Optimizing Development of a Complex Software by Using and Extending Design Patterns[*]

Fabien Costantini[#], Christian Toinard[#], Nicolas Chevassus[+]

[#]CEDRIC – Centre D'Etudes et de Recherche en Informatique CNAM
292 rue Saint-Martin 75141 Paris Cedex 03 France
costanti@cnam.fr, toinard@cnam.fr
[+]EADS Research Center
12 rue Pasteur BP 76 92152 Suresnes Cedex France
nicolas.chevassus@eads-nv.com

ABSTRACT

The design of large software is improved by re-use of well-defined design patterns. However, the way these techniques can be really used and combined all together into large software is less addressed. Our paper describes a case study of software re-use. It shows how a virtual prototyping application can be designed through integration and extension of existing patterns. First, the Observer defines one (subject) to many (observers) dependencies mechanism with automatic notification/update features. This pattern is widely extended to address a-synchronism and to support automatic subscriptions. Second, the Composite makes it possible to easily apply polymorph operations to an objects' tree. We show how to extend the composite to handle relation constraints between the objects with the same interface. We also explain how we combine the Prototype and Composite patterns to manage the creation of different types of graphical entities. At last, some Singleton extensions are proposed to provide better control on that pattern.

Keywords: design patterns, software reuse, lessons learned, software quality.

1 Introduction

Our case study takes place within a virtual prototyping application. That application enables to create and allocate space for different graphical 3D entities. The application manages different sub-systems organized hierarchically (i.e. mechanical structures, hydraulic & electrical components, …). Computational modules verify functional constraints over that hierarchy. That software is used widely at the early stage of aircraft design. Moreover, a collaboration framework [CSTG2000] has been integrated within that standalone application.
Our paper shows how different patterns can be extended, combined and reused efficiently to achieve a better design.

2 CategorizedObserver

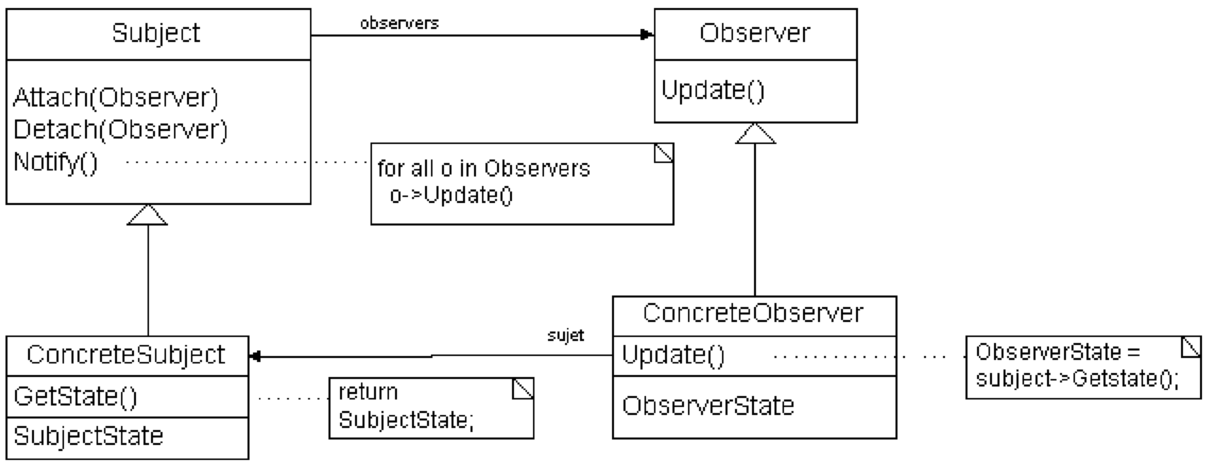
Intent

We want to define a model/view mechanism with automatic attachment, notification and update features between different categories of models from their respective representations.
Let us consider [GHJV95] which defines an Observer pattern to further abstract coupling between one subject (the model) and its observers (representations).

