement persistent or componentized business objects
- each business object can exist in two states: an unmanaged state in which the object exists only in the application's primary memory and a managed state in which the object has been inserted or transformed so that it exists in the component container or persistence repository

4. Structure



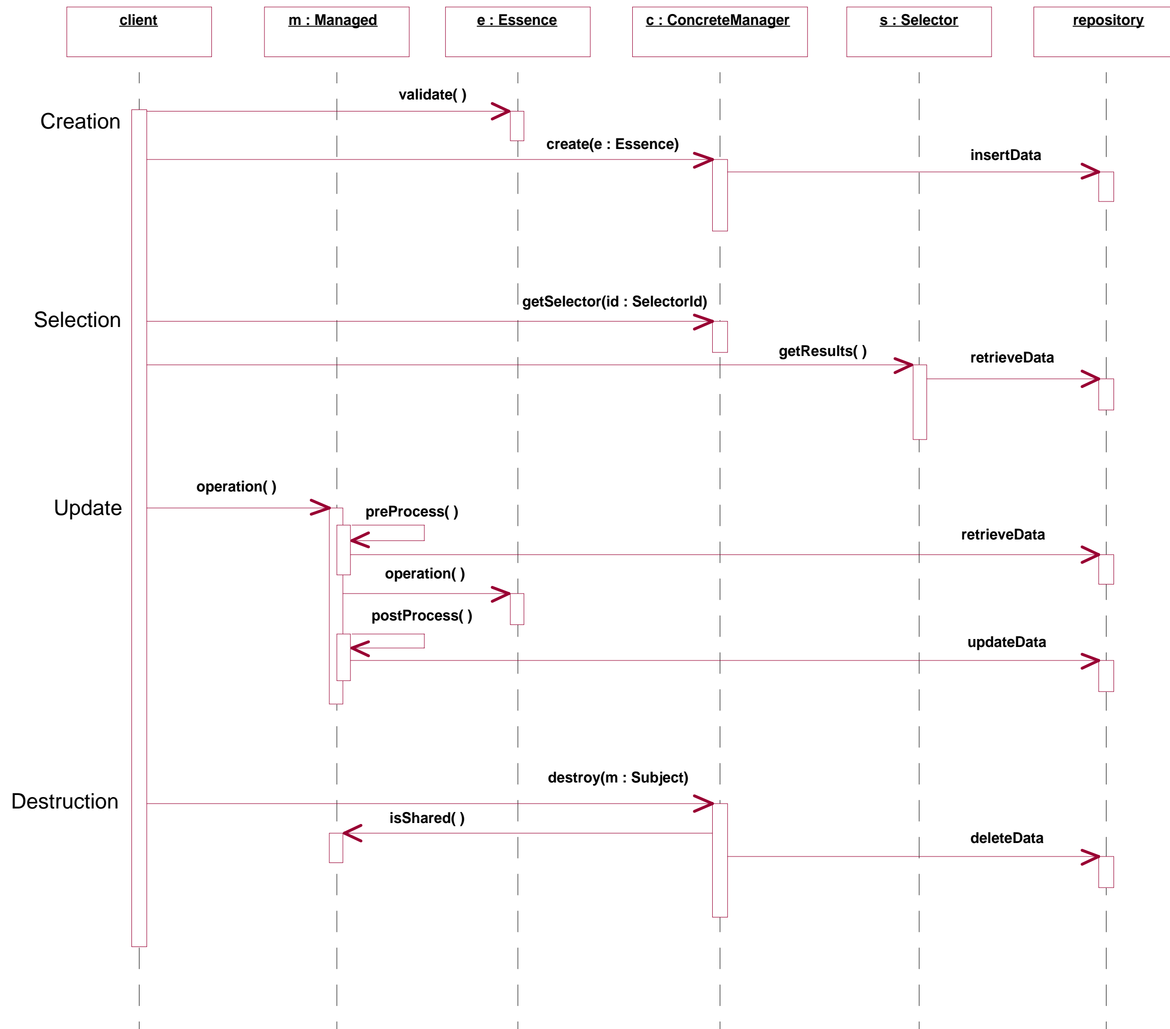
5. Participants

- AbstractManager (TradingManager)
  - declares operations for creating and destroying Managed objects
  - declares an operation for choosing Selectors in order to retrieve Managed objects
- ConcreteManager (PersistenceManager)
  - implements operations for creating and destroying Managed objects
  - maintains the set of Selectors used to retrieve Managed objects
- Subject (Portfolio, Order, Report,...)
  - declares an interface for some business logic
- Managed (PersistentPortfolio, PersistentOrder, PersistentReport,...)
  - implements the interface declared by Subject
  - provides additional functionality to maintain its "managed" state
- Essence (PortfolioEssence, OrderEssence, ReportEssence,...)
  - implements the interface declared by Subject
  - represents the unmanaged essence of the Subject
- Selector (PortfolioSelector, OrderSelector, ReportSelector,...)
  - declares an operation for identification
  - declares operations for parameterizing the selection process
  - declares an operation for obtaining the list of Subjects retrieved by the selection process
- ConcreteSelector (PortfoliosForClient, AllPortfolios,...)
  - implements the interface declared by Selector
  - provides a specific selection mechanism used in the selection process

6. Collaborations

The Abstract Manager pattern has the following collaborations:

- *Creation* - an Essence is validated and passed to the ConcreteManager via the AbstractManager interface which uses the Essence to create a Managed object by inserting the appropriate data into the ConcreteManager's external repository.
- *Selection* - client code gets a Selector from the ConcreteManager via the AbstractManager interface. The Selector is optionally configured with additional selection information provided by the client code. The Managed objects that are retrieved by the selection process are returned to the client code as a list of Subjects.
- *Update* - client code uses an Subject which references a Managed. The Managed object performs the Subject's operations and executes any additional processing to maintain its state in the external repository.
