Anais do Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software

(SBCARS 2007)

29 a 31 de Agosto de 2007

Campinas, São Paulo, Brasil

Promoção

SBC – Sociedade Brasileira de Computação

Edição

Cecília Mary Fischer Rubira Universidade Estadual de Campinas

Organização

Instituto de Computação – Unicamp

Realização

Instituto de Computação – Unicamp

Universidade Estadual de Campinas – Unicamp

Si57a

Anais do Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (2007 : Campinas, SP).

Anais do Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software, 29 a 31 de agosto de 2007, Campinas, SP, Brasil / Cecília Mary Fischer Rubira. -- Campinas, SP : UNICAMP 2007

1. Componentes de software. 2. Software – Confiabiliadade. 3 – Software – Arquitetura. 4 – Software – Reutilização. I. Rubira, Cecília Mary Fischer. II. Título

CDD - 001.642

- 001.6425

- 005.3

Índices para Catálogo Sistemático

1.Componentes de software	001.642
2.Software – Confiabilidade	001.6425
3.Software – Arquitetura	001.6425
4.Software – Reutilização	005.3

Projeto gráfico: Maurício Bedo - Digital Assets, Campinas, SP.

Editoração:

Leonel Aguilar Gayard	Instituto de Computação – Unicamp
Cecília Mary Fischer Rubira	Instituto de Computação – Unicamp

Esta obra foi impressa a partir de originais entregues, já compostos pelos autores

Sumário / Contents

Sessões técnicas (ST) / Technical Sessions (TS)

Technical Session I: Software Product Lines

Technical Session II: Methods and Models for Software Reuse

Technical Session III: Software Architectures and Components

Technical Session IV: Model-driven Development and Web Services

Palestras convidadas / Invited Talks

Tutoriais convidados / Invited Tutorials

Mini-cursos convidados / Invited short courses

Prefácio

É com alegria e satisfação que apresento os Anais do Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS 2007), promovido pela Sociedade Brasileira de Computação (SBC), (http://www.ic.unicamp.br/sbcars2007).

O SBCARS 2007 reúne pesquisadores, estudantes e profissionais com interesses em engenharia de software baseada em componentes, arquiteturas e reutilização de software. O SBCARS 2007 é o sucessor do VI Workshop Brasileiro de Desenvolvimento Baseado em Componentes (WDBC 2006), avaliado como evento nacional nível "B", de acordo com o Qualis Ciência da Computação (Qualis-CC) da CAPES. O sucesso das edições anteriores do WDBC fez com que o "workshop"se tornasse sólido e mais abrangente para se transformar num simpósio na sua edição de 2007.

O SBCARS 2007, realizado no Campus da Universidade Estadual de Campinas (UNICAMP), Campinas, SP, de 29-31/8/2007 e organizado pelo Instituto de Computação-UNICAMP, recebeu 42 artigos completos, dos quais foram selecionados 14 para apresentação e publicação nestes anais, com uma taxa de aceitação de 33,3%. O processo de avaliação garantiu que cada submissão tivesse pelo menos três avaliações e foi apoiado pela ferramenta JEMS fornecida pela SBC. Além disso, 50% dos artigos aceitos para publicação estão escritos em inglês. Acreditamos que isso seja uma evidência de atividade crescente nessa área de pesquisa e da importância da realização continuada de muitos SBCARS.

O SBCARS 2007 traz duas novidades em relação ao VI Workshop de Desenvolvimento Baseado em Componentes (WDBC 2006). A primeira é a chamada para submissões de ferramentas voltadas para a área de componentes, arquiteturas e reutilização de software, organizada pelo Prof. Antônio Prado e Valdirene Fontanette do Departamento de Computação da UFSCAR. Os trabalhos descrevendo as ferramentas selecionadas estão publicados nos "Anais da Sessão de Ferramentas do SBCARS 2007", serão apresentados oralmente na sessão de ferramentas e as ferramentas serão demonstrados durante o evento. A segunda novidade é uma seleção dos melhores artigos do SBCARS 2007 a ser publicada numa edição especial do "Journal of Universal Computer Science", em abril de 2008 (http://www.jucs.org/ujs/jucs/info/special_issues/in_preparation.html). Os autores dos artigos aceitos serão convidados a submeterem uma versão estendida dos seus trabalhos, que então será novamente avaliada.

O SBCARS 2007 têm uma abrangência nacional e inclui também a participação de pesquisadores internacionais de renome tanto no seu comitê de programa quanto nos trabalhos publicados nos seus anais. Além disso, o programa do evento se destaca pela excelência das palestras ministradas por pesquisadores internacionais, tutoriais e mini-cursos com temas atuais, sessões técnicas e de ferramentas. O evento apresenta também uma sessão industrial dedicada a discutir avanços da indústria na aplicação prática dos conceitos de reutilização de software, bem como, identificar novos desafios de pesquisa.

Agradeço imensamente o apoio recebido pelo Instituto de Computação, CNPq, CAPES, FAEPEX-UNICAMP e FAPESP para a realização desse evento. Agradeço

também o patrocínio recebido da empresa Digital Assets, que contribuiu muito para a organização desse evento, da Microsoft e também da SBC.

Agradeço também o trabalho e a dedicação da comissão organizadora, do comitê de programa, dos avaliadores dos artigos, dos palestrantes convidados e de todos os pesquisadores que submeteram trabalhos para este Simpósio.

Campinas, 7 de agosto de 2007.

Cecília Mary Fischer Rubira

Coordenadora do SBCARS 2007

Comissão Organizadora

- Cecília Mary Fischer Rubira (Coordenadora Geral) — Instituto de Computação (IC) - UNICAMP
- Antônio Francisco Prado e Valdirene Fontanette (Coordenadores da Sessão de Ferramentas) — Departamento de Computação (DC) - UFSCar
- Profa. Ariadne Rizzoni Carvalho — Instituto de Computação (IC) - UNICAMP
- Profa. Thelma Chiossi — Instituto de Computação (IC) - UNICAMP
- Patrick Henrique da Silva Brito — Instituto de Computação (IC) - UNICAMP
- Leonardo Pondian Tizzei — Instituto de Computação (IC) - UNICAMP
- Leonel Aguilar Gayard — Instituto de Computação (IC) - UNICAMP
- Ana Elisa de Campos Lobo — Instituto de Computação (IC) - UNICAMP
- Ivan Perez — Instituto de Computação (IC) - UNICAMP
- Claudia Regina da Silva — Instituto de Computação (IC) - UNICAMP
- Kleber Bacili — Digital Assets, Campinas,SP
- Ana Martini — Digital Assets, Campinas,SP
- Maurício Bedo — Digital Assets, Campinas,SP

Comitê de Programa

- Cecília Mary Fischer Rubira (Coordenadora do Comitê de Programa) — Instituto de Computação (IC) - UNICAMP
- Cláudia Maria Lima Werner (Vice-Coordenadora do Comitê de Programa) — COPPE - UFRJ

Membros do Comitê de Programa

- Alessandro Garcia — University of Lancaster
- Alexander Romanovsky — University of Newcastle
- Ana C.V. de Melo — USP
- Ana Paula Bacelo — PUCRS
- Antônio Francisco Prado — UFSCar

- Carlos Lucena — PUC-Rio
- Cecília M.F. Rubira — UNICAMP
- Cláudia Werner — UFRJ
- Cristina Gacek — University of Newcastle
- Eliane Martins — UNICAMP
- Glêdson Elias — UFPB
- Guilherme Travassos — UFRJ
- Itana Gimenes — UEM
- Ivica Crnkovic — University of Mälardalen
- José Maldonado — USP-São Carlos
- Mehdi Jazayeri — University of Lugano
- Patrícia Machado — UFCG
- Paulo Borba — UFPE
- Paulo Masiero — USP-São Carlos
- Paulo Merson — Software Engineering Institute
- Regina Braga — UFJF
- Rogério de Lemos — University of Kent
- Rosana Braga — USP-São Carlos
- Sílvio Meira — UFPE
- Thaís Vasconcelos Batista — UFRN

Revisores

- Alessandro Garcia
- Alexander Romanovsky
- Alexandre Alvaro
- Aline Vasconcelos
- Ana C. V. de Melo
- Ana Elisa Lobo
- Ana Paula Bacelo
- Antonio Francisco Prado
- Carlos Lucena
- Cecília Rubira
- Cidiane Aracaty Lobato
- Cláudia Werner
- Cláudio Nogueira Sant'Anna
- Cristina Gacek
- Daniela Francisco Brauner
- Eduardo Santana de Almeida
- Eliane Martins
- Elisa Nakagawa
- Emanuela Cartaxo
- Fabiano Cutigi Ferrari
- Fernando Castor Filho
- Gledson Elias
- Guilherme Travassos
- Hélio Guardia
- Hyggo Almeida
- Itana Gimenes
- Ivica Crnkovic
- Jair Cavalcanti Leite
- Jobson Massollar
- Leonardo Murta
- Leonardo Pondian Tizzei
- Leonel Aguilar Gayard
- Maíra Athanázio de Cerqueira Gatti
- Marcelo Eler
- Márcio Aguiar Ribeiro
- Marco Pereira
- Marco Antônio Araújo
- Marcos Chaim
- Masiero Paulo
- Mehdi Jazayeri
- Patricia Machado
- Patrick da Silva Brito
- Paula Donegan
- Paulo Asterio Guerra
- Paulo Borba
- Paulo Merson
- Regina Braga
- Reginaldo Ré
- Rodrigo Spínola
- Rogerio de Lemos
- Rosana Braga
- Silvio Lemos Meira

- Thais Vasconcelos Batista
- Uirá Kulesza
- Valdirene Fontanette
- Vinícius Cardoso Garcia
- Wilkerson Andrade

Sociedade Brasileira de Computação

Diretoria

- Presidente: José Carlos Maldonado (ICMC - USP)
- Vice-Presidente: Virgílio Augusto Fernandes Almeida (UFMG)

Diretorias:

- Administrativa: Carla Maria Dal Sasso Freitas (UFRGS)
- Finanças: Paulo Cesar Masiero (ICMC - USP)
- Eventos e Comissões Especiais: Marcelo Walter (UFPE)
- Educação: Edson Norberto Cáceres (UFMS)
- Publicações: Karin Breitman (PUC-Rio)
- Planejamento e Programas Especiais: Augusto Sampaio (UFPE)
- Secretarias Regionais: Aline dos Santos Andrade (UFBA)
- Divulgação e Marketing: Altigran Soares da Silva (UFAM)

Diretorias Extraordinárias:

- Regulamentação da Profissão: Ricardo de Oliveira Anido (UNICAMP)
- Eventos Especiais: Carlos Eduardo Ferreira (USP)
- Cooperação com Sociedades Científicas: Taisy Silva Weber (UFRGS)

Conselho

Mandato 2007-2011

- Cláudia Maria Bauzer Medeiros (UNICAMP)
- Roberto da Silva Bigonha (UFMG)
- Cláudio Leonardo Lucchesi (UNICAMP)
- Daltro José Nunes (UFRGS)
- André Ponce de Leon F. de Carvalho (ICMC - USP)

Mandato 2005-2009

- Ana Carolina Salgado (UFPE)
- Jaime Simão Sichman (USP)
- Daniel Schwabe (PUC-Rio)

Suplentes - Mandato 2007-2009

- Vera Lúcia Strube de Lima (PUCRS)
- Raul Sidnei Wazlawick (UFSC)
- Ricardo Augusto da Luz Reis (UFRGS)
- Jacques Wainer (UNICAMP)
- Marta Lima de Queiroz Mattoso (UFRJ)

Technical Session I: Software Product Lines

Design Issues in a Component-based Software Product Line

Paula M. Donegan*, Paulo C. Masiero

Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (USP)
Caixa Postal 668 – 13.560-970 – São Carlos – SP – Brazil
{donegan,masiero}@icmc.usp.br

Abstract. *A software product line to support urban transport systems is briefly described and the design of two of its features is discussed. Different solutions based on components are shown for these two features and their variabilities. In particular, an analysis is made of how their design is influenced by the development process adopted, by the decision to use black-box (off-the-shelf) components or white-box components that may be created or adapted depending on application requirements, and by the decision of automating or not the composition process. Additionally, alternatives for deciding how to define iterative cycles and increments of the product line are discussed.*

1. Introduction

A software product line (SPL) consists of a group of software systems sharing common and managed features that satisfy the specific needs of a market segment or a particular objective and are developed in a predefined manner given a collection of core assets [Clements and Northrop, 2002]. The design of an SPL can use various software design techniques that facilitate reuse, such as object-oriented frameworks, components, code generators, design patterns, features diagrams and aspect-oriented languages. Several papers emphasize the difficulty of gathering, representing and implementing variabilities in the context of SPLs [Bachmann et al, 2004; Becker, 2003; Bosch et al, 2001, Junior et al, 2005]. Variability in an SPL differentiates products of the same family [Weiss and Lai, 1999].

This paper has two main objectives: to illustrate different solutions based on components to represent variabilities of an SPL and to discuss how these solutions are influenced by the adopted development process, by the decision to use black-box or off-the-shelf (COTS) components (without access to the source code) which are reused as they are or to use white-box components (with access to the source code) which may be created or adapted according to application requirements, and by the decision of automating the composition process. The solutions are presented in the context of an SPL (being developed as an academic project by the authors) to simulate support of urban transport systems.

The organization of the paper is as follows: Section 2 briefly describes the SPL; Section 3 presents some generic alternatives for the iterative and incremental development of an SPL; Section 4 summarizes the process used for the SPL; Section 5

* With financial support of FAPESP (*Fundação de Amparo à Pesquisa do Estado de São Paulo*)

discusses design of components for two features of the SPL; Section 6 presents some conclusions.

2. The Software Product Line to Control Electronic Transportation Cards

The SPL used as an example in this paper concerns management of electronic transport cards (ETC) named ETC-SPL. These systems aim to facilitate the use of city transport, mainly buses, offering various functionalities for passengers and bus companies, such as use of a plastic card to pay fares, automatic opening of barrier gates, unified payment of fares, integration of journeys and supply of on-line travel information to passengers.

The software allows the integration and automation of the transport network, with a centralized system that maintains the data of passengers, cards, routes, buses and journeys. The buses are equipped with a validator that reads a card and communicates with the central system (for example using RFID – Radio Frequency Identification) to debit the fare on the passenger's card. There may also be a bus integration system that permits the user to pay a single fare for multiple trips. In addition, passengers can go on-line and look up their completed trips and card credit.

The system domain was analysed and the ETC-SPL is being designed with the objective of generating at least three applications (or products) based on the analysis of three existing ETC systems in Brazilian cities: São Carlos (São Paulo), Fortaleza (Ceará) and Campo Grande (Mato Grosso do Sul).

3. Development Process of Software Product Lines

The literature describes various processes for the development of an SPL [Gomaa, 2004; Atkinson et al, 2000]. In general, they recommend that an organization wanting to develop an SPL has developed at least three similar applications belonging to the same domain [Roberts and Johnson, 1998; Weiss and Lai, 1999]. The evolution of an SPL may be proactive (ad hoc) or reactive (planned) [Sugumaran et all,2006]. An intermediate approach, called extractive, occurs when a second or third application is developed, parts of the code of one or more of the existing software products are generalized in such a way that they can be reused, until at a certain moment all the code is refactored so that new applications are capable of reusing a substantial part of the core assets.

In the case of a proactive evolution, the organization can use a process based on reverse engineering or forward engineering that differ basically in their first phase, as proposed by Gomaa (2004). In the process based on reverse engineering, artifacts of analyses, such as use cases and conceptual models, are recreated from existing systems. In the process based on forward engineering the same artifacts are derived from various sources, such as existing requirements documents and processes for requirements capture. From this point on, both processes are similar and domain analysis considers the use cases which are common to all applications of the domain, constituting the kernel of the SPL, and those which are optional (existing only for some of the SPL products) or alternative (choosing from a set of possibilities). A general conceptual model is created representing the common and variable parts. Afterwards a features diagram can be developed to synthesize the common and variable parts of the SPL.

There are several models of processes for SPL development, all beginning with the domain analysis phase described superficially in the previous paragraph. One alternative is then to elaborate the design for the entire modeled domain. The implementation can be done afterwards, in one version only or in various partial increments. This alternative seems to be uneconomic and complex [Gomaa, 2004; Atkinson and Muthig, 2002].

Another option is to follow a more agile iterative and incremental process, in which the SPL is first designed and implemented in a version that contains only kernel features, and then incremented by the design and implementation of subgroups of optional and alternative variabilities, as proposed by Gomaa (2004). The SPL is based on components and variabilities of several different mechanisms such as inheritance, extensions (e.g. the strategy design pattern), configuration, template instantiation and generation can be implemented [Bosch, 2000].

The choice of increments to be produced in each iterative cycle can be done horizontally or vertically and this has a great influence on the design of the SPL architecture and on the components that implement variabilities, as is shown in Section 5. The horizontal increments are planned by including a subgroup of features that attend to a specific application but do not necessarily contain all possible variabilities of each feature included in the increment. The vertical increments implement, in a general and complete way, all the variabilities of a subgroup of chosen features, but do not necessarily produce a specifically desired application. Using the ETC-SPL as an example, a horizontal version could be one that would generate the ETC system for the city of São Carlos. A vertical version for the ETC-SPL would be an SPL containing all the possible forms of journey integration specified during the domain analysis. These possibilities are shown schematically in Figure 1.

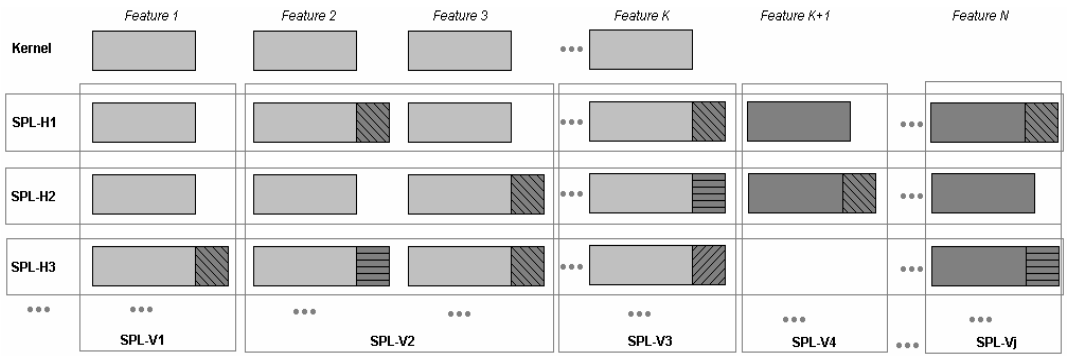


Fig. 1 – Vertical and horizontal increments

The behavior of variabilities in horizontal versions is shown in Figure 1 by the different shadings of variabilities extending a basic feature contained in the kernel. The figure illustrates, for example, features that do not appear in the kernel and do appear in a later version, features that appear in the kernel and are extended in one way for one version and in a different way for another version, etc. With the adoption of an evolving process such as that shown in Figure 1, each variability needs a careful design, because it may require refactoring in later versions. In other words, a design that is adequate for one version may not be so for a later version.

Horizontal increments are more realistic economically, in the sense that the SPL evolves as new applications need to be incorporated to the line, even though they can require more rework as the line evolves. On the other hand, vertical increments, even when not producing a previously foreseen application after the first iterations, have the advantage of allowing each chosen feature to be analysed and designed globally, including all its variabilities for the domain.

Another important decision to be made is how to develop the applications during the phase of application engineering, either using a manual process of generation of components that implement the SPL (in this case they correspond to the software assets available) or using an automated process, for instance a code generator. A solution that is adequate for a manual increment before automation may not be the best for a generative process. Thus it seems best to analyse all specified variabilities of a certain feature, before committing to a definite automated solution. However, if the SPL is composed mainly of black-box (off-the-shelf) components they impose an important restriction that leads to a solution using component composition and automatic generation of glue code.

Specifically for the ETC-SPL, we started with the specifications of three known products, the ETC systems of Fortaleza, Campo Grande and São Carlos, which were the applications to be generated initially. We considered it important to have a complete application early on, therefore opting to use horizontal iterative cycles in which each increment allows generation of one of these applications.

4. Development of the ETC-SPL: Iterative Cycles and Kernel Architecture

The points observed in the previous section were taken into consideration and PLUS (Product Line UML-based Software Engineering) [Gomaa, 2004] was used for the development of the ETC-SPL. The SPL Engineering was divided into two phases: in the Inception phase domain analysis yielded initial use cases, the feature diagram and a conceptual model, among other artifacts; for the Elaboration phase five iterations have been planed, each one producing a version of the ETC-SPL:

- Iteration 1: Comprising only features of the kernel.
- Iteration 2: Version 1 + features and variabilities of the application of Fortaleza.
- Iteration 3: Version 2 + features and variabilities of the application of Campo Grande.
- Iteration 4: Version 3 + features and variabilities of the application of São Carlos.
- Iteration 5: Version 4 with all variabilities but automatically generated with an Application Generator.

The features diagram for the kernel of the ETC-SPL (common features) is presented in Figure 2 using the notation of Gomaa (2004) and Figure 3 shows the architecture of the ETC-SPL kernel. The architecture is composed of three distributed modules, of which there is one occurrence of the server module (*ETCServer*) and various occurrences of each of two client modules (*Bus* and *WebAccess*). Internal to these modules are the components of the SPL, derived following the processes

suggested by Gomaa (2004) and by Cheesman and Daniels (2001), the latter used specifically for the identification of business and system components. The design based on components of some variabilities of the ETC-SPL is presented and discussed in Section 5. The additional features of the ETC systems of each city are the following:

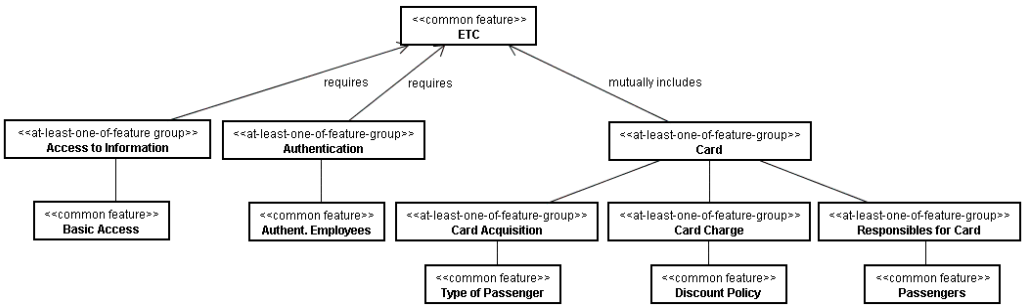


Fig. 2 – Features diagram for the kernel ETC-SPL

- **Fortaleza**: Form of Integration (Transport Hub), Card Payment, User Companies.
- **Campo Grande**: Additional Access, Form of Integration (Integration (Time, Integration Route, Number of Integration Trips), Transport Hub), Card Restriction (Number of Cards), User Companies.
- **São Carlos**: Additional Access, Passenger Authentication, Form of Integration (Integration (Time, Integration Route)), Card Restriction (Card Combination), Trip Limit.

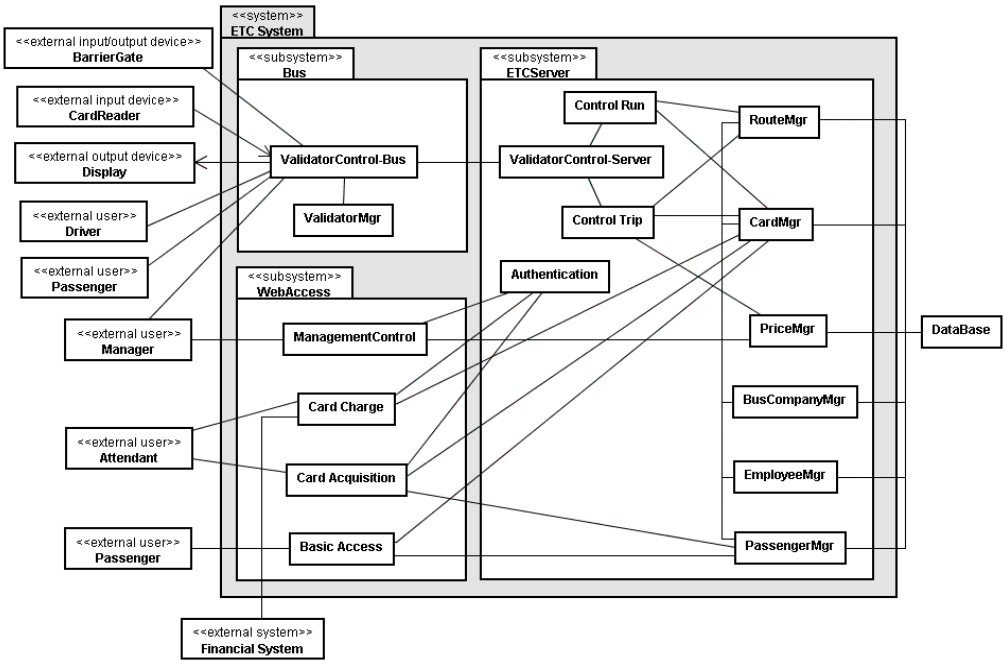


Fig. 3 – Kernel architecture of the ETC-SPL

5. Design Decisions for the Features of Forms of Integration and Card Payment

Two features of the ETC-SPL are discussed with the objective of illustrating how design decisions are influenced by the decisions taken related to the SPL development process adopted, to the type of component, and to the manner of composition (manual or automated). One feature (*Form of Integration*) uses new classes to model and implement its feature and another feature (*Card Payment*) uses subclasses (with new attributes and methods) to do so. For simplicity and space reasons, the models of classes that are illustrated show only the attributes.

5.1 Design of features related to "Forms of Integration"

We initially will consider the optional feature *Transport Hub* exclusive to the cities of Fortaleza and Campo Grande, which are considered in version 2 of the ETC-SPL. Figure 4 shows part of the features diagram related to this feature. The ETC system of Fortaleza has only bus transport hubs as a form of trips integration. The transport hubs work as a special terminus where passengers can change buses without paying another fare. Other more sophisticated ways of integration occur in the ETC systems of the other two cities, corresponding to other variabilities of the *Form of Integration* feature group.

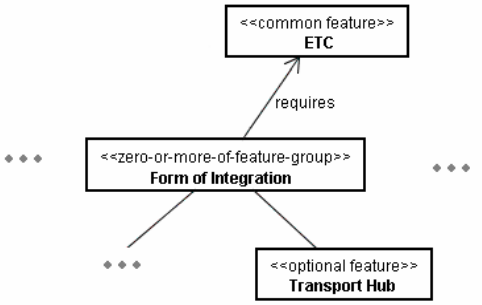


Fig. 4 – Part of the features diagram related to the *Transport Hub* feature

Figure 5 shows the model of classes used to implement the operations related to the routes of the bus company. The classes *Route*, *Run*, *TransportMode* and *Bus* (represented with the stereotype[*] <<*kernel*>> in the figure) are wrapped in a kernel component called *RouteMgr*, represented previously in Figure 3. The design of the feature *Transport Hub* requires the addition of a *TransportHub* class to the model. Generally, the inclusion of new features to the SPL design implies adding and/or modifying classes, operations and attributes. In the same way, the components may need adaptations or compositions such that variabilities reflect on the components' architecture. There are many ways of treating these changes, each having advantages and disadvantages that reflect on the decisions taken for the SPL's design.

One way of treating the inclusion of operations and attributes inside existing classes and the inclusion of new classes is to add them directly inside their components and change operations according to new needs. For the given example, there should be two components, the kernel component *RouteMgr* and the alternative equivalent

[*] Gomaa (2004) sometimes uses more than one stereotype to classify more specifically elements of a SPL.

component, which could be called *RouteTransportHubMgr*. They would be used respectively for the kernel application and the application of Fortaleza.

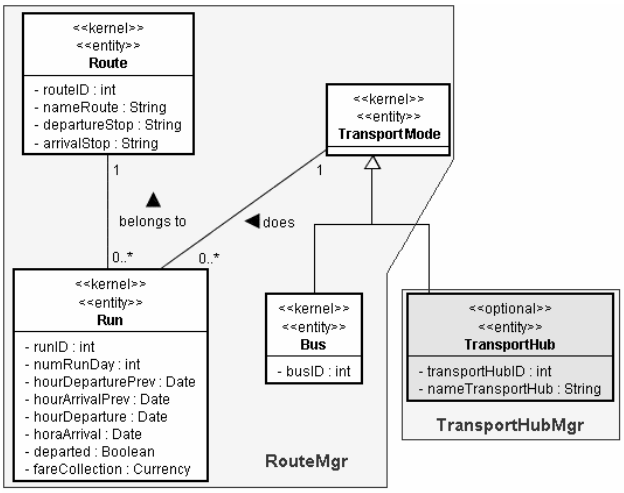


Fig. 5 – Fragment of the class model related to the *Transport Hub* feature

This solution's advantage is its facility of implementation and composition. There are, however, many disadvantages. There can be future problems with maintainability because the solution tends to duplicate code and a future modification can demand the update of both alternative components. Besides, the original component, *RouteMgr*, has to be a white-box because its adaptation requires the knowledge and access of its internal structure.

To include classes, operations and attributes without having internal access to the implementation of previously developed components, corresponding to the SPL's assets, it is necessary to design black-boxes. Therefore, to design the ETC-SPL, we preferred to use these kind of components and, in this way, operations and attributes that would be added to existing classes are separated into new classes so that new variability specific components are created. These components can then be assembled with components already designed to create new ones satisfying the new requirements. The components differ because they implement distinct variabilities, however they can be formed by kernel components that are reused. The way in which they are connected and the required implementation to join these different components can also be changed. The disadvantage of this solution is a greater communication between components, which can decrease efficiency.

Consequently, instead of including the class *TransportHub* inside the component *RouteMgr*, a new component is created for the class with its attributes and operations, called *TransportHubMgr* and the component *RouteMgr* is reused without any alteration. The use of these components is managed by another component (*Controller RouteTransportHubMgr*) and the three components are then wrapped in a composed component called *RouteTransportHubMgr*. These components' architecture details are shown in Figure 6. The interfaces are not altered and the components requiring them do not need any modifications. The interface *IRouteMgt* is required by the components

Control Trip, *Control Run* and *ManagementControl*, the interface *IUpdateRun* is required by *Control Run* and *ICollectFare* is required by *Control Trip*.

In the Campo Grande version, besides the *Transport Hub* feature, there is also the feature *Integration*. The integration can be done using defined integration routes, provided that the trip remains inside a specified time limit. There is also a maximum number of integration trips within the time interval allowed. These features can be seen in Figure 7.

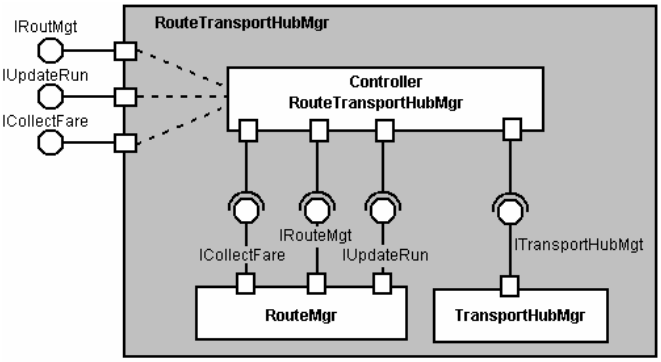


Fig. 6 – Composed component *RouteTransportHubMgr*

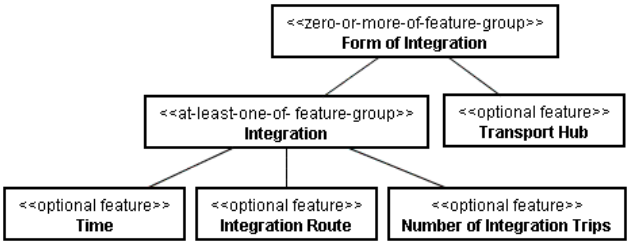


Fig. 7 – Part of the features diagram related to the "*Form of Integration*" feature

In the model of classes, the feature *Integration Route* is represented by an optional class called *IntegratedRoute* and is associated to the *Route* class, shown in Figure 8. Only this specific feature will be shown as an example, because the other features are found in different parts of the model and are not related to the *RouteMgr* component. Since Campo Grande also uses transport hubs, the component *RouteTransportHubMgr* can be reused for this version and, so that the components' integrity is maintained and to keep them as black-boxes, the optional *IntegratedRoute* class is integrated in a new component called *IntegratedRouteMgr*. Similarly as in the previous example, a controller component is created and the three components are then internal parts of the composed component *RouteTransportHubIntegrationMgr* seen in Figure 9.

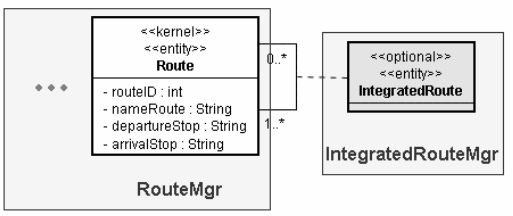


Fig. 8 – The feature "*Integration Route*" in the class model

This component has another interface as the corresponding components of the earlier versions do not have the *IVerifyIntegratedRoute* interface. With integration this interface is required by the *Control Trip* component which consequently also has to be altered to treat the provided result according to the business rules of the integration feature. The solution can be to use composition, designing a new component or to separate the additional interest in an aspect [Kiczales, 1996; Suvée et al, 2006] so that the component does not need to be replaced and there can be an enhancement in the process of the variabilities' composition of the SPL [Heo and Choi, 2006]. The other interfaces remain the same as in the Fortaleza version.

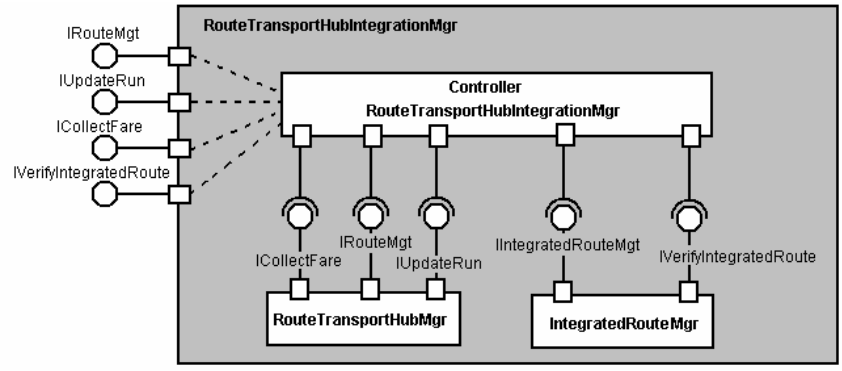


Fig. 9 – Composed component *RouteTransportHubIntegrationMgr*

In the São Carlos version, the feature *Integration Route* also exists but does not have the *Transport Hub* feature. Therefore the components related to the transport hub cannot be reused and a new composition is needed. For this version the components that can be reused are *RouteMgr*, developed in version 1 (kernel), and *IntegratedRouteMgr*, developed in version 3 (Campo Grande). A new controller is necessary to compose these components and form the composed component *RouteIntegrationMgr*, whose architecture can be seen in Figure 10.

5.2 Design of features related to "Card Payment"

In the ETC system of Fortaleza, some passenger types have to purchase the bus card. This feature does not exist in the other two cities and is designed in the iteration of version 2, not reflecting in other iterations. This feature is shown in the partial features diagram of Figure 11. The cards can be of different categories according to the type of passenger and information about passenger trips may be stored. A card can also have various associated payments related to charges made for the card. When the feature

Card Payment is present, the payment can also refer to the purchase of a card. These requirements lead to the classes' model presented in Figure 12.

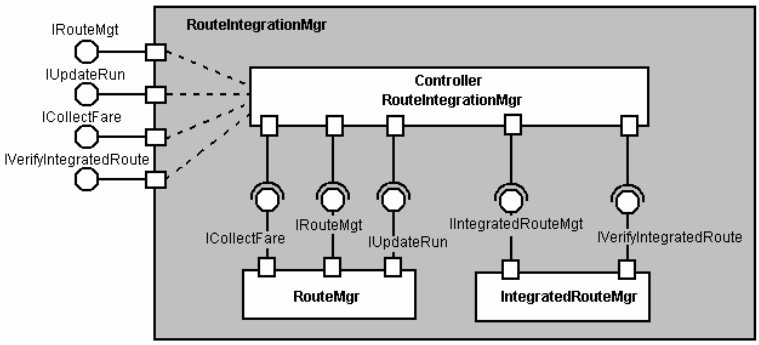


Fig. 10 – Composed component *RouteIntegrationMgr*

Figure 12 shows the classes *Card*, *PassengerType*, *Trip* and *Payment* that are part of the ETC-SPL and are encapsulated in the *CardMgr* component. The *Card Payment* feature implies variation points in the classes *PassengerType* and *Payment*, altering attributes and operations of these classes, different to the previous example, in which it was necessary to insert a new class into the model. One option is to use parameterized classes, but this option was not adopted to keep interests separated [Gomaa, 2004] so as to maintain the components as black-boxes. We chose then to use classes with variation points and separate the *Card Payment* feature in a new component called *PaymentMgr* with the *IPaymentMgt* as a provided interface. The specification of this interface is shown in Figure 13. Both classes are inserted in one component because they have the same interest (*Card Payment*) and are always used together. If it was important to differentiate them, two interfaces could be provided by the component, separating the methods implemented by different classes.

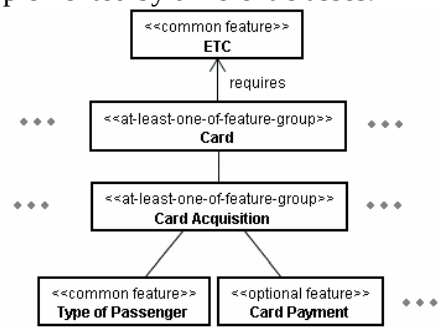


Fig. 11 – Part of the features diagram related to the *Card Payment* feature

The components *CardMgr* and *PaymentMgr* are managed by a controller so that the information of the passenger type and the payment remain separated in different components and in different tables in a relational data base and can be treated and joined if needed. Their composed component is called *CardPaymentMgr* and its architecture is shown in Figure 14. The interfaces of the composed component are not changed and therefore it is not necessary to change the components that require the interfaces of this component, for example the *Control Trip* component.

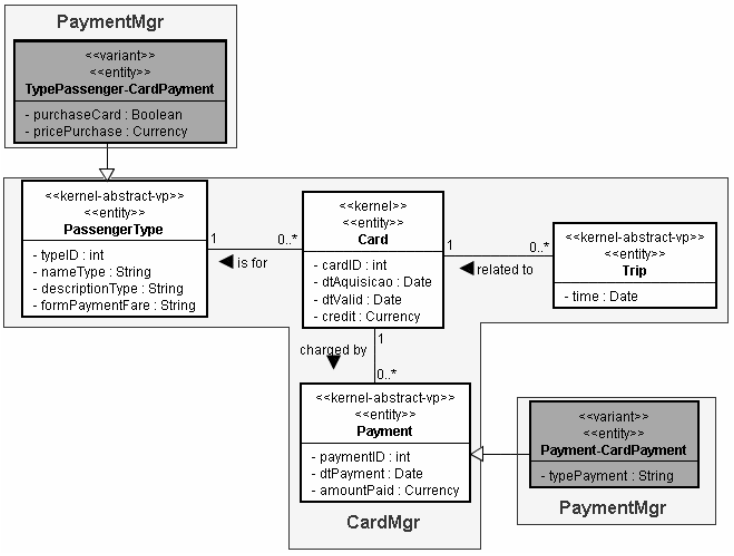


Fig. 12 – Part of the classes' model related to the "*Card Payment*" feature

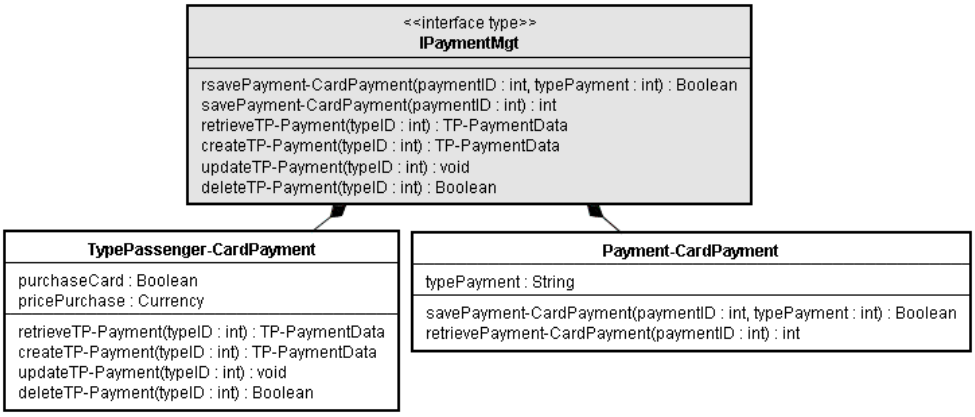


Fig. 13 – Specification of the *IPaymentMgt* interface and related classes

Independently of this composed component, the feature *Card Payment* needs a modification in the *Card Acquisition* component so that it records the reception of a payment in the financial system by requiring the *IFinancialSystemMgr* interface from the *Financial System* component. This way the *Acquisition Card* component is substituted by a composed component that is specific for the *Card Payment* feature.

This example also shows how a white-box solution would be easily created by a generator by maintaining the component's name and its interfaces and, when needed, changing its composition by adding subclasses according to options chosen on the generator tool. The solution presented requires more communication between classes that implement the component's interfaces and can be more inefficient, but even so we chose this solution so that black-box components could be used. The controllers correspond to generated glue code to connect the lower level assets.

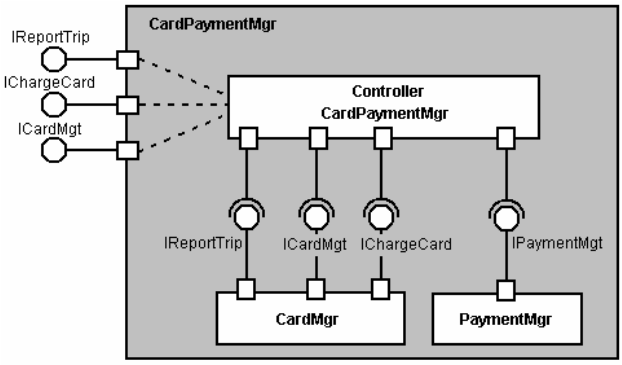


Fig. 14 – Composed component *CardPaymentMgr*

6. Using a Code Generator

An automated process is planned to be implemented in iteration 5 to generate applications for this SPL. It is relatively easy to see that the list of features shown in Section 4 is an initial sketch of the Application's Modeling Language (AML) (Weiss and Lai, 1999) for the ETC domain and based on it an automated application generator can be created. We intend to use the configurable code generator Captor developed by our research group [Schimabukuro et al, 2006].

For the solution presented here, based on black-box components, the generator code will act like a "configurator", starting from the kernel architecture, replacing and including the necessary black-box components from the library of core assets and generating glue code for each pair of components being composed. Note that if the automation had been done before iteration 5, each new horizontal version designed and implemented would need considerable rework in the generator.

Another solution that could be used for this case, considering white-box components, would be to make the generator perform the changes inside each component thereby generating additional classes and modifying other elements inside the components according to the need of each application. The generator in this case would be much more complex and act as a composer, according to the definition of Weiss and Lai (1999). Thus, less core assets would be needed. Both solutions are acceptable, however, but depend on the previous decision of using black-box or white-box components. A combination of both is also possible.

The choice of automating the composition process influences the design as well as the moment of introducing the automation in the SPL. If automation is used to generate the products from the first version, this decision influences the design of the new versions of the SPL. Also, for each new horizontal iteration, considerable rework in the application generator would be needed.

7. Final Considerations

In the current state of development of the ETC-SPL the design of the kernel and of version 2 (Fortaleza) have already been made. Some other features have also been designed vertically with the intention of investigating different solutions to those shown

in this paper. The implementation of the kernel is ongoing, aiming to create the SPL's assets.

A lesson learned so far from the development of the ETC-SPL is that having decided to evolve the line in horizontal iterations, it is very important that some time be taken to analyse how feature groups will evolve in the following iterations before committing to a design that cannot be easily modified or reused in the next versions. The example discussed in this paper for the route integration feature has shown the trade-offs between horizontal and vertical development. It also showed that the decision of using black box or white box components is crucial. In general, it is better to design a black box component and compositions of them, as it is always possible later, if an off-the-shelf component is not found, that implements the required interface, to design one that performs the required function.

References

Atkinson, C; Bayer, J.; Muthig, D. (2000) Component-Based Product Line Development: The KobrA Approach. *1st Software Product Line Conference*, 19p.

Atkinson, C; Muthig, D. (2002) Enhancing Component Reusability through Product Line Technology. *Proceedings of the 7th International Conference in Software Reuse (ICSR02)*, Springer Press, p. 93-108.

Bachmann, F.; Goedicke, M.; Leite, J.; Nord, R.; Pohl, K.; Ramesh, B.; Vilbig, A. (2004) A Meta-model for Representing Variability in Product Family Development. *Proceedings of the 5th International Workshop on Software Product-Family Engineering*, Springer, p. 66-80.

Becker, M. (2003) Towards a General Model of Variability in Product Families. *Proceedings of the 1st Workshop on Software Variability Management*, Groningen, 9p.

Bosch, J. (2000) Design et Use of Software Architectures: adopting and evolving a product-line approach, Addison-Wesley, 354p.

Bosch, J.; Florijn, G.; Greefhorst, D.; Kuusela, J.; Obbink, H.; Pohl, K. (2001) Variability Issues in Software Product Lines. *Proceedings of the 4th International Workshop on Product Family Engineering*, Springer, p. 11-19.

Cheesman, J.; Daniels, J. (2001) *UML Components: a simple process for specifying component-based software*, Addison-Wesley, 176p.

Clements, P.; Northrop, L. (2002) *Software Product Lines: Practices and Patterns*, Addison-Wesley, 563p.

Gomaa, H. (2004) *Designing Software Product Lines with UML*. Addison-Wesley, 701p.

Heo, S-H.; Choi, E. M. (2006) Representation of Variability in Software Product Line Using Aspect-Oriented Programming. *Proceedings of the 4th International Conference on Software Engineering Research Management and Applications (SERA)*, 8p.

Junior, E. A. O.; Gimenes, I. M. S.; Huzita, E. H. M.; Maldonado, J. C. (2005) A Variability Management Process for Software Product Lines. *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, Cascon, p. 225-241.

Kiczales, G. (1996) Aspect-Oriented Programming. *ACM Computing Surveys (CSUR)*, v.28, n. 4es, p. 220-242.

Roberts, D; Johnson, R. (1998) Evolving Frameworks: a pattern language for developing object-oriented frameworks. In: Martin, R.C.; Riehle, D.; Buschman, F. *Pattern Languages of Program Design 3*, Addison-Wesley, p. 471-486.

Schimabukuro, E. K. J.; Masiero, P. C.; Braga, R. T. V. (2006) Captor: A Configurable Application Generator (in Portuguese). *XIII Tools Session of the Brazilian Symposium of Software Engineering*, 6p.

Sugumaran,V.; Park, S.; Kang, K.C. (2006) Software Product Line Engineering. *Communications of the ACM*, Vol 49, No. 12, p. 29-32.

Suvée, D.; Fraine, B. D.; Vanderperren, W. (2006) A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development. In: *Component Based Software Engineering*, p. 114-122.

Weiss, D. M.; Lai, C. R. R. (1999) *Software product-line engineering: a family-based software development process*. Addison-Wesley, 426p.

AIPLE-IS: An Approach to Develop Product Lines for Information Systems Using Aspects

Rosana T. Vaccare Braga[1], Fernão S. Rodrigues Germano[1], Stanley F. Pacios[1], Paulo C. Masiero[1]

[1]Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo
Department of Computing Systems
Caixa Postal 668 – 13560-970 – São Carlos – SP – Brazil

rtvb@icmc.usp.br, fernao@icmc.usp.br, stanley.pacios@gmail.com, masiero@icmc.usp.br

Abstract. *Product lines for information systems present variabilities both in non-functional and functional features. Aspects are being used successfully in the implementation of non-functional features, as they provide intuitive units for isolating the requirements associated to this type of features. However, aspects could also be used to implement some product line features that refer to functional requirements. Using that approach, the instantiation of specific products could be done by combining the desired aspects into the final product. In this paper, we propose an approach, named AIPLE-IS, to incrementally build a product line for information systems using aspects. The product line core is developed first, followed by the addition of optional features through aspects. A case study for a product line in the domain of information systems for psychology clinics is presented to illustrate the approach.*

1. Introduction

Object-oriented Programming (OOP) is an established programming paradigm, with well defined development processes, e.g. the Unified Process (Jacobson et al 99). On the other hand, Aspect-Oriented Programming (AOP) (Kiczales et al, 97; Elrad et al, 01) is a relatively new programming technique that has arisen to complement OOP, so the software community is still exploring it and evaluating its costs and benefits.

Research about concepts and languages for aspect orientation (AO) has already attained a mature stage. However, processes for AO are still topics under study (Baniassad et al, 06). Recent works by several authors (Pearce and Noble, 06; Griswold et al, 06; Apel et al, 06) have contributed to solve specific problems of different development phases. In particular, research focused on dealing with aspects in the early development stages of requirements engineering and architecture design are gaining more focus in the last few years (Baniassad et al., 2006).

The need for techniques that help design and develop better quality software in less time is one of the software engineering concerns. Many software products are developed for artifacts already specified and implemented using software reuse techniques. In this context, the software product line (SPL) approach appears as a proposal for software construction and reuse based on a specific domain (Bosch, 00).

This technique has already shown its value on OO development, and can both benefit AO development and benefit from it.

In this paper product line engineering is considered as the development of software products based on a core architecture, which contains artifacts that are common to all products, together with specific components that represent variable aspects of particular products. Product line commonalities and variabilities can be represented as system features (Kang et al, 90) and they can be related both to functional or non-functional software requirements. Thus, it is interesting to investigate how aspects can improve modularization of SPL parts, isolating interests and benefiting SPLs, allowing the creation of more pluggable and interchangeable features.

In this paper, we propose and approach for incrementally developing an SPL, in which aspects are used in a systematic way to ease the introduction of functional features in the SPL, without changing the remaining features. The approach has been created based on a concrete product line development, which refers to a psychology clinic control system. In brief, the motivation for developing this work is the need for processes and techniques for aspect-oriented analysis and design; the growing interest of the software community in early aspects; and the need for approaches to develop aspect-oriented product lines.

The remaining of this paper is organized in the following way. Section 2 gives an overview of the proposed approach, named AIPLE-IS. Section 3 presents the SPL core development in more details, while Section 4 describes the product creation phase. A case study to illustrate the approach is presented along sections 3 and 4. Section 5 discusses related work. Finally, Section 6 presents the conclusions and ongoing work.

2. Overview of the proposed approach

Our approach for Aspect-based Incremental Product Line Engineering for Information Systems (AIPLE-IS) is illustrated in Figure 1. It has two main phases: Core Development and Product Creation. The Unified Modeling Language –UML (Rational, 00) is used as a modeling notation, combined with artifacts from the Theme/Doc approach notation (Clarke et al, 05). In the first phase (*Core Development*), a domain analysis is done to identify both fixed and variant points of the domain. The fixed part is implemented in this phase and is here denoted as the SPL core assets, because they define the minimum features that a single product of the family will have. These core assets are implemented using aspects where necessary to ease the future inclusion of variant features in the subsequent phase, as explained in Section 3.

In the second phase (*Product Creation*) several iterations occur to develop specific features needed to produce SPL concrete products. Each increment will result in a set of features needed to obtain a particular product, but that can also be reused in other products. Aspect-oriented techniques are used whenever possible to isolate features into aspects. Products are obtained by composing aspects and base code according to specific requirements. This activity can be executed as soon as the core assets are implemented, as there may be products that consist only of basic functionalities, or it can be executed later by combining basic and optional features.

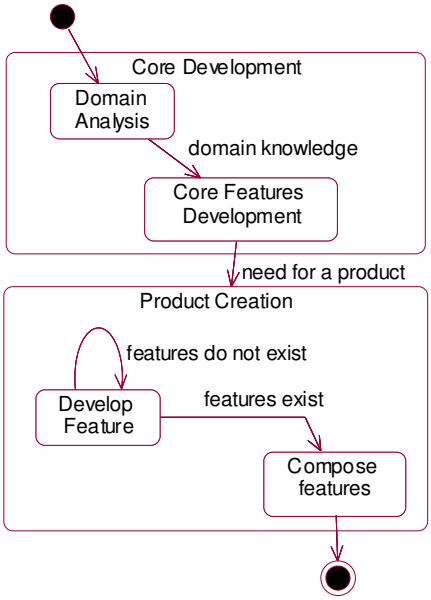


Figure 1. AIPLE-IS overview

3. Core Development

This phase aims at identifying and implementing the SPL core assets. It has two activities, as shown in Figure 1: domain analysis and core features development.

3.1. Domain Analysis

The domain analysis is conducted to capture the domain knowledge, i.e., to identify the functionality present in different applications of the same domain. This activity is extensive and, thus, is out of the scope of this paper to describe it in detail, as any existing domain analysis method could be used, such as those by Prieto-Diaz (1990) or Kang et al (1990). Gomaa (2004) also presents an approach to domain analysis that contains most of the good principles used by these authors, but his process is updated according to more recent notations.

The domain knowledge obtained in this phase should be properly documented to identify the SPL features, which can be mandatory, optional, or alternative. Mandatory features are those that should be present in all SPL members. Optional features can be present in one or more SPL members. Alternative features form a small set of features from which one or more are chosen to be part of an SPL member (exclusive alternative is also possible). The features model notation (Kang et al, 90) is used and a number is added to each feature to ease its future reference in subsequent phases. During domain analysis, it is important to discover mainly the mandatory features, and also those that are more likely to be needed later. More uncommon features are searched secondarily.

The domain analysis phase is outlined in Figure 2 (using BPMI notation (Arkin, 2002)), which shows its activities and artifacts produced. As it can be observed in the figure, the process starts with the study of one or more systems in the domain aiming at

creating, for each of them, a Features Document, a Requirements Document, a Features Model, a Conceptual Model and a Feature-Requirements Mapping. Those are named "individual" versions, i.e., each of them represents a single system. Other domain study activities can be used to help in the creation of these documents, as well as to eventually help in the domain analysis or even in AIPLE-IS subsequent phases.

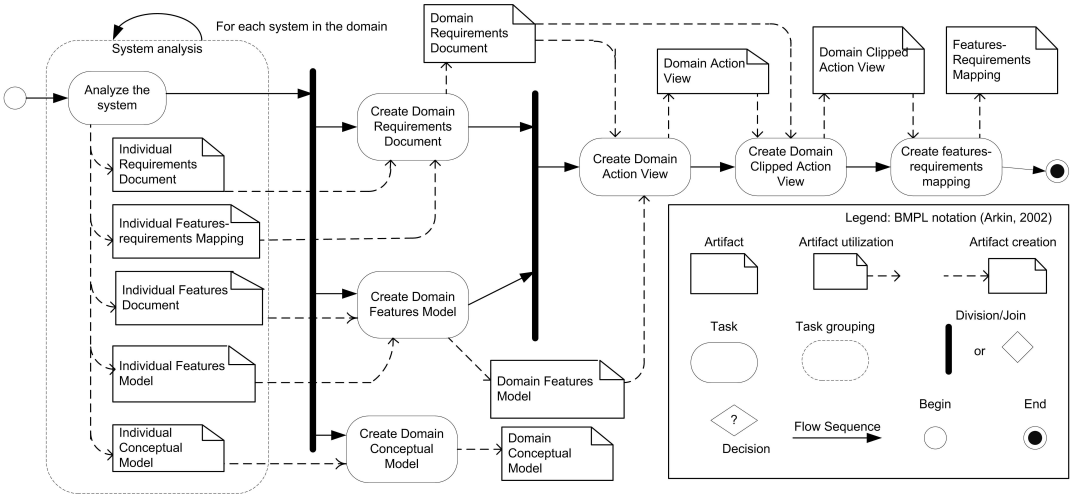


Figure 2. AIPLE-IS - Domain Analysis activities

In order to complete the domain analysis, a final version of each artifact is produced, named "domain" version, so that these domain versions encompass and organize all the content of the individual versions. Thus, a Domain Features Model is produced based on individual features documents and models, a Domain Conceptual Model is produced based on individual conceptual models, a Domain Requirements Document is produced based on individual requirements documents, and a Features-Requirements mapping is created based on individual features-requirements mappings. The other documents presented in Figure 2, as for example the action view and the clipped action view, are optionally created to help clarifying to which feature a requirement belongs to. They are based on the Theme approach notation proposed by Clarke (2005). The difference is that Clarke uses them to represent individual systems, while here they are used to represent the entire domain. In this phase we are not worried about which features are crosscutting or not.

To illustrate the usage of AIPLE-IS, we introduce an example implemented as part of a master thesis at ICMC-USP, where AIPLE-IS was used to develop part of a psychology clinic control system product line, here simply referred to as "PSI-PL". The possible products instantiated for this SPL are systems for managing several similar but different psychologist offices, psychologist hospitals, and other similar institutions. The PSI-PL domain analysis has been conducted based on the reverse engineering of three systems: the first is a private small psychologist office, and the other two are different hospitals (one private and one public) dedicated to attend psychology patients. The reverse engineering produced several individual models that were then used as basis to produce the domain model. A small part of the PSI-PL domain conceptual model and of the features model are shown in Figure 3 and Figure 4, respectively. These models illustrate several domain concepts, which can be mandatory or not (in the features model

of Figure 4, features with a filled circle are the mandatory ones, while those with a hallow circle are optional features). Figure 5 illustrates a small piece of the Domain Requirements Document (requirements were simplified to be presented here). Other artifacts obtained, such as the Domain Features Document, and the Features-Requirements Mapping are not shown here due to space restrictions.

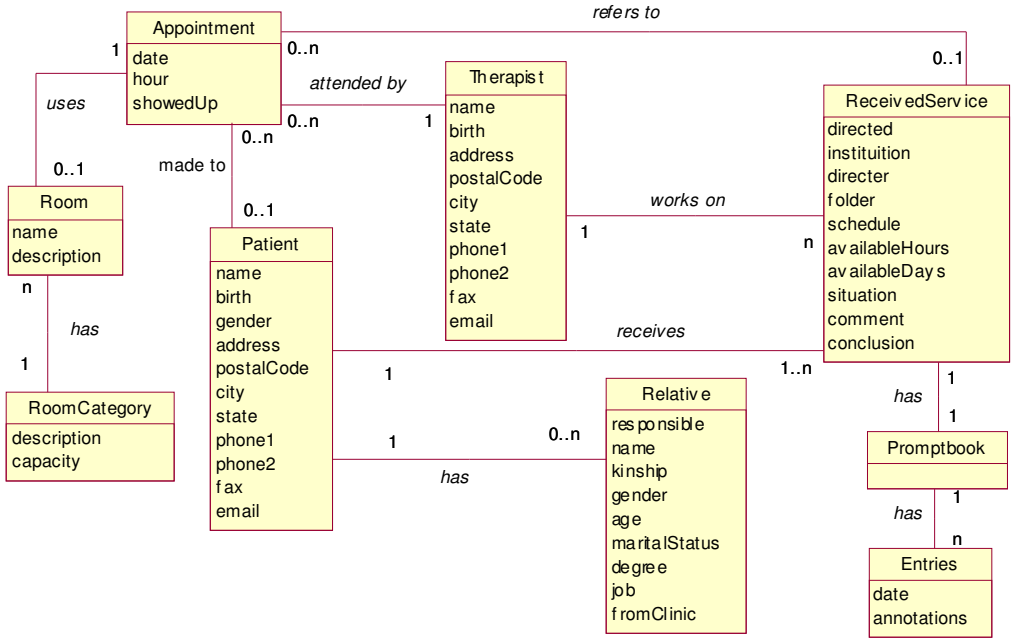


Figure 3. Partial Domain Model

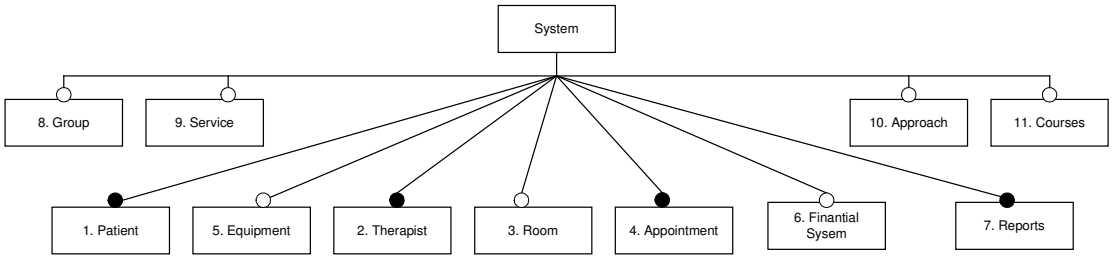


Figure 4. Partial Features Model

1 – The system should allow the inclusion, search, modification, and removal of patients from the clinic. Patients have the following attributes: name, birth date, address, zip code, state, phone, e-mail, identification document number.
7 – The system should allow the inclusion, search, modification, and removal of information about the service that the patient is receiving at the clinic, with the following data: therapist name, type of service, patient name, available dates/times, diagnosis, …
24 – The system should allow the inclusion, search, modification, and removal of appointments, containing the following data: patient name, therapist name, room, day/time scheduled, and service to be performed.
42 – The system should allow the management of information about the possible types of service offered by the clinic.

Figure 5. Example of three domain requirements

3.2. Core Features development

The core features development aims at implementing all SPL common features. It includes activities such as defining the SPL core architecture, designing the software, implementing, and testing it. This is an extensive activity that presents many of the issues of developing a conventional system and, additionally, some more specific issues that arise due to the fact that we are developing a product line using aspects. Thus, we recommend the use of an object-oriented programming language that has an associated extension to support aspect-oriented characteristics. In the PSI-PL case study, Java and its aspect extension, AspectJ, have been used (AspectJ, 2006). MySQL relational database was used for objects persistence.

The definition of the SPL architecture depends on several factors related to the particular project issues, such as non-functional requirements that can influence on performance, flexibility, etc. For example, the architecture could be based on an object-oriented framework, on a set of components, or simply on a set of objects. In a lower level, it should be decided whether to use a layered-architecture, for example. These decisions involve the particular interests of the organization, so we consider this phase as being out of the scope of this paper. The PSI-PL architecture followed the MVC pattern (Buschmann et al., 1996).

After defining the architecture, the mandatory features are analyzed, designed, and implemented. The features model (produced in the previous phase) is a source for identifying the mandatory features. For example, in the PSI-PL case study, domain engineers have determined that the product line core base should consist of features Patient, Appointment, and Therapist. This is the minimum functionality expected from a member of the product line, probably used in small psychologist offices. AOP is used in this phase to provide a mechanism through which optional features are more easily introduced in the subsequent phase.

Even intending to isolate features, in certain moments they influence one another, as the final system expected behavior contains the interaction among features. The development of the features that influence other features is easier if the features that will be influenced are already designed and implemented. If they are not, the design of the influence is postponed until they appear in the design. So, a practical advice is to create first the features that are more independent of others. For example, in the PSI-PL case, requirement #1 of Figure 5 describes the Patient feature and it is easy to see that it is independent of other features, so it should be created first. The same is true for feature Therapist. On the other hand, as can be seen on requirement #24 of Figure 5, Appointment depends on both Patient and Therapist, so its creation should be delayed. To identify the influence among features, the clipped action view diagram developed in the analysis phase can be used, as exemplified in Section 4.1.

To ease the isolation of features and the identification of their dependencies, AIPLE-IS suggests the development in three steps involving analysis and design, as can be seen in Figure 6. Each step produces a part of the feature design. The first step creates the design part that deals with the feature interest more strictly and exclusively as possible, i.e., free of other features influence. To make this possible, the requirements associated to the feature are rewritten to withdraw any behaviors that might refer to other features. Then, the analysis and design proceed, creating use cases, class diagrams,

collaboration diagrams, etc., similarly to conventional OO analysis and design. Information about the feature is obtained from the artifacts resulting from domain analysis phase. The numbering present in the features model is used, together with the features-requirements mapping, to find the corresponding detailed requirements.

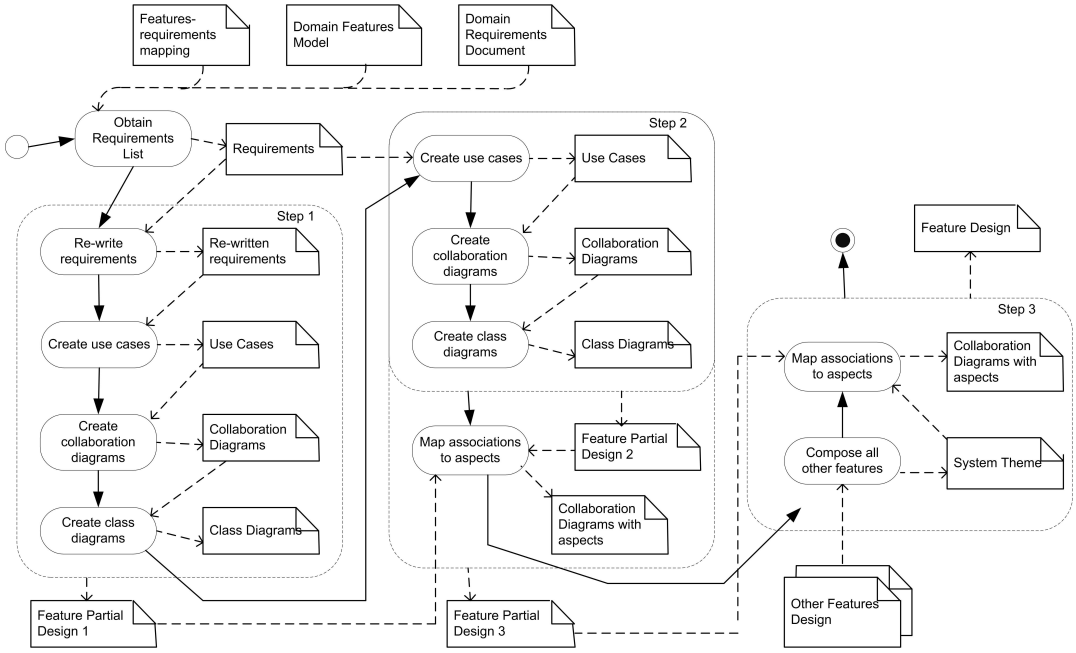


Figure 6. AIPLE-IS – Features Development activities

As an example, consider the design of the feature relative to requirement #24. It could be rewritten so that only Appointment is mentioned, i.e., citations to Patient and Therapist are withdrawn. The result is the requirement "The system should allow the inclusion, search, modification, and removal of appointments, containing the following data: room, day/time scheduled, and service to be performed". This requirement is used as basis for creating use cases, class diagrams, etc.

The second step creates the feature design part that exists due to the presence of other features. Here, the requirements are reviewed to consider those that have interests tangled and scattered in the requirements document. A refined design is created for the same feature, now considering the presence of other features. It is at this point that AOP begins to act. Once the elements to be added by other features are known, they are designed separately, allowing a modular implementation using AOP. Special annotations are done in the class diagrams and collaboration diagrams to denote the presence of aspects. We do not show this here due to space limitations, but any extension of UML to deal with aspects can be used. In the PSI-PL example, aspects are used in this step to create the associations between Appointment and Patient and between Appointment and Therapist. This aims at easing possible subsequent changes in associations, as explained in Section 4.1 and is according to the idea of relationship aspects introduced by Pearce and Nobel (2006).

The third step creates the feature design part that exists due to the new feature influence in the rest of the system. In order to find out exactly which influence the

feature causes in the rest of the system, it is required to have the design of the rest of the system. This is obtained using the composition of base themes (also according to the Theme approach). All existing features of the SPL are composed together to obtain this design, which is named as the "System" theme. Having the design of the System theme, it is possible to identify how the feature under development will be composed with the rest of the system. In the PSI-PL example, as we are beginning the development, we do not have a System yet, so this step is skipped. If we had a system, we would have to check how Appointment influences the rest of the system.

There are some guidelines that have to be followed to implement the features. They are summarized in Section 4.1, as they are common both to mandatory and optional features, and can be found elsewhere in more detail (Pacios et al, 06).

4. Product Creation

This phase aims at creating the concrete products and increasing the features repository. It has two activities, as shown in Figure 1: develop feature and compose features. The products are created on demand, and optional features are developed only when requested by a specific product.

4.1. Develop Feature

This activity is responsible for incrementally adding new features to the product line, using aspect techniques when appropriate, until all variable features identified in the domain analysis are developed. In fact, this phase can be extended as needed to add new features identified after the domain analysis, as part of the product line evolution. To implement a particular feature, the following general guidelines have been proposed (Pacios et al, 06):

- *G1 - New classes*: if a feature implies in the creation of one or more new classes, these should be implemented as conventional classes (with no need to use AOP);
- *G2 - New attributes and/or methods*: if the feature implies in the creation of new attributes or methods in existing classes, they could be introduced into the existing classes through intertype declarations, but other mechanisms could be used, for example, the properties pattern (Yoder et al., 2001);
- *G3 - Change in the behavior of existing methods*: if the feature existence implies in the modification of existing methods, this is done with aspects and advices;
- *G4 - New association*: if the feature implies in creating new associations between existing classes, or between a new class and an existing one, they are implemented with aspects, to guarantee the connectivity with the feature and its removal if necessary. N to 1 associations are generally implemented through an attribute included in one of the classes (the N side) to represent the reference to the other class (the 1 side). So, guideline G2 is applicable here.
- *G5 - Removal of existing associations*: if the presence of one feature requires removing one or more associations between existing classes (probably to add other different associations), then a mechanism is needed to remove them. To make that possible, the existing associations should have been included through aspects, so that just omitting the aspect that included it, is enough to remove the association.

