Using Meta-Level Techniques to Personalize O-O Applications

Andrés Fortier, Gustavo Rossi, Juan Cappi
LIFIA-Facultad de Informática-UNLP, Argentina
[andres, gustavo, jcappi]@lifia.info.unlp.edu.ar

Abstract

In this paper we discuss how to use reflective techniques for personalizing object-oriented applications. This approach is based on a clear separation of concerns, namely: base application functionality, user profile management, and personalization rules; our approach simplifies the evolution of Web Applications when adding personalization features (such as recommendations, special offers, individual interfaces, etc). We first explain why personalization functionality should be dealt by separating concerns. Next we introduce a simple example and focus on different personalization patterns, emphasizing on behavior personalization. We also show which design structures are the most appropriated for obtaining seamless extensions to existing software. We finally discuss some further aspects such as using meta-level constructs for designing personalized applications.

1-Introduction

Personalization has become a very important issue in software applications ranging from word-processors to e-commerce and other Web Software. Building customized software is not a new problem; however, the increasing popularity of the World Wide Web, and the evolution of hardware appliances make personalization a hotter topic. Designing personalized software may imply dealing with different concerns (modeling the user, implementing personalized interfaces, modeling customization rules, implementing complex algorithms, etc) that must be seamlessly integrated.

However, while user modeling, profile derivation and personalization algorithms (such as collaborative filtering) have been extensively discussed in the literature [Oreizy 99], less attention has been paid to the modeling and design process of this kind of software. In this position paper we claim that personalization is an interesting and critic aspect of an application's evolution. This is true not only because of the difficulties inherent to the problem (such as integration of concerns) but also because of the evolvable nature of personalized applications. More precisely, as personalization requirements and policies change over time, software maintenance becomes a nightmare. We have been studying the problem of web applications' evolution; we have analyzed design problems related with personalization and identify usual personalization patterns [Rossi 01a, Rossi 01b]. In this position paper we present a solution to the personalization puzzle that improves separation of concerns.

The structure of the paper is as follows: We first discuss the problem of designing personalized applications in the context of applications' evolution and why separation of concerns is a key solution for coping with this problem. Next we show a first approach, by describing the right separation of concerns and how a generic model can be obtained. Finally we will go one step further and show how thinking of personalization from the meta level point of view will give a better approach to address a generic framework for personalizing applications.

Though we exemplify with applications in the e-commerce domain, the approach is general enough to be applied in other domains. For the sake of conciseness we do not address in this position paper our approach for user modeling; instead we focus on the implementation of personalized behaviors.

2-Designing Personalized Software

We can think of personalization as adding new concerns to the problem of application's design, namely the user (or role) profile and the personalization rules. Not only we need to model the basic application domain (e.g. an electronic store) but also we have to model the user and the ways in which this model will interact with the underlying application model (roughly speaking the personalization rules). As previously stated we think that this problem is related with the application's evolution because requirements related with personalization tend to change over time.

Let us suppose that we are building an e-commerce application and we want to keep track of how many times each user has bought some sort of product, so that we can perform some site statistics later.

Even though this update is fairly simple, an interesting discussion might rise to decide which object of the model should be responsible for triggering the update in the user profile (and of course to decide what the user profile should be responsible of), or to figure out were the logic of that update should be coded. But most answers that may come up will not be completely satisfactory as we will need to add new behaviors to domain objects; behaviors that, by the way, do not belong to the original objects' responsibilities. Suppose that the marketing department decides that we should record every product the user has bought and later, because of policy changes, only those products that fulfill some condition. We are faced to repeatedly change code that has already been worked out. Unfortunately the pace of changes related with personalization (or other customization features) may be fast or even hysteric.

This example gets even worse when other personalization features that involve complex rules and user profile data are introduced (think for example on price discounts, recommendation algorithms and personalized check-out processes). It is important to note that in the current state of affairs in the business world, upgrades related with personalization are usually unpredictable and we have to deal with them (as with most changes in business rules) in a quick and efficient way.

3-Decoupling concerns using wrappers

As we stated before, we should try to avoid mixing the basic application's logic with personalization code. In [Rossi 01c] we presented a set of micro-architectures to address this problem, showing how to achieve a well-defined separation of concerns. The idea (originally devised for e-commerce applications) is based on the interception of messages to objects whose behavior has to be personalized by using a wrapper between the personalized object and the message sender. The wrapper should collaborate with other personalization-specific objects to decide how to answer to that message. As an example, suppose we want to personalize the price of a product depending on an algorithm (that may be even different according to the user's profile). In Figure 1 we present a first approach for solving the personalization problem. Notice that there is a clear separation of concerns; Class Product is designed (and evolves) without personalization features in mind, while all personalization issues are handled by objects that do not belong to the core application's model.