- *Destruction* - client code passes an Subject which references a Managed object to the ConcreteManager via the AbstractManager interface. The ConcreteManager removes the Managed object from the external repository and destroys the Managed object in memory.



7. Consequences

The Abstract Manager pattern has the following benefits:

- *Separates lifecycle from business logic*. Abstract Manager allows you to evolve lifecycle mechanisms independently of business domain mechanisms. This separation reduces risks associated with relying on a single component container or persistence technology. It also increases the application's flexibility to change as new component and persistence technologies become available.
- *Enables multiple kinds of component container or persistence repositories*. A single application may need to run in a variety of operating environments, each necessitating a different component container or persistence repository. Abstract Manager enables applications to choose between multiple kinds of operating environments to suit a particular deployment requirement.
- *Provides extensible selection mechanism*. Abstract Manager provides an open-ended selection mechanism that can be extended to handle practically any kind of query and retrieval required by an application. It accomplishes this with a simple and compact interface.

Abstract Manager has the following liabilities:

- *Introduces additional layers of indirection that could slow performance.* By encapsulating the lifecycle mechanisms behind generic interfaces, performance can be slowed due to the additional indirection and the mapping between the generic interfaces and the concrete repository mechanisms.
- *Complex implementations vs. easy usage.* Each ConcreteManager/ConcreteSelector/Managed implementation becomes internally more complex due to mapping the native lifecycle API to conform to the common lifecycle interfaces. However the internal complexity of the lifecycle interfaces enables the rest of the application to become simpler.

8. Implementation

The following implementation issues should be considered when using the Abstract Manager pattern:

- *Mapping Subjects to Managers.* An application may contain a large number of SUBJECTS, however not all SUBJECTS may be related. Partition the SUBJECTS into related groups by analyzing the using and containment relations between SUBJECTS. Also consider how SUBJECTS participate in transactions. Create a ABSTRACTMANAGER for each group.

