Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Universidad Nacional de La Plata - Argentina
2001

Personalization in Object-Oriented Systems

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange program)

By: Sofie Goderis

Promotor: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Prof. Dr. Gustavo H. Rossi (Universidad Nacional de La Plata)

Abstract

Personalization is a concept with raising importance in web-applications. It intends to adapt an application to the user's individual needs such that each user gets the idea that the system was created just for him/her and knows what he/she likes. Although it might seem obvious to personalize web-applications, personalization should be viewed in a broader perspective and the ideas of personalization can for instance be applied onto object-oriented systems. The dissertation discusses the personalization of object-oriented systems. The goal of this dissertation is to investigate how, why and where we can introduce personalization in object-oriented systems. We will do so by first studying existing approaches and make a comparison. Then we will introduce our approach for personalizing systems, i.e. logic meta-programming, a declarative meta-language that lets you reason about the structure of an object-oriented base-language. We will prove our proposed approach by showing examples for each different type of personalization.

Contents

List of Figures

Acknowledgements

This dissertation obviously would never have been realized without the tremendous support which was given to me. Therefore, I wish to express my gratitude towards :

Prof. Dr. Theo D'Hondt and Prof. Dr. Gustavo H. Rossi for promoting this dissertation, and for supporting me.

Isabel Michiels, Johan Brichau and Tom Tourwé for proofreading my work and their valuable advice.

Annya Romanczuk for being the spirit behind EMOOSE, together with all the other people that put a lot of effort into this master program.

My fellow students, friends and family for being there.

Chapter 1

Introduction

The object-oriented paradigm has been around for the last three decades, but only recently it became increasingly popular. To the larger public it is still quite "new" and industry just started using this paradigm on a bigger scale. However, during all these years a lot of research has been conducted, and still is being carried out. All the work conducted so far lead to interesting, and far developed, analysis and design techniques. Furthermore implementation techniques, and reuse techniques are well known.
Combining object-orientation with other research areas such as personalization, leads to interesting solutions. Using an object-oriented approach to design and implement web-applications for instance leads to better evolvable and maintainable systems [SR98].

Personalization is a concept with raising importance, especially in web-applications. It intends to adapt an application to the user's individual needs such that each user gets the idea that the system was created just for him/her and knows what he/she likes. Although it might seem obvious to personalize web-applications, personalization should be viewed in a broader perspective and the ideas of personalization can for instance be applied onto object-oriented systems. Furthermore web-applications are becoming more object-oriented as well, with an object-oriented analysis, design and (partial) implementation.

When looking at different approaches to combine personalization with object-oriented systems, there are four important approaches. A first approach is the hard coded approach, where personalization is hard coded into the system. This is obviously not advisable because of maintenance and evolution problems.
A second approach makes a split between the conceptual level and the navigational model. Personalization is then added to this second model such that the basic system (conceptual level) is not affected by personalization. This makes the system better evolvable, but nevertheless there is still a hard-coding of personalization at navigational level.
Thirdly, we find a decoupling of the basic system and its personalization at base-level, either by the use of design patterns, components or value-models.
Finally a fourth approach decouples at meta-level. The basic system then resides at base-level, while the personalization for this system resides at meta-level. Using this approach allows a clear and clean separation between the basic system and its personalization. This makes it more easy to change personalization aspects without touching the basic system.

Thesis

This dissertation deals with personalization of object-oriented systems. Current uses of personalization are studied, together with why and where in object-oriented systems personalization tends to be useful. We discuss the different personalization aspects in object-oriented systems and we explain how these aspects can be introduced into these systems. We then demonstrate the integration of personalization in OO-systems by using a logic meta-programming approach.

In *chapter 2* we give an overview of the current practices of personalization. *Chapter 3* then explains why and where we want personalization in object-oriented systems. Furthermore different approaches to achieve this kind of application are explained. Next *chapter 4* deals with logic meta programming, which was chosen to be our approach to add personalization to object-oriented systems, and *chapter 5* gives a practical example of doing so. Finally in *chapter 6* we summarize our work and discuss some future research.

Chapter 2

Personalization

Before discussing our work, we give an overview of current work conducted with regards to personalization. We will shortly explain what it is, and where personalization is used, together with the different aspects that are important when personalizing (e.g. user profile).

2.1 What is Personalization

If we look up the term *personalize* in a dictionary (Merriam-Webster), we read the following definition :

> **personalize**
> *transitive verb*
> 1. personify
> 2. to make personal or individual; specifically : to mark as the property of a particular person

When we apply this definition to computer science, we can say that personalization indicates the efforts made to adapt an application to the individual user's needs.

2.1.1 Individual user's needs

In the beginning, when software systems were developed, the first step was to find out what the end-users expect from the system. However each individual might have different expectations, the final requirements grouped these expectations into one whole. The end-user would have to use the system as it was, because the system was not capable of considering the individual user.

In a next step applications started to consider different kind of user-groups. The advantage of this is that the system will be more adapted towards each user group. For instance a project manager will use the system in a different way than the secretary.

Nowadays, software systems, in particular web applications, adapt more and more often to the needs of the individual user. Depending on the user and his/her previous use of the system, the application might respond differently. For instance by giving a new user a lot of information on how to use the system, while this information can easily be skipped for advanced users.

2.1.2 Does personalization jeopardize privacy?

Personalization and privacy issues will always go together. Manber et al. [MPR00] state that any company collecting private information must guard that information with its life. There will always be tension between the use of personal data to improve service to users, and the use of the same data to derive profits for the company. That is why the designers of Yahoo! have full-time inside people that serve as champions of the consumer, as well as outside observers and auditors [MPR00].

Also the current laws have to be adapted, as Volokh discusses in [Vol00]. We won't go in further detail on these laws.

2.2 Current use

Almost all research conducted related to personalization is situated in the field of the world wide web, and more typically for e-commerce businesses, using recommendation algorithms. In this section some more about these kind of algorithms is said. Also the user profile is discussed, because it is an important aspect of personalization systems. The section finishes with personalization in other fields besides e-commerce systems.

2.2.1 Personalizing the web

Mulvenna et al. [MAB00] state that personalization aims to provide users with what they want or need, without having to ask for it explicitly. Personalization technologies involve software that learns patterns, habits and preferences, and on the internet its primarily use is in systems that support e-business. Personalization helps users to find solutions, and it empowers e-business providers with the ability to measure the quality of that solution [MAB00]. Initial attempts were limited to check-box personalization, where portals allow the users to select the links they would like on their "personal" pages, but this implies the users to know in advance the content of interest to them.
Collaborative filtering allows users to take advantage of other users' behavior (more on this in section 2.2.2). This personalization technique allows a more intelligent manner of achieving personalization, but still requires users to divulge some personalization information on their interests, likes and dislikes,...
Observational personalization attempts to circumvent the need for users to divulge any kind of personal information. Clues to how services, products and information need to be personalized is assumed to be hidden in records of users' previous navigation behavior. This technique consists of three principal components : analytics, representation and deployment [MAB00]. Web mining is currently the main technique to achieve the necessary *analysis* (section 2.2.3). *Representation* could be achieved with XML (section 2.2.3) and *deployment* means the obtained knowledge is carried out, for instance using recommender systems (section 2.2.2).

2.2.2 Recommender Systems

Recommendation is the most common personalization in e-commerce systems. It suggests products to the customers and provide consumers with information to help them decide which products to purchase [SKR99]. These products can be recommended based on the

demographics of the consumer, the top overall sellers on a site, or on an analysis of the past buying behavior of the consumer. Schafer et al. indicate that the three ways in which a recommender system enhances e-commerce are :

- Converting Browsers into Buyers : by helping consumers find products they wish to purchase.
- Increasing Cross-sell : by suggesting additional products for the consumer to purchase (e.g. based on the already chosen products).
- Building Loyalty : by creating a value added relationship between the site and the consumer through the presentation of custom interfaces that match consumer needs.

Taxonomy for recommender applications

Schafer et al. investigated six e-commerce businesses that use one or more variations of recommender system technology and conclude this research with a taxonomy for recommender applications (see figure 2.1) [SKR99].
The taxonomy separates the attributes of recommender systems in three categories : Functional I/O, Recommendation method and Other design issues.

Functional I/O can be subdivided in following subcategories :

- targeted customer inputs : used to provide personalized recommendations. These inputs can originate from implicit navigation, explicit navigation, keywords and item attributes, and purchase history. *Implicit navigation* is inferred from the customer's behavior without the customer's awareness of their use for the recommendation process. *Explicit navigation* indicates inputs that are intentionally made by the customer with the purpose of informing the recommender system about his/her preferences. *Keywords and item attributes* are implicit or explicit inputs, but extending the single category or item of interest, and *purchase history* is an implicit form of ratings.
- community inputs : how multiple individuals in the community, or the community as a whole, perceive items. These inputs include attribute assignments, external item popularity and community purchase history. The *attribute assignments* assign community based labels and categories to items. *External item popularity* reflects popularity in broader communities.
- outputs can be suggestions, predictions, and individual ratings and reviews. *Suggestion* is the most common type of output. This can be either the recommending of a single item to recommending of an ordered set of items. A *prediction* predicts the rating the customer would give to the item. *Individual ratings and reviews* are given by other community members and might be an indication to the individual user.

The Recommendation Method is the specific process that is used in actually e-commerce applications. The five methods are :

- raw retrieval : recommends whatever the customer has requested
- manually selected : recommending based on a manually created list by the editor, author, artist, critics,...

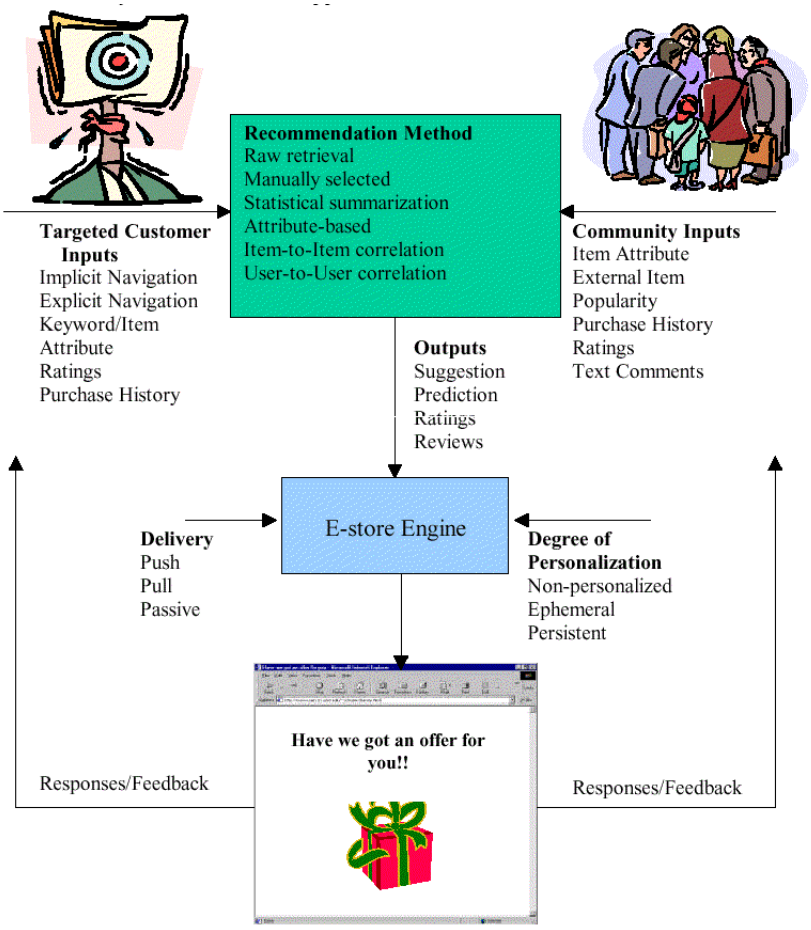

Fig. 2.1: Taxonomy for Recommender Applications

- statistical summaries of community opinion
- item attribute based : based on the properties of the items and customer interests
- item-to-item correlation : identifies items frequently found in association with items in which the customer has expressed interest.
- user-to-user correlation (collaborative filtering) : recommends products based on the correlation between the buying customer and other costumers who have purchased products from the e-commerce site.

Two Other Design Issues are degrees of personalization and delivery.
Recommendations might be produced at varying degrees of personalization. The degrees that are distinguished are :

- non-personalized : the same recommendations are made to each customer of the system.
- ephemeral personalization : the customers current inputs are used to customize the recommendation to the customers current interests
- persistent personalization : recommendation differs for different customers, even when they are looking at the same items.

Another issue is delivery, which is either :

- push delivery : delivering recommendations when the customer is not interacting
- pull delivery : customer controls when recommendation is displayed
- passive delivery : recommendation is presented in the natural context of the e-commerce application.