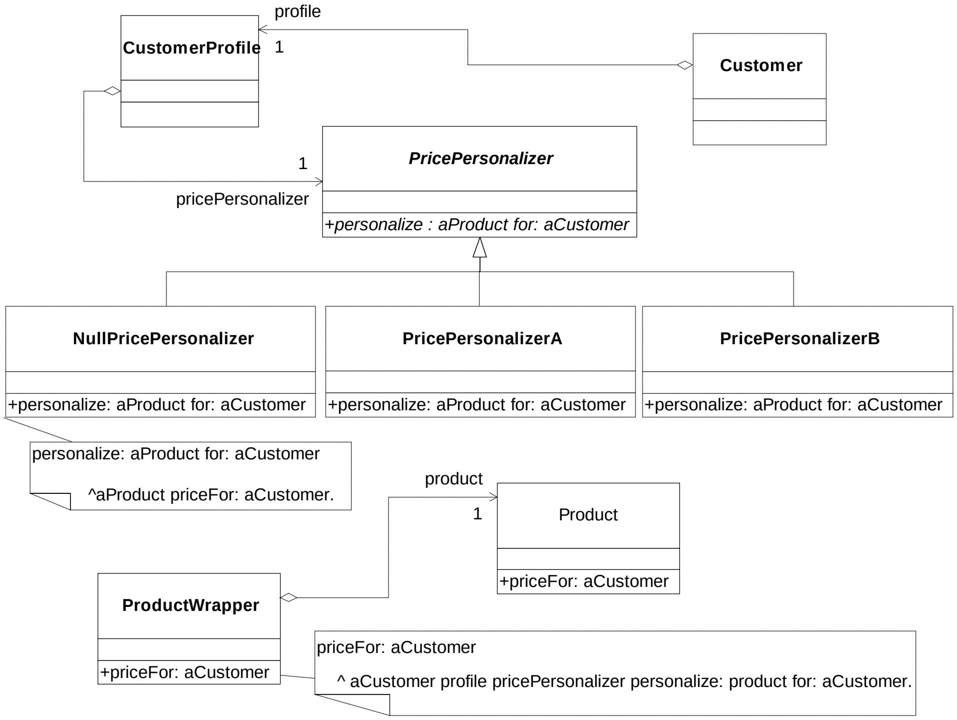

profile
CustomerProfile
1
Customer
1
PricePersonalizer
pricePersonalizer
+personalize : aProduct for: aCustomer
NullPricePersonalizer
+personalize: aProduct for: aCustomer
PricePersonalizerA
+personalize: aProduct for: aCustomer
PricePersonalizerB
+personalize: aProduct for: aCustomer
personalize: aProduct for: aCustomer
^aProduct priceFor: aCustomer.
product
1
Product
+priceFor: aCustomer
ProductWrapper
+priceFor: aCustomer
priceFor: aCustomer
^ aCustomer profile pricePersonalizer personalize: product for: aCustomer.

Figure 1 : Using wrappers and personalizers in an e-commerce application

This approach may have a great impact on the application's evolution; these are some of its consequences:

- **The designer can focus on one concern at a time.** Since the problems are solved in different modules of the system, which are almost independent, the designer can first solve core application issues and then move to the personalization stuff.
- **The behavior of domain objects will not be affected by policies changes.** As opposite to the previous counting example, changes in the algorithms or policies will not affect the behavior or semantics of products.

Though the architecture in Figure 1 solves some of the previously mentioned problems, it presents some drawbacks:

- **When should the object be wrapped?** It is very important to determine when to wrap the domain object so that there are no references to the wrapped object that can cause inconsistency. Generally this is done in the constructor of the domain object, which means that at least some code in the domain classes will be consciously referencing a personalization-related object.
- **Self sent messages.** The wrapper is like an external shell for the object, but it is unable to catch messages that the object sends to itself. This may cause incoherence in the behavior of the application. Following the price example suppose that the product implements the message addTo: aRecommendationList for: aCustomer, which adds itself to the list of recommended products, if its price is lower than a value. When the product asks itself for the price (self priceFor: aCustomer) the base method will be called and no personalization will be applied (i.e. no discount will be applied).

- **It may not scale to complex personalization features.** Even though this design is well suited for simple personalization examples, it lacks of flexibility when for example we want to combine different ways of applying algorithms. Suppose that we want to apply more than one algorithm to the result of a method according to a set of conditions. Should we code a huge case statement in each personalizer? How do we apply a cascade of algorithms on the result of a method? What if we need a little variation of the personalizer for a subclass of Product? Should we re-code almost all the personalizer to accommodate this difference?

