Seamless Personalization of E-Commerce Applications

Juan Cappi[1]. Gustavo Rossi[1]. Andrés Fortier[1]. Daniel Schwabe[2]

1 LIFIA-Facultad de Informática-UNLP, Argentina
{jcappi,gustavo,andres}@lifia.info.unlp.edu.ar
2 Depto. de Informática, PUC-Rio, Brazil
schwabe@inf.puc-rio.br

Abstract. In this paper we present an original approach for personalizing complex Web applications, in particular e-commerce applications. This approach is based on a clear separation of concerns, namely: base application functionality, user profile management, and personalization rules, and supports seamless addition of personalization features (such as recommendations, special offers, individual interfaces, etc). We first explain our view of e-commerce applications as views on application models, and briefly explain why personalization functionality should be dealt by separating concerns. We next introduce a simple example and focus on different personalization patterns, emphasizing on behavior personalization. We show which design structures are the most appropriated for obtaining seamless extensions to existing software. We finally discuss some further aspects in building customized e-commerce software.

1-Introduction

Personalization has become a very important issue in e-commerce applications. The increasing popularity of the World Wide Web, the myriad of platforms supporting some kind of browsing, and the very nature of e-commerce have reshaped this problem. Not only we have to build applications customized to the individual; we must also cope with constant changes of personalization policies and rules (that in general follow the evolution of the business model itself).

Designing personalized e-commerce applications implies dealing with different concerns. It may mean building different interfaces (customized to a particular appliance), providing personalized navigation paths; offering different pricing policies, customized check-out procedures, etc. All these features involve managing information about the user (and his profile), implementing specific algorithms and interfaces, designing and applying rules and other related design problems.

User modeling and profile derivation from internet-gathered information has been extensively discussed in the literature [Perkowitz 00]. Recommendation algorithms and mechanisms (such as push vs. pull) have been recently surveyed [Schafer 99] and almost all electronic stores now include some kind of personalized behavior. A broader approach for understanding general requirements for customized Web Applications is presented in [Kappel 00].

However, while most of the “algorithmic” issues of personalization have been already studied, little attention has been paid to the modeling and design process of this kind of software; in particular, which design structures may help to cope with its increasing complexity. This aspect is critic not only because of the difficulties inherent to the problem but also because of the evolvable nature of (personalized) e-commerce applications. More precisely, as personalization requirements and policies change over time, software maintenance becomes a nightmare.

Suppose for example that product prices are personalized according to the buying history of the user; however, on a particular week there is a sale for certain products for which the mentioned policy does not apply. Or we decide to change the discount policy and apply it only when the user spent more than a certain amount of money. How do we guarantee software stability in this context? How do we simplify the addition of new rules? How do we support new features in the user profile that were not foreseen? How do we avoid melting business rules into the application code? The answer is separation of concerns.

By clearly understanding and decoupling the design concerns involved in a personalized e-commerce application, we can keep the software manageable; we can also provide a conceptual framework for reusing designs and design experience. We have a better way to understand the kind of interactions appearing in a personalized application from an abstract point of view (independent of the specific aspects of the application), and to simplify them by following existing design patterns.

We show that a clear separation of those concerns related with personalization allows adding personalized behavior to applications conceptual models with a minimum amount of code manipulation. We stress that reasoning over design objects gives us better insight on the personalization process.

The structure of this paper is as follows: We first introduce the OOHDM approach for building e-commerce applications. Next, we discuss the problem of designing personalized applications. We identify the most common personalization patterns and analyze the particular design concerns they involve. We introduce a stereotypical example to illustrate our ideas briefly and show personalization hot spots in the example. We next introduce personalized wrappers and rule objects; for each design mechanism we discuss the problem which motivates it and its solution. We finally explain how to scale-up towards a generic architecture for personalized applications and discuss some ongoing research issues.

2-Our View of 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 [Schwabe 98]. Those models are built using object-oriented primitives with a syntax close to UML.

The concern of the conceptual model is to represent domain objects, relationships and the intended applications’ functionality. As we show later in this paper, most personalization mechanisms involve dealing with objects and algorithms that are expressed as part of the conceptual model.

The cornerstone of the OOHDM approach is that the user does not navigate through conceptual objects, but through navigation objects (nodes). Nodes are defined as views on conceptual objects, using a language that is similar to object database view-definition approaches. As we consider Web applications as hypermedia applications, we define links connecting nodes, as views on conceptual relationships. Nodes are further organized in sets called navigational contexts. Indexes provide shortcuts for accessing nodes. The complete syntax for nodes, links, contexts and indexes definition can be found in [Schwabe 98]. OOHDM provides a straightforward way to build applications customized to different user roles building different navigational models (views) for each profile.

