Uncommon Design Patterns

STEFANO FAGO - 2003

Uncommon Design Patterns

G.O.F. Design Patterns are design tools
wide adopted by developers and architects working with Object Oriented Technology and, on them, you may experience a wide literature.
These Slides try to introduce some Design Pattern less visible but very practical!

Uncommon Design Patterns

The Patterns are:

- *Null Object*
- *Encapsulated Context*
- *Role Object*
- *ISP & Object Adaptation*
- *Essence*

Null Object

It proposes the adoption of an object, in relation to a given Type, to expresses the concept of NO ACTION or EMPTY ELEMENT, respecting the contract (the Interface) of the Type of which is a surrogate. The issue that you want to solve is to avoid the use of NULL/NIL in situations where it is necessary to return a reference to an Object, although there will be no processing.

Null Object

Supposing some code:

```
InfoList availableUnits =  manager.getOperativeUnitFor(...);
for (Iterator iterator = availableUnits.iterator(); iterator.hasNext();) {
    // do something with unit
}
```

to be able to processe without error we have to use a nullity check:

```
if(availableUnits != null){
  for (Iterator iterator = availableUnits.iterator(); iterator.hasNext();) {
   // do something with unit
  }
}
```

Null Object

We can avoid this issue with Null Object

```
InfoList getOperativeUnitFor(String nationCode, String areaCode) {
  InfoList result =
  // ...seach for...
  // if there no Unit and Empty List is returned...
  result = new VoidInfoList();
  return result;
}
```

so the code can became:

```
InfoList availableUnits =  manager.getOperativeUnitFor(...);
  for (Iterator iterator = availableUnits.iterator(); iterator.hasNext();) {
     // do something with unit
  }
```

Encapsulated Context

In complex systems, it becomes necessary to use common data, with global visibility, such as configuration data. The use of global elements is a practice not recommended in Object Oriented Systems, and even if the GOF Singleton Pattern can offer a way out, its abuse can be detrimental.

Encapsulated Context

Supposing to have different Component that need to share common services such as Logging, Order Management and recieve also specific messages. Every Component must have a method with signature:

```
process(SomeMsg msg, OrdersManager ordMan, Logger log){
   ...
}
```

Encapsulated Context

If the number of Common Services grow, we can have a signature mess!

So we need a single object to encapsulate the different services without using global object but using injection on method...

```
process(ProcessContext ctx){
  Msg msg = ctx.getActualMsg(); ctx.log(...);
  ctx.processLastOrder(); ...
}
```

Role Object

On large size design is easy to deal with a modeling problem: an element can plays different roles depending on the context in which it is placed.

The idea of Role can be developed from the concept of interface and use multiple inheritance to compose different roles: this approach can lead to an object unwieldy and difficult to manage!

Role Object

Objective of the Role Object Pattern is to offer a solution to the problem of roles thanks to a set of Role Objects that are dynamically added or removed compared to a Core Object.

Role Object

We can use the simple example of Employee and different roles that this element can plays!

interface Employee:

```
interface Employee{
        String getName();
        long getId();
        boolean hasRole(String role);
        EmployeeRole getRole(String role);
        void removeRole(String role);
        void addRole(String role);
      }
```

Role Object

Now we can define the Core Object and Role Objects:

```
class EmployeeCore implements Employee{
   private Map roles = ...
 public boolean hasRole(String role) { return roles.containsKey(role);}
 public EmployeeRole getRole(String role) {return (EmployeeRole)
   roles.get(role);}
public void addRole(String role, EmployeeRole impl) {roles.put(role,
   impl);}
 public void removeRole(String role) {roles.remove(role); }
... }
```

Role Object

```
class EmployeeRole implements Employee{
  private EmployeeCore core; ...
  String getName(){ return core.getName(); }  ...
  boolean hasRole(String role) { return core.hasRole(); }
 ... }

class Director extends EmployeeRole{  Director(EmployeeCore core)...}
class Buyer extends  EmployeeRole{ Buyer(EmployeeCore core)...}
```

Role Object

We can construct the complex object:

```
EmployeeCore [] empls = new EmployeeCore[3];
empls[0] = new EmployeeCore();
empls[1] = new EmployeeCore();
empls[2] = new EmployeeCore();
Director d = new Director(empls[0]);
empls[0].addRole("director",d);
Buyer b = new Buyer(empls[2]);
empls[2].addRole("buyer",b);
```

We can hide complex object with a manager:

```
Employee emp = manager.getEmployee(...);
```

Role Object

Follow, the sample usage:

```
Employee emp = manager.getEmployee(...);

if (emp.hasRole("director")) {
  Director director = (Director)emp;
  ...
}

if (emp.hasRole("buyer")) {
  Buyer buyer = (Buyer)emp;
  ...
}
```

ISP & Object Adaptation

Inspired by PEP 246 (Python Language)

any object might "embody" a protocol:

adapt(component, protocol[, default])

- checks if component directly implements protocol
- checks if protocol knows how to adapt component
- else falls back to a registry of adapters indexed by type(component) [[or otherwise, e.g. by URI]]
- last ditch: returns default or raises an exception

ISP & Object Adaptation

- We first define ad adaptable protocol then define the adapters and register them; at last we use type to play ;-)

- Our simple protocol:

```
public interface Adaptable
{
  public Object adapt(Class targetType);
}
```