  For example, a more complete description of the portfolio trading server from the Motivation section has the following subjects: Portfolio, Order, Report, Client, Company, and CommissionSchedule. Portfolios contain multiple Orders. Multiple Reports are associated with an Order. Clients belong to a Company and have an associated CommisionSchedule. These subjects can be divided into two groups: trading {Portfolio, Order, Report} and billing {Client, Company, CommissionSchedule}. The TradingManager interface is the ABSTRACTMANAGER for the trading subjects. The BillingManager is the ABSTRACTMANAGER for the billing subjects.
- *Identifying Subjects.* SUBJECTS may need to provide multiple schemes for identifying themselves. A common approach is to generate object ids (OIDs) for each new SUBJECT instance. The OID acts as a SUBJECT's unique address and can be used to represent relationships between SUBJECTS or as a selection key for use in SELECTORS. However an additional identification scheme may be provided to identify SUBJECTS based on an attribute, such as a name or timestamp, in order to compare an ESSENCE, which does not yet have an OID, to a MANAGED object.
- *Validating business objects.* Business objects need to be validated prior to becoming managed by the system. This can be accomplished by using the ESSENCE as a representation of the business object before it has been inserted into the component container or persistence repository by the CONCRETEMANAGER. The ESSENCE can provide a `validate()` method and may also provide an `isValid()` query method so the CONCRETEMANAGER can check if an ESSENCE has been validated.
- *Parameterizing Selectors.* Many concrete selection mechanisms, such as stored procedures or prepared statements, require data to be passed to them in order to configure their queries to retrieve the desired objects. The SELECTOR interface can accommodate these mechanisms by declaring various setInput(...) methods which the concrete implementations can override if necessary:

```
interface Selector
{
    void setInput(int input);
    void setInput(double input);
    void setInput(String input);
    // ...
}
```

```
class ConcreteSelectorThatTakesOneStringParam
{
    private String queryInput;
    // ...
    public  void   setInput(int input) {}
    public  void   setInput(double input) {}
    public  void   setInput(String input) { queryInput = input }
    // ...
}
```

The SELECTOR's `setInput(...)` methods should be general enough to be applicable for all CONCRETESELECTOR implementations allowing the Selector to be open for extension but closed to change [Meyer88][Martin96]. The SELECTOR interface can also provide versions of the `setInput(...)` method that can accommodate queries that require multiple inputs either by specifying the inputs' names or the inputs' positions, or by passing in a Parameter Object [Fowler99] that contains all of the inputs:

```
interface Selector
{
    void setInput(String name,double input);
    void setInput(int position,String input);
    void setInput(Parameter input);
    // ...
}
```

- *Returning Selectors that have state*. The ABSTRACTMANAGER declares methods for getting SELECTORS and returning them for use in client code. If the SELECTORS maintain state in the form of input data, there may be conflicts as each caller of the SELECTOR sets its input data overwriting the previous caller's input data. The Prototype design pattern [GHJV95] can eliminate this problem by having the CONCRETEMANAGER's `getSelector()` method return a clone of the requested SELECTOR. SELECTORS that do not have state can implement a `clone()` method that returns a reference to themselves. The CONCRETEMANAGER then becomes a Prototype-based factory [GHJV95][Vlissides98a].
- *Configuring the ConcreteManager with Selectors*. There are two approaches to configuring the CONCRETEMANAGER with its SELECTORS. The first approach is to have the CONCRETEMANAGER explicitly create and store instances of the ConcreteSelectors needed by the client. The second approach is to have the ABSTRACTMANAGER provide an interface for registering SELECTORS. The first approach has the advantage of freeing the clients from the responsibility of configuring the selectors but it has the disadvantage of requiring the CONCRETEMANAGER to be recompiled and relinked if additional CONCRETESELECTORS are added to the system. The second approach has the advantage of flexibility in that additional CONCRETESELECTORS can be added without affecting the CONCRETEMANAGER but it clutters up client code with configuration details. By combining both approaches, we can reach a balanced medium in which the default CONCRETESELECTORS are configured by the CONCRETEMANAGER and custom CONCRETESELECTORS can be added by clients.
- *Preventing dangling references*. It is important to keep a SUBJECT's external state and its internal state synchronized. A mismatch between the internal and external state can occur when the ABSTRACTMANAGER's destroy operation destroy's a Subject that is shared through multiple references. This will cause some clients to reference a Subject that no longer exists in the component container or persistence repository (i.e. its external state has been removed). The available language features will shape the solution:
  - In C++ and other languages that permit "smart pointers", we can prevent dangling references by introducing reference counting into the MANAGED objects. Each object keeps track of how many references it has. The CONCRETEMANAGER can only destroy those ob-