Finally, the abstract interface model specifies the look and feel of navigation objects together with the interaction metaphor. Separating the interface from the navigation specification allows us to cope with varying interface technologies in a modular way. As interface objects are specified as Observers [Gamma 95] of navigation objects, we also have a simple way for customizing the interface to different Web appliances (as browsers, cellular phones, etc) [Schwabe 98].

3-Modeling and Design of Personalized Applications

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. Following the OOHDM framework (that partitions the design space into conceptual, navigational and interface models) we can personalize:

- the algorithms and processes described in the conceptual model.
- the contents and structure of nodes and the link topology in the navigational model
- the interface objects and their perceivable aspects and the interaction styles.

These personalization aspects can be addressed at the role or at the individual levels. As OOHDM naturally supports role-based personalization we do not discuss it further in this paper. For the sake of conciseness we will neither discuss interface personalization though the basic principles discussed later can be easily apply to this aspect.

Once we understand the different personalization patterns in an abstract way, we must discuss how to relate specific applications' requirements with these patterns and then map these patterns onto concrete designs. For example, if we need to personalize products' prices in an e-store, we first realize that this is a particular case of node content personalization and then we use the most appropriate abstractions for materializing design. In this paper we present some general design solutions to these problems; we present them in a domain and application independent way, so they can be reused in different systems and modeling approaches.

There are basically three concerns related with personalization: the user profile, the personalization rules and the application of those rules for a particular individual. Even in trivial e-commerce applications it can be shown that hard-coding some of these aspects in core application objects may yield a difficult to maintain application

as shown in section 4. To make matter worse, we usually need to improve existing applications adding personalization features that were not foreseen (see for example the evolution of amazon.com, cnn.com or yahoo.com); we should do it without affecting existing application logic.

The key for obtaining a good design is recognizing that the pace of evolution of customization rules (and the associated user profiles) is faster than changes in the basic application model so they must be clearly decoupled. We next introduce an example to discuss these issues more concretely.

4-An Example

Suppose an electronic store that sells *products* to *customers.* The customer may put products in his *shopping cart*, indicating the number of items of each product. 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. In Figure 1 we present an object model representing key features as classes.

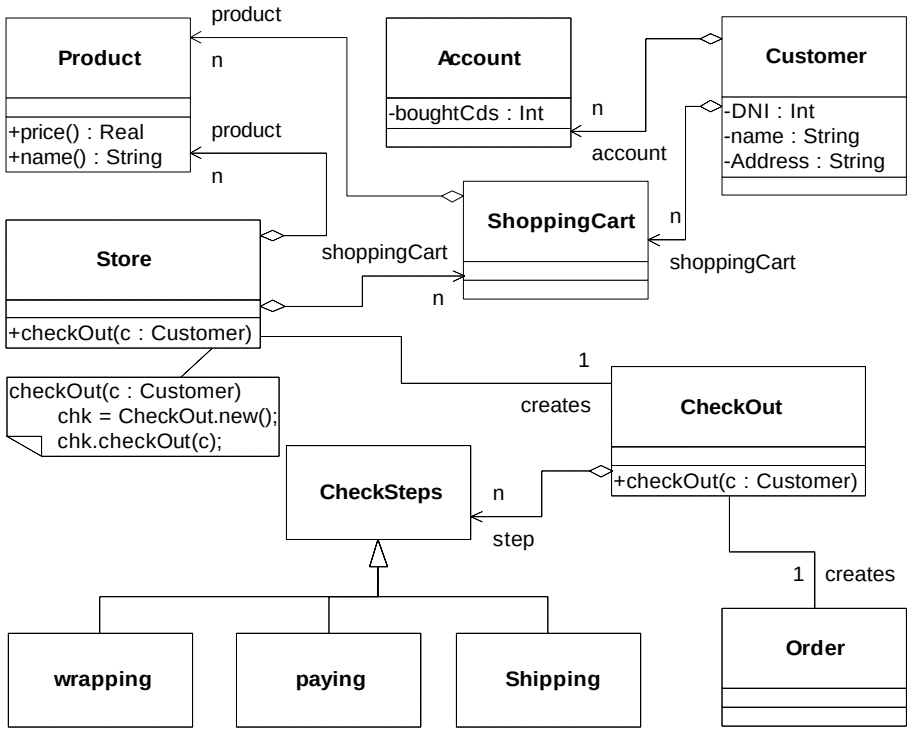


Fig. 1. Conceptual Model for the e-store.