As an example, consider the PSI-PL again. After the core features were implemented, it was decided to include the "Room" feature, which consists of allowing the control of rooms where appointments occur. This was an optional feature (see Figure 4), and it implied in the creation of two new classes, Room and RoomCategory. Room is associated to an existing class, Appointment. The implementation of this new feature was quite simple to execute using aspects. Following guideline G1, a new class, Room, was created, together with an aspect to introduce the new attribute (roomNumber) in the Appointment class to represent the association between Appointment and Room (G4).

More specific guidelines that need to be observed during the features implementation are summarized next. They are more suitable for the development of information systems, considering that in this work the three-tier architecture has been chosen (with an interface layer, a business application layer, and a persistence layer), and persistence is done using relational databases.

For each existing class of the business application layer that receives new attributes or methods, an aspect is created to: introduce new attributes and respective methods; supply the initial value for the new attributes; guarantee that the class methods that handle its attributes also handle the new attributes; and treat possible business rules associated to these new attributes.

To ease the introduction of new attributes and their treatment, meta-attributes can be used: one named "fields" and another named "values" (these can be vectors whose elements are strings with the fields and values names, respectively). The use of meta-attributes makes it possible for the aspects to introduce their new attributes in the corresponding meta-attribute, avoiding having to create an advice or intertype declaration to include new attributes. Functions that receive all object attributes by parameters, or that return all these attributes, are modified to receive and return objects of vector type. A particularly common case of functions of these types are the database query functions. On the other hand, by using meta-attributes the advantage of static variable checking is lost. Other possible solutions would be to use the Java language reflection or active object models (Yoder et al., 2001).

This same guideline can be applied to include associations between classes. The association is represented by a reference from one class to the other. So, a field can be added in both vectors to deal with the referential attribute. The additional methods necessary to handle the new attributes are included through intertype declarations.

The interface layer has to reflect the modifications that occur in the application business classes that they represent. In the particular case of information systems, most application classes have a corresponding graphical user interface (GUI) class, which might need new widgets placed on them due to the inclusion of new attributes. So, a mechanism to introduce these widgets in GUI classes is needed. A possible solution is to divide the construction of the GUI screen in several parts, so that it is easy to identify pointcuts where to place the new widgets and the respective treatment. For example, the GUI creation method should have at least four parts: create variables, initialize variables, position the widgets on the screen, and treat events related to the widgets. Thus, an aspect can be created for each GUI class, and advices can be used to introduce the treatment of the new attributes in each method.

Returning to the PSI-PL example, in terms of its GUI, after introducing the new "Room" feature, it is necessary to include an additional widget so that the final user can choose the room where the appointment is scheduled. This can also be done with an aspect, which adds this widget and corresponding behavior to the GUI class responsible for entering the appointment data.

So, this first evolution of the PSI-PL produced an increment that allows the creation of two different products: simple psychology office and office with room's allocation. The second iteration to evolve the PL considered that a patient can be scheduled not only to one therapist, but to a service that is performed by a therapist. This implies that one patient can be registered in several different services offered by a hospital, each of which is attended by a different therapist. For example, he or she can participate in a career advice therapy and in a couple therapy, so that different appointments are made for them. Service is an optional feature of PSI-PL (see Figure 4).

To design this feature, initially its requirements are rewritten to withdraw any behavior that do not belong to Service itself, as for example requirement #7 of Figure 5 is re-written as "The system should allow the inclusion, search, modification, and removal of information about the service received at the clinic, with the following data: type of service, available dates/times, diagnosis, …". This is enough to develop a complete design for the Service feature itself, without the influence of other features (Patient and Therapist in this case). Then, the artifacts Action View and Clipped Action View, obtained in the domain analysis, are used to help visualizing which part of the functionality results from the influence of other features. Figure 7 (a) shows the Action View corresponding to the Service feature, described in requirements 7 and 42, but also mentioned in requirement 24 (which deals with Appointment). It can be observed that features Patient and Therapist affect the Service feature directly through requirement 7.

When the features-requirements mapping was built, it was decided that the Service feature is dominant in requirement 7, so Patient and Therapist features will affect the Service feature. This decision is reflected in the clipped action view of Figure 7 (b). In requirement 24, the dominant feature is Appointment, as service is just one more detail in its main goal, which is to make an appointment. The creation of the clipped action view is the right moment to review decisions related to features-requirements relationships. The clipped action view indicates that Service will influence Appointment.

Finally, to finish the Service feature design, a comparison is done with the rest of the system to detect any intersections. In this case, this intersection is empty, as all classes are new and thus should be implemented simply using OO classes.

Regarding the organization of the product line code, to improve reuse of the features separately, code artifacts (such as classes and aspects) that refer to one feature should be put together in one or more packages. New classes can be placed in a separate package, and a package could be created for each new association among different features. That way, it is easier to reuse the new classes or just the associations. A features-packages mapping can be created to ease the subsequent features composition.

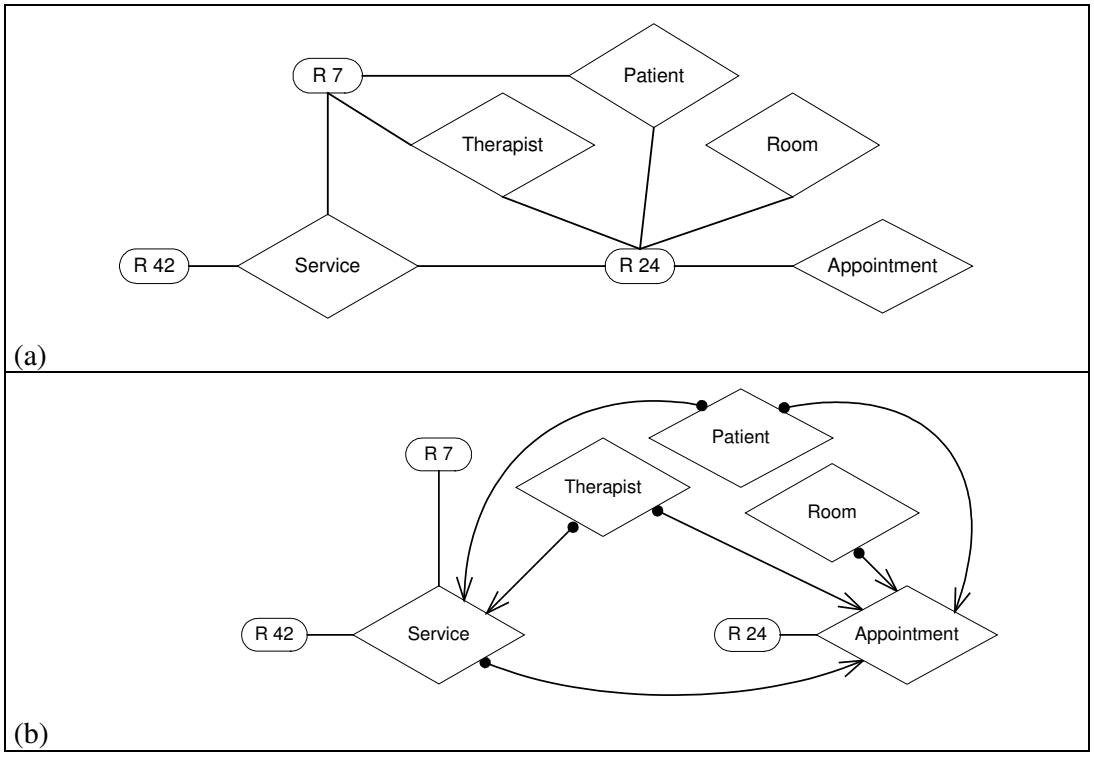


Figure 7. Action View (a) and Clipped Action View (b) for Service Feature

4.2. Compose Features

In this phase, concrete products are instantiated from the product line. The software engineer needs to choose the variabilities of a particular product and then use composition rules to join the base code with the code of each optional feature implemented in the product line. As aspects are used in the product line development, a special tool may be necessary to make this composition. In this work, the AJC compiler and byte-code weaver for AspectJ and Java were used.

Each feature chosen to be part of the product must be present in the features model. Then, the features-packages mapping is consulted to determine if the chosen feature requires other features. Thus, the total set of features that must be present on the product is obtained. Among these features, it is verified which are already implemented in the repository. Features that are still not implemented must be so (see Section 4.1), as they are developed on demand for the product instantiation.

Besides implementing the features, in the PSI-PL example it was necessary to implement the main interface of the system, which is not associated to any feature, but is required to allow the access to the system functionalities. It was also necessary to implement a persistence layer to persist objects into the MySQL relational database.

Several product instantiations were done to evaluate the proposed approach. The combination process was done manually, and several different products were obtained by combining the base code with the optional features produced so far.

5. Related Work

Several works have been proposed that relate aspect-oriented programming and product line development. Alves et al. (2004) describe an incremental process for structuring a PL from existing mobile device game applications, in which continuous refactoring of existing code is done using aspect-oriented programming to isolate commonality from variability within the products. In our approach, several refactorings mentioned by Alves et al. can be used as a means of isolating the aspects from the original code, but we do not impose the existence of a code, as our reverse engineering step aims at extracting knowledge about the domain instead of code itself. Another relevant difference here is that our work considers features in a high granularity level (for example Appointment is a feature) while in Alves work a feature could be a very fine-grained characteristic of a game, for example the way images are loaded.

Loughran et al. (2004) propose an approach that joins framing and aspect-oriented techniques to integrate new features in product lines. This allows parameterization and reconfiguration support for the feature aspects. The framing technology has the disadvantage of being less intuitive, besides needing previous knowledge of the subsequent features. An advantage is to parameterize aspects at runtime. Although these techniques are applicable only in the coding phase, we plan to investigate how they could fit our process.

Apel et al. (2006) propose the integration of aspects and features at the architectural level through aspectual mixing layers (AMLs), also aiming at incremental software development. Our approach uses the same idea of integrating aspects and features at the architectural level, and our composition of core features and optional features can be thought of as a way, although very simplistic, of having the same effects of using AMLs. However, we do not make use of special language constructs and we do not treat the problem of aspects dependency and precedence.

Mezini and Ostermann (2004) present an analysis of feature-oriented and aspect-oriented modularization approaches with respect to variability management for product lines. Their discussion about weaknesses of each approach indicate the need for appropriate support for layer modules to better attend the development of software for product families, improving reuse of isolated layers. Even using AspectJ in our approach, we try to make it flexible to allow the use of other languages that support AOP and overcome AspectJ limitations.

Although all these approaches have points in common with our approach, our focus is on proposing a systematic process through which product lines for information systems can be built using AOP, so parts of these approaches can be incorporated into our process. For example, crosscut programming interfaces (XPIs) (Griswold at al, 2006) could be used to decouple aspect code from the advised code; and languages like Caesar or the concept of AMLs could be used instead of simply using AspectJ.

6. Concluding Remarks

AIPLE-IS allows incremental development of product lines using aspects. It is composed of a sequence of short, well-defined and reproducible steps, with well-defined

activities and artifacts. It considers the product line features as the main application assets, so the features are isolated, encapsulated, and designed with aspect orientation.

AOSD techniques facilitated the SPL features implementation. With the encapsulation supplied by AO, features become more cohesive, easier to be combined, and reusable. The very nature of aspect-oriented programming is responsible for easing features combination. OOAD techniques have been combined with AOSD to optimize design with separation of concerns. The approach has integrated AOSD with AO implementation, i.e., it supplies the basis for creating the whole product design, and also AO implementation techniques (guidelines) that can be used aiming at a good code with high cohesion and low coupling, reinforcing reusability.

The guidelines for AO implementation can also be used independently of other approach activities. However, it is more guaranteed to have a good code having a good design. This also means that the approach deals with the aspects problem in the development initial phases. For example, the requirements are already grouped by features. This problem is being largely discussed nowadays in the software community (e.g. Baniassad et al, 06).

Although the incremental nature of AIPLE-IS is an advantage, the code can become more complex with the introduction of new features that may involve modification of associations among existing features. This causes the code to have less intuitive meaning, which can be a disadvantage and imply in more difficult maintenance. Ongoing work is being done to tackle with this problem.

To help instantiating products, application generators could be used. A master dissertation research is being conducted with this goal. Captor (Shimabukuro et al, 06) is an application generator that automatically builds applications based on high level specification languages persisted in a repository. It is being extended to allow aspect oriented composition. Finally, other case studies should be performed to validate AIPLE-IS with other examples, possibly in other domains.

References

Alves, V., Matos Jr, P., and Borba, P. (2004) “An Incremental Aspect-Oriented Product Line Method for J2ME Game Development”, Workshop on Managing Variability Consistently in Design and Code (in conjunction with OOPSLA´2004).

Apel, S., Leich, T., Saake, G. (2006) Aspectual Mixin Layers: Aspects and Features in Concert. In: Proc. of International Conference on Software Engineering, p. 122-131.

Arkin, A., 2002. Business Process Modeling Language (BPML), Version 1.0. http://www.bpmi.org/ (last access: december, 2006)

AspectJ. The AspectJ Project. Disponível para acesso na URL: http://eclipse.org/aspectj/, em 10/11/2006.

Baniassad, E. L. A., Clements, P., Araújo, J., Moreira, A., Rashid, A.; Tekinerdogan, B., 2006. Discovering Early Aspects. In IEEE Software, v. 23, n 1, p. 61-70.

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7.

Buschmann F. et al., 1996. Pattern-oriented software architecture: A System of Patterns, Wiley.

Clarke, S.; Baniassad, E. L. A., 2005. Aspect Oriented Analysis and Design. Addison-Wesley Professional, ISBN 0321246748.

Elrad, T.; Filman, R. E.; Bader, A., 2001. Aspect Oriented Programming, Communications of the ACM, 44(10), October.

Gomaa, H., 2004. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley.

Griswold, W. G.; Shonle, M.; Sullivan, K.; Song, Y.; Tewari, N.; Cai, Y.; Rajan, H., 2006. Modular Software Design with Crosscutting Interfaces. IEEE Software, vol. 23, no. 1, p 51-60.

Jacobson, I.; Booch, G.; Rumbaugh, J., 1999. The Unified Process. IEEE Software (May/June).

Kang, K., et al., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Kiczales, G. Lamping, J. Menhdhekar, A. Maeda, C. Lopes, C. Loingtier, J. M. Irwin, J., 1997. Aspect-oriented programming. In: Proc. of the European Conference on Object-Oriented Programming, Springer-Verlag, p. 220–242.

Loughran, N., Rashid, A., Zhang, W., and Jarzabek, S. (2004) "Supporting Product Line Evolution with Framed Aspects". Workshop on Aspects, Components and Patterns for Infrastructure Software (held with AOSD 2004).

Mezini, M. and Ostermann, K. (2004),Variability Management with Feature-Oriented Programming and Aspects, Foundations of Software Engineering (FSE-12), ACM SIGSOFT.

Pacios, S. F.; Masiero, P. C.; Braga, R. T. V., 2006. Guidelines for Using Aspects to Evolve Product Lines. In: III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, p.111-120.

Pearce, D. J.; Noble, J., 2006. Relationship aspects. In: Proc. of the 5th international conference on Aspect-oriented software development, p. 75-86.

Prieto-Diaz, R.; Arango, G., 1991. Domain analysis and software system modeling. IEEE Computer Science Press Tutorial.

Rational, C., 2000. Unified Modeling Language. Available at: http://www.rational.com/uml/references (last access: December, 2006).

Shimabukuro, E. K.; Masiero, P. C.; Braga, R. T. V., 2006. Captor: A Configurable Application Generator, Proceedings of Tools Session of the 20th Simpósio Brasileiro de Engenharia de Software, p.121-128 (in Portuguese).

Yoder, J.W.; Balaguer, F.; Johnson, R., 2001. Architecture and Design of Adaptive Object Models. SIGPLAN Not. 36, p. 50-60.

GenArch – A Model-Based Product Derivation Tool

Elder Cirilo[1], Uirá Kulesza[1, 2], Carlos José Pereira de Lucena[1]

[1]Laboratório de Engenharia de Software – Departamento de Informática
Pontificia Universidade Católica do Rio de Janeiro (PUC-RIO), Brasil
{ecirilo,uira,lucena}@inf.puc-rio.br

[2]Departamento de Informática – Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa – Portugal

Abstract. *In this paper, we present a model-based tool for product derivation. Our tool is centered on the definition of three models (feature, architecture and configuration models) which enable the automatic instantiation of software product lines (SPLs) or frameworks. The Eclipse platform and EMF technology are used as the base for the implementation of our tool. A set of specific Java annotations are also defined to allow generating automatically many of our models based on existing implementations of SPL architectures.*

1. Introduction

Over the last years, many approaches for the development of system families and software product lines have been proposed [27, 7, 8, 14]. A system family [23] is a set of programs that shares common functionalities and maintain specific functionalities that vary according to specific systems being considered. A software product line (SPL) [7] can be seen as a system family that addresses a specific market segment. Software product lines and system families are typically specified, modeled and implemented in terms of common and variable features. A feature [10] is a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in SPLs.

Most of the existing SPL approaches [27, 7, 8, 14] motivate the definition of a flexible and adaptable architecture which addresses the common and variable features of the SPL. These SPL architectures are implemented by defining or reusing a set of different artifacts, such as object-oriented frameworks and software libraries. Recently, new programming techniques have been explored to modularize the SPL features, such as, aspect-oriented programming [1, 20], feature-oriented programming [5] and code generation [8]. Typical implementations of SPL architectures are composed of a set of different assets (such as, code artifacts), each of them addressing a small set of common and/or variable features.

Product Derivation [28] refers to the process of constructing a product from the set of assets specified or implemented for a SPL. Ideally the product derivation process would be accomplished with the help of instantiation tools to facilitate the selection, composition and configuration of SPL code assets and their respective variabilities. Over the last years, some product derivation tools have been proposed. Gears [13] and pure::variants [24] are two examples of tools developed in this context. Although these tools offer a set of useful functionalities for the product derivation, they are in general complex and heavyweight to be used by the mainstream developer community, because

they incorporate a lot of new concepts from the SPL development area. As a result, they suffer from the following deficiencies: (i) difficult to prepare existing SPL architecture implementations to be automatically instantiated; (ii) definition of many complex models and/or functionalities; and (iii) they are in general more adequate to work with proactive approaches [17].

In this context, this paper proposes a model-driven product derivation tool, called GenArch, centered on the definition of three models (feature, architecture and configuration). Initial versions of these three models can be automatically generated based on a set of code annotations that indicate the implementation of features and variabilities in the code of artifacts from the SPL. After that, a domain engineer can refine or adapt these initial model versions to enable the automatic product derivation of a SPL. The Eclipse [25] platform and model-driven development toolkits available (such as EMF and oAW) at this platform were used as a base for the definition of our tool.

The remainder of this paper is organized as follows. Section 2 presents background of generative programming and existing product derivation tools. Section 3 gives an overview of our product derivation approach based on the combined use of models and code annotations. Section 4 details the GenArch tool. Section 5 presents a set of initial lessons learned from the implementation and use of the GenArch tool. Finally, Section 6 concludes the paper and provides directions for future work.

2. Background

This section briefly revisits the generative programming approach (Section 2.1). Our SPL derivative approach is defined based on its original concepts and ideas. We also give an overview of existing product derivation tools (Section 2.2).

2.1 Generative Programming

Generative Programming (GP) [8] addresses the study and definition of methods and tools that enable the automatic generation of software from a given high-level specification language. It has been proposed as an approach based on domain engineering [4]. GP promotes the separation of problem and solution spaces, giving flexibility to evolve both independently. To provide this separation, Czarnecki & Eisenecker [8] propose the concept of a generative domain model. A generative domain model is composed of three basic elements: (i) *problem space* – which represents the concepts and features existent in a specific domain; (ii) *solution space* – which consists of the software architecture and components used to build members of a software family; and (iii) *configuration knowledge* – which defines how specific feature combinations in the problem space are mapped to a set of software components in the solution space. GP advocates the implementation of the configuration knowledge by means of code generators.

The fact that GP is based on domain engineering enables us to use domain engineering methods [4, 8] in the definition of a generative domain model. Common activities encountered in domain engineering methods are: (i) domain analysis – which is concerned with the definition of a domain for a specific software family and the identification of common and variable features within this domain; (ii) domain design – which concentrates on the definition of a common architecture and components for this

domain; and (iii) domain implementation – which involves the implementation of architecture and components previously specified during domain design.

Two new activities [8] need to be introduced to domain engineering methods in order to address the goals of GP: (i) development of a proper means to specify individual members of the software family. Domain-specific languages (DSLs) must be developed to deal with this requirement; and (ii) modeling of the configuration knowledge in detail in order to automate it by means of a code generator.

In a particular and common instantiation of the generative model, the feature model is used as a domain-specific language of a software family or product line. It works as a configuration DSL. A configuration DSL allows to specify a concrete instance of a concept [8]. Several existing tools adopt this strategy to enable automatic product derivation (Section 2.2) in SPL development. In this work, we present an approach and a tool centered on the ideas of the generative model. The feature model is also adopted by our tool as a configuration DSL which expresses the SPL variabilities.

2.2 Existing Product Derivation Tools

There are many tools to automatically derive SPL members available in industry, such as Pure::variants and Gears. Pure::variants [24] is a SPL model-based product derivation tool. Its modeling approach comprises three models: features, family and derivation. The feature model contains the product variabilities and solution architectures are expressed in a family model. Both models are flexibly combined to define a SPL. Since a product line specification in this tool can consist of any number of models, the “configuration space” is used to manage this information and captures variants. The features are modeled graphically in different formats such as trees, tables and diagrams. Constraints among features and architecture elements are expressed using first order logic in Prolog and uses logic expression syntax closely related to OCL notation. This tool permits the use of an arbitrary number of feature models, and hierarchical connection of the different models. The pure::variants does not require any specific implementation technique and provides integration interfaces with other tools, e.g. for requirements engineering, test management and code generation.

Gears [13] allows the definition of a generative model focused on automatic product derivation. It defines three primary abstractions: feature declarations, product definitions, and variation points. Feature declarations are parameters that express the variations. Product definitions are used to select and assign values to the feature declaration parameters for the purpose of instantiating a product. Variation points encapsulate the variations in your software and map the feature declarations to choices at these variation points. The language for expressing constraints at feature models is propositional logic instead of full first-order logic.

3. Approach Overview

In this section, we present an overview of our product derivation approach based on the use of the GenArch tool. Next section details the tool by showing its architecture, adopted models and supporting technologies. Our approach aims to provide a product derivation tool which enables the mainstream software developer community to use the concepts and foundations of the SPL approach in the development of software systems

and assets, such as, frameworks and customizable libraries, without the need to understand complex concepts or models from existing product derivation tools.

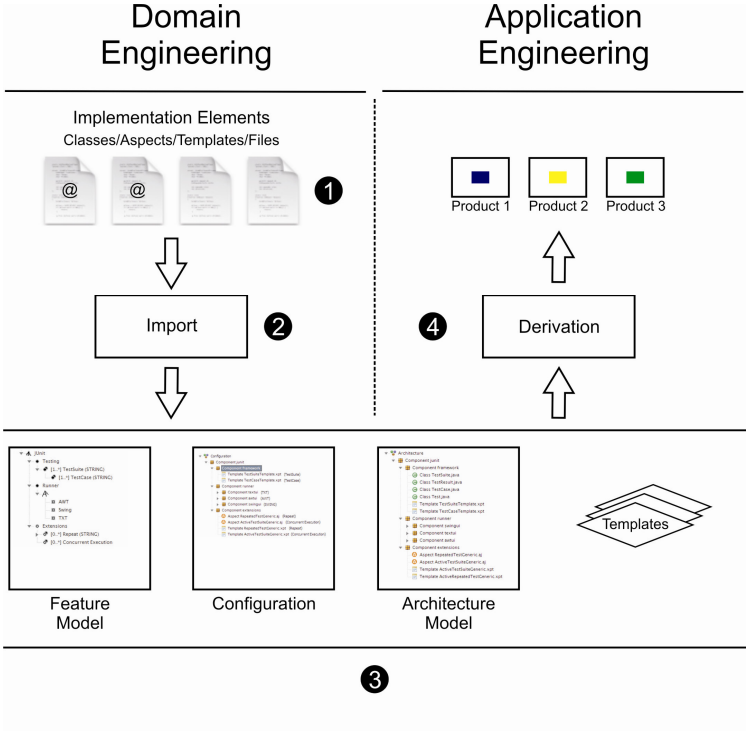


Figure 1. Approach Overview

Figure 1 gives an overview of our approach. Initially (step 1), the domain engineers are responsible to annotate the existing code of SPL architectures (e.g. an object-oriented framework). We have defined a set of Java annotations[1] to be inserted in the implementation elements (classes, interfaces and aspects) of SPL architectures. The purpose of our annotations is twofold: (i) they are used to specify which SPL implementation elements correspond to specific SPL features; and (ii) they also indicate that SPL code artifacts, such as an abstract class or aspect, represent an extension point (hot-spot) of the architecture.

After that, the GenArch tool processes these annotations and generates initial versions of the derivation models (step 2). Three models must be specified in our approach to enable the automatic derivation of SPL members: (i) an architecture model; (ii) a feature model; and (iii) a configuration model. The architecture model defines a visual representation of the SPL implementation elements (classes, aspects, templates, configuration and extra files) in order to relate them to feature models. It is automatically derived by parsing an existing directory containing the implementation elements (step 2). Code templates can also be created in the architecture model to specify implementation elements which have variabilities to be solved during application engineering. Initial versions of code

[1] Although the current version of GenArch tool has been developed to work with Java technologies, our approach is neutral with respect to the technology used. It only requires that the adopted technologies provides support to the definition of its annotations and models.

templates are automatically created in the architecture models based on GenArch annotations (see details in Section 4.2).

Feature models [16] are used in our approach to represent the variable features (variabilities) from SPL architectures (step 3). During application engineering, application engineers create a feature model instance (also called a configuration [9]) in order to decide which variabilities are going to be part of the final application generated (step 4). Finally, our configuration model is responsible to define the mapping between features and implementation elements. It represents the configuration knowledge from a generative approach [8], being fundamental to link the problem space (features) to the solution space (implementation elements). Each annotation embedded in an implementation element is used to create in the configuration model, a mapping relationship between an implementation element and a feature.

The initial versions of the derivation models, generated automatically by GenArch tool, must be refined by domain engineers (step 3). During this refinement process, new features can be introduced in the feature model or the existing ones can be reorganized. In the architecture model, new implementation elements can also be introduced or they can be reorganized. Template implementations can include additional common or variable code. Finally, the configuration model can also be modified to specify new relationships between features and implementations elements. In the context of SPL evolution, the derivation models can be revisited to incorporate new changes or modifications according to the requirements or changes required by the evolution scenarios.

After all models are refined and represent the implementation and variabilities of a SPL architecture, the GenArch tool uses them to automatically derive an instance/product of the SPL (step 4). The tool processes the architecture model by verifying if each implementation element depends on any feature from the feature model. This information is provided by the configuration model. If an implementation element does not depend on a feature, it is automatically instantiated since it is mandatory. If an implementation depends on a feature, it is only instantiated if there is an occurrence of that specific feature in the feature model instance created by the application engineer. Every template element from the architecture model, for example, must always depend on a specific feature. The information collected by that feature is then used in the customization of the template. The GenArch tool produces, as result of the derivation process, an Eclipse/Java project containing only the implementation elements corresponding to the specific configuration expressed by the feature model instance and specified by the application engineers.

4. GenArch – Generative Architecture Tool

In this section, we present the architecture, adopted models and technologies used in the development of the GenArch tool. Following subsections detail progressively the functionalities of the tool by illustrating its use in the instantiation of the JUnit framework.

4.1. Architecture Overview

The GenArch tool has been developed as an Eclipse plug-in [25] using different technologies available at this platform. New model-driven development toolkits, such as Eclipse Modeling Framework (EMF) [6] and openArchitectureWare (oAW) [22] were

used to specify its models and templates, respectively. Figure 2 shows the general structure of the GenArch architecture based on Eclipse platform technologies. Our tool uses the JDT (Java Development Tooling) API [25] to browse the Abstract Syntax Tree (AST) of Java classes in order to: (i) parse the Java elements to create the architecture model; and (ii) to process the GenArch annotations.

The feature, configuration and architecture model of GenArch tool are specified using EMF. EMF is a Java/XML framework which enables the building of MDD based tools based on structured data models. It allows generating a set of Java classes to manipulate and specify visually models. These classes are generated based on a given meta-model, which can be specified using XML Schema, annotated Java classes or UML modeling tools (such as Rational Rose). The feature model used in our tool is specified by a separate plugin, called FMP (Feature Modeling Plugin) [3]. It allows modeling the feature model proposed by Czarnecki et al [8], which supports modeling mandatory, optional, and alternative features, and their respective cardinality. The FMP also works with EMF models.

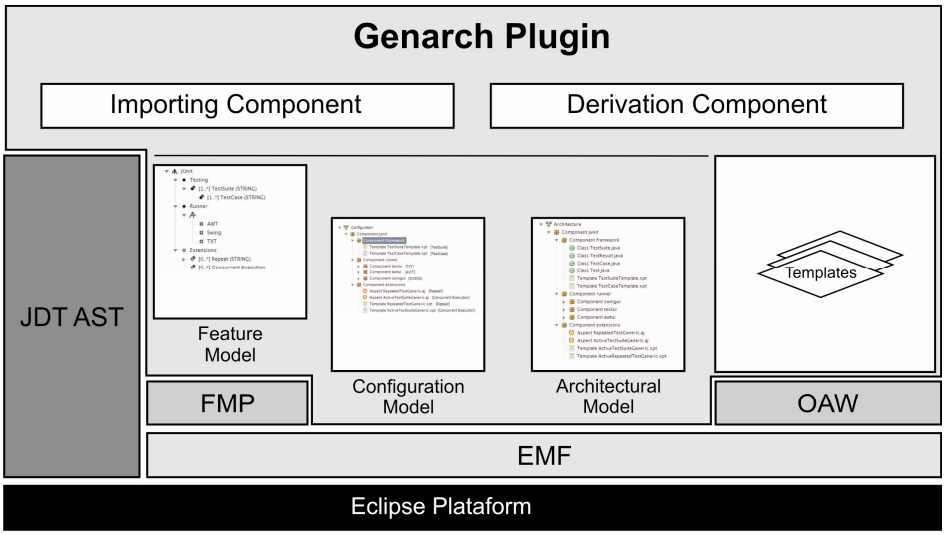


Figure 2. Genarch Architecture

The openArchitectureWare (OAW) plug-in [22] proposes to provide a complete toolkit for model-driven development. It offers a number of prebuilt workflow components that can be used for reading and instantiating models, checking them for constraint violations, transforming them into other models and then, finally, for generating code. oAW is also based on EMF technology. Currently, GenArch plug-in has only adopted the XPand language of oAW to specify its respective code templates (see details in Section 4.4).

4.2. Annotating Java Code with Feature and Variabilities

Following the steps of our approach described in Section 3, the domain engineer initially creates a set of Java annotations in the code of implementation elements (classes, aspects and interfaces) from the PL architecture. The annotations are in general embedded in the code of implementation elements representing the SPL variabilities. Table 1 shows the two kinds of annotations supported by our approach: (i) `@Feature` – this annotation is used to indicate that a particular implementation element addresses a specific feature. It

also allows to specify the kind of feature (mandatory, alternative, or optional) being implemented and its respective feature parent if exists; and (ii) `@Variability` – it indicates that the implementation element annotated represents an extension point (e.g. a hotspot framework class) in the SPL architecture. In the current version of GenArch, there are three kinds of implementation elements that can be marked with this annotation: abstract classes, abstract aspects, and interfaces. Each of them has a specific type (hotspot or hotspot-aspect) defined in the annotation.

Table 1. GenArch Annotations and their Attributes

@Feature	
Attributes	
Name	Name of feature
Parent	The parent of feature
Type	alternative, optional or mandatory
@Variability	
Attributes	
Type	hotspot or hotspotAspect
Feature	Contains the feature associated with the variability

```
@Feature(name="TestCase",parent="TestSuite",type=FeatureType.mandatory)
@Variability(type=VariabilityType.hotSpot,feature="TestCase")
public abstract class TestCase extends Assert implements Test {
    private String fName;

    public TestCase() {
        fName= null;
    }
    public TestCase(String name) {
        fName= name;
    }
    ...
}
```

Figure 3. `TestCase` class annotated

Along the next sections, we will use the JUnit testing framework to illustrate the GenAch functionalities. In particular, we are considering the following components of this framework:

(I) Core – defines the framework classes responsible for specifying the basic behavior to execute test cases and suites. The main hot-spot classes available in this component are `TestCase` and `TestSuite`. The framework users need to extend these classes in order to create specific test cases to their applications;

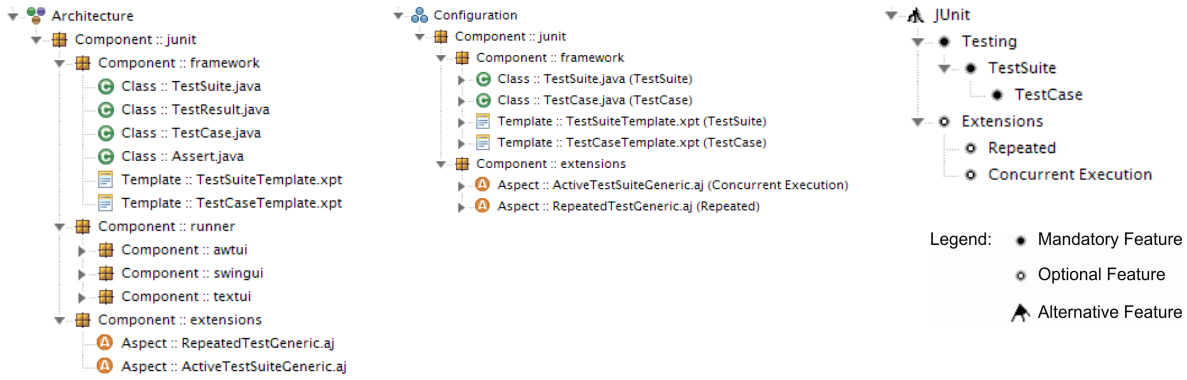
(II) Runner – this component is responsible for offering an interface to start and track the execution of test cases and suites. JUnit provides three alternative implementations of test runners, as follows: a command-line based user interface (UI), an AWT based UI, and a Java Swing based UI; and, finally,

(III) Extensions – responsible for defining functionality extending the basic behavior of the JUnit framework. Examples of available extensions are: a test suite to execute test suites in separate threads and a test decorator to run test cases repeatedly.

Figure 3 shows the `TestCase` abstract class from the JUnit framework marked with two GenArch annotations. The `@Feature` annotation indicates that the `TestCase` class is implementing the TestCase feature and has the TestSuite as feature parent. It also shows that this feature is mandatory. This means that every instance of the JUnit framework requires the implementation of this class. The `@Variability` annotation specifies that the `TestCase` class is an extension point of the JUnit framework. It represents a hot-spot class that needs to be specialized when creating test cases (a JUnit framework instantiation) for a Java application. Next section shows how GenArch annotations are processed to generate the initial version of the derivation models.

4.3. Generating and Refining the Approach Models

In the second step of our approach, an initial version of each GenArch model is produced. The architecture model is created by parsing the Java project or directory that contains the implementation elements of the SPL architecture. The Eclipse Java Development Tooling (JDT) API [25] is used by our plug-in to parse the existing Java code. During this parsing process, every Java package is converted to a component with the same name in the architecture model. Each type of implementation element (classes, interfaces, aspects or files) has a corresponding representation in the architecture model. Figure 4(a) shows, for example, the initial version of the JUnit architecture model. Every package was converted to a component and every implementation element was converted to its respective abstraction in the architecture model. As we mentioned before, architecture models are created only to allow the visual representation of the SPL implementation elements in order to relate them to a feature model.



(a) Architecture Model **(b) Configuration Model** **(c) Feature Model**

Figure 4. The JUnit GenArch Models – Initial Versions

The GenArch annotations are used to generate specific elements in the feature, configuration and architecture models. These elements are also processed by the tool using the Eclipse Java Development Tooling (JDT) API [25]. The JDT API allows browsing the Abstract Syntax Tree (AST) of Java classes to read their respective annotations. The `@Feature` annotation is used to create different features with their respective type in the feature model. Every `@Feature` annotation demands the creation of a new feature in the feature model. If the `@Feature` annotation has a `parent` attribute, a parent feature is created to aggregate the new feature. Figure 4(c) shows the partial

feature model generated for the JUnit framework. It aggregates, for example, the TestSuite and TestCase features, which were generated based on the `@Feature` annotation presented in Figure 3.

On the other hand, each `@Variability` annotation demands the creation of a code template which represents concrete instances of the extension implementation element that is annotated. The architecture model is also updated to include the new template element created. Consider, for example, the annotated `TestCase` class presented in Figure 3. The `@Variability` annotation of this class demands the creation of a code template which represents `TestCase` subclasses, as we can see in Figure 4(a). This template will be used in the derivation process of the JUnit framework to generate `TestCase` subclasses for a specific Java application under testing. Only the structure of each template is generated based on the respective implementation element (abstract class, interface or abstract aspect) annotated. Empty implementations of the abstract methods or pointcuts from these elements are automatically created in the template structure. Next section shows how the code of every template can be improved using information collected by a feature model instance.

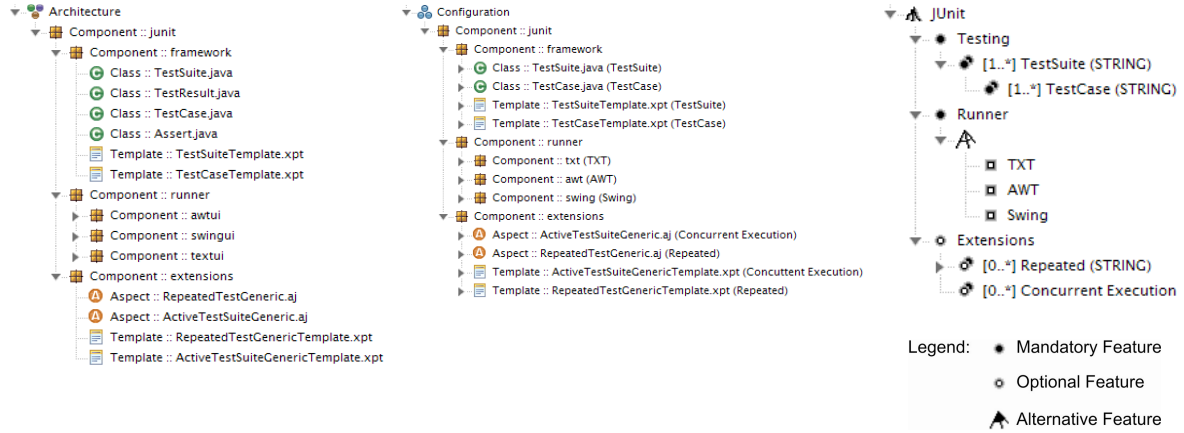
The GenArch configuration model defines a set of mapping relationships. Each mapping relationship links a specific implementation element from the architecture model to one or more features from the feature model. An initial version of the configuration model is created based on the `@Feature` annotations with attribute `type` equals to `optional` or `alternative`. When processing these annotations, the GenArch tool adds a mapping relationship between the feature created and the respective implementation element annotated. The current visualization of our configuration model shows the implementation elements in a tree (similar to the architecture model), but with the explicit indication of the feature(s) which each of them depends on. Figure 4(b) shows the initial configuration model of the JUnit framework based on the processed annotations. As we can see, the `RepeatedTestAspect` aspect[2] depends explicitly on the Repeated feature. It represents a mapping relationship between these elements, created because the `RepeatTestAspect` is marked with the `@Feature` annotation and its attributes have the following values: (i) `name` equals to `Repeated`; and (ii) `type` equals to `optional`.

In the GenArch tool, every template must depend at least on a feature. The `@Variability` annotation specifies explicitly this feature. This information is used by the tool to update the configuration model by defining that the template generated depends explicitly on a feature. Figure 4(b) shows that the `TestCaseTemplate` depends explicitly on the TestCase feature. It means that during the derivation/instantiation of the JUnit framework, this template will be processed to every TestCase feature created in the feature model instance.

After the generation of the initial versions of GenArch models, the domain engineer can refine them by including, modifying or removing any feature, implementation element or

[2] In this paper, we are using the implementation of the JUnit framework, which was refactored using the AspectJ language. Two features of the Extension component were implemented as aspects in this version: (i) the repetition feature – which allows to repeat the execution of specific test cases; and (ii) the active execution feature – which addresses the concurrent execution of test suites. Additional details about it can be found in [18, 20].

mapping relationship. Figure 5 shows a partial view of the final versions of the JUnit framework models. As we can see, they were refined to completely address the features and variabilities of the JUnit.



(a) Architecture Model **(b) Configuration Model** **(c) Feature Model**

Figure 5. The JUnit GenArch Models – Final Versions

4.4. Implementing Variabilities with Templates

The GenArch tool adopts the XPand language from the oAW plugin [22] to specify the code templates. XPand is a very simple and expressive template language. In our approach, templates are used to codify implementation elements (classes, interfaces, aspects and configuration files) which need to be customized during the product derivation. Examples of implementation elements which can be implemented using templates are: concrete instances of hot-spots (classes or aspects) and parameterized configuration files. Every GenArch template can use information collected by the feature model to customize its respective variable parts of code.

```
01 «IMPORT featuremodel»
02 «DEFINE TestSuiteTemplate FOR Feature»
03   «FILE attribute + ".java"»
04     package junit.framework;
05     public class «attribute»  {
06     public static Test suite() {
07     TestSuite suite = new TestSuite();
08     «FOREACH features AS child»
09     suite.addTestSuite(«child.attribute».class);
10     «ENDFOREACH»
11     return suite;
12     }
13   }
14   «ENDFILE»
15 «ENDDEFINE»
```

Figure 6. The `TestCaseTemplate`

Figure 6 shows the code of the `TestSuite` template using the XPand language. It is used to generate the code of specific JUnit test suites for Java applications. The `IMPORT`

statement allows using all the types and template files from a specific namespace (line 1). It is equivalent to a Java import statement. GenArch templates import the `featuremodel` namespace, because it contains the classes representing the feature model. The `DEFINE` statement determines the template body (line 2). It defines the name of the template as well as the meta-model class whose instances will be used in its customization. The `TestSuiteTemplate` template, for example, has the name `TestSuiteTemplate` and uses the `Feature` meta-model class to enable its customization. The specific feature to be used in the customization of each template is defined by the mapping relationship in the configuration model. Thus, the `TestSuiteTemplate`, for example, will be customized using each TestSuite feature specified in a feature model instance by the application engineer (see configuration model in Figure 5(b)).

Inside the `DEFINE` tags (lines 3 to 14) are defined the sequence of statements responsible for the template customization. The `FILE` statement specifies the output file to be written the information resulting from the template processing. Figure 6 indicates that the file name resulting from the template processing will have the name of the feature being manipulated (`attribute`) plus the Java extension (`.java`). The following actions are accomplished in order to customize the `TestSuiteTemplate`: (i) the name of the resulting test case class is obtained based on the feature name (`attribute`); and (ii) the test cases to be included at this test suite are specified based on the feature names (`child.attribute`) of the feature child (`child`). The `FOREACH` statement allows the processing of the child features of the TestSuite feature being processed for this template.

4.5. Generating SPL Instances

In the fourth and last step of our approach (Section 3), called product derivation, a product or member of the SPL is created. The product derivation is organized in the following steps: (i) choice of variabilities in a feature model instance – initially the application engineer specifies a feature model instance using the FMP plug-in which determines the final product to be instantiated; (ii) next, the application engineer provides to GenArch tool, the architecture and configuration model of the SPL and also the feature model instance specified in step (i); and (iii) finally, the GenArch tool processes all these models to decide which implementation elements needs to be instantiated to constitute the final application requested. The selected implementation elements are then loaded in a specific source folder of an Eclipse Java project. The complete algorithm used by GenArch tool can be found in [18, 19, 20]

5. Lessons Learned and Discussions

This section provides some discussions and lessons learned based on the initial experience of use the GenArch tool to automatically instantiate the JUnit framework and a J2ME SPL game [18]. Although these examples are not so complex, they allow to illustrate and exercise all the tool functionalities. Additional details about the models and code generated for these case studies will be available at [29].

Synchronization between Code, Annotations and Models. In the current version of the GenArch tool, there is no available functionality to synchronize the SPL code, annotations and respective models. This is fundamental to guarantee the compatibility of the SPL artifacts and to avoid inconsistencies during the process of product derivation. Besides, it is also important to allow that specific changes in the code, models or

annotations will be reflected on the related artifacts. At this moment, we are focusing at the implementation of a synchronization functionality which tries to solve automatically the inconsistencies between models, code and annotations, and if it is not possible it only shows the inconsistency to the product line engineer, as a GenArch warning or error. The following inconsistencies are planned to be verified by our tool: (i) removing of features from the feature model which are not longer used by the configuration model or annotation; (ii) removing of mapping relationships in the configuration model which refer to non-existing features or implementation elements; (iii) removing of implementation elements from the architecture model which do not exist anymore; and (iv) automatic creation of annotations in implementation elements based on information provided by the configuration model. Finally, the implementation of our synchronization functionality can also enable the automatic generation of implementation elements with annotations, from existing GenArch models.

Integration with Refactoring Tools. Application of refactoring techniques [12, 21] is common nowadays in the development of software systems. In the development of SPLs, refactoring is also relevant, but it assumes a different perspective. In the context of SPL development, refactoring technique needs to consider [2], for example: (i) if changes applied to the structure of existing SPL implementation elements do not decrease the set of all possible configurations (products) addressed by the SPL; and (ii) complex scenarios of merging existing programs into a SPL. Although many existing proposed refactorings introduce extension points or variabilities [2], the refactoring tools available are not strongly integrated with existing SPL modeling or derivation tools. It can bring difficulties or inconsistencies when using both tools together in the development of a SPL. The integration of GenArch with existing refactoring tools involves several challenges, such as, for example: (i) to allow the creation of `@Feature` annotations to every refactoring that exposes or creates a new variable feature in order to present it in the SPL feature model to enable its automatic instantiation; and (ii) refactorings that introduce new extension points (such as, abstract classes or aspects or an interface) must be integrated with GenAch to allow the automatic insertion of `@Variability` annotations. Also the functionality of synchronization of models, code and annotations (discussed in Section 5.1) is fundamental in the context of integration of GenArch with existing refactoring tools, because it guarantees that every refactoring applied to existing SPL implementation elements, which eventually cause the creation of new or modification of existing annotations, will be synchronized with the derivation models.

SPL Adoption Strategies. Different adoption strategies [17] can be used to develop software product lines (SPLs). The proactive approach motivates the development of product lines considering all the products in the foreseeable horizon. A complete set of artifacts to address the product line is developed from scratch. In the extractive approach, a SPL is developed starting from existing software systems. Common and variable features are extracted from these systems to derive an initial version of the SPL. The reactive approach advocates the incremental development of SPLs. Initially, the SPL artifacts address only a few products. When there is a demand to incorporate new requirements or products, the common and variable artifacts are incrementally extended in reaction to them. Independent from the SPL adoption strategy adopted, a derivation tool is always needed to reduce the cost of instantiation of complex architectures implemented to SPLs.

We believe that GenArch tool can be used in conjunction with proactive, extractive or incremental adoption approaches. In the proactive approach, our tool can be used to annotate the implementation elements produced during domain engineering in order to prepare those elements to be automatically instantiated during application engineering. Also, the extractive approach can demand the introduction of GenArch annotations in classes, interfaces or aspects, whenever new extension points are exposed in order to gradually transform the existing product in a SPL. Finally, the reactive approach requires the implementation of the synchronization functionality (Section 5.1) in the GenArch tool, because it can involve complex scenarios of merging products.

Architecture Model Specialization. The architecture model supported currently by GenArch tool allows to represent SPL architectures implemented in the Java and AspectJ programming languages. However, our architecture model is not dependent of Java technologies, only the GenArch functionalities responsible to manipulate the implementation elements were codified to only work with Java and AspectJ implementation elements. Examples of such functionalities are: (i) the parser that imports and processes the implementation elements and annotations; and (ii) the derivation component that creates a SPL instance/product as a Java project. In this way, the GenArch approach, as presented in Section 3, is independent of specific technologies. Currently, we are working on the definition of specializations of the architecture model. These specializations have the purpose to support other abstractions and mechanisms of specific technologies. In particular, we are modifying the tool to work with a new architecture model that supports the abstractions provided by the Spring framework [15]. It is a specialization of our current architecture model. It will allow to work not only with Java and AspectJ elements, but also with Spring beans and their respective configuration files.

6. Conclusions and Future Work

In this paper, we presented GenArch, a model-based product derivation tool. Our tool combines the use of models and code annotations in order to enable the automatic product derivation of existing SPLs. We illustrated the use of our tool using the JUnit framework, but we also validated it in the context of a J2ME Games SPL. As a future work, we plan to evolve the GenArch functionalities to address the following functionalities: (i) synchronization between models, code and annotations (as described in Section 5.1); (ii) to extend the GenArch parsing functionality to allow the generation of template structure based on existing AspectJ abstract aspects; (iii) to address the aspect-oriented generative model proposed in [20] in order to enable the customization of pointcuts from feature models.

Acknowledgments. The authors are supported by ESSMA/CNPq project under grant 552068/2002-0. Uirá is also partially supported by European Commission Grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE).

References

[1] V. Alves, P. Matos, L. Cole, P. Borba, G. Ramalho. “Extracting and Evolving Mobile Games Product Lines”. Proceedings of SPLC'05, pp. 70-81, September 2005.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C. Lucena. Refactoring Product Lines, Proceedings of the GPCE´2006, ACM Press: Portland, Oregon, USA.

[3] M. Antkiewicz, K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004.
[4] G. Arrango. Domain Analysis Methods. In Software Reusability, New York, pp. 17-49, 1994.
[5] Y. Smaragdakis, D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM TOSEM, 11(2): 215-255 (2002).
[6] F. Budinsky, et al. Eclipse Modeling Framework. Addison-Wesley, 2004.
[7] P. Clements, L. Northrop. Software Product Lines: Practices and Patterns. 2001: Addison-Wesley Professional.
[8] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[9] K. Czarnecki, S. Helsen, U. Eisenecker. Staged Configuration Using Feature Models. In Proceedings of the Third Software Product-Line Conference, September 2004.
[10] K. Czarnecki, S. Helsen. Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal, 45(3), 2006, pp. 621-64.
[11] K. Czarnecki, M. Antkiewicz, C. Kim. "Multi-level Customization in ApplicationEngineering". CACM, Vol. 49, No. 12, pp. 61-65, December 2006.
[12] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison, 1999.
[13] Gears/BigLever, URL: http://www.biglever.com/, January 2007.
[14] J. Greenfield, K. Short. Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools. 2005: John Wiley and Sons.
[15] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, C. Sampaleanu. Professional Java Development with the Spring Framework, Wrox, 2005.
[16] K. Kang, et al. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Pittsburgh, PA, November 1990.
[17] C. Krueger. "Easing the Transition to Software Mass Customization". In Proceedings of the 4th PFE, pp. 282–293, 2001.
[18] U. Kulesza, et al. Mapping Features to Aspects: A Model-Based Generative Approach. Early Aspects@AOSD´2007 Post-Workshop Proceedings, LNCS 4765, Springer-Verlag.
[19] U. Kulesza, C. Lucena, P. Alencar, A. Garcia. Customizing Aspect-Oriented Variabilites Using Generative Techniques. Proceedings of SEKE'06, July 2006.
[20] U. Kulesza. Uma Abordagem Orientada a Aspectos para o Desenvolvimento de Frameworks. Rio de Janeiro, 2007. Tese de Doutorado, DI/PUC-Rio, Abril 2007.
[21] W. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
[22] openArchitectureWare, URL: http://www.eclipse.org/gmt/oaw/,
[23] D. L. Parnas. On the Design and Development of Program Families. IEEE Transactions on Software Engineering (TSE), 1976. 2(1): p. 1-9.
[24] Pure::Variants, URL: http://www.pure-systems.com/, January 2007.
[25] S. Shavor, et al. The Java Developer's Guide to Eclipse. Addison-Wesley, 2003.
[26] T. Stahl, M. Voelter. Model-Driven Software Development: Technology, Engineering, Management. 2006: Wiley.
[27] D. Weiss, C. Lai. Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley Professional, 1999.
[28] S. Deelstra, M. Sinnema, J. Bosch. Product derivation in software product families: a case study. Journal of Systems and Software 74(2): 173-194, 2005.
[29] GenArch – Generative Architectures Plugin, URL: http://www.teccomm.les.inf.puc-rio.br/genarch/.

Technical Session II: Methods and Models for Software Reuse

Automatic Generation of Platform Independent Built-in Contract Testers

Helton S. Lima[1], **Franklin Ramalho**[1], **Patricia D. L. Machado**[1], **Everton L. Galdino** [1]

[1]Formal Methods Group, Federal University of Campina Grande
Campina Grande, Brazil

{helton,franklin,patricia,everton}@dsc.ufcg.edu.br

Abstract. *Automatic generation of built-in contract testers that check pairwise interactions between client and server components is presented. Test artifacts are added to development models, i.e., these models are refined towards testing. The refinement is specified by a set of ATL transformation rules that are automatically performed by the ATL-DT tool. We focus on the KobrA development methodology and the Built-In Testing (BIT) method. Therefore, development and testing artifacts are specified by UML diagrams. To make transformation from development to testing models possible, a UML 2 profile for the BIT method is proposed. The overall solution is part of the MoBIT (Model-driven Built-In contract Testers) tool that was fully developed following Model-Driven Engineering (MDE) approaches. A case study is presented to illustrate the problems and solutions addressed.*

1. Introduction

Component-based software development practices can promote the development of high quality software that can be more effectively verified and validated. This is due to the principles behind the practices such as uniformity, encapsulation and locality. Also, since component interfaces are usually specified, for instance, using UML diagrams, functional test cases can be derived from them favoring an effective test coverage of component services and improving reliability and productivity in the testing process, particularly if testing activities are automated.

Deriving functional test cases from functional model specifications to check conformity between an implementation and its specification is commonly known as Model-Based Testing (MBT) [El-Far and Whittaker 2001]. Test suites can be constructed as soon as specifications are produced, favoring reuse of development artifacts and also providing a strong link between development and testing teams. Test cases are related to the intended functional requirements.

Research has been conducted towards effective model-based testing approaches for component-based software. Notable progress has already been made, including methods, techniques and tools [Atkinson and Gross 2002, Briand et al. 2006, Barbosa et al. 2007]. However, there are still challenges to be faced to cope with the increasing complexity and diversity of systems. For instance, apart from test case generation from UML artifacts, a number of artifacts are needed to support test execution such as test drivers, context dependencies and concrete instances of method arguments. These artifacts can also be constructed from development artifacts. Also, as components are developed to be reused and because applications may run in different platforms, it is

crucial that test artifacts can be easily modified to reflect development models variabilities. In this sense, there is a growing interest in combining MBT with Model-Driven Engineering (MDE) strategies, where test artifacts generation is defined through model transformations that can be automated. This is know as Model-Driven Testing (MDT).

The expected benefits of using a MDE approach for testing are twofold. The first is to lower the cost of deploying a given component and corresponding testers on multiple platforms through reuse of the models. The second is to reach a higher level of automation in the test case generation process, through the specification of transformations among those models. In addition, the separation of concerns inherent of MDE added by the level of abstraction and orthogonality deals perfectly with MDT perspectives.

This paper presents a model-driven testing solution for component-based software that supports the automatic implementation of testers according to the Built-In contract Testing (BIT) method proposed in [Atkinson and Gross 2002]. BIT is based on testing components functionality on the client side and testing interfaces on the server side of a pairwise contract. This method is integrated with the KobrA component development methodology [Atkinson et al. 2002]. UML is the central notation in both methods. To allow the automatic mapping from KobrA component specifications to BIT artifacts, we propose a BIT profile and rules for transforming pure UML 2 Kobra diagrams to UML 2 Kobra diagrams fully annotated with BIT profile. Transformations are automated by using the ATL-DT tool [AMMA Project 2005], a MDE based tool that provides ways to produce a set of target models from a set of source models. As a result, the Model-driven Built-In contract Testing (MoBIT) tool was developed.

The reasons for choosing KobrA and BIT are the following. On one hand, KobrA provides a reasonable coverage of the main activities of component-based software development. Also, it is independent of a specific platform. Moreover, it has a clear separation of concerns and is fully based on UML models and OCL specifications, that are the pivotal elements within the MDE vision. Moreover, KobrA is defined towards MDE guidelines: UML models, OCL specifications, profiles, transformations between them (step-by-step described in natural language) and meta-models. On the other hand, BIT is the testing method prescribed to KobrA applications, also based on UML artifacts. In addition, it is flexible in the sense that different test case generation algorithms can be easily incorporated. Moreover, it deals with different aspects of modelling (structural, functional, behavioral diagrams) and generates test artifacts based on them. Furthermore, it covers all aspects concerning test specification and realization.

The paper is organized as follows. Section 2 presents basic terminology on MDE and MDT and briefly describes the KobrA and BIT methods. Section 3 presents the architecture of the MoBIT solution. Section 4 introduces the BIT profile whereas Section 5 introduces the transformation rules. Section 6 presents a case study that illustrates the application of the MoBIT solution focusing on the application of the proposed transformation rules. Section 7 discusses related works. Finally, Section 8 present concluding remarks and pointers for further work.

2. Background

This section briefly presents the main concepts and methods addressed in this paper.

2.1. MDE

Model-Driven Engineering (MDE) is a software engineering approach defined by the OMG. Its main goal is to speed-up the development process by providing the highest possible level of automation for the following tasks: implementation, migration from one implementation platform to another, evolution from one version to the next, integration with other software.

The key idea of the MDE is to shift the emphasis in effort and time during the software life cycle away from implementation towards modelling, meta-modelling and model transformations. In order to reach this goal, MDE prescribes the elaboration of a set of standard models. The first one is the Computational Independent Model (CIM) that captures an organization's ontology and activities independently of the requirements of a computational system. The CIM is used as input for the elaboration of the Platform Independent Model (PIM), which captures the requirements and design of a computational system independently of any target implementation platform. In turn, the PIM serves as input for the elaboration of the Platform Specific Model (PSM), a translation of the PIM geared towards a specific platform. Finally, the PSM serves as input to the implementation code.

The expected benefits of this MDE approach are twofold. The first is to lower the cost of deploying a given application on multiple platforms through reuse of the CIM and PIM. The second is to reach a higher level of automation in the development process, through the specification of transformations among those standard models.

2.2. KobrA

KobrA [Atkinson et al. 2002] is a component-based software development methodology where engineering practices are supported mostly by the use of the UML notation for modelling. The main focus is on Product-Line Engineering (PLE) with the development cycle split into two parts: one deals with the development of a framework and the other with the development of an application as a concrete instance of the framework. KobrA frameworks and applications are all organized in terms of hierarchies of components. Also, the method adopts MDE principles in order to allow core assets to be developed as PIMs as well as to provide a highly cost-effective and flexible technique for implementing KobrA components. The methodology presents guidelines on how to break large models into smaller transformable models; approaches for organizing CIMs and PIMs as components; and what diagrams to use.

With a strict separation of concerns between product (what to produce) and process (how should it be produced), KobrA is aimed at being simple and systematic. Also, it is independent of specific implementation technologies. Moreover, quality assurance practices such as inspections, testing and quality modelling are integrated with development activities. Furthermore, systems and components are treated uniformly: all significant elements should be viewed as components.

KobrA supports the principle of encapsulation in which a component is defined in terms of a specification that describes its provided services and a realization that describes how it provides these services. In the sequel, we focus on component modelling and specification in the KobrA methodology which is the basis of our proposal.

The basic goal of component specification is to create a set of models that collectively describe the external view of a component. A component specification may contain the following sets of artifacts: structural model, functional model, behavioral model, non-functional requirements specification, quality documentation and decision model. The first three artifacts encompass the functional requirements of the component by capturing distinct aspects of the component properties and behavior. The remaining artifacts describe information related to quality and product line engineering and are out of the scope of this paper.

The structural model is composed of UML class and object diagrams that describe the structure of a component that is visible from its interface. Also, it presents classes and relationships by which the component interact with its environment. On the other hand, the functional model describes the visible effects of the services supplied by the component by a set of operation specifications following a standard template. Finally, the behavioral model shows how the component behaves in response to stimuli. This is specified by behavioral state machine diagrams and tables.

2.3. BIT

The correct functioning of a component-based system is dependent on the correct interaction between components. This is usually specified by contracts between components. A contract is a set of rules that governs the interaction of a pair of components and characterizes the relationships between a component and its clients, expressing each part's rights and obligations [Meyer 1997].

Assembling reusable components to build applications requires validation at deployment time of the interfaces and contracts between these components: the ones that requires some services (clients) and the ones that provide the services (servers). For instance, a client can only properly deliver its services if the components it depends on fulfill their contracts. To address this problem, [Atkinson and Gross 2002] proposed the Built-In contract Testing (BIT) method integrated with KobrA where the behavior of the servers are made explicit through a specific embedded testing interface. Using this interface, the clients are able to verify if the servers fulfill their contracts. Contracts are basically the component specifications produced by the KobrA methodology.

This testing method is an approach for reducing manual effort to system verification within component-based development at the moment the components are integrated. The components are generally implemented and tested without the concern of environment faults. By equipping components with the ability to check their execution environments, it is possible to test the client-server integration at runtime.

According to the BIT method, component specification and realization artifacts are augmented by test artifacts. These artifacts consists in testing components and interfaces to be embedded in a client component. Testing components are composed of methods that define and support the execution of test cases.

Following this idea, it is necessary to create a testing interface at the server side, based on the behavior state machine diagram, that is, according to the KobrA methodology, the main artifact that models the behavior of the component. This testing interface is quite simple, consisting on two different types of operations: the first one, sets the component to a specific desired state, whereas the second returns a boolean that informs if

the current state of the component is the one desired or not. This interface is accessible through a testing component that is a server's subclass, generally named *Testable Server*.

Using the servers testing interface, the client implements test cases that access this interface and checks the conformity between required and provided services. These test cases are held on a testing component on the client side, generally named *Server Tester*. The execution of the test cases are also responsibility of the client. This is defined in the *Testing Client* component.

To assemble the components, there is a specific component called *Context* that is responsible for the configuration task, creating the instances of the components and passing the references to the components that depend on other components.

This paper presents a solution to the automatic generation of these testing components: the *Testable Server*, the *Server Tester* and the *Testing Client*, from the structural and behavioral specifications of each server component following the KobrA methodology. The generation of the test cases inside the *Server Tester* are also addressed but are out of the scope of this paper.

2.4. MDT

Model-driven Testing (MDT) [Heckel and Lohmann 2003] is a MBT approach, based on MDE practices, in which test artifacts are automatically generated according to predefined transformation rules that derive, from development models, test cases and their test oracles and the structure of the necessary execution support for different platforms. Development models are required to be abstract and independent of platform.

MBT test cases have been traditionally derived from development artifacts. But this has proven to be ineffective in practice since such artifacts are usually incomplete, particularly concerning information that is vital for testing such as alternative cases, context dependency and constraints. In this sense, MDE practices can play an important role (in addition to the ones already mentioned): the necessary test artifacts can be automatically identified and generated from development artifacts, making it easier to pinpoint missing information.

In general, UML is the central modelling notation for current MDT approaches. Test cases are usually defined as sequences of activation of the system under test that are derived from behavioral models such as state diagrams. Oracles - responsible for deciding whether a test has passed or failed - are decision procedures that are usually based on constraints such as pre- and post-conditions and invariants, possibly written in OCL. Context information is usually identified from class diagrams and method functional specification.

The outcome of a MDT process is a set of testing components that implement drivers for the selected test cases along with all execution support that is needed.

3. MoBIT Architecture

MoBIT is a CASE tool for MDT that was itself built using MDE. It consists in model transformations specified and executed in the Atlas Transformation Language (ATL) [AMMA Project 2005] applied on component models, such as KobrA models instantiating the OMG meta-models of UML 2 [Object Management Group 2006] and OCL 2 [Object Management Group 2006].

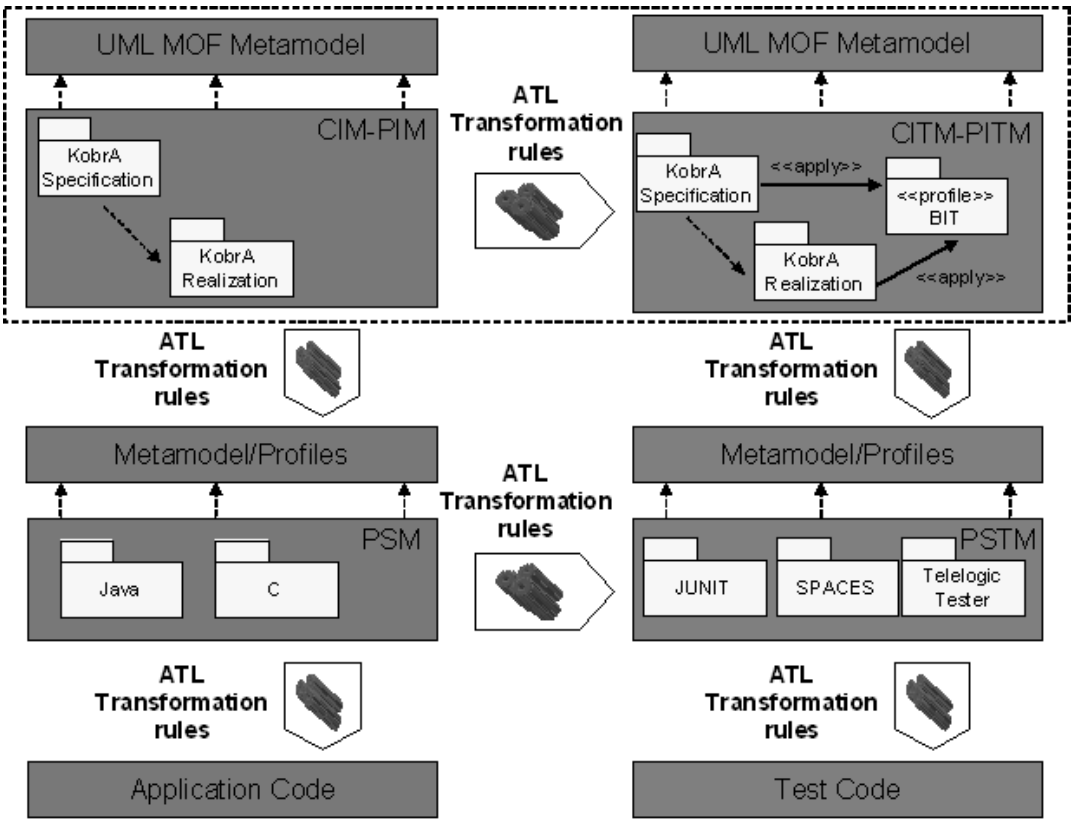


Figure 1. MoBIT Architecture

The internal architecture of MoBIT is shown in Figure 1. It consists of a set of modules containing models, meta-models and profiles. At the top level, there are two modules concerning the CIM-PIM and CITM-PITM. The CIM-PIM are computational and platform independent KobrA models – specification and realization – that do not include information either about the platform on which their components are implemented or about the testing artifacts built to annotate these models. These testing artifacts are present at CITM-PITM that refine CIM-PIM by adding elements according to the BIT method. In fact, the CITM-PITM are KobrA components fully annotated with stereotypes and UML elements defined in the BIT profile and UML metamodel, respectively.

At the middle level there are two modules concerning the PSM and the PSTM. The former includes details about the specific platform on which the KobrA application is implemented, whereas the latter refines the former (together the CITM-PITM) with details about the testing specific platform. The bottom level includes the component code and its testing code.

As shown in Figure 1, there is a separation of concerns in the MoBIT architecture. On the one hand, at the left side of the figure (CIM-PIM, PSM and application code), we find the modules containing models concerning the KobrA application and its implementation independently of any testing approach to be pursued during the development process of the components. On the other hand, at the right side of the figure (CITM-PITM, PSTM and test code), we find the modules containing models concern-

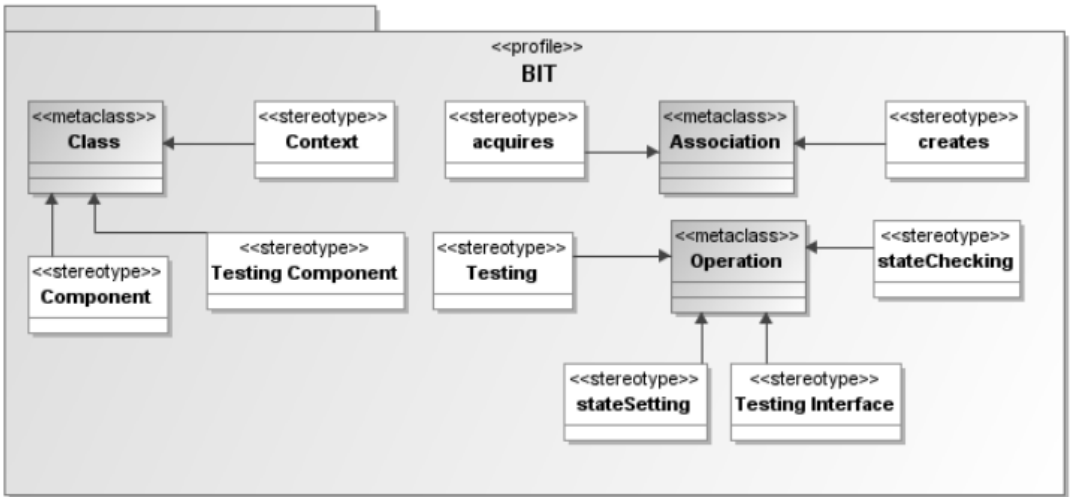


Figure 2. BIT Profile

ing the testing method, artifacts and platform to be incorporated and pursued during the development process of the components.

