Interception Patterns

Andrés Fortier. Juan Cappi. Gustavo Rossi.

LIFIA-Facultad de Informática-UNLP, Argentina
{andres,jcappi,gustavo}@lifia.info.unlp.edu.ar

Abstract. Developing applications that support evolution in a seamless way is not an easy task; in an early design stage, some concepts may be confused or some potential functionality not foreseen, resulting in bad separations of concerns. In this kind of designs, future changes may imply huge modifications if a high-quality object-oriented model is the desired product. In theory, object oriented programming is supposed to help us to cope with sudden changes, but that doesn't mean that, because we use an object oriented language, all the problems will be solved; in the end, the design is up to us. In this paper we show how to introduce a new spot in an object oriented design, to allow better behavior delegation. We will also show how flexible architectures can be built around any of the patterns that conform this pattern system, allowing to cope with new behavior in a "peaceful" way.

1 Motivation

To get started, let's think of the problem that appears when we try to extend an application to support new features (for example, personalization). Roughly speaking, the personalization logic usually involves three different aspects: the rules, the user profile and the application of those rules to particular business objects (a deep discussion about how rules should be modeled can be found in [Arsanjani 00]). Although these concepts can be modeled with objects, the problem arises when we need to decide *who* has the responsibility of triggering the customization process. To see this clearer, let's think of a simple example: suppose there is a class `Product`, and a class `CD` (subclass of `Product`). Any product can be asked for its price, which, as expected, is the price that will be charged to the customer. Suddenly, the marketing department of the company we are working for, decides that the customer will get a 10% discount if he has bought more than 15 CDs in the last month. A working (but naive) solution would be to let the CD know the profile and, when asked for its price, the CD would calculate the final price based on the user information. Let's say that we adopt that solution. Now, the marketing department realizes that the discount was too generous, so the company's policy changes again and, to get a 10% discount, the user needs to buy 20 CDs and at least one book. What kind of changes do we have to make to the system to incorporate the new policy? How do we cope with all the possible (unknown) future changes?

At this point we can clearly see that changing the message `price` for every policy variation is not a desirable solution; if we keep changing the core of our model to adapt to policy variations, we would never get a stable version our system.

This example is just a particular instance of a more general problem: how do we cope with concerns that don't belong to the application model, when they partially overlap with the core objects' implementation? If we model those concerns in the same class, we would generate code that is hard to maintain and evolve.

While developing this technique, we realized that we have already run into similar problems when building frameworks to distribute objects or to make them persistent. In all of these frameworks, one of the main goals is to achieve a clear distinction between the framework and the application model; in an ideal implementation, the code of the application's logic should not be obscured by the functionality provided by the framework.

This problem (usually known as *the cross cutting concern problem*) is tackled essentially by an increasingly popular technology called AOP. In this paper, instead of presenting a language extension, we are going to show how to change an object's behavior with the elements provided by the object oriented paradigm (i.e. objects and messages). Although we think it would be really important to explain the similarities and differences between all the approaches that can be used to tackle this kind of problem, we will stay away from such a discussion, since a thoughtful comparison would be a large topic to cover and it is out of this paper's scope.

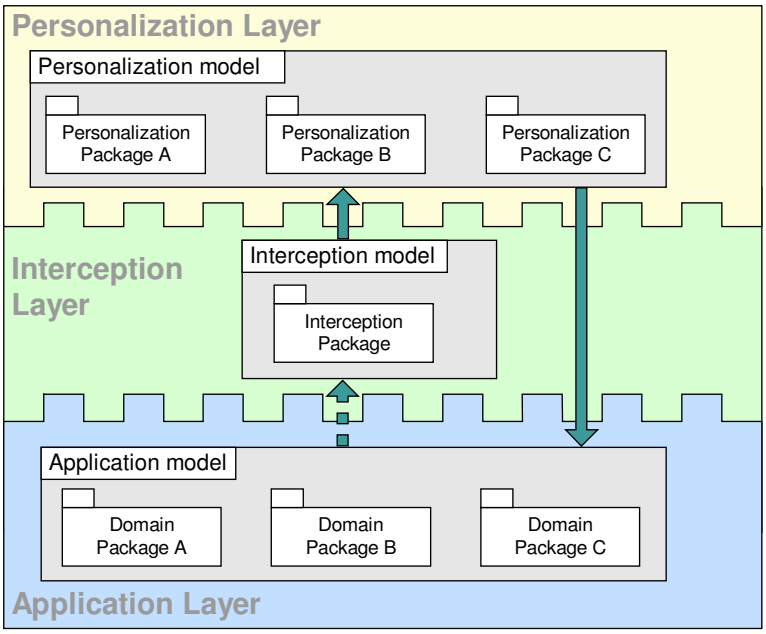
2 General Solution

To allow for an (almost) independent evolution of the model and those concerns that are somehow orthogonal (like discount policies), a clear separation of concerns between the objects that model the personalization logic and the ones that actually model the application domain must be achieved.

To accomplish this, we recognize the need for a technique to intercept messages that are sent to application objects, so that the execution of the methods related to those messages can be altered in a way that doesn't impose big changes in the original code. It would be also desirable to know how this interception can be performed in an organized way and to be able to select between different methods, depending on specific forces (design and implementation stage, access to the source code, remaining time before release, etc).

Taking the personalization example, before returning the real price, the interception system would delegate the calculation of the discount to the personalization logic, leaving the method `price` in `Product` unchanged.

