Mapping Personalization Policies into Software Structures

Gustavo Rossi, Andrés Fortier, Juan Cappi

LIFIA-Facultad de Informática-UNLP
Calle 50 y 115, La Plata (1900) Buenos Aires, Argentina
Tel/Fax: 54 221 4236585
{gustavo,andres,jcappi}@lifia.info.unlp.edu.ar

Abstract. In this paper we analyze the process of personalizing complex e-commerce applications from a software design point of view. We stress that separation of concerns is a key strategy for obtaining high quality and evolvable implementations. In particular, we show that a naive mapping of personalization policies into software rules may result in a bad design compromising the overall software stability. We first explain our perception of e-commerce applications as views on application models, and briefly explain why personalization functionality should be dealt by separating concerns. We next discuss different mechanisms to intercept behavior from the application model to trigger personalization code and finally we compare our approach for combining rules with other object-oriented design structures in contrast to the existing rule models.

1-Introduction and Related Work

Personalization has become a very important issue in e-commerce applications. Though originally confined to recommender systems, personalization now involves different aspects of these applications and it deserves to be considered a critical design problem. While some personalized applications may seem trivial from the design complexity point of view (for example my.xx sites such as my.yahoo.com or mycnn.com), others provide personalized procedures or adaptive interfaces according to the interface appliance of the current user.

To obtain a personalized application we need to design the user profile, implement personalization policies and map them to the proper application's component, since incorrect design choices may yield a difficult to maintain application. Considering that personalization policies usually vary together with the underlying business (or application) domain, software evolution is another challenge.

Recently, some authors have proposed reference design architectures for customized software [Koch 02, Kappel 01]. They emphasize a clear separation of concerns among the application objects, the personalization rules (generally designed as event-condition-action rules) and the user profile. In general personalization rules are implemented using Rule Objects [Arsanjani 01], thus decoupling rules´ conditions from application-specific code and thus improving modularity. The user profile is usually partitioned in two parts: application-independent, and application-specific.

The former is generally represented as a set of flat objects whose attributes address some user preferences (language, location, etc). These attributes contain values such as “Basketball”, “Buenos Aires” and the like. The latter may have information related with the roles of the user in the application (manager, customer, etc) and eventually with his application’s usage history such as books bought and pages visited, etc. Condition objects check this information to trigger action objects in rule objects; thus, when a rule is activated its condition interacts with the profile to determine if the corresponding action must be executed.

While we consider that this approach is correct, a naive mapping from personalization policies into rule objects may yield a large and flat set of similar rules thus complicating maintenance. Meanwhile, considering information in the user profile as just information records neglects polymorphism and therefore the number of needed rules (more concretely conditions of rules) increases. In this paper we introduce the idea of *personalizer objects*, which abstract different personalization strategies (including rules) for achieving a more modular and stable design.

The structure of the rest of this paper is as follows: to put the discussion in context, we present our object-oriented view of e-commerce applications. Next, we discuss existing architectural constructs for decoupling concerns in personalized software; we then discuss two finer level problems in designing these applications: how to transparently trigger personalization code and how to use objects polymorphism to build modular personalizer objects that may combine different personalization policies. Though this paper is based on our previous experience developing the Object-Oriented Hypermedia Design Method (OOHDM) [Schwabe 98], the underlying ideas can be used in a straightforward way with other object-oriented approaches.

2-The OOHDM framework for E-commerce Applications

The key concept in OOHDM is that Web (in particular e-commerce) application models involve a Conceptual, a Navigational Model and an Interface Model [Schwabe98]. These models are built using object-oriented primitives with a syntax close to UML [UML00]. While in the Conceptual model we describe application’s objects and behaviors, in the Navigational model we focus on the navigational semantics of the application, by describing nodes, links and indexes.

The cornerstone of the OOHDM approach is that the user does not navigate through conceptual objects, but through navigation objects (nodes), which are defined as views on conceptual objects. Links connect those nodes and are usually derived from conceptual relationships.

OOHDM is complemented with a set of navigation and interface patterns that record design expertise to be reused across different applications [Rossi 99]. In particular, we have mined recurrent personalization patterns in Web applications [Rossi 01a]. These (coarse grained) patterns allow us to focus on *what* can be personalized before addressing which concerns are involved in the personalization design process. A detailed discussion on the OOHDM approach for building personalized software can be found in [Schwabe 02].

