Manager

The Manager design pattern puts funcitonality that considers all objects of a class into a separate managing object. This separation allows the independent variation of manager functionality and its reuse for different object classes.

Authors Peter Sommerlad and Frank Buschmann,
Siemens AG, Dept. ZFE T SE 2, D-81730 Munich
peter.sommerlad@zfe.siemens.de
+49-89-636-48148, FAX: +49-89-636-40898

Example Suppose you are developing an object-oriented application, e.g. a library information system, that has to use a relational database for storing and retrieving its domain objects, let's say books. Assume each domain object is uniquely identified by a key, for example, the ISBN of a book. We want to 'optimize' database access and only instantiate objects from the database that are currently referenced by the program. In addition, we need to retain object identity versus table contents, that means we do not want to instantiate two objects from a single table row.

In C++ these requirements could be accomplished by introducing static class members.

```
class Book {
    // ... many things left out
public:
    Book(String ISBN, String Authors, String Title);
    String getAuthors() const;
    String getISBN() const;
    String getTitle() const;
    String getPublisher() const;
    const JpegImage *getCover() const;
    HTMLpage *getTableOfContents() const;
// static members for retrieving books
    static Book *search(String isbn);
    static List<Book*> findBooksofAuthor(String author);
};
```

Now consider we create a subclass of class book for pattern books. This subclass gets additional operations, for example, get a list of all patterns described in the pattern book. Adding such a derived class can break the code of the static members of the base class, forcing us to recompile all dependend files. In addition if we want to extend the access functionality, for example by searching for books by a specific publisher or with a specific pattern we need to extend the base-class.

Context Object handling

Problem When designing a class of business (or domain) objects, we often face the challenge of dealing with all instances of this class in a homogeneous way. For example, searching instances with a specific property, requires access to a collection of all instances. Using class variables and class methods for this purpose may restrict flexibility too much. It is not easy to subclass the object class independent of the way the objects are managed. In C++ we have to recompile all dependent files, when we vary the way objects are managed, for example, by substituting the collection type.

Another problem are persistent objects that reside in a database. To delegate work to such objects we have to create an in memory represenative. Especially when using a relational database instead of an object-oriented database this requires addional work with several points of possible failures.

To summarize we often want to control the life-cylce of all objects of a class in a homogenous way. That leads to a third situation when we want to get rid of in-memory copies of persistent objects. Then we need to keep track of objects used.

Often several classes of an application share such requirements. It would be nice if code managing these objects is also shared and not copied. In particular you want to address the following *forces*:

- All objects of a class should be accessible as a whole.
- Variation of the objects' implementation should not affect the way they are managed.
- Additional services related with the handling of the domain objects, like storing them in a database, should be easy to add.
- Several classes require almost identical control of their objects.

Solution Introduce a separate component, the *manager*, that treats the collection of managed objects as a whole. The manager component deals with issues related to accessing, creating, or destroying the managed objects.

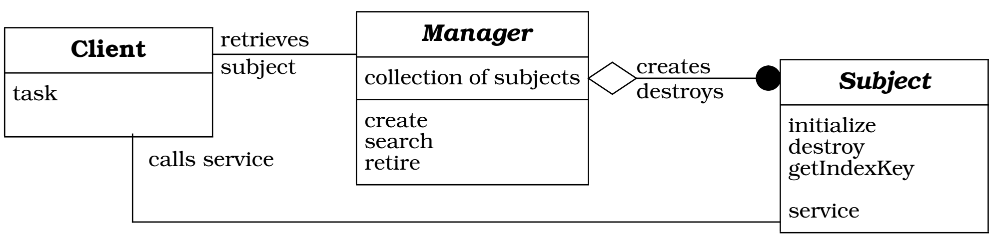
Clients requiring a specific managed object retrieve a reference to it from the manager. From thereon all interaction is done directly with the domain object. If no longer needed a client may return the domain object to the manager.

The managed objects perform their task on behalf of the client. We refer to them as *subjects* in this pattern.

Structure The client is coded againts the subject's interface. However to obtain an object of this class it needs to exploit the manager object.