It is important to highlight that following a MDE approach, transformations may be defined between any pair of the MoBIT modules. This paper is focused on the current state of the work, consisting of transformations from CIM-PIM to CITM-PITM modules, emphasized by the elements inside of the dotted rectangle of Figure 1. We illustrate and discuss them in the next sections.

4. BIT Profile

UML profiles are mechanisms to extend UML with specific domain semantics, allowing any metaclass of the UML specification [Object Management Group 2006] to be annotated with stereotypes representing a specific concept of the domain. In order to formalize the BIT concepts and make the transformation from KobrA diagrams to KobrA diagrams annotated with BIT elements possible, we propose a UML 2 profile, that defines a set of stereotypes which identifies BIT elements inside KobrA artifacts.

In this profile, we specify each element prescribed by the BIT method as stereotypes to be applied to KobrA diagrams. The set of stereotypes and the UML 2 metaclasses that may be extended with them can be seen in Figure 2. The semantics associated with each stereotype is briefly explained below.

- $<< Context >>$: The class responsible for creating both client and server instances and configuring the connection between them, passing the reference of the server to the client.
- $<< Component >>$: A component of the system, responsible for performing functionalities of the system. Can be either a client (requiring a service) or a server (offering a service) or even both (a client of a component but, at the same time, server of another).
- $<< TestingComponent >>$: A class created specifically with testing purposes, responsible for testing activities. On the client side, this is the component that contains the test cases to check the compatibility with the server and/or executes

the test. On the server side, this is the component that makes public the internal states of the server through an specific interface.

- $<< acquires >>$: An association between client and server, where the client receives a reference of the server on configuration time.
- $<< creates >>$: An association between the context and a client or a server, indicating the creation of an instance of a component by a context.
- $<< Testing >>$: Operations with specific testing proposes. It annotates all operations of the *Testable Server* (see Section 2.3).
- $<< TestingInterface >>$: Operations that is part of an interface with testing proposes. This stereotype is applied on the operations of the *Testable Server*. These are public operations that deals with the existing states of the server (defined in the server's state machine diagram): they can change the states or just inform the current state of the server.
- $<< stateSetting >>$: An operation responsible for setting the component to a specific state.
- $<< stateChecking >>$: An operation responsible for checking if the component is in a specific state, returning true or false.

As it can be seen in Figure 2, the three first stereotypes must be applied to the *Class* metaclass since this is the UML metaelement used to model components in KobrA models. The four last ones must be applied to the *Operation* metaclass, identifying the testing methods that form the testing interface of the *Testable Server* element and the remaining ones must be applied to the *Association* metaclass, annotating the relationships between the components. An example of use of this profile can be seen in the case study, detailed in Section 6.

5. Transformation Rules

Although QVT [Object Management Group 2006] is the current OMG's purpose for specifying transformations in the MDE vision, we have adopted ATL for specifying the MoBIT transformations. While QVT tools have still low robustness and its use is not widely disseminated, ATL has a framework widely used by an increasing and enthusiast community, with full support to model operations through a plug-in developed on the Eclipse [Budinsky et al. 2003] framework, where, in an integrated way, one can specify and instantiate metamodels as well as specify and execute transformations on them. In addition, ATL has a wide set of transformation examples available at the literature and discussion lists, in contrast of QVT, whose documentation is poor and not didactic.

The transformations from the CIM-PIM patterns to the CITM-PITM patterns are implemented as ATL matched rules. In essence, these are rewrite rules that match input model patterns expressed as UML elements on the input metamodel and produce output patterns, also expressed as UML elements with a few additional, ATL-proper imperative constructs that cut and paste the matched input model elements in new configurations in conformance to the output metamodel.

Due to lack of space, we show below only an excerpt from one MoBIT ATL transformation rule named CIMPIM2CITMPITM (Figure 3). This rule specifies how to transform a pure KobrA application into one fully refined with testing artifacts conformant to the BIT method. Lines 1-2 defines the name of the ATL transformation module (one ATL module may contain one or more ATL rules) and the input and output model variables,

respectively. Line 3 defines the rule named CIMPIM2CITMPITM whose specification is between lines 4-50. Lines 4-6 state that the Class element (annotated with the stereotype $<< Context >>$) is that one from the source UML meta-model to be transformed, whereas lines 10-50 specify which UML elements have to be transformed to. This rule creates several new UML elements, such as one subclass for the client component (lines 11-16), a tester component for testing the server component (lines 30-33) and one association outgoing from the former and targeting the latter, annotated with the stereotype $<< acquires >>$ (lines 41-49). In addition, all remaining testing artifacts prescribed by the BIT method are generated during the transformation. For instance, the generated server tester component (lines 30-33) must have an operation named 'executeTest' (lines 34-37) and be extended with the stereotype $<< TestingComponent >>$ (lines 38-40).

6. Case Study

An illustrative KobrA specification structural model is shown in Figure 4. It is a simplified and extended version of the library system (LS) presented in [Atkinson et al. 2002]. LS manages enrolled users and available items for loaning as well as records all performed loans, returns and reservations executed in a library environment. In this model we find some prescribed KobrA artifacts whose semantics is defined as follows.

In Figure 4, we focus on the structural model of the LoanManager class representing the component under specification, whose offered services range the management and record of item loans. In order to offer these services, the LoanManager requires a set of other services. Therefore, it plays dual role in the system: (1) as server, offering to the Library component a set of services identified by some of its operations, such as loanItem(), reserveItem() and printLoanInformation(); (2) as client, requiring a set of other services, such as printing services offered by the ReportWriter component or item querying services offered by the ItemManager component.

The creation and integration of each pair of client-server components is executed by the Assembler component. This component is responsible for creating the component under specification (LoanManager) and its client (Library). Moreover, it configures all server components in its clients, integrating each pair of client-server components.

The behavior of the LoanManager component is partially illustrated in Figure 5. It is an extension of that presented in [Atkinson et al. 2002] to the LS. In this figure, we show: (1) the behavioral state machine describing the major states of the LoanManager; (2) the operations that can be executed in each one of theses states; and (3) the events that lead to state transitions. Starting with the initial state, control passes to the unconfigured state. Then, the state of the LoanManager changes from the unconfigured to neutral when the Assembler component configures and integrates it with the remaining ones. The LoanManager remains on this state until the user account is identified, when the control passes to the accountIdentified state. On entering the accountIdentified state, any of the actions contained in this state may be executed by the LoanManager component. Then, whenever the user account is closed, there is a transition back to the neutral state.

From these two models we are able to fully apply the BIT method to the LS application. In Figure 4, the classes filled in grey color and its elements as well as the associations involving them are generated by applying the BIT method on the aforementioned two models.

```
1 module UML2BICT;
2 create OUT : UML2 from IN : UML2;
3 rule CIMPIM2CITMPITM{
4    from cont:UML2!Class(
5        cont.extension
6            ->exists(e | e.ownedExtension.stereotype.name = 'Context') )
7    using{
8        server : UML2!Class =  …;
9        client : UML2!Class …; }
10   to
11       subc:UML2!Class(
12          superClass <- Set{client},
13          generalization <- genc,
14          name <- 'Testing' + client.name,
15          ownedOperation <- oo,
16          extension <- cExt),
17       genc:UML2!Generalization(
18          specific <- subc,
19          general <- client),
20       oo:UML2!Operation(
21          name <- 'setTester',
22          visibility <- #public,
23          ownedParameter <- param ),
24       param:UML2!Parameter(
25          name <- 'tester',
26          type <- sTester),
27       cExt:UML2!Extension(ownedExtension <- cExtEnd),
28       cExtEnd:UML2!ExtensionEnd(type <- cSte),
29       cSte:UML2!Stereotype(name <- 'TestingComponent'),
30       sTester:UML2!Class(
31          name <- server.name +  'Tester',
32          ownedOperation <- ooTester,
33          extension <- cExtTester),
34       ooTester:UML2!Operation(
35          name <- 'executeTest',
36          visibility <- #public,
37          ownedParameter <- param),
38       cExtTester:UML2!Extension( ownedExtension <- coeTester  ),
39       coeTester:UML2!ExtensionEnd(type <- ct),
40       ct:UML2!Stereotype(name <- 'TestingComponent'),
41       assoc1: UML2!Association(
42          extension <- aExt,
43          memberEnd <- Set{prop11, prop12},
44          navigableOwnedEnd <- Set{prop12}),
45       prop11: UML2!Property(type <- subc),
46       prop12: UML2!Property(type <- sTester),
47       aExt:UML2!Extension(ownedExtension <- aExtEnd),
48        aExtEnd:UML2!ExtensionEnd(type <- aSte),
49        aSte:UML2!Stereotype(name <- 'acquires')
50 }
```

Figure 3. CIMPIM2CITMPITM Transformation Rule

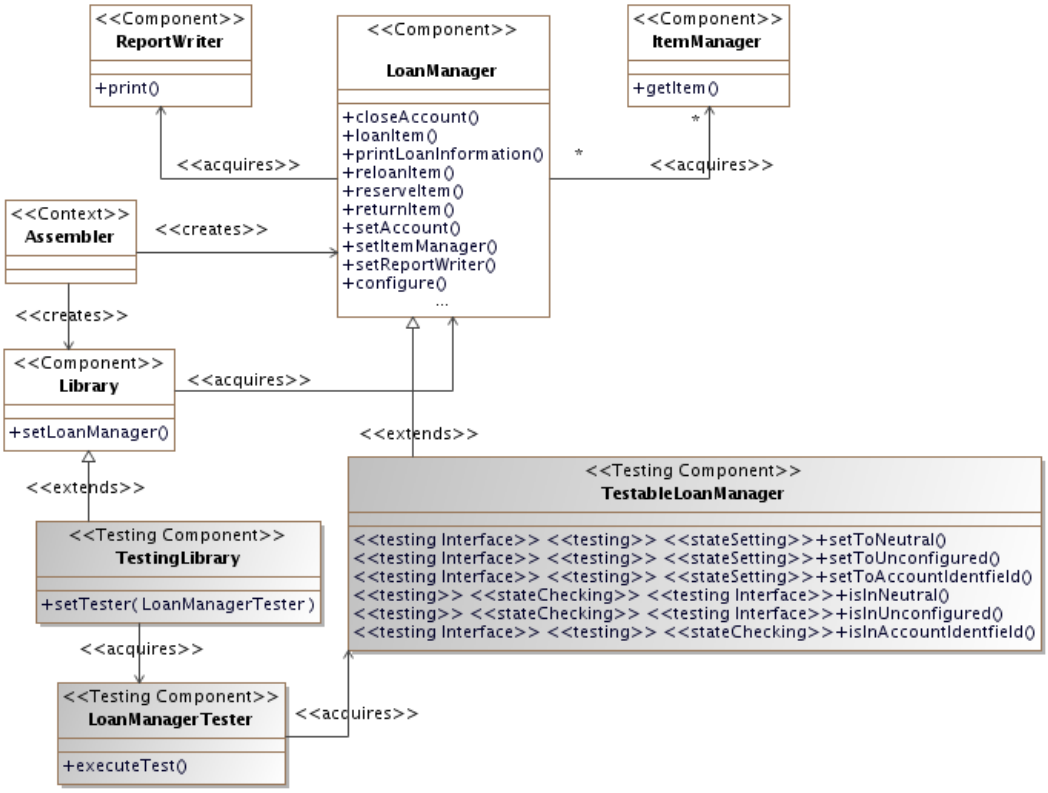


Figure 4. LoanManager Structural Model

The MoBIT tool through its architecture, metamodels, profiles and transformation rules is able to automatically incorporate these artifacts to the original KobrA structural model, *i.e.*, automatically generates the elements filled in grey color from those unfilled. For instance, the MoBIT automatically generates the filled elements by applying the CIMPIM2CITMPITM rule shown in Section 5 as follows:

- Lines 11-16 and 27-29 generate the TestingLibrary component;
- Lines 17-19 generate the hierarchy between Library and the TestingLibrary components;
- Lines 20-26 generate the operation setTester() inside the TestingLibrary component;
- Lines 30-33 and 38-40 generate the LoanManagerTester component;
- Lines 34-37 generate the operation executeTest() inside LoanManagerTester;
- Lines 41-49 generate the association between TestingLibrary and LoanManagerTester components.

The TestableLoanManager component and its state checking and state setting operations deriving from the behavioral state machine shown in Figure 5 as well as its relationships are also automatically generated by the MoBIT tool. However, due to the lack of space, we do not illustrate in this paper the transformation rule concerning this refinement.

Although our case study covers only one client-server pair, the tool is able to generate built-in testers for any number of pairs since: (1) a state machine is specified to

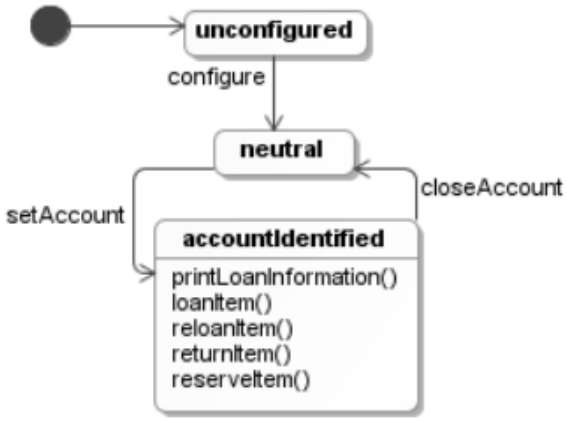


Figure 5. LoanManager Behavioral State Machine Diagram

the server; (2) a *creates* relationship is specified from the client to the *Context* class; (3) a *creates* relationship is specified from the server to the *Context* class; and (4) an *acquires* relationship is specified from the client to the server.

7. Related Works

A number of attempts have already been made towards the theory and practice of MDT. For instance, [Berger et al. 1997] discusses the use of MDT for automatic test case generation and briefly describes a general environment defined at DNA Enterprises that is generally based on the use of existent tools. The underlying MDT concepts are preliminary and no details are given, for instance, concerning the models used. Also, [Heckel and Lohmann 2003] focuses on models and processes for testing distributed components. Even though MDE is taken into account, automation through transformation rules is not considered as well as the use of MDT is only motivated. Moreover, [Engels et al. 2006] presents a testing process based on models and design by contract using the JML language for unit testing. Although automation is considered, this is not based on the use of the MDE technology.

A definite step towards the practice of MDT has been made by OMG by defining the UML 2.0 Testing Profile (U2TP). However, this is not directly integrated with methodologies such as KobrA and BIT since the concepts handled may differ, mainly regarding component based concepts and testing artifacts. The BIT profile presented in this work is a crucial step to make automatic transformation from KobrA models following the BIT method to the U2TP format.

Some approaches have been developed towards MDT based on U2TP. [Born et al. 2004] proposes a MDT approach for telecommunication domains. For instance, they propose a metamodel for validating specific telecommunication domain models on which they work. However, they do not adopt a full MDE architecture as that proposed in our work. In particular, they do not adopt any transformation definition language (such as QVT or ATL) nor explicitly refer to any MDE models (CIM, PIM or PSM). Moreover, although they have adopted the KobrA methodology, the testing artifacts are

manually modeled in separation from the development models.

Another U2TP based approach is presented by [Dueñas et al. 2004]. They propose a metamodel for software testing of product families based on the U2TP profile. The focus is on state diagrams from which test cases are devised.

Finally, [Dai 2004] presents a methodology for applying the U2TP profile to both development and testing models. The general ideas are very similar to ours. However, it is not mentioned in the paper whether the methodology and, particularly, transformation rules have been developed and experimented. Also, the focus is on object-oriented applications, not dealing with the additional complexity of component based systems, *i.e.*, its approach is not eligible for KobrA applications.

8. Concluding Remarks

In this paper, we overviewed MoBIT, a testing tool that automatically generates KobrA components fully annotated with BIT artifacts from pure KobrA component structural and behavioral models that consists of UML class and behavioral state machine diagrams. Because KobrA and BIT are independent of a specific platform and are fully towards MDE guidelines, MoBIT supports the synergetic synthesis of principles from two software engineering approaches, MDE and MDT. In addition, MoBIT was built by pursuing a MDE approach itself. The expected benefits of MoBIT by pursuing these approaches are twofold: (1) to lower the cost of deploying a given component and corresponding testers on multiple platforms through reuse of CIM-PIM and CITM-PITM; and (2) to reach a higher level of automation in the development and test case generation processes, through specification of transformations among those standard models.

The current implementation of MoBIT consists of several ATL rules that cover 100% of UML class properties and behavioral state machines that have a direct translation as BIT artifacts on the KobrA specification models according to the BIT method. As a result, the tool is an Eclipse plugin, and the user can perform the transformation (from models created in the ATL-DT tool itself or imported as a XMI file from another tool) through an intuitive interface. Since XMI is the OMG standard for exchanging metadata information, the integration with other tools, including other test case generation tools, just requires from these ones the support to XMI.

Although KobrA is fully defined towards MDE principles, its current version remains on previous version of UML instead of the current UML 2. However, in MoBIT, we are handling UML 2 with no onus because particularly the UML class and behavioral state machines artifacts prescribed by BIT are in conformance between these two UML versions.

The proposed BIT profile and the MoBIT transformations may be used together with any other methodology than KobrA. To achieve this, the adopted methodology must only prescribe the artifacts required by the BIT method.

In future work, we plan to extend MoBIT to cover all UML diagrams and OCL expressions. We also intend to systematically support each pair of module translations present in the MoBIT architecture: CIM-PIM to PSM, CITM-PITM to PSTM, PSM to PSTM, PSM to application code, PSTM to test code, and application code to test code. In addition, we will study how ATL transformation rules involved in the functional vertical

transformations (CIM-PIM, PSM, application code) may be reused in the test vertical transformations (CITM-PITM, PSTM, test code).

References

AMMA Project (2005). Atlas transformation language. http://www.sciences.univ-nantes.fr/lina/atl/.

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J. (2002). *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley.

Atkinson, C. and Gross, H.-G. (2002). Built-in contract testing in model-driven, component-based development. In *ICSR Work. on Component-Based Develop. Processes*.

Barbosa, D. L., Lima, H. S., Machado, P. D. L., Figueiredo, J. C. A., Juca, M. A., and Andrade, W. L. (2007). Automating functional testing of components from uml specifications. *Int. Journal of Software Eng. and Knowledge Engineering*. To appear.

Berger, B., Abuelbassal, M., and Hossain, M. (1997). Model driven testing. Technical report, DNA Enterprises, Inc.

Born, M., Schieferdecker, I., Gross, H.-G., and Santos, P. (2004). Model-driven development and testing - a case study. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, pages 97–104, University of Twente.

Briand, L. C., Labiche, Y., and Sówka, M. M. (2006). Automated, contract-based user testing of commercial-off-the-shelf components. In *ICSE '06: Proc. of the 28th Int. Conference on Software Engineering*, pages 92–101, New York, NY, USA. ACM Press.

Budinsky, F., Brodsky, S. A., and Merks, E. (2003). *Eclipse Modeling Framework*. Pearson Education.

Dai, Z. R. (2004). Model-driven testing with uml 2.0. In *Second European Workshop on Model Driven Architecture (MDA) with an Emphasis on Methodologies and Transformations*, Canterbury, England.

Dueñas, J. C., Mellado, J., Cerón, R., Arciniega, J. L. ., Ruiz, J. L., and Capilla, R. (2004). Model driven testing in product family context. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, University of Twente.

El-Far, I. K. and Whittaker, J. A. (2001). Model-based software testing. *Encyclopedia on Software Engineering*.

Engels, G., Güldali, B., and Lohmann, M. (2006). Towards model-driven unit testing. In *Workshops and Symposia at MoDELS 2006*, volume 4364 of *LNCS*.

Heckel, R. and Lohmann, M. (2003). Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 82(6).

Meyer, B. (1997). *Object-oriented Software Construction*. Prentice Hall.

Object Management Group (2006). Catalog of omg modeling and metadata specifications. http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

Towards a Maturity Model for a Reuse Incremental Adoption

Vinicius Cardoso Garcia[1], **Daniel Lucrédio**[2], **Alexandre Alvaro**[1], **Eduardo Santana de Almeida**[1], **Renata Pontin de Mattos Fortes**[2], **Silvio Romero de Lemos Meira**[1]

[1] Informatics Center – Federal University of Pernambuco
C.E.S.A.R – Recife Center for Advanced Studies and Systems
Recife, Pernambuco

{vcg,aa2,esa2,srlm}@cin.ufpe.br

[2]Institute of Mathematical and Computing Sciences
São Paulo University, São Paulo, Brazil

{lucredio,renata}@icmc.usp.br

Abstract. *Software has been reused in applications development ever since programming started. However, the reuse practices have mostly been ad hoc, and the potential benefits of reuse have never been fully realized. Systematic reuse offers the greatest potential for significant gains in software development productivity and quality. Organizations are looking for ways to develop a software reuse program. The strategy for adopting a reuse technology should be based on a vision for improving the organization's way of doing business. Thus, this paper presents a Reuse Maturity Model proposal, describing consistence features for the incremental reuse adoption.*

1. Introduction

As has been frequently discussed [Biggerstaff and Richter 1987, Frakes and Isoda 1994, Lim 1994, Rine and Nada 2000b, Poulin 2006], the practice of software development has become increasingly worried by high cost, poor productivity, and poor or inconsistent quality. One of the reasons for this problem is the insistence on the part of many software development organizations to develop from the ground up similar systems over and over again, rather than to treat their previous experiences and previously-developed systems as assets to be captured, created, and evolved so that they can contribute directly to future development activities [Rine and Nada 2000b].

Software reuse, the use of existing software artifacts or knowledge to create new software, is a key method for significantly improving software quality and productivity [Frakes and Isoda 1994]. Thus, reuse has been advocated as a means of revolutionizing the development process.

Although reusability is a big challenge on software development area, its promise has been largely unfulfilled. The main inhibiting factors have been the absence of a clear reusability strategy and the lack of specific top-management support, which can lead to resistance from project managers and programmers [Biggerstaff and Richter 1987, Frakes and Fox 1995, Moore 2001, Morisio et al. 2002, Rine 1997a].

A reuse adoption model helps an organization to understand how reuse will change the way it does business, and how it should plan for that change [Wartik and Davis 1999].

The recent interest in characterizing reuse with maturity models and adoption processes is a clear sign of progress toward making reuse a natural part of development. Reuse maturity models provide the necessary framework for the development of tools and methods to aid in the reuse adoption or to support in the organization development process [Griss 1994, Prieto-Díaz 1993, Rine and Nada 2000a].

In this context, this paper describes the initial RiSE[1] Maturity Model, which provides a structured set of reuse engineering, management and support practices related to the implementation of a software reuse program in an organization.

This paper gives an overview of the RiSE Maturity Model, which has been developed within the RiSE project [Almeida et al. 2004]. More precisely, this paper describes: **(i)** other reuse-related maturity models, which have been studied as background information; **(ii)** the approach taken to develop the Maturity Model; and **(iii)** the reuse maturity levels inside the Maturity Model.

The remainder of the paper is organized as follows: Section 2 present the problem related with reuse adoption by software development organizations. Section 3 discusses related work. The proposed approach is described in details in Section 4. Finally, the concluding remarks and future work are presented in Section 5.

2. The problem

Many software development organizations believe that investing in software reuse will improve their process productivity and product quality, and are planning or developing a reuse program [Frakes and Isoda 1994, Tracz 1990]. Unfortunately, there is still not enough data available on the state-of-the practice of utilizing or managing software reuse. The majority of current information available on software reuse models comes from the literature [Card and Comer 1994, Frakes and Fox 1995, Rine and Sonnemann 1998, Rine and Nada 2000b]. After 3 years of experience in industrial reuse projects, the RiSE initiative identified that a critical problem in today's practice of reuse is a failure to develop necessary details to support valid software reuse models. The existing models, including many important and well established works, as described in section 3, do not present such details, which makes their practical usage difficult.

Another problem is related to the question: *what kind of assets can be reusable?* Some researchs [Frakes and Fox 1995, Morisio et al. 2002, Rine 1997a] show that software reuse can happen in others phases of the development cycle, such as analysis and project, obtaining more benefits than source code level. Moreover, high level abstractions can be useful for analysts and designers allowing that previous experiences and knowledge can be reused reducing the probability of risks.

Our research, reported in this paper, investigated the success factors for software reuse, the way they impact in a software reuse program, and how they can be used to construct a framework for a reuse maturity model. This work is based on empirical data collected in a survey that investigated the reuse situation in several organizations in Brazil [Lucrédio et al. 2007], which covered many factors related to software reuse.

[1]Reuse in Software Engineering Group, `http://www.rise.com.br`

3. Reuse Adoption Models: A Brief Survey

Some software reuse models have been proposed to face the reuse adoption problem. The vast majority of reuse models attempt to provide some measure of a reuse program success in a systematic way. In this section, we present a brief survey on these approaches.

Holibaugh et al. [Holibaugh et al. 1989] presented the first cost model for software reuse developed at *Software Engineering Institute* (SEI). In this work, Holibaugh et al. described a framework to determine a cost-effective approach to start the reuse adoption. For this framework, it can be noticed: a clear evolution; fulfillment of existent gaps; and a natural ripening in the reuse adoption models. Phases and activities, such as analysis and domain engineering, as well as a well defined software development methodology are considered as a primordial condition for a systematic reuse process in an organization.

In 1991, Koltun and Hudson [Koltun and Hudson 1991] presented the first version of the Reuse Maturity Model (RMM). The model was specified through workshops accomplished by the *Software Productivity Consortium* (SPC). The model, in fact, provides a concise form of obtaining information on reuse practices in organizations. The model is composed of five levels and ten dimensions or aspects of reuse maturity were also enumerated. The main obstacles to reach the highest reuse levels, as pointed out by Koltun and Hudson, were: Cultural, Institutional, Financial, Technical and Legal. This model was not applied in real case studies but will be considered as the main insight for the Margaret Davis work [Davis 1992].

Starting from his experience and the best features of four success cases, Prieto-Díaz defined an incremental model for the implementation of software reuse programs [Prieto-Díaz 1991]. The approach was practical, effective and with potential to turn reuse into a regular practice in the software development process. Prieto-Díaz also points out the support of the high administrative management as one of the main factors for the success in reuse programs, because those programs demanded changes in the way software is developed.

One of the most interesting features in Prieto-Díaz's model is the definition of an essential organizational structure, in the first stage of the model implementation, containing teams for: assets management; assets identification and qualification; assets maintenance; assets development; support to the reuse program through consultancy and training; and classification of the assets in the repository system. These teams perform the basic roles in the reuse program.

In November 1992, the *Fifth Workshop on Institutionalizing Software Reuse* (WISR) was held. Margaret Davis presented the reuse maturity model of the STARS project [Davis 1992]. The first reuse maturity model, presented by Koltun and Hudson had important influence in this one, because Hudson participated directly in the STARS project. This maturity model is the base for organizations to formulate their short and long term strategies to improve the level of reuse practice in their business domains. Moreover, Margaret Davis believes that the maturity model can be used with other principles, such as a reuse practice level evaluation tool, or a way to encourage the reuse adoption through incentive programs.

The mains issues in Margaret Davis model is the high up-front risk of reuse adoption, because a significant initial investment is needed. However, one positive point is that

the model was designed to be independent of a particular development model.

In the next year, Ted Davis [Davis 1993] presented the *Reuse Capability Model* (RCM), an evolution of the STARS' reuse maturity model. RCM aids in the evaluation and planning for improvements in the organization's reuse capability. RCM is used together with the reuse adoption process defined by SPC [SPC 1993]. The reuse adoption process is a solution to implement a reuse program and it is based on the implementation model defined by Prieto-Díaz [Prieto-Díaz 1991].

The RCM has two components: an assessment model and an implementation model. The **assessment model** consists of a set of critical success factors to assess the present state of its reuse practice. From this assessment, the organization will get a list of its strengths and a list of potential improvement opportunities. The **implementation model** helps in prioritizing the critical success factor goals by partitioning them into a set of stages. This model will be considered as the main insight for the Wartik and Davis [Wartik and Davis 1999] work.

The utilization of a specific reuse adoption process is a great issue of Ted Davis work. RCM, in conjunction with this adoption process, helps the organization in business decisions and in how its practices work in its development process. But, according to Rine and Sonnemann, in their study of software reuse investment success factors [Rine and Sonnemann 1998], RCM proved to be unstable. It is a hierarchical model with each level building on the previous level. It produced substantially different results depending on whether or not the level questions were answered bottom up or top down. However, the six measurements used to evaluate software reuse capability did not correlate very well.

Not directly related of reuse adoption model we can notice the standard ISO/IEC 12207, addressed to aiming the organizations in their software development process. The standard ISO/IEC 12207 - Software Life-Cycle Process [ISO/IEC 1998] offers a framework for software life-cycle processes from concept through retirement. It is especially suitable for acquisitions because it recognizes the distinct roles of acquirer and supplier. ISO/IEC 12207 provides a structure of processes using mutually accepted terminology, rather than dictating a particular life-cycle model or software development method. Since it is a relatively high-level document, 12207 does not specify the details of how to perform the activities and tasks comprising the processes.

Wartik and Davis [Wartik and Davis 1999] present a new version of the reuse adoption model of SPC [Davis 1993]. The model is based on a set of phases that help the organization to measure its progress towards the reuse adoption. Each phase has specific goals, integrated into the Synthesis methodology [Burkhard 1993]. Besides, the phases are spent in such a form that the organization can avoid the risks, or at least, to reduce them significantly in the reuse adoption program through the selection of the main features of each phase to achieve their goals.

The main motivations for a new model, according to the lessons learned after the observation of the organizations in several stages in the reuse adoption process were: **reduce the initial risk in the reuse adoption**, so that organizations can recognize the need to define the reuse goals, making it easier to commit with a reuse adoption model that demands a commitment of resources incrementally with base in a constant evolution and

understanding of their benefits; and, **integration with a reuse-based software development process**, merging the reuse adoption model with the Synthesis methodology of SPC, reducing the number of decisions related to reuse that the organization should take, making the efforts for the adoption simpler.

Another reference model for software reuse called Reuse Reference Model (RRM) was presented by Rine and Nada [Rine and Nada 2000a]. RRM incorporates both technical and organizational elements that can be applied to establish a successful practice of software reuse in the organization. The technical elements consist of technologies that support reuse, such as *CASE* tools and a software reuse process, among others. Organizational elements include management of the reuse program, market analysis, financing and training.

In order to increase their software development capabilities, Brazilian software industries and research universities are working cooperatively to implement a strategy aiming to improve software processes of Small and Medium-size Enterprises Brazilian organizations since 2003. The main goal of this initiative is to develop and disseminate a Brazilian software process model (named MPS Model) [SOFTEX 2007] based on software engineering best practices and aligned to Brazilian software industry realities. The focus of the MPS Model is on small settings, since it provides mechanisms to facilitate software process improvement (SPI) implementation of the most critical software processes. The adequate implementation of such processes promotes subsequent SPI implementation cycles and software process maturity growth.

However, such as the ISO/IEC 12207, MPS does not have some practices related to reuse activities or to aiming to adopt reuse practices in the software development process. Therefore, we believe that can be possible for some reuse practices to be integrated to these models, specially to ISO/IEC 12207 (the basis of MPS and CMMI [Chrissis et al. 2004]) to specify a new strategy to help in the improvement of productivity and quality in the software development process in organizations.

Based on the research results and case studies, Rine and Nada conclude that the level of reuse, as defined in RRM, determines the capability of improvements in the productivity, quality and time-to-market of the organization.

3.1. Discussion

From this brief survey we may notice that it is clear that development *"for"* (domain engineering) and *"with"* (application engineering) reuse are needed. Another factor is the concern with reuse since early stages of the software life cycle, beginning with the elaboration of the business plan. Reuse should be considered in the first phases of the software development cycle.

The reuse community agrees [Davis 1993, Rine and Nada 2000a, SPC 1993, Wartik and Davis 1999] that characterizing reuse with maturity models and adoption processes is a clear sign of progress toward making reuse a natural part of development. As shown in this section, there are some experiences and projects involving reuse adoption models and programs. However, the organizations are still unsure of adopting a reuse program because there is not a widely accepted model. Models proposed until now fail in transferring the reuse technology to the entire organization, in an incremental and systematic way. The main problem is that most works classify the reuse adoption as an atomic

initiative, and introducing atomic, radical changes is not appealing to most organizations. A more gradual evolution is more suited to reuse [Rine and Nada 2000b]. Thus, through this survey we identify the main requirements of some reuse adoption models to specify an effective reuse maturity model, in order to implement a systematic and incremental approach to reduce the risks and improve the possibilities of success in this journey. This model will be presented in the next section.

4. Towards a RiSE Maturity Model

The RiSE Maturity Model was developed during the RiSE project [Almeida et al. 2004] through discussions with industry practitioners. Initially, the model included four perspectives addressing organizational, business, technological and process issues. The idea is to provide the organizations the possibilities to develop and improve these perspectives separately. However, our experience in the RiSE project[2] and work reports in the literature showed that the organizational, business and technological perspectives are interrelated, and have to be implemented in parallel, while the processes perspectives are relevant for highly mature organizations with respect to software reuse program adoption. Therefore, the individual perspective are combined in the final version of the maturity model and the activities related to their implementation are defined in terms of specific practices.

4.1. Model Structure

The main purpose of the RiSE Maturity Model is to support organizations in improving their software development processes. In particular, the model has to serve as a roadmap for software reuse adoption and implementation. It aims at facilitating the improvement of the engineering practices with respect to their effectiveness and efficiency, performing reuse activities properly and achieving the quality goals for the products.

Current models for process improvements like CMMI and ISO 9001:2000 typically address organizational processes. Although they also discuss software engineering activities, they do not provide much insight in the way of performing these tasks, neither discuss how particular technologies could support the improvement initiatives.

The model structure intend to be flexible, modular and adaptable to the needs of the organizations that will use them. Thus, the model was based in two principles: modularity and responsibility. Modularity, in the sense of process with less coupling and maximum cohesion. Responsibility, in the sense of the possibility to establish one or more (team) responsible for each process, or activity (perspectives and factors). This structure easiest the model implementation in places where various professionals can be involved in the reuse adoption.

The RiSE Maturity Model includes: **(i)** Reuse practices grouped by perspectives (Organizational, Business, Technological and Processes) [Brito et al. 2006, Lucrédio et al. 2007] and in organized levels representing different degrees of software reuse achieved; and, **(ii)** Reuse elements describing fundamental parts of reuse technology, e.g. assets, documentation, tools and environments.

The maturity levels provide general characterization of the organizations with respect to the degree of reuse adoption and implementation, i.e. a maturity level indicates what reuse practices and assets are expected to be in place in an organization.

[2]The RiSE group has been involved in 4 industrial projects related to reuse adoption since 2004.

The practices implemented at the lower maturity levels are a basis for the implementation of the activities at the upper levels. This means that an organization at a certain level of maturity has to successfully implement all the practices from this level and the ones below it. The same is valid for the reuse elements associated with a level.

The following maturity levels are defined: **Level 1**: Ad-hoc Reuse; **Level 2**: Basic Reuse; **Level 3**: Initial Reuse; **Level 4**: Integrated Reuse; and, **Level 5**: Systematic Reuse. The levels are defined in details in section 4.2. Each level consists of a list of reuse practices and reuse elements associated to them.

Goals are defined for each maturity level. They are used to guide the assessment of the implementation of the maturity model. More precisely, while the execution of the practices defined at a maturity level could vary, the achievement of the goals for that level and the levels bellow is mandatory to conclude that the level is achieved.

4.2. RiSE Maturity Model Level Definitions

The RiSE Maturity Model consists of the following elements: Maturity Levels, Goals assigned to each level, Perspectives (Organizational, Business, Technological and Processes) and Practices grouped in levels and perspectives.

Five levels are identified in the RiSE Maturity Model as shown on Figure 1.

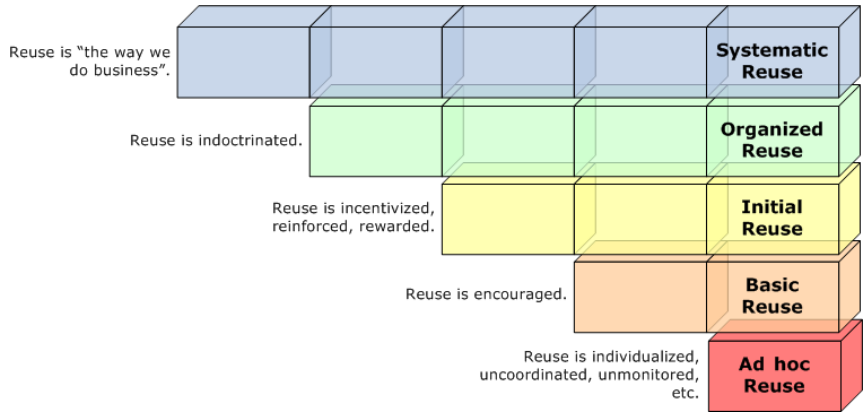


Figure 1. RiSE Maturity Model levels

A maturity level represents the level of reuse adoption. Each level has particular goals associated to it, which are used to determine whether it is completely achieved. The maturity levels also allow foreseeing which reuse-related activities will be performed in an organization in the future. However, an organization does not need to evolve one stage at a time. It may decide its own evolution path, according to strategical and/or business goals, by choosing which factors should be improved first, until the organization reaches the desired reuse capability. The RiSE Maturity levels are presented next.

Level 1: Ad hoc Reuse.

Traditional software development is performed. Reuse practices are sporadically used or not used at all and is discouraged by management. This practices are performed as an individual initiative (personal goal; as time allows). The costs of reuse are unknown.

Level 2: Basic Reuse.

This level is characterized by a basic usage of potentially reusable assets. It encompasses basic reuse-oriented engineering activities. The assets are used for aiding in the implementation and production of components and documentation. Simple tools and techniques are used to develop reusable assets (documents, design, code, etc). Typically, technical assets are developed, which, in this level of maturity, include all the requirements of the system with no distinction between business and domain specific aspects.

Code and documentation are generated by methods of reuse-based tools or available COTS supporting this operation, but not developed by the organizations. This means that the assets reuse is performed by a tool and no particular engineering effort on defining such asset is needed. Developers modify the generated assets manually to complete the implementation of the application.

From the business perspective, the benefits for an organization applying reuse at this level consist in acquiring experience in software and system design reuse.

The following goals and practices are defined at level 2.

- Goal 1: Using reuse best practices to designing and implementing the software product.
- Goal 2: Use the technical assets to build up software (code and documentation).

Level 3: Initial Reuse.

At this level a separation between business and domain-related assets is introduced. The objective is to maintain the implementation issues independent from the business issues in order to increase the efficiency of the development process by reusing assets for projects of a different nature which have similar business requirements. This is essentially important for system families. Additionally, initial steps towards automating the engineering process are made.

Reuse practices are standardized and deployed in the whole organization (institutionalized). Engineering process knowledge is stored in a reuse repository (asset manager) [Koltun and Hudson 1991]. Metrics on reuse activities are also defined and analyzed to ensure their good performance with respect to predefined organization-wide policies.

The key difference between level 2 and level 3 is that at level 2 any project might implement a different approach to reuse provided that the intent of the specified practices is fulfilled. At level 3 projects are getting some reuse guidelines and assets from the organization and then adapting them to the project, following organization's conventions, using them and providing feedback to the organization in terms of lessons learned and suggestions for improving the practice or asset descriptions.

From the business perspective, the gained benefits for an organization applying software reuse when reaching this level consist in establishing the basis of formalizing and maintaining organizational know-how and application domain knowledge. Another benefit is standardizing the way-of-doing of all the projects, making reuse processes more mature in the sense they are formally described, followed by all projects and quality ensured.

The following goals and practices are defined at level 3.

- Goal 1: Separate business-specific aspects.

- Goal 2: Institutionalize reuse practices and assets.
- Goal 3: Use software reuse project's defined process.

Level 4: Organized Reuse.

The Organized Reuse level is characterized by a better integration of all reuse abstractions levels. At the highest abstraction level, reuse is indoctrinated. Staff members know the reuse vocabulary and have reuse expertise. Reuse occurs across all functional areas.

At the level 4, domain engineering is performed. Reuse-based processes are in place to support and encourage reuse and the organization has focus on developing families of products. The organization has all data needed to decide which assets to build/acquire, because it has a reuse inventory organized along application-specific lines.

From the point of view of the processes, in level 4 the reuse practices inside the projects have started to be quantitatively managed. Metrics are consistently collected and analyzed as defined by the organization.

From the business perspective, all costs associated with an asset's development and all savings from its reuse are reported and shared. Additionally, reuse practices and assets progress and quality in all projects are statistically controlled through standardized metrics that leads to a better control and estimates of the projects objectives. In this level, rework efforts are also reduced due to early detection.

The following goals and practices are defined at level 4.

- Goal 1: Enhance the organization competitive advantage.
- Goal 2: Integrate reuse activities in the whole software development process.
- Goal 3: Ensure efficient reuse performance.

Level 5: Systematic Reuse.

In the Systematic Reuse level, the whole organization's knowledge is planned, organized, stored and maintained in a reuse inventory (asset manager), and used with focus in the software development process. All major obstacles to reuse have been removed. All definitions, guidelines, standards are in place, enterprise-wide.

Domain engineering [Almeida 2007] practices are put in place. In fact, all reusable assets and knowledge are continuously validated in order to make strategic assets reusable. All software products are generalized for future reuse. Domain analysis is performed across all product lines. All system utilities, tools, and accounting mechanisms are instrumented to track reuse.

From the reuse inventory perspective, the development process supports a planned activity to acquire or develop missing pieces in the catalog. From the technological perspective, the organization has automated support integrated with development process.

From business perspective the organization maximizes the benefits of having implemented the whole software reuse approach. The organization is able to express its system development know-how in the form of reusable assets and even more, they are the start point for a rapid and automatic production of the implementation. This reduces effort consumption and accelerates time to market. All costs associated to a product line or a particular asset and all savings from its reuse are reported and shared.

The following goals and practices are defined at level 5.

- Goal 1: Reuse is *"the way we do business"*.
- Goal 2: Establish and maintain complete reuse-centric development.

4.3. Perspectives and Factors

After an extensive literature review [Almeida et al. 2005, Frakes and Fox 1995, Morisio et al. 2002, Rine 1997b, Rine 1997a] and from our experience in reuse projects, we identified some factors related to software reuse [Brito et al. 2006, Lucrédio et al. 2007], that were considered as a basis for this maturity model specification, in order to guide the organizations in the reuse evaluation and/or adoption.

Four perspectives are defined in the RiSE Maturity Model: **Organizational**, **Business**, **Technological** and **Processes**.

The Organizational perspective addresses activities that are directly related to management decisions necessary to setup and manage a reuse project. The business perspective addresses issues related to the business domain and market decisions for the organization. The technological perspective covers development activities in the software reuse engineering discipline and factors related to the infrastructure and technological environment. The processes perspective includes only activities that support the implementation of the engineering and the project management practices.

In the RiSE Maturity Model, fifteen factors were considered, divided into these four perspectives. This division is useful for organizations, which can put special focus on different parts of reuse adoption, one perspective at a time. Another possibility is to assign particular teams for each perspective, according to technical skills and experience, so that each group of factors may be dealt with simultaneously by specialized professionals.

Figure 2 shows the factors related to the organizational perspective and their distribution across the RiSE Maturity Model levels. Figure 3 shows the factors related to the business perspective. Figure 4 shows the factors related to the technological perspective. And finally, Figure 5 shows the factors related to the processes perspective.

Factors of influence	Levels				
	1. Ad-hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Planning for reuse	• Nonexistent.	• Grassroots activity. • Reuse is viewed as single-point opportunities. • Individual achievements are rewarded.	• Targets of opportunity • Organization responsible for reuse. • A key business strategy.	• Business imperative. • Reuse occurs across all functional areas.	• Part of a strategic plan. • Discriminator in business success.
Software reuse education	• Lack of expertise by the staff members (engineers and managers). • Frequent resistance to reuse.	• Basic definitions of reuse are agreed upon.	• The staff has the expertise and how to obtain benefits with reuse.	• The staff members know the reuse vocabulary and have reuse expertise.	• All definitions, guidelines, standards are in place, enterprise-wide.
Legal, Contractual, Accounting considerations	• Inhibitor to getting started.	• Internal accounting scheme for sharing costs, allocating benefits.	• Data rights and compensation issues resolved with customer.	• Royalty scheme for all suppliers and customers.	• Software treated as key capital asset.
Funding, Costs and Financial Features.	• Costs of reuse are unknown.	• Costs of reuse are "feared".	• Payoff of reuse is "known" and understood for a given domain. • Investments made in reuse, payoffs expected. • Costs of reuse are "known".	• All costs associated with an asset's development and all savings from its reuse are reported and shared.	• All costs associated to a product line or a particular asset and all savings from its reuse are reported and shared.
Rewards and incentives	• Reuse is discouraged by management.	• Reuse is encouraged.	• Reuse is motivated, reinforced, rewarded.	• Reuse is indoctrinated.	• Reuse is "the way we do business".
Independent reusable assets development team.	• Individual initiative (personal goal; as time allows).	• Shared initiative.	• Dedicated individual.	• Dedicated group.	• Corporate group (for visibility not control) with division liaisons.

Figure 2. RiSE Maturity Model Levels: Organizational Factors

Factors of influence	Levels				
	1. Ad-hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Product family approach	• Isolated products. • No family product approach.	• Common features and requirements across the products. • Commonalities and reuse possibilities were identified.	• Product line domain analyses performed.	• Focus on developing families of products. • Domain Engineering performed.	• Domain analyses performed across all product lines. • Product family approach.
Software reuse education	• Chaotic development process; unclear where reuse comes in.	• Reuse questions raised at design reviews (after the fact). • Development process defined (some reuse activity indications).	• Design emphasis placed on reuse of off-the-shelf parts. • Product line domain analyses performed. • Shared understanding of all the activities needed to support reuse.	• Focus on developing families of products. • Reuse-based processes are in place to support and encourage reuse. • Domain Engineering performed.	• All software products generalized for future reuse. • Domain analyses performed across all product lines. • Product family approach.

Figure 3. RiSE Maturity Model Levels: Business Factors

Factors of influence	Levels				
	1. Ad-hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Repository systems usage	• Salvage yard (No apparent structure to collection).	• Catalog identifies language- and platform-specific parts. • Simple structures like Concurrent Versions Systems. • Considered mainly source code.	• Catalog includes generic data processing functions. • Considered software components, reports and document models.	• Catalog organized along application-specific lines. • Have all data needed to decide which assets to build/acquire. • Considered screen generators, database elements and test cases.	• Planned activity to acquire or develop missing pieces in catalog. • Considered all artifacts of software development life cycle.
Technology support	• Personal tools, if any.	• A collection of tools, e.g. CM, but not specialized to reuse. • General-purpose analyzers combined to assess reuse levels.	• Classification aids & synthesis aids. • Standardization on components and architecture. • Tools customized to support reuse.	• Digital library separate from development environment.	• Automated support integrated with development system. • Fully integrated with development and reporting systems.

Figure 4. RiSE Maturity Model Levels: Technological Factors

All the perspectives describe activities specific to software reuse. This means that activities which are typical for traditional software development are not included in this model.

The RiSE Maturity Model: **(i)** Supports constant evolution, in an incremental way, with five levels of reuse maturity; **(ii)** Defines reuse-specific engineering, management and support practices that are expected to be put in place at each level of maturity so that any organization can adopt the RiSE Maturity Model.

Factors of influence	Levels				
	1. Ad-hoc	2. Basic	3. Initial	4. Organized	5. Systematic
Quality models usage	• No quality model adoption.	• Some quality activities were incorporated in the software development process.	• Software development process guided by a quality model.	• High quality model usage in the engineering department.	• Quality model completely adopted in the organization activities.
Software reuse measurement	• No metrics on level of reuse, payoff, or cost of reuse.	• Number of lines of reused code factored into cost models.	• Manual tracking of reuse occurrences of catalog parts.	• Analyses performed to identify expected payoffs from developing reusable parts.	• All system utilities, software tools, and accounting mechanisms instrumented to track reuse.
Systematic reuse process	• No reuse-based process.	• Some reuse activities were adopted in the development process. • Planning to adapt the software development process of the organization for a reuse-based process.	• Development process of the organization is adapted to reuse concepts.	• Reuse benefits and concepts are clear for the engineering team. • Development process is reuse-based.	• Systematic reuse process is enterprise-wide.
Origin of the reused assets	• No reuse assets.	• Build from scratch, some times indirectly.	• Build from existent products; adapting existing products.	• Build from existing products; extracted through a reengineering process.	• Planning the design and building of reusable assets according to product family.
Previous development of reusable assets	• No development of reusable assets.	• Parallel with development.	• Before development.	• Before development.	• Before development.

Figure 5. RiSE Maturity Model Levels: Processes Factors

5. Concluding remarks and Future Works

This paper describes the specification of the initial RiSE Maturity Model. It describes the approach for creating the model, its current structure and the levels it comprises. The results of the work on the RiSE project have been also taken into account during the development of the RiSE Maturity Model.

The RiSE Maturity Model is recommended to be used as a reference model in a Reuse Adoption Program, i.e. as a basis for estimating the level of software reuse practice within an organization. As future work, the RiSE Maturity Model aims at identifying the strengths of an organization with respect to software reuse and the opportunities for improvements. Such an analysis will serve as a catalyst for introducing reuse engineering and management practices and tools in a consistent manner, consolidating the strengths and favoring the understanding of the organization weak points.

Needless to say that the correct implementation of software reuse and the benefits for an organization adopting reuse in their processes can be evaluated only based on quantitative data. Therefore appropriate Reuse Business and Engineering metrics will be defined and are recommended to be used within the maturity model to measure the achievement of the respective objectives, the efficiency of the applied practices and the quality of the results obtained.

It is so hard to evaluate a model. To do this, we planning to apply our model in an industrial environment, at CESAR and Digital Port (`http://www.portodigital.org.br/`), to get more feedbacks from experts. We planning yet, to make the reuse model totally conformant with the ISO/IEC 12207 and ISO/IEC 15504 [ISO/IEC 1999].

The transition between one level to another for some factors still seems very subjective. Thus, another future work is the definition of specific guidelines to aid the organization to implement a reuse assessment (compliant with ISO/IEC 15504) to evaluate the current reuse practice stage and plan the next activities to implement the reuse adoption program. These specific question will help in organization reuse practices evolution.

An appraisal of the RiSE maturity level for one or more organizations (initially only small or medium organizations) will be performed to develop such guidelines and heuristics that help assess the maturity level of an organization.

Acknowledgment

This work is partially supported by Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) - Brazil, process number: 674/2005; and *CompGov: Biblioteca Compartilhada de Componentes para E-gov*, MCT/FINEP/Ação Transversal process number: 1843/04.

References

Almeida, E. S. (2007). *RiDE: The RiSE Process for Domain Engineering*. Phd thesis, Federal University of Pernambuco (sandwich period at Universität Mannheim).

Almeida, E. S., Alvaro, A., Lucrédio, D., Garcia, V. C., and Meira, S. R. L. (2004). Rise project: Towards a robust framework for software reuse. In *IEEE International Conference on Information Reuse and Integration (IRI)*, pages 48–53, Las Vegas, USA. IEEE/CMS.

Almeida, E. S., Alvaro, A., Lucrédio, D., Garcia, V. C., and Meira, S. R. L. (2005). A survey on software reuse processes. In *IEEE International Conference on Information Reuse and Integration (IRI)*, pages 66– 71, Las Vegas, Nevada, USA. INSPEC Accession Number: 8689289 Digital Object Identifier: 10.1109/IRI-05.2005.1506451 Posted online: 2005-09-12 09:08:08.0.

Biggerstaff, T. J. and Richter, C. (1987). Reusability framework, assessment and directions. *IEEE Software*, 4(2):41–49.

Brito, K. S., Alvaro, A., Lucrédio, D., Almeida, E. S., and Meira, S. R. L. (2006). Software reuse: A brief overview of the brazilian industry's case. In *5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE), Short Paper*, Rio de Janeiro, Brazil. ACM Press.

Burkhard, N. (1993). Reuse-driven software processes guidebook. version 02.00.03. Technical Report SPC-92019, Software Productivity Consortium.

Card, D. N. and Comer, E. R. (1994). Why do so many reuse programs fail? *IEEE Software*, 11(5):114 – 115.

Chrissis, M. B., Konrad, M., and Shrum, S. (2004). *CMMI : Guidelines for Process Integration and Product Improvement*. Addison-Wesley Professional, 1st edition edition.

Davis, M. J. (1992). Stars reuse maturity model: Guidelines for reuse strategy formulation. In *Proceedings of the Fifth Workshop on Institutionalizing Software Reuse*, Palo Alto, California, USA.

Davis, T. (1993). The reuse capability model: A basis for improving an organization's reuse capability. In *Proceedings of 2nd ACM/IEEE International Workshop on Software Reusability*, pages 126–133. IEEE Computer Society Press / ACM Press.

Frakes, W. B. and Fox, C. J. (1995). Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–87. ACM Press. New York, NY, USA.

Frakes, W. B. and Isoda, S. (1994). Success factors of systematic software reuse. *IEEE Software*, 11(01):14–19.

Griss, M. L. (1994). Software reuse experience at hewlett-packard. In *16th International Conference on Software Engineering (ICSE)*, page 270, Sorrento, Italy. IEEE/CS Press.

Holibaugh, R., Cohen, S., Kang, K. C., and Peterson, S. (1989). Reuse: where to begin and why. In *TRI-Ada '89: Proceedings of the conference on Tri-Ada '89*, pages 266 – 277, Pittsburgh, Pennsylvania, United States. ACM Press.

ISO/IEC (1998). ISO/IEC 12207 software life cycle processes. International Standard 12207, ISO (the International Organization for Standardization) and IEC (the International Elechtrotechnical Commission).

ISO/IEC (1999). ISO/IEC 15504. information technology - process assesment part 1 - concepts and vocabulary (2004); part 2 - performing an assesment (2003); part 3 - guidance on performing an assesment (2004); part 4 - guidance on use for process improvement and process capability determination (2004); and part 5 - an exemplar process assesment model (1999). International Standard 15504, ISO (the International Organization for Standardization) and IEC (the International Elechtrotechnical Commission).

Koltun, P. and Hudson, A. (1991). A reuse maturity model. In *4th Annual Workshop on Software Reuse*, Hemdon, Virginia: Center for Innovative Technology.

Lim, W. C. (1994). Effects of reuse on quality, productivity and economics. *IEEE Software*, 11(5):23–30.

Lucrédio, D., Brito, K. S., Alvaro, A., Garcia, V. C., Almeida, E. S., Fortes, R. P. M., and Meira, S. R. L. (2007). Software reuse: The brazilian industry scenario. *submitted to the Journal of Systems and Software, Elsevier*.

Moore, M. M. (2001). Software reuse: Silver bullet? *IEEE Software*, 18(05):86.

Morisio, M., Ezran, M., and Tully, C. (2002). Success and failure factors in software reuse. *IEEE Transactions on Software Engineering*, 28(04):340–357.

Poulin, J. S. (2006). The business case for software reuse: Reuse metrics, economic models, organizational issues, and case studies. Tutorial notes.

Prieto-Díaz, R. (1991). Making software reuse work: An implementation model. *ACM SIGSOFT Software Engineering Notes*, 16(3):61–68.

Prieto-Díaz, R. (1993). Status report: Software reusability. *IEEE Software*, 10(3):61–66. IEEE Computer Society Press. Los Alamitos, CA, USA.

Rine, D. C. (1997a). Success factors for software reuse that are applicable across domains and businesses. In *ACM Symposium on Applied Computing*, pages 182–186, San Jose, California, USA. ACM Press.

Rine, D. C. (1997b). Supporting reuse with object technology - guest editor's introduction. *IEEE Computer*, 30(10):43–45.

Rine, D. C. and Nada, N. (2000a). An empirical study of a software reuse reference model. *Information and Software Technology*, 42(1):47–65.

Rine, D. C. and Nada, N. (2000b). Three empirical studies of a software reuse reference model. *Software: Practice and Experience*, 30(6):685–722.

Rine, D. C. and Sonnemann, R. M. (1998). Investiments in reusable software. a study of software reuse investiment success factors. *The Journal of Systems and Software*, 41:17–32.

SOFTEX (2007). Mps.br official web site (hosted by association for promoting the brazilian software excellence - softex).

SPC (1993). Reuse adoption guidebook, version 02.00.05. Technical Report SPC-92051-CMC, Software Productivity Consortium.

Tracz, W. (1990). Where does reuse start? *ACM SIGSOFT Software Engineering Notes*, 15(2):42 – 46.

Wartik, S. and Davis, T. (1999). A phased reuse adoption model. *The Journal of Systems and Software*, 46:13–23.

LIFT: Reusing Knowledge from Legacy Systems

Kellyton dos Santos Brito[1, 2], Vinícius Cardoso Garcia[1, 2], Daniel Lucrédio[3], Eduardo Santana de Almeida[1, 2], Silvio Lemos Meira[1, 2]

[1]Informatics Center - Federal University of Pernambuco (UFPE)

[2]Recife Center for Advanced Studies and Systems (CESAR)

[3]ICMC - Institute of Mathematical and Computer Sciences - University of São Paulo (USP)

{ksb, vcg, esa2, srlm}@cin.ufpe.br, lucredio@icmc.usp.br

Abstract. *Software maintenance tasks are the most expensive activities on legacy systems life cycle, and system understanding is the most important factor of this cost. Thus, in order to aid legacy knowledge retrieval and reuse, this paper presents LIFT: a Legacy InFormation retrieval Tool, discussing since its initial requirements until its preliminary experience in industrial projects.*

1. Introduction

Companies stand at a crossroads of competitive survival, depending on information systems to keep their business. In general, since these systems many times are built and maintained in the last decades, they are mature, stable, with few bugs and defects, having considerable information about the business, being called as legacy systems [Connall 1993, Ulrich 1994].

On the other hand, the business dynamics demand constant changes in legacy systems, which causes quality loss and difficult maintenance [Lehman 1985], making software maintenance to be the most expensive software activity, responsible for more than 90% of software budgets [Lientz 1978, Standish 1984, Erlikh 2000]. In this context, companies have some alternatives: **(i)** to replace the applications by other software packages, losing the entire knowledge associated with the application and needing change in the business processes to adapt to new applications; **(ii)** to rebuild the applications from scratch, still losing the knowledge embedded in the application; or **(iii)** to perform application reengineering, reusing the knowledge embedded in the systems.

Reengineering legacy systems is a choice that prioritizes knowledge reuse, instead of building everything from scratch again. It is composed of two main tasks, Reverse Engineering, which is responsible for system understanding and knowledge retrieval, and Forward Engineering, which is the reconstruction phase. The literature [Lehman 1985, Jacobson 1997, Bianchi 2000] discusses several methods and processes to support reengineering tasks, as well as specific tools [Paul 1992, Müller 1993, Storey 1995, Finnigan 1997, Singer 1997, Zayour 2000, Favre 2001, Lanza 2003a, Lanza 2003b, Schäfer 2006] to automate it. However, even with these advances, some activities are still difficult to replicate in industrial context, especially in the first step

(reverse engineering) when there are a lot of spread information, sometimes with few or no documentation at all. Thus, tools that can aid and automate some of these activities are extremely essential. Despite the new tools available today some shortcomings still exist, such as: **(i)** the recovery of the entire system (interface, design and database), and trace the requirements from interface to database access, instead of only architectural, database or user interface recovery; **(ii)** the recovery of system functionality, i.e, ***what*** the system does, instead of recovering only the architecture, that shows ***how*** the system works; **(iii)** the difficult of managing the huge data amount present in the systems and the high dependency of the expert's knowledge; and **(iv)**, although existing tools address a proper set of requirements, such as search [Paul 1992], cognitive [Zayour 2000] or visualization capabilities [Schäfer 2006], they normally fail to address all the requirements together.

Thus, in this work, we present LIFT: Legacy InFormation retrieval Tool, which is a tool for knowledge retrieval from legacy systems, designed to fulfill the shortcomings identified in current tools. The remainder of this paper is organized as follows. In Section 2, we present the background of reengineering and reverse engineering, in order to clarify the terms and concepts used. In Section 3, we present the set of requirements of LIFT, based on a survey on reverse engineering tools, in conjunction with its architecture and implementation. Section 4 presents more details about the tool's functionality. In Section 5 we present a case study using the tool. Finally, in Section 6 we discuss some conclusions and future directions.

2. Background

According to the literature [Chikofsky 1990, Sommerville 2000, Pressman 2001], Reverse Engineering *is the process of analyzing a subject system to identify their components and interrelationships, in order to create representations in another form or at a higher abstraction level, as well as to recover embedded information, allowing knowledge reuse*. In this sense, reengineering is *the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form*. In other words, reengineering is composed by a reverse engineering phase followed by a delta, which is reorganization or any alteration, and forward engineering.

The reengineering has basically four objectives [Sneed 1995, Bennett 2000]: **(i) to improve maintainability**: generating documents and building more structured, cohesive and coupled systems; **(ii) migration**: moving the software to a better or less expensive operational environment or convert old programming languages into new programming languages; **(iii) to achieve greater reliability**: the reengineering process has activities that reveal potential defects, such as re-documentation and testing; and **(iv) preparation for functional enhancement**: improving the software structure and isolating them from each other, make it easier to change or add new functions without affecting other modules. A great number of approaches and tools have been proposed to face the reengineering, and Garcia et al. [Garcia 2004, Garcia 2005] classified the main approaches in **(i)** *Source to Source Translation*, **(ii)** *Object recovery and specification*, **(iii)** *Incremental approaches* and **(iv)** *Component-based approaches*.

In reengineering, understanding the code is crucial to ensure that the intended behavior of the system is not broken. Studies show that these activities are the most

expensive tasks [Standish 1984, Erlikh 2000] and one of the efforts to decrease these costs is the development of software visualization tools [Bassil 2001], which make use of sophisticated visualization techniques to amplify cognition, and software exploration tools, which provide basic navigation facilities, i.e., searching and browsing [Robitaille 2000]. The boundary between these two categories is fuzzy, and Schäfer et al. [Schäfer 2006] defined an hybrid category, called visual exploration tools.

In this new category of visual exploration tools, Schäfer et al. [Schäfer 2006] discussed and justified five functional requirements and observed that existing tools only support a subset of them, in general each one covering different requirements. The requirements are: **(i)** *Integrated Comprehension*, **(ii)** *Cross-Artifact Support*, **(iii)** *Explicit Representation*, **(iv)** *Extensibilit,* and **(v)** *Traceability*.

Our work focuses on knowledge retrieval from legacy systems, based on reverse engineering, aiming to recognize and reuse the knowledge embedded in legacy systems, allowing the system maintenance or the forward engineering. We studied several software exploration and visualization tools, identifying their main requirements, in order to recognize not only five, but a solid set of initial requirements for a visual exploration tool. This study served as a basis for the definition of the LIFT tool, which is a tool for knowledge retrieval from legacy systems that complains to these identified requirements. In addition, we defined a new requirement that define techniques of automatic analysis and suggestions to user. Next section describes these requirements and the tool.

3. LIFT: Legacy Information Tool

In order to identify fundamental requirements for our visual exploration tool, we performed a survey covering the most known software tools with visualization and/or exploration capabilities. The survey included nine works: Scruple [Paul 1992], Rigi [Müller 1993], TkSee [Singer 1997], PBS [Finnigan 1997], SHriMP [Storey 1995], DynaSee[Zayour 2000], GSEE [Favre 2001], CodeCrawler [Lanza 2003a, Lanza 2003b] and Sextant [Schäfer 2006]. The identified requirements are shown in Table 1, and classified as follows:

R1. Visualization of entities and relations: Harel [Harel 1992] claimed that "*using appropriate visual formalisms can have spectacular effect on engineers and programmers*". An easy visualization of system modules and relationships, usually presented by a Call Graph, is an important issue of a software visualization tool, providing a graphical presentation of system and subsystems.

R2. Abstraction mechanisms and integrated comprehension: The visualization of large systems in one single view usually presents many pieces of information that are difficult to understand. Thus, the capability of presenting several views and abstraction levels as well as to allow user create and manipulate these views is fundamental for the understanding of large software systems.

R3. User interactivity: In addiction to the creation of abstractions and views, other interactivity options are also important, such as the possibility of the user take notes on the code, abstractions and views, which allows the recognition of information by another user or by the same user at some point again. This way, duplicated work can be

avoided. Another issue is the possibility of an easy switch between the high level code visualization and the source code, to permit the user to see both code representations without losing cognition information.

R4. Search capabilities: During software exploration, related artifacts are successively accessed. Thus, it is highly recommended to minimize the artifact acquisition time, as well as the number and complexity of intermediate steps in the acquiring procedure. In this way, the support of arbitrary navigation, such as search capabilities, is a common requirement in software reverse engineering tools.

R5. Trace Capabilities: Software exploration requires a large cognitive effort. In general, the user spends many days following the execution path of a requirement to understand it, and often it is difficult to mentally recall the execution path and the already studied items. Thus, to prevent the user from getting lost in execution paths, the tools should provide ways to backtrack the flows of the user, show already visited items and paths, and also indicate options for further exploration.

R6. Metrics Support: Visual presentations can present a lot of information in a single view. Reverse engineering tools should take advantage of these presentations to show some useful information in an effective way. This information can be metrics about cohesion and coupling of modules, length, internal complexity or other kinds of information chosen by user, and can be presented, for example, as the color, length and format of entities in a call graph.

R7. Cross artifacts support: A software system is not only source code, but a set of semantic (source code comments, manuals and documents) and syntactic (functions, operations and algorithms) information spread in a lot of files. So, a reverse engineering tool should be capable of dealing with several kinds of artifacts.

R8. Extensibility: The software development area is in constant evolution. The technologies and tools are in constant change, and their lifetime is even shorter. Thus, the tools must be flexible, extensible, and not technology-dependent, in order to permit its usage with a high range of systems and increase its lifetime.

R9. Integration with other tools: As software reuse researchers advocate [Krueger 1992], it is not necessary reinvent new solutions when others already exist, and a tool should permit that features present in other tools could be incorporated into it, as well as adopting standards to permit communication between distinct tools.

R10. Semi Automatic Suggestion: In general, the software engineer's expertise and domain knowledge are important in reverse engineering tasks [Sartipi 2000]. However, in many cases this expertise is not available, adding a new setback to system understanding. In these cases, the tool should have functionalities that automatically analyze the source code and perform some kind of suggestions to user, such as automatic clustering and patterns detection. However, we identified that this kind of requirement is not present in existent tools, and recognize it as a new requirement for knowledge recovery of software visualization and exploration tools.

3.1. Architecture

Based on the requirements defined in the previous section, we defined the LIFT architecture. In addition, we designed it to fulfill other non functional requirements,

such as Scalability, Maintainability and Reusability, and defined the main modules, the most important components, and expansion and integration points. The tool architecture is shown in Figure 1, and consists of four modules: *Parser*, *Analyzer*, *Visualizer* and *Understanding Environment*.

Table 1: Requirements of Software Visualization and Exploration Tools

Requirement	Tools								
	Scruple	Rigi	TkSEE	PBS	SHriMP	DynaSee	GSEE	Code Crawler	Sextant
Call Visualization		X	X	X	X		X	X	X
Abstraction Mechanisms		X			X				
User Interactivity	X	X		X	X		X	X	
Search Capabilities	X	X	X		X	X	X		
Trace Capabilities			X		X	X			X
Metrics Support		X						X	
Cross Artifacts Support		X	X	X					X
Extensibility		X	X	X					X
Integration			X	X		X			X
Semi Automatic Suggestion									

***Parser*:** It is responsible for organizing the source code. Thus, the legacy code is parsed and inserted in a structured database, in two steps: Initially, all code is parsed and stored in a first structure. Next, the parsed code is organized into a higher abstraction level, used by the application. This separation is useful to allow scalability, because the tool accesses the structure that contains only useful information, instead of all source code statements. The separation also allows an easy use of the toll with several technologies, since the use of a different input language can be made only by changing the parser component. Figure 2a shows some tables of the first parser and Figure 2b shows tables of the organized structure.

Analyzer: It is responsible for analyzing the code inserted in the structured database and to generate representations. First, by using pre-processing information, application modules are classified as interface and business one. In the first version of tool, this classification is based on language characteristics. For example, in NATURAL/ADABAS systems used in the case study, modules headers contain information if it is a map (interface module) or a program (business module). Next, the system database is inferred from legacy database instructions. Thus, a call graph is created with module information, such as size, type and source code comments.

Still within the analyzer, a second step is performed, to analyze and deduce the useful information. The tool uses mainly two methods: **(i)** minimal paths algorithm and **(ii)** cluster detection.