To deal with this kind of requirements we found out that the best way is to diagram the whole system in a three layered architecture [Rossi 02]. If we apply that architecture to the personalization example (as shown in the figure below), we would find that the objects that represent the application model are in the first layer. The second layer provides the mechanism for intercepting messages between objects that reside in the first layer and the objects that model the policies. Finally, in the third layer we find the policies' model. This architecture allows a clear decoupling between the application model and the framework, letting the two models to practically evolve in an independent fashion.



In the following section we show different patterns for achieving interception, ranging from intrusive code to transparent interception. For concrete examples we will generally refer to personalization, distribution or persistence frameworks, since those are the most "classical" cases where this problem appears.

3 Pattern System

In this section we present a set of closely related patterns to achieve behavior interception (i.e. intercept a message send to modify an object's behavior in response to that message), according to a given context. For the sake of simplicity, and since the motivation is in its essence the same (behavior interception), we use the initial motivation as each pattern's motivation, describing the context in which they should be applied.

Pattern name	Intent
Intrusive interception	Capture a method invocation without requiring any "behind the scenes" processing at the cost of mixing interception code with application's code.
Subclass Interception	Capture a method invocation without altering the original method's code. Produce an important behavioral change with few modifications in the original source code.
Proxy Interception	Capture a method invocation without cluttering the application code with the interception's layer behavior. The interception design should scale and be able to evolve along with the model.

3.1 Intrusive interception

3.1.1 Intent

Capture a method invocation without requiring any "behind the scenes" processing, in a way that the programmer is 100% in charge of what is going on. Allow the interception to take place at any spot inside the method, at the cost of mixing a few lines of interception code with the domain object's code.

Also avoid any kind of performance lost by incurring in meta-level constructs or complex mechanisms.

3.1.2 Solution

This first solution presents maybe the simplest way of intercepting behavior and delegate it in some other layer; as a matter of fact, it just exploits one of the main concepts behind the OO paradigm. The idea is to delegate the personalization behavior to another object by sending a message to it in the part of the code where is desired (as we will show later in the implementation section, there are two basic ways of doing this without altering the state definition of a class).

This technique is based on the Observer pattern [Gamma 95], which is implemented in Smalltalk to accomplish the dependency mechanism: an object (the *observer* or *client*) registers itself to another object (the *subject* or *observed*) so that when the subject changes, the observer gets a notification. The observed is aware that some kind of dependency exists and informs its clients by sending to itself a variation of the message `changed`, eventually specifying which aspect has changed. The default behavior of the message changed (which is implemented in the `Object` class) is to notify all its dependents.

This strategy has proved to be effective in the design of user interfaces (for more information on this topic refer to the Observer pattern [Gamma 95] and the MVC architecture [Krasner 88]).

3.1.3 Implementation

Using the personalization framework as an example, we may think of customizable objects as having a little knowledge about which methods should be personalized. So, when a method has to be personalized, a variation of the message `personalize` is sent to himself, which will be captured by the right personalizer.

A possible solution (like the one present in Smalltalk for the observer pattern) is to let the class `Object` have the behavior needed to react to the message `personalize`. Generally, in languages like Java, Delphi and the like (i.e. when not working inside an environment) it's not possible to add new behavior to the base class (`Object`), so we have to look for another solution. Taking this into an account and having in mind the fact that it's not desirable to alter the original class' state definition we show two possible solutions:

3.1.4 Variation 1: Redefine the original class' superclass

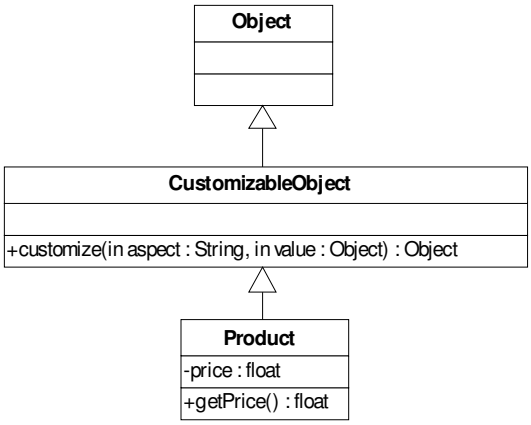
The idea is to define a class which implements the behavior related to the interception mechanism, and also takes care of the delegation `(CustomizableObject)`. Then we must change the original class' superclass from `Object` to `CustomizableObject`.

Even though this technique is effective, it is considered by the authors to be a bad OO design and should not be taken as a programming habit. It also should be noticed that this technique doesn't scale; think of tackling the following problems with this approach:

- Persistence
- Distribution
- Customization

So you end up with `PersistentObject`, `DistributedObject` and `CustomizableObject`. But, in some cases, you may need persistence and distribution, in other cases customization and persistence and, on a third development, distribution and customization. Clearly there is no way of accomplish this without having extra (an maybe undesired) functionality (i.e. customizable and persistent objects may also carry the ability to be distributed).

3.1.4.1 Structure



As we can see in the figure, the class `Product` is a subclass of `CustomizableObject` instead of `Object`. If we think in terms of a three layered architecture, the class `CustomizableObject` would be in the interception layer and the class `Product` in the domain layer.