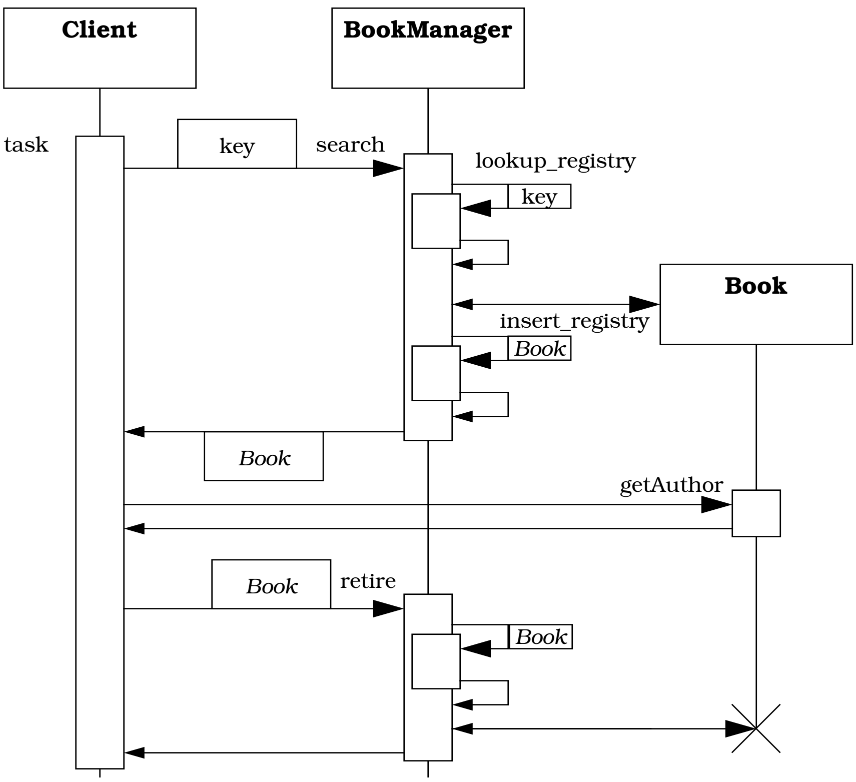
The manager is implemented as a Singleton. It contains some collection holding references to all instantiated subjects. The manager is the only object creating and destroying subjects. Typical managing functionality includes searching subjects for a specific key. Since the manager object is not directly included within the subjects class it is possible to subclass and extend the manager independently.

The subject class implements the domain objects in this pattern. It provides any domain services required by the clients. If multiple subject and related manager classes exist in a system all subject classes may implement an interface requireed by the managers. An example is the `getIndexValue` to obtain the key of a subject for searching.



Dynamics The following scenario depicts a typical situation when a manager pattern is used.

- First the client asks the single manager object for a subject identified by some key (the ISBN for our book example).
- The manager searches its registry of all subject objects if the required one is already available. In this scenario it is not.
- The new subject (a book) is created.
- Before the manager returns the newly created subject, it inserts it into the registry for later reference.
- Now the client is ready to perform the required functionality with the subject (book->getAuthor()).
- After using the subject for some time the client returns responsibility to the manger, which in this case decides to delete the subject.



Implementation To implement the Manager pattern perform the following steps

1 Identify the ‘class’-services required for your objects. In our library information system we need access for searching books. Since books are stored in a database also some initialization code is needed. So the following functions might be needed:

```
void initialize(Database *db);
Book *search(String ISBN);
List<Book*> findBooksofAuthor(String author);
```

2 Define the interface required for the manager component. We take the required functionality from step 1 and put that into the manager class. Very often it is useful to provide the client with an iterator through all managed objects, here the Factory Method [GHJV95] `makeBookIterator()` is included. Use Singleton pattern[GHJV95] to give access to the single manager object. The resulting code looks like:

```
class BookManager {
     Map<String,Book*> loadedBooks;
     BookManager();
     friend class Book; // for accessing constructor only
     static BookManager *theManager; // singleton
public:
     // implement singleton:
     static BookManager *getInstance() ;
     Book *search(String ISBN);
     List<Book*> findBooksofAuthor(String author);
     Iterator<Book*> makeBookIterator();
     void retire(Book *oldbookobject);
};
```

Implementing Singleton as shown in [GHJV95] delivers one drawback, because the C++ static member variables require more than a single point of reference. That means, in addition to declaring the `static BookManager *theManager` we also need to define the corresponding variable in some implementation file as

```
BookManager *BookManager::theManager = 0;
```

For syntactical convenience see the Manager Template variant, where we give an example for implementing manager access via the subject class. However, this requires bi-directional ‘friendship’ in C++.

3 Design and implement the domain object class (the subject). For our library information system we get the class definition for books:

```
class Book {
     // ... some things left out
     String isbn,authors,title,description,publisher;
     // books are added by the manager only
     Book(String ISBN, String Authors, String Title);
     virtual ~Book();
     friend class BookManager;
public:
     String getAuthors() const;
     String getISBN() const;
     String getTitle() const;
     String getPublisher() const;
     const JpegImage *getCover() const;
     HTMLpage *getTableOfContents() const;
};
```

4 Implement the manager component. If subject references are shared, the manager may not blindly delete objects that are retired by one client, another client might still use the reference. In C++ this can be overcome by using the Counted Pointer idiom and returning handles instead of pointers to the clients.

If you want to access subjects in a database, look at the Proxy pattern in [GHJV95] and [BMRSS96], as well as into the pattern language Crossing Chasms [BW96] if it is a RDBMS.