More on the taxonomy and recommender systems is to be found in [SKR99].

Collaborative filtering

Collaborative Filtering Collaborative Filtering is one of the earliest and most successful recommender technologies. It builds a database of preferences for products by consumers and any new consumer is matched against the database to discover *neighbors*. These neighbors are other consumers that historically have had a similar taste to the new consumer. Products that were liked by the neighbors, are recommended to this new user, as he will probably also like them [SKKR00].
Two fundamental challenges for recommender systems are its scalability (more and more potential neighbors are searched), and improving quality of the recommendations for the consumers. The two types of characteristic errors for recommender systems are :

- false negatives : products are not recommended, though the consumer would like them, and
- false positives : products are recommended, though the consumer does not like them.

In the e-commerce domain it is more important to avoid false positives, since these errors lead to angry consumers [SKKR00].

Automated Collaborative filtering Automated Collaborative filtering (ACF) systems predict a user's affinity for information or items [HKR00]. The filtering decisions in these kind of systems are based on human, and not machine, analysis of content. Each user of the system rates items that they have experienced, and doing so establish a profile of interest. The ACF system will then match the user with people of similar interests or tastes (i.e. neighbors). Using ratings from neighbors, recommendations are generated for the user.
The advantage of an ACF system is that it does not depend on error-prone machine analysis of content. However, there are several reasons why ACF systems are not trusted for high-risk content domains, namely :

- ACF systems compute predictions based on models that are heuristic approximations of human processes.
- ACF systems base their computations on extremely parse and incomplete data.

Therefore recommendations are often correct, but occasionally also very wrong. However, if users know the reasons behind a recommendation, they are more likely to trust that recommendation [HKR00].
Using an explanation facility in recommender systems, helps the user to understand the system, and previous work on expert systems (another type of decision aide) has shown that explanations provide considerable benefits [HKR00], such as :

- justification : the user understands the reasoning behind a recommendation.
- user involvement : the user is allowed to add his knowledge and inference skills to the complete decision process.
- education : the user may better understand the strengths and limitations of the system.
- acceptance : greater acceptance because the system's limits and strengths are fully visible to the user, and its suggestions are justified.

Herlocker et al. conclude that ACF systems in combination with explanations result in filtering systems that are more accepted, more effective, more understandable and give the user greater control. More on their research and experiments can be found in [HKR00].

Achieving Better Recommendations

Good et al. discuss in [GSK+99] that combining collaborative filtering with personal agents will lead to better recommendations. They combine collaborative filtering with information filtering agents. *Collaborative filtering recommenders* use the opinions of other users to predict the value of items for each user in the community, and *information filtering recommenders* look at the content of items to determine which are likely to be of interest of value to a user [GSK+99].

Figure 2.2 shows the four key models Good et al. use in their experiments : [GSK+99]

- Pure collaborative filtering using the opinions of other members of the community.
- A single personalized "agent"
- A combination of many agents (i.e. information filtering recommenders)

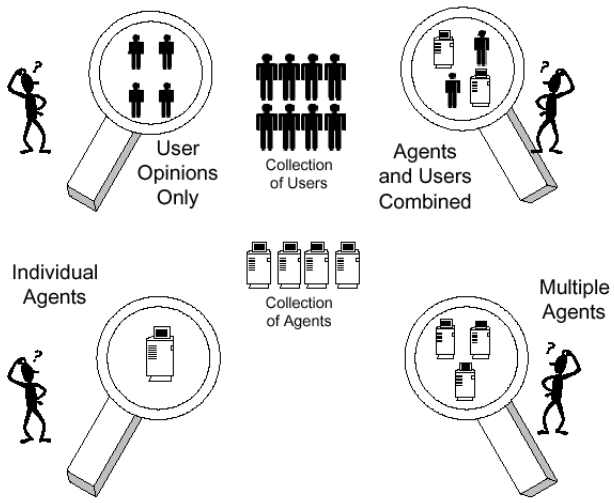

Fig. 2.2: Key Models to achieve better Recommendations

- A combination of community member opinions and multiple agents

The authors implemented and evaluated the different models, and state the following four conclusions :

- The opinions of a community of users (model 1) does not provide better recommendations than a single personalized agent (model 2).
- A personalized combination of several agents (model 3) provides better recommendations than a single personalized agent (model 2).
- The opinions of a community of users (model 1) does not provide better recommendations than a personalized combination of several agents (model 3) .
- A personalized combination of several agents and community opinions (model 4) provide better recommendations than either agents (model 3) or user opinions alone (model 1).

The authors proved that indeed a mixed collaborative filtering solution, using users and agents, does indeed provide better recommendation systems.

Recommendation helps to create queries

Belkin [Bel00] describes how recommendation can be used to help users in creating queries. In general, information systems require users to specify what they want the system to retrieve. However, people engaging in large-scale information systems typically are unfamiliar with the underlying operations of the systems, the vocabularies the systems use to describe the information objects in their databases, and even the nature of the databases themselves. This means that, when users seek information, the information system should recommend courses of action, which will help the users to better understand their problems, and thus to use the system's resources more effectively. [Bel00]
Belkin's research indicates that users are willing to give up some measure of control if they have sufficient reason to trust the system recommendations. They will accept suggestions while maintaining control over how these suggestions are applied.

2.2.3 Creating a User Profile

What is a user profile?

When developing personalization applications, one of the key technical issues is the problem of how to construct accurate and comprehensive profiles of individual customers that provide the most important information describing about who these customers are and how they behave [AT99].
The two basic types of data used when constructing user profiles are [AT99] *demographic data* describing who the user *is*, and *transactional data* describing what the user *does*. A *profile* is a collection of data, and Adomavicius et al. classify this information into two components :

- factual profile : contains specific facts about the user, including demographic data and facts derived from transactional data
- behavioral profile : models the behavior of a user.

Dynamic generation of the User Profile

Cingil et al. establish in [CDA00] an architecture for providing automatically generated, machine processable, dynamic user profiles to Web servers while conforming to users' privacy preferences. This architecture is depicted in figure 2.3.

On the *client side* resides a user agent that captures navigational history of the user, and that logs this information as an XML[1] *logfile*. Through the use of queries, described with an XML-Query Language, a *user profile* in RDF[2] is generated.
On the *server side* this profile information is used to deliver the user *personalized content*. Moreover the user profiles are used to create *like-minded user groups* (User Clusters) that can be used for recommendation (see section 2.2.2). To keep the valuable profile information conform with the *user's privacy* constraints, P3P[3] is used.

Web Site Evaluation based on User Logging

User profiles, and more in particular user's log files, can be used to evaluate what is taking place on web sites, which in its turn will lead to better designed and improved web sites. Web companies compete for each potential customer, and the key to winning this competition is knowledge about the needs of potential customers, and the ability to establish personalized services that satisfy these needs [Spi00].

Site should serve its users Even before personalizing products, a site should fit the needs of its user, and should be able to serve its users. Users that have difficulties in understanding how the site should be explored, are disappointed, and thus potential customers are lost. Also their traces will blur statistics about popular pages and products, which leads to invalid

[1]Extensible Markup Language : XML data is self-describing through content-oriented tags, and enables to add meaning to the data

[2]Resource Description Framework : used to process metadata for providing interoperability between applications that exchange machine understandable information.

[3]Platform for Privacy Preferences : a World Wide Web Consortium initiative to determine an overall architecture for enabling privacy on the Web.

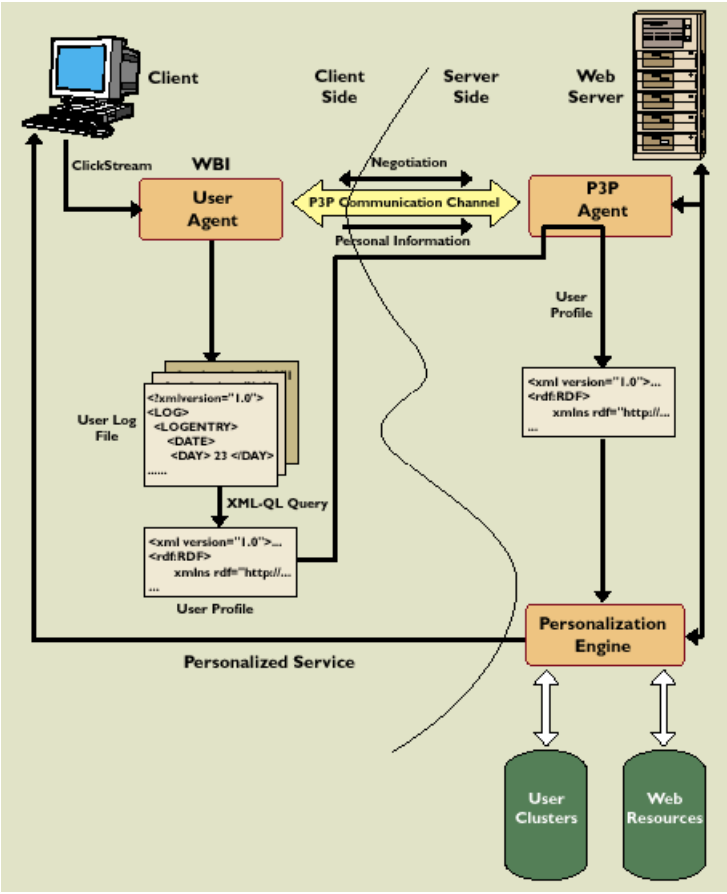

Fig. 2.3: Architecture for Generating Dynamic User Profiles

conclusions (concerning personalization) [Spi00]. A web site is a network of related pages, and users will not engage in exploring the site unless they find its structure intuitive.

Evaluating a web site In order to improve and personalize a web site, its current usage should be evaluated. To do so the operations performed by the users are recorded, and inspected or analyzed.
Each site is electronically administered by a Web Server. This server logs all activities that take place in a web server log. Since all traces left by the users are thus stored in this log, we can extract information that indirectly reflects the site's quality by applying data mining techniques [Spi00].
Data mining is a methodology to extract knowledge from data. A first step consists of preparing the web log for analysis. Some sources of difficulties to take into account are [Spi00] :

- The user identity is absent, and user distinction is based on heuristics.
- Caching implies that revisits of the same page are not recorded by the web server, although revisits are essential in the analysis of navigation behavior.
- Recorded activities might not be appropriate for studying how users perceive the site.

When the log has been prepared for analysis, a miner should test whether the site is being used in accordance with the design objectives. This mining software should provide a sophisticated interaction between the site's designer and the mining software. More on software suited for this kind of data mining is described by Spiliopoulou [Spi00].

Data mining software will return a set of navigation patterns, which are then to be analyzed by an analyst. Finally, the site is to be restructured according to the mining results.

2.2.4 Personalization in other fields

Although personalization is used mostly in e-commerce business systems, also in other fields it can be useful, and also there personalization is gaining importance.

Television Listings

With digital television making its entrance, it is interesting to start using personalized television listings. Digital television will make it increasingly difficult to discover what programs are on in a given week, as well as locating a small set of relevant programs for a quiet evening's viewing [SC00]. Although electronic program guides (EPG's) are provided to help users navigate through the digital maze of program schedules, these EPG's do not offer little more than a static category based view and the burden of searching remains with the user. These guides will have to evolve towards personalized guides taking user preferences into account [SC00].
PTV (www.ptv.ie) is an Internet System that provides a personalized information service for television viewers, and doing so it offers an innovative solution to the personalized guides problem [SC00]. The main function of PTV is constructing personalized TV guides for each individual user and each guide will contain programs the user is known to enjoy as well as program recommendations relevant to this user, based on his current profile. PTV's content database is made up of

- a schedule database storing the current channel schedules (e.g. channel and time information) , and
- a program database containing information about individual programs and films (e.g. name, genre, country of origin, director, writer, ...).

PTV generates electronic TV guides personalized for individual user by means of the ClixSmart personalization engine [SC00]. The ClixSmart[4] content personalization engine monitors online activity of users and automatically constructs profiles for these users capturing their domain and behavioral preferences. ClixSmart employs two different content filtering strategies, namely

- content based filtering to recommend similar items to the items the user liked in the past, and
- collaborative recommendation to recommend similar items that similar users also liked.

By integrating both these filtering strategies, the ClixSmart personalization engine provides a unique and powerful personalization solution [SC00].

PTV thus allows user to create personalized televisions listings and based on the user profile also new recommendations will be made to the user. Furthermore PTV learns about the user's specific and general viewing preferences, by example by means of user ratings. At this moment there is no way of capturing information about what the user actually watches, and this information is does not considered by PTV. Nevertheless in the near future users will be able to access systems like PTV through their television set, and this will make the PTV system only more usefull [SC00].

