Named Objects

The Named Object design pattern is useful for managing a group of type-related components with a given configuration that are used as singletons or prototypes in your program. It provides a means to transparently differentiate objects not only by their class membership, but by different configuration data. It can be viewn as an extension of the Manager pattern [PLOPD3].

Authors

Marcel Rüedi and Peter Sommerlad
IFA Informatik, Zurich, +41-1-273 4646
rueedi@ifa.ch, sommerlad@ifa.ch

Remarks: This pattern is part of a larger collection of patterns it refers to. Due to limited resources and because its main author Marcel doesn´t yet have written many patterns it is proposed to a Writing workshop at EuroPLoP 98. One or more of the other patterns can be found among the patterns that are ´writer´s workshopped´.
Additional Remarks: We are still looking for a good name for this pattern. We tried several longer ones and none gave us a good feeling.

Example

Suppose we are developing a flexible framework to produce HTML pages dynamicly. Part of the data to be displayed comes from legacy mainframe applications other parts are gathered from local SQL-Databases. One of the most important parts in the framework is the page class. It is responsible for the gathering of the data and the production of the output message. Because the layout of WWW-pages differ with a broad variety and is likely to undergo changes often, the page class should keep the layout information within external configuration data. The configuration data defines fine grained part that are commonly used, like Headings, Images, Tables or other HTML tags. On rendering the HTML response the page interprets the configuration data to produce the message piece by piece. To change the layout of a page or to produce differently layouted page it is only necessary to change the configuration.
On the other side the abstract base class provides hooks for retrieving data from external sources like databases or other servers and to prepare it for the final rendering. By subclassing we can develope pages that implement concrete access to specific sources like a SQLpage, a Hostpage, a Mailpage and so on. The actualy request may also be parameterized using configuration data to reuse the same classes for different queries. To build a web application we have to implement the needed subclasses and configure them to produce the different pages that can be requested by the clients webbrowser.
In addition, we want specify Renderers indirectly and reuse the parameter specification. Then we do not need to give parameters literally every time a Renderer is called.

Context

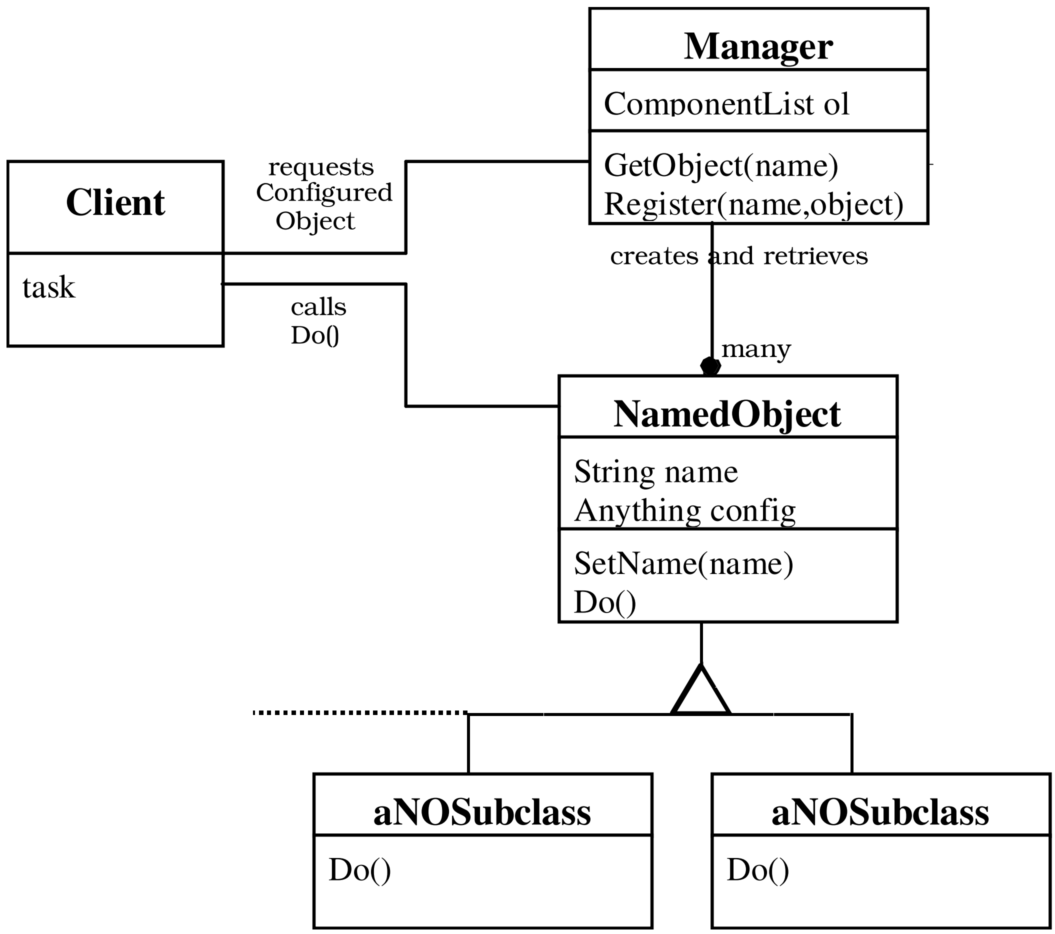
Consider you are developing a flexible piece of software, to be extended in the future. You provide a framework using components which can be configured to produce different behavior. Therefore, extensibility is not only gained by subclassing but also by variing the objects configuration.
You are using a language environment without appropriate Reflection infrastructure.

