Manager

The Manager design pattern encapsulates management of a class' objects into a separate manager object. This allows variation of management functionality independent of the class and the manager's reuse for different classes.

Author

Peter Sommerlad
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, such as a library information system. This system deals with its domain objects, such as books and library users. All books in the library are represented as instances of a Book class. Each domain object is identified and accessed by a unique key attribute, for example, the ISBN of a book.

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

```
class Book {
    // ... many things left out
    static Map<String,Book*> allBooks;
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);
};
```

If we want to substitute the collection holding all books by another type we will need to recompile all client code using the Book class. If we want to implement other search functions, for example searching for books of a specific publisher or by title, we need to extend and modify the Book class again, forcing recompiles of all clients, even if they do not use the new functionality.

A class representing the library users will copy a lot of the managing code to provide access to user objects by their identification.

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 vary the way the objects are managed without changing the class itself. In C++ we have to recompile all files depending on the class, even if they are not affected by the management change itself.

Another problem are objects we want to create on demand, but share among clients afterwards. An example are font description objects in a GUI framework. Creating such a font object is typically an expensive operation and because of the large number of fonts it is not possible to create all font objects up front.

A similar situation arises when we have to deal with persistent objects that reside in a database. To delegate work to such objects we have to create an in memory representative. Especially when using a relational database instead of an object-oriented database this requires additional work with several points of possible failures.

To summarize we often want to control the life-cycle 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.

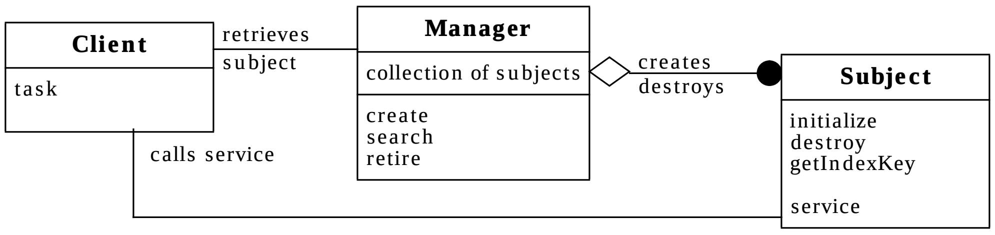
Clients requiring a specific managed object retrieve a reference to it from the manager. From thereon all interaction is done directly with the domain object. If no longer needed a client may return the domain object to the manager.

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 against the subject's interface. However to obtain an object of this class it needs to exploit the manager object.

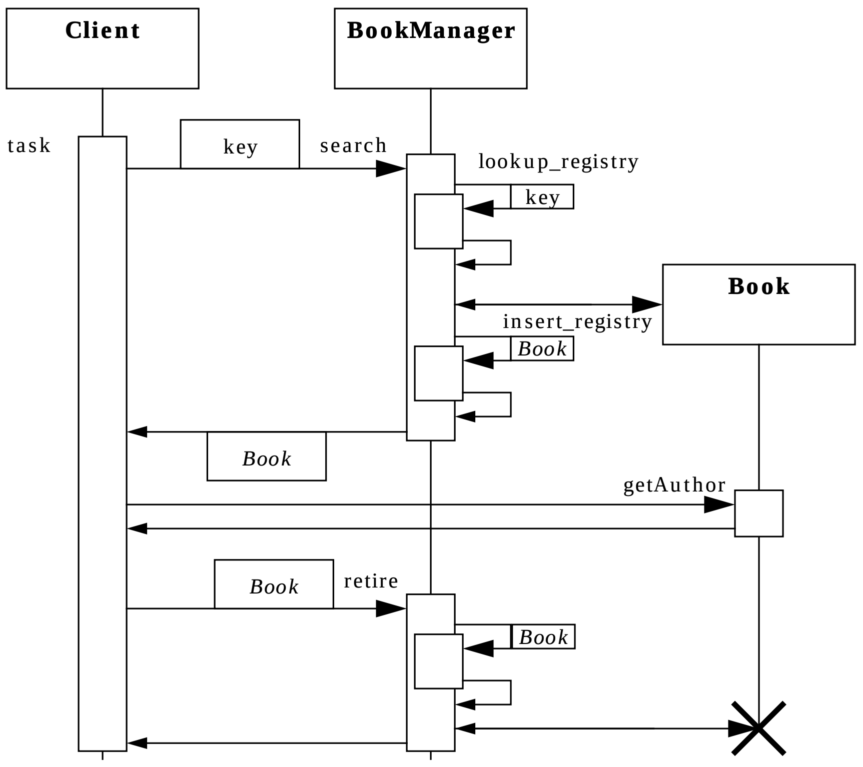
The *manager* is the only object that creates and destroys subjects. It keeps track of managed subjects in a collection of references to the subjects. Typical managing functionality includes searching subjects for a specific key. Since the manager object is not directly included within the subjects class it is possible to modify or subclass and extend the manager independently.