Personalizing Finance Business

Also in Banking Systems personalization becomes important. In earlier days all customers went to the bank to conduct business and to work with the bank to fulfill financial needs. The bank manager knew about his clients and their individual needs, and used this knowledge when trying to match the services of the bank with the clients. However, nowadays customers are no longer tied to a physical location, and they can deal only with any organization, from anywhere [WW00]. Therefor banking institutions moved away from the old static institution they once were.
The "new" banking institutions have to use a new approach in order to maintain a personalized service to the customers (one that will replace the previous human contacts with the bank managers). They still have to understand each customer along with his needs and goals, and knowing the customer can help the business to target its products and services to the best effect of both the business and the customer [WW00]. Personalized systems are "the" solution for these businesses.

2.3 Summary

In this chapter we gave an overview of the current work done in the field of personalization. We explained what personalization is, where it is used, and the different aspects that are important with regards to personalization, such as the user profile.

[4]Developed at the Department of Computer Science at University College Dublin

Chapter 3

Personalization in Object-Oriented Systems

In the previous chapter we saw an overview of the current uses of personalization, but so far object-oriented systems were never considered specifically. In this chapter we will discuss why we want to do personalize such systems, and show where in object-oriented systems personalization can be integrated. Furthermore we list different approaches to achieve this.

3.1 Why personalization in Object-Oriented systems

The *object-oriented paradigm* became very popular during the last decades, and more and more systems are written in an object-oriented language. Until now, research in personalized systems focussed on web applications, but also for this kind of applications object-orientation gains importance, hence this leads to personalized object-oriented systems.
Recently also personalization is growing in importance. However personalization does not have to be restricted to web applications, but in general each system involving different users could preferably be personalized. Therefore, introducing personalization in object-oriented systems is just a matter of evolution, and the use of new paradigms in new applications.

3.2 Where personalization in Object-Oriented systems

During our research we came up with four main categories in object-oriented systems where personalization could be wanted. This section gives an overview of these categories, and for each category an example is given.

When considering personalization in object-oriented systems we encounter three important categories of personalization : contents, behavior and structure. We determined these categories based on the research conducted by the authors of OODHM (Object-Oriented Hypermedia Design Method) [SRG, RSG01].
In [SRG] the authors explain how their OOHDM approach can be used to build personalized Web Applications. Their approach is object-oriented, and in the paper different examples of a class system for an e-commerce application is given. Although we claim we want personalization in other software besides web applications, we will use the e-commerce class system

as a case study in the rest of this chapter[1].

3.2.1 Contents

Goal We want to provide each individual user with a slightly different content for a particular information item. In an e-commerce system for example, different users can obtain different prices for the products, depending on their buying-history, or on special reduction coupons, and so on. Another example is that the information of a product could be displayed in another language for different users. In this case personalization consists of translating the text and displaying this new content instead of the original text.

Strategy We achieve personalized contents by personalizing the data of the system. In an object-oriented system this means that the values of attributes of objects can be different for other types of users. To achieve this the personalization process will change the values when asked for.

3.2.2 Behavior

Goal For certain "actions" in the system, we want to provide different behavior for different users because this will provide individualized responses to particular operations. For instance in an e-commerce system we might want to provide different kinds of check out processes. Some users need a lot of help and a step-by-step process, while others might prefer a one-click-process.

Strategy In an object-oriented system behavior is encapsulated in objects and can be accessed by sending messages. Providing another kind of behavior for another user thus means that calling the same message, on the same object, will result in a different method execution. The personalization process thus will change the current behavior of a message.

3.2.3 Structure

Goal In a personalized system, different users can play different roles or perform different tasks. In an e-commerce system a customer will access information on the content and availability of a product, while a manager is interested in the amount of products that are still in stock, or the price or amount of products sold. Users of the system will basically access the same information objects, but they might view them at different abstraction levels [SRG]. Take as an example the authorization rights for different users. Some users might have access to read the information, while others might also be allowed to alter it.

Strategy The personalization process will determine what kind of access is granted to the user by looking up this information in the user profile. The results of this lookup are then translated into an adaptation of the personalized system, such that authorization rights are implied.

[1]Note that the use of the e-commerce example as a case study is of minor importance since our focus is not on the application, but on the features of object-orientation.

3.3 Possible Approaches

Now we will have a look at possible approaches to implement personalization. Personalization can be hard-coded into the system, but also decoupled, both on base-level and meta-level.

3.3.1 Hard coded into the system

When personalized systems first appeared, the kind of personalization was very limited and it was hard coded into the system. As time evolved, more personalization was added, but programmers continued hard coding these changes into the system. But hard coding personalization algorithms into the base system tangles the personalization algorithm with the functional code, making it hard to understand, and thus hard to evolve and maintain. Although the approach of hard coding personalization is not completely banned yet, it is unadvisable to use it, and better approaches should be considered.

3.3.2 Decoupling using OOHDM

What is OOHDM?

OOHDM stands for Object-Oriented Hypermedia Design Method and is a model-based approach to build large hypermedia applications (e.g. web sites, information systems, interactive kiosks, multimedia presentations, etc.) [RSL99]. In these kind of applications navigation and functional behavior must be seamlessly integrated in the final application. However, during the design process we should be able to decouple design decisions related with the application's navigational structure (i.e. navigational model) from those related with the domain model itself (i.e. conceptual model) [SR98]. These conflicting requirements are tackled by OOHDM.

The OOHDM method

Since the approach discussed in the previous section is obviously not opportune. The people working on the OOHDM method [RSG01, SRG] made a first step towards decoupling the basic system from its personalization. They do this by making a clear separation between the conceptual model and the navigation model that is desired for a particular hypermedia application.

With OOHDM the development of hypermedia applications occurs as a four activity process : Conceptual Design, Navigation Design, Abstract Interface Design, and Implementation (more on this below). During each of these steps a set of object-oriented models are built from previous iterations. Considering these four processes as separate activities allows us not only to concentrate on different concerns at a time, but mainly to obtain a framework for reasoning about the design process and encapsulating design experience specific to each activity [SR98, RSL99].
The authors of [SR98] list the following cornerstones of OOHDM :

- Navigation objects are views of conceptual objects
- Appropriate abstractions are used to organize the navigation space
- Interface issues are separated from navigation issues

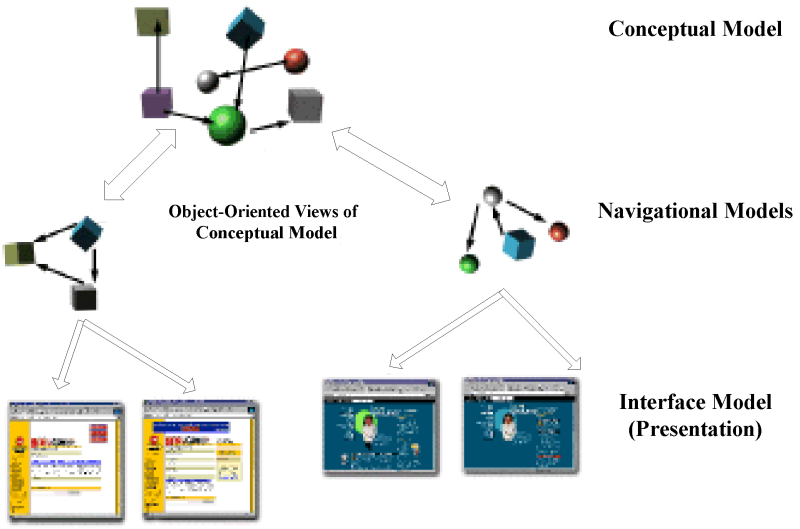

Fig. 3.1: OOHDM Design Models

- There is an explicit identification that there are design decisions that need only be made at implementation time.

Conceptual Design During this phase the model of the application domain is elaborated by means of well known object-oriented modelling principles, augmented with some primitives such as attribute perspectives and sub-systems [SR98].

The Conceptual Model that is created during this phase represents two kinds of objects :

- objects to be perceived as nodes in the navigational model, and
- objects that provide computational support for the application (e.g. algorithms, access to databases,...).

Important is that this resulting model does not include navigation specific information, and thus the model may serve for many applications [SR98].

Navigational Design Typically in hypermedia applications is the notion of navigation. This implies that the user of the application will navigate in a space made out of objects that are different from the conceptual objects, because these objects are customized to the user's profile and tasks. Navigation objects are composed of attributes of possibly several different conceptual object attributes. Furthermore navigation objects can also have their own behavior apart from browsing and navigation, such as updates and computations.
Navigational Contexts are sets of objects that make a meaningful whole in the navigational space, for example based on class attributes or relationships [SR98].

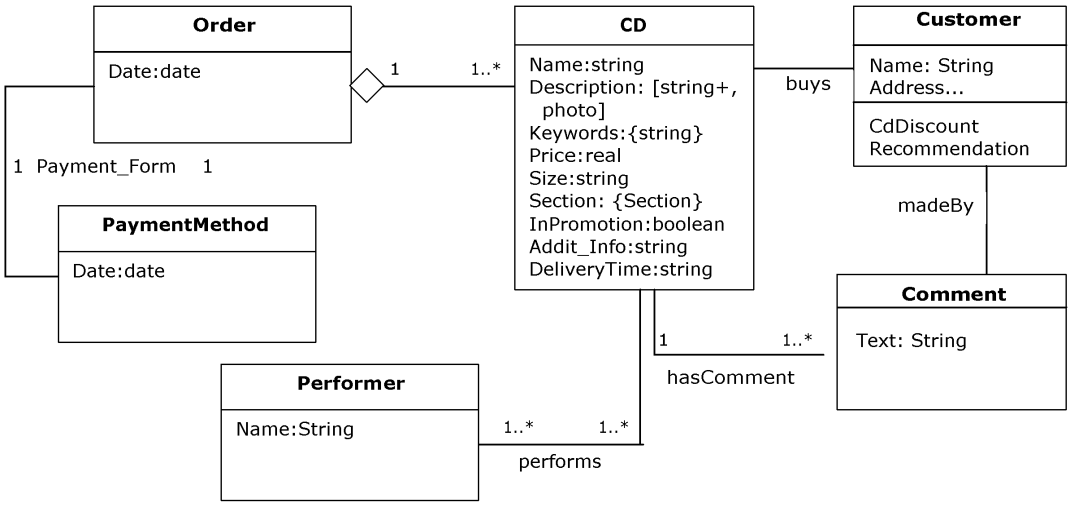

Fig. 3.2: E-commerce : Conceptual Model

Abstract Interface Design The abstract interface design is the link between user interaction and navigation objects. The interface model specifies [SR98] :

- which interface objects the user perceives,
- which interface objects will activate navigation,
- how multimedia interface objects will be synchronized, and
- the interface transformations that will take place.

Implementation phase In this phase the conceptual objects, navigation objects and interface objects are mapped onto the particular runtime environment.

Conceptual vs Navigational level

As explained above, OOHDM makes a separation between the Conceptual Model and the Navigational Model such that the conceptual model may serve as a basis for different applications and navigation is customized for the user's profile and tasks.

Taking again the example of an e-commerce system, the conceptual model, as depicted in figure 3.2, is a representation of what the system has to represent. The system depicted is a simple cd-store where each cd has a performer and possibly some user comments. A customer (user) can buy cd's and then orders for the particular cd will be created.
However, the conceptual model differs from the way the system will be navigated, as shown in figures 3.3 and 3.4. The figure shows the navigation scheme for user's and managers. Although both the customer and the manager are viewing the same conceptual objects, they have a different navigational concept, because they will navigate the database with different purposes. Customers for instance may navigate through comments, but managers only see comments as attributes of cd's. Also, customers may access information about performers, but only managers may navigate the information about users. For the complete example the

user is referred to [SRG].

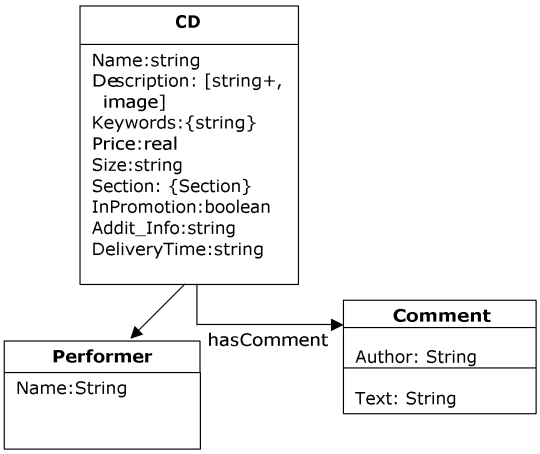

Fig. 3.3: Customer Profile View

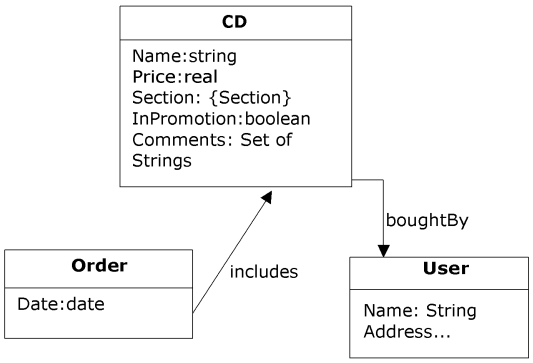

Fig. 3.4: Manager profile View

OOHDM for Personalization

Originally OOHDM was intended to help designing hypermedia applications, but during years of research it became clear that the method can also easily be used to introduce personalization.
An example of this is shown in figure 3.5. The expressions between brackets indicate the values of attributes that are obtained from the user object (e.g. `[Anchor Comments]`). In the example the price of the product depends on the discount that is granted to the user (`[subject.price - user.CDdiscount]`). The discount is different from user to user, but can easily be looked up in the user's profile. Price is an attribute of the object in the navigational model representing the product, but OOHDM allows to write down these kind of personalizations to adapt the attribute.