While variations in the domain model such as adding new products or paying mechanisms can be solved by sub-classing and composing objects, if we want to

introduce some personalization capabilities, the problem gets worse. For example suppose that we want to personalize product prices; one possibility is to let products delegate the price computation to the corresponding customer object as shown in Figure 2. The same strategy can be used with other object' attributes.

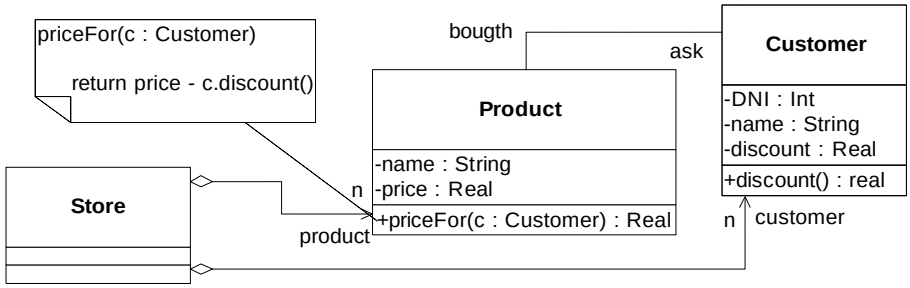


Fig. 2. Hard-coding Personalized Prices in products

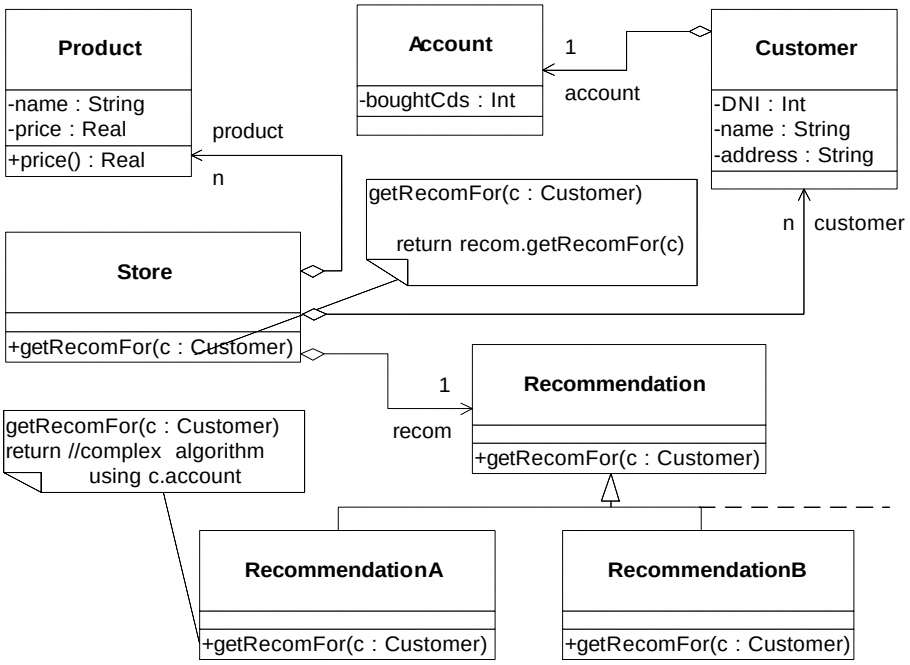


Fig. 3. Adding recommendations to the base model

The problem with this approach is that it works well if no new customization rule must be added or if we don't need different rules in different use contexts, e.g.: when the user browses the product, he gets one price (his price) but when he checks-out, the price may be personalized differently (for example taken into account a special offer when buying some combination of products). If this happens we will end re-designing core business classes constantly and sooner or later the code will become messy.

Under those conditions, it is interesting to notice that recommendations, in fact the most popular kind of personalization feature, are very easy to engineer as shown in Figure 3, where algorithms are implemented using the Strategy design pattern [Gamma 95]. Instead of implementing the recommendation as a method in class Store, different recommendation algorithms are organized in a class hierarchy and thus we can switch algorithms dynamically. Recommendation algorithms will thus interact with customer objects and their accounts to obtain the desired results.

However, if we have different algorithms according to the user profile, the solution in Figure 3 does not scale-up as it may involve a complex if clause for associating the algorithm to the profile. The problem is that we are coupling the basic business behavior (e.g. in class Store) with one that depends on the user profile.

In the following section we elaborate this discussion and show a set of design micro-arquitectures for coping with personalization complexity.