Based on this questions we found out that there is a need of smaller-grained objects that map the concept of rules, conditions and actions in a way that they can be easily composed and interchanged.

4-Personalization in the meta-level

In this section we will show how to improve the design and evolution of personalized software by using reflective capabilities; though this ideas can be easily applied in languages supporting meta-level facilities, we base this explanation on our Smalltalk architecture (that strongly explodes its capability to deal with meta-classes, blocks and methods as objects). We first explain how to intercept messages, defining a notification layer. Then we show how methods are dispatched and handled to end up defining the components that implement the personalization logic.

4.1 Message interception

As explained in [Brant 98] there are many ways to intercept a message in languages like Smalltalk, where classes are objects and the method to be executed in response of a message request is determined in run time. In our approach we use lightweight classes as defined in [Hinkle 94] that uses the inheritance mechanism to forward the messages for its default implementation, but returns a modification of the code in the case the method is personalized. As shown in Figure 2, when the lightweight class is asked for a method it interacts with a method dispatcher to return an appropriate method handler.

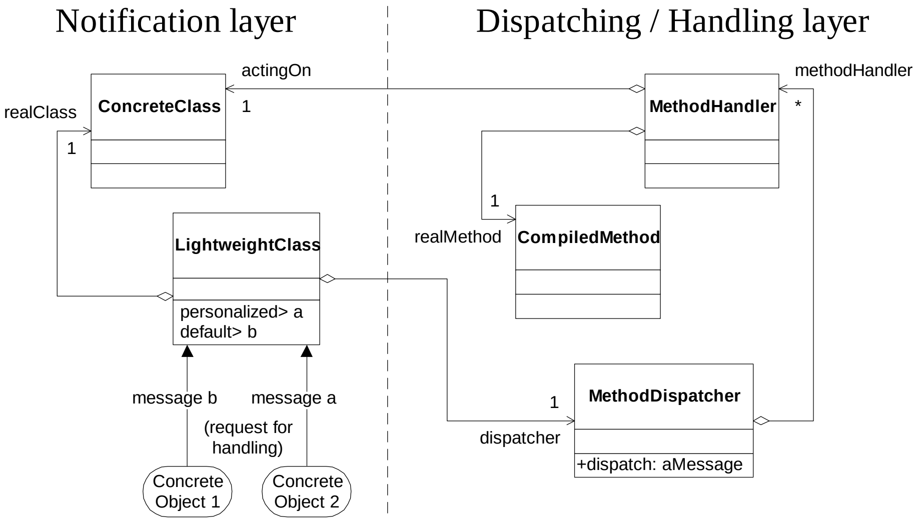
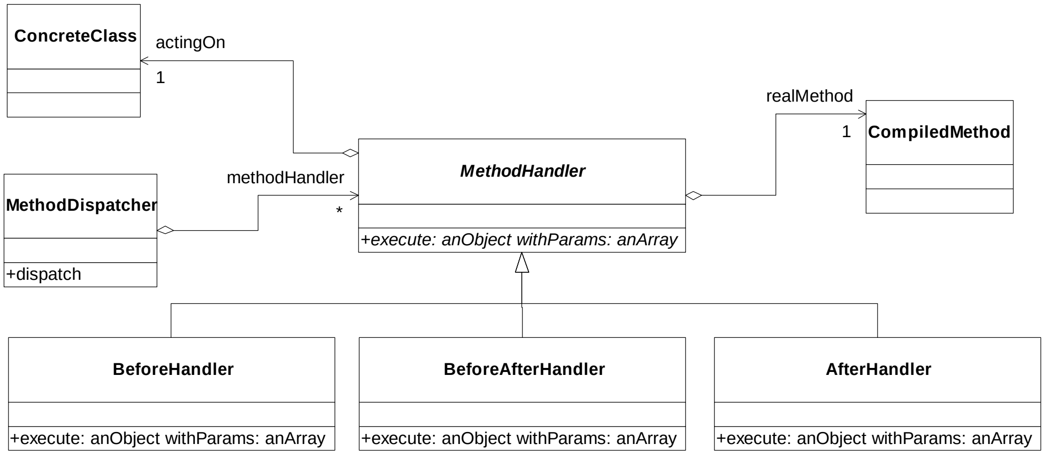


Figure 2 : The interception layer

If the message is personalized, once the dispatcher decides which handler should act, that handler is asked to execute itself. In a generic way we may think of three different ways of handling a message: by taking actions before, after, or before and after the real method is executed (Figure 3).