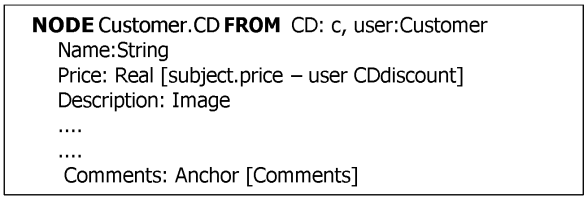

Fig. 3.5: Personalizing Price

Another example (figures 3.6 and 3.7) shows the personalization of the structure of a website. A set of services are grouped in modules and can be refined and personalized for each user. The navigational model for this situation is shown in figure 3.6. Each user will have a "personal page" with his own selected models, and each model can again be personalized with user selected sections. The corresponding modules are selected from the user profile as

shown in figure 3.7. [SRG, RSG01]. For each user the profile contains information about what his/her modules are, and when creating the personalized page, only these modules are selected (`Modules : Set[user getModules]`). The same principle counts for sections, etc.

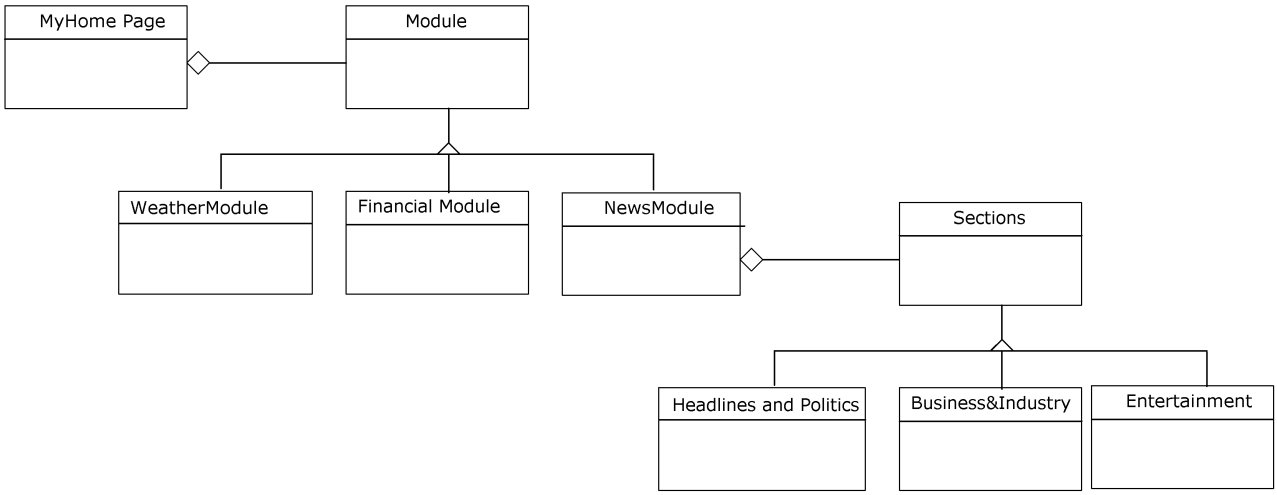

Fig. 3.6: Modules of Services

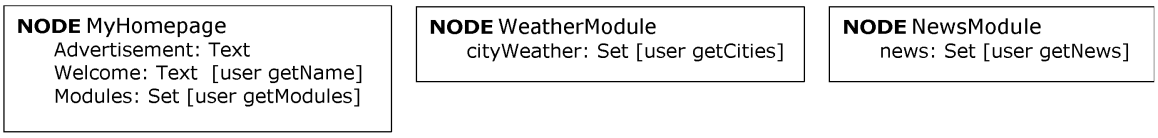

Fig. 3.7: Personalizing Modules

Decoupled or Hard coded?

OOHDM allows a clear decoupling between the conceptual and navigational model. Personalizing navigational nodes is expressed on navigational level, and changing the kind of personalization will not influence the conceptual model. This decoupling is a big advantage on the hard coded approach that was mentioned in section 3.3.1.
Although the OOHDM approach might introduce some decoupling, personalization is still hard coded into the navigational level since it is explicitly added to the node. If the kind of personalization changes, this does not affect the conceptual level, but it still implies hard coded changes at the navigational level. Therefore this approach does not achieve a complete decoupling between the basic system and its personalization.

3.3.3 Decoupling at the base-level

We can go a step further in the process of decoupling the basic system from its personalization and the people working on OOHDM have used several design patterns, but we also consider some other possible approaches.

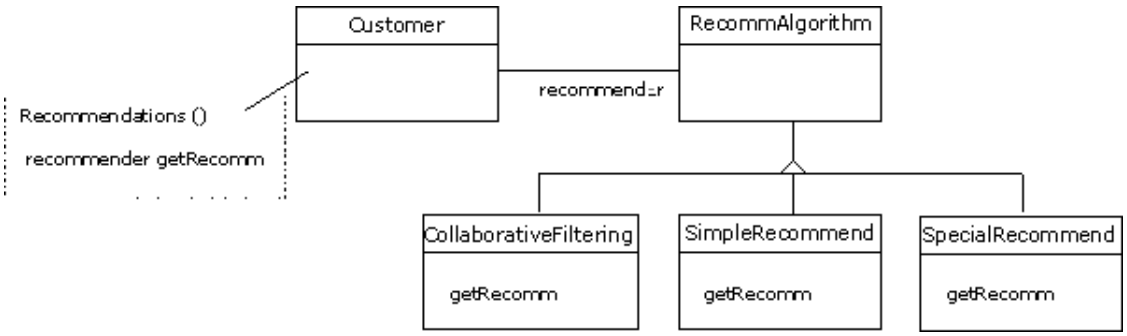

Fig. 3.8: Strategy Pattern : Decoupling recommendation algorithm and user

Using Design Patterns

Design patterns capture solutions that were developed and evolved over time. As developers struggle for greater reuse and flexibility in their software, the patterns evolve and reflect untold design and recoding [GHJV94]. Ever since patterns were explicitly written down and grouped, they became more widely used and accepted as important guidelines when developing object-oriented systems.
Almost all known patterns can be used when creating personalized systems[RSG01]. Using these patterns obviously results in a decoupling between the system and the personalization aspects.

Observer Pattern for decoupling design concerns Using OOHDM (see section 3.3.2) a decoupling is made between conceptual level, navigational level and interface. Rossi et al. [RSG01] show that this is a direct application of the observer pattern [GHJV94]. This pattern thus decouples the three design concerns :

- base information and behavior (conceptual),
- what the user perceives (navigation), and
- how the user perceives it (interface).

Strategy Pattern to decouple algorithms and user object Allowing different users to use different algorithms can easily be achieved by using the strategy pattern [GHJV94], as shown in figure 3.8 [RSG01]. By doing so, recommendation algorithms are decoupled from the user object, and thus by assigning a different algorithm to a different user, the algorithm is personalized.

Also the Adaptive Strategy Design Pattern from Aubert [Aub01] allows different algorithms to be used. We use this pattern if we cannot decide until runtime which version of the algorithm is best suited for the task. Furthermore the client (i.e. user) does not have to worry about how this algorithm is chosen [Aub01].

Adapter Pattern to wrap third party products The adapter pattern [GHJV94] is used in personalized systems to wrap third party products such that its interface is “adapted” to the existing protocol. In this way the customer class stays unchanged, regardless of the

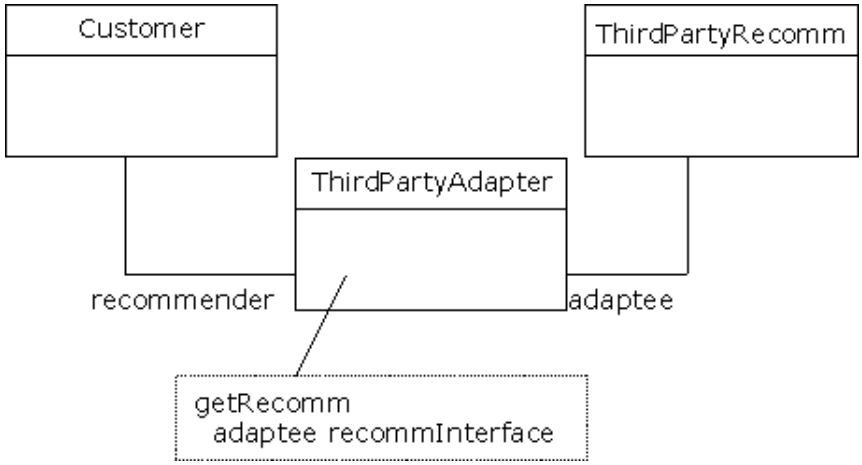

Fig. 3.9: Adapter Pattern : Wrapping Third Party Products

specific features or implementation style of the third party product, as is shown in figure 3.9 [RSG01].

Decorator Pattern to specify small variations Different user profiles can share the same specification and the same information in the final application, with only small variations [RSG01]. In the OOHDM approach these variations are expressed by extending the node classes (navigational level). These extension classes act as decorators [GHJV94] of the extended class. This same approach can be used to add specialized behaviors (e.g. access rights) for each user profile [RSG01].

Other approaches

Other approaches to separate personalization from the base system can be explored, but so far few research has been conducted doing so.

Components Another possible approach is the use of components. If for each type of personalization a component could be provided, then creating a personalized system would boil down to combining the base system's components with the personalization components. If personalization would change because another user is using the system, the affected component would have to be replaced by another one.

ValueModels Using ValueModels could be another approach. They are used in the Model-View-Presenter Framework of VisualWorks for Smalltalk [Val01]. A ValueModel is a wrapper for an object (its value) with a simple generic interface onto its value for its clients . All ValueModels provide the same interface :

- `#value:` to set the value
- `#value` to get the value

and therefore this its client does not need to know where the value comes from and how it is actually accessed. Furthermore, since ValueModels are models, they are capable of

supporting the use of observers [Val01].
When using ValueModels for all possible objects that can be personalized in our system, the personalization would be hidden from the basic system and specified in the ValueModels. Up until now no research has been conducted to investigate the feasibility of this approach, but at first sight using ValueModels seems to introduce an overkill.

3.3.4 Decoupling at the meta-level

Decoupling at the base-level is one option, now we go one step further and consider decoupling personalization by putting it at meta-level. Doing so the meta-level will be used as the layer where the personalization information is stored. Applying this information requires a certain link with the base-level. We consider two possible meta-level approaches : Aspect Oriented Programming and Logic Meta Programming.

Aspect Oriented Programming

What is AOP? Aspect Oriented Programming (AOP) is a programming technique that arose from the objective to achieve less tangled code by allowing a separation between the basic functionality of a system and additional features such as concurrency, real-time properties, persistency, etc. This separation is achieved by providing an aspect language in which aspects (addressing the additional features) can be defined. These aspects are intertwined, by using an aspect weaver, with the basic algorithm in order to come to the desired application.

Typically for an aspect is its crosscutting nature. Crosscutting features are features that affect a group of classes (sometimes even all classes) in the system. For example, consider a system of objects of different types, in which we want every object instantiation to be logged onto a file. This logging feature will affect every class of the software system, because every constructor needs to do its own logging (either by writing to the file, or by calling an other class that will provide the logging). This kind of feature is thus a crosscutting feature.
Aspects tend to reach out and affect other components (groups of objects), whereas objects are not supposed to do this because they are to mind their own business [Asp00].

AOP and Personalization Until now a lot of research concerning AOP was directed towards the use of AOP to separate concerns, namely to separate basic functionality from non-functional crosscutting features such as persistency and concurrency. However the reader should note that also personalization can be considered as such an aspect. After all, the personalization features are affecting multiple parts of the base system. Therefore AOP can be considered as a means to describe personalization, that has to be woven with the base system in order to obtain a personalized version of this base system.