Problem

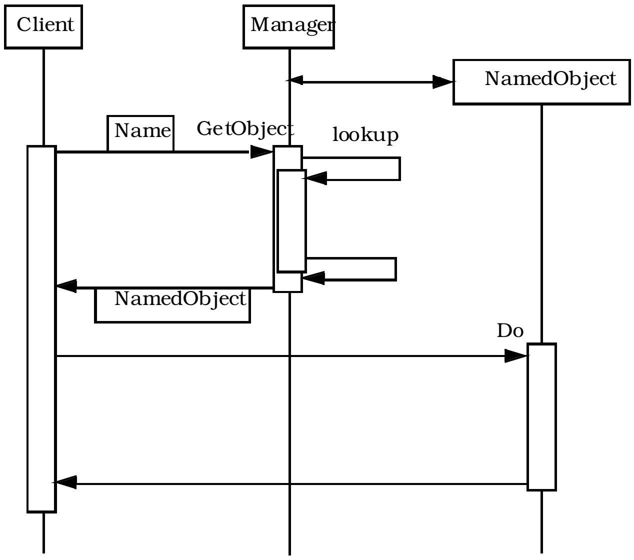
To provide such components to your framework clients the following problems arise:
You need to instantiate your framework components and associate the configuration data with them. However, to provide easy editing of configuration data, this is usually stored in ascii-text files. Using predefined names built into your framework limits the use of configuration data to the framework classes and does not provide its easy use for to-be-defined extension subclasses. How do you create objects with a certain configuration?
On the other side it may be still necessary for the frameworks clients to derive subclasses of your components. How do you create the components using the correct type?
Once the components are created and introduced into the system, they have to be accessed by framework code and by client code. Because the components are a major part of the application and need to be accessed often, the configuration files should only be read once while installing the system

and the clients should either share this instance or work with a copy of it. How can the components be accessed without caring of the concrete type and the configuration?
In particular you want to address the following forces:
You want to initialize a set of objects with different configuration data.
You want to implement a Factory Method or accessor method that instantiates or accesses these objects transparently, regardless which class they belong to.
The configured objects should be known and accessed under a given name.
You want to extend the system with "derived" objects, that share a common class but have different names and configuration data.
During evolution of the system it should be possible to introduce a new class for an already used object without changing client implementation.
The access to the configured objects should be as efficient as possible
The built-in type system of the language is not flexible enough to support these issues directly.
Solution
Name (Baptize?) the objects you are creating. As soon as the object receives its name, the configuration is loaded from the file specified by the name, most straightforwardely the name matches the filename. The object is also now publicly known under this name.
To address the other problems a Manager component is needed. It takes care of the components creation as an instance of the desired Class and setting the proper name. To perform this task, the manager has to be informed which classes exists.
After creation the Manager provides access to the Named Objects. It maps objects names to instances. If a client needs a component it has to request it from the manager using the components name. Depending on the nature of the named object and the systems requirements the manager is free to give the clients a handle to the requested object or to provide the client with a copy of the object by cloning the registered instance.

Structure



Dynamics



Implementation

To implement the Named Object pattern perform the following steps

1. Define a abstract base class for the Named Object interface. It is a good idea to use an Anything to store the configuration data.

```
class NamedObject {
public:
   virtual ~NamedObject;
   virtual void SetName(String Name);
   // This method sets the name of the object.
   // Additionally this will be a good place to load the
   // configuration for this object according to name.
   virtual NamedObject* New()
   // This is a Factory method to create a plain vanilla
   // instance of the object.
   virtual void Do()
   // This is the method called by clients.
   // It performs a task depending on the objects
   // configuration.
private:
   virtual void _loadConfig(String Name);
   String fName;
   Anything fConfig;
};

void NamedObject::SetName(String Name)
{
   fName = Name;
   _loadConfig(Name);
}
void NamedObject* NamedObject::New()
{
   return new NamedObject();
}
void _loadConfig(String Name)
{
   filename = Name+".cfg";
// arbitraily choosen extension for configuration files
   fstream is(filename);
   fConfig.LoadFromStream(is);
}
```

2. The manager defines methods to register and retreive the configured objects using a name. Choose an appropiate container object to store the instances. A Hashtable would normally make a good choice.

```
void Manager::Register(String Name, NamedObject* no)
```

```
{
   // load the configuration data
   no->SetName(Name);
   // store the object in a appropriate container
   _table.put ( Name, no);
}

NamedObject* Manager::GetObject(String Name)
{
   NamedObject* co =
         (NamedObject*)_table.get(Name);
   return co;
}
```