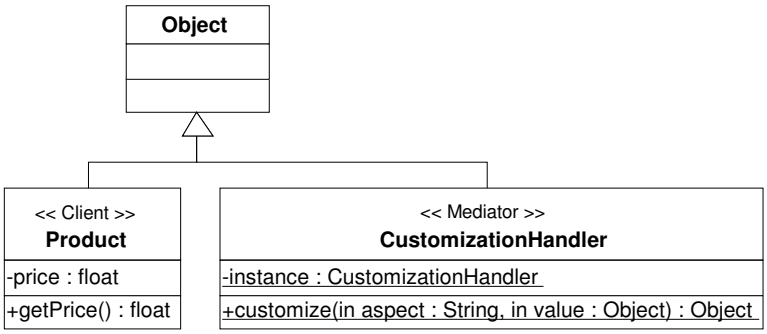
3.1.4.2 Sample code

```
public class Product extends CustomizableObject {
    float price;
    public float getPrice(){
    return ((Float)this.customize("price",Float(price))).floatValue();
    }
}
```

3.1.5 Variation 2: Define a customization request handler

Another way for get a similar approach is to define a Mediator [Gamma 95] (which can be implemented as a Singleton class [Gamma 95]), which listens to the requests of all the personalized objects and dispatches those requests to the objects that have registered some kind of interest.

3.1.5.1 Structure



In this variation the class `Product` (the mediator's client) would be in the application layer and the class `CustomizationHandler` (the mediator) in the interception layer. When the mediator gets the personalize request it redirects it to the appropriate set of rules (which resides in the customization layer).

3.1.5.1 Sample Code

```
public class Product extends Object {
    float price;
    ....
    public float getPrice(){
           return((Float)CustomizationHandler.customize("price",
                                Float(price))).floatValue();
    }
public class CustomizationHandler {
    static CustomizationHandler instance;
    public Object customize (String aspect, Object value){
        if (instance == null) {instance=new CustomizationHandler();}
        return instance.customize (aspect,value);
    }
    ....
}
```

3.1.6 Consequences

1. The code that solves the problem (i.e. the code that models the essence of the class) is mixed with the interception's code (and, as a consequence, with the personalization's code). This may yield difficult to write/maintain code.
2. The distinction between the application- and interception-layer is blurred.
3. The programmer has full control of how the interception is being done and can decide when this is going to occur. As we will show in the next patterns this is not always the case and the end user will only have the ability to customize a message before or after it's execution.
4. There is no need of pre- or post-compilation processing over the source code or bytecodes.
5. There is a minimum delegation, so there is practically no time overhead in the execution of the method.

3.2 Subclass Interception

3.2.1 Intent

Capture a method invocation without altering the original method's code. In this pattern we assume that the behavior adaptation can be achieved by triggering a method before or after the real method's execution. Although this may seem an important restriction, in practice we didn't found any mayor problems when using this approach.

A very important issue here is the relationship between development time versus good OO design. As we will show in the next section, in some cases this patterns allows for an important behavioral change with few modifications in the original source code, but the cost is paid in the future: this pattern doesn't scale to support the model's evolution.

3.2.2 Solution

To be able to effectively replace an object with another, the substitution principle must hold. This principle states that it should be possible to think of a specialized object as if it were a base class object. Generally, typed languages like Java, guarantees that this principle holds; any object can be replaced by another object whose class is a subclass of the former, so a first approach would be to subclass the original class and delegate to the superclass the model's behavior.

It must be clear that we do not consider this to be a good OO design; this pattern reflects the need for a quick-and-dirt implementation that requires minor changes in existing code. A better design can be achieved by using, for example, interface driven restrictions instead of class-driven ones.

Let's take as an example a persistence framework: the main task of the framework is to detect a change in the (persistent) object's state and record it in the desired medium. A possible approach for detecting this kind of changes in an object, is to take a snapshot of it before and after a method's execution; if after the execution the state has changed, the framework should record that modification.

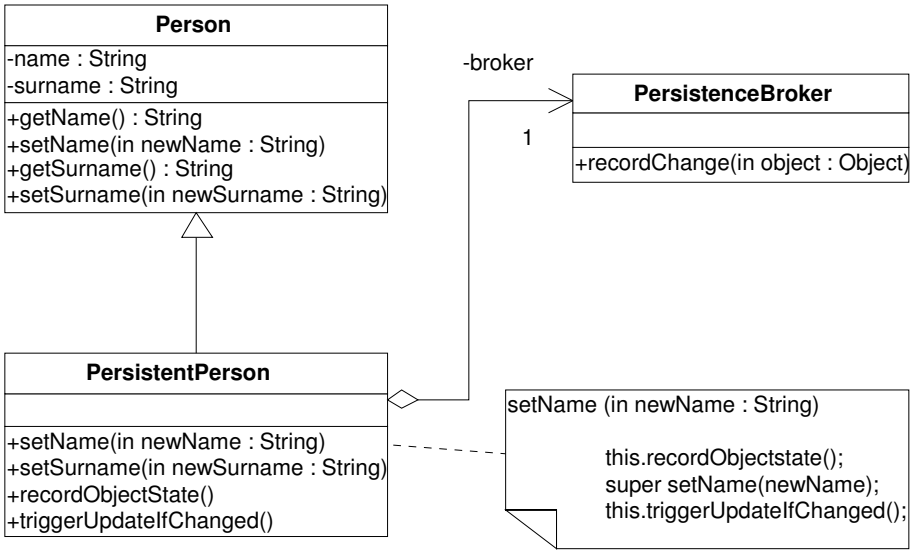
If we apply this solution to the persistent framework example, we would need to define a subclass of the class whose objects are going to be persistent. Once defined we would override every message so that first it records the internal state, then call the inherited method and finally check to see if there are any changes in the object's structure.

3.2.3 Implementation

The implementation of this pattern is quite straightforward and can be done in any OO programming language. The following steps must be followed:

- Create a subclass of the original class
- Redefine the methods that must be intercepted
- Perform any pre-processing
- Call the inherited method and hold the result
- Perform any post-processing
- Return the result

3.2.4 Structure



In this pattern the class `Person` would be part of the application layer, the class `PersistenceBroker` would be part of the framework layer and the class `PersistentPerson` (which makes the connection between them) would be part of the interception layer.