Summarizing we can personalize:

- The algorithms and processes described in the conceptual model (e.g. different recommendation strategies or check-out processes for different users)
- The contents and structure of nodes and the link topology in the navigational model (e.g. personalizing prices in a e-store, configuring personal home pages as in my.yahoo.com)
- The interface objects and their perceivable aspects and the interaction styles.

It is easy to see that most (if not all) types of personalization finally involve some kind of adaptable conceptual model. Therefore, understanding how to personalize a conceptual (application) model is essential for achieving personalized nodes, links and interfaces. For the sake of conciseness we will then focus on how to build a personalizable conceptual model; we will ignore technological aspects and will only refer to language-specific aspects when strictly necessary.

3-Designing personalized software: A high-level approach

As previously said state-of-the-art approaches clearly separate the most important design concerns related with personalization, namely the application model, the personalization rules and the user profile; in [Kappel 01] a separated event model is also defined. The application model itself is usually divided into a fixed part and a variable part (the objects or behaviors affected by the personalization code). Each module can be considered itself a simple micro-architectural component; the interaction between these components should follow well known patterns in order to keep the software manageable.

Separating rules from application code, allows then to evolve in an independent way and, as a result, the core business behavior is not contaminated with (sometimes unstable) if clauses. Using rule objects, or any of their variants [Arsanjani 01] allows us to have a good model to engineer rules. For more details on separating rules from application code, the reader can refer to [Koch 02, Kappel 01]. We next resume the most important interactions that take place in a personalized sample application.

Suppose an electronic store that sells *products* to *customers.* When he decides to buy, the *check-out* process generates an *order* containing the products, *paying mechanism, shipping address* and *delivering options*; he choose if the product should be gift-wrapped. Each customer has an *account* containing his buying history.

Following previously mentioned approaches we will have at least two other software modules for dealing with personalization:

- **The user profile:** that contains information about the customer's interests; it will need to interact with some application's objects such as the *Customer* and *Account* objects (by the way one may argue that it is debatable whether these classes should be part of the user profile). When software adaptation involves other aspects (such as date or time) a separated context model is also necessary.
- **The rule model:** that encapsulates different kinds of rules, for example for calculating a personalized price, for defining a personalized check-out process, etc; when dealing with event/condition/action rules [Kappel 01], separated event objects will be used.

This paper addresses two important aspects that are usually ignored in the existing literature: the design mechanisms that allow triggering the personalization code and proposes a set of guidelines for re-thinking the personalization code by showing when rules should be used and when they should be replaced by simpler personalizer objects. We also show that user profiles should not contain only plain data but also polymorphic objects for some personalizable aspects of the application.

To accomplish a scaleable architecture, we stress that the design of customized applications should be separated in three clear layers: the application layer, the interception layer and the customization layer (as shown in Figure 1). We also take dependencies between these layers to their minimum, so that different approaches can be used to tackle each layer problems, without affecting the others.

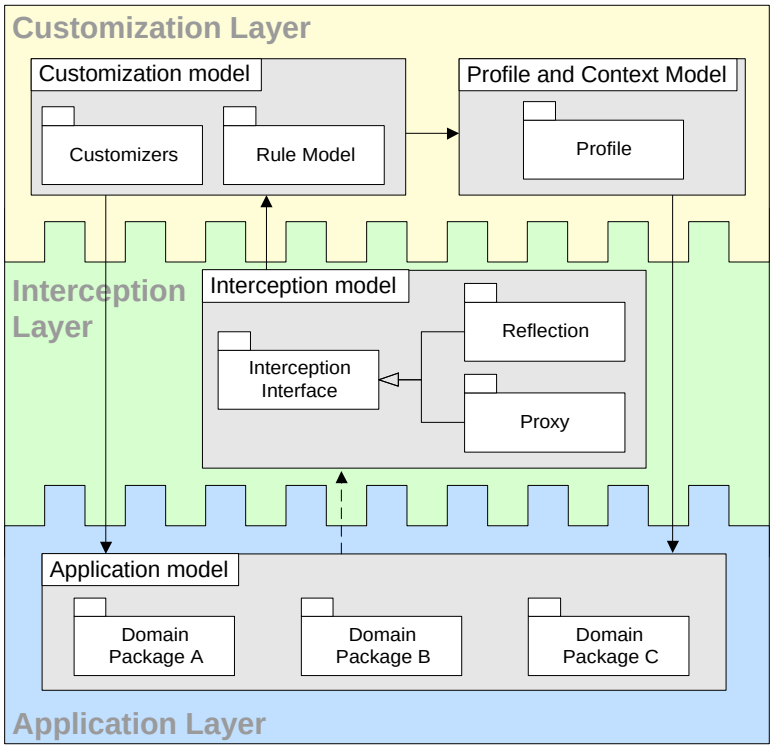

Fig. 1. A three layered model for personalizable applications.