5 Design and implement the client components using the manager and domain objects. The clients obtains a subject reference via the manager and thereon uses the subject reference directly. This situation relates to the Client-Dispatcher-Server design pattern [BMRSS96] where the dispatcher establishes a communication channel between client and server that is used then without accessing the dispatcher. Some example code might look like:

```
Book *abook =
     BookManager::getInstance()->search("0471958697");
if (abook)
cout << "You should read" << abook->getTitle()
     << " it was written by " << abook->getAuthors() ;
```

Variants

Manager Template. If several classes require similar management, like accessing objects in a relational database, it may be feasible to extract a class template for code sharing among manager classes. Individual behavior can be added by subclassing the instantiated template classes. To relieve clients from specifying manager access using the template syntax in many places and to give only a single point of reference for the singleton the following example code uses a slightly different implementation technique for singleton. Such a Manager Template class for objects of type `T` that are uniquely identified by an index code of type `INDEXTYPE` might look like the following code:

```
template <class T, class INDEXTYPE>
class TManager
{
     // this class requires
     //            T::T(isream&)
     // INDEXTYPE  T::getCode()
     // void       T::finalize(ostream&)
     // and a destructor for T
     friend class T; // for T::getManager()
     // TManager<T,INDEXTYPE> created by T::getManager()
protected:
     TManager ();
     virtual ~TManager (); // needed if subclassed
     virtual T* add(T *m);
     virtual T* remove(INDEXTYPE code);
     virtual T* remove(T *m);
public:
// read T object from stream, as text representation
     virtual T* createNew(istream &in=cin);
// read initial data from file:
     virtual void initialize(istream & in ) ;
// for testing store data to file:
     virtual void finalize(ostream & out);

// TManager is for indexed access:
     virtual T* search(INDEXTYPE code);// look up a T
     virtual void retire(T *t);
protected:
     Map<INDEXTYPE,T*> code2index ;  // contains all Ts
};
```

Our Book class from the example may be recoded in the following way:

```
class Book {
     // ... some things left out
     String isbn,authors,title,description,publisher;
     // books are added by the manager only
     Book(String ISBN, String Authors, String Title);
     virtual ~Book();
     friend class TManager<Book,String>;
```

```
public:
     String getCode() const { return getISBN(); }
     String getAuthors() const;
     String getISBN() const;
     String getTitle() const;
     String getPublisher() const;
     const JpegImage *getCover() const;
     HTMLpage *getTableOfContents() const;
// manager access:
     static TManager<Book,String>* getMgr();
};

TManager<Book,String>* Book::getMgr(){
     static TManager<Book,String>* theMgr = 0;
     if (! theMgr) {
          theMgr = new TManager<Book,String>;
          theMgr->initialize("allbooks");
     }
     return theMgr;
}
// adapt createNew to parse book information
Book *TManager<Book,String>::createNew(istream &in){
     String isbn,authors,title;
     in >> isbn >> authors >> title;
     // details omitted ! ...
     return new Book(isbn, authors, title);
```

The client code using books now might read:

```
Book *abook =
     Book::getMgr()->search("0201895277");
if (abook)
cout << "Also a nice book about patterns:"
     << abook->getTitle();
```

Class Methods. In languages with separate class objects, or languages that are not statically typed (like Smalltalk) it may be sufficient to implement the manager functionality with class methods and class variables. There a change of the manager functionality does not imply a recompilation of all subject's subclasses.

Known Uses Flyweight [GHJV95], Command Processor, View Handler [BMRSS96] are well-known patterns that use the Manager pattern as a part of their solution.

Consequences The Manager pattern implies the following **benefits**:

- A typical manager allows to iterate over all managed objects. This can be used to obtain summary information, calculate statistics and update the database rows with information from the transient objects when closing an application.
- It allows independent variation of the 'managing' functionality from the managed objects. For example, it is possible to modify the access layer to the database within the manager class without changing the subject class at all.
- By subclassing a manager's class and exchanging the singleton instance access method it is easy to substitute manager functionality with minimal impact on existing code.
- The Manger Template variant or an equivalent solution alows to reuse the manager code for different, even unrelated subject classes.
- A manager can take the role of a factory to create objects from some subject class hierarchy on demand of the client.

However, the Manager pattern also has some **liabilities**:

- It is easy to misuse the Manager pattern to break encapsulation of the subject classes.
- It may be hard to decide how to divide functionality between manager and subject classes.
- If a single manager object controls a whole hierarchy of subject classes, you will need some kind of meta-information for the managed objects to associate an object's class when instantiating objects. See the Reflection pattern [BMRSS96] for detailed discussions.

See Also The distribution patterns Broker and Client-Dispatcher-Server [BMRSS96] have some similarities with the Manager pattern because they provide access to a collection of server objects on behalf of clients. But instead of being responsible for the complete life-cycle of the servers, broker and dispatcher are only responsible for registered and thus already existing servers.

Credits Many thanks to the shepherd Robert Martin and to Erich Gamma and the participants of the writers workshop at the Componentware Users Conference 1996 in Munich.

References

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[BMRSS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-oriented Software Architecture–A System of Patterns*, J. Wiley & Sons, 1996

[BW96] K. Brown, B. G. Whitenack: *Crossing Chasms: A Pattern Language for Object-RDBMS Integration*, in *Pattern Languages of Program Design 2*, Addison-Wesley, 1996