3.2.5 Sample code

```
public class Person {
    private String name;
    private int age;
    ....
    public void setName(String name){this.name = name;}
public class PersistentPerson extends Person {
    private void recordObjectState()
    {....}
    private void triggerUpdate()
```

```
{....}
public void setName(String name){
    this.recordObjectState();
    super setName(name);
    this.triggerUpdate();
}
```

3.2.6 Consequences

1. The code that solves the problem is separated from the interception's code. As a matter of fact, if we consider `PersistentPerson` to be part of the interception layer, is the interception's layer code the one that invokes the code that belongs to the model.
2. Since `PersistentPerson` is a subclass of Person, we are sure we can replace persons with persistent persons
3. The changes in existing code are minimum. We just need to replace the constructor `new Person()` with `new PersistentPerson()`. There is no need of changing other things like type signatures.
4. The class extension and name replacement can be automatically generated (see Appendix).
5. It's clear that this is a bad OO design:
   - There is no reuse. If we want to convert cars into persistent cars, we would have to subclass Car and repeat almost the same code
   - The design doesn't scale in a straightforward way. If we want to apply the same strategy for customization, we would have to subclass from `PersistentPerson` and not from Person, i.e. we must be aware that our objects are persistent in order to make the customizable.
   - If in the future we decide we need two different subclasses of person, lets say student and teacher, we would have to define two new subclasses (`PersistentStudent` and `PersistentTeacher`).

3.3 Proxy Interception

3.3.1 Intent

Capture a method invocation without cluttering the application code with the interception's layer behavior. Again, let the programmer handle how this interception is achieved. As in the previous pattern, you are also seeking for simplicity and don't want to mess with automatic generated code, maybe accepting to pay an initial effort in generating a basic structure.

You are also seeking for an interception design that does scale and that can evolve along with the model.

3.3.2 Solution

The solution presented here has been widely used in many similar cases, like adding pre- and post- conditions, message tracing for auto-generated diagrams, object distribution, etc.

As stated on the Proxy pattern [Gamma 95], the relationship between the original objects is changed so that a new level of indirection appears. The idea is to put an object between the client and the real object (the *subject*), so that the client thinks it is sending messages to the subject, when actually the proxy is intercepting them and solving its execution by collaborating with the subject, and eventually other objects.

We will take as an example the same problem we used in the last pattern. Another solution for the persistence framework would be to replace the objects that are persistent with proxies, which in turn will reference the real object, so that code can be executed before and after the real method. The code that is executed before the real method takes care of recording the object's state; after the original method is executed and the control is returned to the proxy. Finally the actual state of the object is compared with the old sate; if any change is detected, the proxy tells the persistence layer to update the information in the object store.

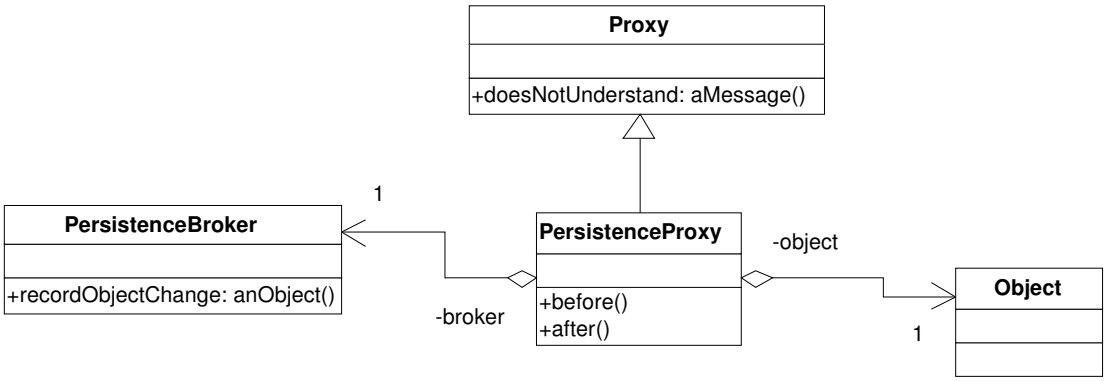
3.3.3 Implementation

This implementation is highly dependent on the programming language being used. In this section we will present four different ways of implementing this pattern.

3.3.4 Variation 1: Generic proxy (Smalltalk)

The idea is to exploit the `doesNotUnderstand:` mechanism present in Smalltalk; when a message is sent to an object, the method dictionary of the object's class is searched for a method that implements that message. If none is found, the same goes on in the superclass chain until it reaches the topmost class in the hierarchy (`Object`). When this happens the object gets a chance to handle the situation: the message `doesNotUnderstand:` is sent to the object, taking a message as a parameter (i.e. an instance of the class `Message`, which holds the selector an the arguments). The default implementation of the `doesNotUnderstand:` message can be found in the `Object` class, which just raises a wall back window.

3.3.4.1 Structure



In this pattern the classes `Proxy` and `PersistenceProxy` would be placed in the interception layer, the class `PersistenceBroker` would be placed in the framework layer and the proxy's subject in the application layer.

3.3.4.2 Sample code

Using this approach we could define a generic proxy that re-implements the `doesNotUnderstand:` message. The code would look something like:

```
doesNotUnderstand: aMessage
    | res |
    self beforeMethod
    res:= self receiver perform: aMessage selector wthArguments: aMessage args
    ^self afterMethod: res.
```

And in the `new` message of the persistent object class we would write

```
new
    ^Proxy on: (self basicNew)
```

In the persistence framework example we would subclass the `GenericProxy` class and redefine the `beforeMethod` and `afterMethod` so that changes in the receiver can be logged.