Minimal path is used to permit the user to follow the entire path from the user interface to data access. In this sense, the analyzer computes all minimal paths from all user interface and business modules to database entities, in order to support the user when following the system sequences. The minimal path implementation is based on the well-known Dijkstra algorithm [Dijkstra 1959].

On the other hand, cluster detection identifies and shows legacy system clusters that possibly can be recognized as a higher level abstraction, an object or component, or modules that can be merged to form one more cohesive structure. There are some approaches of cluster detection that are being used in reverse engineering context, mainly based on *k-means* algorithm [Sartipi 2000] that needs some previous knowledge about the code. We focus on unknown code, and choose the *Mark Newman's edge betweenness* clustering algorithm [Girvan 2002]. In this algorithm, the betweenness of an edge measures the extent to which that edge lies along shortest paths between all pairs of nodes. Edges which are least central to communities are progressively removed until the communities are adequately separated. We performed a small modification in the algorithm, which is the parameterization of the number of edges to be removed, allowing it to be interactively chosen by user.

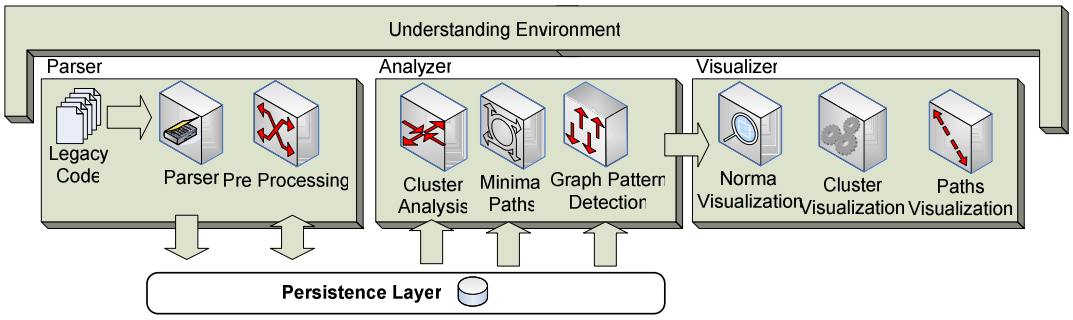


Figure 1: LIFT Architecture

Visualizer: It is responsible for managing the data generated by other modules, and to present these to the user in an understandable way. Visualization is based on a call graph, where modules and relationships are presented according to user preferences for modules and edges properties of thickness, color, format and size. The visualization has three modes, with easy transition among them: (i) default visualization, focusing on configurable attributes of modules and edges color, shape and size; (ii) path visualization, with focus on paths followed by application, with variable set of deep and direction (forward, upward or mixed mode) of paths; and (iii) cluster visualization, with focus on cluster detection and visualization.

Understanding Environment: Integrates the other modules, containing graphical interfaces for the functionalities of parser, code analyzer and visualizer. In addition, the environment provides an easy way to show the source code.

The tool works with the concept of code views. Thus, users can generate and deal in parallel with new sub graphs from previous graphs. The environment allows, for instance, the creation of graphs with only unconnected modules, which in general are dead code or batch programs. Other option is to generate new graphs with the detected clusters, isolating them from the complete application. These views are useful to isolate

modules and paths that identify application requirements.

The environment allows also the creation of system documentations, with areas to document the views, which represent a requirement, and areas to document the modules. The module documentation is created by default with source code comments, extracted in pre-processing. Additionally, the tool also permits the user to view and comment source code, maintaining both the original and the commented versions.

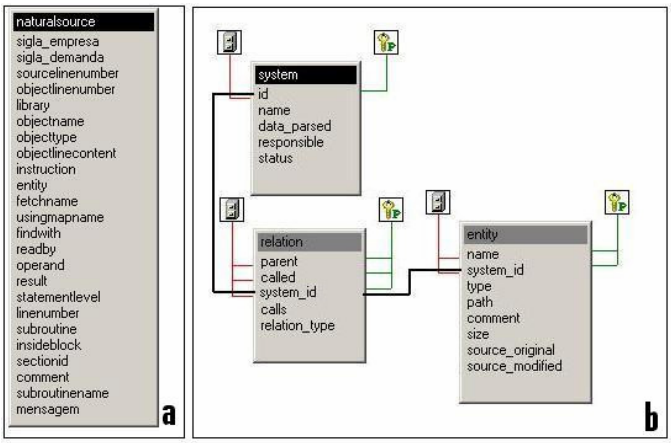


Figure 2: Database Tables used by the Parser

3.2. Implementation

In order to validate and refine the identified requirements and proposed architecture, we developed the LIFT tool in conjunction with the industry where we are engaged in reuse projects which focus on environments and tools to support reuse. These projects are part of the Reuse in Software Engineering (RiSE) project [Almeida 2004]. RiSE's goal is to develop a robust framework for software reuse, in conjunction with the industry, involving processes, methods, environment and tools. In the project, the role of the RiSE group (researchers) is to investigate the state-of-the-art in the area and disseminate the reuse culture. On the other hand, the industry (and its project managers, architects, and system engineers), represented by Recife Center for Advanced Studies and Systems (C.E.S.A.R[1]) and Pitang Software Factory[2], is responsible for "making things happens" with planning, management, and necessary staff and resources.

The LIFT was developed based on the architecture presented in previous subsection. The tool has a three-tier client-server architecture developed in JAVA.

The persistence layer uses SQL ANSI statements, therefore it is database independent. The parser was already implemented by the Pitang Software Factory as a .NET standalone application, and was refined and improved to be incorporated in LIFT. All other modules were developed in JAVA. Cluster analysis was developed based on *Mark Newman's edge betweenness* clustering algorithm and Minimal Paths was based

[1] Recife Center for Advanced Studies and Systems – http://www.cesar.org.br

[2] Pitang software factory – http://www.pitang.com

on *Dijkstra* algoritm, as shown in previous sections. The Visualizer uses the JUNG[3], Java Universal Network/Graph Framework, to implement visualizations.

The current version of LIFT implementation contains 76 classes, with 787 methods, divided into 25 packages, containing 9.420 lines of code (not counted code comments).

4. LIFT Usage

This section presents LIFT from a user's point of view. The initial steps, parsing, organization and call graph generation is performed by simple menu commands, shown in Figure 3a.

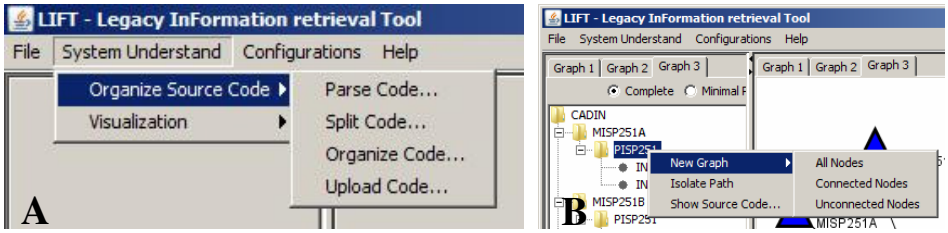


Figure 3. LIFT menus

The main screen, shown in Figure 4, has three areas. The left area (index 1) shows the paths and minimal paths from screens and business modules to database modules. The right area (index 2) shows the selected module information, such as the relations and comments, inserted by user or recognized by source code comments. In the center (index 3) the call graph is shown, with the tree visualization options (index 4).

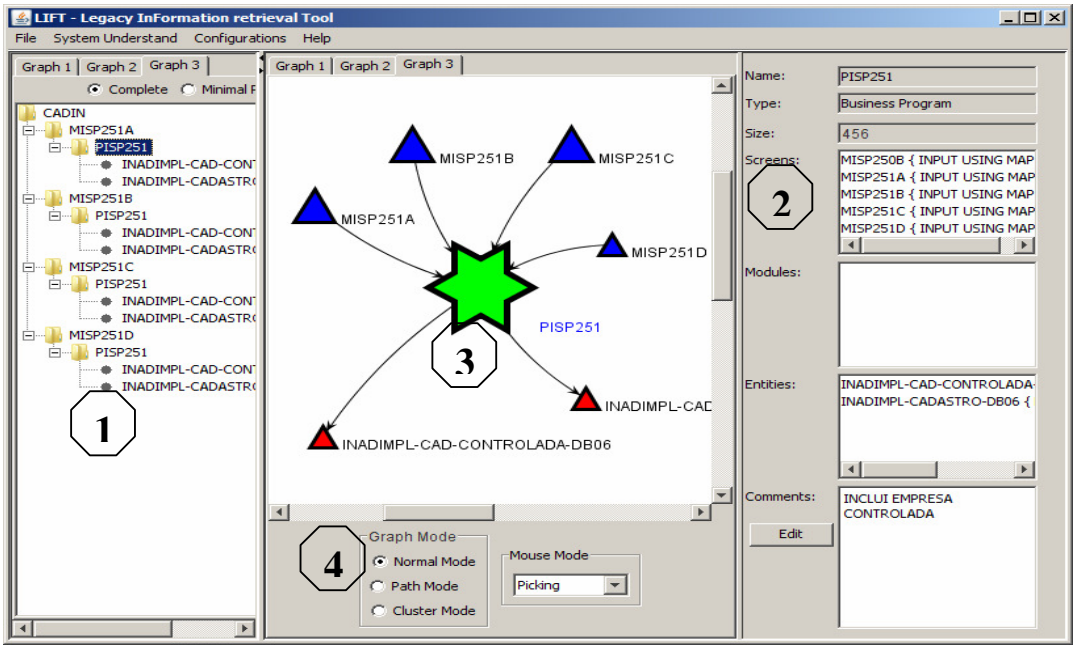


Figure 4: Isolated view in normal mode

[3] JUNG: Java Universal Network/Graph Framework - http://jung.sourceforge.net/

The first step towards system understanding is isolate unconnected nodes, which may be identified as dead code or batch programs. This task is performed by right clicking the paths area and choosing submenus “New Graph” and “Unconnected Nodes”, as shown in Figure 3b. These modules are analyzed separately from other modules. So, in a similar way, a new view containing only connected nodes is generated. In this view, the user tries to discover high coupled and related modules, by cluster detection, as shown in Figure 5a. Therefore, clustered modules are separated in a new view and analyzed in separate, in general resulting in a requirement. This new view is simpler than the complete view with all connected modules, providing an easier visualization of a possible requirement. Thus, by using the functionalities of path mode and analyzing the source code, the user can identify and generate documentation of the requirement. This documentation can be made in the *description area*, present in each view.

These steps are repeated until the entire application is separated into clusters, or no more clusters can be detected. In the last case, remaining modules are analyzed using the path mode (Figure 5b), in order to retrieve these requirements.

5. LIFT Preliminary Evaluation

LIFT is being used in a project involving C.E.S.A.R and the Pitang Software Factory. These institutions have acquired experience in understanding and retrieving knowledge from 2 million LOC of Natural/ADABAS systems, with programs varying from 11.000 LOC to 500.000 LOC. In these projects, only the source code is received and documents describing these are generated. Moreover, two kinds of knowledge retrieval can be performed: to understand for maintenance or understand for reimplementation in another platform.

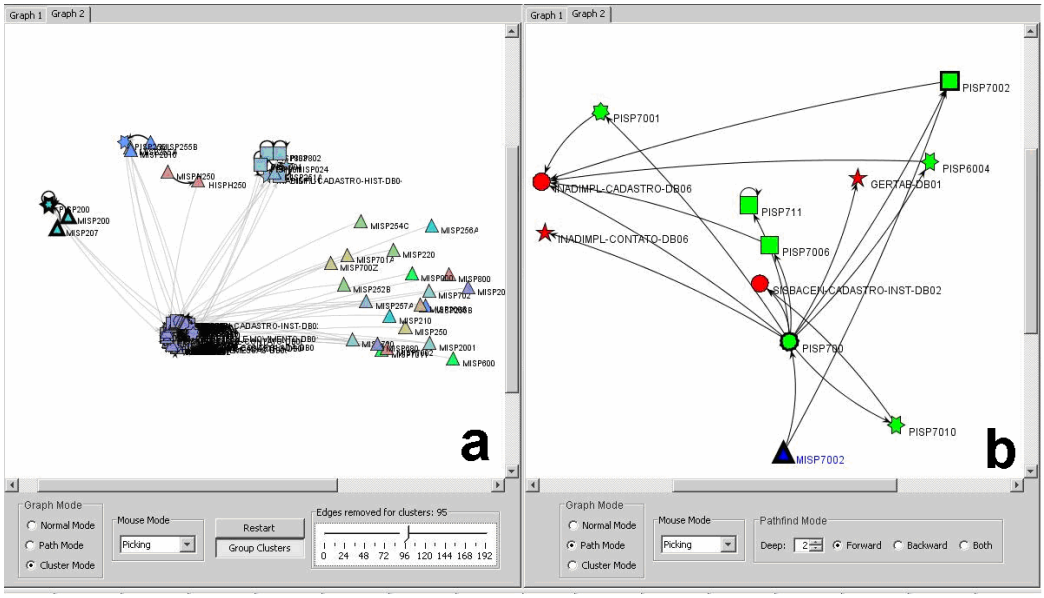


Figure 5: LIFT Cluster and Path Visualization Modes

Initially, LIFT was partially applied in a pilot project of understanding, for maintenance, a 65.000 LOC system. The understanding was performed in seven weeks,

by one Natural/ADABAS expert system engineer (SE) with no knowledge about analyzed code. The engineer recovered 10 high level requirements groups, each one with a proper set of sub requirements, which were validated by the project stakeholders. This pilot served to understand the company process, identify points where the tool could aid, refine it and improve its usability and scalability.

As a second interaction, the tool is being fully applied in a project of understanding for maintenance of a 210.000 LOC system. Initially, we made minor changes in the existing process to use the tool facilities. The modified process is shown in Figure 5 and defines four phases: **(i)** Organize Code, **(ii)** Analyze Code, **(iii)** Understand System and **(iv)** Validate Project. The three first phases are supported by LIFT. The new process is shown in Figure 5.

In this interaction, the source code was parsed and inserted into the database. Next, it was organized and the connected modules were separated from the unconnected ones and assigned for a system engineer for estimation and understanding. Thus, based on cluster analysis, SE experience and stakeholder information, the connected graph was split into 19 high level requirements groups, that would be responsible for each high level requirement. Next, based on the *Pitang staff* experience and company baseline, each group had a preliminary analysis and effort estimation. Thus, the understanding and documentation generation started.

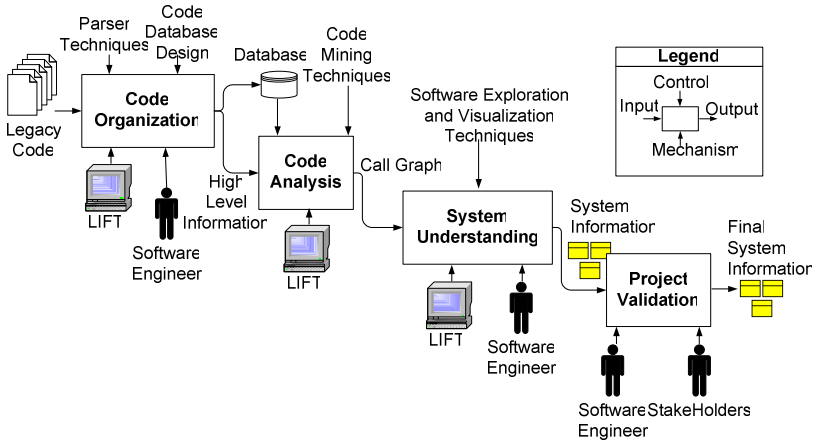


Figure 6: Process used to recovery legacy knowledge with LIFT

The requirements are documented as follows. A context diagram is generated, showing accesses to systems beyond application boundaries, such as access to other systems data or functions. Thus, *description area* comments of each view are mapped to functional requirements descriptions. Each functional requirement has also a tree with the modules and relationships of the correspondent view, that compound the modules and relationships of the requirement, and each one is described in details. In addition, each system data entity is recovered and described in another document section.

When this paper was being written, 12 of 19 high level requirements groups had already been understood. Table 2 shows estimated and executed understanding times of these groups. The real time to understand and document the groups was 38% less than the planned time, showing that the use of LIFT initially decreases the effort of knowledge retrieval tasks.

In addition to comparing estimation and executing time with the tool, we plan to compare final project data with previous project data, such as the number of requirements recovered, and total execution time by lines of code, function points and use case points. These data will be useful to validate these initial results.

Moreover, we are collecting user experiences to validate strong and weak points of the tool, as well as to identify possible new functionalities. Users agree that minimal paths visualization is very useful in knowledge recovery for re-implementation, because the key objective is to know the main application execution path, instead of details. However, the visualization of complete paths is desired in knowledge recovery for maintenance, because of the need for a map of the entire application when maintenance tasks are performed. Additionally, they agree that the use of views to isolate possible requirements and the existence of "*Path Mode*" are very useful to deal with large systems, allowing clean visualizations of large systems.

Another important consideration is that users reported that cluster analysis is useful to identify and isolate related modules, but the applicability of this option was limited when identifying the high level requirements groups because the NATURAL/ADABAS environment has some features that maintain and show to the user a list of the application entry and main modules. However, cluster analysis was useful to identify some of high level requirements not included on this list, and clusters and sub-requirements inside them.

Some points are being considered, as other forms of suggestions, such as text pattern detection in modules names, comments or source code, and document automatic generation.

Eventually, we ended up identifying that the tool can be used to understand large systems. Running in a Pentium IV/512MB Ram station and accessing a Pentium IV/512MB Ram database server, the tool performed tasks in the time shown in Table 3.

Table 2: Reduction in legacy knowledge recovery effort with LIFT

Group	Estimation(h)	Execution(h)	Gain
Group 1	13,00	11,00	15%
Group 2	10,00	8,00	20%
Group 3	18,00	11,00	39%
Group 4	13,00	8,00	38%
Group 5	10,00	6,00	40%
Group 6	2,00	1,00	50%
Group 7	18,00	9,00	50%
Group 8	18,00	8,00	56%
Group 9	10,00	6,50	35%
Group 10	7,00	5,00	29%
Group 11	1,00	1,00	0%
Group 12	1,00	1,00	0%
TOTAL	**121,00**	**75,50**	**38%**

Parse Code and *Organization* are slow tasks, but we consider that this time does not harm the tool's performance because these tasks occur only once in each system. In addition, *Minimal Paths Calculation*, *Analysis* and *Load Graph* tasks take a small time, but are performed few times, that is, only when the application runs and the system is chosen. Besides, *Cluster Detection* is a task that takes little time, in general from 1 to 20 seconds depending on the number of modules and edges involved. Finally, after initial

analysis, the operations of *graph manipulation*, *view creations* and *load details* are instant tasks, with times imperceptibles by user, which provides a good usability and user experience.

6. Concluding Remarks and Future Work

In this paper, we presented LIFT: Legacy InFormation retrieval Tool, which is a tool for knowledge retrieval from legacy systems. The tool requirements were obtained from a set of functionalities of other tools for software visualization or exploration, from our industrial experience and customer needs, as well as a new requirement was identified as a lack on current tools, that is the analysis of source code and the perform of suggestions to user. The tool uses the concept of views, which encloses the system functionalities. Additionally, we also use a new way to store legacy data in database systems, which allows tool scalability.

The tool was partially used in a pilot project to knowledge retrieval of a 65.000 LOC NATURAL/ADABAS system, in order to refine and improves its usability and scalability. Currently, the tool is being fully used in another similar project of software understanding and knowledge recovery for the maintenance of 210.000 LOC. By the end of the second understanding interaction, we intend to compare data with previous projects, in order to get more accurate information and make better conclusions about the effort reduction.

Table 3: LIFT execution times

Task	Time (seconds)	
	65KLOC Application	210 KLOC Application
Parse Code	961s	1520s
Organize Code	660s	963s
Minimal Paths Calculation	19s	84s
Full Analysis and Graph Creation	23s	98s
Interactive Cluster Detection	0s - 20s	0s – 30s
Graph Manipulation	Imperceptible	Imperceptible

The preliminary results show effort reduction of 38% in relation to the estimated time. Also, user interviews demonstrate best applicability of minimal paths in tasks of general understand for re-implementation, and complete paths in tasks of full understanding for maintenance. We observed good user experiences in knowledge recovery using the concepts of requirements isolation in multiple views, cluster detection and *path mode* system navigation. Thus, the tool presents good scalability performing tasks in user acceptable times, running in conventional hardware.

Currently, LIFT is being modified to perform automatic documentation generation: textual requirements document and UML diagrams, such as use case and class diagrams. Also, general and graph pattern detection are being implemented, in order to provide more suggestions to user and increase the effort reduction. Finally, we plan to use the tool to perform reverse engineering and knowledge recovery in systems that use other technologies or development paradigm, such as COBOL or Java.

References

Almeida, E. S., Alvaro, A., Lucrédio, D., Garcia, V. C. and Meira, S. R. d. L. (2004). "RiSE

Project: Towards a Robust Framework for Software Reuse". IEEE International Conference on Information Reuse and Integration (IRI), Las Vegas, USA, p. 48-53.

Bassil, S. and Keller, R. K. (2001). "Software Visualization Tools: Survey and Analysis". Proceedings of International Workshop of Program Comprehension, Toronto, Ont., Canada, p. 7-17.

Bennett, K. H. and Rajlich, V. T. (2000). "Software maintenance and evolution: a roadmap". Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, ACM Press, p. 73-87.

Bianchi, A., Caivano, D. and Visaggio, G. (2000). "Method and Process for Iterative Reengineering of Data in a Legacy System". Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), Brisbane, Queensland, Australia, IEEE Computer Society, p. 86--97.

Chikofsky, E. J. and Cross, J. H. (1990). "Reverse Engineering and Design Recovery: A Taxonomy." IEEE Software Vol.(1), No. 7, p. 13-17.

Connall, D. and Burns, D. (1993). "Reverse Engineering: Getting a Grip on Legacy Systems." Data Management Review Vol.(24), No. 7.

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs." Numerische Mathematik Vol.(1), No. 1, p. 269-271.

Erlikh, L. (2000). "Leveraging Legacy System Dollars for E-Business." IT Professional Vol.(2), No. 3, p. 17-23.

Favre, J.-M. (2001). "GSEE: a Generic Software Exploration Environment". Proceedings of the International Workshop on Program Comprehension (IWPC), Toronto, Ont., Canada, p. 233.

Finnigan, P. J. (1997). "The software bookshelf." IBM Systems Journal Vol.(36), No. 4.

Garcia, V. C. (2005), "Phoenix: An Aspect Oriented Approach for Software Reengineer(in portuguese). M.Sc Thesis." Federal University of São Carlos, São Carlos,

Garcia, V. C., Lucrédio, D., Prado, A. F. d., Alvaro, A. and Almeida, E. (2004). "Towards an effective approach for reverse engineering". Proceedings of 11th Working Conference on Reverse Engineering (WCRE), Delft, Netherlands, p. 298-299.

Girvan, M. and Newman, M. E. J. (2002). "Community Structure in Social and Biological Networks." Proceedings of the National Academy of Sciences of USA Vol.(99), No. 12.

Harel, D. (1992). "Toward a Brighter Future for System Development." IEEE Computer Vol.(25), No. 1.

Jacobson, I., Griss, M. and Jonsson, P. (1997). "Software Reuse: Architecture, Process and Organization for Business Success", Addison-Wesley Professional.

Krueger, C. W. (1992). "Software Reuse." ACM Computing Surveys Vol.(24), No. 2, p. 131-183.

Lanza, M. (2003a). "CodeCrawler - lessons learned in building a software visualization tool". Proceedings of European Conference on Software Maintenance and Reengineering, p. 409-418.

Lanza, M. and Ducasse, S. p. (2003b). "Polymetric Views-A Lightweight Visual Approach to Reverse Engineering." IEEE Transactions on Software Engineering Vol.(29), No. 9, p. 782-795.

Lehman, M. M. and Belady, L. A. (1985). "Program Evolution Processes of Software Change", London: Academic Press.

Lientz, B. P., Swanson, E. B. and Tompkins, G. E. (1978). "Characteristics of Application Software Maintenance." Communications of the ACM Vol.(21), No. 6, p. 466 - 471.

Müller, H. A., Tilley, S. R. and Wong, K. (1993). "Understanding software systems using reverse engineering technology perspectives from the Rigi project". Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canada, p. 217-226.

Paul, S. (1992). "SCRUPLE: a reengineer's tool for source code search ". Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, IBM Press, p. 329-346

Pressman, R. S. (2001). "Software Engineering: A Practitioner's Approach", McGraw-Hill.

Robitaille, S., Schauer, R. and Keller, R. K. (2000). "Bridging Program Comprehension Tools by Design Navigation". Proceedings of International Conference on Software Maintenance (ICSM), San Jose, CA, USA, p. 22-32.

Sartipi, K., Kontogiannis, K. and Mavaddat, F. (2000). "Architectural design recovery using data mining techniques". Proceedings of the 4th European Software Maintenance and Reengineering (ESMR), Zurich, Switzerland, p. 129-139.

Schäfer, T., Eichberg, M., Haupt, M. and Mezini, M. (2006). "The SEXTANT Software Exploration Tool." IEEE Transactions on Software Engineering Vol.(32), No. 9.

Singer, J., Lethbridge, T., Vinson, N. and Anquetil, N. (1997). "An examination of software engineering work practices". Proceedings of conference of the Centre for Advanced Studies on Collaborative research (CASCON), Toronto, Ontario, Canada, IBM Press, p. 21.

Sneed, H. M. (1995). "Planning the Reengineering of Legacy Systems." IEEE Software Vol.(12), No. 1, p. 24-34.

Sommerville, I. (2000). "Software Engineering", Pearson Education.

Standish, T. A. (1984). "An Essay on Software Reuse." IEEE Transactions on Software Engineering Vol.(10), No. 5, p. 494-497.

Storey, M.-A. D. and Müller, H. A. (1995). "Manipulating and documenting software structures using SHriMP views". Proceedings of the International Conference on Software Maintenance (ICSM), Opio, France, p. 275 - 284.

Ulrich, W. (1994). "From Legacy Systems to Strategic Architectures." Software Engineering Strategies Vol.(2), No. 1, p. 18-30.

Zayour, I. and Lethbridge, T. C. (2000). "A cognitive and user centric based approach for reverse engineering tool design". Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research, Ontario, Canada, p. 16.

Um Processo de Desenvolvimento de Aplicações Web baseado em Serviços

Fabio Zaupa[1], Itana M. S. Gimenes[1], Don Cowan[2], Paulo Alencar[2], Carlos Lucena[3]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
Maringá – PR – Brasil

[2]Computer System Group, The University of Waterloo
Waterloo-ON, Canadá

[3]Departamento de Informática, PUC-Rio
Rio de Janeiro – RJ - Brasil

zaupa@embusca.com.br, itana@din.uem.br, dcowan@csg.uwaterloo.ca, palencar@csg.uwaterloo.ca, lucena@inf.puc-rio.br

Abstract. *Web applications are currently widely disseminated. However, traditional development methods for these applications still require a lot of modeling and programming. They do not take much advantage of reuse. This paper presents an environment, called WIDE-PL, that supports the generation of Web applications based on the product line approach and SOA. In particular, this paper describes its application generation process, called ADESE. The evaluation of the process was carried out as a case study. The results show that the process offers several advantages such as it increases reuse and provides an explicit separation between the services and the business process.*

Resumo. *Apesar da disseminação de aplicações Web, o que se nota é que os métodos tradicionais de desenvolvimento de aplicações Web ainda requerem muita modelagem, programação e não tiram muito proveito de reutilização. Este artigo apresenta um ambiente, chamado WIDE-PL, para apoiar a geração de aplicações Web baseado em SOA seguindo os princípios de linha de produto. Em particular, este artigo descreve seu processo de desenvolvimento de aplicações, chamado ADESE. A avaliação do processo foi realizada por meio de um estudo de caso. Os resultados obtidos mostram que o processo oferece vantagens como o aumento da reutilização e uma explícita separação da lógica do negócio dos serviços.*

1. Introdução

A *Web* vem crescendo continuamente nos últimos anos. Muitas organizações, visualizando a possibilidade de realização de novos negócios, expandem suas atividades, criando sistemas de informação baseados em tecnologia *Web*. Esta tecnologia inclui *browsers*, servidores *Web*, linguagens de programação *Web* e protocolos de comunicação que viabilizam o funcionamento dos sistemas de informações através da *Web*. Neste contexto, os sistemas de informação são conhecidos como aplicações *Web*. O número de aplicações *Web* tem crescido exponencialmente, porém a qualidade das aplicações disponíveis não cresce proporcionalmente. Qualidade inclui confiabilidade, disponibilidade, eficiência, usabilidade, entre outros. Casati e Shan (2000) apontam que a fachada *Web* esconde enormes deficiências, operações manuais e sujeitas a erros, inflexíveis e complexas, e sistemas de difícil gerenciamento.

Aplicações *Web* podem ser autocontidas ou podem ser concebidas como uma cooperação de serviços que podem inclusive demandar cooperações inter-organizacionais. Um exemplo disso é um sistema de agência de viagem capaz de chamar serviços oferecidos por outras organizações, como o serviço de reserva de passagem, o serviço de reserva de hotel ou até o serviço de reserva de veículo.

A Arquitetura Orientada a Serviço (SOA[1]) é vista como capaz de oferecer a infra-estrutura tecnológica necessária para que uma aplicação possa ser definida por meio da composição de serviços eletrônicos. Dessa forma, oferece apoio à composição de aplicações distribuídas de uma forma flexível e com baixo custo. Em SOA, a composição de serviços é vista como um processo de negócio dividido em componentes reutilizáveis e interoperáveis.

Apesar da disseminação de aplicações *Web*, o que se percebe é que os métodos tradicionais de desenvolvimento aplicações *Web,* como OOHDM (Schwabe; Rossi, 1998) e OOWS (Pastor; Fons; Pelechano, 2003) ainda requerem muita modelagem, programação e não tiram muito proveito de reutilização. Dessa forma, o *Computer System Group* (CSG) da Universidade de Waterloo no Canadá propôs o WIDE (Cowan et al., 2004), que consiste em um grupo articulado de *frameworks* de apoio ao desenvolvimento de aplicações *Web*. O ambiente WIDE-PL é uma evolução de WIDE para desenvolvimento de aplicações *Web* que segue os princípios de linha de produto (Kang, 1990; Czarnecki et al., 2005). O objetivo de WIDE-PL é apoiar a geração de aplicações *Web* baseado em SOA. Em WIDE-PL, os serviços utilizados em uma aplicação *Web* podem ser especificados e compostos em uma linguagem de especificação e composição de serviços.

Este artigo apresenta o ambiente WIDE-PL e, em particular, um de seus elementos que é o seu processo de desenvolvimento de aplicações, o qual é chamado de ADESE. Este processo consiste de atividades para: (i) definir o domínio da aplicação; (ii) modelar serviços com base nos conceitos de modelo de características de linha de produto; (iii) instanciar o modelo de características; (iv) mapear o modelo instanciado para um correspondente diagrama de classes; (v) implementar os serviços a partir do diagrama de classes; e, (vi) gerar aplicações com base nos serviços definidos. ADESE utiliza WSDL para especificação de serviços e BPEL4WS para especificação de processos de negócio. A avaliação do processo ADESE foi realizada por meio de um estudo de caso. Os resultados obtidos mostram que o processo oferece vantagens em aspectos como: reutilização, separação explícita da lógica do negócio e dos serviços, redução de tempo e custo de desenvolvimento, interoperabilidade e manutenibilidade.

Este artigo está organizado da seguinte forma. A Seção 2 apresenta uma visão da arquitetura *Web* orientada a serviços. A Seção 3 apresenta conceitos básicos de linha de produto. A Seção 4 apresenta o ambiente WIDE-PL. Seção 5 apresenta o processo ADESE cuja avaliação é apresentada na seção 6. A Seção 7 apresenta as conclusões.

2. Arquitetura Orientada a Serviços (SOA)

A *Web* pode ser vista como uma coleção de serviços interconectados por protocolos de comunicação sobre a Internet. Segundo Papazoglou e Georgakopoulos (2003), SOA utiliza serviços eletrônicos como elementos fundamentais para o desenvolvimento de

[1] Do inglês Service Oriented Architecture.

aplicações distribuídas. Um serviço *Web* é um tipo específico de serviço eletrônico, e pode ser caracterizado como um componente aberto, auto-descritivo, com ampla padronização, grande interoperabilidade e acessíveis a outras aplicações ou serviços por meio de um endereço URL. A tecnologia de serviços *Web* tem como principais padrões XML (XML, 2006), WSDL (Christensen et al., 2001), SOAP (SOAP, 2006), HTTP (HTTP, 2006) e UDDI (UDDI, 2006). Em resumo, um serviço *Web* pode ser entendido como uma aplicação que tem uma interface descrita em WSDL, registrada em um diretório de serviços via protocolo UDDI, e interage com clientes por meio da troca de mensagens XML encapsuladas em envelopes SOAP, que são transportadas por protocolo HTTP (Fantinato, Toledo e Gimenes, 2005).

Processos de negócio são utilizados para compor serviços *Web*. Um processo de negócio é uma seqüência de atividades executadas, possivelmente, por múltiplas organizações que atuam cooperativamente para atingir um objetivo de negócio comum. Existem algumas linguagens para especificação de processos de negócio baseados em serviços, dentre as quais destaca-se BPEL4WS (*Business Process Execution Language for Web Services*) a qual chamamos, neste artigo, simplesmente de BPEL. BPEL permite especificar um processo de negócio descrevendo o relacionamento entre os serviços participantes, manipulação de variáveis, fluxo de dados e ações para fluxo de exceções, sendo executados por um motor que age como intermediário na troca de mensagens (BEA Systems et al., 2003).

3. A Abordagem de Linha de Produto

Uma linha de produto de software consiste de um conjunto de sistemas de software, também chamado de família de aplicações, que compartilham um conjunto gerenciável de características de um seguimento particular de mercado ou missão (SEI, 2006) (Clements; Northrop, 2001). As aplicações (ou produtos) da família, chamadas de membros, são desenvolvidas a partir de um conjunto comum de artefatos (*assets*) que constituem a arquitetura da linha de produto. Esta arquitetura deve ser genérica o suficiente para representar todos os membros da linha de produto. Um membro da família é gerado adaptando-se os requisitos da linha de produto para atender às suas necessidades. Eventualmente, novos componentes podem ser desenvolvidos e adicionados à linha de produto, alimentando assim sua constante evolução.

Uma forma comum de representação do domínio de uma linha de produto é o modelo de características[2]. Este modelo permite a representação das capacidades comuns e variáveis de uma família de produtos de um determinado domínio. O conceito de características vem da engenharia de domínio (Kang, 1990) e tem evoluído para atender as demandas de linha de produto (Sochos et al., 2004; Czarnecki et al., 2005; van Gurp et al., 2001). Características são usualmente representadas por meio de diagramas de características que são estruturas em forma de árvore com anotações que indicam os tipos de características de uma família de produtos (Sochos et al., 2004; Czarnecki et al., 2005).

No contexto deste artigo, conceitos de linha de produto são aplicados para apoiar a modelagem dos serviços de domínio do ambiente WIDE-PL que comporão as aplicações *Web* geradas pelo ambiente. Um serviço de domínio é representado por características configuráveis.

[2] Do inglês features.

4. O Ambiente WIDE-PL

O *Computer System Group* (CSG) da Universidade de Waterloo tem desenvolvido tecnologias para portais *Web* desde o início dos anos 90 (Cowan et al., 2004). Como resultado, uma série de portais chamados Espaço Público de Aprendizagem foram construídos. O ambiente usado para apoiar o desenvolvimento dos portais é chamado WIDE (*Waterloo Informatics Development Environment*). WIDE possui um grupo articulado de *frameworks* baseados em XML de apoio ao desenvolvimento de aplicações *Web*, tais como serviços de mapas, serviços de diagramas, serviços de relatórios, serviços de gerenciamento, serviços de controle de acesso, serviços de pesquisa em base de dados, serviços de notificação e serviços de agente. WIDE-PL (*WIDE - Product Line*) é uma evolução do WIDE que tem como objetivo gerar aplicações *Web* baseadas em SOA seguindo os princípios da LP (Kang, 1990; Czarnecki et al., 2005). Em WIDE-PL, os serviços utilizados em uma aplicação *Web* podem ser especificados e compostos em uma linguagem de especificação e composição de serviços. WIDE-PL é composto de serviços de domínio que têm um grupo de dados e metadados associados. Os serviços são coordenados pelo processo de negócio da aplicação. WIDE-PL utiliza um princípio similar ao ambiente proposto por Ceri (2003), porém tem por objetivo gerar aplicações a partir de serviços e de um processo de negócio explícito.

Neste trabalho, utilizou-se a tecnologia de serviços *Web* para especificação e composição de serviços, devido à atual disponibilidade de padrões e de infra-estrutura de apoio que permitem a especificação de serviços em WSDL (Christensen et al., 2001) e especificação de processos de negócio em BPEL (BEA Systems et al., 2003). A vantagem de se utilizar uma linguagem como BPEL para compor os serviços de aplicações que são locais, distribuídos ou baseados em padrões da Internet, é obter uma base tecnológica que minimize a complexidade de usar tecnologias heterogêneas. No entanto, linguagens específicas para WIDE-PL podem ser projetadas no futuro.

Em WIDE-PL, uma aplicação *Web* é representada por um conjunto de características de alto nível (Kang, 1990; Czarnecki et al., 2005). Essas características são abstrações que representam as capacidades de aplicações *Web*. Cada característica de alto nível de uma aplicação é realizada por um serviço de domínio. Cada serviço de domínio abrange as operações e os dados necessários para realizar uma característica da aplicação (Kang, 2002).

A Figura 1A ilustra o modelo de características de alto nível das aplicações *Web* em WIDE-PL, produzido pela ferramenta *Feature Plug-in* (Antkiewicz; Czarnecki, 2004). Os demais modelos de características deste artigo também seguem essa notação. Uma aplicação *Web* em WIDE-PL tem ao menos um serviço de base de dados, um serviço de interação, um serviço de interface e um serviço de processo de negócio. Assim, essas características são representadas como obrigatórias. Serviços opcionais de domínio podem ser adicionados a WIDE-PL de acordo com a evolução do ambiente e as necessidades das aplicações. Exemplos de serviços que podem ser adicionados, baseados na estrutura atualmente disponível no WIDE são: interface interativa, multimídia e agente. As características de um serviço de domínio podem ser refinadas de tal maneira que possibilite aos clientes da aplicação configurar os serviços de acordo com suas necessidades. A Figura 1B apresenta um exemplo do refinamento do modelo de características para o serviço de interface interativa. Este serviço permite a definição

e a operação de diversos tipos de exposição interativa tais como mapas, gráficos e diagrama. A característica de mapa também é refinada, e apresenta as características de desenho, contêiner e resolução como obrigatórias, pois elas são necessárias em qualquer mapa, enquanto as características de interação, mensagem e pesquisa são opcionais, podendo ser inseridas no mapa de acordo com a necessidade da aplicação.

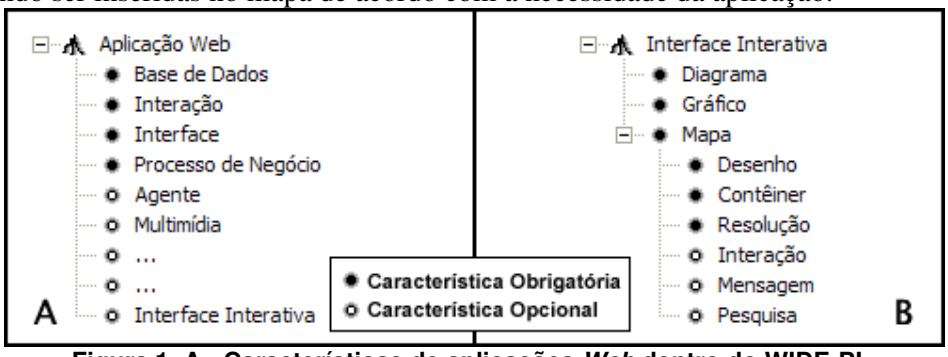


Figura 1. A - Características de aplicações *Web* dentro do WIDE-PL, B - Modelo de características do serviço de interface interativa.

Nas seções seguintes são apresentados os serviços obrigatórios e a arquitetura lógica de uma aplicação gerada em WIDE-PL.

4.1 Serviços Obrigatórios

Os serviços obrigatórios de WIDE-PL são essenciais para o funcionamento de uma aplicação *Web*. São eles:

- Serviço de base de dados: responsável por qualquer interação entre o processo de negócio e a base de dados da aplicação. Com este serviço é possível criar e atualizar a estrutura da base de dados da aplicação, além de manipular e recuperar seus respectivos dados.

- Serviço de interação: responsável por controlar as interações entre o serviço de processo de negócio e os usuários da aplicação. Deve receber um *script* de página gerado no serviço de interface e exibi-lo ao usuário por meio de seu *browser*. Ao receber requisições do usuário, deve tratar e repassar ao serviço de processo de negócio para o devido processamento.

- Serviço de interface: responsável por receber os dados e formatar a página que será exibida ao usuário. Neste serviço é definido o estilo da aplicação, escolhendo a fonte de letra e as cores que serão usadas nas páginas. Também é possível definir o conteúdo da página, informando o que deve ser apresentado no cabeçalho, rodapé, menu e corpo da página. Em relação ao corpo da página, o serviço está preparado para receber, tratar e exibir no local as listas de itens e as solicitações de formulários vindas do serviço de processo de negócio. Depois de inseridas todas as configurações, um *script* referente à página pode ser gerado.

- Serviço de processo de negócio: coordena a execução de uma aplicação, definindo o comportamento da aplicação. Contém as regras de negócio, a seqüência de invocação dos serviços participantes, as ações de controle de exceções, a utilização de funções e variáveis. Somente este serviço pode interagir com os demais serviços utilizados pela aplicação, assim qualquer requisição ou resposta a um serviço deve passar por ele.

4.2 Arquitetura Lógica

A arquitetura lógica de aplicações geradas por WIDE-PL visa deixar explícita a separação entre o processo de negócio que controla a aplicação e seus serviços. Dessa forma, a aplicação é adaptável uma vez que o processo de negócio pode ser alterado sem necessariamente modificar os serviços, assim como serviços podem ser adicionados e modificados causando impacto mínimo no processo de negócio. Esta arquitetura, conforme mostra a Figura 2, é composta de três camadas:

- Coordenação: responsável por fazer a coordenação do processo de negócio da aplicação. Contém o serviço de interação e o serviço de processo de negócio.
- Aplicação: contém uma configuração dos serviços de domínio, obrigatórios ou opcionais, selecionados para executar uma aplicação. Os serviços de domínio são invocados pelo serviço do processo do negócio.
- Repositório de dados: contém os dados e os metadados que podem ser usados pelos serviços da camada de aplicação.

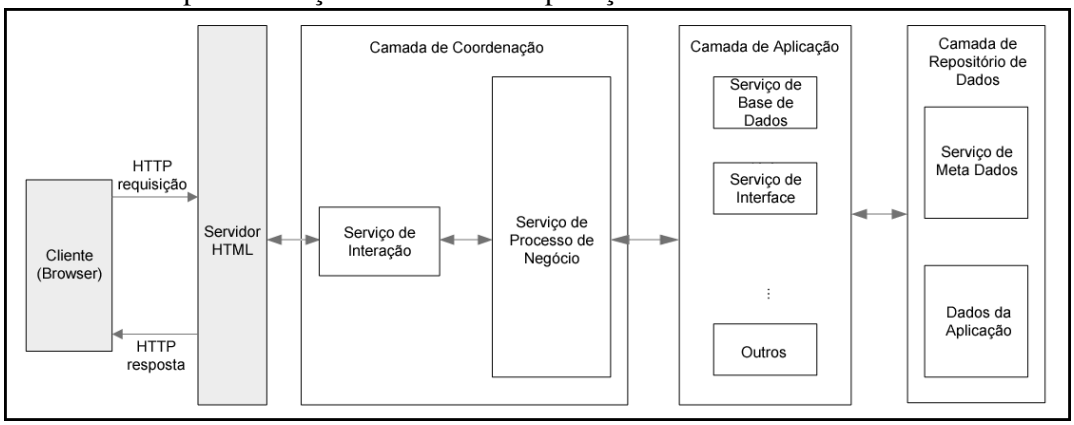


Figura 2. Visão geral da arquitetura lógica do WIDE-PL.

A seqüência para execução de uma aplicação de acordo com a arquitetura lógica de WIDE-PL é: (i) a aplicação é iniciada na camada de coordenação ativando o serviço de processo de negócio; (ii) o serviço de processo de negócio faz, de acordo com o fluxo da especificação da aplicação, solicitações a outros serviços contidos na camada de aplicação; (iii) os serviços contidos na camada de aplicação podem acessar a camada de repositório de dados na busca por dados e metadados referentes à aplicação, como é o caso do serviço de base de dados; (iv) após as solicitações para carregar a página, o serviço de processo de negócio solicita ao serviço de interface que gere o *script* de página a ser exibido ao usuário. Depois de gerado, o *script* é repassado do serviço de processo de negócio ao serviço de interação, que o exibe ao usuário por meio de seu *browser*; e (v) o usuário pode fazer as interações pertinentes e enviar as requisições novamente ao serviço de interação, que as tratam e repassam ao serviço de processo de negócio para continuar o fluxo da especificação do processo de negócio da aplicação, consistindo no retorno ao passo (ii). Os passos descritos ocorrem repetidas vezes até o encerramento do serviço de processo do negócio, quando a aplicação é encerrada.

No ambiente WIDE-PL, o serviço de processo do negócio é o único ponto de controle e responsável por coordenar a execução da aplicação, assim o seu o gerenciamento é por orquestração (Peltz, 2003). Dessa forma, o serviço de processo de negócio contém o motor de execução e os outros serviços utilizados na aplicação

possuem apenas uma infra-estrutura básica capaz de executar suas partes no processo e comunicar-se com o processo de negócio.

5. O Processo ADESE

A Figura 3 apresenta o processo ADESE representado graficamente por um diagrama de blocos SADT (Ross; Schoman, 1977) que contém as três etapas do processo e as informações manipuladas por ele. Essas etapas são descritas a seguir.

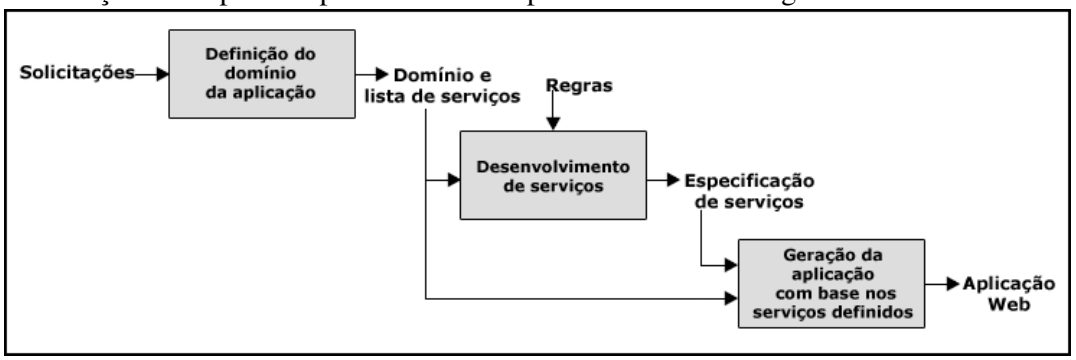


Figura 3. Representação gráfica do processo ADESE.

Etapa 1 - Definição do domínio da aplicação

Nesta etapa é definido o domínio da aplicação por meio de uma descrição textual simples. Esta descrição é tomada como base para realizar a análise de domínio que consiste em gerar os modelos de características que determinam a lista dos respectivos serviços obrigatórios e opcionais necessários para o desenvolvimento da aplicação *Web*.

Etapa 2 - Desenvolvimento de serviços

Esta etapa deve ser executada cada vez que um novo serviço é necessário em uma aplicação *Web*. Caso os serviços necessários já estejam disponíveis em WIDE-PL deve-se passar a etapa seguinte. Para o desenvolvimento de cada novo serviço é necessário:

- Definir o modelo de características do serviço: consiste definir o modelo por meio da captura de seus atributos e variabilidades.
- Instanciar o modelo de características: consiste em selecionar uma configuração do modelo contendo as características obrigatórias e opcionais a serem utilizadas na aplicação.
- Mapear o modelo de características para o diagrama de classes UML: consiste em converter o modelo de características em um correspondente diagrama de classes identificando os elementos que compõem o diagrama como classes, relacionamentos, atributos e operações.
- Implementar o serviço a partir do diagrama de classes: consiste nos seguintes passos: (i) programar as classes encontradas no diagrama de classes; (ii) converter as classes em um serviço; e, (iii) especificar o serviço gerado em uma interface padronizada.

O mapeamento do modelo de características para o diagrama de classes visa converter um modelo que resume as capacidades comuns e diferentes de uma aplicação, para um diagrama que representa a sua estrutura física. As regras definidas neste processo para conversão do modelo de características para diagrama de classes são:

- Identificar classes: criar uma classe para cada característica não "folha" do modelo de características.

- Identificar relacionamentos: adicionar o relacionamento de agregação entre as classes que tem relacionamento no modelo de características, sendo que a classe agregadora é a classe que, no relacionamento, tem nível mais alto no modelo de características.

- Identificar operações e atributos: transformar as características "folhas" do modelo de características em operações ou atributos das classes já criadas, sendo que cada característica "folha" que será transformada em operação ou atributo deverá ser incluída na classe que representa sua respectiva característica de nível mais alto no modelo de características.

- Identificar operações da classe principal: a característica raiz do modelo de características é a classe principal do diagrama de classes e as operações incluídas nesta classe serão as operações convertidas em operações do serviço. Assim, caso alguma outra classe do diagrama contenha operações que devam ser operações do serviço, estas operações deverão ser incluídas na classe principal, e na implementação, as operações contidas na classe principal deverão fazer a invocação da operação original nas outras classes.

- Incluir operações e atributos auxiliares: operações e atributos adicionais como os casos das operações *get* e *set* podem ser incluídos no diagrama para auxiliarem posteriormente na execução de alguma operação do serviço.

Etapa 3 - Geração da aplicação com base nos serviços definidos

Esta etapa visa gerar uma aplicação *Web* baseada em serviços de acordo com o domínio escolhido na Etapa 1. As atividades que compõe esta etapa são:

- Criação de um processo de negócio síncrono: consiste em criar a estrutura básica do processo de negócio da aplicação. Esta estrutura básica deve conter informações como as definições iniciais do processo, variáveis iniciais, serviços participantes iniciais e uma definição vazia do fluxo do processo. Como se trata de um processo síncrono, este será iniciado pela execução de uma operação contida no serviço de processo de negócio e encerrado após a execução de todo seu fluxo, retornando alguma informação à operação do serviço de processo de negócio que o iniciou.

- Adição de serviços: consiste em adicionar à especificação do processo de negócio da aplicação, os serviços necessários para a geração da aplicação. Os passos para adicionar os serviços são: (i) importar para a ferramenta de desenvolvimento de escolha do desenvolvedor a especificação de cada um dos serviços necessários para geração da aplicação; (ii) configurar a especificação de cada serviço necessários para a geração da aplicação incluindo uma extensão da linguagem, que são códigos da linguagem usados apenas em casos específicos, com a finalidade de permitir a comunicação entre o processo de negócio e os seus respectivos serviços; e, (iii) adicionar os serviços já configurados ao processo de negócio.

- Especificação do processo de negócio: consiste em complementar a especificação do processo de negócio com o fluxo da aplicação de acordo com

as regras da linguagem utilizada, as regras de negócio do domínio da aplicação e as regras para execução, que estabelecem a seqüência em que os serviços devem executados, de acordo com a arquitetura lógica do WIDE-PL.

6. Avaliação do Processo ADESE

Esta seção apresenta a avaliação do processo ADESE por meio de um estudo de caso, cujo desenvolvimento completo pode ser encontrado em Zaupa (2007). Este estudo consistiu de duas aplicações *Web* independentes provenientes de diferentes domínios, sendo uma do domínio de controle de estoques e outra do domínio de locação de vídeos. As aplicações concebidas utilizam os mesmos serviços obrigatórios, porém seus respectivos serviços de processo de negócio são compostos de acordo com as regras específicas de cada domínio. Além disso, as aplicações possuem serviços opcionais que são necessários a cada domínio. As ferramentas de apoio utilizadas no desenvolvimento do estudo de caso foram: (i) Netbeans IDE 5.5 (Netbeans, 2006) com a função de programar, compilar e executar os serviços e os processos de negócio; (ii) *Feature Plug-in* com a função de definir o modelo de características; e, (iii) o banco de dados MySQL com a função de armazenar e buscar dados usados nas aplicações.

A seguir serão apresentadas as etapas do processo ADESE para o desenvolvimento da aplicação de locação de vídeo, bem como comentários sobre as partes que seriam comuns para o desenvolvimento da aplicação de controle de estoque.

Etapa 1 - Definição do Domínio da Aplicação

A aplicação deve controlar a locação de vídeos, assim deve possuir cadastros de clientes, filmes, fitas, controle de locações, reservas e pagamentos, além de controle de acesso dos usuários da aplicação. Para o desenvolvimento desta aplicação serão necessários os serviços obrigatórios já disponíveis em WIDE-PL e um serviço opcional de calendário que será responsável por informar datas e horários à aplicação, como a data do dia, o horário do momento, diferença de dias entre datas, calcular uma data a partir da soma de dias, entre outras.

Etapa 2 - Desenvolvimento de Serviços

Nesta etapa é apresentado o desenvolvimento do serviço opcional de calendário para ilustrar o desenvolvimento de um serviço não disponível em WIDE-PL. Para tal, é necessário definir o modelo de característica, instancia-lo, mapeá-lo para o diagrama de classes correspondente e posteriormente implementa-lo. A Figura 4A apresenta o modelo de características obtido para o serviço de calendário. Este modelo contém a característica `Calendário` como característica raiz, e possui o restante das características divididas em dois grupos maiores chamados `Data` e `Horário`, sendo que as características `Hoje`, `Dia`, `Mês` e `Ano`, contida no grupo `Data`, e `Agora`, `Hora` e `Minuto`, contida no grupo `Horário`, são características obrigatórias. Em seguida, o modelo de característica do serviço de calendário foi instanciado definindo-se as características opcionais a serem utilizadas.

O próximo passo consiste do mapeamento do modelo de características instanciado para um diagrama de classes. A Figura 4B apresenta o diagrama de classes produzido seguindo as regras de conversão descritas na seção 5 (Etapa 2).

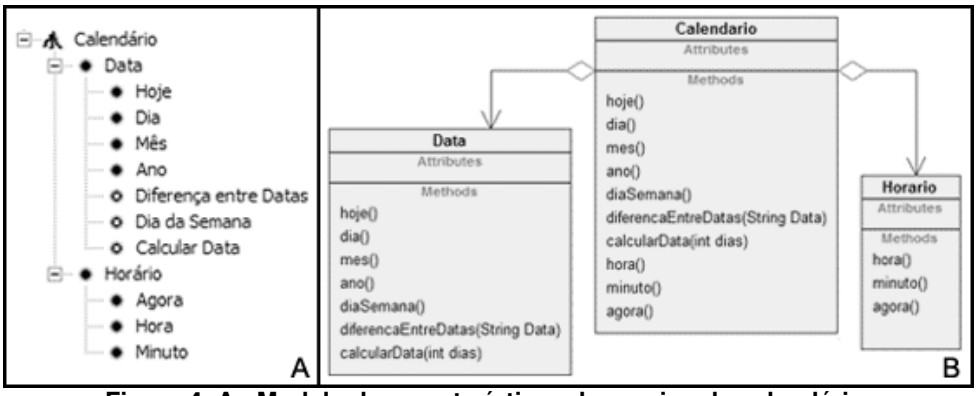


Figura 4. A - Modelo de características do serviço de calendário, B – Respectivo diagrama de classes.

Obtido o diagrama de classes, deve-se implementar o serviço. Para tal, é preciso programar e compilar as classes que compõem o diagrama. Esta programação é feita manualmente pelo desenvolvedor com auxílio de alguma ferramenta de desenvolvimento, como neste estudo de caso, a ferramenta NetBeans (2006). Em seguida, deve-se converter as classes em serviço. A conversão de classes em serviços também depende da ferramenta utilizada. A Figura 5 apresenta parte da estrutura criada pelo NetBeans para o serviço de calendário. A pasta `Serviços Web` contém o serviço, denominado `wsCalendario`, com suas respectivas operações. Estas são as mesmas operações que estão na classe `Calendário`. Após o desenvolvimento do serviço, é necessário especificar sua interface. Conforme já descrito, esta especificação é feita em WSDL. Algumas ferramentas, como é o caso da ferramenta NetBeans, criam automaticamente o documento WSDL correspondente.

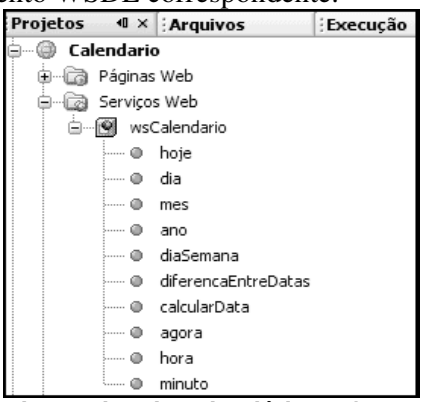


Figura 5 - Estrutura do serviço de calendário na ferramenta Netbeans 5.5.

Etapa 3 - Geração da Aplicação com Base nos Serviço Definidos

Nesta etapa são apresentadas as atividades realizadas e alguns trechos da especificação em BPEL. A primeira atividade é a criação de um processo de negócio síncrono. O processo criado representa o serviço obrigatório de processo de negócio, contendo uma única operação, que quando executada, iniciará a execução do processo de negócio da aplicação de locação de vídeos. A estrutura básica do novo processo contém descrições em BPEL como as definições iniciais do processo, variáveis iniciais, serviços participantes iniciais e a uma definição vazia do fluxo do processo.

Após a criação do processo, é preciso adicionar os serviços necessários. Para tal, deve-se importar sua especificação para a ferramenta de desenvolvimento para que

possam ser configurados. Esta configuração consiste em incluir em cada especificação de serviço, uma extensão da linguagem, com a finalidade de estabelecer a comunicação entre o processo de negócio e o respectivo serviço. O passo final é adicionar os serviços já configurados ao processo. A Listagem 1 apresenta a descrição em BPEL em que aparecem os serviços participantes. Cada linha entre os elementos `<partnerLinks>` e `</partnerLinks>` representa um serviço, caracterizado por um nome e uma referência a sua especificação em WSDL que identificam as suas operações.

```
<partnerLinks>
  <partnerLink name="Calendario" partnerLinkType="ns5:wsCalendarioSEILinkType" partnerRole="wsCalendarioSEIRole"/>
  <partnerLink name="Database" partnerLinkType="ns4:wsDatabaseSEILinkType" partnerRole="wsDatabaseSEIRole"/>
  <partnerLink name="Interaction" partnerLinkType="ns3:wsInteractionSEILinkType" partnerRole="wsInteractionSEIRole"/>
  <partnerLink name="Interface" partnerLinkType="ns2:wsInterfaceSEILinkType" partnerRole="wsInterfaceSEIRole"/>
  <partnerLink name="BusinessProcess" partnerLinkType="ns1:partnerlinktype1" myRole="partnerlinktyperole1"/>
</partnerLinks>
```

Listagem 1. Descrição BPEL dos serviços incluídos no processo de negócio.

Para fazer a especificação do processo de negócio, é preciso inicialmente declarar as variáveis e atribuir valores aos elementos usados durante toda a execução da aplicação, como os valores de conexão com a base de dados, cabeçalho, rodapé, menu e estilo da página. A Listagem 2 apresenta um trecho BPEL em que são atribuídos valores respectivamente às variáveis de `login`, `password`, `localhost` e `drive`, e na seqüência é invocada a operação `connect` do serviço de base de dados enviando as variáveis como parâmetro para validar a conexão com a base de dados.

```
<assign name="Assign1">
        <copy><from>string('')</from><to>$ConnectIn1.parameters/String_1</to></copy>
        <copy><from>string('')</from><to>$ConnectIn1.parameters/String_2</to></copy>
        <copy><from>string('jdbc:mysql://localhost/mestrado')</from><to>$ConnectIn1.parameters/String_3</to></copy>
        <copy><from>string('com.mysql.jdbc.Driver')</from><to>$ConnectIn1.parameters/String_4</to></copy>
 </assign>  <invoke name="ConnectDB" partnerLink="Database" operation="connect" portType="ns2:wsDatabaseSEI"
inputVariable="ConnectIn1" outputVariable="ConnectOut1"/>
```

Listagem 2. Trecho BPEL que invoca a operação connect do serviço de base de dados.

O corpo da página deve ser sempre atualizado com informações pertinentes às interações com o usuário. Para isso é necessário enviar as informações que deverão ser mostradas no corpo da página ao serviço de interface. Após o envio dessas informações ao serviço, a operação `format`, que faz a formatação e gera o *script* de página deve ser invocada. O *script* gerado retorna ao processo de negócio que o encaminha ao serviço de interação para ser exibido ao usuário. A Listagem 3 apresenta a situação descrita.

```
<invoke name="Page1" partnerLink="Interface" operation="format" portType="ns2:wsInterfaceSEI"
inputVariable="FormatFalse" outputVariable="Page"/>
<assign name="Ass3"><copy><from>$Page.result/result</from><to>$ResultInteraction.result/result</to></copy></assign>
<invoke name="Interaction1" partnerLink="Interaction" operation="requestResponse" portType="ns3:wsInteractionSEI"
inputVariable="PageInteraction" outputVariable="ResultInteraction"/>
```

Listagem 3. Trecho BPEL que formata a página e envia ao serviço de interação.

Após a interação com o usuário, as requisições serão tratadas e repassadas ao processo de negócio para continuar o seu fluxo. Para cada uma das funções da aplicação como cadastrar, locar, efetuar pagamento e reservar filmes, os procedimentos são semelhantes aos citados. Para casos em que é necessário exibir uma lista de dados que se encontra na base de dados, a única diferença é recuperar os dados antes de exibi-los através do serviço de base de dados, e repassá-lo ao serviço de interface. Para finalizar o processo de negócio e consequentemente a aplicação, é necessário selecionar a opção correta no menu da aplicação para levar ao fechamento dos laços.

A especificação do processo de negócio da aplicação de controle de estoque segue os mesmos procedimentos descritos para a aplicação de locação de vídeos. A diferença entre a aplicação de locação de vídeos, a aplicação controle de estoque e qualquer outro sistema de informação que venha a ser desenvolvida seguindo o processo ADESE está nas regras de negócio da aplicação. Dessa forma, para cada aplicação devem ser definidas as configurações do processo como cores e fontes, a base de dados que será usada, as informações que serão exibidas no cabeçalho, rodapé, menu e principalmente no corpo da página. A estrutura da linguagem de especificação e as regras para execução da arquitetura lógica de WIDE-PL não mudam de uma aplicação para outra, o que torna a concepção de aplicações *Web* muito flexível.

Finalmente, é necessário um motor BPEL para executar as especificações dos processos de negócio das aplicações. Embora, alguns motores BPEL tenham sido testados no contexto deste trabalho como: o motor embutido no Netbeans 5.5 (Netbeans, 2006), Bexee (Bexee, 2007) e PXE (PXE, 2007), nenhum se mostrou adequado, ou por dificuldades de acesso, ou instalação ou disponibilidade. Assim, optou-se por simular a execução dos processos de negócio em um *servlet* Java, sendo apenas um *servlet* por aplicação, e nele invocar as operações do serviço na mesma seqüência em que são invocados no processo de negócio descrito em BPEL. Os *servlets* das aplicações são ativados pela chamada de seu endereço no *browser*. Soluções mais recentes permitem inclusive a conversão automática de BPEL para Java (B2J, 2007).

6.1 Resultados

Após a realização do estudo de caso foi feita uma análise qualitativa. Esta análise foi realizada comparando-se alguns aspectos do processo ADESE com os métodos tradicionais de desenvolvimento de aplicação *Web*. Os aspectos analisados mostram o potencial da abordagem como segue:

- Reutilização: há indicação de ganho do processo ADESE porque os serviços, uma vez desenvolvidos para uso em uma aplicação *Web*, ficam disponíveis no ambiente WIDE-PL para uso posterior em outras aplicações.

- Separação explícita da lógica do negócio e dos serviços: um potencial de ganho pode ser evidenciado porque a abordagem separa completamente a lógica do negócio, que fica no serviço de processo de negócio, das funções necessárias para o desenvolvimento da aplicação que estão embutidas nos serviços e são invocadas quando necessário pelo processo de negócio.

- Redução de tempo e custo de desenvolvimento: há indicação de ganho porque o maior esforço de desenvolvimento passa a ser no desenvolvimento do processo de negócio da aplicação. Os serviços necessários para compor a aplicação são adicionados ao processo para serem invocados quando for preciso, poupando muita modelagem e programação das funções que constam nesses serviços, reduzindo assim tempo e consequentemente custo.

Entre alguns requisitos não funcionais também se pode citar indicações de ganhos e perdas com o uso do processo ADESE em relação aos métodos tradicionais. Exemplos de ganho são: interoperabilidade porque permite que serviços produzidos em diferentes linguagens e plataformas se comuniquem entre si; e, manutenibilidade porque a aplicação é composta por serviços independentes e com poucas e bem definidas dependências entre si facilitando a manutenção. Exemplos de perda são: desempenho

porque os serviços utilizam XML como formato padrão para a troca de mensagens, e estes são bem maiores que o equivalente em formato binário; e segurança porque aumenta o risco de interceptação dos dados, já que serviços trocam dados pela *Web*. Pode ocorre perda em: disponibilidade porque a aplicação normalmente utilizará serviços que estão distribuídos em vários servidores, e isso aumenta o risco que em algum momento algum desses serviços não esteja disponível.

7. Conclusões

Este artigo apresenta o processo ADESE para desenvolvimento de aplicações *Web* baseado em serviços no contexto de WIDE-PL. Como contribuições, destacam-se a definição e especificação do processo, que inclui em sua etapa de desenvolvimento de serviços, regras de mapeamento do modelo de características para o diagrama de classes. Na literatura, ainda há uma carência de abordagens de mapeamento do modelo de características em outros modelos. Um exemplo é a abordagem proposta por Czarnecki e Antkiewicz (2005), porém o foco é no mapeamento do modelo de características para diagrama de atividades UML. Os resultados obtidos mostram vantagens em relação aos métodos tradicionais. A principal delas é a separação explícita da lógica do negócio e dos serviços. Além disso, há indicações de vantagens em aspectos como: reutilização; redução de tempo e custo de desenvolvimento; interoperabilidade; e, manutenibilidade. O processo ADESE mostra que é possível um desenvolvimento mais automatizado de aplicações *Web* baseado em serviços, e assim abre caminho para uma nova abordagem. Nesta abordagem o foco é no processo de negócio, e serviços são adicionados e suas operações são invocadas de acordo com as regras de negócio do domínio da aplicação. Algumas desvantagens foram observadas como o atual estágio da tecnologia em que ainda falta um motor BPEL eficiente para executar os processos de negócio das aplicações, bem como em requisitos como desempenho, segurança e disponibilidade. Porém, essas desvantagens podem ser superadas com o avanço da tecnologia.

Referências

Antkiewicz, M.; Czarnacki, K. (2004), **FeaturePlugin: Feature Modeling Plug-in for Eclipse,** In Eclipse '04: Proceedings of the OOPSLA, Canada, ACM Press.

B2J (2007), **BPEL to Java Subproject Eclipse**, http://www.eclipse.org/stp/b2j, Março.

BEA Systems, IBM, Microsoft, SAP AG, Siebel Systems (2003), **Business Process Execution Language for Web Services Version 1.1**.

Bexee (2007), **Bexee BPEL Execution Engine,** http://sourceforge.net/projects/bexee, Janeiro.

Casati, F., Shan, M-C (2000), **Process Automation as the Foundation for E-Business,** In: Proceedings of 26th International Conference on Very Large Databases, Egito.

Ceri, S. et al. (2003), **Architectural Issues and Solutions in the Development of Data-Intensive Web Applications**, Proc. of CIDR'03, USA.

Christensen, E., Curbera, F., Meredith, Weerawarana, Sanjiva (2001), **Web Services Description Language (WSDL)** 1.1, W3C Note 15.

Clements, P., Northrop, L. (2001), **Software product lines: practices and patterns,** 1ed, Boston: Addison-Wesley, p. 608.

Cowan, D., Fenton, S., Mulholland, D. (2004), **The Waterloo Informatics Development Environment (WIDE)**. CSG Internal Note.

Czarnecki, K., Helzen, S.; Eisenecker, U. (2005), **Staged configuration through specialization and multi-level configuration of feature models. To appear in special issue on "Software Variability: Process and Management"**, Software Process Improvement and Practice, 10(2).

Czarnecki, K., Antkiewicz, M. (2005), **Mapping features to models: A template approach based on superimposed variants,** GPCE 2005, v. 3676, p. 422, Springer.

Fantinato, M., Toledo, M., Gimenes, I (2005), **Arquitetura de Sistemas de Gerenciamento de Processos de Negócio Baseado em Serviços,** Rel.T., Unicamp.

Gimenes, I. et al (2005), **O projeto preliminar de WIDE-PL**, Relato de estágio de pós-doutorado na Universidade de Waterloo, Canadá, CAPES-MEC, Brazil.