jects that are not shared, possibly throwing an exception to indicate an attempt to destroy a shared object. If the objects are only shared in the same process or on the same machine the reference count can exist in-memory or shared memory. However, if the objects are to be shared by multiple machines on a network, the reference counting must exist in an external repository such as a relational database or on a server-side CORBA object.

- In Java it is difficult to do “smart pointers” and reference counting. The Observer design pattern [GHJV95] provides us with an alternative. We can define an Observer similar to the Java Beans VetoableChangeListener [CL98], which broadcasts a client’s attempt to destroy a SUBJECT. If there are other clients interested in the SUBJECT they throw an exception to prevent the destroy operation from completing. Each client of the ABSTRACTMANAGER will have to implement the Observer’s interface and register with the ABSTRACTMANAGER.

9. Sample Code.

The sample Java code is taken from the example given in the motivation section. It illustrates some of the key features required for using Abstract Manager to integrate a persistence infrastructure into an application.

The TradingManager interface declares methods the creating, destroying, and selecting Portfolios and other business objects.

```
interface TradingManager
{
    void               createPortfolio(PortfolioEssence p);
    void               destroyPortfolio(Portfolio p);
    PortfolioSelector  getPortfolioSelector(String id);
    // Orders and Reports not shown ...
}
```

The PersistenceManager class implements the TradingManager as an object-to-relational mapper that stores and retrieves objects to and from a relational database.

```
class PersistenceManager
{
    private Connection   connection;
    private ObjectOutput writer;
    private List         selectors;
    // ...

    public void createPortfolio(PortfolioEssence p)
    {
        PortfolioSelector selector = getPortfolioSelector( "PORT_BY_NAME" );
        selector.setInput( p.getName() );

        if ( !p.isValid() )
            throw new CreationException( "Portfolio has not been validated." );
        if ( !selector.getResults().isEmpty() )
            throw new CreationException( "Portfolio already exists." );

        p.writeExternal( writer );
    }
```

```
    public void destroyPortfolio(Portfolio p)
    {
        if ( p.isShared() )
            throw new DestructionException( "Can't destroy a shared Portfolio." );

        Statement destroyer = connection.createStatement();
        destroyer.executeUpdate("delete from ENTRY where PORT_ID="+p.getId());
         destroyer.executeUpdate("delete from PORTFOLIO where PORT_ID="+p.getId());
        p.markAsDestroyed();
    }

    public PortfolioSelector getPortfolioSelector(String id)
        { return (PortfolioSelector)selectors.get( id ); }
    // ...
}
```

The Portfolio interface is one of the business object interfaces that is managed by the TradingManager. It provides basic operations for manipulating portfolios as used by a stock trading application.

```
interface Portfolio
{
    void    setName(String n);
    void    addEntry(Entry e);

    String  getName();
    Entry   getEntry(Symbol s);
    // ...
}
```