3. Implement the registration of the classes to the manager during system start-up. One Way to do this is to provide a Installer class for NamedObject. You need to instantiate it for each subclass that has to be stored by the manager.

```
class NamedObjectInstaller {
public
   NamedObjectInstaller();
};
NamedObjectInstaller::NamedObjectInstaller(String Name,
NamedObject* co)
{
   Manager::GetInstance()->register(Name, co);
}
```

With C++ it is usefull to write a Register Makro to minimize typing effort. The makro creates a static Installer object that registers a instance of the class under the class name.

```
#define RegisterConfiguredObject(name) \
static NamedObjectInstaller \
name##Registerer ("name", new name())
```

To register a subclass call the makro within the implementation file :

```
RegisterConfiguredObject(SubclassA);
```

Note: With a language like Java this part can be integrated direcrly into the manager install method, because of the language class loading feature.

4. Define the way how the manager installs the named objects. A simple way to do is to list for each class all configured objects that are instances of the class. On installing the manager read the class-object definition and register the components as instances of the given class.

```
void  Manager::Install( )
{
   // read in the classes - configured objects lists
   List classList = _readConfig();
   ListIterator classListIter(classList);
   //iterate over the the classes
   for (classListIter.first();
         classListIter.hasMoreElements();
         String className=classListIter.next()->key())
   {
         NamedObject noClass = GetObject(className);
         List noList = (List)*classListIter.value();
         ListIterator noListIter(noList);
         // iterate over the named objects list
         for (noListIter.first();
               noListIter.hasMoreElements();
               String noName=noListIter.next()->key())
         {
               // Register a new instance with a name
               Register(coName,coClass->New() );
         }
   }
}
```

5. Whenever an configured object has to be instantiated or accessed use the manager's GetObject method.

Variants

Instead of returning a handle to the stored object to clients, and thus sharing the the instances among all clients, the Named Object and the Manager class could easily implemented to give each client a copy of the instance. To do so add a cloning method to the NamedObject interface:

```
class NamedObject {
public:
    ...
    virtual NamedObject* Clone();
private:
    ...
}
NamedObject* NamedObject::Clone()
{
    NamedObject* clone = new NamedObject();
    clone.fName=this->fName;
    clone.fConfig=this->fConfig;
}
```

The Manager now clones the stored objects in the GetObject method and returns them instead of the originals.

```
NamedObject* Manager::GetObject(String Name)
{
    NamedObject* no =
            (NamedObject*)_table.get(Name);
    return no->Clone();
}
```

Known Uses

The application framework WebDisplay makes use of Named Object for several key abstractions like Pages, Renderers and Actions.

Consequences

The Named Object design pattern implies the following *benefits*:

- *Separation of configuration data from class implementation*. This adds another dimension to develope different components without limiting the already existing ones.
- *The class used to instantiate a configured object can easily changed*. If a class is to be replaced by a newer version, it can be better to build the new version as a new subclass to prevent breaking existing applications. New applications on the other side can immediatly start using the new class.
- *Functionally new objects can be developed by configuration only*. Because the configuration data often represents a higher level of abstraction new functionality can be developed with less effort. The amount of reused code is increased.
- *Allows flexible prototyping*. If a new component requires subclassing to fulfill its task, a placeholder for it can be defined just by configuration to integrate it into the application before the writing and testing of the new code is finished.

However, the pattern also has its *liabilities*:

- *Increased Complexity.* The configuration data builds an additional layer of varying the systems behavior. Developers have to decide if a new feature should be obtained by configuring or by subclassing from an existing class. Debugging suffers also from the increased complexity because the source of errors might lay within the code or the configuration data.
- *New subclasses have to implement the registration to the manager to become accessible.* In languages, that do not support runtime class loading, additional code is required for registering each subclass.
- *Increased number of files.* Each Named Object adds an additional configuration file to the files that have to be managed within the Project.
- *Loss of type safety.* Especially if the objects retreived by name are downcasted to a "expected" type. Thereforethe pattern works best if clients operates only with the interface provided by the baseclass.

See Also

Named Object can be implemented nicely using Anythings [MäBi96] [SoRü98] for configuration data. Besides the installation part, the Manager mainly plays the role of an Registry holding the named objects.

Credits

We like to thank our shepherd Robert Hirschfeld as well as André Weinand and Erich Gamma for their design of WebDisplay.

References

[Beck96]
Kent Beck*: Smalltalk Best Practice Patterns*, Addison-Wesley, 1995
[GHJV95]
E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
[MäBi96]
Kai-Uwe Mätzel, Walter Bischofberger: *The Any Framework A Pragmatic Approach to Fleibility*, USENIX COOTS Toronto, 1996
[POSA96]
F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal*: Pattern-oriented Software Architecture–A System of Patterns*, J. Wiley & Sons, 1996
[PLOPD3]
Pattern Languages of Program Design 3, Addison-Wesley, 1997
[Riehle96]
Dirk Riehle: *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose*, UbiLab Tecnical Report 97.1.1, 1996
[SoRü98]
Peter Sommerlad, Marcel Rüedi: *Do-it-yourself Reflection*, patterns submitted for EuroPLoP 98.