[*] Work supported by the European Commission through the IST-AIT VEPOP project under contract number IST-1999-13346, EADS Corporate Research Lab, EADS Airbus, CNAM-CEDRIC, Flow-Master, University of Oulu

The standard Observer scheme

The following diagram describes the standard structure:



Observer

Initially, a concrete observer uses the abstract Attach() method to connect itself to one subject. The subject then keeps a reference to this new observer for further notification calls. The concrete subject will use the Notify() method when its internal state has changed. In Notify(), all the attached observers will be updated by invoking their respective Update() method.

This scheme works well when

- Observers know the concrete subject at the time they want to connect to.
- There are few subjects, because storing observers references in each subject may be expensive otherwise
- the lifetime cycle of subjects and observers and their creation order are well known to enable an explicit ‘per subject’ attachment

But what happens if an application would like to automatically connect a category of subjects with a category of Observers?

Moreover, how to manage unknown creation order of subjects and observers? What mechanism will guaranty that an observer will be notified of latter subjects existence and state changes?

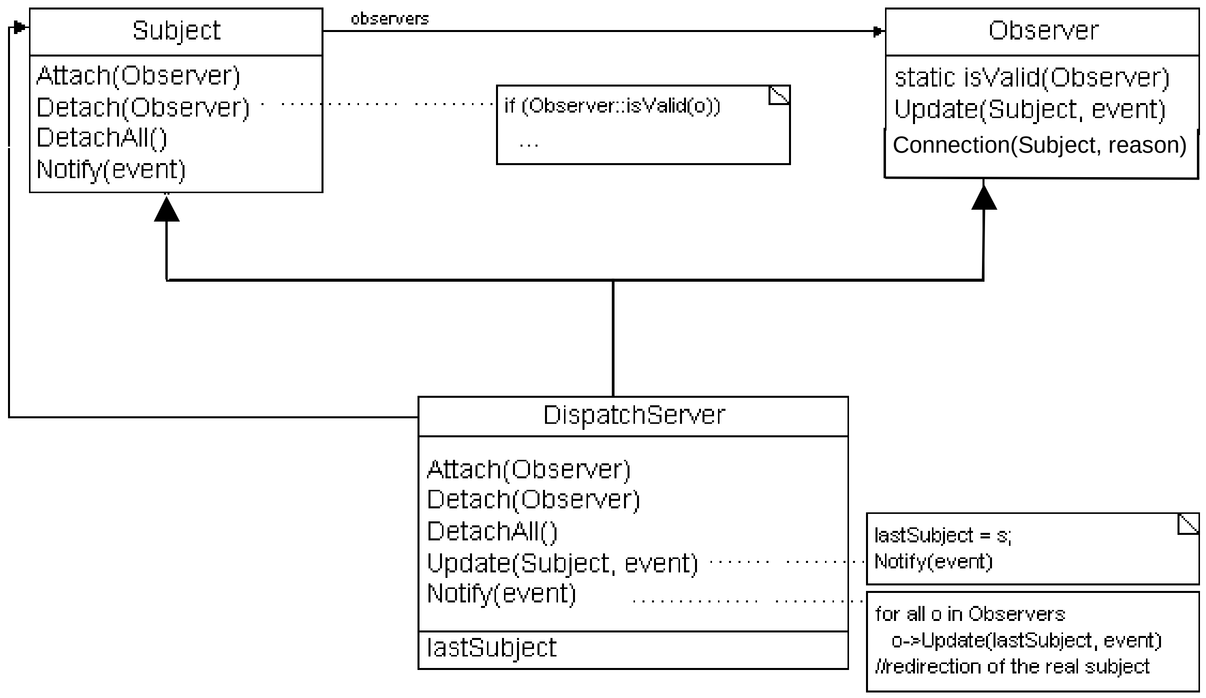
[BMRSS96] introduces a pattern variant called ‘GateKeeper’ where subject and observers communicate indirectly. But, this variant does not address our needs.

Our Observer extension answers such requirements.

The DispatchServer extension

Principle

The DispatchServer consists of a new indirection level between subjects and observers.
It acts as a subject or as an observer and so, will inherit from both the abstract Subject and Observer classes as shown:



Observer with DispatchServer

On one side, a category of observers will invoke the DispatchServer Attach() method at creation time on the unique instance of the DispatchServer associated to their category. This unique instance is implemented with the Singleton pattern (see [GHJV95]) and is seen here as a subject.