Kang, K. (1990), **Feature-oriented domain analysis (FODA) - feasibility study**, Technical Report CMU/SEI-90-TR-21, SEI/CMU, Pittsburgh.

Kang, K., Lee, J., Donohoe, P. (2002), **Feature-oriented Product Line Engineering**, IEEE Software.

Netbeans (2006), **Netbeans IDE 5.5**, http://www.netbeans.org, Abril.

Papazoglou, M., Georgakopoulos, D. (2003), **Service-oriented computing**, Communications of the ACM: Service-Oriented Computing.

Pastor, O., Fons, J., Pelechano, V. (2003), **OOWS**, Department of Information Systems and Computation Technical University of Valencia.

Peltz, C. (2003), **Web Services Orchestration and Choreography**, HP Company.

PXE (2007), **Process Execution Engine,** http://sourceforge.net/projects/pxe, Janeiro.

Ross, D., Schoman, K. (1977), **Structured Analysis for Requirements Definition**, IEEE Transactions on Software Engineering 3(1).

Schwabe, D; Rossi, G (1998), **Developing Hypermedia Applications using OOHDM**, Hypermedia Development Processes, Methods and Models, Hypertext, USA.

SEI (2006), Software Engineering Institute, **A framework for software product line practice 4.2,** http://www.sei.cmu.edu/productlines/framework.html, Abril.

Sochos, P; Philipow, I; Riebish, M (2004) **Feature-oriented development of software product lines: mapping feature models to the architecture.** Springer, p.138.

SOAP (2006), **Simple Object Access Protocol,** http://www.w3.org/TR/SOAP, Abril.

UDDI (2006), **Universal, Description, Discovery and Integration,** http://www.uddi.org, Abril.

van Gurp, J., Bosch, J., Svahnberg, M.(2001), **On the notion of variability in software product lines**, in: Proc. The Working IEEE/IFIP, WICSA, The Netherlands.

XML (2006), **Extensible Markup Language,** http://www.w3.org/TR/REC-xml, Maio.

Zaupa, F. (2007), **Um Processo de Desenvolvimento de Aplicações Web baseado em Serviços**, Dissertação de Mestrado em Ciência da Computação, UEM, Maringá.

Technical Session III: Software Architectures and Components

Comparando Modelos Arquiteturais de Sistemas Legados para Apoiar a Criação de Arquiteturas de Referência de Domínio

Aline P. V. de Vasconcelos[1,2], Guilherme Z. Kümmel[1], Cláudia M. L. Werner[1]

[1]COPPE/UFRJ – Programa de Engenharia de Sistemas e Computação
Caixa Postal: 68511 – CEP: 21941-972 – Rio de Janeiro – RJ – Brasil

[2]CEFET Campos (Centro Federal de Educação Tecnológica de Campos)
Rua Dr. Siqueira, 273 – Pq. Dom Bosco – CEP: 28030-130 - Campos dos Goytacazes-RJ - Brasil

{aline,kummel,werner}@cos.ufrj.br

Abstract. *Large organizations usually have legacy systems that represent effort and resources invested, besides encompassing rich business knowledge. They frequently develop systems of the same domain, what has been motivating the migration to reuse approaches such as Domain Engineering (DE) and Product Line (PL). Within a certain domain, a reference architecture (DSSA) represents the basis for application instantiation. In this context, legacy systems are essential information sources for domain specification. Therefore, this paper presents an approach for legacy system architectures comparison to support the creation of DSSAs, which can be used in the context of DE and LP.*

Resumo. *Grandes empresas costumam possuir sistemas legados que representam esforço e recursos investidos e embutem rico conhecimento sobre o negócio. Em geral, elas desenvolvem sistemas no mesmo domínio, motivando a migração para abordagens de reutilização como Engenharia de Domínio (ED) e Linha de Produtos (LP). Num dado domínio, a arquitetura de referência (DSSA) representa a base para a instanciação de aplicações. Nesse contexto, os sistemas legados são fontes de informação essenciais para a especificação do domínio. Assim, este artigo apresenta uma abordagem de comparação de arquiteturas de sistemas legados para apoiar a criação de DSSAs, as quais podem ser utilizadas no contexto da ED e LP.*

1. Introdução

A maioria das organizações que desenvolve software costuma construir sistemas de software em um domínio de aplicação particular, repetidamente entregando variantes de produtos pela adição de novas características (SUGUMARAN *et al.*, 2006). Dessa forma, é comum que essas organizações estejam migrando para abordagens de Engenharia de Domínio (ED) (PRIETO-DIAZ & ARANGO, 1991) e Linha de Produtos (LP) (LEE *et al.*, 2002). Uma vez que as arquiteturas de referência de domínio representam o elemento-chave para uma abordagem de ED ou LP de sucesso, a construção desses artefatos merece atenção nesse processo de migração.

As arquiteturas de referência de domínio representam a base para a instanciação de aplicações, reutilização e adaptação dos artefatos de domínio produzidos. Elas são o elemento central das DSSAs (*Domain Specific Software Architectures*), que, segundo XAVIER (2001), são mais do que uma arquitetura para um determinado domínio de aplicações, mas sim uma coleção de elementos arquiteturais especializados para um determinado tipo de tarefa (domínio) e generalizados para que seja possível seu uso efetivo através do domínio. Vale ressaltar, entretanto, que neste trabalho os termos arquitetura de referência de domínio e DSSA são utilizados como sinônimos. Tanto na ED quanto na LP, as arquiteturas de referência de domínio devem representar os elementos arquiteturais do domínio, seus relacionamentos, semelhanças e diferenças.

A fim de apoiar a criação dessas arquiteturas de referência de domínio, sistemas legados disponíveis para o domínio podem ser analisados. Grandes empresas costumam possuir uma base de sistemas legados que representam esforço e recursos investidos no passado, além de embutirem conhecimento sobre o negócio que muitas vezes não pode ser obtido de nenhuma outra fonte de informação, o que motiva a sua evolução contínua e reutilização em novos esforços de desenvolvimento. Entretanto, as abordagens de ED e LP existentes não costumam prover processos sistemáticos com técnicas de apoio à análise de um conjunto de sistemas legados no domínio, a fim de detectar as suas semelhanças e diferenças (ex: FORM (KANG *et al.*, 2002), CBD-Arch-DE (BLOIS, 2006), KobrA (ATKINSON *et al.*, 2002) e PLUS (GOMAA, 2004)). Algumas abordagens, como a de (GOMAA, 2004), comentam a importância de se extrair e realizar a consistência de modelos de aplicações do domínio, mas não indicam como fazê-lo.

Nesse contexto, este artigo apresenta uma abordagem de comparação de modelos arquiteturais de sistemas legados para apoiar a criação de arquiteturas de referência de domínio, que podem ser utilizadas no contexto da ED e da LP, denominada ArchToDSSA (KÜMMEL, 2007). ArchToDSSA está inserida em um contexto de trabalho mais amplo, sendo parte da abordagem LegaToDSSA (VASCONCELOS, 2007), a qual envolve uma fase de engenharia reversa para a recuperação das arquiteturas dos sistemas legados, i.e. ArchMine, e uma fase de comparação das arquiteturas recuperadas para a detecção das suas semelhanças e diferenças, e criação da DSSA, i.e. ArchToDSSA. A fase de engenharia reversa é importante porque, em geral, a documentação dos sistemas legados não se encontra atualizada em relação ao código.

ArchToDSSA também está inserida no contexto do projeto Odyssey (ODYSSEY, 2007), que visa o desenvolvimento de um ambiente de reutilização baseado em modelos de domínio de mesmo nome, i.e. o ambiente Odyssey. Dessa forma, ela adota a notação Odyssey-FEX (OLIVEIRA, 2006) para a representação das semelhanças e diferenças do domínio. Segundo OLIVEIRA (2006), as semelhanças e diferenças do domínio podem ser expressas através das suas variabilidades e opcionalidades. As variabilidades do domínio envolvem elementos arquiteturais invariantes, i.e. que não podem ser modificados na instanciação de aplicações, e elementos arquiteturais que representam pontos de variação (VPs), que podem ser configurados através da seleção de uma ou mais variantes. As opcionalidades do domínio são expressas através de elementos arquiteturais opcionais, i.e. que podem ou não ser selecionados na instanciação de aplicações, e de elementos arquiteturais mandatórios, que devem necessariamente ser selecionados na instanciação de aplicações

do domínio. As propriedades de variabilidade e opcionalidade são ortogonais na Odyssey-FEX, o que significa que um elemento que representa um VP, invariante ou variante pode ser ao mesmo tempo mandatório ou opcional.

Em relação à comparação dos modelos para a detecção das suas semelhanças e diferenças, elementos equivalentes entre modelos de diferentes sistemas podem estar descritos de forma diferente, possuindo, por exemplo, nomes diferentes, embora sendo semanticamente equivalentes. Isso porque apesar dos sistemas analisados fazerem parte de um mesmo domínio, eles não foram necessariamente desenvolvidos por uma mesma equipe, em um mesmo período de tempo ou dentro de um mesmo departamento. Abordagens de comparação de modelos para a detecção de diferenças costumam considerar diferentes versões de um mesmo modelo na comparação, onde os elementos equivalentes possuem o mesmo nome ou identificador (ex: (CHEN *et al.*, 2003) (MEHRA *et al.*, 2005) (OLIVEIRA, 2005)), sendo essas informações utilizadas como base para estabelecer a comparação. Como essas premissas não podem ser assumidas na comparação de modelos de diferentes sistemas legados em um domínio, a comparação se torna uma tarefa mais árdua, devendo levar em conta a grande variedade de elementos equivalentes com nomes e estruturas distintos encontrados nos diferentes sistemas analisados. Além disso, ArchToDSSA visa a detecção das variabilidades e opcionalidades do domínio, o que também não costuma ser o foco das abordagens existentes de comparação de modelos.

Partindo desta Introdução, o restante do artigo está organizado da seguinte forma: a Seção 2 discute trabalhos relacionados em ED e LP, descrevendo seu apoio à especificação de arquiteturas de referência, além de trabalhos de comparação de modelos; a Seção 3 apresenta a abordagem ArchToDSSA, descrevendo seu processo constituído de 3 fases, a saber: detecção de opcionalidades, detecção de variabilidades e criação de DSSA; a Seção 4 mostra um exemplo prático de utilização da abordagem e o seu apoio ferramental; e a Seção 5 apresenta conclusões e trabalhos futuros.

2. Trabalhos Relacionados

A maior parte das abordagens de ED e LP existentes costuma oferecer apoio à especificação de arquiteturas de referência de domínio no sentido da engenharia progressiva, i.e. partindo da análise para o projeto (ex: FORM (KANG *et al.*, 2002), CBD-Arch-DE (BLOIS, 2006), KobrA (ATKINSON *et al.*, 2002) e PLUS (GOMAA, 2004)). Assim, não contemplam o apoio à análise dos sistemas legados no processo, embora argumentem que esses representem uma das fontes de informação essenciais para a análise de domínio. Nesse contexto, ArchToDSSA oferece apoio à comparação das arquiteturas recuperadas de sistemas legados no domínio para a detecção das suas opcionalidades e variabilidades e, conseqüentemente, criação da DSSA.

As abordagens de ED e LP que focam na engenharia reversa, em geral, não oferecem um apoio sistemático à extração e análise de modelos dos sistemas legados para apoiar a criação de arquiteturas de referência de domínio, estando focadas em algum aspecto específico, como a extração ou avaliação da adequação de componentes à arquitetura de referência (GANESAN & KNODEL, 2005; KNODEL & MUTHIG, 2005) ou detecção e refatoração de variabilidades no código (ALVES *et al.*, 2007). Em contrapartida, ArchToDSSA provê técnicas e critérios para a detecção de

opcionalidades, variabilidades e criação/exportação da DSSA, complementando, no contexto da abordagem LegaToDSSA (VASCONCELOS, 2007), um processo sistemático de apoio à extração e análise de modelos de sistemas legados no domínio.

Finalmente, as abordagens de comparação de modelos para a detecção de diferenças costumam considerar diferentes versões de um mesmo modelo na comparação, onde os elementos equivalentes devem possuir o mesmo nome, tipo ou identificador (ex: (CHEN *et al.*, 2003) (MEHRA *et al.*, 2005) (OLIVEIRA, 2005)). Dessa forma, não contemplam arquiteturas de diferentes sistemas, onde elementos equivalentes podem possuir nomes e estruturas distintos. ArchToDSSA, por outro lado, incorpora um dicionário de sinônimos que permite que elementos com nomes distintos mas semântica igual sejam considerados equivalentes. Além disso, possibilita a comparação de nomes por *substrings* comuns e permite considerar ou não o tipo e a estrutura sintática dos elementos durante a comparação.

3. ArchToDSSA: Comparação de Modelos Arquiteturais de Sistemas Legados para Apoiar a Criação de Arquiteturas de Referência de Domínio

ArchToDSSA visa a comparação de modelos arquiteturais recuperados de sistemas legados para apoiar a criação de arquiteturas de referência de domínio, que podem ser utilizadas no contexto da ED e da LP, e a partir da qual novas aplicações no domínio podem ser instanciadas. A abordagem pode ser adotada, por exemplo, por empresas que possuam uma base de sistemas legados e estejam pretendendo migrar para abordagens de ED e de LP.

A fim de identificar elementos opcionais ou mandatórios e variantes, invariantes ou VPs, gerando arquiteturas de referência, a abordagem ArchToDSSA propõe um processo de 3 fases, conforme mostra a Figura 1. O processo envolve a Detecção de Opcionalidades, Detecção de Variabilidades e Criação da DSSA. O papel responsável pela execução das atividades é o Engenheiro de Domínio ou o engenheiro em uma abordagem de LP, responsável pela elaboração da arquitetura de referência de domínio. Vale ressaltar que ArchToDSSA gera arquiteturas de referência parcialmente especificadas, podendo gerar diferentes arquiteturas a partir de um mesmo conjunto de arquiteturas legadas, sendo gerada, entretanto, uma arquitetura por vez. As DSSAs devem ter a sua especificação complementada pelo engenheiro que apóia o processo.

O processo apresentado na Figura 1 segue a notação do metamodelo SPEM (*Software Process Engineering Metamodel*) (OMG, 2005), definido pela OMG, que faz uso de uma notação estendida de diversos modelos da UML com o intuito de permitir a modelagem de processos. Como pode ser visto na Figura 1, que representa uma extensão do diagrama de atividades da UML, uma vez executado um processo de engenharia reversa, como, por exemplo, ArchMine (VASCONCELOS, 2007), que gere um conjunto de Arquiteturas Recuperadas de Sistemas no Domínio, a atividade de Detecção de Opcionalidades pode ser iniciada. Ela analisa as arquiteturas recuperadas e gera como saída uma lista de Elementos Equivalentes no domínio, que representam elementos semanticamente correspondentes nas diferentes arquiteturas analisadas, e as Arquiteturas Recuperadas com as suas Opcionalidades. Com base nesses artefatos gerados pela primeira atividade do processo, a atividade de Detecção de Variabilidades detecta os pontos de variação e variantes do domínio, gerando Arquiteturas Recuperadas

com Opcionalidades, VPs e Variantes. A Criação da DSSA utiliza as informações geradas ao longo do processo para gerar Arquiteturas de Referência de Domínio parcialmente especificadas.

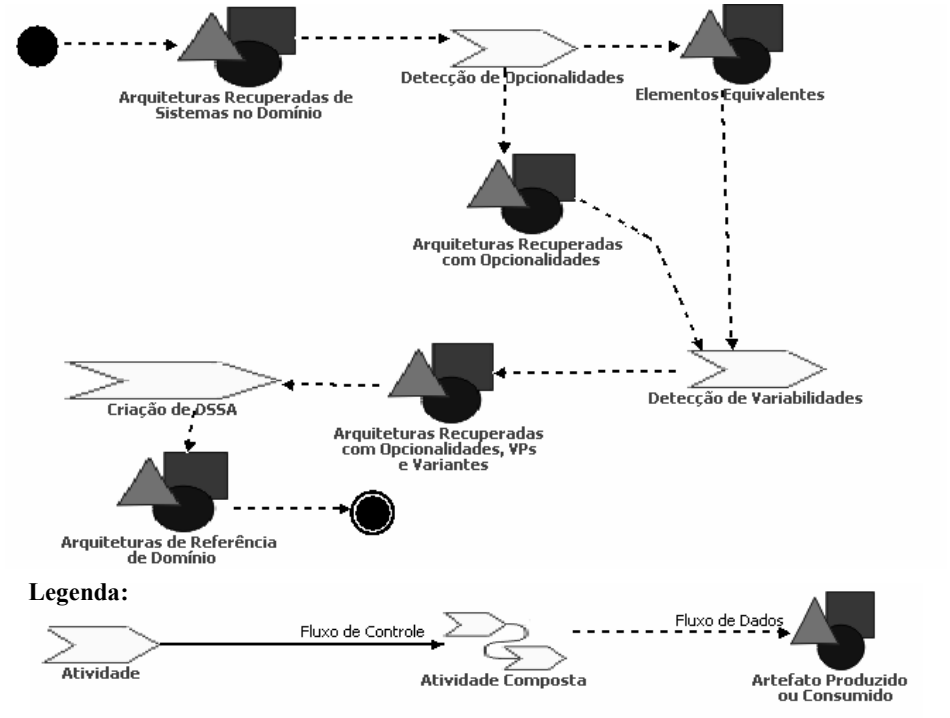


Figura 1. Processo proposto da abordagem ArchToDSSA

ArchToDSSA assume como pré-requisitos que os sistemas sejam Orientados a Objetos (OO) e que as suas arquiteturas estejam representadas em UML, descritas em formato XMI. Assim, os elementos arquiteturais são representados através de pacotes e a sua estrutura interna através de classes, com os seus relacionamentos. As subseções a seguir descrevem detalhadamente as fases ou atividades do processo de ArchToDSSA.

3.1. Detecção de Opcionalidades

Nesta primeira fase da abordagem, o objetivo é identificar dentre as arquiteturas recuperadas dos sistemas legados, as similaridades e diferenças entre seus respectivos elementos arquiteturais. Dessa forma, é possível identificar elementos mandatórios e opcionais no domínio, ou seja, as opcionalidades do domínio em questão. Essas opcionalidades são identificadas tanto em nível de elemento arquitetural quanto em nível de suas classes (estrutura interna).

Para a identificação de elementos mandatórios e opcionais, o primeiro passo é a identificação de equivalências (*matches*). Equivalências representam elos de ligação entre elementos semanticamente correspondentes de diferentes arquiteturas. Elementos equivalentes podem apresentar o mesmo nome ou nomes diferentes nas arquiteturas comparadas em função delas se originarem de diferentes aplicações do domínio. A Figura 2 apresenta um exemplo considerando um domínio escolar, onde "Aluno" e "Estudante", por exemplo, representam o mesmo conceito, embora apresentando nomes

diferentes.

Essa equivalência semântica em ArchToDSSA é obtida através de um dicionário de sinônimos que é alimentado pelo Engenheiro de Domínio. A cada ligação estabelecida pelo Engenheiro de Domínio entre elementos de diferentes arquiteturas com nomes diferentes, uma entrada no dicionário é criada, definindo esses nomes como sinônimos. O Engenheiro de Domínio pode ainda definir diretamente sinônimos no dicionário sem necessariamente estar efetuando ligações entre as arquiteturas. Assim, a comparação das equivalências com novas arquiteturas no mesmo domínio é simplificada através do dicionário, o qual é constantemente alimentado durante o processo.

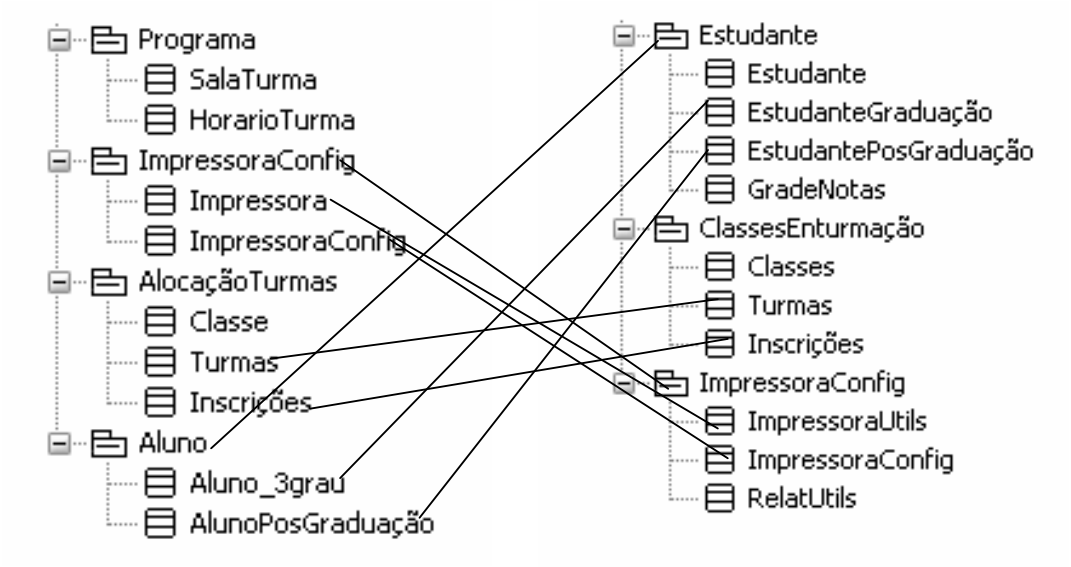


Figura 2. Equivalências entre diferentes arquiteturas de um domínio

Além da comparação de nomes iguais ou sinônimos, a abordagem admite ainda 3 possibilidades de configuração para a determinação de equivalências:

- **Lista de Palavras Ignoradas**: ArchToDSSA permite que palavras sem semântica "forte", como Gerente, Utils, Controlador etc., sejam ignoradas na detecção de equivalências. Essas palavras não contribuem semanticamente com a comparação. Assim, uma lista de palavras ou abreviações que devem ser ignoradas é mantida.
- **Comparação por *Substrings***: é possível que os nomes sejam divididos em partes durante a comparação. Os critérios para a divisão em partes envolvem a detecção de uma letra maiúscula ou um sinal de sublinhado. Dessa forma, no nome "ImpressoraUtils" poderão existir duas partes, i.e. "Impressora" e "Utils", e no nome "AlunoPosGraduação" três partes, i.e. "Aluno", "Pos" e "Graduação". Após dividir os nomes em partes, é verificado para cada parte (em cada nome), se a referida parte se encontra na "lista de palavras a serem ignoradas", caso se encontre, essa parte é retirada da lista de partes da palavra. Em seguida, cada parte do primeiro nome é comparada com cada parte do segundo nome. Se uma parte de um nome possui uma parte igual em outro nome, essas duas partes são marcadas e não podem mais ser envolvidas na comparação das outras partes dos nomes comparados. Assim, por exemplo, as *strings* BaBeBa e BaBaBe são

consideradas iguais, uma vez que as *substrings* "Ba", "Be" e "Ba" são encontradas na segunda *string*. Entretanto, as *strings* BaBeBa e BaBeBe são diferentes uma vez que "Ba" e "Be" são encontradas na segunda *string*, mas "Ba" não é encontrada em uma segunda ocorrência.

- **Comparação de Elementos do Mesmo Tipo**: indica se deve ser considerado o tipo do elemento na comparação, i.e. classes só devem ser comparadas com classes e pacotes só devem ser comparados com pacotes.

Em relação ao exemplo apresentado na Figura 2, observa-se que a comparação é realizada por sinônimo, comparação de *substrings*, palavras ignoradas e tipo do elemento arquitetural. Por exemplo, "Aluno" pacote (i.e. elemento arquitetural) corresponde a "Estudante" pacote, ao passo que "Aluno_3grau" classe (i.e. parte da estrutura interna de um elemento arquitetural) corresponde a "EstudanteGraduação", também classe, nas diferentes arquiteturas. "Graduação" e "3grau" são sinônimos nesse caso, bem como "Aluno" e "Estudante".

Considerando a lista de palavras ignoradas e a comparação por *substrings*, observa-se que a *substring* "Utils" é ignorada nessa comparação e que, dessa forma, "Impressora" na 1ª arquitetura se torna equivalente à "ImpressoraUtils" na 2ª arquitetura. Além disso, considerando ainda a comparação por tipo do elemento, observa-se que "ImpressoraConfig" pacote corresponde a "ImpressoraConfig" pacote e que "ImpressoraConfig" classe corresponde a "ImpressoraConfig" também classe.

Todas essas opções podem ser determinadas como aplicáveis ou não pelo Engenheiro de Domínio, uma vez que a sua utilização em ArchToDSSA é flexível. Nessas comparações entre as arquiteturas, é sempre registrado o número de arquiteturas onde a equivalência é detectada. Quando elementos equivalentes são encontrados em todas as arquiteturas comparadas, eles são candidatos a elementos mandatórios no domínio, ao passo que se os elementos equivalentes são encontrados em algumas das arquiteturas comparadas apenas, eles são candidatos a elementos opcionais do domínio.

3.2. Detecção de Variabilidades

Na segunda fase de ArchToDSSA, as arquiteturas recuperadas e a lista de equivalências obtidas na fase 1 são analisadas para que se possa definir os VPs e as suas respectivas variantes no domínio, ou seja, as variabilidades do domínio em questão.

Conforme mencionado, a abordagem recebe como entrada arquiteturas recuperadas de sistemas legados OO, representadas através de modelos de pacotes e de classes da UML e descritas em formato XMI. Com base nessas informações, é possível descobrir os relacionamentos entre as classes. Os relacionamentos de herança e de implementação, especificamente, são de grande interesse para a descoberta de possíveis VPs e suas respectivas variantes. De acordo com a Odyssey-FEX (OLIVEIRA, 2006), notação adotada por ArchToDSSA para a representação de variabilidades e opcionalidades do domínio, um VP pode ser representado por uma superclasse ou interface no modelo de classes, e suas variantes podem ser representadas por subclasses ou classes que implementam uma interface no modelo de classes.

Além disso, é possível que o Engenheiro de Domínio determine em quantas arquiteturas uma dada interface ou superclasse deve possuir equivalência para que ela possa ser considerada um VP. Dessa forma, ArchToDSSA é capaz de indicar candidatos

a VPs e variantes, sendo que as variantes candidatas são todas as subclasses encontradas nas diferentes arquiteturas onde a superclasse que representa o VP possua equivalência ou todas as classes nas diferentes arquiteturas que implementem a interface que representa o VP.

Entretanto, essas regras de representação de VPs e variantes da Odyssey-FEX, embora tendo sido derivadas com base em estudos empíricos (OLIVEIRA, 2006), envolvendo a análise de modelos de classes de três domínios distintos, podem não ser válidas para todos os casos. Assim, é importante que o Engenheiro de Domínio possa determinar aleatoriamente VPs e variantes nas diferentes arquiteturas comparadas, bem como recusar sugestões de VPs e variantes detectados através do uso da abordagem.

3.3. Criação da DSSA

Finalmente, na última fase de ArchToDSSA, é selecionada uma das arquiteturas recuperadas como base para a criação da arquitetura de referência de domínio. A seleção de uma arquitetura é necessária para evitar a existência de estruturas inconsistentes na arquitetura de referência resultante, o que poderia ocorrer caso elementos de diferentes arquiteturas fossem combinados aleatoriamente para gerar a arquitetura de referência. Dessa forma, assegura-se que todos os elementos e relacionamentos dessa arquitetura "base" estarão incondicionalmente presentes na arquitetura de referência. Vale ressaltar que a arquitetura de referência, assim como as arquiteturas analisadas por ArchToDSSA, é representada em formato XMI. Elementos opcionais de outras arquiteturas, que não a arquitetura "base" selecionada, podem também compor a arquitetura de referência.

Diferentes arquiteturas de referência podem ser criadas para o mesmo conjunto de sistemas legados, com base em diferentes escolhas referentes à arquitetura base e elementos opcionais realizadas pelo Engenheiro de Domínio. A fim de facilitar essas escolhas, ArchToDSSA apresenta em cada arquitetura os elementos opcionais, mandatórios, VPs e variantes. Entretanto, essas escolhas não são apoiadas por ArchToDSSA, podendo ser levada em conta a experiência do Engenheiro do Domínio.

As informações de opcionalidades e variabilidades são combinadas, uma vez que representam propriedades ortogonais na Odyssey-FEX, sendo então estabelecidos VPs mandatórios e opcionais, bem como variantes mandatórias e opcionais, além de invariantes mandatórias e opcionais. Elementos opcionais selecionados de outras arquiteturas que não a arquitetura “base” são marcados como "não definidos" na arquitetura de referência, marcação essa permitida pela notação Odyssey-FEX, o que indica que eles ainda precisarão ter a sua especificação complementada na arquitetura de referência. Esses elementos são propositadamente incorporados de forma "solta" à arquitetura de referência, uma vez que não há como garantir que seus relacionamentos na sua arquitetura original se mantenham na arquitetura de referência, pois muitos dos elementos com os quais eles se relacionam direta ou indiretamente, podem não possuir equivalência na arquitetura "base". Ainda que possuam equivalência, não há como assegurar que as suas estruturas internas sejam exatamente iguais nas diferentes arquiteturas para garantir que as ligações continuem fazendo sentido.

A exceção se encontra para o caso das variantes opcionais incorporadas de outras arquiteturas, que são ligadas na arquitetura de referência através de herança ou

implementação de interface. Mas, ainda assim, elas são marcadas como "não definidas", pois um exame minucioso da sua compatibilidade estrutural e semântica em relação ao VP ainda deve ser realizado pelo Engenheiro de Domínio. Convém ressaltar, ainda, que caso sejam selecionados elementos equivalentes opcionais de diferentes arquiteturas para a criação da arquitetura de referência, ArchToDSSA admite apenas uma ocorrência do elemento na arquitetura final. Se ele existir na arquitetura "base" escolhida, será então priorizada a sua ocorrência nessa arquitetura.

Os elementos selecionados de outras arquiteturas, que não a arquitetura "base", estarão hierarquicamente subordinados ao seu elemento superior mais direto com equivalência na arquitetura "base" (i.e. um subpacote ou um pacote). Caso essa equivalência não seja encontrada em nenhum nível da hierarquia para o elemento incorporado de outra arquitetura, ele é alocado diretamente à raiz da arquitetura de referência, ou seja, a um pacote "DSSAmodel" criado no XMI.

Conforme mencionado, as arquiteturas de referência resultantes são representadas através de XMI. Uma vez que opcionalidades e variabilidades, bem como o conceito de "não definido", não fazem parte do metamodelo da UML, essas propriedades são representadas através de um mecanismo de extensão da UML, i.e. valores chaveados (*tagged values*). Valores chaveados permitem definir novos tipos de propriedades, adicionando tuplas de <nome, valor> a qualquer elemento de modelo. Dessa forma, cada elemento na arquitetura de referência resultante conterá 3 tuplas de <nome, valor>, a saber:

- Nome = "opcionalidade"; valor = "mandatório" ou "opcional";
- Nome = "variabilidade"; valor = "VP", "variante" ou "invariante";
- Nome = "não definido"; valor = "verdadeiro" ou "falso".

Os elementos "não definidos" na arquitetura de referência gerada a caracterizam como parcialmente especificada, devendo ser analisada e complementada pelo Engenheiro de Domínio antes de ser instanciada para novas aplicações.

4. Exemplo de Utilização da Abordagem e seu Apoio Ferramental

A fim de ilustrar a utilização da abordagem ArchToDSSA, um exemplo hipotético envolvendo um conjunto de 3 arquiteturas no domínio de telefonia móvel é apresentado nesta seção. Em (STOERMER & O'BRIEN, 2001), os autores comentam que um número mínimo de 3 a 4 arquiteturas de sistemas no domínio deve ser analisado para a detecção de opcionalidades e variabilidades do domínio.

A abordagem ArchToDSSA é executada com o apoio da ferramenta ArchToDSSATool, a qual pode ser utilizada de forma isolada ou integrada ao ambiente de reutilização Odyssey. Em ambos os casos, ela lê e gera arquivos em formato XMI, descrevendo modelos arquiteturais em UML. Dessa forma, embora possa ser utilizada de forma integrada ao Odyssey, ArchToDSSATool apresenta uma certa independência de ambiente de desenvolvimento.

A Figura 3 apresenta a tela principal da ArchToDSSATool, contemplando a primeira fase da abordagem. As 3 arquiteturas do domínio de telefonia móvel foram carregadas na ferramenta. ArchToDSSATool permite selecionar a arquitetura desejada em cada coluna, como pode ser visto pela seleção da XMICelular1 na primeira coluna,

apresentando as arquiteturas lado a lado, de duas em duas. A ferramenta trabalha com o conceito de *working set* (conjunto de trabalho), podendo ser definido um *working set* para cada domínio, contendo as arquiteturas dos sistemas no domínio, o dicionário e as configurações estabelecidas para a criação da DSSA.

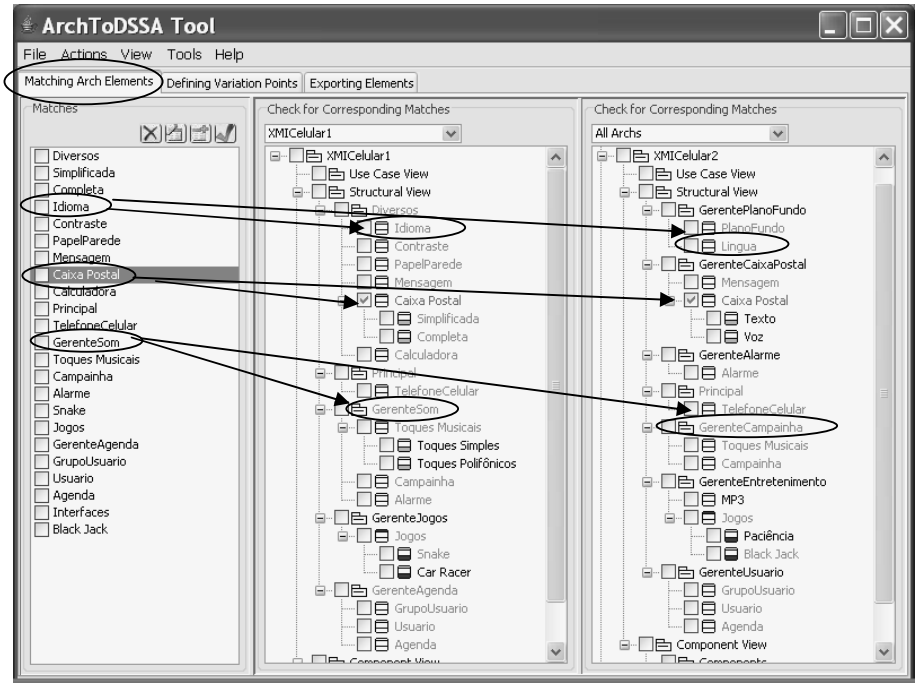


Figura 3. Tela principal da ferramenta ArchToDSSATool

A Figura 4 apresenta a tela de configuração de parâmetros para a realização da primeira fase da abordagem, ou seja, a detecção de equivalências (*matches*) e, conseqüentemente, de opcionalidades do domínio. Através desta tela, o usuário pode indicar: se deseja que o dicionário seja automaticamente alimentado quando valida ou cria uma equivalência (*match*) – opção: *Auto Feed Dictionary when Validating Match;* se o dicionário deve ser utilizado para sugerir equivalências – opção: *Use Dictionary when Suggesting Matches*; se a comparação por *substrings* deve ser adotada – opção: *Use Substring Comparison when Suggesting Matches*; e se a comparação deve levar em conta o tipo do elemento, ou seja, classes só devem ser comparadas com classes e pacotes com pacotes – opção: *Compare only Similar Elements when Suggesting Matches*. Além disso, o usuário pode ainda indicar em quantas arquiteturas a equivalência deve ser encontrada para que ela seja sugerida como um *match*.

Uma vez que todas as opções apresentadas na Figura 4 tenham sido selecionadas e o dicionário do domínio alimentado, a ferramenta ArchToDSSATool pode gerar, de forma automática, uma lista de equivalências entre as arquiteturas, como mostra a coluna da esquerda da tela principal da ferramenta na Figura 3. Neste caso, a equivalência ressaltada, i.e. Caixa Postal, possui correspondência em todas as arquiteturas, representando um elemento mandatório. A equivalência Idioma leva em conta ocorrências das palavras Idioma e Língua nas diferentes arquiteturas, uma vez que esta correspondência foi alimentada no dicionário de sinônimos. A equivalência

GerenteSom inclui ocorrências de GerenteSom e GerenteCampainha (Figura 3), uma vez que as palavras Som e Campainha também representam sinônimos no dicionário.

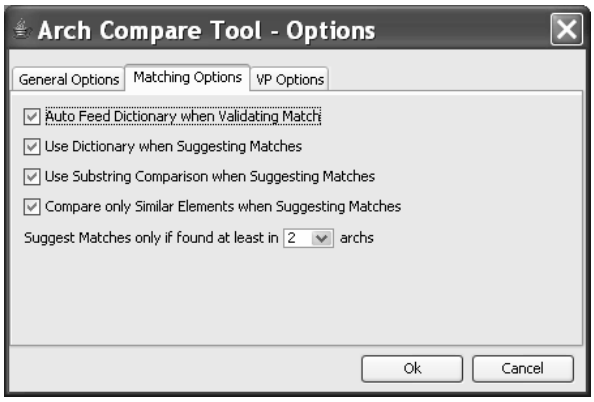


Figura 4. Tela de configuração de parâmetros da ArchToDSSATool

O Engenheiro de Domínio deve validar as equivalências que julgar corretas, tendo ainda a possibilidade de criar outras equivalências manualmente. Dessa forma, ele pode passar à fase 2 da abordagem, i.e. Detecção de Variabilidades, como mostra a Figura 5. Neste exemplo, o ponto de variação Jogos, que representa uma interface no modelo de classes, está selecionado, possuindo as variantes *Snake, Car Racer*, Paciência e *Black Jack*. Além desse VP, a ferramenta também detectou automaticamente o VP Caixa Postal, com as variantes Simplificada, Completa, Texto e Voz, e o VP Toques Musicais (Figura 5). Esses elementos representam hierarquias nas arquiteturas comparadas, ou seja, uma estrutura de herança que é uma candidata a VP.

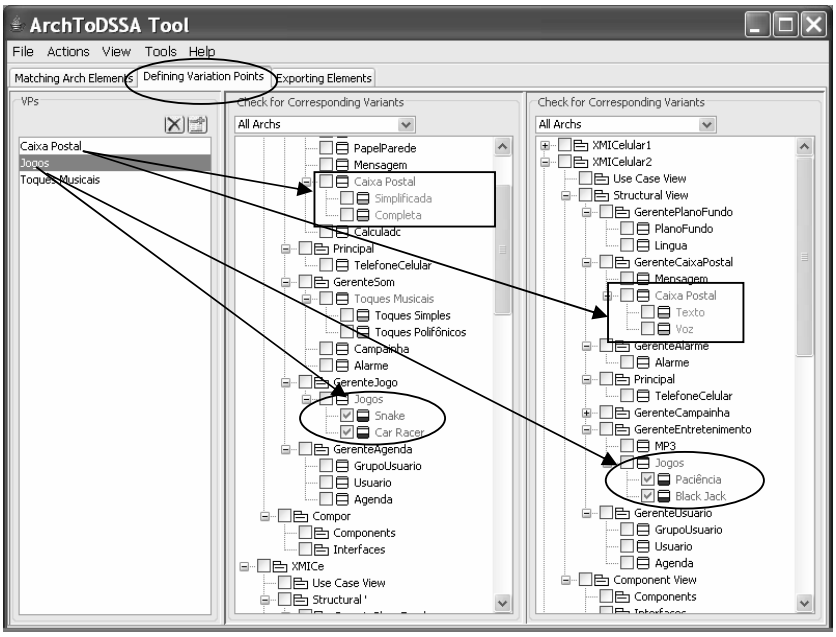


Figura 5. Detecção de variabilidades na ArchToDSSATool

A fase seguinte é a fase de criação e exportação da DSSA, para o ambiente Odyssey ou para o formato XMI, contendo as opcionalidades e variabilidades definidas. A Figura 6 apresenta a DSSA criada para o domínio de telefonia móvel no ambiente de

reutilização Odyssey. Os pacotes e classes ficam na *Structural View* do domínio no ambiente (lado esquerdo da Figura 6), podendo ser visualizados através de diagramas. Vale ressaltar que no ambiente Odyssey o modelo arquitetural gerado pode ser mapeado para o modelo de características do domínio, através de regras de mapeamento propostas em (OLIVEIRA, 2006). Dessa forma, é possível gerar um modelo de termos do domínio, que reflita a sua semântica e apóie a Análise do Domínio.

Como pode ser observado, Jogos é um VP (Figura 6), e como a arquitetura XMICelular1 foi a arquitetura-base selecionada para a criação da DSSA, as variantes Paciência e Black Jack ficam como não definidas (i.e. not defined), uma vez que constam da arquitetura XMICelular2 (Figura 5). Além disso, todas as variantes são opcionais, como pode ser visto pelo estereótipo Optional. Jogos, entretanto, é um VP mandatório. A classe BlueTooth também representa um elemento opcional do domínio. Vale ressaltar que no Odyssey as propriedades não definido, opcionalidade e variabilidade são visualmente apresentadas como estereótipos.

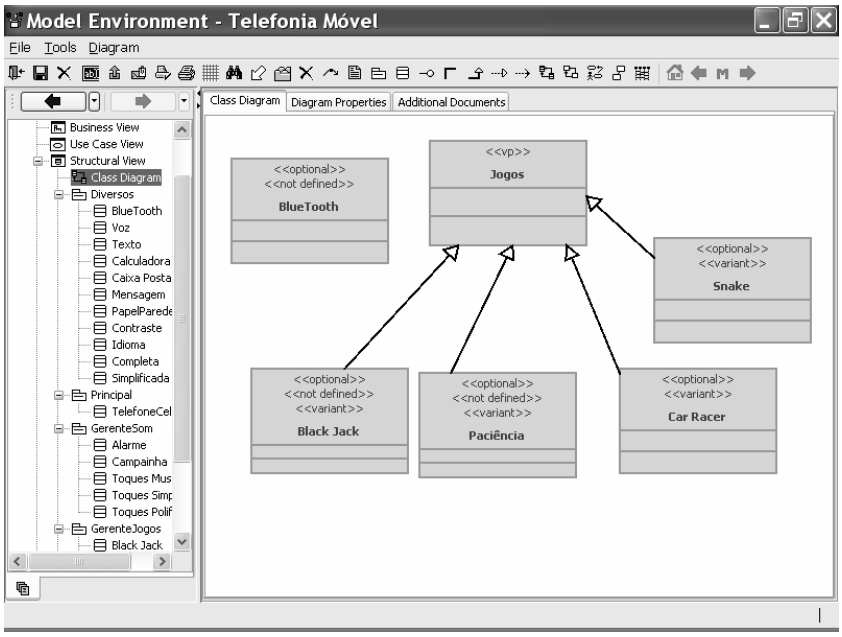


Figura 6. DSSA para o domínio de telefonia móvel no ambiente Odyssey

5. Conclusões e Trabalhos Futuros

ArchToDSSA representa uma abordagem de comparação de modelos arquiteturais de sistemas legados para apoiar a criação de arquiteturas de referência de domínio, que podem ser utilizadas no contexto da ED e LP. A abordagem apóia a geração de arquiteturas de referência de domínio parcialmente especificadas, permitindo a geração de diferentes DSSAs para o mesmo conjunto de arquiteturas legadas. Dessa forma, além de representar uma abordagem de comparação e extração de diferenças entre modelos, como outras abordagens existentes (CHEN *et al.*, 2003; MEHRA *et al.*, 2005; OLIVEIRA, 2005), ArchToDSSA contribui para a área de reutilização de software, identificando opcionalidades e variabilidades estruturais no domínio. Embora ArchToDSSA seja voltada à análise de sistemas legados, vale ressaltar que a abordagem pode ser utilizada também para a comparação de modelos arquiteturais

desenvolvidos através da engenharia progressiva, desde que esses atendam aos pré-requisitos mencionados para ArchToDSSA, ou seja, serem OO e possuírem as arquiteturas descritas em UML/XMI. ArchToDSSA é atualmente limitada à análise de sistemas com essas características.

As opcionalidades e variabilidades são identificadas tanto em nível de elemento arquitetural quanto em nível de suas classes, i.e. estrutura interna. Entretanto, nesse momento, a opcionalidade das classes não impacta a opcionalidade do elemento arquitetural, o que pode ser visto como uma necessidade para trabalho futuro. Além disso, seria interessante detectar diferenças em um nível de granularidade mais fino, ou seja, em nível de métodos e atributos, pois opcionalidades e variabilidades em um domínio podem se manifestar nesse nível "fino" de granularidade (MORISIO *et al.*, 2000). Seria interessante detectar também variabilidades e opcionalidades em outros modelos de domínio, como os modelos comportamentais, além de detectar variações também em termos dos requisitos não-funcionais do domínio.

Finalmente, ArchToDSSA e o seu apoio ferramental foram avaliados através de dois estudos de observação (KÜMMEL, 2007), um estudo envolvendo estudantes e o outro envolvendo profissionais da indústria, tendo sido analisados modelos arquiteturais hipotéticos. Esses estudos permitiram obter *feedback* e indícios sobre a eficiência de ArchToDSSA no apoio à criação de DSSAs.

Referências

ALVES, V., MATOS JR, P., COLE, L.*, et al.*, 2007, "Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming", *Transactions on Aspect-Oriented Software Development (TAOSD): Special Issue on Software Evolution, To Appear*.

ATKINSON, C., BAYER, J., BUNSE, C.*, et al.*, 2002, *Component-based Product Line Engineering with UML*, Boston, Addison-Wesley Longman Publishing Co., Inc.

BLOIS, A.P.B., 2006, *Uma Abordagem de Projeto Arquitetural Baseado em Componentes no Contexto de Engenharia de Domínio*, Tese de D.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

CHEN, P., CRITCHLOW, M., CARG, A.*, et al.*, 2003, "Differencing and Merging within an Evolving Product Line Architecture". In: *International Workshop on Software Product-Family Engineering*, pp. 269-281, Siena, Italy, November.

GANESAN, D., KNODEL, J., 2005, "Identifying Domain-Specific Reusable Components from Existing OO Systems to Support Product Line Migration". In: *First International Workshop on Reenegineering Towards Product Lines (R2PL),* pp. 16-20, Pittsburgh, PA, USA, November.

GOMAA, H., 2004, *Designing Software Product Lines with UML: from Use Cases to Pattern-Based Software Architectures*, Addison-Wesley Professional.

KANG, K.C., LEE, J., DONOHOE, P., 2002, "Feature-Oriented Product Line Engineering", *IEEE Software*, v. 9, n. 4 (Jul./Aug 2002), pp. 58-65.

KNODEL, J., MUTHIG, D., 2005, "Analyzing the Product Line Adequacy of Existing Components". In: *First Workshop on Reengineering towards Product Line*

(R2PL), pp. 21-25, Pittsburgh, PA, USA, November.

KÜMMEL, G., 2007, *Uma Abordagem para a Criação de Arquiteturas de Referência de Domínio a partir da Comparação de Modelos Arquiteturais de Aplicações*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

LEE, K., KANG, K.C., LEE, J., 2002, "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering". In: *Software Reuse: Methods, Techniques, and Tools : 7th International Conference, ICSR-7, Proceedings* pp. 62 - 77, Austin, TX, USA, April, 2002.

MEHRA, A., GRUNDY, J., HOSKING, J., 2005, "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design". In: *20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pp. 204-213, Long Beach, CA, USA, November.

MORISIO, M., TRAVASSOS, G.H., STARK, M.E., 2000, "Extending UML to Support Domain Analysis". In: *Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, pp. 321-324, Grenoble, France, September.

ODYSSEY, 2007, "Odyssey: Infra-Estrutura de Reutilização baseada em Modelos de Domínio". In: http://reuse.cos.ufrj.br/odyssey, accessed in 03/05/2007.

OLIVEIRA, H.L.R., 2005, *Odyssey-VCS: Uma Abordagem de Controle de Versões para Elementos da UML*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, RJ, Brasil.

OLIVEIRA, R., 2006, *Formalização e Verificação de Consistência na Representação de Variabilidades*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

OMG, 2005, *Software Process Engineering Metamodel (SPEM) Specification, version 1.1*, formal/05-01-06, Object Management Group.

PRIETO-DIAZ, R., ARANGO, G., 1991, "Domain Analysis Concepts and Research Directions", *PRIETO-DIAZ, R., ARANGO, G. (eds), Domain Analysis and Software Systems Modeling, IEEE Computer Society Press*, pp. 9-33.

STOERMER, C., O'BRIEN, L., 2001, "MAP: Mining Architectures for Product Line Evaluations". In: *3rd Working IFIP Conference on Software Architecture (WICSA)*, pp. 35-44, Amsterdam, Holland, August.

SUGUMARAN, V., PARK, S., KANG, K.C., 2006, "Software Product Line Engineering", *Communications of the ACM*, v. 49, n. 12.

VASCONCELOS, A., 2007, *Uma Abordagem de Apoio à Criação de Arquiteturas de Referência de Domínio baseada na Análise de Sistemas Legados*, Tese de D.Sc., PESC - COPPE, UFRJ, Rio de Janeiro, Brasil.

XAVIER, J.R., 2001, *Criação e Instanciação de Arquiteturas de Software Específicas de Domínio no Contexto de uma Infra-Estrutura de Reutilização*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

Suporte à Certificação de Componentes no Modelo de Representação X-ARM

Michael Schuenck[1], Glêdson Elias[1]

[1]Grupo COMPOSE (Component-Oriented Service Engineering)
Departamento de Informática - Universidade Federal da Paraíba (UFPB)
58.059-900 – João Pessoa – PB - Brasil
{michael, gledson}@compose.ufpb.br

Abstract. *Component repositories are commonly indicated as important resources for software reuse. In such repositories, in which independent producers and consumers can share software components, the evaluation of the quality of software components by third-party certifiers offers more security and confidence for consumers. In such a context, this paper introduces the resources supported by the XML-based Asset Representation Model (X-ARM) to describe component certifications. Such a model allows the description of certifications to be driven by distinct certification models, characterizing an important contribution since different certifiers can adopt distinct certification models.*

Resumo. *Repositórios de componentes são recursos comumente apontados como promotores do reuso de software. Nestes repositórios, onde produtores e consumidores independentes compartilham componentes de software, a avaliação da qualidade dos componentes por entidades certificadoras provê mais segurança e confiança aos consumidores. Neste sentido, este artigo introduz os mecanismos oferecidos pelo modelo de representação de componentes X-ARM (XML-based Asset Representation Model) para descrever certificações de componentes. Este modelo possibilita a descrição de certificações realizadas segundo diferentes modelos de certificação, caracterizando assim uma importante contribuição, pois diferentes entidades certificadoras podem adotar distintos processos de certificação.*

1. Introdução

Reuso de software é um importante mecanismo para incremento de produtividade e, consequentemente, redução de custos no desenvolvimento de software. Neste contexto, repositórios de componentes surgem com o potencial de facilitar ou mesmo habilitar o reuso de software ao possibilitar o compartilhamento, de forma livre ou não, de componentes de software entre diferentes produtores e consumidores [1]. Para que consumidores possam encontrar os componentes mais adequados às suas necessidades, repositórios realizam a indexação dos componentes, possibilitando que eles sejam posteriormente buscados por consumidores. Normalmente, a indexação se baseia em metadados dos componentes.

Por outro lado, conforme destacado por Bass [2], a inexistência de componentes de software com qualidade certificada constitui um inibidor do reuso de software, visto que consumidores temem que falhas desconhecidas sejam introduzidas em suas

aplicações. Diversas perspectivas, modelos e técnicas foram propostos e são utilizados para garantir a produção de software com qualidade. No entanto, no caso do reuso de componentes, existe a necessidade de um cuidado adicional com a qualidade, dado que este tipo de reuso normalmente se baseia na produção e no consumo de componentes por indivíduos que geralmente não se conhecem. Neste sentido, torna-se importante o papel de entidades certificadoras, sem relação com produtores de componentes, responsáveis por atividades de avaliação e certificação da qualidade dos componentes [3]. Um exemplo de entidade certificadora real é a *Underwriters Laboratories* [4].

Para a execução das atividades de certificação, um dos recursos mais conhecidos são modelos de certificação, que guiam as atividades de verificação dos componentes, provendo um mecanismo padronizado e repetível por meio da definição de um conjunto de características que devem ser avaliadas e pontuadas. O exemplo mais conhecido de modelo de certificação é o modelo ISO/IEC 9126-1 [5].

No entanto, apesar da existência de alguns modelos e técnicas de certificação de software [5][6][7][8][9], a maioria dos repositórios não provê mecanismos para garantir a qualidade de seus componentes e para permitir a busca com base em atributos de qualidade. Isto pode ser concluído com base no fato de que, dentre os modelos de representação de metadados de componentes encontrados na literatura [10][11][12], nenhum permite a representação de informações de certificação. Logo, a indexação e busca de componentes a partir de parâmetros qualitativos pode não ser possível. Por exemplo, um consumidor pode não ser capaz de encontrar componentes certificados de acordo com o modelo ISO/IEC 9126-1 ou componentes certificados e considerados de alta qualidade por mais de uma entidade certificadora.

Neste sentido, considerando a carência de iniciativas para descrição da certificação de componentes, este artigo introduz os recursos oferecidos pelo modelo X-ARM (*XML-based Asset Representation Model*) para representação de informações de certificação nos próprios metadados dos componentes. O modelo X-ARM é um modelo para representação de metadados de *assets* baseado no RAS [10]. No contexto do X-ARM, o termo *asset* designa um conjunto de artefatos criados ou reusados em um processo de desenvolvimento baseado em componentes. Cada conjunto de artefatos semanticamente relacionados é empacotado em um único arquivo, juntamente com um documento XML referente à descrição X-ARM, chamado manifesto.

O X-ARM foi projetado de forma a representar diversos tipos de informações importantes para o desenvolvimento de software baseado em componentes, sendo, portanto, um modelo bastante extenso. É importante ressaltar que os recursos providos pelo X-ARM para descrição de componentes já foram apresentados em [13]. No entanto, em [13], os recursos relacionados à certificação não foram discutidos, e, portanto, caracterizam o foco e a contribuição deste artigo.

Dentre os recursos do modelo X-ARM para a representação da certificação, se destacam como contribuição: a possibilidade de diferentes entidades certificadoras certificarem um mesmo componente; a viabilidade de uma entidade especificar e utilizar um modelo de certificação próprio; a habilidade de descrever certificações segundo quaisquer modelos de certificação; e a possibilidade de também descrever certificações dos próprios produtores dos componentes.

O restante deste artigo está dividido da seguinte forma: a Seção 2 apresenta as principais características de alguns modelos de certificação de software existentes; a

Seção 3 introduz as principais características do modelo X-ARM; a Seção 4 trata dos recursos deste modelo para representação de informações de certificação; a Seção 5 demonstra o uso do X-ARM na descrição de certificações; por fim, a Seção 6 apresenta algumas considerações finais.

2. Modelos de Certificação

A fim de possibilitar a especificação dos mecanismos oferecidos pelo modelo X-ARM para representação de informações de certificação, alguns modelos de certificação foram identificados, bem como as formas como são organizados. Assim, esta seção sumariza as conclusões desta revisão da literatura de forma a permitir, principalmente, a validação da abordagem de descrição de certificações oferecida pelo modelo X-ARM, conforme apresentado nas próximas seções.

Na literatura não existe uma grande variedade de modelos de certificação de software, sendo que o mais conhecido é o modelo ISO/IEC 9126-1 [5]. Este modelo se baseia na existência de características, sub-características e atributos. Os fatores de qualidade avaliados são classificados em seis características: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* e *portability*. Estas características são subdivididas em 27 sub-características. Por sua vez, as sub-características são decompostas em atributos, aos quais são associados os valores resultantes das certificações.

É importante destacar que o modelo ISO/IEC 9126-1 foi concebido para a certificação de software em geral. Dada sua aceitação, alguns esforços no sentido de adaptá-lo à certificação de componentes foram realizados. Exemplos destes esforços são os trabalhos de Bertoa e Vallecillo [6], de Álvaro et al. [7], e de Carvallo e Franch [8]. De forma simplificada, estes trabalhos alteram o modelo ISO/IEC 9126-1 tanto pela adição quanto pela remoção de características, sub-características e atributos, porém, mantendo a mesma organização hierárquica. Os dois primeiros trabalhos citados se concentram exclusivamente na definição de um modelo de certificação para componentes. Por outro lado, com uma motivação um pouco diferente, o trabalho de Carvallo e Franch estende o modelo ISO/IEC 9126-1 de forma a representar também fatores não técnicos de componentes, como informações de licença, custo e suporte, além de informações sobre o produtor do componente.

Como exemplo de uma abordagem não derivada do modelo ISO/IEC 9126-1, um *framework* para modelos de qualidade de componentes, denominado ABCDE, é apresentado em [9]. Este *framework* é definido por meio de cinco categorias de propriedades de avaliação: *acceptance*, *behavior*, *constraints*, *design* e *extension*. Por ser um *framework*, modelos de certificação podem ser criados a partir do ABCDE.

3. O Modelo X-ARM

O X-ARM é um modelo de metadados criado com o objetivo de propiciar o armazenamento, localização e recuperação não apenas de componentes, mas de *assets* em geral. Alguns exemplos de *assets* são casos de uso, diagramas UML, planos de teste, especificações de interfaces e de componentes, códigos-fonte e componentes.

Dentre as informações de um *asset* representadas pelo X-ARM, destacam-se o processo de desenvolvimento utilizado, certificados de qualidade, quem o certificou, o propósito do *asset*, qual modelo de componente é utilizado, quais interfaces são providas e requeridas, quais eventos são disparados e utilizados, quais componentes o compõe, e o relacionamento com outros *assets*. Apesar de ser bastante extenso, o

X-ARM define a representação de diversas informações como opcional, tornando-o facilmente adaptável a diferentes cenários e processos de desenvolvimento.

O X-ARM foi originalmente definido para o contexto de um repositório distribuído e compartilhado de componentes [14]. No entanto, ele também pode ser empregado por repositórios construídos com diferentes abordagens, já que a descrição de *assets* independe da arquitetura do repositório. Ao pressupor a existência de um repositório de componentes, a adoção do X-ARM apresenta alguns aspectos positivos, como, por exemplo, a validação da existência de *assets* semanticamente relacionados a um determinado *asset* e a possibilidade de oferecer um *front-end* para acompanhamento do processo de desenvolvimento de software baseado em componentes.

Por ser baseado no RAS, o X-ARM também utiliza o conceito de *profile*, cuja finalidade é separar conceitos, dividir complexidade e aumentar o reuso dos *assets*. O X-ARM define quatro *Profiles*: *Artifact Profile*, *Model Profile*, *Component Profile* e *Interaction Profile*. A Figura 1 ilustra a hierarquia de *profiles* do X-ARM.

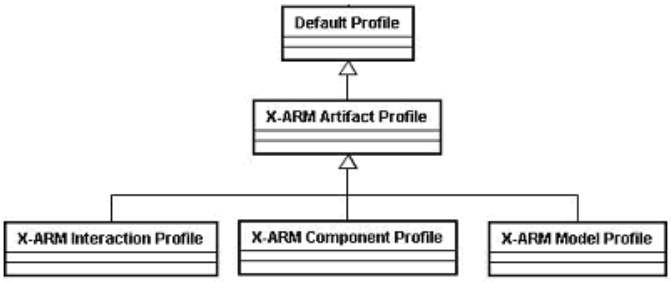


Figura 1: Hierarquia de Profiles do X-ARM

O *Artifact Profile* herda todos os elementos XML do *Default Profile*, definido pelo RAS, e acrescenta novos elementos, sendo o responsável por representar as funcionalidades básicas a todos os tipos de *assets* definidos pelo X-ARM.

O *Model Profile, Interaction Profile* e *Component Profile* estendem o *Artifact Profile*. O *Interaction Profile* descreve eventos, exceções e interfaces, que são os mecanismos usados por componentes para se comunicar com outros componentes ou aplicações. Com o *Component Profile* é possível descrever especificações e implementações de componentes, adicionando os elementos apropriados para descrever tais tipos de *assets*. Por fim, a intenção do *Model Profile* é padronizar os valores usados para representar características como certificação, classificação e negociação, habilitando buscas e comparações padronizadas entre diferentes *assets*.

4. Certificação de Componentes e Produtores no X-ARM

No modelo X-ARM, a certificação de um determinado *asset* é representada na forma de uma *descrição de certificação*, também denominada no modelo como certificado de qualidade. É importante ressaltar que um mesmo *asset* pode possuir diversas descrições de certificação, sendo cada uma delas produzida por diferentes entidades certificadoras.

Na hierarquia de *profiles* do X-ARM, as descrições de certificação de um determinado *asset* são representadas usando elementos do *X-ARM Artifact Profile*. É importante destacar que as descrições de certificação são representadas usando o *profile* mais básico, no caso o *X-ARM Artifact Profile*, ao invés de serem representadas com o *X-ARM Component Profile*. Como benefício desta abordagem, o X-ARM permite a

descrição da certificação de qualidade de qualquer tipo de *asset*, e não apenas de especificações e implementações de componentes de software, que são os tipos de *assets* que podem ser representados com o *X-ARM Component Profile*.

Para uniformizar ou padronizar as informações contidas nas descrições de certificação produzidas pelas diferentes entidades certificadoras, o X-ARM requer que tais entidades adotem *modelos de certificação*. No X-ARM, um modelo de certificação é representado usando elementos do *X-ARM Model Profile*. Assim, um modelo de certificação descrito com este *profile* também é um *asset*. A representação de modelos de certificação é de fundamental importância, pois permite a validação das informações contidas nas descrições de certificação dos *assets*, representadas com o *X-ARM Artifact Profile*. Desta forma, o X-ARM obriga que todas as descrições de certificação estejam em conformidade com seus respectivos modelos de certificação, que, por sua vez, são adotados pelas entidades certificadoras dos *assets*.

É importante ressaltar que, além de possuir elementos para representar a certificação de *assets*, o X-ARM também define elementos que permitem a representação da certificação dos produtores dos *assets*. Assim, de forma análoga ao adotado para *assets*, produtores possuem descrições de certificação que são produzidas por entidades certificadoras adotando modelos de certificação. Em resumo, no X-ARM, *assets* e produtores podem possuir diversas descrições de certificação que são geradas em conformidade com seus respectivos modelos de certificação.

Para não limitar os modelos de certificação que possam vir a ser adotados, e, assim, seja possível utilizar quaisquer processos de certificação, os modelos e as descrições de certificação de *assets* e produtores são representados na forma de pares chave/valor, sendo que as possíveis chaves e restrições para os valores são especificadas por *assets* de modelo de certificação.

As próximas seções apresentam como os modelos de certificação de *assets* e produtores podem ser representados usando o *X-ARM Model Profile*, e, em seguida, como as descrições de certificação de *assets* e produtores podem ser representadas usando o *X-ARM Artifact Profile*.

4.1. Representação de Modelos de Certificação

Em uma análise da estrutura dos principais modelos de certificação existentes, é fácil perceber que a maioria destes modelos adota estruturas hierárquicas para organizar as características e atributos avaliados durante o processo de certificação. Desta forma, no X-ARM, os modelos de certificação também são organizados em uma estrutura hierárquica de pares chave/valor. A Figura 2 ilustra os elementos do *X-ARM Model Profile* adotados para representar os modelos de certificação. Na Figura 2, a notação de classes da UML é usada para ilustrar os elementos e os atributos XML definidos pelo X-ARM para representar modelos de certificação.

O elemento <*asset*> é o elemento raiz das descrições X-ARM. Ele possui vários atributos e elementos filhos. Porém, dado o escopo deste trabalho, apenas os atributos e elementos relacionados à certificação de *assets* e suas funcionalidades são apresentados. Mais informações sobre os outros atributos e elementos podem ser encontradas em [13].

O atributo *id* do elemento <*asset*> é usado para identificar de forma única o modelo de certificação. Este atributo é importante para que outros *assets* possam indicar

o modelo de certificação que foi adotado em suas certificações. O nome do modelo de certificação é representado pelo atributo *name* do elemento <*asset*>.

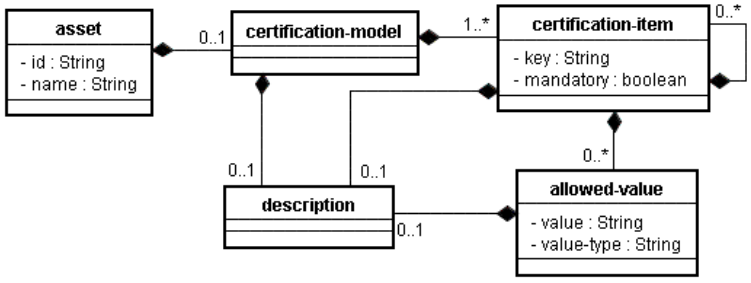


Figura 2: Especificação de modelos de certificação

Um modelo de certificação pode possuir um ou mais itens de certificação, que são os parâmetros verificados durante a avaliação da qualidade dos *assets*. Assim, o elemento <*certification-model*> possui um ou vários elementos <*certification-item*>. Em cada item, o atributo *key* identifica o nome de um parâmetro de certificação. Os itens de certificação que compõem um modelo podem ser obrigatórios ou opcionais. Para representar tal possibilidade, o atributo *mandatory* indica se o item especificado é obrigatório ou não em uma descrição da certificação de um *asset*. Além disso, um item pode se especializar em vários outros itens, criando assim uma estrutura hierárquica para o modelo de certificação.

Para cada item, opcionalmente, podem ser definidos os valores permitidos usando o elemento <*allowed-value*>, que possui os atributos *value* e *value-type*. O primeiro atributo indica um valor possível para o item de certificação e o segundo atributo indica o tipo deste valor, sendo possíveis apenas os tipos *string*, *integer, unsigned integer, float* e *unsigned float*. O atributo *value* pode ter um único valor discreto, múltiplos valores discretos, intervalos de valores numéricos ou uma combinação destas formas. Múltiplos valores discretos são representados pelos valores discretos separados por vírgula, por exemplo: "*yes, no*". Para representar um intervalo de valores numéricos, deve-se utilizar a seguinte notação: "*min_value..max_value*". Por exemplo, o valor "1..10" representa um intervalo de 1 a 10. Para especificar um intervalo de valor ilimitado utiliza-se o termo "*". Por exemplo, o valor "1..*" representa um intervalo de 1 até o limite superior permitido pelo respectivo tipo. No caso do tipo *string*, o termo "*" também indica que o item pode possuir qualquer valor. Por fim, é possível combinar valores discretos, múltiplos valores discretos e intervalos de valores, como por exemplo: "1..10, 25, 30..40, 50, 60".

Para facilitar o entendimento, todos os elementos de um modelo de certificação podem possuir uma descrição textual associada usando o elemento <*description*>.

É importante destacar que os modelos de certificação de *assets* e produtores são representados usando a mesma estrutura e elementos do *X-ARM Model Profile*. A Seção 5 apresenta exemplos de modelos de certificação de *assets* representados no X-ARM.

4.2. Representação de Descrições de Certificação

Depois que um modelo de certificação está descrito com a sintaxe definida pelo *X-ARM Model Profile*, é possível gerar descrições de certificação que adotam o modelo. Neste sentido, o *X-ARM Artifact Profile* define o elemento <*certification-info*>, cuja estrutura

é ilustrada na Figura 3. Este elemento é opcional, devendo ocorrer apenas em *assets* que possuem algum tipo de certificação.

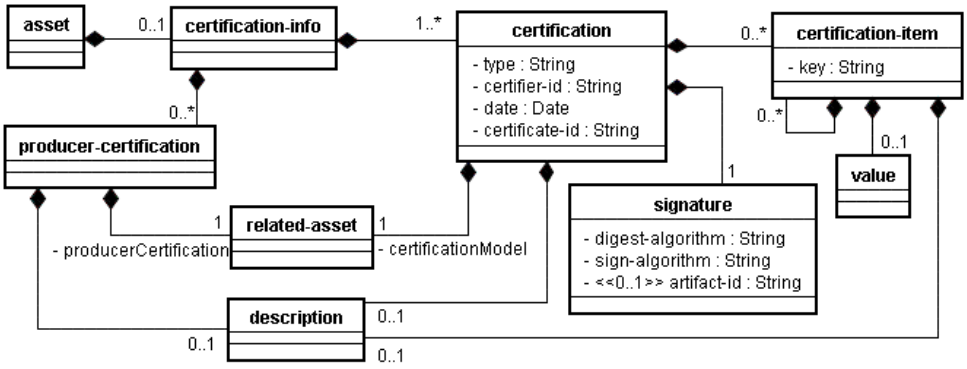


Figura 3: Especificação de descrições de certificação

Como pode ser observado na Figura 3, o elemento <*certification-info*> possui dois elementos: <*certification*> e <*producer-certification*>. O elemento <*certification*> é usado para representar informações da certificação do próprio *asset*. Por sua vez, o elemento <*producer-certification*> é usado para representar informações da certificação do produtor do *asset*.

Na descrição de certificação de um *asset*, como pode ser visto na Figura 3, o elemento <*certification*> inclui o tipo (*type*), a data da emissão da descrição de certificação (*date*), o identificador da entidade certificadora (*certifier-id*) e o identificador do próprio certificado (*certificate-id*). O tipo da certificação se refere ao nome do modelo de certificação adotado. O identificador da entidade certificadora corresponde ao esquema de nomeação usado pelo repositório de componentes para indicar a entidade que realizou a certificação do *asset*. O identificador do certificado é usado para identificar diferentes descrições de certificação de um determinado *asset*.