3.3.4.3 Consequences

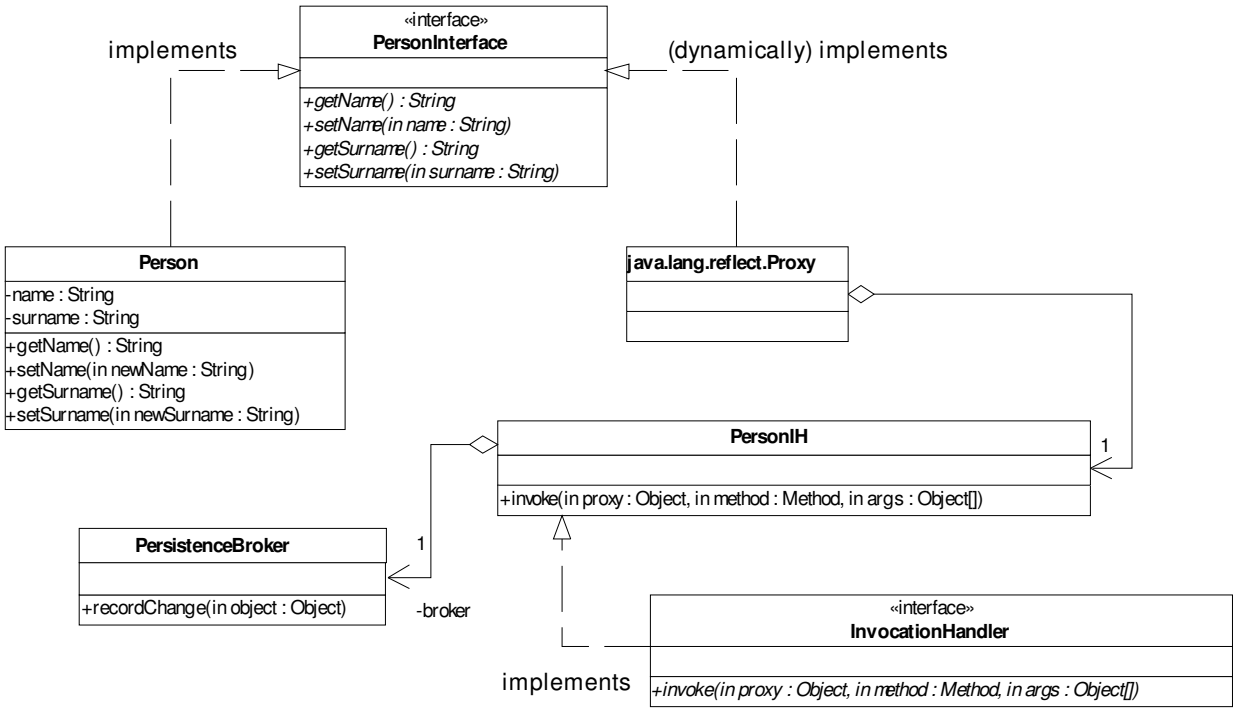
1. Once defined the proxy, only a small change needs to be done: redefine the `new` class method in all the classes we want to "proxy".
2. Due to the Smalltalk nature, no "type" problems arises
3. Interception code is clearly decoupled from the business code.
4. The use of a primitive like `perform` may slow down the execution of the method.
5. If not treated carefully, object identity may be lost.
6. The approach does scale. Any proxy can be proxied.

3.3.5 Variation 2: Generic proxy (Java)

A similar idea can be applied in Java, although it is restricted in comparison with its Smalltalk counterpart. Starting in JDK 1.3, a new class named `Proxy` appeared as part of the `java.lang.reflect` package. The basic idea is to create a proxy that implements the same interface as the object being replaced, so that the real object can be replaced by the proxy and no type problem arises.

To determinate how a message will be handled, an object that implements the `InvocationHandler` interface must be given to the proxy. The `InvocationHandler` interface consists of a single method called `invoke`, which is in charge of redirecting the message to the real object.

3.3.5.1 Structure



In this variation the interface `PersonInterface` is the key to connect the application layer (class `Person`) with the interception layer (class `Proxy`); the broker is part of the framework layer.

3.3.5.2 Sample code

```
public class PersonIH implements InvocationHandler {
    Object obj;

    public PersonIH(Object o){
          obj = o;
    }

    public static Object newInstance(Object obj){
        return java.lang.reflect.Proxy.newProxyInstance(
               obj.getClass().getClassLoader(),
                 obj.getClass().getInterfaces(),
                 new PersonIH(obj));
    }
```

```
    public Object invoke(Object proxy,
                         Method method,
                         Object[] args) throws Throwable {
        Object result;
        this.recordState();
        result = method.invoke(obj, args);
        this.compareState();
        return result;
    }
}

public class Person implements PersonInterface {
//class definition
}
```

And in the application code we may find:

```
PersonInterface p = (PersonInterface)(PersonIH.newInstance(new Person()));
p.setName("John");
```