5-Separating Personalization Concerns

The key strategy for obtaining evolvable personalized e-commerce applications is decoupling the main concerns of these applications. From the above discussion we conclude that the core application logic should be separated from personalization rules. Other authors have also suggested keeping business rules separated from business objects as the former tend to vary faster than the latter [Arsanjani 99, Yoder 01].

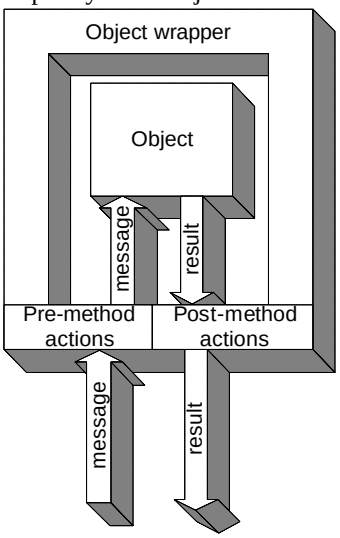
As said before, the personalization logic itself involves three different aspects: the rules, the user profile and the application of those rules to particular business objects. Some of these concerns may partially overlap with the core objects as shown in the example. Even though the application was not conceived as personalized, it contains objects that will be useful as part of the customer profile (as Customer itself and its account), and behaviors for keeping the user profile up to date (such as those adding products to the customer's account). We next show step by step our approach for adding personalized behaviors in a non-intrusive way.

5.1 Personalizer Wrappers

The first problem to solve deals with changing the behaviors of some core business objects to personalize them. A simple instance of this problem is the method returning the price of a product. A more elaborated example can be found if we want to personalize certain steps of the checkout procedure. There are two constraints in this problem: we don't want to mix the basic code in the product with the "new" code for personalization and we also want to be able to easily "switch" these personalized behaviors in different contexts, e.g. prices when you buy one product, many products, in a special offer, etc.

The best solution to this design problem is based on the Decorator pattern [Gamma 95], which is used to dynamically add new behaviors to a given object. For each class we want to personalize we can define a corresponding wrapper and connect it to the personalized object in an instance basis (i.e. some objects may not be decorated). The

solution is shown in Figure 4, where we can see (in 4.a) the concept of wrapping. It consists in intercepting messages, eventually processing them without intervention of the base object and returning the result. Wrappers allow changing the behavior of an object without re-writing its class; as wrappers work at the instance level, we can wrap only some objects in a class. Figure 4.b shows the corresponding UML diagram.



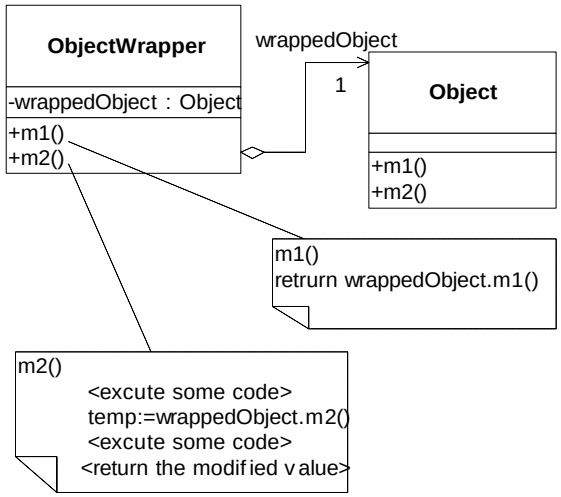


Figure 4.a: Functionality of a Wrapper

Figure 4.b: UML design for wrappers

Fig. 4. Decorators for personalized behavior

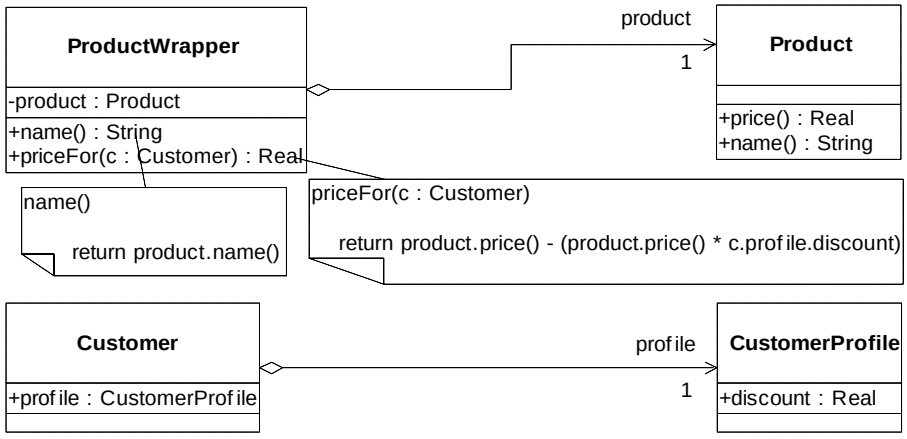


Fig. 5. Implementing personalized behaviors in wrappers

A first approach would be to allocate the personalized behavior in the wrapper, which might end up collaborating with other objects to complete the task. Following with the example of the product's price, the wrapper traps the message *price*, asks the user profile the discount that should be applied and calculates the real price (Figure 5).