Cada elemento <*certification*> pode possuir elementos <*certification-item*>, responsáveis por descrever detalhes da certificação através de pares de chave/valor, representados pelo atributo *key* e pelo elemento <*value*>, respectivamente. É importante destacar que a estrutura de pares chave/valor deve seguir a hierarquia e as restrições impostas pelo respectivo modelo de certificação, que é identificado por meio de um elemento <*related-asset*>. Desta forma, um repositório de componentes pode validar as descrições de certificação, assegurando que as mesmas estão em conformidade com os respectivos modelos de certificação.

No sentido de assegurar que o certificado foi de fato emitido pela entidade certificadora especificada, o X-ARM define o elemento <*signature*>, que contém uma assinatura digital do certificado e as informações sobre os algoritmos utilizados na geração desta assinatura. Esta assinatura é gerada pela entidade certificadora considerando todos os arquivos incluídos no pacote do *asset* e o manifesto, ou seja, o documento XML com a descrição X-ARM do *asset*.

Na descrição de certificação de um produtor, como pode ser observado na Figura 3, o elemento <*producer-certification*> é usado para representar a certificação do processo de software adotado pelo produtor do *asset*. Esta representação é realizada apenas com uma referência a outro *asset*, criado especificamente para descrever a certificação do produtor. Esta referência é indicada usando o elemento <*related-asset*>.

Neste *asset*, a certificação do produtor é descrita da mesma forma que qualquer outro *asset*, usando o elemento <*certification*>. O benefício de manter estas descrições em *assets* separados é possibilitar o reuso das informações de certificação de um produtor nos diferentes *assets* que ele tenha criado.

De forma semelhante à representação de modelos de certificação, para auxiliar o entendimento, também podem ser associadas descrições textuais aos elementos <*certification*>, <*certification-item*> e <*producer-certification*> usando o elemento <*description*>.

4.3. Relacionamento entre Modelos e Descrições de Certificação

Uma vez que os elementos definidos pelo X-ARM para descrição de modelos de certificação e para descrição de certificações de *assets* e produtores foram apresentados de forma isolada, esta seção se dedica a explicar com mais detalhes o inter-relacionamento entre *assets* com diferentes papéis na representação de certificações. Assim, a Figura 4 ilustra um modelo conceitual onde este inter-relacionamento é identificado.

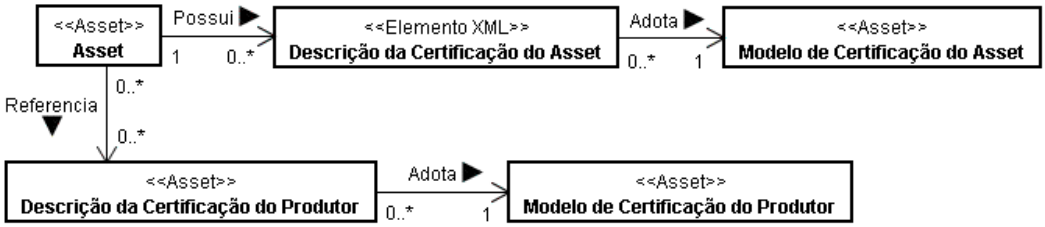


Figura 4: Modelo conceitual da representação de certificações

Com base na Figura 4, é possível perceber algumas características da representação de certificações. Primeiramente, um *asset* pode possuir diversas descrições de certificação, sendo cada uma emitida por diferentes entidades certificadoras e representada por um elemento <*certification*> em particular. Cada descrição de certificação se baseia apenas em um único modelo de certificação. Obviamente, para que as descrições de certificação de um *asset* sejam consistentes, elas não podem ter sido realizadas pela mesma entidade certificadora, exceto se estas certificações adotarem modelos de certificação distintos, cujas características e aspectos avaliados são diferentes. Por outro lado, um mesmo modelo de certificação pode ser adotado em diversas descrições de certificação de diferentes *assets*.

No caso de certificações de produtores, cada *asset* indica as diversas descrições de certificação do seu respectivo produtor. Assim como nas descrições de certificação dos *assets*, cada descrição de certificação do produtor deve ser gerada em conformidade com apenas um único modelo de certificação. De forma inversa, um modelo de certificação de produtor pode validar várias descrições de certificação de produtores.

4.4. Certificação em Repositórios X-ARM

Depois de apresentados os elementos para modelos e descrição de certificação, bem como o inter-relacionamento entre os *assets* envolvidos nas certificações, esta seção propõe um procedimento para certificação de *assets* e produtores. É importante mencionar que o procedimento é apresentado sob o ponto de vista de um repositório de componentes que adote o X-ARM como modelo de descrição dos seus *assets*.

Considerando que as certificações de *assets* e produtores são realizadas por entidades certificadoras, logicamente, as descrições de certificação são geradas pelas próprias entidades certificadoras. Por outro lado, o X-ARM não estabelece um ator específico com a responsabilidade de descrever os modelos de certificação. No entanto, no caso de entidades certificadoras que adotam processos de certificação proprietários, o mais natural é que as próprias entidades certificadoras descrevam os modelos de certificação adotados na certificação dos *assets*. Porém, é possível ainda que diferentes entidades certificadoras utilizem os mesmos modelos de certificação. Este é o caso de modelos de certificação públicos, definidos por instituições de padronização como a ISO. Assim, no caso de modelos de certificação públicos, a expectativa é que a própria instituição de padronização descreva o modelo de certificação. Após a instituição de padronização descrever um modelo de certificação público, qualquer entidade de certificação pode adotá-lo sem a necessidade de descrevê-lo novamente, bastando referenciá-lo com um identificador provido pelo repositório de componentes.

De forma resumida, o procedimento de certificação em um repositório que adote o X-ARM é constituído pela seqüência de atividades apresentada na Figura 5. Nesta figura, que representa um diagrama de atividades UML, considera-se que o *asset* a ser certificado é inicialmente descrito e registrado no repositório pelo seu produtor.

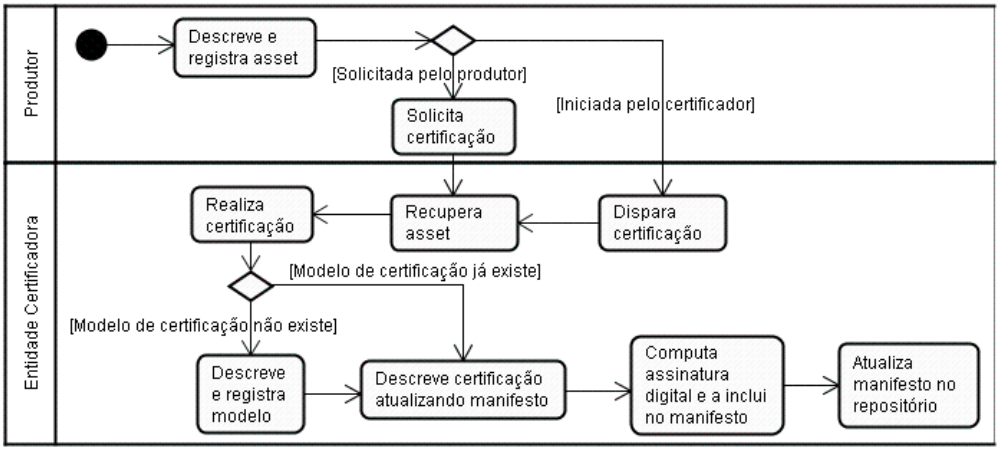


Figura 5: Diagrama de atividades envolvidas na certificação

Assim, a primeira atividade do diagrama ilustrado na Figura 5 representa a geração do manifesto de um *asset* e o subseqüente registro do *asset* no repositório. A partir de então, o processo de certificação do *asset* pode ser iniciado pela entidade certificadora. Em geral, a certificação pode ser disparada a pedido do próprio produtor ou por iniciativa unilateral da entidade certificadora. Na Figura 5, tais alternativas são representadas pelas atividades *solicita certificação* e *dispara certificação*.

Vale ressaltar que, na prática, existe um debate sobre a imparcialidade do processo de certificação adotar a abordagem requisitada pelo produtor ou disparada pela entidade certificadora. No entanto, o X-ARM não se limita a uma abordagem específica, deixando sob a responsabilidade de cada repositório selecionar a abordagem mais adequada às necessidades de seus respectivos produtores.

Uma vez que a entidade certificadora identifica a necessidade de certificar um dado *asset*, esta entidade recupera o *asset* do repositório, e, guiando-se pelo modelo de certificação adotado, a entidade certificadora realiza a certificação do *asset*.

Em seguida, a entidade certificadora deve gerar a descrição da certificação com o *X-ARM Artifact Profile*. Para gerar a descrição da certificação, existe a pré-condição do *asset* que representa o modelo de certificação adotado já ter sido previamente cadastrado no repositório. Assim, se o modelo de certificação ainda não existe, ele deve ser primeiramente descrito usando o *X-ARM Model Profile*, conforme apresentado na Seção 4.1.

Depois que a descrição da certificação do *asset* for concluída, a assinatura digital da certificação é computada, utilizando-se as definições apresentadas na Seção 4.2 para o elemento <*signature*>, e adicionada ao manifesto. Somente após a atualização do manifesto no repositório é que o *asset* pode ser considerado certificado.

5. Exemplos de Uso de Modelos e Descrições de Certificação

Nesta seção, exemplos de modelos e descrições de certificação de *assets* são apresentados. O objetivo é evidenciar o potencial do modelo X-ARM para representar diferentes modelos de certificação e instanciar tais modelos para gerar as respectivas descrições de certificação dos *assets*. Vale ressaltar que, por limitação de espaço, exemplos de modelos e descrições de certificação de produtores não serão apresentados. No entanto, tais modelos e descrições de certificação de produtores podem ser especificados de forma bastante semelhante.

A partir dos elementos definidos pelo *X-ARM Model Profile*, apresentados na Seção 4.1, a Figura 6 apresenta um exemplo de representação parcial do modelo de certificação ISO/IEC 9126-1 [5].

```
1. <asset id=" org.iso.isoiec91261-1.0" name="ISO/IEC 9126-1" ... >
…
2.     <certification-model>
3.         <certification-item key="Functionality" mandatory="false">
4.             <certification-item key="Accuracy" mandatory="false">
5.                 <certification-item key="Correctness" mandatory="false">
6.                     <allowed-value value="0..100" value-type="integer"/>
7.                 </certification-item>
8.             </certification-item>
9.             <certification-item key="Security" mandatory="false">
10.                <certification-item key="Data Encryption" mandatory="false">
11.                    <allowed-value value="yes, no" value-type="string"/>
12.                </certification-item>
13.                <certification-item key="Controllability" mandatory="false">
14.                    <allowed-value value="0..100" value-type="integer"/>
15.                </certification-item>
16.                <certification-item key="Auditability" mandatory="false">
17.                    <allowed-value value="yes" value-type="string"/>
18.                    <allowed-value value="no" value-type="string"/>
19.                </certification-item>
20.            </certification-item>
21.        </certification-item>
22.        <certification-item key="Efficiency" mandatory="false">
23.            <certification-item key="Resource Behavior" mandatory="false">
24.                <certification-item key="Memory Utilization" mandatory="false">
25.                    <allowed-value value="*" value-type="integer"/>
26.                </certification-item>
27.                <certification-item key="Disk Utilization" mandatory="false">
28.                    <allowed-value value="*" value-type=" integer "/>
29.                </certification-item>
30.            </certification-item>
31.        </certification-item>
…
32.    </certification-model>
33. </asset>
```

Figura 6: Modelo de Certificação ISO/IEC 9126-1

Na Figura 6 são apresentadas partes das características *Functionality* (linhas 3 a 21) e *Efficiency* (linhas 22 a 31). No primeiro caso, estão representadas as sub-características *Accuracy* (linhas 4 a 8) e *Security* (linhas 9 a 20). A primeira é subdividida em *Correctness* (linhas 5 a 7), que tem como valores permitidos números inteiros entre *0* e *100*. Por sua vez, a sub-característica *Security* tem os atributos *Data Encryption*, *Controllability* e *Auditability*. *Data Encryption* (linhas 10 a 12) e *Auditability* (linhas 16 a 19) definem os valores "*yes*" e "*no*" como permitidos. Já o item *Controllability* (linhas 13 a 15) tem números inteiros entre *0* e *100* como permitidos. Por fim, a característica *Efficiency* (linhas 22 a 31) apresenta o item *Resource Behavior* (linhas 23 a 30), subdividida nos atributos *Memory Utilization* (linhas 24 a 26) e *Disk Utilization* (linhas 27 a 29), que podem possuir valores inteiros.

Em três pontos do exemplo foram colocadas reticências para simplificar o exemplo. Na primeira e na última linha, as reticências indicam a existência de outros atributos e outros elementos que descrevem o elemento <*asset*>, respectivamente. Em ambos os casos, as informações omitidas estão fora do contexto deste artigo. Na terceira ocorrência de reticências, na linha 31, foram omitidos os demais itens de certificação definidos pelo modelo ISO/IEC 9126-1. Reticências também são utilizadas nos exemplos subseqüentes com finalidades similares.

Com base no exemplo da Figura 6, a Figura 7 apresenta um exemplo de descrição de certificação de um *asset* usando o modelo ISO/IEC 9126-1.

```
…  1. <certification-info>
2.          <certification type="ISO/IEC 9126-1" certifier-id="br.compose"
                        certificate-id=”br.compose.certificateA” date="2007-05-10">
3.              <related-asset id="org.iso.isoiec91261-1.0" name=”ISO/IEC 9126-1”
                        relationship-type="certificationModel"  required="true"/>
4.              <certification-item key="Functionality">
5.                  <certification-item key="Accuracy">
6.                      <certification-item key="Correctness">
7.                          <value>87</value>
8.                      </certification-item>
9.                  </certification-item>
10.                 <certification-item key="Security">
11.                     <certification-item key="Data Encryption">
12.                         <value>yes</value>
13.                     </certification-item>
14.                     <certification-item key="Controllability">
15.                         <value>98</value>
16.                     </certification-item>
17.                     <certification-item key="Auditability">
18.                         <value>yes</value>
19.                     </certification-item>
20.                 </certification-item>
21.             </certification-item>
22.             <certification-item key="Efficiency">
23.                 <certification-item key="Resource Behavior">
24.                     <certification-item key="Memory Utilization">
25.                         <value>3000000</value>
26.                     </certification-item>
27.                     <certification-item key="Disk Utilization">
28.                         <value>1200000</value>
29.                     </certification-item>
30.                 </certification-item>
31.             </certification-item> …
32.             <signature digest-algorithm="SHA1" sign-algorithm="RSA">
                    2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
33.             </signature>
34.         </certification>
35. </certification-info> ...
```

Figura 7: Descrição de certificação baseada no ISO/IEC 9126-1

No exemplo da Figura 7, observa-se na linha 2 que o tipo de certificação adotado foi o *ISO/IEC 9126-1* e que a certificação foi realizada pela entidade certificadora cuja identificação é *br.compose*. Neste exemplo, como pode ser visto na linha 3, o modelo de certificação adotado está especificado no *asset* identificado por *org.iso.isoiec91261-1.0*. Seguindo as restrições deste modelo, foram incluídas algumas ocorrências do elemento <*certification-item*>, que descrevem a certificação do *asset*. Por exemplo, as sub-características *Correctness*, *Data Encryption*, *Controllability*, *Auditability*, *Memory Utilization* e *Disk Utilization* possuem os valores *87*, "*yes*", *98*, "*yes*", *3000000* e *1200000*, respectivamente. Por fim, o elemento <*signature*> (linhas 32 a 33) representa a assinatura digital emitida pela entidade certificadora, além dos algoritmos utilizados para gerar a assinatura, neste caso o SHA1 e o RSA. Neste exemplo, apenas o *asset* foi certificado, mas não o seu produtor, pois não existe nenhuma ocorrência do elemento <*producer-certification*>.

Para ilustrar a capacidade do modelo X-ARM descrever certificações realizadas a partir de diferentes modelos de certificação, a Figura 8 apresenta a especificação parcial de um outro modelo de certificação, sendo neste caso baseado no *framework* ABCDE [9]. Neste exemplo, nota-se que foram representadas as categorias *Acceptance* (linhas 3 a 7), *Constraints* (linhas 8 a 15) e *Extension* (linhas 16 a 20) e alguns de seus itens de avaliação. Os itens *Published evaluation* (linhas 4 a 6) e *Platform spec* (linhas 9 a 11) podem possuir qualquer valor textual. Já item *Ease of use* (linhas 12 a 14) pode possuir valores inteiros entre *0* e *100*, enquanto que o item *Portable across platforms* (linhas 17 a 19) pode possuir apenas os valores "*true*" e "*false*". Como pode ser observado no atributo *mandatory*, todos estes itens estão especificados como não obrigatórios.

```
1. <asset id="br.compose.abcde-1.0" name="ABCDE" ... > …
2.     <certification-model>
3.         <certification-item key="Acceptance" mandatory="false">
4.             <certification-item key="Published evaluation" mandatory="false">
5.                 <allowed-value value="*" value-type="string"/>
6.             </certification-item>
7.         </certification-item>
8.         <certification-item key="Constraints" mandatory="false">
9.             <certification-item key="Platform spec" mandatory="false">
10.                <allowed-value value="*" value-type="string"/>
11.            </certification-item>
12.            <certification-item key="Ease of use" mandatory="false">
13.                <allowed-value value="0..100" value-type="integer"/>
14.            </certification-item>
15.        </certification-item>
16.        <certification-item key="Extension" mandatory="false">
17.            <certification-item key="Portable across platforms" mandatory="false">
18.                <allowed-value value="true, false" value-type="string"/>
19.            </certification-item>
20.        </certification-item> …
21.    </certification-model>
22. </asset>
```

Figura 8: Modelo de certificação baseado no framework ABCDE

Por fim, a Figura 9 ilustra um exemplo de descrição de certificação baseada no modelo ABCDE, apresentado na Figura 8. No exemplo, observa-se na linha 2 que o tipo de certificação adotado foi o *ABCDE* e que a certificação foi realizada pela entidade certificadora cuja identificação é *br.compose*. Como pode ser visto na linha 3, o modelo de certificação adotado está especificado no *asset* indicado por *br.compose.abcde-1.0*. Dado que este modelo define os itens *Acceptance* e *Published evaluation* como não obrigatórios, estes itens não foram descritos. Todos os demais itens foram representados

seguindo as restrições deste modelo. Por exemplo, as sub-características *Platform spec*, *Ease of use* e *Portable across platforms* possuem os valores "*OS: Windows 98+, Processor: x86, 32 bits*", *80* e "*false*". Por fim, o elemento <*signature*> (linhas 17 a 18) representa a assinatura digital emitida pela entidade certificadora.

```
… 1. <certification-info>
2.    <certification type="ABCDE" certifier-id="br.compose"
                    certificate-id=”br.compose.certificateB” date="2007-05-12">
3.       <related-asset id=" br.compose.abcde-1.0" name=”ABCDE”
                    relationship-type="certificationModel"  required="true"/>
4.      <certification-item key="Constraints">
5.         <certification-item key="Platform spec">
6.              <value>OS: Windows 98+, Processor: x86, 32bits</value>
7.         </certification-item>
8.         <certification-item key="Ease of use">
9.              <value>80<value>
10.         </certification-item>
11.      </certification-item>
12.       <certification-item key="Extension">
13.          <certification-item key="Portable across platforms">
14.             <value>false</value>
15.          </certification-item>
16.      </certification-item>
17.      <signature digest-algorithm="SHA1" sign-algorithm="RSA">
              3bc5a3b46d3e27acdf961bc2be35a1842b94bd312
18.      </signature>
19.   </certification>
20. </certification-info> ...
```

Figura 9: Descrição de certificação baseada no framework ABCDE

6. Considerações Finais

O modelo X-ARM foi criado com o objetivo de descrever os principais tipos de informações referentes ao desenvolvimento de software baseado em componentes, incluindo, por exemplo: processo de desenvolvimento, classificação, formas de negociação, informações de qualidade e os próprios componentes, por meio de suas interfaces providas e requeridas, seus eventos disparados e recebidos. Trata-se, portanto, de um modelo rico e extenso. Logo, dada a limitação de espaço, este artigo se limitou a apresentar seus mecanismos para descrição de certificações de componentes.

Um importante aspecto positivo do X-ARM é não limitar-se à descrição de certificações de componentes de acordo com um único modelo de certificação específico. Para tal, o X-ARM define que não apenas descrições de certificação sejam representadas, mas também, seus respectivos modelos de certificação, que habilitam a validação das certificações. Como vantagem adicional desta abordagem, tem-se a padronização das informações de certificação, já que, apesar da flexibilidade, todas as informações seguem uma estrutura e sintaxe pré-definida. Por outro lado, um aspecto negativo do X-ARM é a necessidade do esforço adicional requerido para também representar os modelos de certificação. No entanto, este aspecto não chega a ter um impacto negativo tão forte, visto que, tendo sido representado, um dado modelo de certificação pode ser reusado e referenciado por inúmeras descrições de certificação.

As facilidades de representação de informações de certificação do X-ARM constituem uma importante contribuição para o contexto de repositórios de componentes. Esta contribuição torna-se ainda mais relevante quando considerado que os outros modelos de representação de componentes existentes, tais como OSD [11], CDML [12] e o próprio RAS [10], não tratam da questão de certificação. Portanto, ao contrário do X-ARM, estes outros modelos não permitem aos consumidores realizar

buscas ou selecionar *assets* com base em informações de certificação, reduzindo a segurança e confiança dos mesmos, e, assim, dificultando o reuso em larga escala.

Embora as contribuições do X-ARM sejam importantes, ainda existem algumas melhorias que podem ser incluídas. Por exemplo, na especificação de modelos de certificações, é possível enriquecer as regras para restrição dos valores permitidos adotando expressões regulares. Outros esforços futuros serão concentrados em ferramental para preenchimento e validação automática de certificações de componentes usando os respectivos modelos de certificação. Desta forma, para facilitar o preenchimento, tal ferramental pode adotar formulários dinâmicos, cujos campos mudam conforme o modelo de certificação adotado.

Referências

[1] Frakes, William; Kang, Kyo (2005). "Software Reuse Research: Status and Future". IEEE Transactions On Software Engineering, Vol.31, Nº 7, July.

[2] Bass, L. et al. (2000) "Market Assessment of Component-Based Software Engineering". SEI, Technical Report CMU/SEI-2001-TN-007.

[3] Wallnau, Kurt C (2004). "Software Component Certification: 10 Useful Distinctions". Technical Note, CMU/SEI-2004-TN-031.

[4] Underwriters Laboratories. http://www.ul.com. Acessado em 02/06/2007.

[5] ISO 9126 (2001). "Information Technology – Product Quality – Part1: Quality Model", International Standard ISO/IEC 9126, International Standard Organization.

[6] Bertoa, Manuel F.; Vallecillo, Antonio (2002). "Quality Attributes for COTS Components". Proceedings of the 6th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Spain.

[7] Alvaro, A.; Almeida, E. S.; Meira, S. R. L (2005). "Quality Attributes for a Component Quality Model". 10th International Workshop on Component-Oriented Programming (WCOP), Glasgow, Scotland.

[8] Carvallo, Juan Pablo; Franch, Xavier (2006). "Extending the ISO/IEC 9126-1 Quality Model with Non-Technical Factors for COTS Components Selection". International Workshop on Software Quality, May 21-21, Shanghai, China.

[9] B. Meyer (2003), "The Grand Challenge of Trusted Components", 25th International Conference on Software Engineering (ICSE), USA, pp. 660–667.

[10] Reusable Asset Specification. (2005). http://www.omg.org/docs/ptc/05-04-02.pdf. Acessado em 02/11/2005.

[11] Hoff, Arthur van; Partovi, Hadi; Thai, Tom (1997). "The Open Software Description Format (OSD)". Submission to the World Wide Web Consortium (W3C). http://www.w3.org/TR/NOTE-OSD. Acessado em 22/11/2006.

[12] Varadarajan, Sriram (2001). "ComponentXchange: An E-Exchange for Software Components". Dissertação de Mestrado, Indian Institute of Technology.

[13] Schuenck, Michael; Negócio, Yuri; Elias, Glêdson (2006). "Um Modelo de Metadados para Representação de Componentes de Software". VI Workshop de Desenvolvimento Baseado em Componentes.

[14] Oliveira, João Paulo; Schuenck, Michael; Elias, Glêdson. "ComponentForge: Um Framework Arquitetural para Desenvolvimento Distribuído Baseado em Componentes. VI Workshop de Desenvolvimento Baseado em Componentes, 2006.

Mineração de Componentes para a Revitalização de Softwares Embutidos

Marcelo A. Ramos[1,2], Rosângela A. D. Penteado[2]

[1]ADC/LAC - Centro de Desenvolvimento de Aplicações para a América Latina
VeriFone do Brasil
São Paulo, SP, Brasil

[2]DC/UFSCar - Departamento de Computação
Universidade Federal de São Carlos
Rod. Washington Luís, Km 235
Caixa Postal 676, CEP 13565-905, São Carlos, SP, Brasil
marcelo_r1@verifone.com, {marcelo_ramos, rosangel}@dc.ufscar.br

Abstract. *In order to reduce costs, to minimize risks, to anticipate deadlines and to optimize resources of projects of new products it must, whenever possible, to reuse artifacts of existing similar products. These artifacts must be of easy adaptation to meet the requirements of the new products. This paper proposes the use of mining of generic software components from embedded legacy systems to collect and to document the knowledge contained in such systems, to achieve their revitalization and to produce a core of reusable assets to support the fast development of similar products. Software Product Line techniques are used for domain modeling and components design. A case study for POS (Point of Sale) terminals domain is presented.*

Resumo. *Para reduzir custos, minimizar riscos, antecipar prazos e otimizar recursos de projetos de novos produtos deve-se, sempre que possível, reutilizar artefatos de produtos similares existentes. Esses artefatos devem ser de fácil adaptação para satisfazerem aos requisitos dos novos produtos. Este trabalho propõe o uso de mineração de componentes genéricos de software a partir de sistemas embutidos legados, para reunir e documentar os conhecimentos contidos nesses sistemas, efetuar suas revitalizações e produzir um núcleo de ativos reutilizáveis para apoiar o rápido desenvolvimento de produtos similares. Técnicas de Linha de Produtos de Software são utilizadas para a modelagem de domínio e projeto de componentes. Um estudo de caso para o domínio de terminais POS (Point Of Sale) é apresentado.*

1. Introdução

No domínio de sistemas embutidos, variabilidades do hardware ocorrem muitas vezes de maneira não planejada. Componentes antigos são freqüentemente substituídos por outros mais modernos, baratos e eficientes, produzindo versões melhoradas do produto original. Possivelmente, alguns desses componentes nem existiam quando a primeira versão do produto foi entregue. Muitas empresas desse domínio desejam que a evolução de seus softwares ocorra de maneira semelhante, com a reutilização parcial ou integral dos artefatos de suas aplicações bem sucedidas, para a criação de novas aplicações com pouca ou nenhuma manutenção [Graaf, Lormans e Toetenel 2003]. Nesse contexto, a mineração de componentes a partir do código legado e a sua posterior reconexão ao sistema original é uma técnica que pode não só modernizar o código existente, mas também estender as funções do sistema, garantindo-lhe uma sobrevida.

Este trabalho propõe o uso de mineração de componentes genéricos de software a partir de sistemas embutidos legados, para reunir e documentar os conhecimentos neles contidos, efetuar suas revitalizações e produzir um núcleo de ativos[1] reutilizáveis para apoiar o rápido desenvolvimento de uma Linha de Produtos de Software (LPS). Além dessa introdução sua organização é a seguinte: A Seção 2 descreve fundamentos teóricos, presentes na literatura, relacionados ao contexto deste trabalho e relevantes para a sua realização; A Seção 3 reúne trabalhos relacionados à presente proposta; A Seção 4 apresenta um modelo de processo para a mineração de ativos para LPS, a partir de sistemas embutidos legados, que pode ser usado para apoiar a revitalização de seus códigos, como mostra a Seção 5; A Seção 6 ilustra um estudo de caso para o domínio de terminais POS (*Point Of Sale*), para exemplificar a presente proposta e, finalmente, na Seção 7 são colocadas as considerações finais e os trabalhos futuros.

2. Fundamentos Teóricos

Sistemas embutidos são sistemas computacionais especializados, hardware e software, que integram sistemas com funcionalidade mais ampla, realizando tarefas específicas e pré-definidas. Softwares escritos para sistemas embutidos são freqüentemente chamados de softwares embutidos ou *firmwares*. Apesar da variedade de projetos, esses softwares geralmente atuam em hardwares com conjunto reduzido de recursos e capacidade limitada de processamento e armazenamento de informações [Vahid e Givargis 2002].

Reuso de software se refere à construção de sistemas a partir de artefatos existentes, ao invés de desenvolvê-los a partir do zero [Krueger 1992]. Diferentes técnicas têm sido pesquisadas com a finalidade de elevar o reuso para níveis de abstração mais altos, como por exemplo: Componentes [Szyperski 2002] e Linha de Produtos de Software [Clements e Northrop 2001].

Componentes de software são elementos de composição, independentes do contexto, implementados para uma determinada especificação[2], distribuídos de maneira autônoma e que, por meio suas interfaces, agregam funções aos sistemas que integram [Szyperski 2002]. Porém, no domínio de sistemas embutidos, componentes geralmente

[1] Tradução do termo original em inglês *core assets*.
[2] Por exemplo: COM (***C**omponent **O**bject **M**odel*), EJB (***E**nterprise **J**ava **B**ean*), etc.

não são elementos autônomos (*run-time component*s), mas sim códigos escritos em linguagens de alto nível, que podem ser conectados ao código de uma aplicação durante a criação de versões do sistema (*build-time components*) [Crnkovic 2005]. Componentes com alto grau de generalidade e fácil adaptação permitem a criação de soluções reutilizáveis para requisitos similares em diferentes domínios. Para isso, devem agregar propriedades intrínsecas dos domínios aos quais se aplicam, para apoiar a implementação das variabilidades neles existentes [Bergey, O'Brien e Smith 2000].

Linha de Produtos de Software consiste em um conjunto, ou família, de produtos pertencentes a um determinado domínio, que tem como base uma arquitetura comum. Seu objetivo é ampliar a eficiência dos processos de desenvolvimento, explorando a identificação e o reuso das similaridades e controlando as variabilidades dos membros de uma família [Atkinson et al 2002]. Um processo genérico de engenharia para LPS é constituído de duas atividades principais: Engenharia de Domínio, que cria o núcleo de ativos reutilizáveis e a infra-estrutura de desenvolvimento da LPS, e Engenharia de Aplicação, que desenvolve produtos membros da família a partir desses recursos [Ziadi, Jézéquel e Fondement, 2003]. O núcleo de ativos reutilizáveis de uma LPS é composto por seus requisitos, modelos de domínio, arquiteturas, componentes de software etc.

Mineração de ativos frequentemente se refere ao processo de extração não trivial de artefatos potencialmente úteis e reutilizáveis, embutidos nos sistemas legados de uma empresa [Bergey, O'Brien e Smith 2000].

Features são propriedades de um domínio, visíveis ao usuário, que permitem identificar semelhanças e diferenças entre os sistemas de software desse domínio. Com o objetivo de auxiliar a identificação de propriedades importantes ou especiais de um domínio durante a fase de análise, Kang et al (1990) introduziram o modelo de *features* em seu método FODA (***F**eature **O**riented **D**omain **A**nalysis)*. Nesse modelo, as *features* são organizadas hierarquicamente em forma de árvore e são unidas por relacionamentos estruturais formando agrupamentos. Cada *feature* possui um especificador próprio que a define como obrigatória, opcional ou alternativa. A Figura 1 mostra um modelo de *features* que exemplifica as definições propostas por Kang et al (1990).

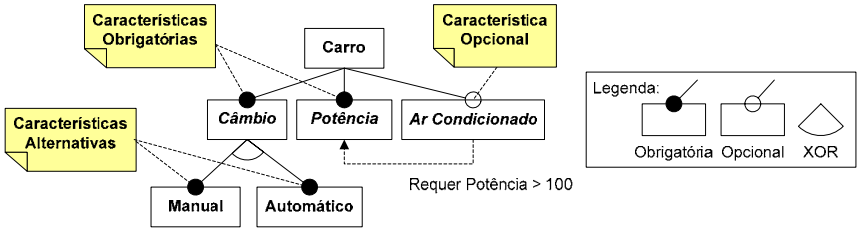


Figura 1. Modelo de *features* {Adaptado de [Kang et al 1990]}

Diferentes notações têm sido propostas para ampliar a representatividade desse modelo em relação a diferentes tipos de relacionamentos estruturais. Por exemplo, Czarnecki e Eisenecker (2000) distinguem relacionamentos XOR, entre *features* alternativas, mutuamente exclusivas, de OR, entre *features* opcionais, que podem ser combinadas entre si. Riebisch et al (2002) expandem os relacionamentos OR com a notação de multiplicidade da UML [Booch, Rumbaugh e Jacobson 1999], ampliando o número de combinações possíveis entre suas *features*. Gomaa (2004) propõe uma forma alternativa de representar as *features* de um domínio, baseada na UML.

3. Trabalhos relacionados

Muitos dos métodos para a criação de LPS propõem o planejamento antecipado de todos os produtos que podem ser derivados da infra-estrutura comum de desenvolvimento, a ser criada para uma determinada família. Assim, não mencionam claramente como prover o apoio a variabilidades não planejadas. Adicionalmente, muitos deles não citam a possibilidade e a maneira de criar LPS a partir dos artefatos legados de um domínio. Por esse motivo, não foram incluídos nessa seção.

Métodos para a criação de LPS no domínio de sistemas embutidos incluem, geralmente, uma atividade inicial de reengenharia para criar ativos reutilizáveis a partir de sistemas legados, para apoiar o desenvolvimento futuro de uma família de produtos. Um exemplo é o método FOOM (***F**eature-based **O**bject **O**riented **M**odeling*) [Ajila e Tierney 2002]. No entanto, tarefas potencialmente complexas, como a recuperação e a transformação da arquitetura, ainda são realizadas integralmente antes que uma única variabilidade, mesmo que simples, seja disponibilizada na forma de um novo produto. Um método incremental, que permita agregar gradativamente ao código legado apoio às variabilidades do domínio, com base nas necessidades imediatas e particulares de cada empresa, pode ser uma alternativa para reduzir prazos, riscos e custos associados à criação de famílias de produtos. Porém, nenhum dos métodos consultados adota essa abordagem. Alguns trabalhos, realizados em outros domínios de aplicação, podem apoiar evoluções de softwares embutidos para LPS, segundo a abordagem descrita. A seguir são mencionados alguns deles.

Mehta e Heineman (2002) propõem uma metodologia para a modernização de sistemas de software, que agrega *features*, testes de regressão e engenharia de software baseada em componentes. A proposta prevê a modernização de um conjunto de *features* identificadas durante testes de regressão. O código associado a cada *feature* é identificado, refatorado, convertido em componente de software e inserido novamente na aplicação. Os componentes criados podem ser reutilizados em outras aplicações. O'Brien (2005) observa que se os paradigmas de desenvolvimento do sistema legado e dos componentes recém criados forem distintos, *gateways* devem ser desenvolvidos para permitir a reconexão de seus códigos.

Bergey, O'Brien e Smith (2000) propõem a realização da mineração de ativos para LPS em quatro passos: 1) Reunir as informações preliminares, 2) Decidir pela mineração e definir sua estratégia, 3) Obter entendimento técnico detalhado dos ativos existentes e 4) Recuperar os ativos. Bosch e Ran (2000) consideram essencial a criação de um modelo de *features* para apoiar a realização dessa tarefa.

4. Um Modelo de Processo para a Mineração de Ativos para LPS

Comumente tecnologias tradicionais de desenvolvimento de software não consideram necessidades específicas associadas à criação de softwares embutidos e restrições usuais desse domínio, como a limitação de memória e as variações do hardware [Graaf, Lormans e Toetenel 2003].

Com base na metodologia de Mehta e Heineman (2002), o modelo de processo proposto prevê a criação de componentes de software a partir de *features* potencialmente úteis, contidas em sistemas embutidos legados. No entanto, o processo de extração de *features* utilizado é resultante de adaptações feitas aos quatro passos

propostos por Bergey, O'Brien e Smith (2000) para a mineração de ativos para LPS e considera, adicionalmente, o uso de modelos de *features* para apoiar a sua realização, como proposto por Bosch e Ran (2000). Dessa forma, não se tem como requisito obrigatório a existência de artefatos provenientes de tarefas supostamente realizadas em desenvolvimentos anteriores, como modelos de análise, relatórios de testes etc. O modelo de processo adaptado é descrito a seguir e considera inicialmente a existência do código legado e da documentação do produto, *i.e.*, hardware, sistema operacional etc.

A reunião de programadores, usuários e outras pessoas ligadas direta ou indiretamente aos sistemas legados inicia o processo e tem como objetivo elicitar as deficiências atuais e as necessidades imediatas e futuras desses sistemas, passo (1). Com essas informações e com a ajuda do cliente é possível efetuar análises preliminares de viabilidade técnica e econômica do projeto de revitalização dos sistemas, com base nas necessidades obtidas anteriormente. Quando aprovada a continuidade da revitalização essas necessidades devem guiar a modelagem de uma estratégia apropriada para a realização do projeto, assim como estabelecer seu escopo, seu objetivo e suas prioridades, passo (2). O passo (3) inicia com a formação de um grupo técnico de pessoas, que contém um ou mais especialistas do domínio, para auxiliar o entendimento da documentação e do código legado existentes, além de esclarecer, da melhor maneira possível, conceitos específicos do domínio e suas particularidades. As prioridades do projeto ajudam a definir o conjunto de conhecimentos a ser adquirido inicialmente, que pode abranger todo um sistema ou apenas uma de suas partes, caso a mineração deva ser realizada de forma gradativa. Técnicas de LPS para modelagem de domínios são utilizadas para documentar os conhecimentos obtidos nessa etapa e apoiar o próximo passo da mineração. De posse das informações coletadas até esse ponto, são iniciadas as atividades de mineração, passo (4), que dependem do objetivo do projeto e da estratégia estabelecida previamente.

Um modelo de *features* é utilizado para a modelagem do domínio e o apoio ao desenvolvimento futuro de componentes genéricos de software. A criação desse modelo deve ser feita de modo que cada *feature* tenha um único componente associado. Para isso, podem ser necessários refinamentos sucessivos do modelo para a remoção de divisões e junções de *features*, Figura 2, descritas por Sochos, Riebisch e Philippow (2006) no método FArM (***F**eature-**Ar**chitecture **M**apping*).

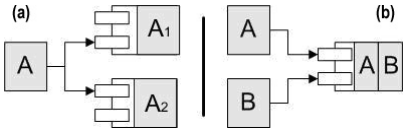


Figura 2. Divisão da *feature* A em dois componentes A1 e A2 (a); Junção das *features* A e B em um único componente AB (b) {Adaptado de [Sochos, Riebisch e Philippow 2006]}

Em uma mineração gradativa, o modelo de *features* do domínio pode ser inicialmente simples, detalhando apenas as *features* que representam as necessidades imediatas e deficiências atuais dos sistemas legados. À medida que os componentes são desenvolvidos, podem ser feitos refinamentos ou adições de novas *features* ao modelo existente. Esse processo é realizado iterativamente.

O código legado das aplicações é utilizado para guiar o projeto das interfaces dos componentes para preservar a funcionalidade existente. Técnicas como herança e parametrização podem ser usadas para estender essa funcionalidade, permitindo que variabilidades do domínio sejam inseridas nesses sistemas. Para cada *feature*, deve-se efetuar uma busca no código legado procurando identificar todas as funções diretamente relacionadas a ela. A partir dessas funções, um Mapa de Conexão (MCo) é construído e usado como base para o projeto das interfaces do componente associado à *feature*. Quando houver um conjunto de sistemas similares, o MCo pode retratar de forma unificada a conexão de seus códigos com uma determinada *feature* comum. Isso aumenta a generalidade da interface do componente em desenvolvimento, ampliando as possibilidades de seu reuso no projeto de novos produtos de uma mesma família. A Figura 3 mostra de forma simplificada as atividades do modelo de processo apresentado. A seção seguinte apresenta a proposta de revitalização de códigos legados a partir desse processo.

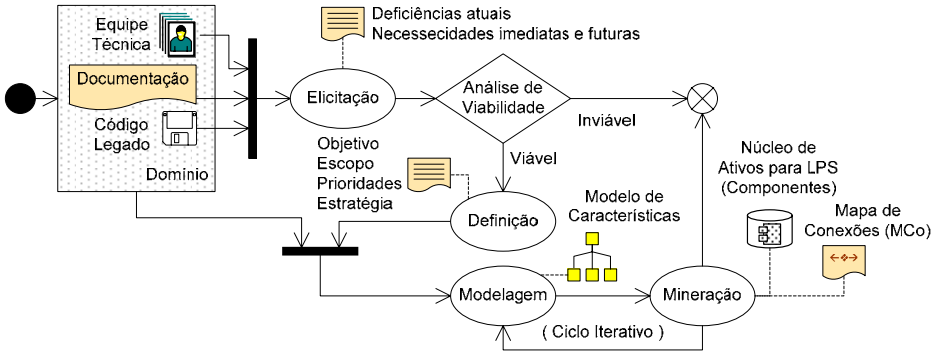


Figura 3. Modelo de processo para a mineração de ativos para LPS

5. Revitalização de Códigos Legados a partir da Mineração de Ativos

A realização do processo para a mineração de ativos para LPS descrito na Seção 4 resulta em: a) um modelo de *features*, que retrata as propriedades de um domínio, b) em um MCo, que documenta a conexão dessas *features* com o código dos sistemas legados pertencentes a esse domínio e c) em um núcleo de componentes de software, cujas interfaces reproduzem as funções atuais presentes nesse código. Embora possam ser utilizados de forma independente para a criação de novos produtos, esses componentes podem também, de forma apropriada, ser reconectados ao código legado para melhorar a sua estruturação e agregar novas funções à aplicação, proporcionando uma sobrevida ao produto original. Essa forma de revitalização considera a construção de *gateways*, como proposto por O'Brien (2005), permitindo o uso de diferentes paradigmas para o projeto dos componentes. O MCo e a documentação das interfaces dos componentes produzidos durante a mineração de ativos fornecem as informações necessárias para a construção dos *gateways*, que devem atuar como adaptadores de interface. Para a reconstrução dos sistemas originais usando os artefatos recém-criados, deve-se remover do código legado as implementações das funções contidas nos componentes mantendo, porém, suas chamadas, que serão redirecionadas para os componentes pelos *gateways*. Dessa forma, a revitalização dos sistemas legados é realizada sem nenhuma alteração na estrutura original de seus códigos, tornado o processo transparente para os mantenedores desses sistemas.

A revitalização, quando realizada de forma gradativa, deve facilitar a realização de testes para a validação dos sistemas reconstruídos, uma vez que as alterações são focadas em *features* isoladas e inteiramente implementadas nos componentes.

A seguir, um estudo de caso é apresentado para exemplificar o processo de revitalização aqui descrito.

6. Estudo de Caso

Para a realização deste estudo de caso foram tomadas três aplicações legadas similares para Transferências Eletrônicas de Fundos (TEF), desenvolvidas para terminais POS (*Point of Sale*), pertencentes à empresa VeriFone do Brasil, responsável por gerir o sigilo das informações contidas nos artefatos disponibilizados. Esses terminais são equipamentos pequenos, leves, portáteis e versáteis, com um conjunto reduzido de componentes periféricos padronizados, controlado por um sistema embutido microprocessado, que atendem às necessidades de um amplo segmento de mercado. No entanto, o custo, a capacidade de processamento e armazenamento de informações, a interface simplificada com o usuário e os dispositivos de comunicação, segurança e leitura de cartões com chip e trilha magnética direcionam seu uso para a realização de TEF com cartões de pagamento. A Figura 4 mostra um terminal POS e seus principais componentes periféricos.

Figura 4. Terminal POS e seus principais componentes periféricos[3]

Os documentos existentes das três aplicações foram entregues juntamente com os códigos legados. Essas aplicações são distintas, sem documentação de projeto, possuem estrutura modular e código escrito em linguagem C, com níveis insatisfatórios de reuso. As variabilidades existentes são decorrentes das freqüentes evoluções tecnológicas do hardware e foram implementadas na forma de blocos de código condicionais, denominados Módulos Selecionáveis[4]. O objetivo da empresa é evoluir rapidamente cada uma de suas aplicações legadas para diferentes plataformas de hardware, porém similares, preservando as regras de negócio nelas embutidas, as quais já passaram por um processo rigoroso de certificação e não devem ser modificadas. Programadores experientes da empresa atuaram como Especialistas do Domínio durante o processo. O processo para a mineração de ativos para LPS e a revitalização dos códigos legados a partir dos componentes produzidos foi realizado como segue.

[3] Terminal POS Vx-570 da empresa VeriFone Inc. (http://www.verifone.com)
[4] Denominação interna, criada pelos programadores da empresa.

Em uma reunião inicial envolvendo áreas estratégicas da empresa foi feito o levantamento das deficiências e das necessidades imediatas e futuras das três aplicações, na visão de cada participante. De posse dessas informações e dos artefatos fornecidos foi possível efetuar uma análise preliminar de viabilidade técnica do projeto de revitalização das aplicações. Diante do interesse em preservar as regras de negócio e permitir variações do hardware decidiu-se pela revitalização gradativa e concomitante das aplicações fornecidas, com enfoque nas *features* do hardware, iniciando com variabilidades de baixo impacto operacional, para facilitar a validação preliminar do processo a partir de seus primeiros resultados. Definida a estratégia, iniciou-se o estudo do domínio com a participação dos especialistas designados pela empresa. Após a leitura da documentação os códigos legados foram analisados e as dúvidas esclarecidas para entendimento dos elementos do hardware e seu uso pelas aplicações. O conhecimento adquirido foi documentado na forma de um modelo de *features*, mostrado na Figura 5. *Features* físicas e virtuais do hardware foram associadas nesse modelo para retratar não apenas a constituição física do equipamento, mas também a sua estruturação lógica do ponto de vista do sistema operacional. Como resultado pode-se observar, por exemplo, o console (*feature* lógica) constituído de tela e teclado (*features* físicas). Essa associação facilitou o mapeamento do modelo de *features* para um modelo de classes e posteriormente de componentes.

Os especialistas do domínio verificaram que informações importantes associadas às *features* do hardware não puderam ser claramente representadas no modelo, resultando na omissão de requisitos importantes. Para suprir essa deficiência foi criada uma extensão para o modelo de *features* denominada Nota, descrita a seguir.

Notas são representadas como na UML, podendo ser conectadas a uma ou mais *features* por meio de âncoras. Por questão de legibilidade, possuem internamente apenas números seqüenciais, que fazem referência a textos explicativos externos ao modelo. De um modo geral, as Notas agregam ao modelo de *features* detalhes do domínio que não podem ser representados graficamente e que desempenham um papel importante na tomada de decisões de projeto. Podem, ainda, conter informações temporárias em um determinado nível de abstração, que servirão como guias para refinamentos do modelo em etapas posteriores do projeto. Quando o texto de uma nota descreve uma informação relevante para a implementação da *feature*, uma letra 'i' é acrescentada à frente do número seqüencial.

Na seqüência foram iniciadas as atividades de mineração dos ativos para LPS descritas na Seção 4, com base na estratégia inicialmente definida. Como as aplicações legadas não implementam apoio à conexão de teclado externo e sendo essa uma variabilidade de baixo impacto operacional e importante para a estratégia de negócios da empresa, optou-se por validar o processo a partir de sua implementação. Foram realizadas buscas nos códigos legados das três aplicações para identificar as funções diretamente associadas à *feature* escolhida. Para apoiar o desenvolvimento do componente correspondente, foram adicionadas informações sobre propriedades desejáveis da *feature* e importantes limitações dos teclados interno e externo, que foram anotadas como extensão do MCo, ilustrado na Tabela 1.

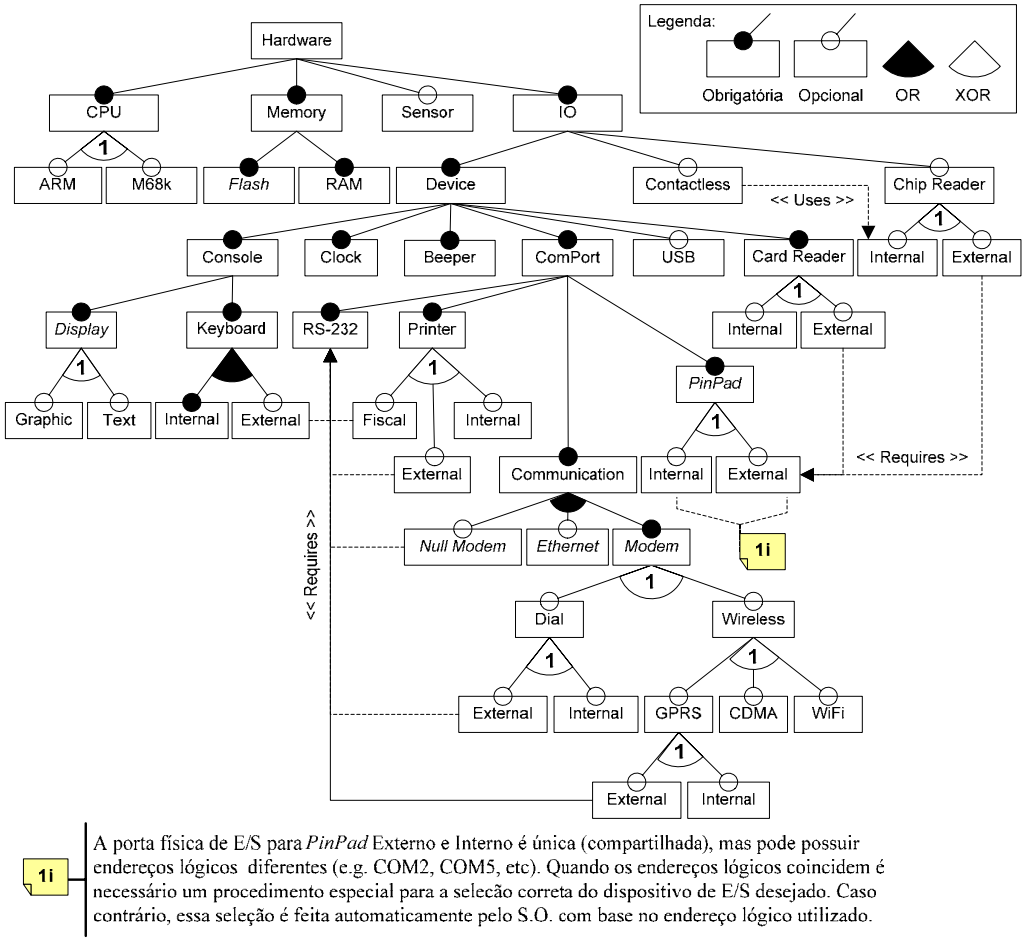


Figura 5. Modelo de *features* do hardware de terminais POS[5]

Tabela 1. Mapa de Conexões (MCo) da *feature Keyboard*

***Feature*:**		**Hardware -> IO -> Device -> Console -> Keyboard**	
Função	**Parâmetros**	**Retorno**	**Comentários**
`KBHIT`	Nenhum	Status: bool	Retorna TRUE se houver tecla pressionada.
`get_char`	Nenhum	Tecla : int	Aguarda o pressionamento de uma tecla e retorna seu valor.
Limitações: ▪ O sistema operacional não permite escritas no *buffer* do teclado interno. ▪ O teclado externo não possui um buffer de teclas provido pelo sistema operacional. **Novos Requisitos:** ▪ Os teclados, interno e externo, devem compartilhar um *buffer* comum de teclas, provido pelo componente, o qual deve permitir operações de escrita e leitura. ▪ A fim de flexibilizar o seu uso, a conexão do teclado externo pode ser feita nas portas RS-232 ou PinPad do terminal POS, com parâmetros de comunicação configuráveis.			

O desenvolvimento do componente foi iniciado a partir do MCo e do modelo de *features* do hardware, a partir do qual foi instanciado um sub-modelo para retratar apenas as *features* a serem tratadas, como mostra a Figura 6. Nesse sub-modelo foram

[5] Os nomes das *features* estão em inglês por exigência da empresa.

realizados e documentados refinamentos em relação ao modelo inicial, para apoiar os novos requisitos contidos no MCo. Uma dependência do teclado externo em relação à *feature PinPad* e uma nova *feature* de software, inexistente no código legado, os arquivos tipo INI[6], para permitir o armazenamento dos parâmetros de comunicação, foram adicionados.

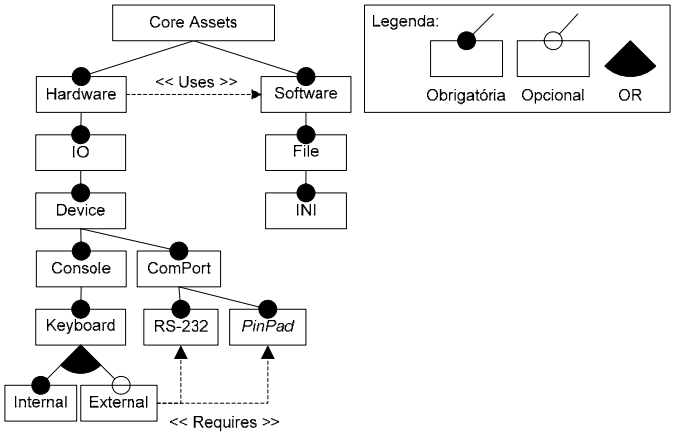


Figura 6. Sub-Modelo de *Features* para o componente *Keyboard*

Nesse modelo as *features Internal Pinpad* e *External Pinpad* não foram retratadas, pois apenas a capacidade de comunicação serial da *feature PinPad*, provida pela *feature ComPort*, deve ser implementada. O mapeamento do modelo de *features* da Figura 6 para um modelo de classes em UML foi feito a partir das informações contidas no MCo e na documentação do sistema operacional, que contêm os serviços disponibilizados para cada dispositivo de hardware. O resultado é o modelo de classes mostrado na Figura 7, no qual os teclados interno e externo foram unificados pela classe *Keyboard*, à qual foi adicionada a capacidade de escrita, atendendo aos requisitos do projeto. À classe *ComPort* foi adicionado um método para a configuração dos parâmetros de comunicação, para atender a esse mesmo conjunto requisitos.

A classe para tratamento de arquivos do tipo INI, não incluída neste artigo, implementa a leitura de sessões do arquivo, delimitadas por colchetes, armazena suas chaves de configuração na memória e permite a consulta de seus valores. A partir desse modelo foi desenvolvido um modelo de componentes para documentar os serviços oferecidos pelos novos ativos reutilizáveis criados, como mostra, de forma simplificada, a Figura 8. A Figura 9 mostra o arquivo de configuração HARDWARE.INI criado, com as duas possibilidades de conexão para o teclado externo, EXT-RS232 e EXT-PINPAD, em conformidade com os requisitos dessa *feature*.

Uma vez criados os componentes foi iniciada a implementação dos *gateways*, guiado pelo conteúdo do MCo. Como as funções listadas representam chamadas diretas à API (***A**pplication **P**rogramming **I**nterface*) do sistema operacional, não houve a necessidade de remover suas respectivas implementações do código legado. Porém, devido ao conflito de nomes, essas foram renomeadas e receberam um prefixo _G_ para identificar que estão implementadas no *gateway*. Nenhuma outra alteração foi feita no código legado das aplicações.

[6] Arquivo padrão do sistema operacional Windows para o armazenamento de configurações do ambiente.

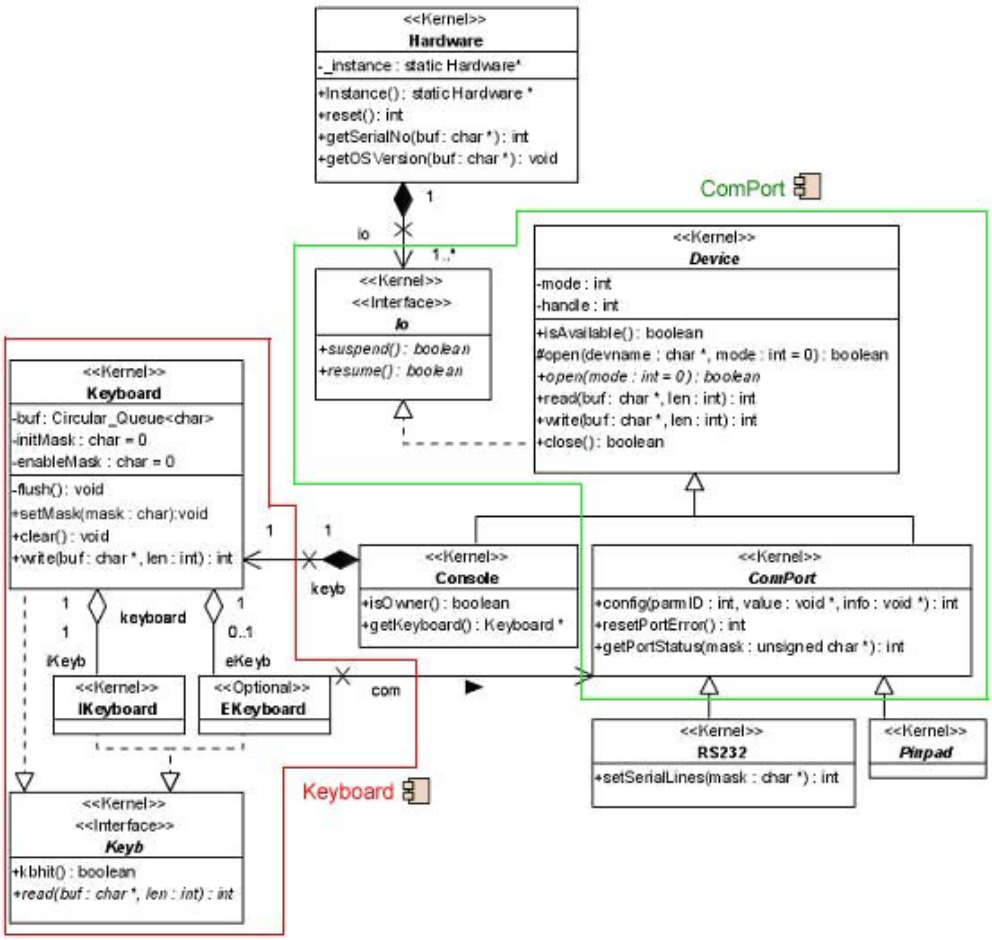


Figura 7. Modelo de Classes da *Feature Keyboard*

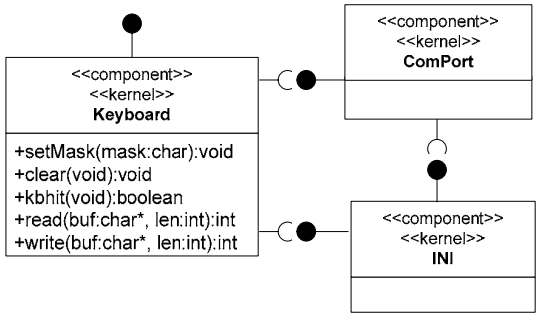


Figura 8. Modelo Simplificado de Componentes da *Feature Keyboard*

```
[CONSOLE]
KEYBOARD=EXT-PINPAD
[EXT-RS232]
# <1 = COM1, 2 = COM2>
COM=1
# MODE: <protocol>, <baudrate>, <data length>, <parity>, <stop bits>, <stx_char>, <etx_char>
MODE=CHAR,115200,8,N,1
[EXT-PINPAD]
COM=2
MODE=CHAR,19200,7,E,1
```

Figura 9. Arquivo HARDWARE.INI com as duas possibilidades de conexão para o teclado externo, EXT-RS232 e EXT-PINPAD

O *gateway* desenvolvido para a *feature Keyboard* é mostrado na Figura 10. Maiores detalhes dessa implementação foram omitidos devido à restrição de espaço.

```
using namespace std;

#include "Console.h"
#include "Keyboard.h"
#include "Sdk.h"
#include "Gateway.h"
namespace Hw
{    class Console   ;
     class Keyboard  ;
}
Hw::Console*    _c  = NULL  ;
Hw::Keyboard*   _k  = NULL  ;

void initConsole(void)
{    /* Defines static variability control through hardware.ini configuration file */
          _c = new Hw::Console( "HARDWARE.INI", "CONSOLE" );
     /* Address the Keyboard component .. */
     _k = _c->getKeyboard(); /* .. configured by the key "Keyboard =" in the [CONSOLE] section */
}
int _G_kbhit (void)  /* Keyboard MCo's KBHIT() function gateway */
{        if  (   !_k )initConsole( ) ; /* Console owns the Keyboard and must be initialized firstly */
     return  (int)_k->kbhit( )       ; /* Redirect the call to the Keyboard component */
}
int _G_getchar(void) /* Keyboard MCo's get_char() function gateway */
{
     char    ch                      ;
     while(!_G_kbhit())              ; /* Once a key is detected ..  */
               _k->read(&ch, 1)      ; /* .. reads it                */
     return  (int)    ch             ;
}
```

Figura 10. *Gateway* para a *feature Keyboard*

Após a sua construção, o *gateway* foi compilado com os códigos legados das aplicações juntamente com a implementação dos componentes, que agora passam a compor o núcleo de ativos reutilizáveis da empresa e podem apoiar o rápido desenvolvimento de produtos similares. Como resultado ressalta-se que foi possível operar as aplicações originais com ambos os teclados, de maneira configurável, preservando seus respectivos conteúdos, conforme solicitação inicial do cliente. Os resultados foram apresentados ao cliente, que aprovou a continuidade da revitalização, para a criação de componentes para as demais *features* do hardware.

7. Considerações Finais e Trabalhos Futuros

Para o domínio de sistemas embutidos, o apoio às variabilidades do hardware permite a oferta simultânea de diferentes produtos similares ao mercado, constituindo uma importante estratégia de negócios para as empresas. No domínio de terminais POS, o rigoroso processo de certificação das aplicações para TEF requer que as regras de negócio, embutidas nos códigos legados, permaneçam inalteradas e isoladas de erros durante a criação de novos produtos. Especialistas do domínio são comuns nesses ambientes, facilitando a realização de muitas das atividades propostas neste trabalho.

O estudo de caso apresentado, mostrou que é possível gerenciar as variabilidades do hardware por meio de componentes de software parametrizáveis, isolados do código legado e conectados a ele por meio de *gateways*. Exemplificou o uso de mineração de componentes para promover a revitalização de múltiplas aplicações similares, concomitantemente. Ao seu final, foram obtidos resultados satisfatórios de reuso, mesmo com a realização de uma revitalização incompleta, porém planejada.

O apoio unificado aos teclados interno e externo pelo componente *Keyboard* revelou uma situação não prevista inicialmente, na qual a junção de características, Figura 2(b), foi útil para atender aos requisitos de uma *feature.*

A estrutura Unix-like do sistema operacional dos terminais POS utilizados não permite às aplicações o acesso direto ao hardware, cujos recursos são disponibilizados por meio de uma API de comandos. Essa particularidade facilitou a construção do MCo, diminuindo a necessidade de interpretações do código legado.

Apesar de aplicada ao domínio de softwares convencionais, a técnica proposta por Mehta e Heineman (2002) pôde ser adaptada para o domínio de terminais POS, que possui *features* particulares, descritas anteriormente. A ela foram feitas as seguintes melhorias: 1) Além de modernizar o código legado, por meio da extração de *features* e inserção de componentes equivalentes, foi possível revitalizá-lo, estendendo sua funcionalidade para atender a novos requisitos; 2) O conjunto de componentes criados permitiu não só o reuso de *features* do software legado em outras aplicações, mas também a implementação de variabilidades do domínio em seu próprio código, permitindo a rápida criação de uma família de produtos; 3) A substituição dos testes de regressão por modelos de *features*, para apoiar a extração de *features* do código legado, permitiu a visualização das similaridades entre diferentes aplicações e das variabilidades do domínio, possibilitando a criação de interfaces de componentes mais flexíveis e a revitalização concomitante de produtos similares, além da recuperação progressiva da documentação, guiada pelas prioridades estabelecidas pelo cliente; 4) O uso de *gateways* para isolar o código dos componentes do código legado permitiu o uso de paradigmas mais modernos de desenvolvimento para a sua criação e também tornou o processo de revitalização transparente para os mantenedores do código legado, que puderam efetuar manutenções em pleno andamento da revitalização; 5) A revitalização gradativa dos códigos legados foram realizadas com baixo risco para o cliente, que pôde avaliar precocemente os custos e eficiência do processo a partir dos primeiros resultados parciais obtidos.

Durante o mapeamento do modelo de *features* em classes de objetos e componentes foi observada a ocorrência repetitiva de determinadas soluções, para as quais será realizado um estudo futuro para a identificação de possíveis padrões. Ferramentas generativas e de apoio à gerência de configuração devem ser construídas para facilitar a criação e a manutenção de famílias de produtos. Devem ser pesquisadas técnicas alternativas para a extração de características a partir de códigos legados.

8. Referências Bibliográficas

Ajila S.A.; Tierney P.J. (2002). The FOOM Method - Modeling Software Product Line in Industrial Settings. Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, USA. <http://www.sce.carleton.ca/faculty/ajila/SERP02 - Camera Ready.pdf>. Acesso: 24.06.07

Atkinson, C.; Bayer, J.; Bunse, C.; Kamsties, E.; Laitenberger, O.; Laqua, R.; Muthig, D.; Paech, B.; Wust, J.; Zettel, J. (2001) Component-Based Product Line Engineering with UML. 1st Edition, Addison-Wesley Professional, USA.

Bergey J.; O'Brien L.; Smith D. (2000). Mining Existing Assets for Software Product Lines. Technical Note, CMU/SEI-2000-TN-008, SEI, USA.

Booch, G.; Rumbaugh, J.; Jacobson, I. (1999). The Unified Modeling Language Reference Manual, Addison-Wesley, USA.

Bosch, J.; Ran, A. (2000). Evolution of Software Product Families. Proceedings of the International Workshop on Software Architectures for Product Families. LNCS, v. 1951/2000, p. 168-183, Springer-Verlag, UK.

Clements, P.; Northrop, L. (2001). Software Product Lines: Practices and Patterns. 1st Edition, Addison-Wesley Professional, USA

Crnkovic I. (2005). Component-based software engineering for embedded systems. Proceedings of the 27th International Conference on Software engineering, p. 712-713. ACM Press, USA.

Czarnecki, K.; Eisenecker, U.W. (2000). Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, USA.

Gomaa H. (2004). Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional, USA.

Graaf, B.; Lormans, M.; Toetenel H. (2003). Embedded Software Engineering: The State of the Practice. IEEE Software, v. 20, p. 61-69.

Kang, K.; Cohen, S.; Hess, J.; Novak, W.; Peterson S. (1990). Feature-Oriented Domain Analysis (FODA): Feasibility Study. CMU/SEI-90-TR-21, SEI, USA.

Krueger C. W. (1992). Software reuse. ACM Computing Surveys, v.24(2), p. 131-183.

Mehta A., Heineman G.T. (2002). Evolving legacy system features into fine-grained components. Proceedings of the 24th International Conference on Software Engineering, Orlando, FL. p. 417-427. ACM Press, USA.

O'Brien, L.(2005). Reengineering. SEI, USA. <http://www.cs.cmu.edu/~aldrich/courses/654-sp05/handouts/MSE-Reeng-05.pdf>. Acesso: 17.06.07.

Riebisch, M.; Böllert, K.; Streitferdt, D.; Philippow, I. (2002) Extending Feature Diagrams with UML Multiplicities. 6th World Conference on Integrated Design & Process Technology, USA. <http://www.theoinf.tu-ilmenau.de/~riebisch/publ/IDPT2002-paper.pdf>. Acesso: 17.06.07.

Sochos, P.; Riebisch, M. e Philippow I. (2006). The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines. 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, p. 308-318.

Szyperski C. (2002) Component Software: Beyond Object-Oriented Programming. 2nd Edition, Addison-Wesley, USA.

Vahid, F.; Givargis, T. (2002), Embedded System Design: A Unified Hardware/Software Introduction. John Wiley & Sons, USA.

Ziadi, T.; Jézéquel, J-M.; Fondement, F. (2003). Product line derivation with UML. In Proceedings Software Variability Management Workshop, University of Groningen. <http://lglpc35.epfl.ch/lgl/members/fondement/docs/papers/Ziadi03b.pdf>. Acesso: 17.06.07.

Construction of Analytic Frameworks for Component-Based Architectures

George Edwards, Chiyoung Seo, Nenad Medvidovic

University of Southern California
Los Angeles, CA 90089-0781 U.S.A.

{gedwards,cseo,neno}@usc.edu

Abstract. *Prediction of non-functional properties of software architectures requires technologies that enable the application of analytic theories to component models. However, available analytic techniques generally operate on formal models specified in notations that cannot flexibly and intuitively capture the architectures of large-scale distributed systems. The construction of model interpreters that transform architectural models into analysis models has proved to be a time-consuming and difficult task. This paper describes (1) how a reusable model interpreter framework can reduce the complexity involved in this task, and (2) how such a framework can be designed, implemented, utilized, and verified.*

1. Introduction

Modern-day component technology provides software architects with powerful mechanisms for designing, implementing, deploying, and evolving large-scale distributed systems. In particular, component technology encompasses both highly effective strategies for reuse and integration of existing software (*e.g.*, off-the-shelf components), as well as run-time platforms (*e.g.*, component-based middleware) that hide low-level complexity beneath high-level design abstractions. Perhaps more subtly, component-based software engineering also offers a basis for the construction of analysis models that enable the discovery and prediction of critical system properties, such as performance, reliability, and resource consumption. Although techniques for analyzing software systems with respect to such properties are not new, the assembly of systems from independently deployable and executable units makes these techniques more relevant and practical.