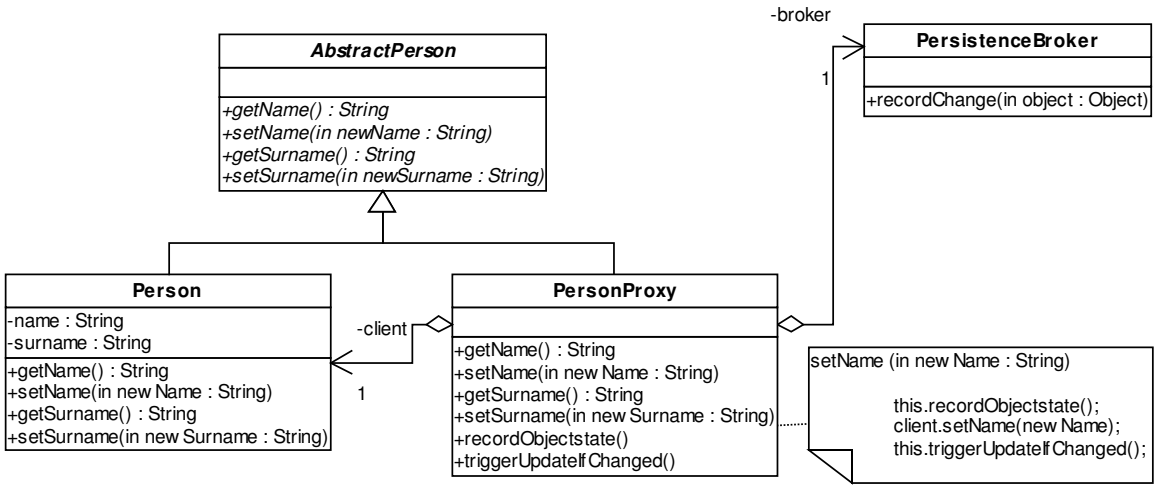
3.3.5.3 Consequences

1. All the parameters constraints must be interface-based. Due to Java's type system, and since `Proxy` is not a subclass of `Person`, class-based restrictions would forbid to replace an instance of `Person` with an instance of `Proxy`. This is an important item to take into an account, since, if the project is not designed from scratch with this in mind, mayor changes may be required in the source code.
2. The code used to instantiate the original class (i.e. `new Person()`) must be replaced to instantiate proxies (i.e. `(PersonInterface)(PersonIH.newInstance(new Person()))`).

3.3.6 Variation 3: Abstract subclass and delegation

This variation is the one that best fits the "classic" proxy class diagram. In this variation, the `Proxy` class and the original object's class share the same (abstract) superclass. This structure guarantees (again, because of the substitution principle) that, if all the types constraints are based in the abstract superclass, the concrete person and the proxy are interchangeable one with another.

3.3.6.1 Structure



In this variation the class `AbstractPerson` is used to establish class-based constraints. The class `Person` is part of the application layer, the class `PersonProxy` is part of the interception layer and the class `PersistenceBroker` is part of the framework layer.

3.3.6.2 Sample code

```
public abstract class AbstractPerson{
    public abstract String getName();
    public abstract void setName(String n);
    ....
    }
public class Person extends AbstractPerson{
    private String name;
    public String getName() {return name;}
    public void setName(String n) {name=n;}
    ....
    }
public class PersonProxy extends AbstractPerson{
    private Person client;
    private PersistenceBroker broker;
    private void recordObjectState() {...}
    private void triggerUpdate() {...}
    public String getName(){
           String res;
           this.recordObjectState();
           res=client.getName();
           this.triggerUpdate();
           return res;
           }
   public void setName(String n){
           this.recordObjectState();
           client.setName(n);
           this.triggerUpdate();
           }
}
```

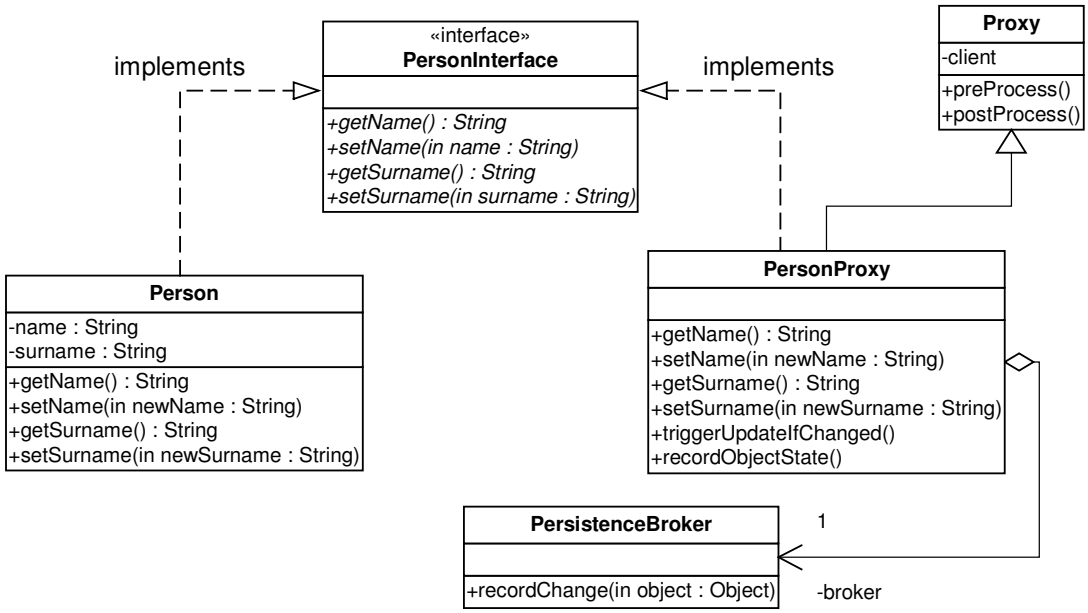
3.3.6.3 Consequences

1. All type constraints must be declared based on the abstract class, and not on a particular subclass.
2. As in the previous variation, if this isn't take into an account from the beginning of the project, mayor changes may be required to implement this kind of interception.
3. The code used to instantiate the original class (i.e. `new Person()`) must be replaced to instantiate proxies (i.e. `new PersonProxy(new Person())`).
4. Interception code has to be re-implemented for every class that has to be persistent.