The *subject* class implements the domain objects in this pattern. It provides the domain services required by clients.



Dynamics The following scenario shows the library example implemented with the manager pattern. The scenario demonstrates how a subject is requested, used and retired by the client.

- First the client asks the single manager object for a subject identified by some key (the ISBN in 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 management services required for your subjects.* The kind of services you need strongly depend on the nature of your application. It is common to provide functions for retrieving a subject, for example by searching by a key, creating a new subject, iteration [GHJV95] over all available subjects, and subject retirement that may result in the deletion of subjects.

➥ 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);
Iterator<Book*> makeBookIterator();
```

❑

2 *Define the interface of the manager component.* We take the required functionality from step 1 and put that into the manager class. Typically the manager acts as a Factory [GHJV95] for subject objects. If you want to manage instances of a subject class hierarchy the factory function of the manager needs to acquire information which subclass to instantiate. This information can either be given by a client as a parameter, or calculated upon creation, for example by reading a type information from a database.

3 If you have to ensure that only one manager object exists you can apply the Singleton pattern[GHJV95].

➥ Access to the iterator over all books the Factory Method [GHJV95] `makeBookIterator()` is included. 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);
    };
    BookManager *BookManager::theManager = 0; // singleton
```
❏

4 *Design and implement the subject class.* This step is dependent on your concrete application. However, if you have to consult an external medium like a database to create a subject instance you have to decide where to put this code. It can be either in subject's constructor or in the manager class. If the manager is responsible for accessing external media you may be able to implement the subject class independent of where and how the instance information is stored.

➥ 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;
    };
```
❏

5 *Implement the manager component.* After the subject class is finished you are able to implement the manager's functionality. You need to decide what kind of collection to use for holding the subject references. Eventually several dictionaries are needed if you want to search subjects for different aspects.

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 [BMRSS96] and returning handles instead of pointers to the clients.

If you make the manager responsible for accessing external information like a database on subject creation it becomes feasible to exchange that information source at will. So you are able to implement reading instances from a file and test your system before you invest in a full-fledged database management system. However using a database raises more topics like transaction control that can be discussed here.

6 *Design and implement client components using the manager and subject.* 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.

➥ Our 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 your system contains multiple classes that may take the subject role in the manager pattern it can be a good idea to extract all common manager code into a single template manager class. This is possible if all subject classes implement a common interface required by the managers. An example for such a method is `getIndexValue()` to obtain the key of a subject for searching.

Individual management behavior can be added by subclassing the instantiated manager template class.

➥ In our example such a Manager Template class parameters subject type `T` and index type `INDEXTYPE` can be used to manage not only the books in the library but also the users. The code might look as follows.

```
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
};
```

To relieve clients from specifying template syntax for individual manager in many places and to give only a single point of reference for the Singleton the example code uses a slightly different implementation technique for Singleton. It provides the subject class with a function `getMgr()` for accessing its manager object.

Our Book class from the example can be modified 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 can 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. However the principle of separation of concerns might be applied get to a solution according to the Manager pattern.

Known Uses Flyweight [GHJV95], Command Processor, View Handler [BMRSS96] are well-known patterns that use the Manager pattern as a part of their solution. Each proposes a central component like flyweight manager, command processor or the view handler that keeps track of existing subjects which are flyweight instances, command objects or view objects respectively.

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

- *Iteration over 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.
- *Independent variation of management functionality*. For example, it is possible to modify the access layer to the database within the manager class without changing the subject class at all.
- *Substitution of management functionality on demand*. 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.
- *Reuse of management code for different subjects*. The Manger Template variant allows the reuse of management code for different, even unrelated subject classes.

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

- *Misuse*. It is easy to misuse the Manager pattern to break encapsulation of the subject classes.
- *Splitting functionality between manager and subject classes*. It may be hard to decide how to divide functionality between manager and subject classes. For example, where to deal with database access and error handling.
- *Meta information required*. If a single manager object controls a hierarchy of subject classes, you usually 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 workshops at PLoP '96 and 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