Analysis of non-functional properties is critical in weighing design alternatives. Nearly all non-trivial architectural decisions come down to trade-offs between multiple desirable properties, and a software architect is required to engineer the right balance between conflicting goals. Furthermore, subtle interactions between components can result in unforeseeable and unpredictable system behaviors. Quantitative evaluation of non-functional properties therefore provides an architect with concrete rationale for fundamental design decisions and reduces the risk associated with a large-scale development and/or integration project [2].

To effectively analyze the non-functional properties of a component-based system, methods and tools are needed that support the integration of component technologies and analysis technologies [1]. A *component technology* consists of a component model along with a development environment and/or run-time platform. The component model imposes rules that define the well-formedness of component instances and assemblies. Component technologies (such as Enterprise Java Beans) provide the basis for the modeling, implementation, and deployment of software architectures. An *analysis technology* consists of a system analysis technique and tools that support the utilization of that technique. An analysis technique is a process for applying a computational theory to

system models (such as Layered Queuing Networks, or LQNs) to enable automated prediction of system properties and behaviors. Software and system analysis techniques are required to make assumptions about the systems to which they are applied. For example, a LQN assumes that each software server accepts requests from a single queue [13]. Such an assumption can be enforced by a component middleware platform at system construction-time and at run-time. Component technologies and analysis technologies are well-suited to integration because a component technology can be used during system construction to enforce the assumptions required by an analysis technology.

Unfortunately, the integration of component and analysis technologies is anything but straightforward in practice. Component-based architectures are generally specified using high-level design languages that emphasize abstraction and flexibility, while analysis techniques operate on formal models that are frequently specified in much lower-level, more rigid notations. The consequence of this is that software architects are frequently required to construct multiple system models for different purposes. For example, the safety experts on an architecture team may build and analyze fault trees, while energy management experts construct and execute specialized power simulations.

This paper presents an approach to achieving the seamless integration of component technologies and analysis technologies. At the core of our approach is the development of highly flexible *model interpreter frameworks*, which implement semantic mappings between a component model and an analysis model, yet lend themselves to a wide variety of non-functional analyses. A model interpreter framework can be reused by a software architect to rapidly construct analyzable models from domain-specific architectures. To illustrate the approach, we describe our implementation of a model interpreter framework in XTEAM, a modeling and analysis framework targeted at mobile and resource-constrained software systems. Our evaluation of XTEAM demonstrates that (1) an interpreter framework can provide accurate predictions of the non-functional properties of a software architecture, and (2) a single interpreter framework can be used to rapidly and successfully implement a broad range of analysis techniques.

The remainder of this paper is organized as follows. Section 2 further motivates this work. Section 3 describes our approach, while Section 4 discusses in detail our implementation of the XTEAM model interpreter framework. Section 5 evaluates the framework quantitatively and illustrates its benefits through the use of an example. Overviews of related work and conclusions round out the paper.

2. Background and Motivation

Model-driven engineering (MDE) [4] offers an attractive strategy for analyzing the non-functional properties of software architectures. MDE technologies enable the construction of *domain-specific modeling languages* (DSMLs) through the use of metamodels. Metamodels capture the elements, attributes, relationships, views, and constraints present in a modeling language, and can be easily modified, adapted, composed, enhanced, and evolved [5]. In this way, MDE offers an intuitive way to incorporate the parameters of an analytic theory into both general-purpose and domain-specific component models. MDE technologies provide access to the information contained in architectural models through well-defined interfaces. Customized *model interpreters* can then be constructed that perform system analysis and visualization, automated synthesis of implementation artifacts, *etc*. Figure 1 delineates the main MDE concepts and processes.

Model interpreters can be used to implement semantic mappings, or transformations, between high-level architectural models and the low-level analysis models amenable to rigorous prediction of component assembly properties. However, numerous practical challenges remain. In order to motivate the discussion in the remainder of this paper and illustrate the need for a new approach to the construction of model interpreters, this section describes a typical MDE-based process for modeling and analyzing a software architecture, and demonstrates how this process can be simplified and improved.

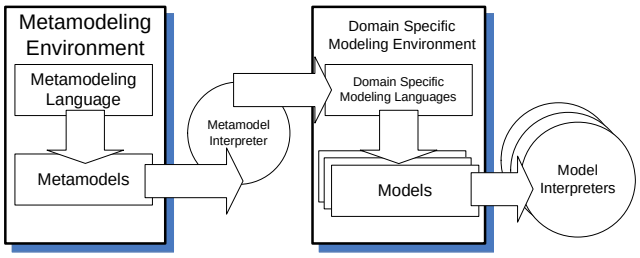


Figure 1. High-level view of the MDE process.

Consider a large-scale software development project. The software architecture team has decided to employ an MDE-based modeling and analysis process, and has consequently constructed an architectural model that includes some domain-specific elements (such as hardware and middleware) in addition to the canonical architectural constructs (component, connector, *etc.*). The team now plans to analyze the performance of the system through the use of a LQN model. Applying the standard MDE strategy, the team constructs a model interpreter that transforms the architectural model into a LQN, which is then analyzed to determine a set of performance-related metrics, such as system throughput and service utilization, under various loading conditions.

As the development program progresses, the need for additional analyses becomes apparent. For example, as the system's deployment of software components to hardware hosts is further refined, questions arise about how deploying certain components to mobile hosts will impact the system's energy consumption. The architecture team is instructed to study deployment alternatives with respect to energy consumption. To do so, they implement a new model interpreter that transforms the architecture model into the input to a cycle-accurate energy consumption simulator.

As other forms of analysis are requested, the team is forced to expend significant resources implementing model interpreters. For each new interpreter, the team must:

- Find a computational theory that derives the relevant properties from a system model.
- Determine the syntax and semantics of the modeling constructs on which the computational theory operates.
- Discover the semantic relationships between the constructs present in the architectural models and those present in the analysis models.
- Determine the compatibility between the assumptions and constraints of the architectural models and the analysis models, and resolve conflicts between the two.
- Implement a model interpreter that executes a sequence of operations to transform an architectural model into an analysis model.
- Verify the correctness of the transformation implemented by the interpreter.

This paper demonstrates how the use of a model interpreter framework allows an architecture team to perform the above tasks only *once* for a broad *family* of analysis techniques, rather than repeating the process for *each* analysis technique. The use of interpreter frameworks can significantly improve the utility and appeal of MDE-based architectural development.

3. Model Interpreter Frameworks

Applying MDE to the analysis of component-based systems requires software architects to construct semantic mappings between component models and analysis models. The primary contribution of this paper is a novel approach that can greatly reduce the complexity involved in this task. Our approach is to leverage general purpose architectural modeling constructs and a widely applicable analytic representation to construct a *model interpreter framework* that abstracts most of the semantic mapping required for analysis, while still providing the extensibility to accommodate both domain-specific modeling elements and analyses. This section, therefore, focuses on defining specifically what an interpreter framework is, what the requirements of one are, and what capabilities an interpreter framework provides. In the next section, we focus on how an interpreter framework can be implemented and give concrete examples of how one can be used.

3.1. Definition

In the model-driven engineering paradigm, a *model interpreter* is a software component that operates on the information captured in a system model to produce some useful artifact. Model interpreters invoke an API provided by a modeling environment to extract the model structure and properties. A model interpreter codifies the semantics of the modeling constructs on which it operates by defining the consequences of the use of those constructs within a given context.

A *model interpreter framework*, then, is an infrastructure for constructing a family of model interpreters. In order to be useful, such a framework must encapsulate logic or algorithms that are useful in a wide variety of contexts. However, an interpreter framework is not a library of functions; rather, it is an active component that can be extended and enhanced in specific, predefined ways. Furthermore, an interpreter framework necessarily makes assumptions about the models on which it operates, and is therefore only applicable to a certain class of models. In the context of MDE, which advocates the inclusion of domain-specific constructs in modeling languages, this implies that a common base of domain-independent constructs exists on which the framework can operate; domain-specific constructs are then handled by framework extensions.

One example of a model interpreter framework is a component that synthesizes "glue-code" for a given middleware platform from a model of a software application. Such a model likely includes both domain-independent constructs, such as objects or components, and domain-specific constructs, such as the representation of the application business logic. A model interpreter framework can be constructed that utilizes the component interface specifications and topology to generate middleware glue-code, but leaves open extension mechanisms to insert logic that interprets the domain-specific behavior (*e.g.*, to generate component implementations).

When applied to the analysis of component-based systems, a model interpreter framework enables a family of analytic techniques to be applied to a component model by constructing from a high-level architectural model a more directly analyzable representation of a system, such as a discrete event simulation or Markov chain. In this context, the domain-independent elements of the model are those concepts common to all component-based architectures. The domain-specific elements of the model are the parameters of a relevant analytic theory, plus any additional domain- and platform-specific extensions and constraints. The model interpreter framework abstracts the semantic

mapping from architectural constructs to analysis constructs, while providing the extensibility to accommodate the logic that measures and records non-functional properties according to an analytic technique. The role of a model interpreter framework in the analysis of component-based software architectures is illustrated in Figure 2.

3.2. Assumptions and Requirements

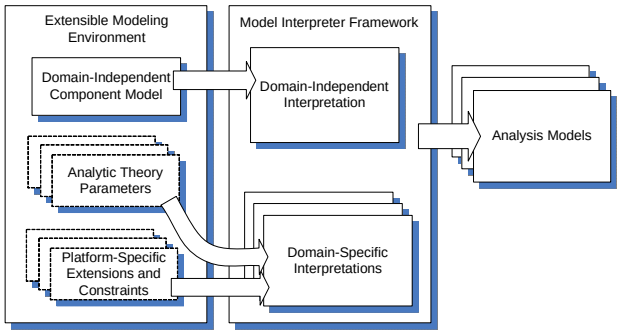


Figure 2. The role of a model interpreter framework in the analysis of component-based architectures.

As alluded to in the previous subsection, a model interpreter framework must make several important assumptions about the models to which it will be applied. It also must satisfy several requirements in order to be effective. In this subsection, we enumerate these assumptions and requirements and describe their consequences in terms of model interpreter frameworks in general. We also describe, for each assumption and requirement, the specific implications for model interpreter frameworks that provide non-functional analysis of application architectures.

3.2.1.Assumptions

Assumption 1. System models contain domain-independent elements that are sufficient to implement an interpretation. Interpreter frameworks encapsulate logic that operates on domain-independent constructs. It therefore follows that models must contain a sufficient set of domain-independent constructs to implement some useful interpretation. Modeling languages that consist exclusively of domain-specific constructs are not amenable to an interpreter framework. This assumption is clearly satisfied in the case of architecture models: the set of domain-independent elements common to most component-based architectures includes components, connectors, interfaces, and events.

Assumption 2. The interpretation of domain-independent elements is not dependent on the interpretation of domain-specific elements. The implementor of a model interpreter framework cannot know the types of domain-specific extensions that will be present in system models. Consequently, the framework logic must operate exclusively on domain-independent modeling elements, and the semantics of those elements cannot change within different domain-specific contexts. This assumption is not a significant problem for architectural models: the domain-specific modeling elements in this case are generally either the parameters of an analytic theory that will be applied to the model or platform-specific constraints on components and connectors.

Assumption 3. Domain-specific constraints do not violate domain-independent constraints. Constraints on the set of well-formed models are fundamental to every modeling language, and a model interpreter framework relies on these constraints in applying semantics to a model. Within a given domain, additional constraints are present; capturing these constraints is a crucial part of creating a domain-specific modeling language. Clearly, for a model interpreter framework to execute, these domain-specific constraints cannot contradict any domain-independent constraints.

This last assumption can, in some cases, constitute a major challenge when

applying an interpreter framework to architectural models. The constraints of a component model may be irreconcilable with the assumptions required by an analytic theory. However, more commonly, these constraints and assumptions can be brought into alignment by *co-refinement*, a process proposed by Wallnau et al. [1]. Co-refinement may weaken or strengthen the constraints of a component model, which either expands or reduces the set of well-formed models, respectively, in order to accommodate the assumptions of an analytic theory. Similarly, the assumptions of a analytic theory can be strengthened or weakened in order to reconcile conflicting component constraints.

3.2.2.Requirements

Requirement 1. The model interpreter framework abstracts the details of domain-independent interpretation. The manipulations performed by an interpreter framework are necessarily at least somewhat complex (otherwise, the reuse of the framework would be of little value). An interpreter framework should insulate architects from the details of these manipulations in order to enable reuse without forcing the architect to understand or modify the framework logic. This requirement, can, however, be relaxed in some cases, in order to increase the flexibility of the framework. Exposing the details of the interpretation process increases the complexity of utilizing the framework, but also allows the architect to implement certain analyses that would not otherwise be possible.

Requirement 2. The model interpreter framework produces an artifact useful in a wide variety of contexts. In order to maximize the benefits provided by reuse of an interpreter framework, the framework must produce a representation of the system that is flexible enough to be used for a variety of purposes. For example, some analysis models, such as discrete event simulations, enable the realization of an extensive family of analytic theories. Other analysis models, such as fault trees, are much more narrowly targeted, and enable a much smaller set of analytic theories. The latter types of analysis models are therefore not strong candidates for construction of an interpreter framework.

Requirement 3. The model interpreter framework provides extension mechanisms sufficient to accommodate domain-specific interpretation. The inclusion of extension mechanisms within an interpreter framework is the crucial feature that allows them to be applied to domain-specific models. As noted earlier, contemporary software architectures are increasingly incorporating domain-specific information as a strategy for managing complexity. Extension mechanisms are created through the use of design patterns such as Template Method, Strategy, and Functor [6]. These patterns allow domain-specific logic to be inserted into the interpreter framework at points of variability. Of course, the interpreter framework designer cannot predict every possible variability point. The choice of whether to include an extension mechanism at a potential point of variability is a design trade-off between flexibility and usability; that is, the inclusion of additional variability points makes the framework more widely applicable, but also increases the burden on a software architect utilizing the framework in a domain-specific context.

4. The Design of a Model Interpreter Framework

In this section, we describe in detail the design of a model interpreter framework we implemented as part of the eXtensible Toolchain for Evaluation of Architectural Models (XTEAM) [2], an environment that leverages the MDE paradigm to provide a reusable infrastructure for realizing domain-specific architectural analyses.

4.1. The XTEAM Toolchain

The eXtensible Toolchain for Evaluation of Architectural Models (XTEAM) allows an architect to analyze architectural models through a model interpreter framework that maps component models to executable simulations. Furthermore, XTEAM incorporates mechanisms to accommodate domain-specific extensibility at both the modeling and analysis phases of the architectural evaluation process.

A high-level view of XTEAM is shown in Figure 3. Using the Generic Modeling Environment (GME) [7], we created a domain-independent component model by composing the elements of the xADL Structures and Types ADL [19] and the Finite State Processes (FSP) ADL [20]. GME uses this component model to create a modeling environment in which architectural models that conform to the component model can be created. We also implemented a model interpreter framework that maps the component model to an analysis model — a discrete event simulation — and implements appropriate extension mechanisms, which are described further in the next subsection.

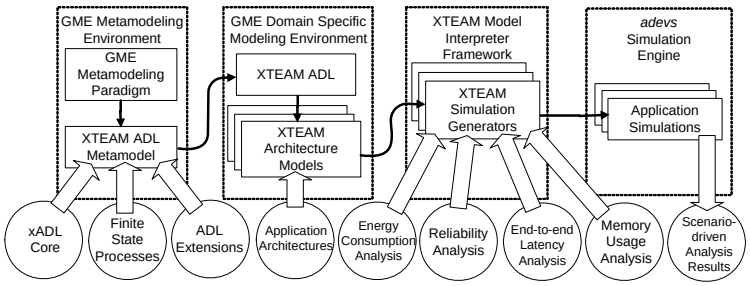


Figure 3. The eXtensible Toolchain for Evaluation of Architectural Models.

An architect takes advantage of the extensibility in XTEAM in the following way. First, the component model is enhanced to include attributes and elements that capture the parameters of a relevant analytic theory. XTEAM currently implements modeling extensions for energy consumption [11], reliability [15], latency, and memory usage analyses as examples. The architect then utilizes the extension mechanisms built into the model interpreter framework in such a way as to generate simulations that measure, analyze, and record the properties of interest. This has been accomplished for the four analyses listed above to demonstrate the capability.

The modeling capabilities of XTEAM are described fully in [2]; the remainder of this paper focuses on the design and evaluation of the XTEAM interpreter framework.

4.2. The XTEAM Model Interpreter Framework

The XTEAM model interpreter framework implements a semantic mapping between a flexible and extensible domain-independent component model and a simulation model. As described above, XTEAM provides a modeling environment built on top of GME that allows an architect to extend the XTEAM component model by defining new elements, attributes, and constraints that (1) tailor the model to a specific component technology, such as the OSGi platform [8] or CORBA Component Model (CCM) [9] and (2) allow the inclusion of the parameters required by an analytic theory.

When invoked by an architect, the XTEAM interpreter framework traverses the architectural model, building up a discrete event simulation model in the process. The interpreter framework maps components and connectors to discrete event constructs, such as atomic models and static digraphs. The FSP-based behavioral specifications are

translated into the state transition functions employed by the discrete event simulation engine. The interpreter framework also creates discrete event entities that represent various system resources, such as threads.

The interpreter framework employs the Strategy pattern [17] to enable an architect to implement domain-specific extensions, as depicted in Figure 4. The Strategy pattern allows a set of related algorithms to be transparently interchanged within different contexts. The different algorithms are abstracted by a common interface. In the XTEAM interpreter framework, each algorithm generates code that encapsulates logic to realize a particular analytic theory. For example, the logic may implement equations that calculate non-functional system properties based on the parameters defined in the model and equations defined by the theory. The algorithms are invoked at specific times during the interpretation process, such that the code generated by those algorithms will be invoked when various events occur during an actual simulation run. These events include a component receiving or sending data, invoking an interface, initiating or completing a task, *etc*.

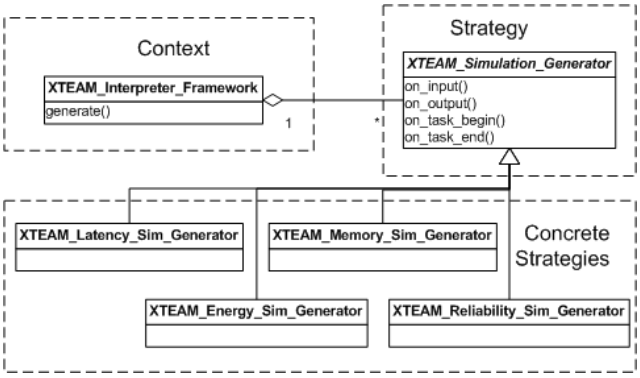


Figure 4. High-level design of the XTEAM model interpreter framework.

To illustrate the process used by an architect to realize a given analysis using an interpreter framework, we now describe the implementation of the XTEAM energy consumption simulator. The energy consumption estimation technique described in [11] provides a mechanism for estimating software energy consumption at the level of software architecture. The estimation technique provides equations that enable the calculation of energy costs based on a number of parameters, including data sizes and values, characteristics of the hardware hosts, and network bandwidth. Energy is used by the system whenever either (1) data is transmitted over the network or (2) the software is required to perform computation. Consequently, the equations defined by the energy consumption estimation technique were inserted into the Strategy methods corresponding to the sending and receiving of data and the invocation of an interface. The equations calculate the energy cost of a given data transmission or computation based on the parameters defined in the model, and record these values for later examination by architects. The implementation of the other XTEAM simulation generators follows the same approach.

5. Evaluation

This section describes how XTEAM was utilized to provide a key non-functional analysis that ultimately guided the choice of architectural style for a given application. Furthermore, this section compares the predictions of system properties made by XTEAM with measured values taken from the executing system. In our experiments, XTEAM simulations were shown to produce predicted values for system energy consumption that fell within 10% of the observed values, and guided the software architects to the correct choice of architectural style. This result illustrates the utility of XTEAM in making fundamental architectural decisions early in the development cycle.

5.1. Application Scenario

To illustrate the importance of non-functional analysis, consider the MIDAS family of sensor network applications [18]. An instance of MIDAS consists of sensors, gateways, hubs, and PDAs. Sensor nodes collect data about the environment and transmit that data to gateways over wireless links. Gateways manage groups of sensors, aggregate and fuse sensor data, and forward the fused data to hubs. Hubs analyze fused sensor data, generate visualizations of the data, and provide a user interface for configuring and managing the system. PDAs provide mobile access to the data visualizations and system management capabilities. The distributed software system, which is described and analyzed in this section, is implemented on top of a lightweight, component-based middleware platform, called Prism-MW [10], which enables architecture-based development of distributed applications in embedded and pervasive environments.

The MIDAS system is subject to a number of non-functional requirements. For this evaluation, we analyzed an instance of MIDAS that provides building monitoring services, such as intrusion and fire detection. In this scenario, the MIDAS hardware devices are not connected to a continuous power supply, but instead run on battery power. Therefore, the system's efficiency with respect to energy consumption has a critical impact on the longevity of the system services.

One of the most influential factors in the system's overall energy consumption is the cost of sending and receiving data over the wireless network. As a result, the type and frequency of interactions between software components has a major impact on the system's energy usage. Component interactions are, in turn, governed to a large extent by the choice of an architectural style. It was crucial, therefore, that the MIDAS system employ an energy efficient architectural style, while still fulfilling numerous other functional and non-functional requirements. Based on the system requirements as a whole, two candidate architectural styles, client-server and publish-subscribe, were selected. Two models of the MIDAS security application — each using one of the candidate styles — were then created in XTEAM and compared with respect to energy consumption.

5.2. Modeling and Analysis

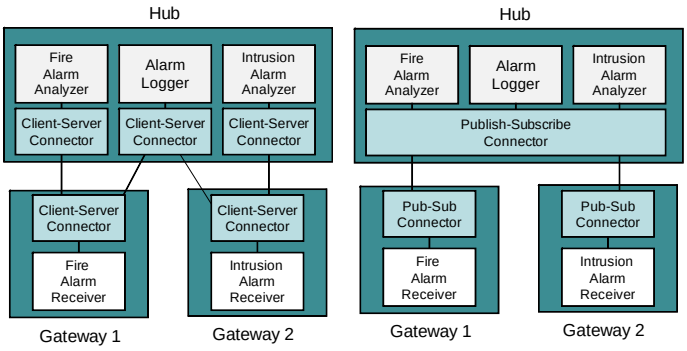


Figure 5. A subset of MIDAS designed in client-server (left) and publish-subscribe (right) styles.

Figure 5 shows the same subset of MIDAS designed using the two styles. The *FireAlarmReceiver* and *IntrusionAlarmReceiver* components deployed on the gateways translate, aggregate and fuse alarm events received from the sensors periodically, and propagate them to the components deployed on the hub. The *Analyzer* components deployed on the hub analyze the alarm data and determine whether there is actually a fire or intrusion. If the *FireAlarmAnalyzer* (*IntrusionAlarmAnlyzer*) component concludes that there is a fire (intrusion), it transmits a sensor-activation message to the *FireAlarmReceiver* (*IntrusionAlarmReceiver*) component, which in turn sends an activation signal to all the fire (intrusion) sensors.

For the client-server architecture, we modeled the behavior of client-server connectors based on a request-response protocol. Client-server connectors are frequently implemented as middleware stubs and skeletons. The behavior of the application components was also modeled according to the above scenario: the *Receiver* components act as clients and invoke interfaces on the *Analyzer* and *Logger* components via their local client-server connectors. The client-server connector on a gateway then transmits a request event (from its local *Receiver* component) to the *Analyzer* and *Logger* components separately, which indicates that each request requires two transmissions on each gateway.

For the publish-subscribe architecture, we modeled the behavior of connectors based on a typical publish-subscribe interaction protocol. For example, the *FireAlarmAnalyzer* sends a message to the connector that requests a subscription to fire alarm events. When a component, such as *FireAlarmReceiver*, publishes a fire alarm event, the connector routes the event to each subscribed component. The behavior model of each component is essentially the same as that in the client-server architecture, except that components publish and subscribe to events. For instance, the *FireAlarmAnalyzer* has the same behavior for processing fire alarm events as in the client-server architecture, but includes additional logic that transmits event subscription requests to the publish-subscribe connector. In this architecture, the publish-subscribe connector can optimize the transmission of events based on the location of publishers and subscribers (as is done in the publish-subscribe service implementations of widely-used middleware platforms [16]). Therefore, compared with the client-server architecture, the publish-subscribe architecture may require fewer events to be sent over the wireless network, but incurs the additional overhead of managing lists of publishers and subscribers.

XTEAM requires the following host-specific energy costs to analyze the above two architectural styles with respect to their energy costs:

1. *The communication energy cost on each host due to transmitting and receiving data over the network.* Previous research [3] has shown that the energy consumption of wireless communication is directly proportional to the size of transmitted and received data and can be expressed as a linear equation with the size of data exchanged. Our energy estimation tool [11] details the steps for determining the communication energy cost on a specific hardware platform.
2. *The energy consumption on each host due to processing a subscription and retrieving a set of subscribers for a published event.* These energy costs can be determined by leveraging the measurement setup described in [11].

Note that we do not need to consider the computational energy cost of most component event processing (*e.g.*, the energy consumption of the *FireAlarmAnalyzer* due to processing a fire alarm) in comparing the energy costs of the candidate architectural styles because this cost is the same for both styles.

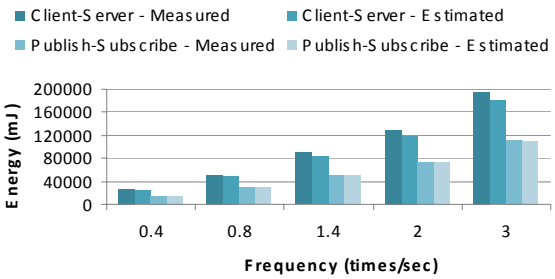


Figure 6. Comparison of the energy consumption of MIDAS using the client-server and publish-subscribe styles.

Once the architectural model was parameterized with the above information, the XTEAM energy consumption simulation generator was invoked. XTEAM allows simulations to include various stochastic behaviors, such as the frequency of client requests, the probability of cache misses, or the

sizes and values of data. In this case, the timing and size of events was determined stochastically, and four different average rates of event transmission were simulated. The results of the energy consumption simulation are shown in Figure 6. The XTEAM analysis predicted that utilizing the publish-subscribe style would result in significant energy savings. The next section describes how we verified the correctness of this result.

5.3. Verification

In order to determine the accuracy of the energy consumption estimates made by XTEAM, we need to know the *actual* energy consumption of the distributed software system. To this end, we used a digital multimeter and the experimental setup described in [11]. The MIDAS application discussed in Section 6.1 was implemented using both the client-server and publish-subscribe styles on top of Prism-MW. We used the same average frequencies and sizes of alarm events as those simulated in XTEAM, measured the energy consumption on each host, and finally calculated the software system's overall energy consumption by summing up the three hosts' energy costs.

For each candidate style, we compared the actual overall energy consumption with the energy consumption estimates generated by XTEAM for different rates of event transmission. As shown in Figure 6, the predicted energy consumption fell within 10% of the measured energy consumption in all the scenarios analyzed. In addition, the publish-subscribe style was determined to be much more energy-efficient for this scenario because (1) the publish-subscribe style requires fewer events to be sent over the wireless network and (2) the energy savings obtained by the reduced data exchange over the network exceeds the energy overhead due to processing subscription requests and retrieving the set of subscribers for each published event.

This result demonstrates that although the architectural models cannot be parameterized with perfect accuracy — especially when XTEAM is being applied early in the architectural development process — the predictions provided by XTEAM are accurate enough to enable architects to successfully determine trade-offs between relatively course-grained design alternatives, such as the choice of architectural style.

5.4. Limitations

Although the experiment described above establishes both the quality and utility of XTEAM predictions of non-functional properties, there are several limitations to the applicability of our approach. First, XTEAM's model interpreter framework relies on the ability to measure and quantify a given system property. Properties that are difficult or impossible to quantify, such as usability [23], cannot be predicted using XTEAM's discrete event simulation-based analysis. Second, the properties of a component assembly must be derivable from a composition of (1) the properties of individual components, (2) the overall software architecture, and (3) the system's usage profile. For example, properties that depend on the environment in which the system is used are not amenable to analysis in XTEAM. An example of such a property is security [22], which is heavily impacted by characteristics of the computing infrastructure (*e.g.*, network and operating system) and external, human factors. While these types of concerns can be added to XTEAM's modeling language through metamodel extensions, XTEAM's focus is on software architecture, and consequently the corresponding extensions to the model interpreter framework would likely require significant effort. Finally, XTEAM is intended to predict system run-time properties rather than lifecycle properties related to construction

activities. For example, the maintainability of a system is derivable from its software architecture [21], but is not compatible with XTEAM's dynamic analysis. The non-functional property classification scheme described in [22] provides a good mechanism for determining what properties can be effectively analyzed by XTEAM.

6. Related Work

This section establishes the broader context in which our work resides. First, we discuss a conceptual framework that provides a basis for the ideas discussed in this paper. Second, we describe a representative approach to component-based architectures.

6.1. Prediction-Enabled Component Technology

Predication-enabled component technology (PECT) is a proposed framework for the integration of component technologies and analysis technologies [1]. A PECT can be used to determine the emergent properties of a highly complex assembly of software components when certain characteristics of the individual components can be certified. PECT relies on component design tools and run-time environments to enforce the assumptions required by each analysis technique applied to the system.

A PECT instance includes a construction framework and one or more reasoning frameworks [12]. The *construction framework* constitutes the design and implementation facilities, such as modeling environments and code generators, that are used to develop a component-based system. The construction framework relies on an *abstract component technology*, or ACT, to represent component models and run-time platforms. Software architecture and design models, or *constructive models*, that conform to the ACT are created in the construction framework. A *reasoning framework*, on the other hand, constitutes the analysis facilities to be applied to the system. A reasoning framework applies system analysis techniques, or *property theories*, through the use of an *analysis environment*. Discrete event simulators and fault-tree analysis tools are examples of analysis environments. *Interpretations* transform constructive models into analysis models. Component characteristics, which constitute the parameters of property theories, are codified in a component's *analytic interface*. This interface is leveraged by the reasoning framework to apply a system-wide analysis of non-functional properties.

PECT leverages many of the core concepts of MDE to support analysis of the non-functional properties of large-scale component assemblies. PECT establishes an intuitive way of organizing and relating the elements of component technologies and analysis technologies, and outlines a strategy for integrating component models and analysis models that leverages their complementary characteristics. For these reasons, we believe PECT provides a useful conceptual framework for additional research in the modeling and analysis of component-based systems. However, PECT does not address the fundamental challenge described in Section 2; that is, it does not help a software architect discover and realize any particular domain-specific model interpretation.

6.2. CALM and Cadena

Cadena is an extensible environment for the modeling and development of component-based architectures [14]. The Cadena Architecture Language with Metamodeling (CALM) supports the specification of platform- and domain-specific component models, which are leveraged by Cadena to provide automated enforcement of architectural con-

straints. In this way, CALM and Cadena provide a modeling environment that can be readily integrated with a wide variety of component technologies.

CALM is based on a three-tiered typing system. At the *style* tier, an architect defines the kinds of components, connectors, and interfaces that exist within a particular component model or architectural style. The style tier is essentially a metamodeling layer that defines a language of architectural constructs. At the *module* tier, the component and interface types that may exist within a specific application architecture are declared. Finally, at the *scenario* tier, component types are instantiated into a particular configuration or assembly. At each tier, Cadena automatically enforces the constraints imposed by the type system defined at the tier above.

The modeling capabilities of CALM and Cadena provide a powerful and intuitive mechanism for creating application architectures that conform to domain-specific component models. Cadena also provides an integrated model-checking infrastructure, Bogor, which enables automatic verification of the logical properties of a system, such as event sequencing. However, Cadena provides little support for the implementation of additional, domain-specific types of non-functional analysis. Thus, Cadena also forces architects to develop model interpreters from scratch in most cases.

7. Conclusions

This paper presented an approach to the construction of analytic frameworks that enable the prediction of the non-functional properties of component-based systems. Such frameworks allow the rapid construction of model interpreters, which is one of the most complex and difficult activities in the model-driven engineering paradigm. In order to achieve this result, model interpreter frameworks must make several important assumptions about the models to which they are applied, and fulfill a set of design requirements. This paper also demonstrated the process of constructing, utilizing, and validating a model interpreter framework using an example.

Our ongoing work in this area is two-fold. First, we are constructing additional interpreter frameworks and integrating them into the XTEAM environment, in order to more clearly define the scope of applicability of the approach described in this paper. For example, we hope to identify a small set of analysis models for which interpreter frameworks can be constructed that will provide broad coverage of the analysis techniques present in the software architecture literature. Second, we are continuing to apply the current XTEAM interpreter framework in several R&D contexts. For example, we are utilizing XTEAM for the continuing development of the MIDAS family of applications and conducting a rigorous analysis of the impact of styles on non-functional properties.

8. References

[1] S.A. Hissam, J.A. Stafford, K.C. Wallnau (2002). Packaging Predictable Assembly. In *Proc. of the ACM Working Conf. on Component Deployment*, pp. 108-124.

[2] G. Edwards, et al. (2007). Scenario-Driven Dynamic Analysis of Distributed Architectures. In *Proc. of Fundamental Approaches to Software Engineering*.

[3] L.M. Feeney, et. al. (2001). Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment. In *Proc. of IEEE INFOCOM*.

[4] D.C. Schmidt (2006). Model-Driven Engineering. *IEEE Computer*, pp. 41 - 47.

[5] A. Ledeczi, et al. (2001). On metamodel composition. In *Proceedings of the 2001 IEEE International Conference on Control Applications*, pp. 756 - 760.

[6] M. Fayad, D. C. Schmidt (1997). Object-oriented application frameworks. *Communications of the ACM*, pp. 32 - 38.

[7] The Generic Modeling Environment. http://www.isis.vanderbilt.edu/projects/gme/

[8] OSGi: The Open Services Gateway Initiaive. http://www.osgi.org/

[9] CCM: The Corba Component Model. http://www.omg.org/

[10]S. Malek, M. Mikic-Rakic, et al. (2005). A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. *IEEE Trans. on Soft. Engineering*.

[11]C. Seo, et al. (2006). Energy Consumption Framework for Distributed Java-Based Software Systems. Tech. Report USC-CSE-2006-604, Univ. of Southern California.

[12]K. Wallnau (2003). Volume III: A Technology for Predictable Assembly from Certifiable Components. Tech. Report CMU/SEI-2003-TR-009, Software Eng. Institute.

[13]M. Woodside. Tutorial Introduction to Layered Modeling of Software Performance. Carleton University, http://sce.carleton.ca/rads.

[14]A. Childs, et al. (2006). CALM and Cadena: Metamodeling for Component-Based Product-Line Development. *IEEE Computer*.

[15]R. Roshandel, S. Banerjee, L. Cheung, N. Medvidovic, and L. Golubchik (2006). Estimating Software Component Reliability by Leveraging Architectural Models. In *Proc. of the 28th International Conference on Software Engineering*.

[16]G. Edwards, D. C. Schmidt, A. Gokhale, B. Natarajan (2004). Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems. *Proc. of the 42nd Annual ACM Southeast Conference*.

[17]E. Gamma, R. Helm, R. Johnson, and J. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

[18]S. Malek, C. Seo, et al. (2007). Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. *Proc. of the 29th International Conference on Software Engineering (ICSE 2007).*

[19]E. Dashofy, et al. (2002). An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proc. of the 24th International Conference on Software Engineering*, pp. 266 - 276.

[20]J. Magee, et al. (1999). Behaviour Analysis of Software Architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture*, pp. 35 - 50.

[21]N. Lassing, et al. (2002). Experiences with ALMA: Architecture-Level Modifiability Analysis. *Journal of systems and software*, Elsevier, pp. 47-57.

[22]I. Crnkovic, et al. (2005). Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes. Architecting Dependable Systems III, Springer, LNCS 3549, Editor(s): R. de Lemos et al., pp. 257-278.

[23]Eelke Folmer, et al. (2004). Software Architecture Analysis of Usability. In *Proc. of the IFIP Working Conf. on Eng. for Human-Computer Interaction*, pp. 321-339.

Technical Session IV: Model-driven Development and Web Services

Usando Ontologias, Serviços Web Semânticos e Agentes Móveis no Desenvolvimento Baseado em Componentes

Luiz H. Z. Santana, Antonio Francisco do Prado, Wanderley Lopes de Souza, Mauro Biajiz

Departamento de Computação (DC) – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13565-905 – São Carlos – SP

{luiz_santana, prado, desouza, mauro}@dc.ufscar.br

Abstract. *This paper presents an approach that combines Ontologies, Semantic Web Services and Mobile Agents, for the Component-Based Software Development. The Ontologies are employed to improve the problem domain analysis, and to get software components with a semantic description, which may be reused in a wide variety of applications. The Semantic Web Services are used as software components distributed over the Internet, and are composed to perform complex application tasks. The Mobile Agents manager the use of the Semantic Web Services, and can move through the network nodes in order to find, to composite and to monitor these services.*

Keywords: Component-Based Development, Ontologies, Semantic Web Services, Mobile Agents

Resumo. *Este artigo apresenta uma abordagem, que combina Ontologias, Serviços Web Semânticos e Agentes Móveis, para o Desenvolvimento Baseado em Componentes. As Ontologias são empregadas para aprimorar a análise do domínio do problema e para obter componentes de software com descrições semânticas, os quais podem ser reutilizados numa grande variedade de aplicações. Os Serviços Web Semânticos são utilizados como componentes de software distribuídos pela Internet e são compostos para realizar tarefas complexas de aplicações. Os Agentes Móveis gerenciam o uso de Serviços Web Semânticos e podem migrar através dos nós da rede a fim de encontrar, compor e monitorar esses serviços.*

Palavras-chave: Desenvolvimento Baseado em Componentes, Ontologias, Serviços Web Semânticos, Agentes Móveis

1. Introdução

Dentre as inúmeras pesquisas, que visam melhorar o Processo de Desenvolvimento de Software (PDS), destaca-se cada vez mais o uso de componentes de software. O Desenvolvimento Baseado em Componentes (DBC) é caracterizado pela integração, composição e adaptação de componentes pré-existentes, com ênfase no reuso de software, e apresenta vantagens tais como o aumento de produtividade e da qualidade de software [Lucrédio et al. 2004].

Um tópico importante de pesquisa no DBC é a construção de componentes suficientemente genéricos para serem reutilizados em diferentes aplicações. Uma

possível solução para esse problema é desenvolver um processo de construção de componentes para um domínio específico, baseando-se na análise desse domínio [Arango 1989]. Neste sentido, a Engenharia de Software introduziu recentemente o uso de ontologias, a qual possibilita a formalização do conhecimento, um melhor entendimento sobre o domínio e a sua representação num nível alto de abstração [Evermann e Wand, 2005].

Evolução dinâmica e não antecipada de software é um outro tópico importante no DBC, tendo em vista que os requisitos iniciais para a construção de aplicações podem ser modificados, causando assim alterações em seus componentes [Ebraert et al. 2005]. Uma possível solução para esse problema é o uso de Serviços Web Semânticos. Esses serviços encontram-se distribuídos pela Internet, podem ser modificados e usados pelas aplicações sem que seja necessário reconstruí-las. Entretanto, o uso desses serviços ainda oferece desafios: encontrar o serviço necessário; compor um conjunto desses serviços para atender a tarefas complexas; e monitorar um desses serviços ou uma composição dos mesmos, a fim de garantir o sucesso na execução da tarefa.

Freqüentemente Agentes de Software são empregados para facilitar o uso de Serviços Web Semânticos [Charif-Djebbar e Sabouret 2006, Lee 2007]. Interpretando as especificações desses serviços, Agentes de Software podem derivar conjuntos de regras simples, que definem o uso de Serviços Web Semânticos. Os agentes podem também migrar através dos nós de uma rede, a fim de lidar com a heterogeneidade das plataformas de hardware e software e com a distribuição dos serviços.

Em função do exposto, este trabalho apresenta uma abordagem para DBC, que combina Ontologias, Serviços Web Semânticos e Agentes Móveis. O uso de Ontologias visa à obtenção de componentes mais genéricos e padronizados, formalizando e incrementando a descrição do conhecimento do domínio de um problema. Os Serviços Web Semânticos possuem descrições semânticas que orientam o seu emprego nas soluções de problemas complexos que ocorrem na Internet. Os Agentes Móveis realizam a gerência, encontrando, compondo e monitorando a execução, do uso desses serviços, sendo que esses agentes podem migrar, através dos nós da rede, para utilizar serviços distribuídos pela Internet.

A seqüência desse artigo está organizada da seguinte forma: a seção 2 fornece uma visão geral de Análise de Domínio e Ontologias; a seção 3 versa sobre Serviços Web Semânticos, Agentes de Software e apresenta um framework que usa todas essas tecnologias; a seção 4 discorre sobre a abordagem proposta para DBC; a seção 5 descreve um estudo de caso, que foi desenvolvido empregando-se essa abordagem; a seção 6 compara trabalhos correlatos ao deste artigo; finalmente a seção 7 tece algumas conclusões e aponta para trabalhos futuros.

2. Análise de Domínio e Ontologias

O objetivo da Análise de Domínio é evitar que o processo de elicitação e codificação do conhecimento seja refeito para toda aplicação construída num mesmo domínio. Isso impede que o processo seja exposto a erros e inconsistências que já poderiam ter sido resolvidas, além de evitar perda de tempo, esforço e conseqüentemente recursos [Arango 1989]. Modelos de domínio, produtos da análise, são usados pela comunidade de reuso de software como especificações em alto nível de abstração. Estes modelos são

uma formulação genérica o suficiente para representar um conjunto de problemas, conhecimentos ou atividades similares. Uma das formas de obter e representar esses conhecimentos baseia-se em Ontologias.

Segundo [Gruber 1993], "Uma ontologia é uma especificação formal e explícita de conceitos compartilhados". A utilização de ontologias nessa etapa do processo de desenvolvimento traz vantagens como: melhoria da comunicação entre as pessoas envolvidas, uma vez que facilita a obtenção de um consenso sobre o vocabulário e os significados dos termos num domínio; formalização do conhecimento, já que a especificação do domínio em ontologias elimina contradições e inconsistências, resultando em especificações não ambíguas; e, principalmente, a representação do conhecimento para reuso, já que a ontologia descreve o conhecimento do domínio de forma explícita no seu mais alto nível de abstração. Possibilitando especializar o conhecimento durante o desenvolvimento de diferentes aplicações num domínio, que tenham propósitos variados, sejam criados por equipes distintas e em diferentes momentos.

Além disso, os modelos de ontologias apresentam vantagens em relação aos modelos tradicionais gerados pela análise de domínio (e.g., modelos de entidades e relacionamentos e modelos de objetos), já que estes são limitados para a representação de conhecimento, estabelecendo apenas significados particulares e estruturação para os conceitos envolvidos.

3. Serviços Web Semânticos e Agentes Móveis

Para aumentar a reutilização de software, pesquisas apontam, como passo fundamental, a sistematização do processo de análise e criação de componentes para um determinado domínio de aplicações [Werner e Braga 2000]. O Desenvolvimento Baseado em Componentes se preocupa com a criação de componentes que possam ser reutilizados em uma situação diferente daquela para qual foram originalmente construídos.

Componentes são unidades de software independentes que encapsulam seu projeto e implementação e oferecem serviços por meio de interfaces bem definidas para o meio externo. Por outro lado, Serviços Web são elementos computacionais autodescritivos e independentes de plataforma, que disponibilizam funcionalidades autocontida, visando o seu reuso e a interoperabilidade com outros sistemas [Papazoglou 2003].

Nesse contexto, um Serviço Web pode ser visto como um componente cujas funcionalidades são acessíveis por mensagens baseadas em *eXtensible Markup Language* (XML) [Yao e Etzkorn 2004, Ha e Lee 2006]. O arquivo de descrição do serviço, geralmente também baseado em XML, possui as informações necessárias para que outros componentes possam interagir com este serviço, incluindo o formato das mensagens para as chamadas os seus métodos, protocolos de comunicação e as formas de localização do serviço. Um dos maiores benefícios dessa descrição é a abstração dos detalhes de implementação do serviço, permitindo que seja acessado independente da plataforma de hardware ou software. A comunicação baseada em XML adiciona também flexibilidade com relação à linguagem de programação tanto na implementação, quanto no acesso ao Serviço Web. Estas características permitem e motivam a implementação de aplicações Web baseadas em Serviço Web por torná-las

fracamente acopladas com as outras partes do código da aplicação. Com isso, as aplicações adquirem uma arquitetura componentizada e flexível em relação às várias plataformas disponíveis no mercado.

Pela utilização de tecnologias da Web Semântica [Berners-Lee et al. 2001] é possível estender a capacidade dos Serviços Web tradicionais, criando os Serviços Web Semânticos. Esses serviços possuem descrições semânticas a fim de obter-se um maior poder de expressão na sua definição, na sua descoberta e no seu acesso [Hepp 2006]. Para realização de tarefas mais complexas uma composição de Serviços Web Semânticos pode usar agentes de software para escolher e combinar os serviços necessários. A execução de uma composição de serviços deve ser monitorada, a fim de detectar exceções e modificações dinâmicas que impeçam a concretização de uma tarefa a ser realizada. Isso permite que uma aplicação que dependa desses serviços se recupere, encontrando outros serviços ou criando uma nova composição que seja equivalente a anterior [Balzer et al. 2004].

Baseado em especificações providas por linguagens como a *Ontology Web Language for Services* (OWL-S) [W3Cb 2004], pode-se criar Agentes de Software, com comportamentos inteligentes e mobilidade para gerenciar o uso de Serviços Web Semânticos. Por sua vez, Agentes Móveis são Agentes de Software em execução num ambiente computacional, capazes de migrar de forma autônoma através dos elementos constituintes desse ambiente e compartilhar os seus recursos, a fim de realizar uma determinada tarefa [Dobson et al. 2006]. Tais agentes são particularmente usados em sistemas distribuídos, onde o poder computacional é descentralizado, pois contribuem para a interoperabilidade através dos elementos desses sistemas. Nesse trabalho, os Agentes Móveis foram escolhidos, uma vez que além de incorporar as funcionalidades dos agentes tradicionais, são capazes de migrar descobrindo, compondo e monitorando a execução dos serviços distribuídos pela Internet.

3.1. Framework para Serviços Web Semânticos, baseado em Agentes Móveis.

Buscando melhorar o processo de DBC e considerando as idéias apresentadas, foi desenvolvido um framework para apoiar a abordagem proposta. Esse framework, denominado FrameAgentesMóveis, combina Serviços Web Semânticos e Agentes Móveis, no desenvolvimento de aplicações a fim de facilitar sua implementação. No modelo da Figura 1 têm-se os principais componentes desse framework. O componente *Agent* é responsável pela: busca, composição e monitoramento da execução de Serviços Web Semânticos. Além disso, caso seja necessário, esse componente pode migrar atravésos nós da rede para encontrar os Serviços Web Semânticos necessários para realizar uma tarefa. As políticas que gerenciam a mobilidade são implementadas no componente *MobilityManager*, que deve ser reutilizado através de sua interface *IMobilityManager* , um dos pontos flexíveis do framework. Seu método *beforeMove* possibilita adicionar, além da necessidade de um Serviço Web Semântico remoto, situações em que o agente deve ou não migrar para realizar uma tarefa. Por exemplo, verificar se um recurso está disponível localmente ou remotamente. Já o método *beforeReceive* possibilita definir o comportamento no caso de receber agentes que migraram de outros nós da rede, podendo implementar os aspectos de segurança, desempenho e outros requisitos não funcionais.

O componente *Reasoner* é utilizado pelos agentes para efetuar inferências sobre as descrições do domínio e dos Serviços Web Semânticos. Os componentes *OntologyManager* e *ServicesRepository* são responsáveis pelo armazenamento e recuperação das descrições das ontologias em *Web Ontology Language* (OWL) [W3Ca 2004] e dos Serviços Web Semânticos em OWL-S, respectivamente.

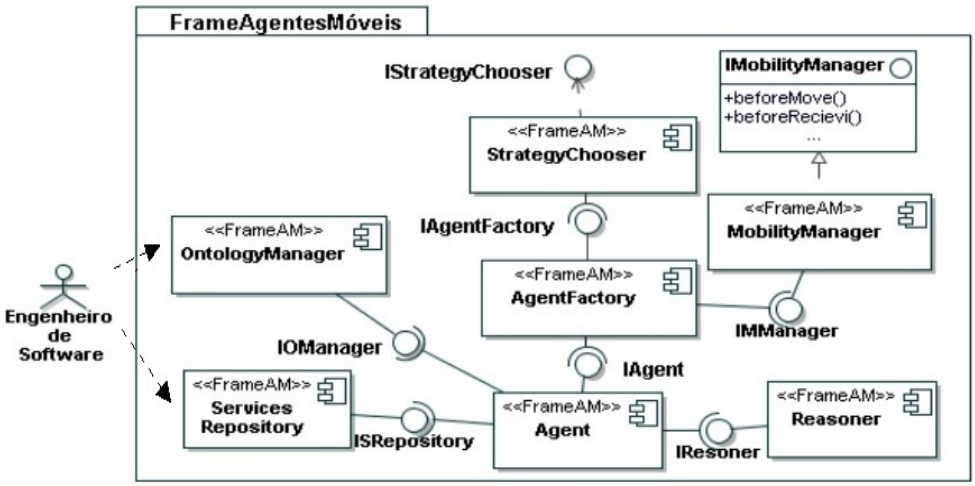


Figura 1. Modelo de Componentes do FrameAgentesMóveis.

O componente *StrategyChooser* foi construído com base no padrão *strategy* [Gamma et al. 1995] para suportar a escolha de diferentes estratégias na utilização dos Serviços Web Semânticos, conforme as necessidades de cada aplicação. Sua interface *IStrategyI* constitui outro ponto flexível do framework, e possibilita, por exemplo, que aplicação defina suas estratégias para adaptação de conteúdo, para comunicação em redes de sensores e integração em *WebLabs* [Coelho et al. 2007]. O componente *AgentFactory* foi construído segundo o padrão *factory* [Gamma et al. 1995], para suportar a criação de agentes conforme a estratégia escolhida no componente *StrategyChooser*.

A construção do FrameAgentesMóveis apóia a utilização da abordagem proposta neste artigo, servindo como um "esqueleto" que estrutura e organiza o desenvolvimento das aplicações, conforme se apresenta a seguir.

4. Abordagem Proposta

A abordagem proposta (Figura 2) é realizada nas quatro etapas do ciclo clássico de desenvolvimento de software (Análise, Projeto, Implementação e Testes). O processo de desenvolvimento é orientado por técnicas que atendem necessidades como a modificação não antecipada de componentes e a utilização de componentes distribuídos.

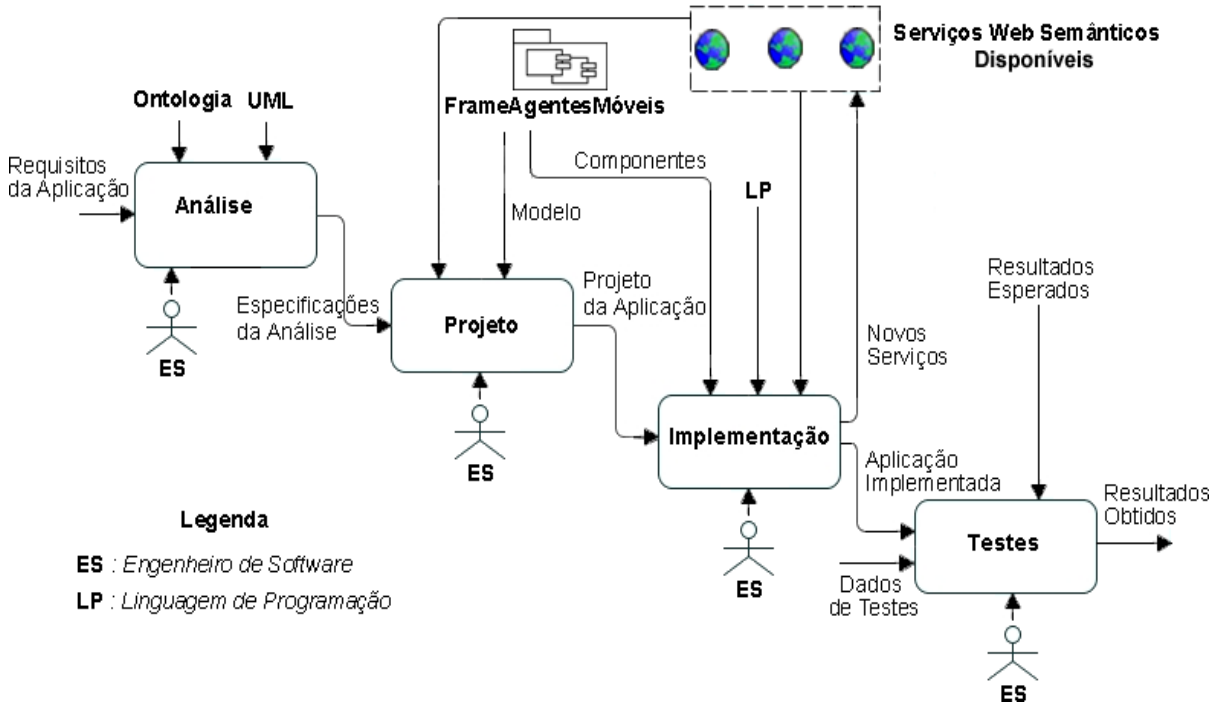


Figura 2. Abordagem proposta, segundo a SADT [Ross 1977].

Na etapa de **Análise** parte-se dos requisitos da aplicação para obter suas especificações em modelos de ontologias e outros modelos conhecidos da *Unified Modeling Language* (UML) [OMG 2004], como o de casos de uso e de seqüência. Os modelos de ontologias representam o conhecimento de maneira explicita, uma vez que estruturam os conceitos da aplicação num nível mais alto de abstração.

A partir das especificações da aplicação e da estrutura do FrameAgentesMóveis, na etapa de **Projeto**, faz-se a modelagem da aplicação considerando as restrições da plataforma de software adotada. Uma das atividades dessa etapa consiste em refinar os modelos de ontologias em modelos de classes. Além disso, são considerados os modelos especificados em UML e os modelos do framework para estruturar a utilização de Agentes Móveis e Serviços Web Semânticos. Nessa etapa também são projetados, os serviços que não estão disponíveis para reuso na Internet e os comportamentos relacionados com a mobilidade dos agentes.

Uma vez projetada a aplicação, faz-se a sua **Implementação**. São implementados os componentes específicos da aplicação e os pontos flexíveis do FrameAgentesMóveis (interfaces *IstrategyChooser* e *ImobilityManager*). São conectados também os Serviços Web Semânticos, conforme definido no projeto. Caso já estejam disponíveis na Internet, esses serviços podem ser reutilizados. Caso não existam, os novos serviços projetados, são implementados e disponibilizados para reuso. Em qualquer caso, o Engenheiro de Software adiciona as descrições semânticas desses serviços, em uma linguagem de marcação da Web Semântica, como a OWL-S utilizada nesse trabalho. A OWL-S possibilita a migração das descrições sintáticas em *Web Services Description Language* (WSDL) para descrições semânticas em OWL. Conforme ilustra a Figura 3, a OWL-S é organizada em três sub-ontologias:

ServiceProfile, que descreve as características, as capacidades do serviço e as transformações que esse serviço é capaz de realizar; **ServiceProcess**, que especifica o protocolo de interação com o serviço; e **ServiceGrounding**, que provê meios para que a comunicação com esse serviço seja feita através de mensagens *Simple Object Access Protocol* (SOAP).

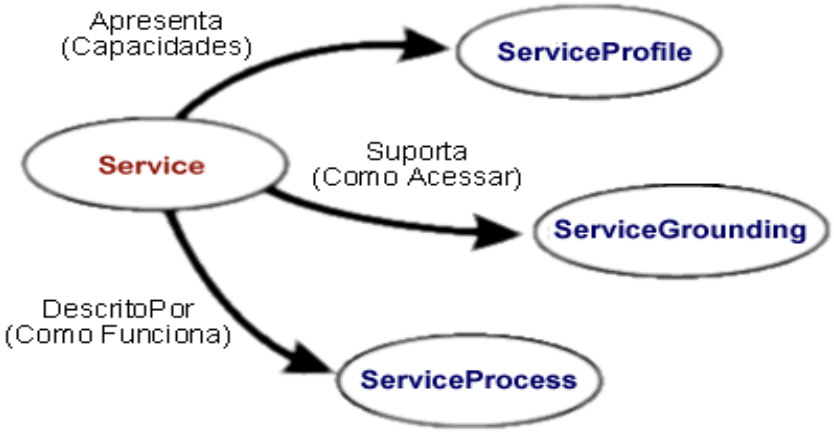


Figura 3. Visão Geral da OWL-S.

Finalmente, realizam-se T**estes** com dados que produzirão os resultados da execução. O Engenheiro de Software compara os resultados obtidos com os resultados esperados para verificar se atendem aos requisitos especificados. Caso não atenda retorna-se às etapas anteriores conforme o problema identificado, para as devidas correções.

5. Estudo de Caso

Para ilustrar o uso da abordagem proposta apresenta-se como estudo de caso uma aplicação no domínio da adaptação de conteúdo para computação ubíqua. A adaptação é realizada por um serviço de tradução de linguagens de marcação. Baseia-se em informações sobre o ambiente de uso de dispositivos móveis (e.g., preferências do usuário, rede de acesso, contrato com o provedor de serviços, características do dispositivo, conteúdo a ser adaptado). Essas informações são descritas em ontologias e denominadas perfis, a fim de que páginas da Web possam ser acessadas por esses dispositivos [Santana et al. 2007].

Para facilitar o entendimento dos requisitos dessa aplicação, as Figuras 4 e 5 fornecem uma visão geral da sua arquitetura. Na Figura 4 um Usuário acessa o serviço de adaptação de conteúdo através de um dispositivo móvel. Um Proxy de Adaptação tem o papel de interceptar requisições desse dispositivo para um Servidor de Conteúdo, que armazena conteúdo em seu formato original. Nesses Proxies estão localizados os Agentes Móveis, que utilizam Servidores de Adaptação disponíveis em Serviços Web Semânticos. Esses servidores realizam tradução de páginas *HyperText Markup Language* (HTML) para *Wireless Markup Language* (WML) e adaptações das imagens presentes nessas páginas.

Numa adaptação local, o fluxo de execução tem inicio após uma requisição de um usuário (1) ser interceptada por um Proxy de Adaptação. Em seguida, este Proxy de Adaptação requisita (2) e recebe (3), do Servidor de Conteúdo, o conteúdo a ser adaptado. De posse desse conteúdo, o Proxy de Adaptação delega ao seu agente a tarefa de realizar a adaptação necessária. Baseado-se na requisição e nos perfis, esse agente encontra, compõe e monitora a execução (4) dos Serviços Web Semânticos necessários. Finalmente, o Proxy de Adaptação recebe (5), armazena em seu cache, e envia os conteúdos adaptados ao Usuário (6).

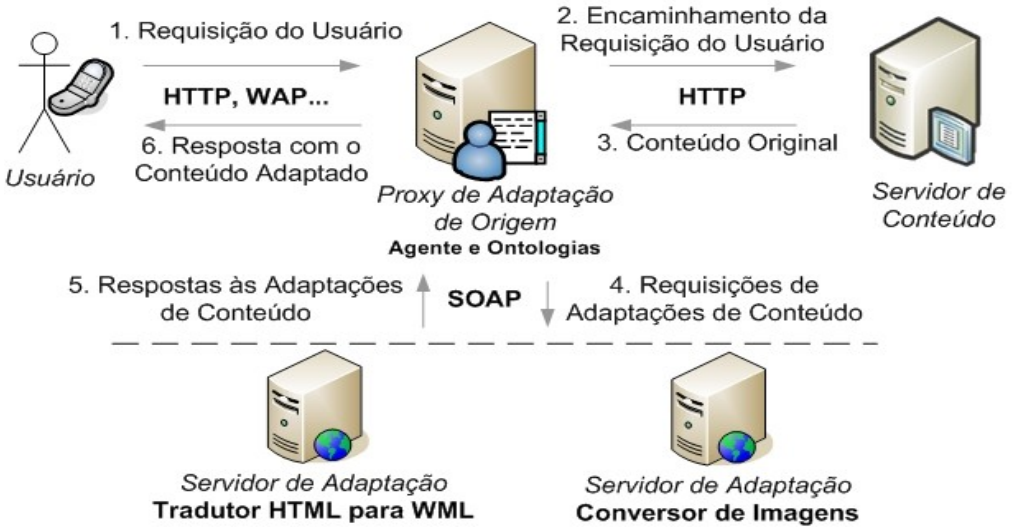


Figura 4. Adaptação Local de Conteúdo.

A Figura 5 considera o caso de uma adaptação remota, quando é necessário um Serviço Web Semântico que não está disponível no Proxy de Adaptação de Origem, nesse exemplo a adaptação de HTML para *Compact HTML* (cHTML). A requisição do usuário é interceptada por um Proxy de Adaptação (1), o agente do Proxy da Adaptação de Origem migra para um Proxy de Adaptação Remoto (2), requisita (3) e recebe (4) o conteúdo a ser adaptado. No Proxy de Adaptação Remoto o Agente utiliza o Servidor de Adaptação necessário (5) e (6), realiza a adaptação e retorna ao Proxy de Adaptação de Origem (7) para enviar o conteúdo adaptado ao Usuário (8).

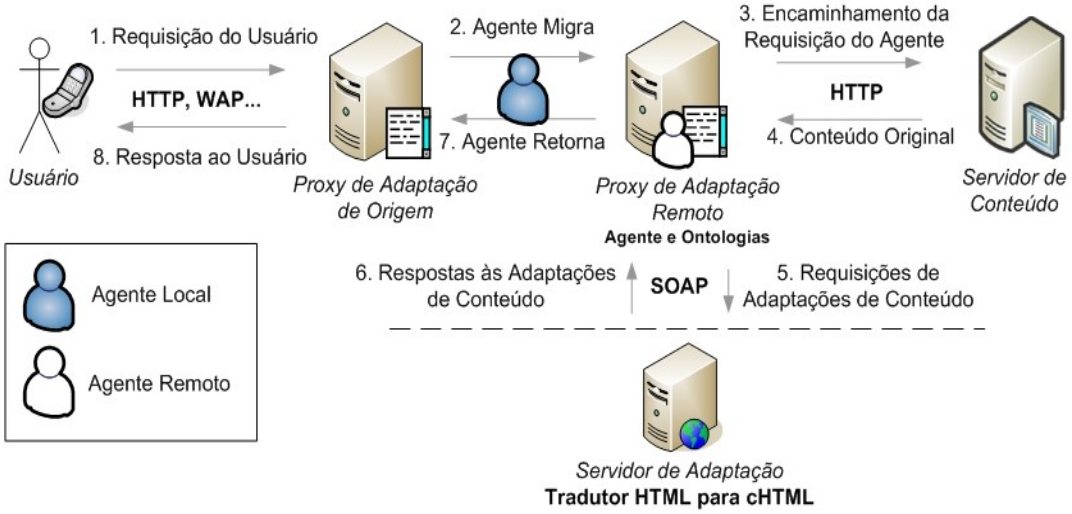


Figura 5. Adaptação Remota de Conteúdo.

5.1. Análise

Dentre os modelos de ontologias especificados tem-se o da Figura 6, que descreve o perfil dos conteúdos adaptados, sendo que *Browser_Accept*, *Others*, *Location*, e *Content_Info* são sub-classes de *Content*. Maiores informações sobre modelos de outros perfis (usuários, redes, dispositivos e contratos de serviços) estão disponíveis em [Forte et al. 2006]. Modelos da UML complementam as especificações nesta etapa, mas não são apresentados por serem bastante conhecidos da comunidade de Engenharia de Software.

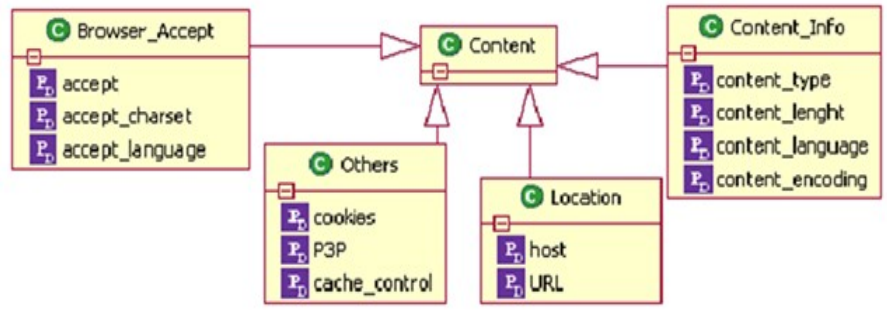


Figura 6. Modelo de ontologia para o perfil de conteúdo.

5.2. Projeto

A partir das Ontologias de domínio, dos modelos UML e dos modelos do FrameAgentesMóveis são obtidos os modelos de Projeto. A Figura 7 apresenta um desses modelos com os componentes reutilizados do framework e os componentes específicos da aplicação (sombreado). O componente *ContentAdaptation* conecta-se com o componente *StrategyChooser* através da interface *IStrategyChooser*, provendo uma estratégia que intercepta as requisições de usuários para adaptações de conteúdo. Esse componente conecta-se com os componentes *ContentTransferProtocol*, *ProfileManager* e *Cache* cujas responsabilidades são: possibilitar a transferência de conteúdos, facilitar a utilização dos perfis, e armazenar conteúdos adaptados para melhorar o desempenho geral da arquitetura.

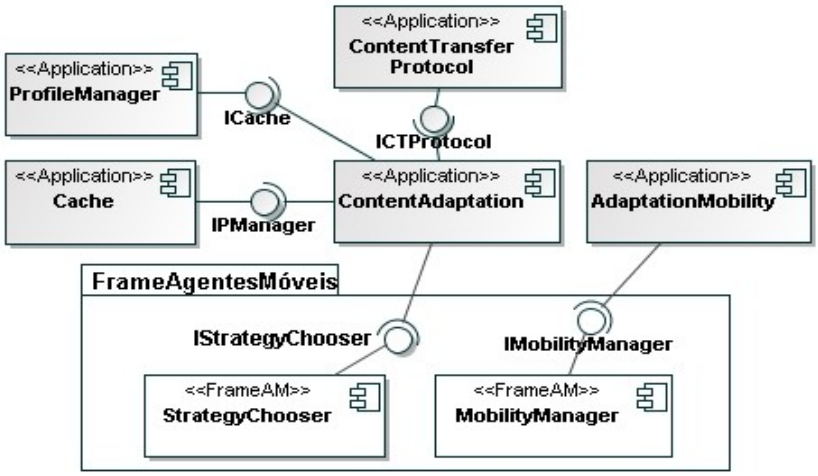


Figura 7. Reutilização do FrameAgentesMóveis.

O componente *AdaptationMobility* conecta-se com o componente *MobilityManager* para definir a mobilidade do agente. Como ilustrado na Figura 8, o fluxo de atividades inicia quando o Proxy de Adaptação de Origem delega uma adaptação de conteúdo ao seu agente. A primeira atividade desse agente é recuperar os perfis que serão utilizados na adaptação. Em seguida, esse agente busca os Serviços Web Semânticos que podem ser utilizados para essa adaptação. Caso nenhum deles seja capaz de realizar sozinho a adaptação, o agente requisita a elaboração um plano de composição de serviços e monitora a execução desse plano, assegurando seu sucesso. Este agente deverá migrar para um Proxy de Adaptação caso seu Proxy de Adaptação de Origem não possua os serviços de adaptação e os perfis necessários ou estiver sobrecarregado.

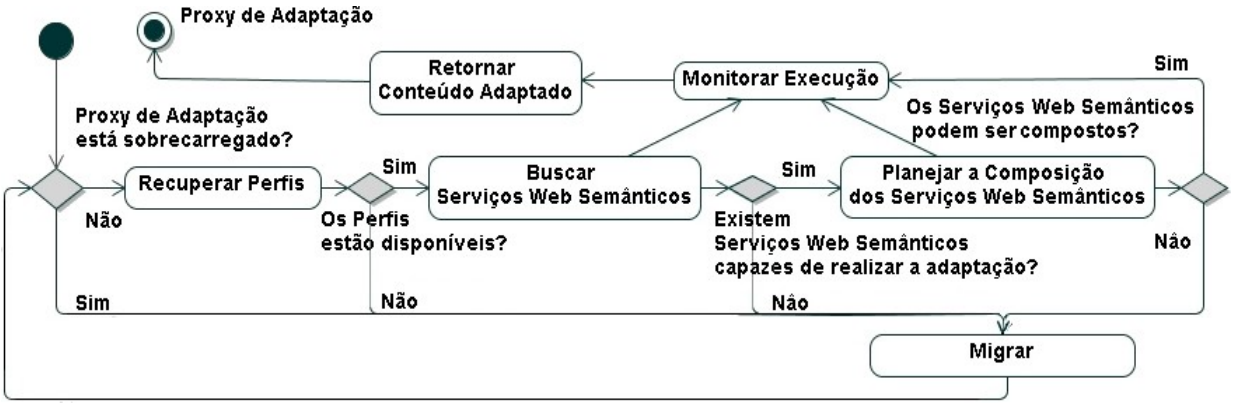


Figura 8. Modelo de Atividades para o Agente Móvel.