Those behaviors that are not personalized are simply delegated to the original object. This same idea can be used for personalizing the checkout process or some of its steps (delivery and paying options, etc).

Even though this solution seems to work fine, we might think on applying a different policy to each particular customer. Of course, it is completely unacceptable to write a wrapper for each aspect for each customer (think of personalizing three aspects of an object, which can be handled in four different ways each: you end up writing 64 wrappers!). Other approach would be to write a potentially huge case statement to decide which algorithm to apply; there is no need to say that this idea won't scale up and the code ends completely messed up. An elegant solution to this problem can be addressed by using the Strategy pattern [Gamma 95], which decouples the algorithms from their implementation. Using this approach, the customer profile delegates the behavior for obtaining the discount to an other object; the wrapper would be just in charge of triggering its execution after getting the corresponding algorithm as shown in Figure 6. Notice that we just delegated the task of selecting which algorithm to apply to the Profile object. Now, we are able to choose different algorithms depending on the customer, and as a side effect we can even change those algorithms in run time as the user profile evolves.

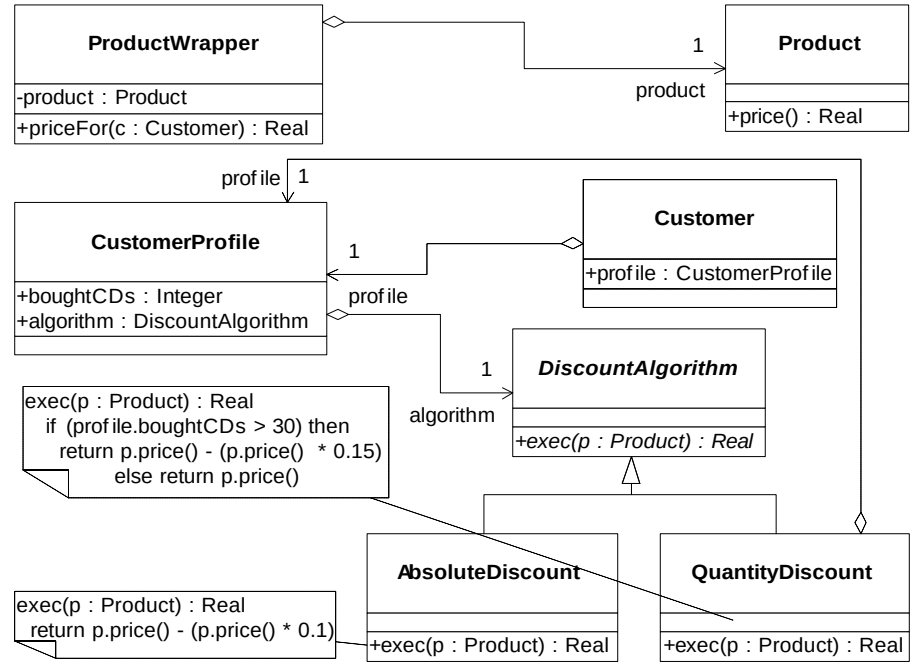


Fig. 6. Decoupling algorithm implementation

Since the way in which we personalize an aspect of an object is generally more than just an algorithm, we may call the wrappers 'personalizers', i.e. objects that take care of personalizing a particular aspect. Summarizing, the main idea is that we engineer each personalized aspect separately and configure the wrapper as necessary, maybe in a per-instance basis.

To show how this approach helps us to cope with more complex situations, suppose that we need to personalize different aspects of the same object for different users; for example while one user has a personalized price, another has a personalized delivery mechanism. This is the typical case when users can select what can be personalized such as in my.yahoo.com. The solution is to delegate the responsibility of handling the appropriate personalizer to the corresponding user profile object, as shown in Figure 7 (where we personalize price and comments on a product). In this solution we further decouple the process of intercepting the message from the personalization code.