The PortfolioEssence class implements the Portfolio interface by providing the actual business functionality for portfolios. PortfolioEssence is the in-memory representation of portfolios. In this example, PortfolioEssence also implements the Externalizable interface [RSBMZ97] in order to serialize its state.

```
class PortfolioEssence implements Portfolio, Externalizable
{
    private String name;
    private Map    entries;

    //  ...
    public void    setName(String n) { name = n; }
    public void    addEntry(Entry e) { entries.put( e.getSymbol(),e ); }

    public String  getName() { return( name ); }
    public Entry   getEntry(Symbol s) { return( entries.get( s ) ); }
};
```

The PersistentPortfolio class implements the Portfolio interface by providing persistence management functionality that surrounds the business functionality of PortfolioEssence. PersistentPortfolio contains a PortfolioEssence (essence), an ObjectInput (reader), and an ObjectOutput (writer) [RSBMZ97] which are used to automatically refresh and persist the state of the PortfolioEssence. The concrete instantiations of reader and writer depend on the type of external repository, in this case a JDBC database.

```
class PersistentPortfolio implements Portfolio
{
    private PortfolioEssence essence;
    private ObjectInput      reader;
    private ObjectOutput     writer;
    //  ...
    public void   setName(String n) { essence.setName( n ); persist(); }
    public void   addEntry(Entry e) { essence.addEntry( e ); persist(); }
    public String getName() { refresh(); return essence.getName(); }
    public Entry  getEntry(Symbol s) { refresh(); return essence.getEntry( s ); }
    public void   refresh() { essence.readExternal( reader ); }
    public void   persist() { essence.writeExternal( writer ); }
};
```

The PortfolioSelector interface declares the methods for: identifying each PortfolioSelector instance, retrieving a List of Portfolios, and setting parameters to configure concrete queries with addition information.

```
interface PortfolioSelector
{
    String  getId();
    List    getResults();
    void    setInput(int input);
    void    setInput(double input);
    void    setInput(String input);
}
```

The PortfoliosForClient class implements the PortfolioSelector interface by constructing an SQL query to retrieve all portfolios owned by the specified client and executing that query on a JDBC Connection. The connection returns a JDBC ResultSet which is passed to a ResultReader, an implementation of the ObjectInput interface, which deserializes the query results into PortfolioEssence.

```
class PortfoliosForClient implements PortfolioSelector
{
    private String      client;
    private Connection connection;
    //  ...
    public void  setInput(String input) { client = input; }
    public String  getId() { return( "GET_CLIENT_PORTS" ); }

    public List getResults()
    {
        Statement    stmt   = connection.createStatement();
        String       query  = "SELECT * FROM PORTFOLIO WHERE CLIENT=" + client;
        ResultSet    results = stmt.executeQuery( query );
        ResultReader reader = new ResultReader( results );
        List         output = new ArrayList();
        PortfolioEssence essence = new PortfolioEssence();

        while ( reader.hasMore() )
        {
            essence.readExternal( reader );
            output.add( new PersistentPortfolio( essence ) );
        }
        return( output );
    }
}
```

The following is an excerpt of code that makes use of the various participants of the Abstract Manager pattern. A list of portfolios are loaded into the application as essences. They are validated and then used to create persistent portfolios. Finally a list of portfolios belonging to a single client are retrieved through the selector.

```
TradingManager manager  = new PersistenceManager();
List                essences = getPortfoliosToCreate();
PortfolioSelector    selector   = manager.getPortfolioSelector( "GET_CLIENT_PORTS" );
String              clientName = "Portfolio Management Inc.";
List                portfoliosForClient = null;

try
{
    for (Iterator terator i = essences.getIterator();i.hasNext();)
    {
        PortfolioEssence essence = (PortfolioEssence)i.next();
        try { essence.validate(); }
        catch (ValidationException e) { e.printStackTrace(); }
        manager.createPortfolio( essence );
    }
}
selector.setInput( clientName );
portfoliosForClient = selector.getResults();
```