More on AOP can be found in [KLM+97, Asp00, DM00, KHH+01].

Logic Meta Programming

The approach of using Logic Meta Programming (LMP) is fully explained in the next chapter. The main idea is to separate personalization issues from the base system by using a logic meta programming language. The LMP language serves as a declarative meta layer and allows us

to express the personalization that is to be applied to the base system in a declarative way. This layer will contain

- facts about the user, and
- rules describing the personalization.

The *facts* are different for each user, but these sets of facts (user profiles) might have to consider some constraints to make sure that personalization rules can depend on certain data being available. For example, each user needs a user identity, and thus each profile needs a fact representing this identity.
The *rules* will interact with the base-level and have an impact on it (in this case the object-oriented system), and by doing the base system will be personalized.

By using the declarative layer to introduce personalization, we keep personalization separated from the basic system. Furthermore describing user profiles as a set of facts appears to be rather natural and can easily be achieved in a declarative language.

Conclusion

Using a meta-level to express personalization is very promising, no matter what approach was chosen. The basic functionality of the base system and the personalization issues to get the system personalized are clearly separated from each other. This makes changing the way of personalization a lot easier without having to change the base system as well. Furthermore one can easily see which parts of the system are personalized without having to browse the whole system.

3.4 Summary

In this chapter we explained why we want to personalize in object-oriented systems, where we want this personalization and what approaches are already explored. We discussed the hard-coded implementation of personalization, as well as decoupling personalization from the base system using OOHDM and decoupling at base-level. Finally we talked about decoupling at meta-level, and more in particular Logic Meta Programming. Since this last approach is the most promising, the rest of this dissertation explores this approach.

Chapter 4

Logic Meta Programming

In chapter 3 we discussed different approaches for achieving personalization in object-oriented systems. The meta-approach, where personalization is described on meta-level, indicated to lead to a clearer separation between the base system and its personalization. Given this advantage, we have chosen a particular kind of meta-programming to describe personalization, namely Logic Meta Programming (LMP).
In this chapter we will handle logic meta programming in detail, explain why it is suited for achieving personalization in object-oriented systems and we will end by introducing a logic meta programming language called SOUL.

4.1 Logic Meta Programming

In Logic Meta Programming a logic language is used at a meta-level to reason about language at base-level. This approach results in a declarative meta-layer on top of an (e.g. object-oriented) base-layer. The language at meta-level is called a logic meta programming language. In this section logic programming and logic meta programming are explained.

4.1.1 Logic Programming

Logic programming is a programming paradigm developed in the seventies. Rather than viewing a computer program as a step-by-step description of an algorithm (like traditional languages), the program is thought of as a logical theory and a procedure call is viewed as a theorem of which the truth needs to be established [Fla94]. Thus, the execution of a program boils down to searching for a proof.
A logic program concentrates on a *declarative specification* of *what* the problem is, and not on a procedural specification of *how* the problem needs to be solved. In order to perform this, the database of a logic program consists of facts and rules which are accessed by queries;

- *Facts* hold static information that is always true in the application domain.
- *Rules* derive new facts from existing ones. The conditional part of the rule should be true in order to conclude the premise of the rule.
- *Queries* are used to access the data in the database. Finding an answer to such a query is carried out by matching it with facts or rules, that are either initial or derived.

In essence, *matching* is proving that a statement follows logically from some other statements. In Prolog this is done by unifying variables in a query with facts from the database [Fla94]. This reasoning process is also called *resolution*, and adds a procedural interpretation to logical formulas, besides the declarative interpretation they already have. Because of this procedural interpretation, logic programming can be used as a programming language. Kowalski's equation *"algorithm = logic + control"* also denotes this [Fla94]. In this equation, logic refers to the declarative meaning of logical formulas, and control refers to the procedural meaning. However, in a purely declarative programming language it is not possible to express procedural meaning.

Prolog is one of the most widely used logic programming languages, though not a purely declarative programming language for the procedural meaning of programs cannot be ignored. Prolog's inference engine uses backtracking to reconsider other possible solutions for its queries.

For an overview on Prolog the reader is referred to [Fla94].

4.1.2 LMP Terminology

Before explaining what Logic Meta Programming is, we will just introduce some terminology that we will use throughout the remainder of this dissertation.

Meta Programming

Maes [Mae87] defines some terminology concerning meta-systems :
A *program* specifies the computational process which will manipulate a representation of entities and data. A *computational system* reasons about and acts upon some part of the world, called the *domain* of the system.
In a *meta-system* the computational system reasons about and acts upon another computational system, called the *base-system*. Therefore, the meta-system has as domain the base-system, and the program of the meta-system is called the *meta-program*. Thus, a meta-program is a program which reasons about another program (i.e. the base-program).

When using a logic programming language as a meta-system we encounter *Logic Meta Programming*, a kind of (multi-paradigm) programming where base-languages are described by means of logic programs. Two paradigms, object-oriented programming and declarative programming, are combined.

Reflection

Maes [Mae87] gives the following definition of reflection :
"A computational system is said to be ***reflective*** *if it incorporates and also manipulates causally connected data representing (aspects of) itself."*
This means that a system is reflective if it can (through the meta-language) reason upon aspects of itself. The base-level is the level that is reasoned about. Moreover, these two levels are causally connected with one another.

One particular way to define (and to construct) a reflective system is by making use of a so-called linguistic symbiosis as introduced by Steyaert [Ste94]. *Linguistic symbiosis* means that computations, specified in different formalisms, are mixed together in a transparent way. This allows us to specify a relationship between a high-level language and its underlying implementation language, in a way that the programmer can profit from both worlds. It can therefore also specify the relation between the meta-level language and the base-level language of Logic Meta Programming.

Linguistic symbiosis between two systems enables introspection and absorption. *Introspection* means that a system can interrogate its implementation and thus makes it possible to retrieve information and to look at the underlying language. From within the meta-level we can access the base-level, e.g. from within the declarative meta layer it is possible to retrieve information of the structure of the object-oriented base language, such as classes, subclasses, methods, instance variables, ...
Absorption is used to indicate how the meta language can act upon the base language. The meta language can really change the underlying language, but by doing so, it can also change itself. For example, it is possible to adapt the object-oriented base language from within the declarative meta language by sending messages of which is is known that they change the base-level. Thus, it also possible to send messages that change the classes implementing the meta-level. When the declarative meta-level sends messages that will affect the classes that implement this meta-level, the declarative meta-level is changing itself.
This means that meta-level and base-level are *causally connected* (changing one level affects the other) and in this way, the system (the symbiosis) can incorporate and manipulate its own representation, which is causally connected with itself.

Although a logic meta programming language does not necessarily induce reflection, we mentioned reflection here because it is an interesting aspect that is related to this matter anyhow.

4.2 Smalltalk Open Unification Language

The Smalltalk Open Unification Language (SOUL) was developed by Roel Wuyts at PROG[1]. SOUL is a Prolog-like logic meta language built on top of Smalltalk. It creates a symbiosis between the declarative and object-oriented paradigm and is used to reason about the structure of object-oriented systems. In this section we will shortly introduce SOUL, because it will be our medium to express personalization.

4.2.1 Setup

As said before SOUL is an LMP language and provides a Prolog-like meta language on top of Smalltalk as base language. The core of SOUL is a logic programming language with a resolution engine and is completely written in Smalltalk. Additionally SOUL is in symbiosis with Smalltalk such that it is possible to reason directly about Smalltalk base programs [Wuy01].

SOUL is based on the software architecture of a rule-based system, but has been refined. Like in Prolog the three basic clauses are *facts*, *rules* and *queries*. *Repositories* are the knowledge

[1]Programming Technology Lab of the VUB (Free University of Brussels)

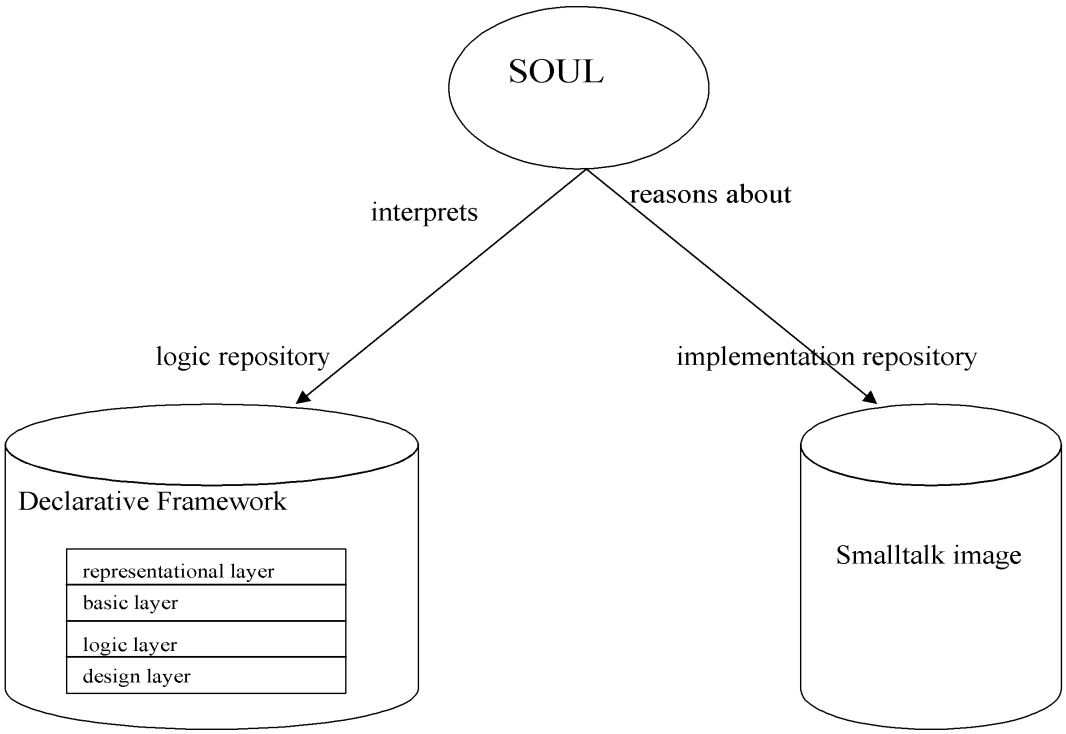

Fig. 4.1: Soul's Setup

bases to be consulted during the interpretation and they consist of a number of facts and rules. It is possible to nest repositories, which is very useful when there is a need to structure the represented knowledge (e.g. in a hierarchy).

As depicted in figure 4.1 SOUL is used to reason about Smalltalk systems. To facilitate this reasoning, a built-in declarative framework is provided and consists of a layered set of rules. The four layers that make up this framework are [Wuy01] :

- representational layer : reifies the base-language concepts, such as classes, methods, instance variables, inheritance,...
- basic layer : adds auxiliary predicates to facilitate reasoning. This layer is needed to reason on a certain level of abstraction with the logic meta-programming language.
- logic layer : contains the predicates that add core logic-programming functionality, such as list handling, arithmetic, program control, repository handling,... This layer is used by all other layers.
- design layer : groups all predicates that express particular design notations, such as programming conventions, design patterns and UML class diagrams.

This framework is described in detail in [Wuy01].

4.2.2 Syntax

For a detailed description of SOUL's syntax the user is referred to [Wuy01], but we will briefly introduce the major structures.

Fact, Rule, Query In SOUL the format of a *rule* is `Rule head if body`. Both head and body are predicates followed by some arguments in between brackets. *Arguments* are terms and can either be a constant, a variable, a compound term or a list. *Variables* are denoted with a question mark. *Predicates* of a rule can be combined with a boolean *or* (semicolon) or boolean *and* (colon). A *fact* is represented by `Fact predicate(x)`. A query is launched with `Query predicate(?X)`. The following is an example of a fact, rule and query :

```
Fact father(prosper, jef).
Fact mother(eufrasie, jef).
Rule parent(Parent,Child) if
        father(Parent,Child);
        mother(Parent,Child).
Query parent(prosper, ?Y).
```

Smalltalk Term A term between brackets is a Smalltalk term and this kind of term contains Smalltalk expressions (code) that can refer to logic variables.

Generate Predicate The *generate predicate* allows to get separate logic results for each selector. The first argument of this predicate specifies the logic variable to bind the results to, the second argument is a stream of solutions. Each of these solutions is bound, one by one, to the first argument.

Quoted String SOUL also provides the *quoted string* language construct (put between curly braces) to specify any kind of source code as it is, without being evaluated. For example in

```
Rule classHtml (?Class, {<html><body><h1> Methods of
                            ?className</h1>
                            <ul>?selectorNameStrings
                            </ul></body></html>}) if
    className(?class, ?className),
    findall( {<li>?sel<?li>},
             classImplements(?class, ?sel),
             ?ms),
    list2String(?ms, ?selectorNameStrings).
```

everything between curly braces is quoted, and thus will not be evaluated when interpreting the rule. Note that these quoted strings can contain logic variables.