On the other side, the category of subjects will invoke at creation time their own Attach() method on the same DispatchServer instance which acts here as an observer. So they will store only one reference of this observer object.

The DispatchServer, as an indirection, must keep track of subjects as well as observers connected to it to propagate the notifications and updates between them, so it contains a list of subjects and a list of observers as well.

Solution

Now let's have a closer look to our Subject and DispatchServer implementation, in particular to the Connection() method:

```
void DispatchServer::Connection(Subject* s, Reason reason) {
   if (reason == ATTACHED)        add       s in _subjects
  else if (reason==DETACHED)    remove  s in _subjects
for each valid o in _ observers
            o->Connection(s, reason);
}
void Subject::Attach(Observer * o) {
            add o in _observers
            o->Connection(this,ATTACHED);
}
void  Subject::Detach(Observer* o) {
   remove o in _observers
   if(o is valid)  o->Connection(this,DETACHED);
}

void DispatchServer::Attach(Observer*o) {
   add o in _observers
   for each s in _subjects
            o->Connection(s, ATTACHED);
}
```

First, notice that each time a subject attaches itself to a concrete DispatchServer, it invokes its Connection() method to register with the DispatchServer.

Second, note that concrete observers Connection() method may be overridden when necessary to allow observers to be notified when a subject has been created or deleted. So, it is possible for an observer to handle new subjects automatically. Connection() here acts as a special notification message that allows a-synchronisms.

Reciprocally, new attached observers after subjects' creation automatically receive a Connection() message as shown in DispatchServer::Attach(). That informs them of subjects created before they attach.

Finally, note that a Observer::isValid() method is also implemented to simplify side effects that may appear when dealing with a-synchronism between subjects and observers.

Uses in our context

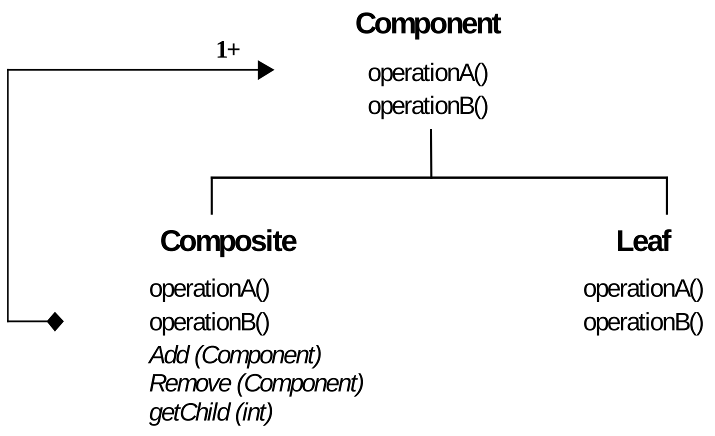
We use this pattern extensively to implement complex notification mechanisms between the system entities represented in the 3D scene graph and their tree views in the Graphical User Interface. We also use it for coupling our collision detection engines in an interesting way: a collision manager is observing elementary collision nodes. The collision manager is observed by also two collision viewers. So, the Collision Manager is a concrete DispatchServer that also interprets and filters the collision nodes before dispatching their notifications to the corresponding views.

3 Composite with component relations constraints

Intent

We want to apply polymorph operations on a tree structure in order to treat leaf objects and composition of objects uniformly. Besides, we want constraint relationships between the component nodes. More precisely, the solution must help in updating automatically parent owner-to-children relations while adding/removing children.

The standard Composite pattern



Composite

The standard Composite [GHJV95] [CS95] can apply operations uniformly on Composite and leaf nodes. Its implementation of operations allows recursive traversal of the tree transparently.
But we would like the component relations to be handled transparently, i.e. when we delete it, we want its reference to be removed from its owner. In our solution, we implement an automatic handling of a coherent single parent to multiple children relations.

Composite with one-to-multiple relations built-in

In the context, the Composite is the unique parent of one or more Leafs.