ConcreteClass
actingOn
1
realMethod
1
CompiledMethod
MethodHandler
methodHandler
*
MethodDispatcher
+dispatch
+execute: anObject withParams: anArray
BeforeHandler
+execute: anObject withParams: anArray
BeforeAfterHandler
+execute: anObject withParams: anArray
AfterHandler
+execute: anObject withParams: anArray

Figure 3 : Method handlers

Once a method has been intercepted, the corresponding behavior must be personalized. In the previous example, if the intercepted message is *price*, we must calculate the price for the corresponding user. As most personalization involves some kind of ruling mechanism, we introduce Rule Objects.

4.2 Rule objects

Personalization (or more generally business) rules are usually expressed in logical terms in the following way: *if a predicate* <p> *is satisfied, the actions* <a_1,a_2,..,a_n> *should be executed* (See for example [Ceri 99]). As rules are not first-class objects we need to analyze the best way to map them into design constructs in order to improve their evolution and maintenance (See for example [Arsajani01]).

As said before, personalizers have the responsibility of executing the personalization code: in fact they implement the concept of a rule. This solution has two main problems. First, as rules may involve many different conditions and corresponding actions, personalizers may become monolithic as they will contain complex *if then else* clauses. The second problem we should solve is related with the evolution of rules. As previously discussed, business rules tend to change quickly; new rules related with an aspect may be added or eliminated. In the case of pricing policies we may have rules that apply when the customer bought many products, others related with the current order, etc. An adequate solution is to take the concept of a rule, map it into a class and present rules as having a condition and one or more actions to execute if the condition is satisfied (Figure 4).

As seen in the figure there are two basic kinds of rules: those that are applied without any interest in the returned value of the base method (DirectRule) and those that somehow depend on it (OnResultRule). To match this organization, proper Action classes have been defined: DirectAction and OnResultAction. When a rule is asked to be applied it evaluates its condition and depending on its value it executes the corresponding action. Notice that we may want to execute more than one action when a rule is applied which motivates Composite actions.