4.2.3 SOUL in Symbiosis with Smalltalk

Creating a Logic Meta-Programming Language

SOUL is a logic meta-language built for reasoning about an object-oriented base language. Thus, somehow object-oriented systems should be represented as logic facts, such that logic programs can be written down using this representation. Therefore we use a logic representation of the parse tree of the system to be reasoned about.
Turning a logic programming language into a logic meta-programming language is performed by creating a knowledge base containing the logic representation of the (base) system to be reasoned about [Wuy01]. The repository containing the logic representation of the base system together with the base predicates forms a logic meta-programming language. But as we will explain in the next paragraph SOUL goes a step beyond this, which makes it more powerful than a plain logic meta-programming language.

A practical usable LMP language

Now that a logic programming language can be turned into a logic meta programming language, it is possible to write regular logic programs that manipulate the representation of the base program (i.e. meta programs). The major drawbacks of this approach [Wuy01] are:

- only the information in the database can be used by the logic meta-programming language,
- the repository can become very large, and
- the actual source code is not linked with the logic representation of the system in the repository.

Since SOUL is to be used practically, it had to be specialized into a true logic meta-programming language and such a language should directly reason upon programs expressed in the base language. To do so an extra mechanism was added.
This extra mechanism consists of a symbiosis between SOUL and Smalltalk. This symbiosis makes each Smalltalk object directly usable as a logical term in SOUL. It even allows to write Smalltalk expressions that can be parameterized by logic variables [Wuy01].

SOUL's *Smalltalk terms* (see section 4.2.2) acts as a reflection operator to absorb Smalltalk objects as logic terms. It is a logic construct that can contain Smalltalk code to execute during logic interpretation and thus it makes it possible to refer to real Smalltalk entities from within SOUL.
The following `class` predicate is an example of such a Smalltalk term.

```
Rule class(?Class) if
    constant(?Class),
    [Smalltalk includes: ?Class name].

Rule class(?Class) if
    variable(?Class),
    generate(?Class, [Smalltalk allClasses]).
```

The class predicate is a predicate that captures the concept of a class in Smalltalk. This predicate is built by defining two rules. The first rule indicates what happens when the class predicate is called with a constant value. The Smalltalk term `[Smalltalk includes: ?Class name]` checks whether the value represents an existing class in the Smalltalk image. The second rule is applied when `?Class` is variable. Then the generate predicate is used to unify that variable with each class in the Smalltalk image. This is done by executing the Smalltalk term `[Smalltalk allClasses]`.
When querying `Query class([Array])` the predicate will verify wether `Array` is an existing class in the Smalltalk image. The query `Query class(?Class)` will unify `?Class` with a class in the Smalltalk image.
The Smalltalk term will allow us to send messages to Smalltalk objects. In the class predicate example, the message `name` is sent to the value unified with `?Class`, and the message `includes:` is sent to the Smalltalk global dictionary to find out if the certain class exists or not (i.e. is included in the global dictionary).

Reflection Note that SOUL is reflective (see terminology in chapter 4.1.2), because SOUL is implemented in Smalltalk, and can thus reason about itself, and even alter its own implementation (e.g. by using assert predicates to add new logic clauses to the current repository) [Wuy01].

4.2.4 SOUL for describing dynamic processes

SOUL was originally developed to support co-evolution of design and implementation such that when one of the two changes there is an effect on the other [Wuy01]. In his dissertation Roel Wuyts [Wuy01] shows how SOUL is used as a style checker (checking violations against programming conventions) and an UML tool (keeping UML diagrams synchronized with the implementation). These applications both involve reasoning about the object-oriented system in a static way.
SOUL was not intended to be used for reasoning on an object-oriented system in a more dynamic way, such that during the reasoning process the actual base system can change. However, SOUL is powerful enough to allow these kind of features to be added easily.

4.3 QSOUL

Currently the people of PROG[1]are working on a successor of SOUL, named QSOUL, which stands for Quasiquoted Smalltalk Open Unification Language. It is a new implementation of SOUL extended with a quasiquoting mechanism to allow declarative code generation.

Although QSOUL is a renewed (and extended) version of SOUL, we chose to use SOUL for this thesis because at the time of writing this dissertation, there was more information available on SOUL. Likely a more finished version of QSOUL and some documentation on that system will be available soon.

4.4 Using LMP for Personalization

In the next part of this dissertation we will use Logic Meta Programming as an approach to achieve personalization for two main reasons. There are two good reasons to do so.

First of all, as explained in chapter 3.3, a clean decoupling of the base system and its personalization will lead to better evolvable and maintainable systems. When trying to achieve this goal, we encounter an evolution towards such decoupling using a meta-level language. This solution also recognizes that the problem of personalization should be treated as a separated aspect of the application, and dealt with in an orthogonal way.

Secondly, describing personalization in a declarative way seems straightforward. Everything known about the user is a set of facts, (a set of logged events) and facts can easily be stored in a declarative way. Furthermore personalization rules are easy to express and modular. They can be maintained efficiently and it is easy to add new rules or edit existing ones. A declarative approach thus seems to be advisable.

4.5 Summary

This chapter explained logic meta programming in detail and we indicated why it is a suited approach for achieving personalization. Furthermore we gave an example of a logic meta programming language (SOUL).

Chapter 5

An Architecture for Personalizing Systems

We already explained why we want to introduce personalization in object-oriented systems (see chapter 3). Now we will show how we will realize this by using the logic meta programming approach, and more specific by using SOUL (see chapter 4.2).
We will start by introducing our general setup for achieving our goals. Then we will have a closer look at the components of the setup : the user profile, the interface to access this profile and the rules we construct to introduce personalization. The last section gives a step by step example of our approach.

5.1 Architecture

5.1.1 General Setup

As seen before, SOUL is a logic meta programming language built on top of Smalltalk. At the base-level resides the object-oriented system that is to be personalized (here this will be Smalltalk). On the level of SOUL (meta-level) we will capture the personalization description. Figure 5.1 depicts our setup to achieve this goal. On the meta-level we make a distinction made between the user profile (facts known about the user), the rules that make up an interface to access these facts (see section 5.2), and rules for personalization (see section 5.3).
Furthermore there is a switch between the SOUL level and the Smalltalk level. SOUL will reason about and adapt the Smalltalk level (see section 5.4.1). On the other hand the Smalltalk level will trigger the SOUL level in order to start the reasoning process (see section 5.4.2).

5.1.2 Organizing the meta-level into Layers

SOUL uses repositories to store all gathered knowledge. These repositories can be nested, which allows us to organize different groups of predicates into different repositories [Wuy01].

Personalization rules (predicates) can be organized into repositories, such that related predicates are grouped (e.g. check-out algorithms, product personalization, recommendations, ...). At this point repositories are not allowed to be organized into ordered hierarchies such that it is possible to imply an order on the rules, namely such that all rules from one repository are

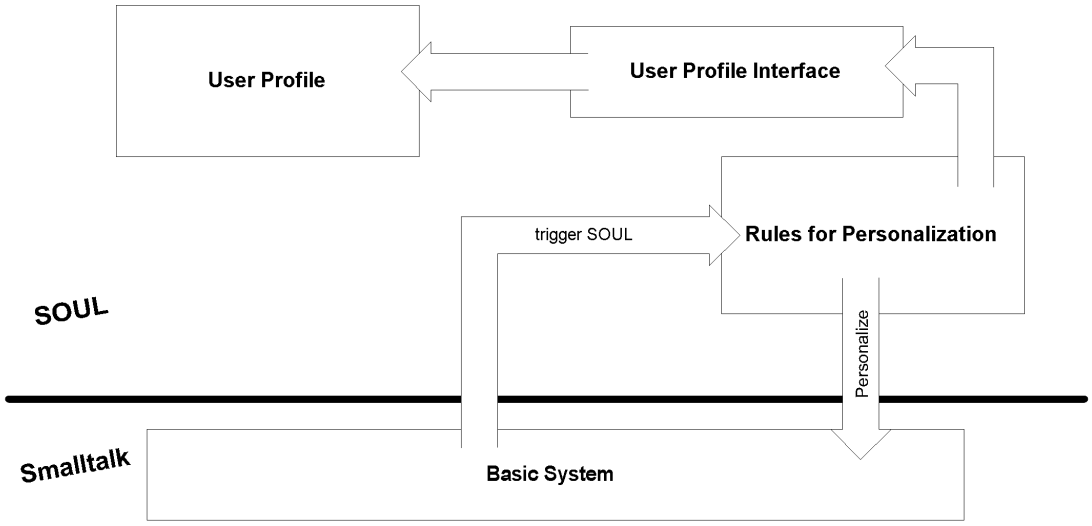

Fig. 5.1: Our Setup

triggered before the rules from another repository. Nevertheless this can be specified logically, but is quite cumbersome. For instance, the rule

```
Rule personalize(?x) if
    xor ( specializedRules.personalize(?x),
          generalRules.personalize(?x)).
```

indicates that the rule `personalize` from the repository `specializedRules` should be tried first. If this one fails, the rule `personalize` from the repository `generalRules` will be tried. Imposing these kind of order on personalization rules requires logic rules similar to this one, but this obviously is not a very attractive way of doing so when a lot of repositories are to be considered, or if different orderings should be applicable.

5.2 The User Profile

The user profile represents the information that is known about the user. In personalized systems this information is crucial, because this is where the personalization will be based on. The user information can be based on history, on the actions a user takes or on information gathered through a survey,...

5.2.1 Facts about the User

The user profile consists of a set of facts about the user. This set of facts can be changed at run-time by using the `assert` (adding a fact) and `retract` (removing a fact) predicates. For each user a different repository is used such that user profiles are clearly separated from each other.

As an example, consider a user named “Jef” with id408. Then the user profile of “Jef” has to contain these facts :

```
Fact name(id408, jef).
Fact age(id408, 28).
Fact boughtProducts(id408, <BoekID20, BoekID403, CDID230>).
```

5.2.2 Querying the User Profile

Rules to query the user profile are grouped together and considered to be a separate part of our setup (see figure 5.1).
On the one hand we have simple queries such as

```
name(?User, ?Value).
```

On the other hand we can construct more complex rules that can be queried, such as

```
Rule boughtProduct(?User, ?Product) if
    boughtProducts(?User, ?List),
    member(?Product, ?List).
```

This rule can be used to check wether the user bought a certain product. The `member` predicate is a standard SOUL predicate (logic layer) and will return true if an element `?Product` is part of a list `?List`. (And if ?List is bound to a list, ?Product will be subsequently bound to an element of the list)

5.2.3 User Information as a part of the base system

Since information about a user is a set of facts, it might seem obvious to store this information in a database. However, one might prefer to keep the user profile as a part of the base system, for instance because the user is conceptually part of the base system, or because the system has directly access to the user information. Nevertheless, this will not change anything to our approach. The rules that now are used to access the user profile on meta-level, would have to be redefined such that they access the user profile residing on the base-level. Since there is absolutely no problem to access the base-level using SOUL, the user information will continue to be easily accessible. An example of this is the following Rule :

```
Rule name(?User, ?Name) if
    equals(?Name, [?User getName]).
```

In this rule the variable `?Name` will be bound to the result of sending the Smalltalk message `getName` to the Smalltalk entity (in this case an instance of the User Class) bound to the variable `?User`.

5.3 Rules for Personalization

Now we will give some examples of personalization rules to achieve these different kinds of personalization, as they were presented in chapter 3. First we will show the plain personalization as it is described by facts and rules. Secondly, we discuss how the base system actually is adapted.

5.3.1 Contents

When personalizing contents we indicate that data can differ from user to user, and typically values of attributes will have to change.

When personalizing contents we assume that this value of attributes has to be retrieved using accessor methods (best practice patterns [Bec97]). Introducing adapted values of these attributes then means that the accessor method has to return a different (calculated) value than originally was set. This is achieved by the use of the method wrapper mechanism, as will be explained in section 5.4.3.

As an example of contents personalization we consider an e-commerce example in which certain customers have different kinds of reductions on products, and thus will pay a different price for these products. We assume that the price of a product is retrieved with an accessor method `getPrice` (on the Smalltalk class `Product`).

Accessing the User Profile

```
Rule changePrice(?Customer, ?Product, ?ResultString) if
        getReduction(?Customer, ?Value).
```

`getReduction` is a rule to access the user profile, and in this case it is used to retrieve the reduction for the user `?Customer`. `?ResultString` will be a quoted string containing some Smalltalk Code (e.g. {‘‘`Smalltalk code to update value`’’}), which can possibly use `?Value`.