On the Component part we add one to one get/set methods for accessing the parent.
On the Composite part we add relations constraints to Add() and Remove() methods so it updates automatically the parent relation of the concerned children.
When deleting a component we clean his relation in its parent.

Uses in our context
We implement our system entities with this composite pattern. A hierarchical organization is established with these entities. Each system entity leaf may also contain one or more Geometrical entities for its underlying 3D representation. So both system and geometrical entities implement this pattern. Thus, we combine advantage of the Composite pattern with the use of coherent one-to-multiple relations services.

4 Prototype

Intent
After being able to manipulate uniformly system and geometrical entities, we want to clone these entities in a flexible way. More precisely, we want to be able of copying an entity without knowing its concrete type.
As our application use C++, we can first think about simply using a copy constructor but it raise two problems :

- C++ constructors cannot be virtual (see [ARM90] §12.1 [BS92] §12.1)
- no implicit 'deep' copy is possible with such constructor, i.e. references contained are not duplicated

The prototype addresses this problem, is defined in [GHJV95] pp. 117 and is also known as *virtual constructor* in [ARM90].

The idea consists of implementing a Prototype abstract class that implements a virtual clone() method. All concrete prototypes will override this method to achieve a correct copy of themselves.

Prototype extension
When combined with the Composite, instead of simply cloning the object with no parameter we can further it an optional parent parameter so it can update as well the new owner of the copy. So, this method look like this Component * clone(Composite* new_parent=0).

Uses in our context
We implement a virtual clone() method in all system and geometrical entities classes. We also add an optional parameter to specify the new parent of the cloned object so we can update as well its one-to-multiple relations in the composite scheme.

5 Singleton

Intent
With this creational pattern, we intent to ensure a class only has one instance and to provide a global point of access to it without references duplication without polluting the namespace with global variables that reference this single object.

The standard Singleton pattern
To ensure an object is constructed and only once, and permit to access it in a well-defined centralized point of access. see [GHJV95] for details.

Some extensions

First, it is often useful not to provoke a singleton construction just to know if one has been created, so we add the Singleton* hasInstance() static method which doesn't create the object on the fly. Second, though most systems free the used memory and the end of a process, we also add a delInstance() static method to clean-up the singleton on demand, when order of destruction is important, or to avoid debugging tools detecting memory leaks that are not really relevant.

Uses in our context

We systematize use of Singleton every time a single instance of object must be guaranteed. Moreover, we implemented a garbage collector SingletonGC class to ensure all remaining singletons are freed at program termination.

6 CONCLUSION

This case study shows how several patterns are extended and combined to implement a hierarchy of application objects and to apply polymorph treatments on that hierarchy.

The proposed extensions for the composite and prototype are used to create easily the hierarchy of objects and manage automatically one-to-many relationships. Then, the CategorizedObserver permits to generalize and simplify behaviors that process categories of events.

The paper also presents an interesting combination of patterns. CategorizedObserver uses the Singleton to maintain a unique reference of a server for each category, it also uses the Memento pattern to access partial object state . But our Subjects also implement the Prototype, Composite and Visitor patterns.

Lessons learned are:

- Design Patterns minimize the time to design large applications. Development time is mainly spent on adaptation and reuse of good design concepts.
- Design Patterns must be implemented in the context of an application. Developing large software requires a good inspection of patterns usability.
- Design Patterns enabled to modify and extent the software more easily with a low regression rate during the industrialization phase. They also contributed to minimize the side effects by improving the overall software modularity and objects' independence.

REFERENCES

[ARM90] M. A. Ellis, B. Soustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[BMRSS96] Franck Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *A System of Patterns.* Wiley, Chichester, 1996.

[BS92] B. Soustrup. The C++ *Langage.* Addisson-Wesley. 2nd edition, 1992.

[CS95] James O. Coplien, Douglas C. Schmidt, *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[CSTG2000] Fabien Costantini, Antoine Sgambato, Nicolas Chevassus, François Gaillard, *An Internet Based Architecture Satisfying the Distributed Building Site Metaphor*. Conference Proceedings, IRMA2000 Multimedia Computing Track, IDEA GROUP PUBLISHING, 2000.

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