ISP & Object Adaptation

- Define some target Contracts and Adapters:

```
public interface Reader {  public Object read();   }
public interface Shape { public float area(); public void draw(); }
public interface Writer { public void write(Object o); }
public class ShapeReaderAdapter{ ... }
public class ShapeWriterAdapter{...}
```

- Register Contracts with relative Adapters:

```
AdapterRegistry.registerAdapter( Shape.class, Reader.class,
  ShapeReaderAdapter.class);
AdapterRegistry.registerAdapter( Shape.class, Writer.class,
  ShapeWriterAdapter.class);
```

ISP & Object Adaptation

- Define some Adaptable:

```
public class Circle implements Shape, Adaptable{
        public float area() {... }  public void draw() {...}
        public Object adapt(Class targetType) {
           return AdapterRegistry.getAdapter(this,targetType);
        }
 }
```

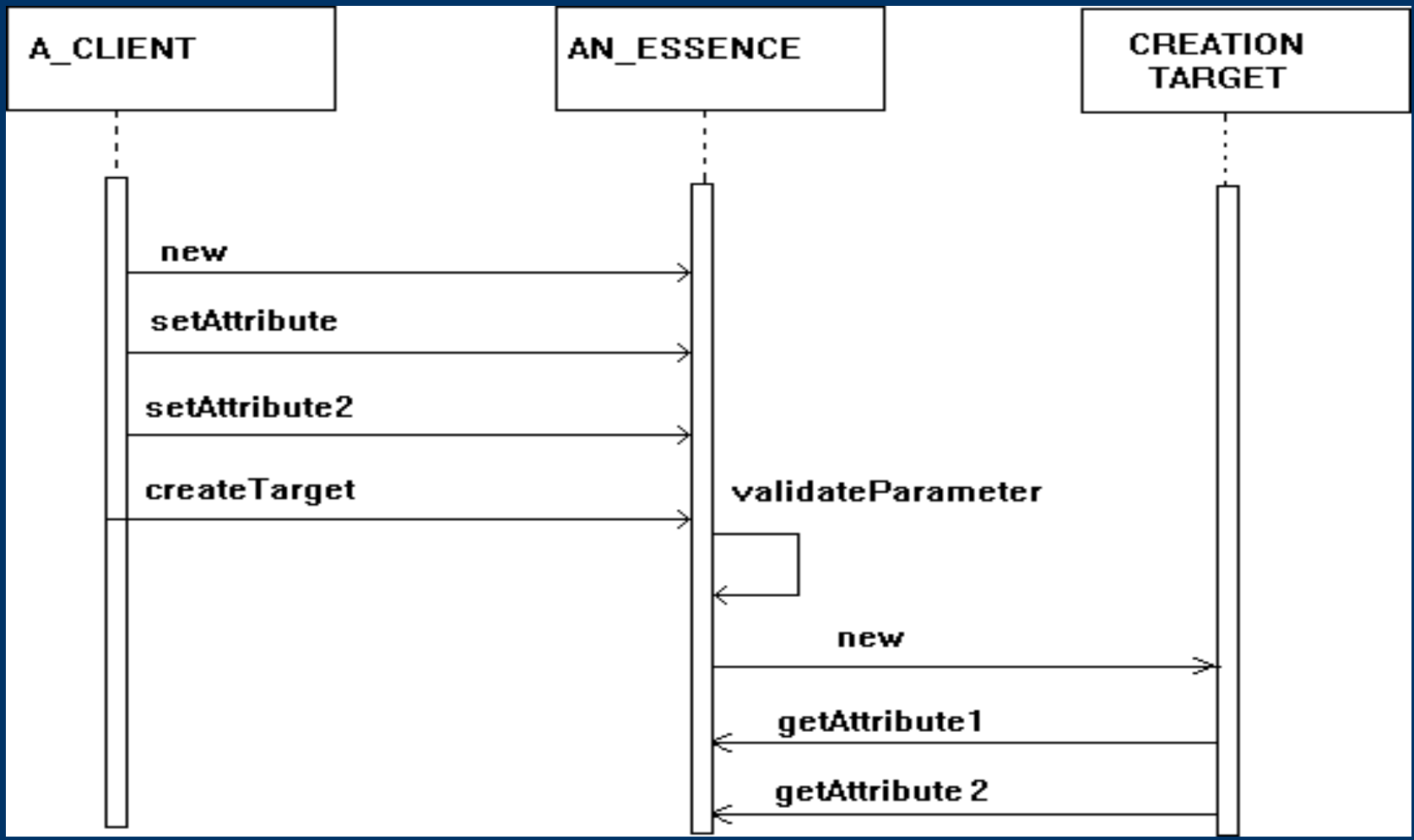
- Use Object Adaptation:

```
Circle circle = new Circle();
Writer writer = (Writer) circle.adapt(Writer.class);
```

Essence Pattern

This Design use a particular class identified as **Essence** that encapsulates the basic properties for the configuration and validation of the target Object. Once the object Essence is instantiated, the validation of the elements is fired and only if this operation has successfully, it's created the object of target Type, set with the same Essence properties.

Essence Pattern



Essence Pattern

```
public class Server {
  public Server(ServerEssence essence) {  ... }
   ... }

public class ServerEssence {
   public void setHost(String host) {  ... }  public String getHost() { ... }
   public void setPort(int port)  { ... }  public int getPort() {  ... }
   // other methods
   public Server getServer()  {
      if (isValid()) return new Server(this);
        throw new InvalidServerConfigurationException(this);
  } ...
}
```