10. Known Uses

Enterprise Java Beans [MH99][Monson00] use the Abstract Manager pattern to achieve portability across EJB component container implementations, such as IBM's Websphere and BEA's WebLogic application servers. An enterprise bean provides client code with a Home interface which acts as the ABSTRACTMANAGER providing lifecycle methods for creating, destroying, and finding SUBJECTS. SUBJECTS implement a Remote interface providing business methods. You can structure the Home interface's create methods to accept ESSENCE objects as the input for creating enterprise beans. The Home interface is implemented by an automatically generated component container class that manages creation, selection, and destruction of Remote objects the same as a CONCRETEMANAGER. The Remote interface is implemented by two classes, an automatically generated proxy that fulfils the MANAGED role and the actual bean implementation that provides the business functionality. EJBs are portable between vendor implementations because applications only depend on the Home and Remote interfaces. When an application changes EJB component container implementations, the component container related classes (CONCRETEMANAGER and MANAGED) are automatically generated for the new component container.

CORBA Object Factories [HV99] use the Abstract Manager pattern in much the same way as Enterprise Java Beans, to achieve portability across implementations. Object factories in CORBA provide lifecycle methods for creating, destroying and finding CORBA objects. An object factory is represented by an IDL interface (AbstractManager) on the client side and a class implementation on the server side. Object factories return references to user defined CORBA business objects (Subject). The IDL compiler provided by a CORBA implementation will automatically generate proxy classes that implement the Object factory and user defined CORBA business object interfaces (CONCRETEMANAGER and MANAGED).

11. Related Patterns

Abstract Manager is a compound design pattern [Riehle97][Vlissides98b] combining the Manager [Sommerlad97], Proxy, Strategy [GHJV95], and (a variation of) Essence [Carlson98] design patterns. The Manager design pattern serves as the center for drawing the other patterns together to form a cohesive whole. The Manager and Proxy design patterns overlap with the $\text{Subject}_{\text{Manager}}$ and the $\text{Subject}_{\text{Proxy}}$ participants combining into the $\text{Subject}_{\text{Abstract Manager}}$ participant. The Manager and Strategy design patterns overlap with the $\text{Manager}_{\text{Manager}}$ and the $\text{Context}_{\text{Strategy}}$ participants combining into the $\text{ConcreteManager}_{\text{Abstract Manager}}$ participant. The Proxy and Essence design patterns overlap with the $\text{RealSubject}_{\text{Proxy}}$ and $\text{Essence}_{\text{Essence}}$ participants combining into the $\text{Essence}_{\text{Abstract Manager}}$ participant. The following shows the correspondence between participants:

$\text{AbstractManager}_{\text{Abstract Manager}}$	$\rightarrow \text{Manager}_{\text{Manager}}$
$\text{ConcreteManager}_{\text{Abstract Manager}}$	$\rightarrow \text{Manager}_{\text{Manager}}$, $\text{Context}_{\text{Strategy}}$
$\text{Subject}_{\text{Abstract Manager}}$	$\rightarrow \text{Subject}_{\text{Manager}}$, $\text{Subject}_{\text{Proxy}}$
$\text{Managed}_{\text{Abstract Manager}}$	$\rightarrow \text{Proxy}_{\text{Proxy}}$
$\text{Essence}_{\text{Abstract Manager}}$	$\rightarrow \text{RealSubject}_{\text{Proxy}}$, $\text{Essence}_{\text{Essence}}$
$\text{Selector}_{\text{Abstract Manager}}$	$\rightarrow \text{Strategy}_{\text{Strategy}}$
$\text{ConcreteSelector}_{\text{Abstract Manager}}$	$\rightarrow \text{ConcreteStrategy}_{\text{Strategy}}$

Abstract Manager is closely related to many of the patterns described in “Connecting Business Object to Relational Databases” [YJW98]. Abstract Manager is one way of implementing the CRUD pattern described in [YJW98]. Abstract Manager is more general than the [YJW98] patterns because Abstract Manager can encapsulate component containers such as EJB in addition to persistence repositories such as relational databases.