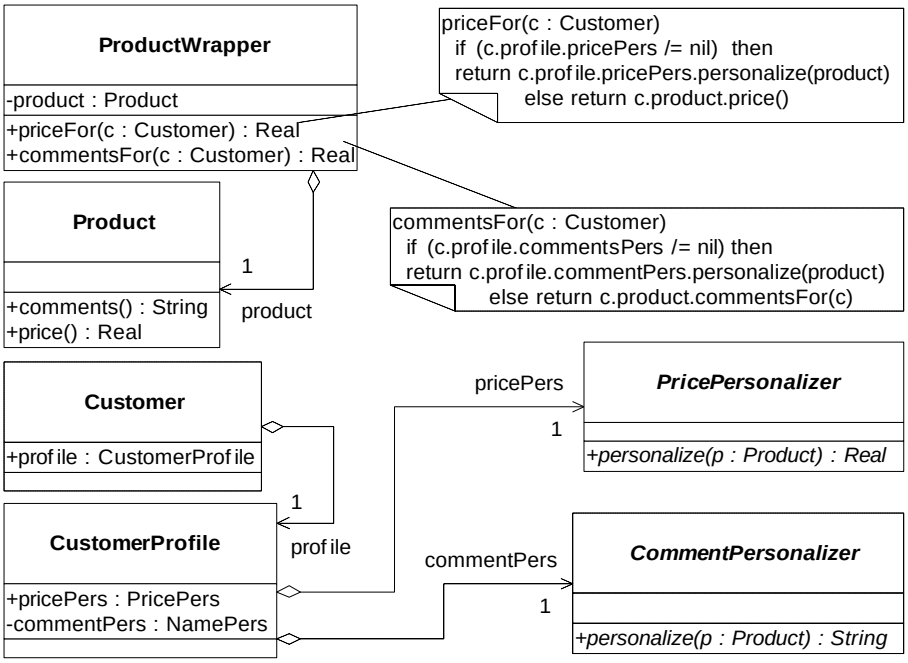


Fig. 7. Personalizers that depend upon the user profile

The product wrapper (See method: *priceFor*) in Figure 7, checks if the profile contains the corresponding personalizer; if it does not exist it just delegates the behavior to the corresponding product object.

This last design can be improved by using the Null Object pattern [Woolf 96], which helps us eliminate the if statement in the Product Wrapper. In Figure 8 we show how we define a hierarchy of personalizers. While *PricePersonalizerA* and *PricePersonalizerB* implement different price policies, the *NullPricePersonalizer* just delegate the task to the original Product object. Null objects are an elegant way to obtain uniform code because they help to eliminate complex if clauses.

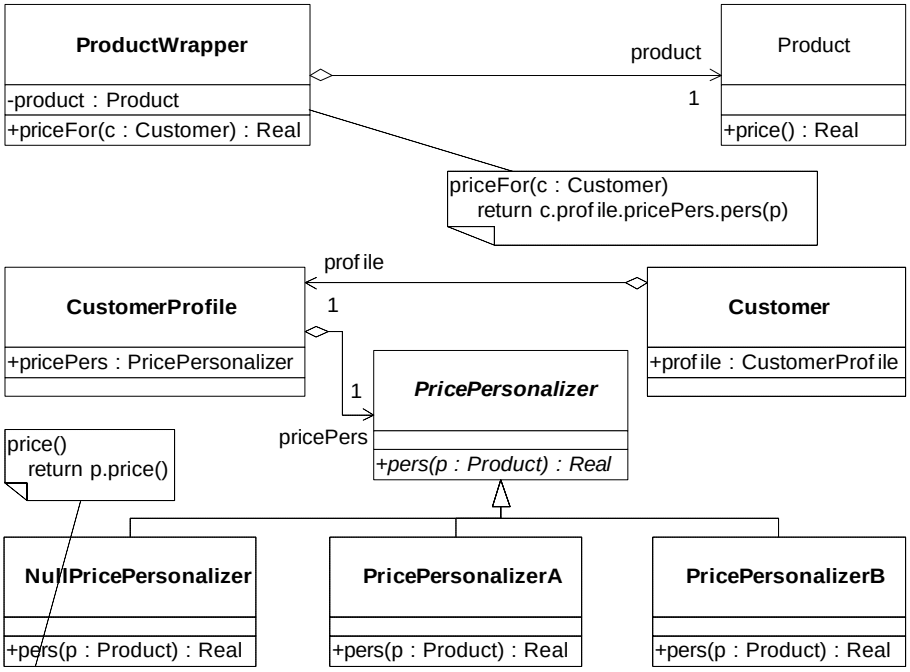


Fig. 8. Using the Null Object pattern

Personalizers allow decoupling the core business model from the personalized behaviors. However when customization rules are complex, we need to further decouple the rules from these objects as shown in the next sub-section.