3.3.7 Variation 4: Interface-based proxy

The last variation on this idea is a cleaner one since we can finally treat independently the class hierarchy with the messages that an object can respond. To accomplish this, we take advantage of the Java's interfaces mechanism; the only drawback on this implementation is that (as in the dynamic proxy example) the real class is obliged to implement a determined interface; all references must be interface-based instead of class-based. As in the proxy example, to avoid huge changes in the source code, interfaces must be used to place constraints since the start of the project.

3.3.7.1 Structure



As in the second variation, the interface `PersonInterface` is used to connect the application layer (class `Person`) with the interception layer (abstract class `Proxy` and concrete class `PersonProxy`). The class `PersistenceBroker` is part of the framework layer.

3.3.7.2 Sample code

```
public interface PersonInterface{
    public String getName();
    public void setName(String n);
    ....
    }
public class Person implements PersonInterface{
    private String name;
    public String getName() {return name;}
    public void setName(String n) {name=n;}
    }
public class PersonProxy extends Proxy
                implements PersonInterface{
    private void preProcess() {...}
    private void postProcess() {...}
    public String getName(){
           String res;
           this.preProcess();
           res=((PersonInterface)client).getName();
           this.postProcess();
           return res;
           }
    public void setName(String n){
           this.preProcess();
           ((PersonInterface)client).setName(n);
           this.postProcess();
           }
    ...
}
```

3.3.7.3 Consequences

1. All type constraints must be interface-based.
2. As in the previous variations, if this isn't take into an account from the beginning of the project, mayor changes may be required to implement this kind of interception.
3. There is no relation between the original object class hierarchy and the implementation of the proxy. This allows building a hierarchy of proxies, improving reuse.

3.3.8 General Consequences

1. There is no need of pre- or post-compilation processing over the source code or bytecodes, so the whole process can be understand by the programmer.
2. Interception code is clearly decoupled from the business code.
3. The approach does scale. If well implemented, any proxy can be proxied.
4. The programmer can only add behavior before or after the original method's execution.
5. If not treated carefully, object identity may be lost.

4 Conclusions

Although good object oriented programming techniques help us to cope with unexpected requirements modifications, designing for change is still a big challenge. We emphasize the fact that good separation of concerns is one of the key points to allow an application to evolve, in many different ways, without becoming a maintenance nightmare.

In this paper we presented different patterns to implement behaviour interception, mostly based in good object oriented programming, in a language independent fashion. Being able to intercept a message call allows us to change, in some way, how the object reacts to that call and what kind of "side effects" it produces, permitting the programmer to alter (probably by delegating in some concern-specific object) an object's behaviour. By using these patterns in a methodical way, flexible architectures can be built to cope with further unforeseen changes, resulting in significant overall cost savings.

5 References

[Arsanjani 00] Ali Arsanjani: "Rule Object: A Pattern Language for Pluggable and Adaptive Business Rule Construction". Proceedings of PLoP2000. Technical Report #wucs-00-29, Dept. of Computer Science, Washington University Department of Computer Science, October 2000. http://jerry.cs.uiuc.edu/~plop/plop2k/proceedings/Arsanjani/Arsanjani.pdf.

[Gamma 95] E. Gamma, R. Helm. R. Johnson, J. Vlissides: "Design Patterns. Elements of reusable object-oriented software", Addison Wesley 1995.

[Krasner 88] G. Krasner and S. Pope. "A cook-book for using the model-view-controller user interface paradigm in Smalltalk-80". Journal of Object-Oriented Programming. Volum1, Number 3, pp 26-49, August/September 1988.

[Liang 98] S. Liang, G. Bracha. "Dynamic Class Loading in the Java Virtual Machine". In proceedings of the ACM Conference on Object-oriented Programming, Systems and Applications, October 1998.

[Hinkle 93] Bob Hinkle, ***Reflective Programming in Smalltalk***. Tutorial #11, OOPSLA '93. www.cs.cmu.edu/afs/cs.cmu.edu/project/clisp/OldFiles/hackers/ram/work/rtl/OOPSLA93-refl-tut.ps

[Rossi 02] Gustavo Rossi, Andrés Fortier, Juan Cappi: "Mapping Personalization Policies into Software Structures". In the Recommendation and Personalization in eCommerce workshop, in The second International Conference of Adaptive Hypermedia and Adaptive Web Based System.

[BCEL] http://jakarta.apache.org/bcel/manual.html

[Reflex] http://www.dcc.uchile.cl/~etanter/Reflex/

[OpenJava] http://www.csg.is.titech.ac.jp/openjava/

[Javassit] http://www.csg.is.titech.ac.jp/~chiba/javassist/

6 Appendix

Even though each of the patterns presented in this paper can be applied to any generic design, in many cases (specially when not thinking about interception at an early design stage of the application) applying this kind of solution may require an important amount of code rewriting (and model re-design). In that situations the effort needed to restructure the model can be such, that the designer may end up preferring the quick-and-dirt solution rather than a good object-oriented design.

The good news is that most of the changes can be automated in some way, taking a great load off to the programmer. In this section we will present some of the available mechanisms to ease the programmers task when he finds himself in this kind of situation.

6.1 Class loaders

Class loaders [Liang 98] are a powerful mechanism for dynamically loading software components on the java platform.

The java virtual machine uses class loaders to load class files and create objects that represent those classes. Since class loaders are ordinary objects (they are instances of subclasses of `ClassLoader`), new classes can be created and class loaders can be replaced, so that we can alter the way in which classes are generated.