4-Triggering Personalization Code

In the approaches cited in this paper it is clear that rules interact with domain code and the user profile, but it is still not clear how the customization code is triggered or, in the case of the event/condition/action model, how the event is generated. The purpose of this layer is to avoid an explicit reference from the application layer to the customization layer, allowing them to evolve in an independent way.

In this section we present different ways of solving this problem; the chosen one may depend on the language used and the degree of decoupling desired between the application layer and the interception layer. Due to space restrictions we just present a short description of the approaches and their main advantages and disadvantages.

4.1 Reflection

Maybe the less intrusive approach, it strongly depends on the reflective capabilities of the language. The idea is to intercept the personalized method invocation and delegate it to an object that knows how to trigger the right personalizer and, when appropriate, send the primitive message to the personalized object.

This approach can be easily implemented in environments like Smalltalk (which naturally support reflection) by capturing messages as shown in [Brant 98] and redirecting it to a message dispatcher. Ideally this is the best solution since there is no need to change the application's code to incorporate the personalization logic.

4.2 Proxies

As stated on the Proxy pattern [Gamma 95], the idea is to put an object between the client and the real object, so that the client thinks it is sending messages to the target object, when the proxy is intercepting them and solving its execution by collaborating with both, the primitive object and eventually a personalizer.

The main problem with this method arises when types are involved (in languages like Java) since to effectively replace one object with another, they have to belong to the same hierarchy; if this situation is not considered at design time (or if we later decide to customize an object) the hole domain hierarchy has to be redesigned.

4.3 Dependency-like mechanism

A well-known technique implemented in Smalltalk is the dependency mechanism: an object (the client) registers itself to another object (the observed) so that when the observed changes, the client 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 indicating which aspect has changed. 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]).

Following this line, we can 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.

Since generally it is not possible to modify de definition of the top class in the hierarchy (usually *Object)* there are (at least) two approaches:

- Define a class *CustomizableObject* that implements the required behavior and make all the customizable objects' classes' subclass of it instead of *Object.*
- Define a Singleton class [Gamma 95] which listens to the requests of all the personalized objects.

5-From Personalization Policies to Personalization Rules

As previously said we want to emphasize the fact that not all personalization policies should be mapped into software rules (even if they are abstractly specified in that way as in [Kappel 01]). For example, how should we map a policy such as: "Information should be personalized according to the user address (e.g. suggesting books that are bought in his country)"? The naive solution would be to map this policy into a set of rules checking the user profile for his address and acting in consequence.

In the same way, we could personalize the information the user perceives to the type of interface appliance he is using, by writing a set of rules that according to the artifact (PDA, internet browser, cellular phone) invokes a different method for generating the corresponding information. In both cases we end with a flat set of rules checking the type of an object (that we have hidden in a string variable).

On the other hand, it is clear that a policy such as: "Discount a 1% of the regular price for each 10 books the customer bought" is easily mapped into a software rule such as If customer.account.buyedProducts > x Then price:= price- y%. Mapping this clause into a rule object implies creating a class for the corresponding condition and action.

However, it is easy to find applications in which the same aspect of an object (e.g. the product's price) should be personalized with policies that combine the two previously shown examples, e.g. "Customers that live in *x* should receive an additional *y*% discount". In this case we would write two different rules and use a conflict solver if necessary (e.g. if we do not want the discount getting too high).

Our thesis is that the two rule are essentially different; while one of them just checks the value of an attribute the other one is just hiding a polymorphic behavior of Country objects related with discounts. In the following sections we refine this discussion.

5.1 Polymorphism vs. Rule Objects

Even though rule objects represent a good solution for decoupling personalization code from base application behaviors, their use should be cautiously evaluated to avoid large sets of flat rules. Our approach is based on the basic idea that rules should not replace polymorphic behaviors; in other words, when the condition of a rule deals with the possible types of an object we should replace rules' condition with operations that delegate to an object belonging to a class hierarchy of polymorphic classes, each of them providing the corresponding implementation of the policy action.

This hierarchy may be usually part of the user profile in which some former string objects will evolve into full-fledged polymorphic objects. In most real applications we have found very easy to identify these situations; for the sake of conciseness we only give some examples related with the previously mentioned e-store.