5.2 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*. As rules are not first-class objects we need to analyze the best way to map them into design constructs.

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 best solution is to decouple Conditions and Actions from the Rule as shown in Figure 9. Moreover, when mapping this design to modern object-oriented languages (such as Java or Smalltalk), conditions and actions can be implemented as light-weighted objects such as blocks (or inner classes in Java) thus simplifying the design shown in Figure 9.

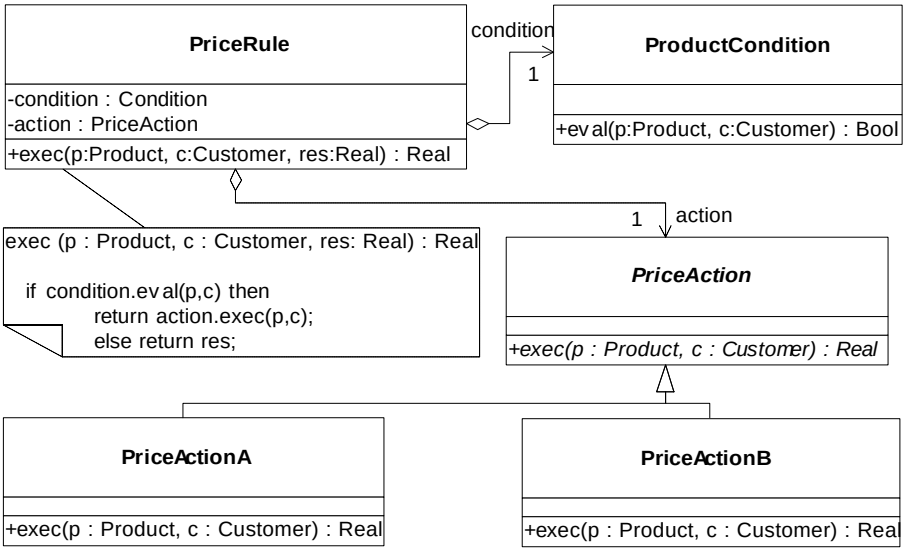


Fig. 9. The design of a rule

Using Conditions and Actions as shown in Figure 9, we can configure different price rules by combining different instances of PriceAction and ProductCondition. The reader should notice that this approach is easy to generalize by defining abstract classes for Conditions and Actions.

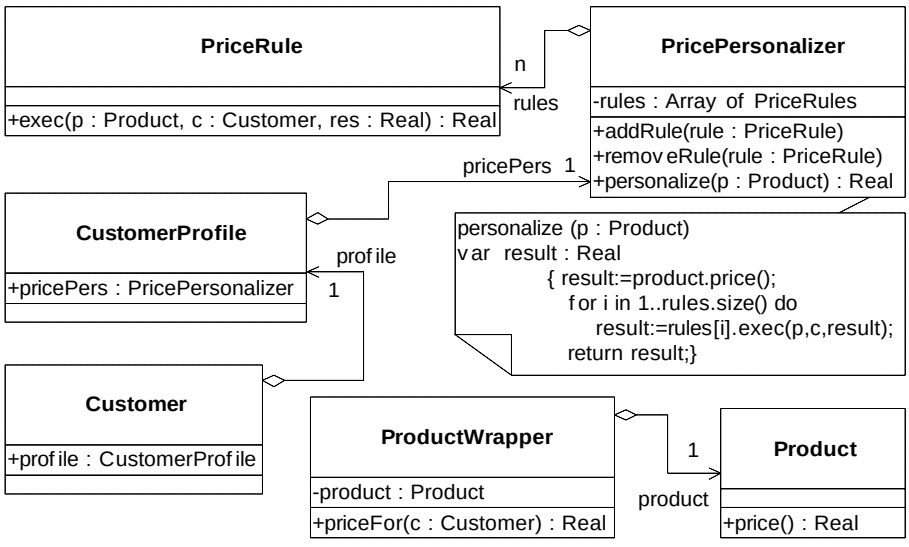


Fig. 10. Decoupling rules from personalizers

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. The solution is to design rules as separate artifacts (thus, decoupling them from personalizers) in order to simplify their organization. This solution, shown in Figure 10, treats rules as objects that may be triggered from personalizers or eventually from other objects as explained in Section 5.4.