The major problem with this approach is the same we faced when using generic proxies: when plan to replace a class, we are forced to work interface-oriented instead of class-oriented. In this particular case, the class to be replaced and the class replacing it, must implement the same interface.

6.1.1 Application

You can use class loaders at subclass interception pattern (section 3.2), by dynamically loading the "concrete subclass". In this case, you don't need to work interface-based, because you are replacing instances of a subclass of the original one, so the type system ensures that the classes can be replaced.

6.2 Automatic Code Generation

If the language we are working on offers full reflective capabilities (as Smalltalk does), most of the patterns earlier presented can be implemented in a non-intrusive way. Unfortunately, this is not the usual case (languages like Java or C++ offers poor or no standard reflective facilities), so new tools must be used to ease the programmers task. As we stated in the introduction, in some cases the changes can be automated, allowing for a simple program to rewrite the code for us. Suppose that you have decided to work with proxies to intercept the behavior declared in a `Person` class; following the structure presented, you rearrange the class hierarchy and now you define the classes `AbstractPerson`, `Person` and `PersonProxy`. The problem arises if most of the code has already been written: the programmer needs to change the way that persons are created and all of the constraints based on `Person` must be switched to `AbstractPerson`. Clearly, this problem can be solved with a kind of smart search-and-replace object that not only does this job, but also takes care of the class rearranging and of the specification of the proxy class.

6.2.1 Application

You can use pre-processing in all of the previous discussed patterns and variations by making convenient modifications as needed. For instance,

- At subclass interception (section 3.2), you can replace the references to Person, by references to `PersistentPerson`.
- At proxy interception pattern, you can have a custom browser to define which messages will be intercepted and the extra code that will be executed. Then a parser can get the code, generate the class definition and replace the references, getting a legible code.

6.3 Lightweight classes

Languages that naturally support reflection capabilities, are more suitable to introduce meta-level features (being Smalltalk one of those). Bob Hinkle presented at OOPSLA '93 [Hinkle 93] an interesting approach to take advantage of reflective capabilities in Smalltak. Such approach is known as Lightweight Class, and the main idea is to replace an object's original class, so that the new class intercepts the desired messages, and redefine its behavior (generally by collaborating with other objects).

6.3.1 Application

You can apply this idea, at the subclass interception pattern (section 3.2) by changing, in run time, the class of (for example) `Person`'s instances, by `PersistentPerson`'s instances.

6.4 Byte code and source code manipulators

In this section, we are going to present a set of toolkits which would make easier both jobs, the byte codes manipulation, and the source code manipulation.

Generally these kinds of systems define some kind of Meta Object Protocol (MOP) so that the programmer doesn't have to deal with bytecodes.

6.4.1 BCEL

The Byte Code Engineering Library (BCEL) [BCEL] is a powerful class library intended to give users a convenient possibility to analyse, create, and manipulate (binary) Java class files. Classes are represented by objects that contain all the symbolic information of the given class: methods, fields and so on.

Such objects can be read from an existing file, be transformed by a program (e.g. a class loader at run-time) and dumped to a file again. Another interesting application is the creation of classes from scratch at run-time.

BCEL is 100% portable and compatible with standard JVMs.

6.4.2 Reflex

Reflex [Reflex] is an Open Reflective Extension of Java, entirely written in Java, and 100% portable and compatible with standard JVMs (from 1.2.2 and above).

It enables the control of some normal Java objects by meta-objects. A meta-object is an object responsible for extending or modifying the semantics of a language mechanism such as method invocation, object creation, etc.

Reflex allows reflection to be introduced only for classes, objects, methods (more generally speaking, language entities) that require it, therefore limiting performance loss.

The toolkit includes the necessary components for using the generic MOP of Reflex, with a class builder that allows the handling of method invocations.

6.4.3 Open-java

OpenJava [OpenJava] is an extensible reflective extension language based on the Java language. The OpenJava MOP (Metaobject Protocol) is the extension interface of the language. Through the MOP, the programmers can customize the language to implement a new language mechanism.

The special feature of the OpenJava MOP is its class meta-object API, through which programmers can handle source code as object oriented language constructs. i.e. classes, methods, fields, etc. Though its translation is performed at compile-time, interfaces are similar to Java Reflection API at runtime and easy to use for high-level translations, like getting information about methods, adding methods, modifying methods and so on.

It is fully written in Java compatible to JDK 1.1, so it can be run on any Java Virtual Machine of JDK 1.1 and Java2.

6.4.4 Javassist

Javassist (*Java* Programming *Assist*ant) [Javassit] makes Java bytecode manipulation simple. It is a class library for editing bytecodes in Java; it enables Java programs to define a new class at runtime and to modify a class file when the JVM loads it. Unlike other similar bytecode editors, Javassist provides two levels of API: source level and bytecode level. If the users use the source-level API, they can edit a class file without knowledge of the specifications of the Java bytecode. The whole API is designed with only the vocabulary of the Java language. You can even specify inserted bytecode in the form of source text; Javassist compiles it on the fly. On the other hand, the bytecode-level API allows the users to directly edit a class file as other editors.

6.4.5 Application

You can apply more than one of those toolkits to a single pattern, or combine some techniques to achieve a complete functionality, but, in both cases, is up to the programmer the decision of which one is better to apply, or which is the better way to apply them.

Have a few examples as a guide:

- Use BCEL, or Javassit at the subclass interception pattern (section 3.2) to create, on the fly, the modified subclass (`PersistentPerson` at the example).
- Use OpenJava to make a language extension that supports a kind of “interception protocol”, so that, you can specify explicitly, the method to intercept and the way in which you want to intercept it.