Personalization rules

```
Rule updatePrice(?ResultString) if
    isReceiver([Product], [#getPrice], [?aProductObject]),
    currentCustomer(?ID),
    changePrice(?ID, [?aProductObject], ?ResultString).
```

This rule specifies that when the method `getPrice` is received by the class `Product`, a possible change can occur and `changePrice` is triggered. The Smalltalk terms (denoted by square brackets) refer to the Smalltalk class `Product`, the method `#getPrice` and the current product object that is receiving the message `#getPrice`.
`currentCustomer()` asks the system for an id of the current customer, otherwise personalizing the system user dependably would be useless.

Performing the personalization It seems interesting to trigger the rule `changePrice` only when the *user profile is updated* with a new reduction value. Each time the message `getPrice` is sent, the adapted method will be executed. However, the (seemingly) overkill introduced with triggering the rules each time *the price of a product is actually asked for*, is necessary if we consider a system where multiple users are present at the same time. After all, the update of the attribute's value depends on the user.

5.3.2 Behavior

Consider a user checking out of some web-application. Depending on the kind of user, a different kind of algorithm to perform the check out process can be used, depending on what user is checking out. Also providing the user with recommendations, is done through the use of different recommendation algorithms in a similar way.

The algorithms to be used can be both put at base-level or meta-level. In the first case, the meta-level will refer to the class containing the algorithm. In the second case, the algorithm code is put in a Quoted code term on meta-level. In either case the method call will be intercepted, the right algorithm selected and performed.
In our example we will assume the second case, because it allows to have a different algorithm for all kinds of users, without having to change the base system if a new algorithm is added. Obviously all possible algorithms should be grouped in a layer on meta-level, to avoid repeating the same code.

User profile contains the algorithm specification The user profile will contain the specific algorithm to be used. For instance,

```
Fact algorithm(id201, checkout, 1Click).
```

indicates that the user with `id201` has as an algorithm for the checkout process `1Click`. `1Click` will be a fact in the algorithm layer that contains the actual Smalltalk code for the algorithm :

```
Fact 1Click({‘‘some Smalltalk code’’}).
```

Note again that the code is put between curly brackets and thus is quoted (i.e. not evaluated).

The algorithm to be used for a certain user, can easily be changed by updating the user's profile. This can be done either statically or dynamically (by retracting the old fact and asserting the new one).

The rule `getAlgorithm` will retrieve the actuall algorithm from the user profile as follows :

```
Rule getAlgorithm(?user, ?action, ?algo) if
   algorithm(?user, ?action, ?name),
   call(?name(?algo)).
```

`?algo` will be bound to the quoted Smalltalk code, which is the algorithm's code. The `call` predicate calls the predicate bound to `?name`. For instance,

```
getAlgorithm(id201, checkout, ?algo)
```

will return the quoted Smalltalk code as specified in `1Click(?algo)` because in `getAlgorithm`, `checkout` is bound to `?action`, and thus `1Click` is bound to `?name`.

Personalization rules The personalization rule to update the algorithm depending on the user, will update the base-level method that executes the algorithm with the new code.

```
Rule checkoutAlgo(?ResultString) if
    currentCustomer(?ID),
    changeAlgorithm(?ID , checkout, ?ResultString).

Rule changeAlgorithm(?User, ?Task, ?AlgoCode) if
    getAlgorithm(?User, ?Task, ?AlgoCode).
```

The `?ResultString` code will again be used by the method wrapper's before method (similar as was done in section 5.3.1).

5.3.3 Structure

Personalizing structure means that different users have different authorization rights. Therefore for some users the personalization rules will have to specify that the information cannot be retrieved. As an example, consider users with and without permission to change the content description of a product (assuming a method `#changeContent`).

Permission in the User Profile A fact `permission` denotes whether a user has permission or not for a certain action. For example the user with id 201 has permission to change content:

```
Fact permission(id201, changeContent, true).
```

Personalization rules When a product receives the message `#changeContent`, the reasoning process will check wether the current user has permission to do so or not. If the user has no permission, an error will be returned by sending the message `noAccess` to the product (note that this is just an example, better strategies should be used).

```
Rule updateContent() if
    isReceiver([Product], [#changeContent], [Product]),
    currentCustomer(?ID),
    changeContent(?ID, false),
    [Product noAccess].

Rule changeContent(?Customer, ?Boolean) if
    hasPermission(?Customer, changeContent, ?Boolean).
```

5.4 Running the System

Up until now we demonstrated how we will express personalization by using SOUL. Now we will explain how we will switch between meta-level and base-level. Running the system requires two actions. On the one hand we want to be able to adapt the Smalltalk base system from the SOUL level. On the other hand we need to trigger the SOUL level from the Smalltalk level in order to start the reasoning process. This setup is depicted in figure 5.2. In the figure, going from SOUL level to Smalltalk level is denoted with underlined words, the other way around is denoted with italic words.

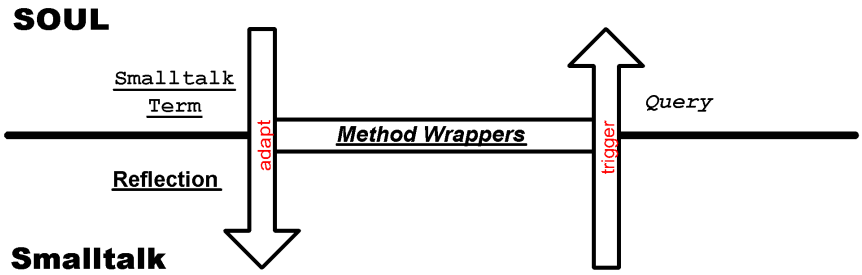

Fig. 5.2: Running the System : Soul ↔ Smalltalk

5.4.1 Adapting the base system

As said before it is the intention to be able to adapt the Smalltalk base-level from the SOUL meta-level. This is wished for because personalization will be described using the SOUL level, and somehow needs to be introduced into the base system.

Using *Smalltalk term* and Reflection to adapt the base system

Because of the Language Symbiosis (using *Smalltalk terms*) in SOUL, we can easily include Smalltalk code in a logic clause such that this code is executed when the rule succeeds, as was explained in section 4.2.3. It will allow us to send messages to Smalltalk objects.
Furthermore Smalltalk is a fully reflective language, and allows, through its meta-level, to introspect and absorb (for terminology see chapter 4.1.2) the Smalltalk system itself.
Combining these two features thus makes it possible to change methods, variables, classes, etc. by using simple Smalltalk message sends. For instance, the Smalltalk message `#subclass:-instanceVariableNames:classVariableNames:poolDictionaries:category:` is used to create a new class. Triggering the following SOUL rule will thus create a new class (subclass of `Object`) with as name `?Name` in the category `Example`:

```
Rule createClass(?Name) if
       [Object subclass: ?Name
               instanceVariableNames: ''
               classVariableNames: ''
               poolDictionaries: ''
               category: 'Example']
```

Adapting the Base System

Smalltalk's reflectiveness allows to change the base system, but it is not opportune to change the system, but merely adapt it. With *changing* the system we refer to permanent changes in the system, such as changing the value of an attribute or replacing the body of a method with new behavior. These kind of changes are feasible in Smalltalk, but they affect the system in such a way that it would be hard to restore the system in its original state.
What we actually want is to *adapt* the system temporarily such that it reflects a correct state for each user without affecting its original state regarding to other users.
To get a better understanding of this idea, consider the example of changing the price attribute of a certain product depending on whom is buying. When the price of the product

is requested, an accessor method is sent to retrieve this value. For certain users, depending on the kind of reduction they might be granted, the method has to return an adapted value. However, this adaptation is not the same for all users (some users might even have no reduction at all). Therefore it is not desired to change the attribute itself, but only what is returned when calling the accessor method.

To obtain an adapted system instead of a changed one, we used the mechanism of *method wrappers* [BFJR98]. These method wrappers will capture the accessor methods, in order to make it possible to make the adaptations. The kind of adaptation made reflects the personalization we are trying to achieve.
Method wrappers will make sure that personalization is performed when needed, but omitted when not desired. More on method wrappers is explained in section 5.4.3.

5.4.2 Triggering the SOUL meta-level

Although now we know how to use SOUL to adapt the base-level, we still need to be able to switch from base-level to SOUL level in order to trigger the SOUL level and perform the reasoning process.

Starting the reasoning process

Triggering SOUL is done by sending `Query test(?x)`. This starts the SOUL evaluator with the aim of solving this query.
In some cases it might be wishful to specify what repository to use. Rules are organized in repositories such that different kinds of personalization are separated from each other. If one knows what kind of personalization is needed, that particular repository can be specified. In this case the SOUL interpreter is called with `SOULParser interprete: 'Query test(?x)' in: aRepository`.

Triggering rules

Statically determined For some queries we know statically (before running the system) at what points in the base system they should be triggered. For instance, updating the price of a product will always be done when accessing the value of this price attribute. Thus, we can use the exact query. For instance `Query updatePrice(?ResultCode)` when updating the price of a product.

Dynamically determined In some cases however, we cannot know in advance what query is to be triggered. In this case, we launch a general query (`Query personalize(?Result)`) that will trigger the meta-level. This general query corresponds to a rule `personalize(?Result)` on the SOUL level. When triggering this rule, we want to query all other rules, such that relevant rules are actually executed, and thus personalize the base-level.
Note that SOUL was originally not intended to do this kind of dynamic reasoning (see chapter 4.2.4), and the proposed solution might seem quite cumbersome. A better solution for this problem is an issue to invest in the future.

When calling a general query no variables can be passed, but this poses no problems since all information that is needed for personalization resides either in the user profile or in the

Smalltalk base-level. Both of these sources of information can easily be accessed from SOUL. Furthermore this information can be updated at runtime by querying rules that assert or retract facts.

Also will the rule `personalize(?Result)` have to specify that all the rules in the repository have to be retrieved and called one by one. The rule to express this might result in a huge unreadable construct. However, rules can be organized in layers (see section 5.1.2) and if for each of these layers a similar general rule is specified, `personalize(?Result)` will only have to call these "layer rules". This will make the whole more readable, and also allows some layers of rules to be discarded if that is wished for.

Dynamic vs Static Determining (statically or dynamically) what query should be called requires further research, and has not been explored in this dissertation due to time constraints. Since we did not investigate a way to determine statically what query should be called when, we assume that we only perform the query `personalize(?Result)` at each possible point where we personalize the base system.
The following rules illustrate how we could change the `personalize(?Result)` queries to allow dynamic determination of the exact query to perform.

```
personalize(?Type, ?Result) if
   var(?Type),
   personalize(?Result).

personalize(?Type, ?Result) if
   atom(?Type),
   call(?Type(?Result)).
```

The first rule will fire when `?Type` is unbound, and thus cannot be determined at run-time. In this case the general query `personalize(?Result)` has to be called.
The second rule will fire when `?Type` is bound at run-time, and thus dynamically determined. In this case a specified query can be called. For instance `personalize(updatePrice, ?Result)` results in querying `updatePrice(?Result)`.
This way all method wrappers contain the same query `personalize(?Result)` to query SOUL. Nevertheless the problem of static calls versus dynamic calls remains to be explored.

When to start the query?

Where exactly the meta-level is triggered from the base system will depend on the personalized system. For example in recommender systems this can be when the user logs in, performs a buying action, when he/she clicks on a certain link, ...

5.4.3 Method Wrappers

Wrappers are a mechanism for introducing new behavior that is executed before and/or after, and perhaps even instead of an existing method. These wrappers can be implemented in several ways, and method wrappers use a slightly different approach than the one normally conducted by Smalltalk programmers [BFJR98].

A method wrapper replaces the old method by a new one that will invoke the old one. This is done by replacing the old method with a method wrapper object. These objects have

- an instance variable `clientMethod` : stores the original method,
- a method `beforeMethod` : contains the before code
- a method `afterMethod` : contains the after code, and
- a method `valueWithReceiver:arguments:` : executes the original method given the receiver and argument array.

In [BFJR98] the method wrapper approach is fully explained, together with how to add the wrappers such that the original system does not have to be recompiled.

Method wrappers to personalize the base system We will use the method wrappers to introduce our personalization into the base system. The `before` or `after methods` of the method wrapper will contain a SOUL query, which will result in some Smalltalk code that has to be executed. This piece of Smalltalk code can also contain a call to the original method, by using the `valueWithReceiver:arguments:` method.
In practice, the before method of the method wrapper will launch a SOUL query, fetch the result of this query and evaluate the piece of code that resulted from this query. The result of this evaluation is returned.
In our example of updating the price (see section 5.3.1), the before method of the wrapper that wraps the method `#getPrice` on the `Product` class will look like :

```
beforeMethod
    Query personalize(?ResultCode).
    string := '[',?ResultCode,']'.
    block := (Compiler evaluate: string)
    ^ block value
```

and thus the code to update the value, specified in `?ResultCode` is executed. `?ResultCode` is a string of code and can be executed as a Smalltalk block.
Note that `string := '[',?Result,']'` is only used to represent the main idea, but should be replaced with some code to retrieve the actual string from `?Result`.