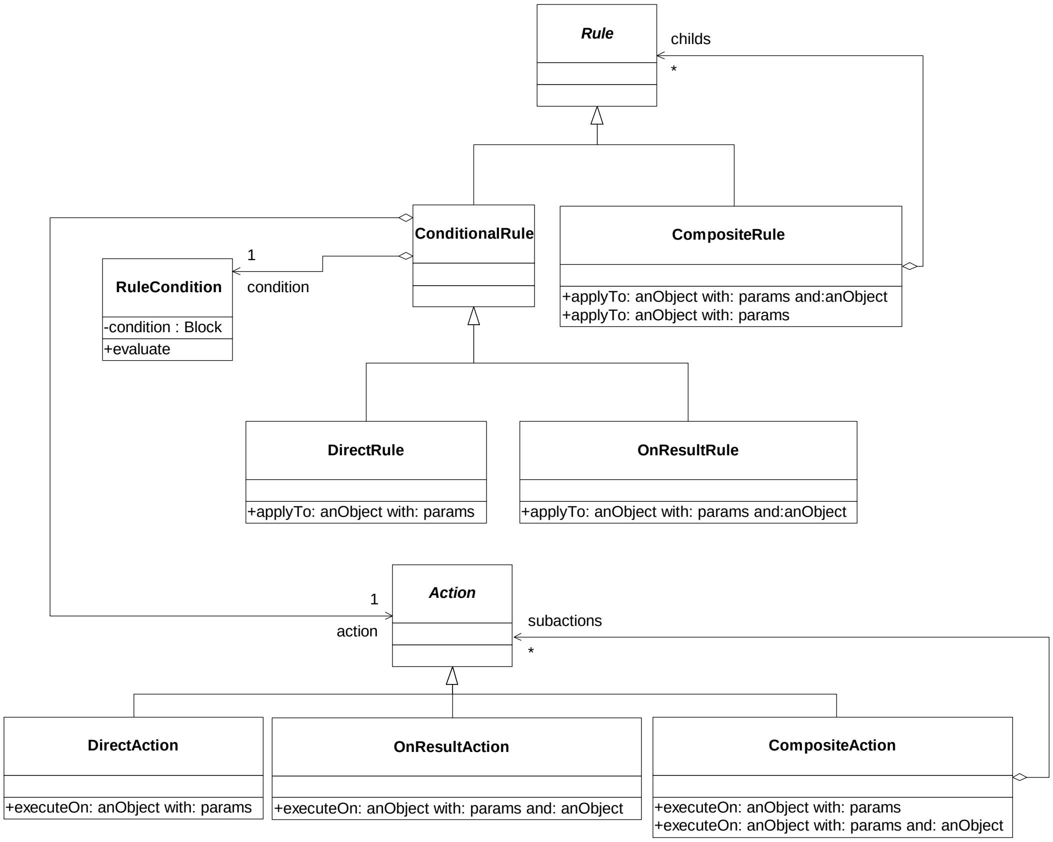

Rule
childs
*
ConditionalRule
CompositeRule
+applyTo: anObject with: params and:anObject
+applyTo: anObject with: params
1
condition
RuleCondition
-condition : Block
+evaluate
DirectRule
+applyTo: anObject with: params
OnResultRule
+applyTo: anObject with: params and:anObject
Action
1
action
subactions
*
DirectAction
+executeOn: anObject with: params
OnResultAction
+executeOn: anObject with: params and: anObject
CompositeAction
+executeOn: anObject with: params
+executeOn: anObject with: params and: anObject

Figure 4 : Rules, conditions and actions

For example let us analyze pricing algorithms in e-stores; the policy of a store may be something like “if the purchase is more than x dollars then a 2% discount is applied”. Now suppose that for Christmas the store wants to apply a 3% discount on all products, so a sub-class of OnResultAction called DiscountAction is defined, which holds onto a discount percentage. A rule is created and configured so that it implements the store discount policy (2%), using an instance of DiscountAction. Then, in Christmas eve the rule is replaced by a composition of rules that cascades both discounts. Notice that the Christmas rule is built using the same DiscountAction that was used for the normal discount and may be reused each time we need to implement a discount.

5-Conclusions and further work

In this position paper we have discussed the problem of personalization from the point of view of application’s evolution. We have presented an initial solution based on objects’ wrappers and improved it using reflection mechanisms. A further analysis on the personalization model showed that representing all the personalization logic in one object (the personalizer) may be too rigid, so we introduced the concept of rules, conditions and actions. Though not shown in this paper, rules may access the user profile to obtain meaningful information about the user, such as purchase history. Decoupling the user profile from the application’s logic is also a key design decision for supporting seamless personalization. We are working on defining an application framework that supports personalization of existing object-oriented applications.

There are three key points that need further analysis:

- Describing the rules as logic predicates, so that the specification can be done without knowing Smalltalk specific syntax. We have experienced with the Soul framework [Wuyts 01] that using meta-level constructs incorporates a Prolog interpreter in the VisualWorks environment.
- We are evaluating if subjectivity techniques [Harrison 93] can be applied to improve the relationships among the personalization rules and the user profile.
- We are analyzing if it is possible to map some of the concerns related with personalization into aspects, in the context of aspect-orientation [Aspect 01]

6-References

[Arsajani01] A. Arsajani: “Rule Object 2001”. In Proceedings of PloP 01, Allerton, IL, September 2001.

[Aspect 01] http://www.parc.xerox.com/csl/projects/aop/

[Brant 98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the Rescue. In Proceedings of ECOOP'98, July 1998

[Ceri 99] Stefano Ceri, Piero Fraternali, Stefano Paraboschi: Data-Driven One-To-One Web Site Generation for Data-Intensive Applications. Proc. VLDB '99, Edinburgh, September 1999.

[Harrison 93] William Harrison and Harold Ossher. "Subject-oriented programming (a critique of pure objects)." In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), September 1993.

[Hinkle 94] Bob Hinkle: Reflective Programming in Smalltalk-80. Tutorial #11, OOPSLA ’94, Portland, Oregon, USA, October 1994.

[Oreizy 99] P. Oreizy, M. Gorlik, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, d. Rosenblum, and A. Wolf: “An architecture-based approach to self-adaptive software”. IEEE Intelligent Systems, pages 54-62, May 1999.

[Rossi 01a] G. Rossi, D. Schwabe, J. Danculovic, L. Miaton: “Patterns for Personalized Web Applications”, Proceedings of EuroPLoP 01, Germany, July 2001.

[Rossi 01b] G. Rossi, D. Schwabe, R. Guimaraes: “Designing Personalized Web Applications”, Proceedings of the 10th International Conference on the WWW (WWW10), Hong Kong, 2001,Elsevier, p.p.

[Rossi 01c] G. Rossi, A. Fortier, J. Cappi, D. Schwabe: “Seamless personalization of E-commerce applications” Submitted paper. In ftp://sol.info.unlp.edu.ar/pub/papers/Seamless%20Personalization.pdf

[Wuyts 01] Roel Wuyts: “A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation” Phd Thesis. Programming Technology Lab, VUB, Brussels, 2001.