Suppose that we want to personalize the price of a product according to the buying history of the customer. Once we intercepted the message *price* a Price personalizer may trigger a Price rule (or a set of rules) that are clearly condition/action rules.

On the other side, suppose that we want to personalize the checkout process according to some user choice or role. For example, some users may have a pre-defined paying or wrapping option (this is called one-click checkout in amazon.com). A naive solution is to have some attributes in the user profile allowing us to execute the corresponding action in a rule. In this case our rules will query the user profile for the value of those attributes and decide the corresponding checkout step. These rules may be difficult to maintain, as they will have to evolve together with the checkout behavior.

A better solution is to let the corresponding Personalizer object (for the store *checkout* behavior) delegate to a user profile object that itself triggers the corresponding checkout step(s). Instead of having an attribute that indicates the user choice, this attribute is an object that performs the desired action. Notice that the user profile now contains more "intelligence": it knows which objects it has to invoke to perform the checkout. We may have different sub-classes for activating this behavior. From the point of view of design complexity this solution requires creating less classes than the rule-based schema, where we need sub-classes for actions and conditions and thus it is easier to maintain.

5.2 Composing Personalizers

It may happen that we need to implement different personalization policies for the same application aspect (e.g. the price of products). Even if we implement all of them as rule objects, we need to provide a way to solve conflicts (for example two different discount policies that should not be applied together).

In the most general case we may want to combine rule objects with other polymorphic behaviors; suppose for example that we want to apply different discounts to members of different purchase circles. Instead of having one condition for each circle we may just delegate the computation of the discount to the purchase circle of the actual customer.

Our approach for solving this problem is treating Personalizer objects as recursive composites [Gamma 95]; the leaves of the composite may be either rules or delegating objects and in each composition level there may be a conflict solver (the composed object whose parts may have conflicts). Then, for each personalizable aspect in our application we have a personalizer object; this object may just initiate the execution of a set of rule objects, may delegate to some "intelligent" profile object (that itself will delegate in some "application" object) or may just recursively delegate to other personalizers.

6-Conclusions and Further Work

In this paper we have outlined an alternative approach for dealing with personalization policies. This approach complements existing work by showing when policies should be implemented as rules, when using polymorphic objects or combination of both types of design structures.

We are now building a tool that let a designer to configure his personalizer objects easily. We are also studying better ways to document a design in which we are using (recursive) personalizers.

7-References

[Arsanjani 99] A. Arsanjani. "Analysis, Design, and Implementation of Distributed Java Business Frameworks Using Domain Patterns" in Proceedings of Technology of Object-oriented Languages and Systems 30, *IEEE Computer Society Press* 1999, pp. 490-500.

[Arsanjani01] A. Arsanjani. "Rule Object Patterns", In Proceedings of PloP 2001.

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

[Cappi 01] J. Cappi, G. Rossi, A. Fortier, D. Schwabe: "Seamless Personalization of E-Commerce applications" in Proceedings of eComo 01 (Workshop on Conceptual Modeling in E-Commerce), Japan, December 2001.

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

[Kappel 01] G. Kappel, W. Retschitzegger and W. Schwinger. "Modeling Ubiquitous Web Applications: The WUML approach". International Workshop on Data Semantics in Web Information Systems (DASWIS-2001), co-located with 20th International Conference on Conceptual Modeling Workshop (ER2001), Yokohama, Japan, November 27-30, 2001

[Koch 02] N. Koch, M. Wirsing: "The Munich Reference Model for Adaptive Hypermedia Applications", to be presented at 2nd International Conference on Adaptive Hypermedia and Adaptive Web Based Systems

[Krasner88] G. Krasner and T. Pope: "A cookbook for using the Model-View-Controller user-interface paradigm in Smalltalk", Journal of Object-Oriented Programming, 1(3), 26-49, 1988.

[Rossi 99] G. Rossi, D. Schwabe, F. Lyardet: "Improving Web Applications with Navigational Patterns". Proceedings of the 8th. International Conference on the WWW (WWW8), Toronto, Canada, 1999.

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

[RuleML 02] The Rule Mark-up Language, in http://www.dfki.uni-kl.de/ruleml/

[Schwabe98] D. Schwabe, G. Rossi: "An object-oriented approach to web-based application design". Theory and Practice of Object Systems (TAPOS), Special Issue on the Internet, v. 4#4, pp.207-225, October 1998.

[Schwabe 02] D. Schwabe, G. Rossi, R. Guimaraes: "Cohesive design of personalized Web applications", IEEE Internet Computing, March 2002.

[UML00] UML reference manual. In www.rational.com/uml.htm