Method Wrappers are just a technique The method wrappers are only a technique for realizing personalization. One may argue that the wrappers are part of the process to get a personalized base system, and thus that our approach to personalize still makes changes to the base system. However, because of the reasons described in section 5.4.1, SOUL can be used to actually add these method wrappers to the system. Brant et al. [BFJR98] describe the code to be used to add wrappers. It is sufficient to create rules using `Smalltalk terms` to introduce this code into the base system. Calling these rules at the beginning of the base system's execution will then add the right constructs. The rest of the personalization and execution remains the same.

5.5 An Example

In this section we give an overview of how the mechanisms in the previous chapter are to be applied step by step. As an example we consider again an e-commerce system, where we want to update the price of a product, depending on the user's discount.

Base-level : Method Wrappers

A first step towards personalization is the addition of method wrappers. How method wrappers can be added is fully explained by Brant et al. [BFJR98].
The before method (or after method) of this method wrapper contains a query call. As we explained in section 5.4.2 we use a general query of the form `Query personalize(?Result)`. As a reminder, this is what the body of before method of a method wrapper might look like :

```
beforeMethod
    Query personalize(?ResultCode).
    string := '[',?ResultCode,']'.
    block := (Compiler evaluate: string)
    ^ block value
```

Method wrappers are added to the basic system manually, but as explained in section 5.4.3, also SOUL can be used to add these method wrappers. The code to add wrappers is then described in rules that are to be triggered at the startup of our e-commerce system.

When updating the price of a product, the accessor method to retrieve the value of price (e.g. `#getPrice`) has been method wrapped, and the before method's body is as described above.

Meta-level : User Profile

On the meta-level we store the user profile as a set of logic facts. For instance user with id408 has the following facts :

```
Fact name(id408, jef).
   Fact age(id408, 28).
   Fact reduction(id408, 100).
```

Meta-level : Personalization rules

On the meta-level we describe the personalization rules. In the previous chapter we showed that different kinds of personalization is possible, together with examples of the different kinds of rules. We shortly repeat the rule that will make sure the price of a product is updated :

```
Rule changePrice(?Customer, ?Product, {"Code to update Price"}) if
        reduction(?Customer, ?Value).
```

The reduction that is assigned to a user, is retrieved from the user profile. For example, if the current customer is the customer (with identity) id408, `?Value` will be bound to `100`. This value is then used in the Smalltalk code (quoted code term) that will calculate the new value

of the product's price.
A simple way to get a new value is

```
Rule changePrice(?Customer, ?Product,
 {^(clientMethod valueWithReceiver: ?Product arguments: []) - ?Value }) if
       reduction(?Customer, ?Value).
```

The first part will call the original method (`clientMethod`) that is wrapped by the method wrapper. Both `?Product` and `?Value` are SOUL variables that will be bound to their respective values when this code is executed.
More interesting strategies can be based on the amount of products a certain user already bought, on special discount codes, on special actions, and so on...

The rule

```
Rule updatePrice(?ResultString) if
   isReceiver([Product], [#getPrice], [?aProductObject]),
   currentCustomer(?ID),
   changePrice(?ID, [?aProductObject], ?ResultString).
```

specifies that when the method `getPrice` is received by the class `Product`, `changePrice` is triggered. The Smalltalk terms (denoted by square brackets) refer to the current `Product` object and method `#getPrice`.
?ResultString will be bound to quoted code term `"Code to update Price"`.

Calling the Query

When the e-commerce system is running and a product is asked for its price, the method wrapper catches the accessor method and launches the query `personalize(?Result)`. All the rules at the meta-level will be triggered, and finally `updatePrice` will succeed because of `isReceiver([Product], [#getPrice], [?aProductObject])`. The query returns with ?ResultString (now bound to a piece of code) as result.
The query result is thus a piece of code that will be executed by the method wrapper (because of `Compiler evaluate: string`). The value of this execution, which in our example is a new value for the price attribute, is returned. The value of the price attribute has been personalized.

5.6 Summary

In this chapter we demonstrated three kinds of personalization (as we lined out in chapter 3) in object-oriented systems. To achieve this we used SOUL, a logic meta programming language. Personalization was thus described using logic rules. We explained the setup needed to change between the SOUL level and Smalltalk level, and the idea of method wrappers as a technique to introduce personalization into the base system. We finished with an example that explains our approach step by step.

Chapter 6

Conclusion and Future Work

We will now discuss the results we obtained in the previous chapters and we will draw conclusions with regard to our initial goal of this thesis.

6.1 Motivation and initial Goal

Object-oriented systems are gaining importance. A lot of research has been done in the field of object-orientation, and this paradigm is now also becoming popular in industry. This growing interest from researchers as well as practitioners makes it a real challenge to expand the use of this paradigm for all of today's applications.

Personalization is another concept with raising importance, especially in e-commerce systems. Personalized systems intend to adapt applications to the user's individual needs. At first, personalization was fairly primitive, but (partly because of strong competition) it became more complicated as time evolved. However, complicated systems are more difficult to maintain and evolve.

It would be profitable to use the object-oriented paradigm to create personalized systems. Web-applications are becoming more object-oriented and object-oriented analysis, design and implementation are used more often. This makes investigating personalization in object-oriented systems interesting.

The objective of this thesis was to investigate how personalization can be introduced into object-oriented systems. We particulary investigated the approach of using logic meta programming to describe user facts and personalization rules on a meta-level.

6.2 Summary and Results

To achieve our goal, we started with an introductory chapter about personalization and its currents applications. We talked about recommender systems, user profiles and other uses of personalized systems.

In the second chapter we dealt with the different approaches to introduce personalization in object-oriented systems. The first one we mentioned is the oldest one, and hard codes

personalization into the system. It is not advisable if you want to pursue a clean separation of concerns.
The second approach (OOHDM) made a first step in the good direction by splitting the conceptual level from the navigational model, and adding personalization to this second model, such that the basic system (conceptual level) is not affected by personalization. This makes the system better evolvable, but nevertheless there is still a hard-coding of personalization at navigational level.
Thirdly we discussed a decoupling of the basic system and its personalization at base-level, either by the use of design patterns, components or value-models.
Finally we looked at decoupling at meta-level. The basic system then resides at base-level, while the personalization for this system resides at meta-level. Using this approach allows a clear and clean separation between the basic system and its personalization. This makes it more easy to change personalization aspects without touching the basic system. Different techniques are suited for this approach, for instance the use of aspect oriented programming and the use of logic meta programming. In this dissertation we chose to explore the latter.

In chapter 3 we explained Logic Meta Programming (LMP) in more detail. We introduced Logic Programming, and then continued towards Logic Meta Programming. Furthermore we discussed the Smalltalk Open Unification Language (SOUL), being an example of such a Logic Meta Programming Language. The chapter concluded with some remarks on the usefulness of LMP for introducing personalization into an object-oriented system.

An example of how SOUL can actually be used to achieve our goal, is presented in chapter 5. First we presented our architecture, clearly showing the separation between base-level and meta-level entities. Next we zoomed in on the user profile and the personalization rules. Finally we discussed how these personalization descriptions can be turned into a practical usable mechanism.

6.3 Final Conclusion

To conclude this thesis, we can certainly state that SOUL proved very useful for introducing personalization in an object-oriented system. SOUL's symbiosis with Smalltalk allows to write quite powerful rules. Furthermore its declarative nature leads to simple personalization descriptions that can be read in a natural way.

Performing a practical case-study is important but not necessary to validate the usefulness of our approach, because our examples are sufficiently conclusive. But, although it was impossible to realize this within the given time constraints, we admit that it forms an important part of our future work.
Although future research still might indicate that SOUL is not as suited as we believe it to be, decoupling the basic system and its personalization using a meta-level approach is a minimal requirement. Further research will definitely lead to important improvements in creating personalized systems.

Bibliography

[Asp00] *www.aspectj.org*, AspectJ homepage, 2000.

[AT99] Gediminas Adomavicius and Alexander Tuzhilin, *User profiling in personalization applications through rule discovery and validation*, Proceedings KDD99, Knowledge Discovery and Data Mining, 1999.

[Aub01] Olivier Aubert, *Adaptive strategy design proto-pattern*, Proceedings of The Second Asian Pacific Pattern Languages of Programming Conference (KoalaPLoP '2001), March 2001.

[Bec97] Kent Beck, *Smalltalk best practice patterns*, Prentice Hall, 1997.

[Bel00] Nicholas Belkin, *Helping people find what they don't know*, Communications of the ACM **43** (2000), no. 8, 58 – 61.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts, *Wrappers to the rescue*, Proceedings ECOOP'98, 1998, pp. 396 – 417.

[CDA00] Ibrahim Cingil, Asuman Dogac, and Ayca Azgin, *A broader approach to personalization*, Communications of the ACM **43** (2000), no. 8, 136 – 141.

[DM00] Wim De Muynck, *Aspect-Oriented Programming, a survey of current research*, Capita selecta, EMOOSE, 1999-2000.

[Fla94] Peter Flach, *Simply logical*, John Wiley and sons, 1994.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns*, Addison-Wesley, 1994.

[GSK+99] Nathaniel Good, J. Ben Schafer, Joseph A. Konstan, Al Borchers, Badrul M. Sarwar, Jonathan L. Herlocker, and John Riedl, *Combining collaborative filtering with personal agents for better recommendations*, AAAI/IAAI (GroupLens Research Project, Department of Computer Science and Engineering, University of Minesota, Minneapolis), 1999, pp. 439–446.

[HKR00] Jonathan Herlocker, Joseph Konstan, and John Riedl, *Explaining collaborative filtering recommendations*, ACM Conference on Computer Supported Cooperative Work (GroupLens Research Project, Department of Computer Science and Engineering, University of Minesota, Minneapolis), December 2000.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, *An overview of AspectJ*, ECOOP (2001).

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, *Aspect-oriented programming*, ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, vol. 1241, Springer-Verlag, New York, NY, 1997, pp. 220–242.

[MAB00] Maurice D. Mulvenna, Sarabjot S. Anand, and Alex G. Bchner, *Personalization on the net using web mining*, Communications of the ACM **43** (2000), no. 8, 122 – 125.

[Mae87] Pattie Maes, *Computational reflection*, Phd thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[MPR00] Udi Manber, Ash Patel, and John Robison, *Experience with personalization of Yahoo!*, Communications of the ACM **43** (2000), no. 8, 35 – 39.

[RSG01] Gustavo Rossi, Daniel Schwabe, and Robson Mattos Guimarães, *Designing personalized web applications*, Proceedings of the 10^{th} International Conference on the WWW (WWW10), Hong Kong, 2001.

[RSL99] Gustavo Rossi, Daniel Schwabe, and Fernando Lyardet, *Web application models are more than conceptual models*, Springer Verlag, November 1999.

[SC00] Barry Smyth and Paul Cotter, *A personalized television listings service*, Communications of the ACM **43** (2000), no. 8, 107 – 111.

[SKKR00] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John Riedl, *Analysis of recommendation algorithms for e-commerce*, ACM Conference on Electronic Commerce (GroupLens Research Project, Department of Computer Science and Engineering, University of Minesota, Minneapolis), 2000, pp. 158–167.

[SKR99] J. Ben Schafer, Joseph A. Konstan, and John Riedl, *Recommender systems in e-commerce*, ACM Conference on Electronic Commerce, 1999, pp. 158–166.

[Spi00] Myra Spiliopoulou, *Web usage mining for web site evaluation*, Communications of the ACM **43** (2000), no. 8, 127 – 134.

[SR98] Daniel Schwabe and Gustavo Rossi, *An object-oriented approach to web-based application design*, Theory and Practice of object Systems (TAPOS), pp. 207–225, october 1998.

[SRG] Daniel Schwabe, Gustavo Rossi, and Robson Mattos Guimarães, *Cohesive design of personalized web applications*, In preparation.

[Ste94] Patrick Steyaert, *Open design of object-oriented languages, a foundation for specialisable reflective language frameworks*, Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, 1994.

[Val01] *www.object-arts.com/educationcentre/overviews/valuemodel.htm*, Object Arts Website, 2001.

[Vol00] Eugene Volokh, *Personalization and privacy*, Communications of the ACM **43** (2000), no. 8, 84 – 88.

[Wuy01] Roel Wuyts, *A logic meta-programming approach to support the co-evolution of object-oriented design and implementation*, Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, January 2001.

[WW00] Nigel Wells and Jef Wolfers, *Finance with a personalized touch*, Communications of the ACM **43** (2000), no. 8, 31 – 34.