Notice that personalizers act as Facades [Gamma 95] and they may eventually implement a policy for handling rules, for example when the same aspect may be personalized using different rules. We can also reuse whole rules when they may be applied to different objects (e.g. product sub-classes, or interfaces), or even define hierarchies of personalizers that may be applied to different products.

5.3 Dealing with evolving user profiles

The user profile is an important component in every personalized application. As shown before, the user profile may not only contain plain information about the user such as his shopping history (see Figure 1) but may also contain rules, references to algorithms, etc. In our design approach the user profile should be clearly separated from business objects and should grow in a transparent way. However this is not always possible: many information stored in the user profile is generated by core business operations (e.g.: buying a product). Personalization rules may even require this information to evolve: suppose for example that we want to apply some price reduction to those customers who add some comments about products (for example after reading a book). In this case, the evolution of the user profile implies an interaction that will be handled (at least at first) by business objects and will require adding some methods and even changing the structure of the profile. From the designer point of view, however, it is important to characterize operations that affect the profile, i.e. those that add information, those that associate personalization rules with it and those that allow rules to query information about the user. These operations should be kept isolated and be decoupled from the rest of the business model when possible.

5.4 Putting all pieces together

In the previous paragraphs we have discussed the most important design problems a developer has to face when personalizing e-commerce applications. Each critical concern has been consciously decoupled yielding a modular architecture that can evolve seamlessly together with the business model's evolution. Each component (i.e.: wrappers, rules and the user profile) can be engineered separately provided that they conform to common protocols. The ideas underlying this architecture can be used as they are, or just as guidelines (or patterns) during e-commerce application building. For example, it is possible to get simpler designs when the forces that drive the application's evolution do not require that all components are completely decoupled (e.g. writing the rules code in personalizers).

It is interesting to remark that these micro-architectural constructs can be used to customize e-commerce applications, even when the customization conditions do not involve individual users. For example suppose that we need to implement a marketing strategy that reduces products' prices during Christmas season. We can also think about wrapping products that intercept this behavior and delegate it to a set of rule objects that check the environmental conditions (e.g. date) and decide the corresponding action.

All these components can be materialized in abstract classes that can be eventually sub-classified for each particular application. The process of personalizing an e-commerce application will then consist in creating instances of corresponding wrappers, personalizers and rules and "plugging" them into base classes as shown earlier in this paper.

6-Conclusions and Further Work

In this paper we have presented a design approach for building personalized e-commerce applications. This approach is based on a clear separation of concerns, namely base business logic, user profile, personalization rules and the application of rules.

We have shown that personalization issues should be carefully considered when building the application's model. In particular we claim that design components related with personalization should not be hard coded in core application objects, but rather built as separated artifacts and conveniently composed with the objects they personalize. We showed that applying well-known patterns (such as Decorator, Strategy and Facade) we can seamlessly extend e-commerce applications with personalization features. The approach is non-intrusive because it provides architectural mechanisms for adding extra functionality (e.g. personalization) without re-coding the base business logic.

We are now developing visual tools for simplifying the personalization process by allowing the designer to plug personalizers to base objects easily. This approach has been used successfully for letting designers describe rules visually [Yoder 01].

We are finally exploring how to extend our approach to a more general scope of customization, not limited to individual customization. The whole architecture can be reused, though rule managers should provide additional functionality for other customization rules. In this case, wrappers should also provide the context of customization (thus allowing rules to "know" which objects may contain information useful for the process). Our approach can then be applied to more general frameworks like the one in [Kappel 00] to solve the problem of ubiquity and customization in e-commerce.

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

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

[Kappel 00] G. Kappel, W. Retschitzegger and W. Schwinger. "Modeling Customizable Web Applications - A Requirement's Perspective". In Proc. International Conference on Digital Libraries: Research and Practice, Kyoto 2000.

[Perkowitz 00] M. Perkowitz, O. Etzioni: "Adaptive Web Sites" In Comm ACM, August 2000, p.p. 152-158.

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

[Schwabe 98] 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.

[Schafer 99] Schafer, J. B.; Konstan, J.; Riedl,, J.; "Recommender Systems in E-Commerce", Proc. of E-Commerce'99, Denver, USA, ACM, 1999.pp.158-166.

[Woolf 96] Bobby Woolf: "The null object pattern". Proceedings of PloP'96, Pattern Languages of Program Design, 1996.

[Yoder 01] J. Yoder, F. Balaguer, R. Johnson: Adaptive Object-Models: With Application to Medical Observations. To be published in Proceedings of OOPSLA 2001.