The Observer design pattern [GHJV95] can be used to prevent dangling references if the implementation language does not allow “smart pointers”.

The Serializer design pattern [RSBMZ97] can be used internally by the MANAGED participant to implement persistence.

The MANAGED participant can use the Adapter design pattern [GHJV95] to adapt component interfaces like those produced by CORBA or EJB to conform to the interface provided by the SUBJECT participant.

The Visitor design pattern [GHJV95] can be used to validate a complex ESSENCE by applying validation logic to each element of the ESSENCE.

The Prototype design pattern [GHJV95] can be used to return clones of SELECTORS, avoiding problems with overwriting SELECTOR state.

Transaction related patterns [YJW98][Lea00][KC97]] can be used to enable SUBJECTS to participate in transactions if necessary.

Acknowledgments

Thanks go to Jim Stern and Alex Shindich for reviewing the first draft of this pattern, and thanks go to my shepherd, Wolfgang Keller, for providing many iterations of helpful feedback.

References

Carlson98 Carlson, Andy. Essence. Pattern Languages of Program Design 4, pp 33-40. Addison-Wesley. 2000.

CL98 Chan, Patrick and Rosanna Lee. The Java Class Libraries Second Edition, Volume 2. pp 1505-1509, Addison-Wesley. 1998.

Fowler99 Refactoring: Improving the Design of Existing Code. pp 295-299, Addison-Wesley. 1999.

GHJV95 Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.

HV99 Henning, Michi and Steve Vinoski. Advanced CORBA Programming in C++. Addison-Wesley. 1999.

KC97 Keller, Wolfgang and Jens Coldewey. Accessing Relational Databases. Pattern Languages of Program Design 3, pp 313-343. Addison-Wesley. 1998.

Lea00 Lea, Doug. Concurrent Programming in Java Second Edition, Design Principles and Patterns. pp 249-260, Addison-Wesley. 2000.

Martin96 Martin, Robert. Open Closed Principle. C++ Report January 1996. SIGS Publications. 1996. http://www.objectmentor.com/publications/ocp.pdf

MH99 Matena, Vlada and Mark Harper. Enterprise Java Beans Specification v1.1. pp 41-47, 88-94, 114-115. Sun Microsystems, Inc. 1999.

Meyer88 Meyer, Bertrand. Object-Oriented Software Construction 1st Edition. pp 23-25. Prentice Hall International Ltd. 1988.

Monson00 Monson-Haefel, Richard. Enterprise Java Beans 2nd Edition. pp 23-29, 135-147, 161-165. O'Reilly, Inc. 2000.

Riehle97 Riehle, Dirk. Composite Design Patterns In Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97). pp 218-228. ACM Press, 1997.
http://www.riehle.org/papers/1997/oopsla-1997.html

RSBMZ97 Riehle, Dirk, Wolf Siberski, Dirk Baumer, Daniel Megert, and Heinz Zullighoven. Serializer. Pattern Languages of Program Design 3, pp 293-312. Addison-Wesley. 1998.
http://www.riehle.org/papers/1996/plop-1996-serializer.html

Sommerlad97 Sommerlad, Peter. Manager. Pattern Languages of Program Design 3, pp 19-28. Addison-Wesley. 1998.

Vlissides98a Vlissides, John. Pluggable Factory Parts I & II. C++ Report November 1998, February 1999. SIGS Publications.
http://www.research.ibm.com/designpatterns/pubs/ph-nov-dec98.pdf
http://www.research.ibm.com/designpatterns/pubs/ph-feb99.pdf

Vlissides98b Vlissides, John. Composite Design Patterns (They Arent' What You Think). C++ Report June 1998, SIGS Publications.
http://www.research.ibm.com/designpatterns/pubs/ph-jun98.pdf

YJW98 Yoder, Joseph, Ralph Johnson, and Quince Wilson. Connecting Business Objects to Relational Databases. Pattern Languages of Programming Conference Proceedings, 1998. http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P51.pdf