Nessa etapa também foram projetados três novos serviços: Tradutor HTML para WML, Conversor de Imagens e Tradutor HTML para cHTML.

5.3. Implementação e Testes

Prosseguindo com a abordagem faz-se a Implementação dos componentes da aplicação (*ProfileManager*, *Cache*, *ContentTransferProtocol*, *ContentAdaptation* e *AdaptationMobility*) e dos Serviços Web Semânticos (*Tradutor HTML para WML*, *Conversor de Imagens* e *Tradutor HTML para cHTML*) previamente projetados. Nessa

etapa foram também adicionadas as descrições OWL-S dos Serviços Web Semânticos, contendo informações, como as entradas, as saídas, e a url de acesso. Por exemplo, na Figura 9 tem-se a descrição do serviço de tradução de HTML para WML.

```
<!-- Service description -->
<service:Service rdf:ID="HTML2WMLService">
      <service:presents rdf:resource="#HTML2WMLProfile"/>
      <service:describedBy rdf:resource="#HTML2WMLProcess"/>
      <service:supports rdf:resource="#HTML2WMLGrounding"/>
</service:Service>

<!-- ServiceProfile description -->
<profile:HTML2WMLService rdf:ID="HTML2WMLrProfile">
      <service:presentedBy rdf:resource="#HTML2WMLService"/>
      <profile:serviceName>HTML2WML</profile:serviceName>
      <profile:hasInput rdf:resource="#htmlpage"/>
      <profile:hasOutput rdf:resource="#wmlpage"/>
</profile:HTML2WMLService >

<!-- ServiceProcess description -->
<process:AtomicProcess rdf:ID="HTML2WMLProcess">...

<!-- ServiceGrounding description -->
<grounding:WsdlGrounding rdf:ID="HTML2WMLGrounding">
 <service:supportedBy rdf:resource="#HTML2WMLService"/>
</grounding:WsdlGrounding>
<grounding:WsdlAtomicProcessGroundingr df:ID="HTML2WMLProcessGrounding">
      <grounding:owlsProcess rdf:resource="HTML2WMLProcess"/>
      <grounding:wsdlDocument>
            http://localhost/HTML2WML/HTML2WMLService?wsdl
      </grounding:wsdlDocument>
```

Figura 9. Descrição de um Serviço Web Semântico.

Finalmente, são realizados os testes da aplicação, conforme ilustra o resultado de uma adaptação de conteúdo apresentado na Figura 10. No caso, uma página da Web, que antes não podia ser apresentada em dispositivos móveis, é adaptada de acordo com as características desse dispositivo.

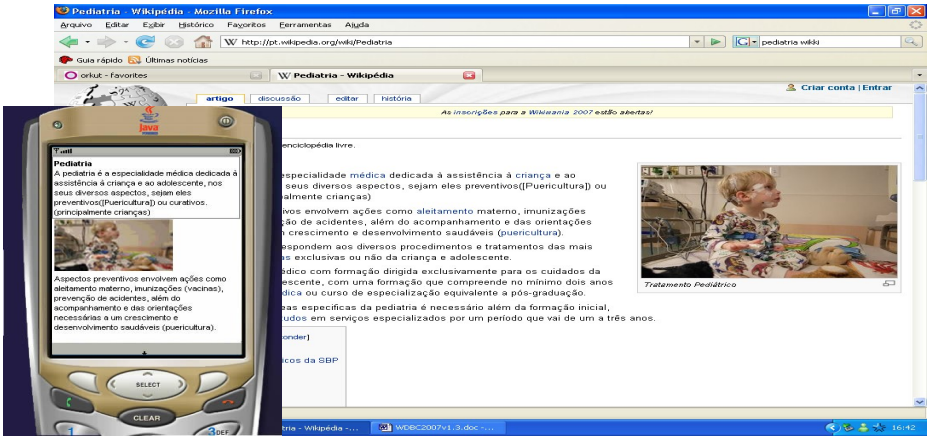


Figura 10. Conteúdo Adaptado.

Concluindo, verificou-se que com as ontologias obteve-se modelos mais genéricos para os componentes da aplicação. Outro ganho foi na implementação onde o tempo foi reduzido devido ao reuso do framework. Além disso, os novos serviços construídos poderão ser reutilizados em outras aplicações do domínio.

6. Trabalhos Correlatos

Existem diversos trabalhos que empregam ontologias, serviços Web e agentes de software para apoiar o Processo de Desenvolvimento de Software. Alguns têm foco apenas nas ontologias como o de [Linhalis e Moreira 2006], que utiliza linguagem natural para descrever os requisitos dos componentes de software, que posteriormente são transformadas em ontologias. A abordagem proposta utiliza os modelos de ontologias complementados por modelos em UML.

Em [Elias et al. 2006] é proposto um repositório de componentes que utiliza descrições sintáticas baseadas em XML para representação de metadados. A abordagem proposta utiliza a linguagem OWL-S que tem semântica para descrever os componentes Serviços Web Semânticos.

O trabalho de [Ha e Lee 2006] usa Serviços Web Semânticos no DBC num ambiente específico de *e-business*. A abordagem proposta é mais genérica e pode ser empregada em outros domínios de aplicações.

Dentre os trabalhos que utilizam Agentes de Software para busca, composição e monitoramento de Serviços Web Semânticos destaca-se o de [Charif-Djebbar e Sabouret 2006]. Este trabalho propõe uma abordagem baseada em sistemas multiagentes, na qual os agentes possuem um protocolo de interação que permite selecionar e compor os serviços. Na abordagem proposta os agentes podem migrar através dos diferentes nós da rede, para utilizar os Serviços Web Semânticos que estão distribuídos.

7. Conclusões e Trabalhos Futuros

Este artigo apresentou uma abordagem para DBC, que emprega Ontologias para descrever domínios e Agentes Móveis para gerenciar Serviços Web Semânticos. Esses agentes migram através dos nós da rede para encontrar os componentes necessários às aplicações, em função dos requisitos do domínio descrito em ontologias e de técnicas da UML. A abordagem dispõe de um framework "caixa-cinza", que possibilita o reuso organizado de componentes e a estruturação das aplicações de acordo com a abordagem proposta. Empregando-se o framework, grande parte da implementação é simplificada.

O estudo de caso ressalta as vantagens do emprego da abordagem proposta, tais como a modificação não antecipada de componentes e o uso de componentes distribuídos. A modificação não antecipada é atendida via a utilização de Serviços Web Semânticos, já que um serviço pode ter seu comportamento alterado sem que seja necessário reconstruir a aplicação. O uso de componentes distribuídos é obtido via a combinação de Serviços Web Semânticos com Agentes Móveis, os quais podem migrar através dos nós da rede para encontrar, compor e monitorar a execução desses serviços.

Dentre os trabalhos futuros destacam-se: (i) refinar a abordagem com base em estudos de casos de outros domínios, como laboratórios virtuais (*WebLabs*) e redes de sensores; (ii) criar ferramentas que auxiliem o Engenheiro de Software nas diferentes

atividades de cada etapa da abordagem; (iii) avaliar o desempenho de aplicações que reutilizam o framework com Serviços Web Semânticos executados por agentes.

Referências

Arango, G. (1989) "Domain analysis – from art to engineering discipline". ACM Sigsoft Software Engineering Notes, vol. 14, no. 3, pp. 152-159.

Balzer, S., Liebig, T. e Wagner, M. (2004) "Pitfalls of OWLS – A Practical Semantic Web Use Case", Anais da International Conference on Service Oriented Computing, pp. 289-298.

Berners-Lee, T., Hendler, J., Lassila, O. (2001) "The Semantic Web" Scientific American, vol. 5, no. 17, pp. 35-43.

Charif-Djebbar, Y. e Sabouret, N. (2006) "Dynamic Service Composition and Selection through an Agent Interaction Protocol". Anais da International Conference on Web Intelligence and Intelligent Agent Technology, pp. 105-108.

Coelho et al. (2007) "Arquitetura e Requisitos de Rede para Web Labs" Anais do Simpósio de Redes de Computadores e Sistemas Distribuídos 2007, pp. 499 – 512.

Ebraert, P., Vandewoude, Y., D'Hondt, T. e Berbers, Y. (2005). "Pitfalls in Unanticipated Dynamic Software Evolution". Anais do Workshop on Reflection, AOP and Meta-Data for Software Evolution, pp. 41-51.

Elias, G., Schuenck, M. Negócio, Y., Dias, J. e Filho, S.M. (2006) "X-ARM: an asset representation model for component repository systems" Anais do Symposium on Applied Computing, pp. 1690 - 1694.

Evermann, J. e Wand, Y. (2005) "Ontology based object-oriented domain modelling: fundamental concepts" Requirements Engineering, vol. 10, no. 5, pp. 146 – 160.

Forte, M., Souza, W.L., e Prado, A.F. (2006) "Utilizando ontologias e serviços web na computação ubíqua". Anais do Simpósio Brasileiro de Engenharia de Software, pp. 287-302.

Dobson, S. et al. (2006) "A Survey of Autonomic Communications" ACM Transactions on Autonomous and Adaptive Systems, vol. 1, no. 2, pp. 223-259.

Gamma E., Helm R., Johnson R. e Vlissides J. (1995) "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley.

Gruber, T.R. (1993) "A Translation Approach to Portable Ontology Specifications". Knowledge Acquisition, vol.5, no. 2, pp. 199-220.

Ha, Y. e Lee, R. (2006) **"**Integration of Semantic Web Service and Component-Based Development for e-business environment" Anais da International Conference on Software Engineering Research, Management and Applications, pp. 315 – 323.

Hepp, M. (2006) "Semantic Web and Semantic Web Services: Father and Son or Indivisible Twins?" IEEE Internet Computing, vol. 10, no 2, pp. 85–88.

Lee,W. (2007) "Deploying personalized mobile services in an agent-based environment**",** Expert Systems with Applications, vol. 32, no. 4, pp. 1194-1207.

Linhalis, F. e Moreira, D.A. (2006) "Ontology-Based Application Server to the Execution of Imperative Natural Language Requests". Anais do International Conference on Flexible Query Answering Systems", p. 589-600.

Lucrédio, D., Almeida, E.S. e Prado, A.F. (2004) "A Survey on Software Components Search and Retrieval".Anais da Euromicro Conference, pp. 152-159.

OMG. Unified Modeling Language (UML) Specification, Versão 2.1.1, Object Management Group, 2004.

Papazoglou, M.P.(2003) "Service-oriented computing: concepts, characteristics and directions"Anais da Conference on Web Information Systems Engineering, pp. 3–12.

Ross, D. (1977) "Structured Analysis (AS): A Language for Communicating Ideas". IEEE Transactions on Software Engineering, vol. 3, no.1, pp. 16-34.

Santana, L.H.Z. et al. (2007) "Serviço de tradução de linguagens de marcação para a Internet". Anais do XXV Simpósio Brasileiro de Redes de Computadores, vol. 1, pp. 541- 554.

W3C (2004a) OWL Web Ontology Language < http://www.w3.org/TR/owl-features/>

W3C (2004b) OWL-S OWL-S: Semantic Markup for Web Services < http://www.w3.org/Submission/OWL-S/>

Werner, C.M.L. e Braga, R. M.M. (2000) "Desenvolvimento Baseado em Componentes".Anais do Simpósio Brasileiro de Engenharia de Software, pp. 297-329.

Yao, H. e Etzkorn, L. (2004) "Towards A Semantic-based Approach for Software Reusable Component Classification and Retrieval" Anais do 42nd annual Southeast regional conference ACM-SE 42, pp. 110 - 115.

CrossMDA: Arcabouço para integração de interesses transversais no desenvolvimento orientado a modelos

[1]Marcelo Pitanga Alves, [2]Paulo F. Pires, [2]Flávia C. Delicato, [1]Maria Luiza M. Campos

[1]Departamento de Ciência da Computação (DCC/IM) - Núcleo de Computação Eletrônica (NCE) - Universidade Federal do Rio de Janeiro (UFRJ) - Bloco C – Cidade Universitária – Ilha do Fundão – Rio de Janeiro, RJ – Brasil.

[2]DIMAp: Departamento de Informática e Matemática Aplicada - Universidade Federal do Rio Grande do Norte - Natal, RN - Brasil

mpitanga@gmail.com,{paulo.pires,flavia.delicato}@dimap.ufrn.br, mluiza@ufrj.br

Abstract. CrossMDA is a framework that encompasses a transformation process to integrate crosscutting concerns in model-oriented systems. Such integration is accomplished by combining the capacities of separation of concerns currently existent in MDA and AOP approaches. CrossMDA uses the concepts of horizontal separation of concerns from AOP to create independent business and aspect models, integrating those models through MDA transformations (vertical separation of concerns). CrossMDA comprises a development process, a set of services and support tools.

Resumo. CrossMDA é um arcabouço que incorpora um processo de transformação para integração de interesses transversais em sistemas orientados a modelo. Essa integração é feita combinando as capacidades de separação de interesses existentes nas abordagens MDA e Programação Orientada a Aspectos (POA). CrossMDA usa o conceito de separação horizontal de interesses da POA pra criar modelos de negócio e aspectos independentes, integrando-os através de transformações MDA (separação vertical de interesses). CrossMDA provê um processo de desenvolvimento e um conjunto de serviços e ferramental de apoio para dar suporte ao processo.

1. Introdução

A crescente complexidade dos sistemas de software atuais, aliada ao constante advento de novas tecnologias e a demanda sempre maior dos usuários finais por qualidade nos sistemas fornecidos, fazem com que as aplicações tenham que incorporar e lidar com um conjunto cada vez maior de requisitos de *software*. Dentre esses requisitos, os requisitos computacionais, como, por exemplo, a necessidade de concorrência, distribuição, persistência e recuperação de falhas afetam um grande número de componentes de um sistema, ou seja, cruzam as fronteiras de tais componentes, sendo responsáveis pelo espalhamento (*scattered*) e entrelaçamento (*tangled*) de funcionalidades, e conseqüentemente do código que as implementam. O espalhamento e o entrelaçamento, por sua vez, dificultam o entendimento, a manutenção e a evolução do código [Tekinerdogan et al. 2004]. Requisitos que não podem ser encapsulados em um único componente e tipicamente ficam espalhados por diversas partes do sistema,

tais como por exemplo, requisitos de monitoramento e sincronismo de código, são conhecidos como interesses transversais.

[Kiczales et al. 1997] apresentaram a abordagem de Programação Orientada a Aspectos (POA) que complementa a Programação Orientada a Objetos (POO) ao oferecer um conjunto de técnicas que permite o apropriado encapsulamento de interesses transversais através de uma nova abstração chamada aspecto, além de fornecer um mecanismo de composição (*weaving*) e reuso do código do aspecto.

A Arquitetura Orientada a Modelos (Model Driven Architecture - MDA) [OMG 2006a] é uma iniciativa do OMG para o desenvolvimento orientado a modelos que propõe três diferentes níveis de abstrações para a modelagem: Modelo Independente de Computação (Computational Independent Model-CIM), Modelo Independente de Plataforma (Platform Independent Model-PIM) e Modelo Dependente de Plataforma (Platform Specific Model-PSM). Os modelos são mapeados de uma abstração para outra através de processos de transformações sucessivas, durante as quais são incluídos novos elementos no modelo, reduzindo sua abstração até o nível de dependência da plataforma computacional aonde o sistema será implementado. A proposta MDA naturalmente provê uma forma de separação vertical de interesses, já que cada modelo contempla elementos específicos daquele nível de abstração, por exemplo, requisitos computacionais são incluídos apenas no modelo PSM. Porém, a separação de interesses segundo a dimensão horizontal não é tratada pela abordagem MDA, ou seja, não há mecanismos para identificar e encapsular interesses transversais dentro de cada modelo.

A separação horizontal de interesses está sendo atualmente abordada na área de Modelagem Orientada a Aspectos (MOA) [AOM 2006] onde trabalhos concentram-se em técnicas para identificação, análise, gerenciamento e representação de interesses transversais na fase de modelagem, usando extensões da UML [Suzuki e Yamamotto 1999, Stein 2002, Stein et al. 2002, Aldawud et al. 2003, Baniassad e Clarke 2004, Chavez 2004]. Porém, a falta de ferramentas adequadas para a modelagem e gerência do relacionamento de elementos do negócio com um determinado interesse transversal (processo de *weaving*) tem sido uma barreira na adoção desses modelos em ambientes MDA. Essa lacuna já vem sendo alvo de investigações que combinam os conceitos de POA com MDA propondo a integração de interesses transversais em modelos através do mecanismo de transformação MDA [Wampler 2003, Chaves e Zancanella 2004, Reina e Torres 2005, Solberg et al. 2005, Simmonds et al. 2005, Graziadei 2005]. Porém, ainda restam questões em aberto quanto à combinação das abordagens POA e MDA principalmente com relação à gerência do processo de combinação (*weaving*) e ao reuso de artefatos de transformação derivados desse processo.

A motivação do presente trabalho é propor uma solução que integre as abordagens POA e MDA e trate essas questões. Com esse intuito, propomos um arcabouço, denominado CrossMDA, o qual contempla um processo de transformação e provê um conjunto de serviços e ferramentas de apoio que implementam esse processo. O CrossMDA permite: i) elevar o nível de abstração na modelagem orientada a aspectos através do uso de modelos PIM de interesses transversais independentes do modelo de negócio; ii) reusar artefatos de interesses transversais no nível de modelos PIM; iii) automatizar o mapeamento do relacionamento de interesses transversais com elementos do modelo de negócio através do processo de transformação da MDA; iv) facilitar o reuso de artefatos de transformação MDA relacionados a interesses transversais; e v) favorecer o reuso de modelos PIM de negócios.

CrossMDA permite o tratamento de aspectos no nível de modelagem e fornece mecanismos que possibilitam a separação de interesses na dimensão horizontal, entre modelos de um mesmo nível, bem como a separação na dimensão vertical, entre modelos de diferentes níveis. A dimensão horizontal permite a modelagem de interesses transversais independentemente dos elementos de negócio. Para tanto, a CrossMDA sugere um processo que utiliza modelos de aspectos no nível de PIM. Tal modelo de aspectos é uma representação abstrata de um determinado interesse transversal, que permite esconder os detalhes de implementação do aspecto para o projetista de negócio, elevando assim o nível da modelagem no PIM.

Quanto à dimensão vertical, ela é endereçada através da implementação de um processo de transformação para gerar modelos das instâncias dos aspectos. Como os modelos de aspectos e de negócio são independentes, é oferecido ao projetista um processo para guiar e documentar os relacionamentos entre o aspecto e o elemento de negócio, a serem utilizados na composição do novo modelo. Através do uso de uma linguagem formal de transformação baseada no padrão MOF-QVT (*Query, View, Transformation*) [OMG 2006c], CrossMDA oferece, ao final de seu processo de composição de modelos (*model weaving*), a geração automática de um programa de transformação, o qual corresponde à implementação da especificação formal do processo de composição de modelos. Esse processo permite a edição, pelo projetista, de composições de modelos já existentes sem perda dos mapeamentos entre os aspectos e elementos de negócio.

Este artigo está estruturado em 4 seções. Na seção 2, é apresentado o arcabouço CrossMDA, seu funcionamento e sua implementação através de um exemplo ilustrativo. Na seção 3 são apresentados os trabalhos relacionados. Finalmente, na seção 4, são apresentadas a conclusão e considerações finais do trabalho.

2. Processo CrossMDA

O processo CrossMDA, apresentado no Diagrama de Processos da Figura 1, é composto de atividades, as quais são por sua vez organizadas em 3 fases: Fase 1 - seleção de fontes, Fase 2 - mapeamento e Fase 3 - composição do modelo.

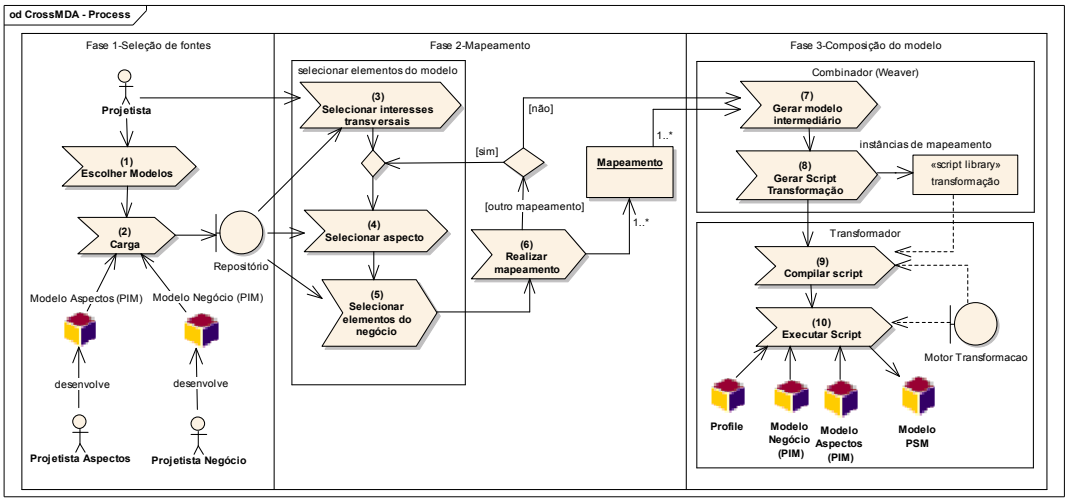


Figura 1. Processo CrossMDA.

A fase 1 engloba as atividades (1) e (2) (ver Figura 1). A atividade (1) consiste em realizar a escolha dos modelos PIM fontes a serem utilizados durante o processo de transformação e a atividade (2) é responsável pela carga e persistência dos modelos no repositório de metadados. Os modelos fontes são de dois tipos: modelo de aspectos e modelo de negócio. O modelo de aspectos consiste na representação abstrata, isto é, independente de plataforma, de interesses transversais, modelados como classes com o estereótipo <<*aspect*>> [Stein 2002] e organizados em pacotes. No CrossMDA, um pacote de aspectos é uma entidade que agrupa aspectos relacionados, ou seja, que dizem respeito a uma mesma categoria de requisito. Por exemplo, um pacote pode conter vários aspectos relacionados a autenticação, outro a *logging*, etc. O modelo de negócio, por sua vez, é composto pelas entidades, classes de serviços, relacionamentos, restrições, diagramas de classes e interação e demais elementos que representam toda a modelagem do processo de negócio.

A fase 2 é responsável por mapear os tipos de relacionamentos entre os aspectos e os elementos do modelo de negócio. Essa fase inicia-se com a atividade (3) que permite ao projetista selecionar os pacotes de interesses transversais que são relevantes ao domínio da aplicação. Em seguida, é iniciado o processo repetitivo de definição de relacionamento que engloba as atividades (4), (5) e (6). A atividade (4) é responsável pela seleção dos elementos aspectuais; a atividade (5) é responsável pela seleção dos elementos do negócio e; a atividade (6) realiza o mapeamento final do relacionamento, armazenando os elementos selecionados nas atividades (4) e (5) juntamente com o tipo designador do **ponto de junção** (*join point*) e os tipos de **adendo** (*advice*) selecionados no modelo de mapeamento.

A fase 3 é a responsável por realizar a composição do novo modelo, incluindo todos os elementos do modelo de negócio existentes e os novos elementos que representam as instâncias dos aspectos mapeados em um nível já dependente de plataforma computacional (PSM). É composta por 4 atividades que representam a **combinação de modelos** (*weaving*) e a **transformação**. A fase é iniciada com as atividades (7) e (8) do **combinador** (*weaver*). A atividade (7) é responsável por gerar um modelo intermediário a partir dos relacionamentos mapeados da fase 2. O modelo intermediário é uma representação que contém a hierarquia de composição de uma instância de uma classe aspecto e a sua dependência com o elemento de negócio ao qual se relaciona. Em seguida, a atividade (8) é iniciada, cuja responsabilidade é transformar o modelo intermediário em uma especificação formal através da geração de um programa de transformação baseada na especificação MOF QVT da OMG [OMG 2006c]. As atividades (9) e (10) representam as funções do transformador de modelos, e consistem respectivamente em compilar e executar o programa de transformação gerado pelo combinador de modelos.

2.1. Serviços do CrossMDA

Esta seção descreve os principais serviços providos por CrossMDA e que formam a base de trabalho para permitir a realização das atividades componentes do processo oferecido pelo arcabouço. Os serviços são: (i) persistência de modelos; (ii) mapeamento de elementos; (iii) combinador e; (iv) transformador de modelos. Para mostrar o funcionamento e a implementação desses serviços utilizamos um exemplo ilustrativo de um sistema de vendas ao qual deve ser aplicado um *interesse* de autenticação para controlar o acesso a qualquer informação do cliente.

Conforme descrito anteriormente, na *fase 1* o projetista seleciona os modelos PIM fontes que serão utilizados durante todas as fases do processo CrossMDA. Na Figura 2 é apresentado um modelo simplificado de aspectos, organizados em pacotes, utilizado como uma das fontes de entrada para o processo, no presente exemplo ilustrativo.

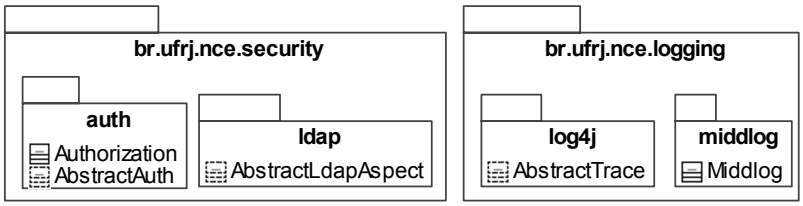


Figura 2. Modelo PIM de aspectos organizados em pacotes de interesses transversais.

Nesse exemplo de modelo, o pacote denominado *br.ufrj.nce.security.auth* contém os aspectos relacionados a autenticação. A Figura 3 ilustra uma classe representando um aspecto abstrato de autenticação. CrossMDA provê suporte para a representação de aspectos abstratos e não abstratos, onde as diferenças típicas entre eles são listadas a seguir. Aspectos abstratos podem possuir **pontos de atuação (***pointcut*) definidos como abstratos e estar ou não associados a um adendo. Um ponto de atuação abstrato não tem conhecimento dos pontos de junção que serão afetados e é um tipo de construção utilizada na POA para a criação de aspectos reutilizáveis. Nesse caso, na realização da classe do aspecto só é necessário definir os designadores de ponto de atuação (*pointcut designator* - PCD) e os pontos de junção. No caso de pontos de atuação abstratos não associados a um adendo, estes são combinados com outros pontos de atuação. Ainda, um adendo pode ser definido sobre um ponto de atuação abstrato e com isso pode-se implementar um comportamento transversal no aspecto abstrato. Aspectos abstratos são a base de desenvolvimento que permitem o seu reuso em diferentes cenários. Já os aspectos não abstratos, por sua vez, podem ter pontos de atuação não associados a um adendo, devendo-se também indicar o tipo do adendo (*after, before, around*) ao se configurar o ponto de atuação.

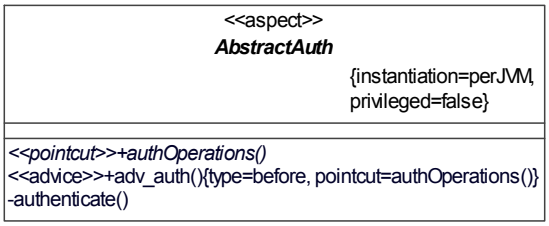


Figura 3. Classe representando um aspecto abstrato de autenticação para o modelo PIM (adaptado da proposta de [Stein 2002]).

O modelo de aspectos pode ser desenvolvido pelo próprio projetista do sistema ou pode-se utilizar algum modelo previamente desenvolvido para tal finalidade, seguindo as características de modelagem apresentadas na seção anterior. É importante notar que esse modelo representa aspectos independentes de plataforma. Ainda, a organização em pacotes torna-se um facilitador no momento da escolha de que tipo de interesse transversal será utilizado, restringindo a quantidade de aspectos a serem apresentados ao projetista durante o processo de mapeamento dos relacionamentos. O outro modelo fonte de entrada é o modelo de negócio que, para o exemplo utilizado, é apresentado na Figura 4.

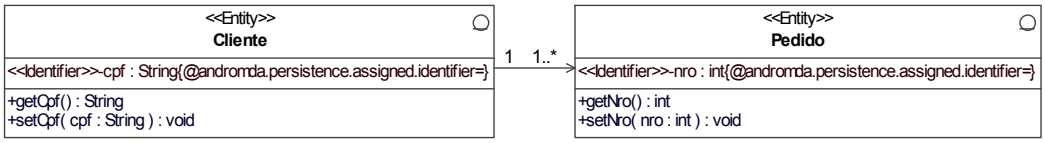


Figura 4. Fragmento do modelo PIM de classes do sistema de vendas utilizando perfil UML da ferramenta AndroMDA [AndroMDA 2006].

Os modelos selecionados são então carregados para o repositório de metadados que é o responsável pela persistência de todos os elementos que compõem os modelos.

2.1.1 Serviço de Persistência de Modelos

Este serviço é o responsável por implementar as operações básicas para permitir a carga e persistência dos modelos e as operações para navegar, recuperar e instanciar novos elementos em modelos existentes. A realização desta tarefa é feita por um serviço de repositório para persistência de metadados. Na implementação do CrossMDA, o repositório escolhido para gerenciar e persistir os elementos do modelo é o *NetBeans Metadata Repository* [NetBeans-MDR 2007]. Tal escolha deve-se principalmente ao fato desse repositório ser uma implementação popular e aberta do padrão OMG MOF (*Meta Object Facility*)[OMG 2006b]. Como o repositório é uma implementação externa ao ambiente de CrossMDA, o arcabouço provê um serviço (Figura 5) responsável pela interação com o repositório.

```
public interface IRepository  {
        public org.omg.uml.UmlPackage getUmlPackage();
        public Object getRepository();
        public void loadModel(String[] fileXmi, String searchRef) throws Exception;
        ....
```

Figura 5. Interface para o serviço de manipulação do repositório.

2.1.2. Serviço de Mapeamento de Elementos

Este serviço provê mecanismos para gerenciar o mapeamento dos relacionamentos entre os aspectos e os elementos de negócio, que é uma atividade chave do processo de CrossMDA. Os mapeamentos suportados em CrossMDA são de dois tipos: (i) pontos de atuação e (ii) intertipos. Porém, nesse artigo trataremos especificamente dos mapeamentos de pontos de atuação. Os mapeamentos de pontos de atuação seguem o padrão de especificação de pontos de atuação da abordagem POA e da linguagem AspectJ [AspectJ 2006, Laddad 2003], devido a essa especificação ser utilizada também por outras linguagens e arcabouços orientados a aspectos (Figura 6).

```
[visibilidade] [abstract] palavra-chave nome([args]) : tipo-pointcut ( assinatura-join point )
```

Figura 6. Definição de um ponto de atuação.

A especificação de um ponto de atuação é realizada através do uso de um tipo de ponto de atuação primitivo e da assinatura de um ponto de junção. Um tipo de ponto de atuação primitivo ou PCD, provê uma definição ao redor dos pontos de junção, por exemplo: um PCD do tipo chamada (*call*) corresponde a uma chamada para um método ou para um construtor. Existem vários PCDs disponíveis [Laddad 2003, 74pp] que são suportados no processo de mapeamento do CrossMDA. Os PCDs também podem ser

combinados através do uso de operadores lógicos, o que permite gerar especificações mais complexas para um ponto de atuação.

Com o objetivo de facilitar o mapeamento entre os elementos de negócio e os interesses transversais selecionados, o CrossMDA oferece para o projetista um processo e um serviço para armazenar os elementos do mapeamento. O processo é composto dos seguintes passos: (i) selecionar o aspecto e, se disponível, uma especificação de ponto de atuação pré-definido, podendo esse ponto de atuação ser ou não abstrato; (ii) selecionar uma ou mais entidades de negócio ou selecionar um ou mais métodos das entidades de negócio; (iii) indicar o tipo do ponto de atuação e; (iv) indicar o tipo do adendo (*before*, *after* ou *around*) [Laddad 2003, 81pp]. No exemplo do sistema de vendas é estabelecida uma regra de negócio em que toda a informação do cliente deve ser controlada via uma autenticação prévia. Para ilustrar os passos acima vamos aplicar um interesse de autenticação para controlar o acesso ao atributo *Cpf* da classe *Cliente*, conforme apresentado na Tabela 1.

Tabela 1: Passos e as informações selecionadas pelo projetista para o mapeamento

Passo	Informação selecionada		
	Tipo		Proprietário
(i)	ponto de atuação abstrato	método authOperations	classe AbstractAuth
(ii)	ponto de junção	método setCpf	classe Cliente
(iii)	tipo do PCD	chamada (*call*)	-
(iv)	tipo do adendo	-	-

Esse interesse é representado no modelo de aspectos por uma classe chamada *AbstractAuth* que especifica um método ponto de atuação abstrato chamado *authOperations* associado a um método adendo do tipo *before* chamado *adv_auth*. Este adendo implementa a programação necessária para efetuar a autenticação antes da chamada ou execução do ponto de junção. A Figura 3 representa a especificação UML desse interesse. Após a execução desses passos, tem-se então, a realização de um mapeamento (Figura 7). Tal mapeamento é persistido através do serviço de mapeamento seguindo um modelo específico do CrossMDA (Figura 8). Neste modelo, os relacionamentos entre os elementos *Aspect* e *Pointcut* possuem cardinalidade (1..n), permitindo assim que um mesmo aspecto possa relacionar vários pontos de atuação com tipos de adendos e tipos de PCD diferentes, da mesma forma que um mesmo ponto de junção pode ser referenciado por vários pontos de atuação com diferentes tipos PCD e, conseqüentemente, por vários aspectos. Cada elemento do modelo representa uma parte da especificação do ponto de atuação e também auxilia no mapeamento final de como o ponto de atuação será aplicado por um adendo do aspecto.

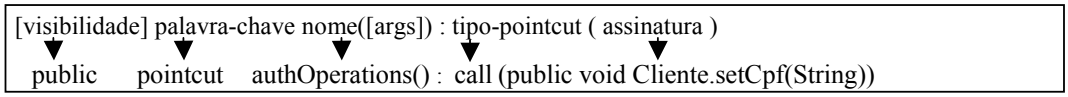


Figura 7. Definição de um ponto de atuação para controlar o acesso ao atributo Cpf da classe Cliente.

Conforme apresentado anteriormente, o serviço de mapeamento prevê ainda através do seu modelo (Figura 7) a capacidade de armazenar mapeamentos de declarações intertipos (*Intertype declaration*). Intertipos permite a declaração de membros e relações que afetam a estrutura e hierarquia de outros tipos. O elemento *Introduction* é o responsável por armazenar os membros (atributos, métodos e construtores) a serem adicionados para um tipo (incluindo outro aspecto) e o elemento

Parents armazena as informações necessárias para que o aspecto possa declarar que outros tipos implementam novas interfaces ou estende uma nova classe.

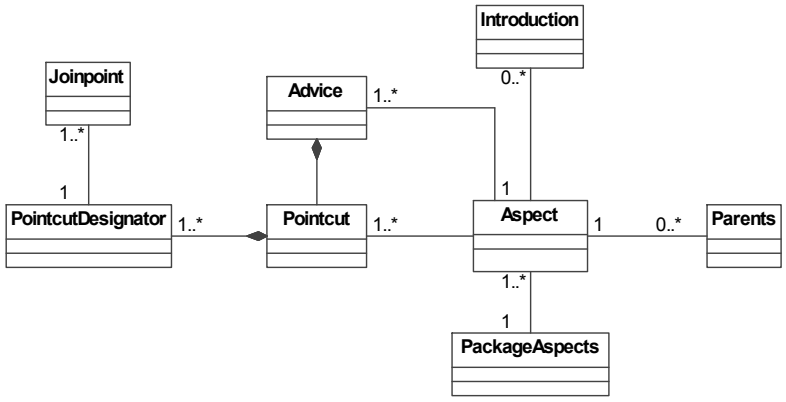


Figura 8. Modelo de mapeamento.

2.1.3. Serviços de Composição do modelo

A composição do modelo é uma fase que finaliza o processo de mapeamento, gerando um modelo que especifica o entrelaçamento entre os aspectos e os elementos de negócios. Esta fase é dividida em 2(duas) subfases que são: (i) combinação (*weaving*) e; (ii) transformação. Para cada sub-fase o CrossMDA provê um serviço.

2.1.3.1 Combinação (*weaving*)

A combinação consiste em integrar o modelo de aspectos ao modelo de negócio gerando as instâncias dos aspectos selecionados e as associações destas com os elementos de negócio. O serviço de combinação do arcabouço CrossMDA é provido por uma classe, chamada de combinador cuja responsabilidade é gerar o programa de transformação. O programa de transformação representa a implementação da especificação formal do processo de composição de modelos, ou seja, a integração entre elementos do modelo de negócio e as instâncias de aspectos mapeados para o novo modelo, o modelo PSM com os aspectos.

A atividade inicia-se quando o combinador recebe um conjunto de instâncias de mapeamentos e gera, internamente no arcabouço CrossMDA, o modelo intermediário. Com base nessa representação intermediária, é iniciada então a geração do programa de transformação. O programa de transformação é gerado através do uso de arquivos de *templates* de códigos (Figura 9) os quais são combinados, isto é, é feita uma fusão dos vários *templates* para a criação de um único *template*, e suas *tags* são substituídas pelas informações dos aspectos, oriundas do modelo de mapeamento, para geração do código final do programa de transformação. Por exemplo, a *tag* <*ASPECT_NAME*> durante a geração do programa é substituída pelo nome de um aspecto. Os *templates* são codificados utilizando a *ATLAS Transformation Language* (ATL) [Jouault e Kurtev 2005], linguagem de transformação proposta pelo ATLAS group (INRIA e LINA, Universidade de Nantes) para a especificação MOF QVT da OMG [OMG 2006c].

O modelo PSM de aspectos gerado em CrossMDA é uma adaptação do modelo de aspectos proposto por [Stein 2002], em que os aspectos são representados por classes da UML marcadas com o estereótipo <<*aspect*>>. Os pontos de atuação são representados por métodos da classe aspecto marcados com o estereótipo <<*pointcut*>.

Os adendos são representados por métodos com o estereótipo <<*advice*>> e com as etiquetas (*TaggedValue*) *type* e *pointcut,* identificando o tipo do adendo (*before, after* ou *around*) e o ponto de atuação respectivamente. Porém, quando realiza-se um aspecto em que um dos pontos de atuação é abstrato, essa definição do adendo não é necessária porque já está definida na classe aspecto abstrata correspondente. As especificações do ponto de atuação, dos PCDs e da assinatura do ponto de junção são mapeados como etiquetas dos métodos pontos de atuação. Ainda, informações relevantes de identificação do aspecto, tais como o seu *namespace* indicando o pacote aonde o aspecto encontra-se no modelo PIM, e o *extends* para indicar a abstração da qual o aspecto deriva, são configuradas como etiquetas da classe.

```
lazy rule newClass  {
from className : String, namespace : String
to t : UML!Class (
  name <-  className,
  visibility <- #vk_public,
  isAbstract <- false,
  namespace <-
    if thisModule.packageExists(namespace) then
       thisModule.getPackage(namespace)
   else
       thisModule.newPackage(thisModule.pckPSM)
   endif,
  stereotype <- thisModule.getStereotype('aspect'))
}
lazy rule newOperation {
from c : UML!Class, s : UML!Operation,
       stereotypeName : String
to t : UML!Operation (
  owner <- c,
  visibility <- #vk_public,
  name <- s.name,
  stereotype <-
    thisModule.getStereotype(stereotypeName),
  ...
} ...
```

```
thisModule.umlClass<-
if thisModule.classExists('<ASPECT_NAME_IMPL>','aspect')
   then thisModule.getClass('<ASPECT_NAME_IMPL>','aspect')
   else thisModule.newClass('<ASPECT_NAME_IMPL>',
                '<ASPECT_OWNER>')
endif;
if thisModule.taggedValueExists(thisModule.umlClass,
      'namespace') then true
   else  thisModule.newTaggedValue(thisModule.umlClass,
            'namespace', Sequence{'<ASPECT_OWNER>'})
endif;
thisModule.umlOperation <-
   if thisModule.operationExists('<ASPECT_NAME_IMPL>',
            '<POINTCUT_NAME>','pointcut')
    then  thisModule.getOperation('<ASPECT_NAME_IMPL>',
            '<POINTCUT_NAME>','pointcut')
   else thisModule.newOperation( thisModule.umlClass,
            thisModule.getOperation('<ASPECT_NAME>',
            '<POINTCUT_NAME>','pointcut'), 'pointcut')
   endif;
if thisModule.taggedValueExists(thisModule.umlOperation,
          'base')  then true
 else thisModule.newTaggedValue(thisModule.umlOperation,
              'base', Sequence{<POINTCUT_VALUE>})
endif;
```

Figura 9. Fragmentos de *templates* de código ATL para criação de classes e métodos.

Como exemplo, a Figura 10 apresenta um fragmento do programa de transformação que representa a integração do aspecto de autenticação com a classe de negócio *Cliente* de acordo com os mapeamentos definidos na seção 2.1.2.

```
thisModule.umlClass<-  if thisModule.classExists('AbstractAuth_Impl','aspect')
                          then  thisModule.getClass('AbstractAuth_Impl','aspect')
                          else  thisModule.newClass('AbstractAuth_Impl',  'br.ufrj.nce.security')  endif;
if thisModule.taggedValueExists(thisModule.umlClass, 'namespace') then true
        else  thisModule.newTaggedValue(thisModule.umlClass, 'namespace', Sequence{'br.ufrj.nce.security'})
endif;
if thisModule.taggedValueExists(thisModule.umlClass, 'extends') then true
        else thisModule.newTaggedValue(thisModule.umlClass,'extends', Sequence{'AbstractAuth'})  endif;
thisModule.umlOperation <-  if thisModule.operationExists('AbstractAuth_Impl', 'authOperations','pointcut')
    then thisModule.getOperation('AbstractAuth_Impl', 'authOperations','pointcut')
    else thisModule.newOperation( thisModule.umlClass,
                thisModule.getOperation('AbstractAuth', 'authOperations','pointcut'),'pointcut')   endif;
if thisModule.taggedValueExists(thisModule.umlOperation, 'base') then true else
  thisModule.newTaggedValue(thisModule.umlOperation,'base', Sequence{'call(public void Cliente.setCpf(String))'})
endif;
```

Figura 10. Fragmentos de uma regra gerada em linguagem ATL a partir de *templates* para a criação de instância da classe aspecto e seu ponto de atuação.

Para uma melhor ilustração da geração do programa de transformação, a Tabela 2 apresenta os elementos mapeados na fase do relacionamento e as *tags* dos *templates* a serem substituídas.

Tabela 2: Mapeamento dos elementos aspectuais nas *tags* dos *templates* para criação de instância da classe aspecto e seu ponto de atuação com seu respectivo ponto de junção referente aos relacionamentos da seção 2.1.2

Nome da Tag	Descrição	Valor
<ASPECT_NAME>	Nome da instância para o aspecto	AbstractAuth
<ASPECT_NAME_IMPL>	Nome de implementação para uma instância de aspecto abstrato	AbstractAuth_Impl
<ASPECT_OWNER>	Identifica o elemento ao qual o aspecto pertence	br.ufrj.nce.security.auth
<POINTCUT_NAME>	Nome da instância para o ponto de atuação	authOperations
<ADVICE_TYPE>	Tipo de ligação para o adendo	-
<POINTCUT_VALUE_ID>	Identificador das regras (valor) para um ponto de atuação	PointcutValueID_1
<POINTCUT_VALUE>	Tipo do designador (PCD) junto a assinatura do ponto de junção	call(public void Cliente.setCpf(String))

2.1.3.2. Transformação do modelo

A atividade de transformação do modelo é iniciada quando um programa de transformação, gerado pelo combinador, necessita ser compilado e executado. CrossMDA provê um serviço (Figura 11) para compilar e executar o programa de transformação e dessa forma gerar o novo modelo, através do uso do motor de transformação da ATL (*ATL engine*) [ATL 2007].

```
public interface IScriptCompiler {
    public void compile (String fileName);
}
```

```
public interface IScriptExecute {
   public int parseArgs(String[] args);
   public String[] setParameters(String script, String in, String out, String libs);
   public void run(); }
```

Figura 11. Interface para serviços de compilação e execução do motor de transformação.

O motor de transformação é um arcabouço que inclui uma máquina virtual (*ATLvm*) e um compilador. Também é fornecido um conjunto de classes escritas usando linguagem Java que oferece, entre outros serviços: (i) *parser,* para realizar a análise sintática do programa de transformação; (ii) *compilador*, para gerar o *byte-code* (arquivo com extensão *asm*) da máquina virtual (*vm)* e; (iii) *executor*, responsável por realizar a carga e execução do programa de transformação.

O resultado final da execução da transformação é um novo modelo que contém as adaptações previstas nas regras declaradas no programa de transformação. Assim, o resultado da execução do programa para o exemplo utilizado é o modelo de negócio adaptado com novas instâncias de classes que representam os aspectos e seus relacionamentos com os elementos de negócio (Figura 12).

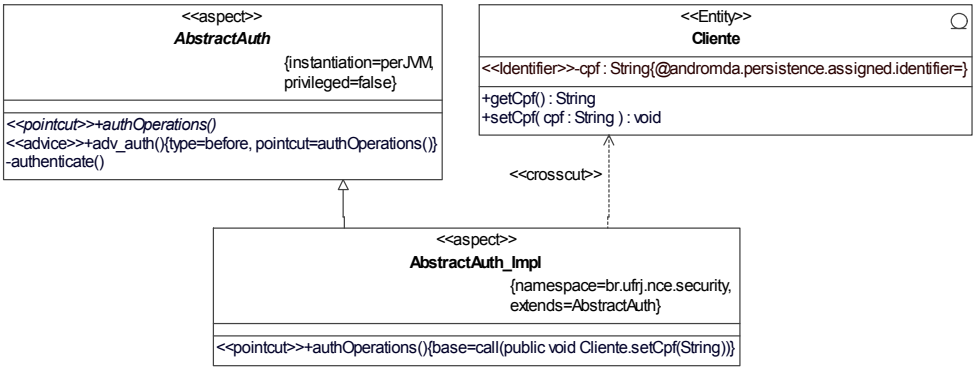


Figura 12. Modelo PSM aplicando-se o aspecto de autenticação na classe Cliente.

3. Trabalhos relacionados

Na área de Desenvolvimento de Software Orientado a Aspecto (DSOA) alguns trabalhos buscam integrar os conceitos da POA na modelagem de sistemas orientados a objetos. Tais trabalhos usam os mecanismos de extensibilidade da UML para adicionar novos elementos de modelagem que representam os conceitos da POA. Em [Aldawud et al. 2003] propõe-se a criação de um perfil UML para DSOA, que oferece aos projetistas meios comuns de representar visualmente os artefatos da POA, criando-se um metamodelo aspectual. Outra abordagem semelhante, denominada Modelo de Projeto Orientado a Aspecto, é proposta por [Stein 2002], na qual elementos da UML são estendidos para representar a semântica dos elementos da linguagem AspectJ [AspectJ 2006]. Esses trabalhos têm a vantagem de utilizar somente os mecanismos de extensibilidade da UML sendo, portanto, facilmente integrados às ferramentas de modelagem existentes. CrossMDA relaciona-se com estas abordagens por utilizar perfis UML na definição de elementos aspectuais que irão compor os modelos PSM gerados.

O trabalho de [Chavez 2004] propõe um arcabouço integrador dos conceitos de POA denominado teoria dos aspectos. Com base nessa teoria foi desenvolvida a linguagem *aSideML* para modelagem de sistemas orientados a aspectos, juntamente com o metamodelo *aSide,* que define a semântica de modelos estruturais e comportamentais representados nessa linguagem. Esse trabalho é um avanço significativo na área, por definir um metamodelo amplo que formaliza a semântica dos elementos relacionados à modelagem orientada a aspectos com UML. Como a linguagem *aSideML* é baseada em um metamodelo específico, faz-se necessária a criação de ferramentas que suportem esse novo metamodelo. Quando tais ferramentas estiverem disponíveis, CrossMDA poderá ter transformadores que gerem modelos no padrão *aSideML*. Dessa forma, o presente trabalho pode ser considerado como complementar ao que foi proposto em [Chavez 2004].

Ainda na linha de modelagem, uma discussão sobre uso de aspectos em modelos de domínio foi proposto por [Steimann, 2005]. Ele argumenta a favor de ter modelos de domínio livres de aspectos baseado na observação da falta de exemplos para aspectos no nível de domínios. Sua hipótese é que a idéia de aspectos em um desenvolvimento orientado a aspectos é um conceito de alto nível. Porém, [Rashid e Moreira, 2006] apresentam argumentos e exemplos que contradizem o trabalho de Steimann mostrando que modelos de domínio podem ter representações de aspectos. Em CrossMDA, os aspectos são representados em modelos separados do modelo de domínio no nível do PIM sendo representados no mesmo modelo somente no nível PSM após o entrelaçamento entre aspectos e negócio realizado pelo processo de transformação.

Na área de MDA, os trabalhos concentram-se na criação de modelos de transformação para facilitar a junção de aspectos com modelos do negócio (os modelos primários). Em [Chaves e Zancanella 2004] é apresentado um conjunto de extensões orientadas a aspectos para UML, chamado *Libra*, que possibilita a especificação de modelos tanto na parte estrutural como comportamental. O modelo de classes é incrementado de forma a representar aspectos e seus relacionamentos com o modelo primário, enquanto para definir os comportamentos é oferecida uma linguagem de ações usando sintaxe XML, com capacidades reflexivas e baseada na semântica de ações da UML. *Libra* utiliza a abordagem de transformação MDA para combinar o modelo de aspectos com os elementos do modelo primário. Apesar desse trabalho evidenciar a viabilidade da junção de POA e MDA para melhor integrar aspectos, por não ser o seu

foco principal, ele não apresenta nenhuma formalização de como essa combinação é realizada e tampouco trata de questões importantes como a especificação e gerenciamento de modelos de composição. Ambos os aspectos são tratados no CrossMDA e a sua formalização é feita através do programa de transformação escrito em uma linguagem formal de transformação [Jouault e Kurtev 2005].

[Reina e Torres 2005, Simmonds et al. 2005] utilizam a linguagem QVT [OMG 2006c] da abordagem MDA para realizar a transformação de modelos. [Reina e Torres 2005] usam a transformação para entrelaçar aspectos de AspectJ e elementos básicos ao nível de PSM antes da geração de código. CrossMDA tem uma abordagem semelhante, porém trabalha com modelos no nível do PIM. Já em [Simmonds et al. 2005] é apresentado um arcabouço que realiza a transformação de modelos de negócio e de aspectos do nível PIM para PSM. O arcabouço trabalha com dois modelos como entrada (primário e genérico de aspectos), os quais são especificados como diagramas de interação da UML. A ligação entre os modelos e a composição do novo modelo é feita através de transformações em QVT baseadas em metamodelos e especificadas pelo projetista. O modelo PSM do aspecto é dependente da plataforma na qual o aspecto será implementada, sendo o modelo de negócio marcado com estereótipos indicando a atuação do aspecto. Da mesma forma, a abordagem CrossMDA também separa os modelos de entrada (primário ou negócio e aspectos). Porém, segue outra abordagem na composição dos modelos, onde os modelos de aspectos são compostos por aspectos modelados em classes da UML [Stein 2002]. O modelo PSM gerado é a realização de uma classe aspecto do modelo PIM junto com a definição dos relacionamentos com o modelo de negócio. Assim, não existe alteração no modelo PIM de negócio, ao contrário da abordagem de Simmonds et. al, que realiza marcações no modelo de negócio para indicar os relacionamentos com aspectos. O modelo PSM gerado no CrossMDA está alinhado com a abordagem de modelagem de ferramentas de transformação modelo-texto existentes, como o AndroMDA [AndroMDA 2006], permitindo assim a continuidade do processo até a geração de código.

4. Conclusão

CrossMDA é um arcabouço que incorpora um processo de transformação para integração de interesses transversais em sistemas orientados a modelos explorando a sinergia entre as abordagens POA e MDA. Tal integração é realizada quando um modelo é transformado de nível PIM para o nível PSM, permitindo, desta forma, que a modelagem de interesses transversais seja ortogonal à modelagem dos processos de negócio. O uso das técnicas de POA auxilia nas atividades de mapeamento dos relacionamentos e na composição do modelo. Já da proposta MDA foram utilizadas a abordagem de transformação, como a base para automatizar o processo de integração dos interesses transversais, e a adoção de uma linguagem de transformação baseada no recente padrão MOF QVT da OMG para geração do programa de transformação. O uso dessa linguagem, aliado à forma de implementação através de *templates* de código, tornam o CrossMDA uma poderosa ferramenta na geração de programas de transformação baseada em aspectos.

Uma preocupação no projeto do CrossMDA é promover um alto grau de reuso de artefatos. No nível de artefatos de transformação, o reuso é alcançado através da utilização de *templates* em ATL capazes de gerar programas de transformação para diferentes sintaxes de linguagens formais de transformação, por exemplo em QVT ou MWDL [Milewski e Roberts 2005]. O uso de *templates* facilita a manutenção e permite

que novas implementações do programa de transformação sejam realizadas sem alterar o código do CrossMDA. Quanto a artefatos de modelo, o reuso é favorecido tanto no nível de PIM quanto PSM. O emprego de modelos PIM de aspectos permite que estes sejam desenvolvidos por qualquer projetista e reutilizados em várias transformações. Por outro lado, modelos PIM de negócio podem ser reaproveitados e entrelaçados com diferentes modelos de aspectos de forma a gerar sistemas que necessitem de diferentes requisitos computacionais. No nível PSM, o CrossMDA gera modelos PSM de aspectos seguindo as tecnologias MDA padrão, no caso XMI, fazendo com que esses modelos sejam utilizáveis por qualquer ferramenta MDA de transformação modelo-texto para geração do código fonte do aspecto. Outra característica importante do CrossMDA é o baixo grau de acoplamento entre os modelos PIM e PSM de aspectos e o modelo de negócio. Essa característica permite um certo grau de independência entre os modelos PIM de negócio e modelos de aspectos, fazendo com que esses modelos (PIM e PSM) possam ser evoluídos sem interferência mútua.

O CrossMDA foi implementado utilizando linguagem de programação Java e inclui as ferramentas necessárias para automatizar todas as atividades do processo proposto através de uma interface gráfica simples e de fácil operação[CrossMDA 2007].

Referências

Aldawud, O.; Elrad, T. and Bader, A. (2003). UML Profile for Aspect-Oriented Software Development. In Third Workshop on Aspect-Oriented Modeling with UML, AOSD'03. Boston, Massachussets, March, 2003.

AndroMDA (2006). Disponível em: http://www.andromda.org. Acesso em: 05/04/2006.

AOM (2006). Aspect-Oriented Modeling Workshop. Disponível em: http://www.aspect-modeling.org. Acesso em: 01/03/2006.

AspectJ (2006), a Java implementation of AOP. Disponível em: http://www.eclipse.org/aspectj. Acesso em: 05/04/2006.

ATL (2007), ATL Home Page. Disponível em: http://www.eclipse.org/m2m/atl/. Acesso em: 11/01/2007.

Baniassad, E. and Clarke, S. (2004). "Theme: An Approach for Aspect-Oriented Analysis and Design" In Proceedings of the 26th ICSE, Edinburgh, Scotland, May 2004.

Chaves, R.; Zancanella, L. C. (2004). Modelos Executáveis Baseados em Aspectos, apresentado no WASP'04 - Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, 18 de Outubro de 2004, Brasília, Brasil.

Chavez, C. F. G. (2004). A Model-Driven Approach for Aspect-Oriented Design. Rio de Janeiro, 2004. 304p. Tese de Doutorado. DI/PUC, Rio de Janeiro, Brasil.

CrossMDA (2007), Disponível em: http://labdist.dimap.ufrn.br/projetos/crossmda.

Graziadei, T. R (2005). Aspect-Oriented Model Weaver, 2005. 127p. Dissertação de Mestrado, Fachhochschule Vorarlberg, Dornbirn, Austria. Disponível em: http://www.sciences.univ-nantes.fr/lina/atl/bibliography/ext_GRAZIADEI05.

Jouault, F. and Kurtev, I. (2005) Transforming Models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda C.; Lopes, C.; Loingtier, J.M.; and Irwin, J. (1997). Aspect-Oriented Programming. Published In ECOOP, Finland, 1997. Springer-Verlarg LNCS 1241.

Laddad, R. (2003). AspectJ in Action, Pratical Aspect-Oriented Programming. Manning Publications CO, 2003, ISBN 1930110936.

Milewski, M. and Roberts, G. (2005). The Model Weaving Description Language (MWDL) - towards a formal Aspect Oriented Language for MDA model transformations. First Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD at the 19th ECOOP.

NetBeans-MDR. (2007). Disponível em: http://mdr.netbeans.org.Acesso em 10/01/2007.

OMG (2006a). MDA Guide version 1.0.1. Formal Document: 03-06-01. Disponível em: http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf. Acesso em: 01/03/2006.

OMG (2006b). OMG Meta-Object Facility (MOF). Formal Document: 2002-04-03. Disponível em: http://www.omg.org/technology/documents/formal/mof.htm. Acesso em: 11/04/2006.

OMG (2006c) MOF QVT. Disponível em: http://www.omg.org/cgi-bin/doc?ptc/2005-11-01. Acesso em: novembro de 2006.

Rashid, A. and Moreira, A. (2006) Domain Models are NOT Aspect Free. Proceedings of MoDELS/UML. Springer, Lecture Notes in Computer Science. Volume 4199, Pages 155-169.

Reina, A.M; Torres, J. (2005). Weaving AspectJ Aspects by means of transformations. First Workshop on Models and Aspects-Handling Crosscutting Concerns in MDSD at ECOOP 2005.

Simmonds D., Solberg A., Reddy R., France R., Ghosh, S. (2005). "An Aspect Oriented Model Driven Framework", Accepted to Ninth IEEE "The Enterprise Computing Conference" (EDOC 2005), Enschede, Netherlands, 19-23 September, 2005.

Solberg, A.; Simmonds, D.; Reddy, R.; Ghosh, S.; France, R. (2005). Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development. 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1 pp. 121-126.

Suzuki, J. and Yamamotto, Y (1999). Extending UML with Aspects: Aspect Support in the Design Phase. 3rd Aspect-Oriented Programming Workshop at the 13th ECOOP. Lisbon, Portugal, 1999.

Stein, D. (2002). An Aspect-Oriented Design Model Based on AspectJ and UML. 2002. 186p. Dissertação de Mestrado, Universidade de Essen, Alemanha.

Stein, D.; Hanenberg, S.; Unland, R. (2002). Designing Aspect-Oriented Crosscutting in UML; 1st International Workshop on Aspect-Oriented Modeling with UML, AOSD 2002, Enschede, The Netherlands, April 22, 2002.

Steimann, F. (2005). Domain models are aspect free. In: MoDELS 2005, 8th International Conference on Model Driven Engineering Languages and Systems. 171185.

Tekinerdogan, B.; Moreira, A.; Araújo, J.; Clements, P. (2004). Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. Workshop Proceedings, University of Twente, TR-CTIT-04-44, 119 pp, October 2004.

Wampler, D. (2003). The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture. Disponível em: http://aspectprogramming.com/papers/AOP and MDA.pdf. Acesso em:Outubro/2005.

Transformando Modelos da MDA com o apoio de Componentes de Software♦

Marco Antonio Pereira[1], Antonio Francisco do Prado[1], Mauro Biajiz[1], Valdirene Fontanette[1], Daniel Lucrédio[2]

[1]Universidade Federal de São Carlos, Departamento de Computação
Rod. Washington Luís, Km 235 - Caixa Postal 676 - Cep.13565-905 - São Carlos, SP - Brasil

[2]Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação
Av. Trabalhador São-carlense, 400, Centro - Caixa-Postal: 668 - Cep. 13560-970 - São Carlos, SP - Brasil

pemarco@gmail.com, {prado, mauro, valdirene}@dc.ufscar.br, lucredio@icmc.usp.br

Abstract. *This article presents the use of software components to supply the Model Drivel Architecture (MDA) in the MVCASE tool. MDA represents the software specifications from its modeling to implementation. In a lower abstract level there are Object-Oriented models supported by the UML component. Using transformation components it is possible to obtain the Object-Relational DataBase models, and to get through, obtain also the codes in Structed Query Language (SQL). A example illustrates the use of these components to instantiate and transform different platforms models and finally obtain their codes in SQL.*

Resumo. *Este artigo apresenta o uso de componentes de software para operacionalizar a Model Driven Architecture (MDA) na ferramenta MVCASE. A MDA representa as especificações do software desde a sua modelagem até a implementação. Em nível menos abstrato, têm-se os modelos Orientados a Objetos suportados pelo componente UML. Utilizando-se de componentes de transformação é possível obter os modelos de Banco de Dados Objeto-Relacional e por fim, códigos em Structured Query Language (SQL). Um exemplo de uso ilustra o uso dos componentes para instanciar e transformar modelos de plataformas diferentes e por fim, obter seus códigos em SQL.*

♦ Projeto Financiado pelo Programa de Apoio à Inovação Tecnológica em Pequenas Empresas (FAPESP - PIPE)

1. Introdução

Muitas organizações que desenvolvem *softwares* utilizam diferentes linguagens, metodologias e ferramentas para aumentar a produtividade da equipe de desenvolvimento e diminuir os custos com o processo de desenvolvimento de *softwares*. Porém, a complexidade obtida pelo uso de diferentes soluções pode acarretar perda de qualidade e de reusabilidade. Visando aumentar a reusabilidade e a qualidade, modelos de *software* podem ser mantidos em diferentes níveis de abstração, a fim de prover o uso de modelos abstratos para se obter novos modelos em plataformas específicas.

Neste contexto encontra-se o *Model-Driven Development* (MDD) [Stahl and Völter 2006], que é um termo usado para expressar a idéia do desenvolvimento orientado a modelos. O foco do MDD são os modelos que representam a abstração do mundo real. Esses modelos não são apenas documentação auxiliar, mas são também artefatos de *softwares* que podem ser compilados diretamente em outros modelos e códigos em linguagens de programação [Czarnecki *et al*. 2005]. O *Object Management Group* (OMG) propôs uma abordagem ao MDD chamada de *Model Driven Architecture* (MDA) [Mda 2003]. A MDA utiliza linguagens padronizadas pela OMG para representar esses artefatos de *software*s.

As linguagens utilizadas na MDA são denominadas de meta-metalinguagens, metalinguagens e linguagens de acordo com o nível de abstração em que são utilizadas na arquitetura. Essas linguagens podem ser integradas em uma ferramenta *Computer-Aided Software Engineering* (CASE) com o objetivo de suportar modelos de diferentes plataformas. Neste artigo, particular ênfase é dada ao desenvolvimento e integração dessas linguagens por meio de componentes de *software* a fim de estender as funcionalidades da ferramenta MVCASE. Essa ferramenta já suporta o desenvolvimento de modelos Orientados a Objetos (OO) e para que a mesma também suporte o desenvolvimento de modelos de Bancos de Dados Objeto-Relacional (BDOR) [Zendulka 2005] e que por fim, possa gerar códigos em *Structured Query Language* (SQL) [Sql 1999] uma extensão baseada em componentes é proposta neste artigo.

Os componentes responsáveis pelas transformações são chamados de *Componentes Transformadores de Modelos* e *Componentes Geradores de Códigos*. Esses tem suas regras de transformações definidas e implementadas, e eles são integrados à MVCASE por meio de suas interfaces. O *Componente Transformador de Modelos* tem a função de mapear os elementos de um modelo OO para elementos de um modelo BDOR. Finalmente, por meio do *Componentes Gerador de Códigos*, os códigos em SQL são obtidos a partir de modelos BDOR. O uso de componentes para operacionalizar a MDA é um fator importante para prover reuso dos *Componentes Transformadores de Modelos* e dos *Geradores de Códigos* em outras ferramentas que também implementem os conceitos da MDA e as linguagens padronizadas pela OMG. A MDA é uma tecnologia que vem evoluindo com o passar do tempo e fatores fundamentais para a adoção prática da MDA, como operacionalização e definição de transformações de modelos ainda continuam em evidência entre os pesquisadores.

Este artigo está estruturado da seguinte maneira: na seção 2 são contextualizadas as linguagens *Meta Object Facility* (MOF) [Mof 2002], *Unified Modeling Language* (UML) [Omg 2004], *Common Warehouse Metamodel* (CWM) [Omg 2003] e *XML*

Metadata Interchange (XMI) [Xmi 2003] com os níveis de abstração da MDA; na seção 3 são apresentadas as definições das regras de transformação; na seção 4 é apresentado um protótipo da MDA implementado na MVCASE e um exemplo de uso, no qual se utiliza de componentes para transformar modelos OO em modelos BDOR e por fim, se obter os códigos em SQL; na seção 5 são descritos trabalhos correlatos; e finalmente, na seção 6, têm-se as conclusões.

2. *Model Driven Architecture*

A *Model Driven Architecture* (MDA) fundamenta-se na idéia da separação entre as especificações de um sistema e os detalhes de sua implementação [Mda 2003]. As especificações do *software* são definidas em altos níveis de abstração, as quais podem originar novas: i) especificações em níveis menos abstratos, como por exemplo, a obtenção de especificações em CORBA e Java a partir de um modelo UML; e ii) especificações para diferentes plataformas, como por exemplo, a obtenção de modelos de Bancos de Dados a partir de modelos UML.

O OMG padronizou linguagens para facilitar a integração entre os modelos da MDA. Essas linguagens são classificadas em um determinado nível de abstração na arquitetura. No nível mais abstrato, encontra-se a meta-metalinguagem, que se auto descreve e descreve as metalinguagens, as quais por sua vez, descrevem as linguagens.

A Figura 1 ilustra os diferentes níveis de abstrações (M_0, M_1, M_2 e M_3) da MDA. As meta-metalinguagem, metalinguagens e linguagens da arquitetura são descritas pelos seus correspondentes meta-metamodelo, metamodelos e modelos.

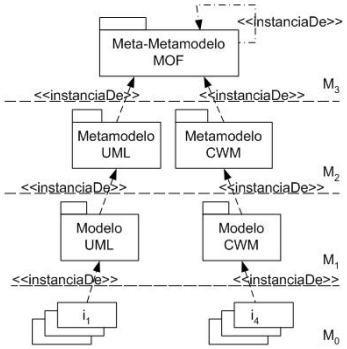


Figura 1 - Níveis de abstrações da MDA [MOF 2002]

No nível M_3, o mais abstrato, encontra-se o *Meta-Metamodelo MOF* (*Meta Object Facility* - MOF), que é uma instância do seu próprio meta-metamodelo. O *Meta-Metamodelo MOF* descreve uma meta-metalinguagem abstrata utilizada para descrever outras metalinguagens para diferentes domínios e também descreve um repositório de metadados para suportar metadados dos metamodelos baseados no próprio *Meta-Metamodelo MOF* [Mof 2002].

Os metamodelos no nível M_2, instâncias do *Meta-Metamodelo MOF*, são criados de acordo com os requisitos de um particular domínio do problema, como por exemplo, para o domínio OO, o *Metamodelo UML* descreve uma metalinguagem para especificar modelos nesse domínio [Omg 2004] e para o domínio *Data Warehouse* (DW) e *Business Intelligence* (BI), o *Metamodelo CWM (Common Warehouse Metamodel* -

CWM) descreve uma metalinguagem para especificar modelos para esse domínio [Poole *et al*. 2003].

Os modelos no nível M_1, o *Modelo UML* e o *Modelo CWM*, são instâncias de seus correspondentes metamodelos do nível M_2. Finalmente no nível menos abstrato, o M_0, têm-se as instâncias executáveis (i_1, i_2,..., i_n) dos modelos, que são descritas de acordo com as correspondentes linguagens de seus modelos do nível M_1.

Os meta-metamodelo, metamodelos e modelos da MDA podem ser descritos em *XML Metadata Interchange* (XMI) [Xmi 2003]. XMI é uma linguagem da OMG que define um conjunto de regras para mapear meta-metamodelo, metamodelos e modelos para documentos *eXtensible Markup Language* (XML) [Xml 2006]. Os documentos XML têm o objetivo de prover, de maneira simples e independente de plataforma, a interoperabilidade através do uso de cadeias de textos dotadas de descrições. XMI utiliza a *Document Type Definition* (DTD) para validar os documentos XML de acordo com os seus respectivos metamodelos. A DTD define a estrutura dos elementos que podem ser descritos nos documentos XML.

3. Transformação de Modelos

A transformação de modelos pode ocorrer entre mesmos níveis e entre diferentes níveis de abstração e também pode ocorrer entre mesmos domínios e diferentes domínios. A transformação é a geração automática de um modelo *destino* a partir de um modelo *origem.* A transformação é definida por um conjunto de regras que juntas descrevem como um modelo na linguagem origem pode ser transformado em um ou mais modelos na linguagem destino [Kleppe *et al*. 2003].

A Figura 2 ilustra a idéia da transformação de modelos. Um modelo independente de plataforma *(Plataform Independent Model* - PIM), pode ser transformado em novos PIMs e novos modelos dependentes de plataforma *(Plataform Specific Model* - PSM). Os novos PIMs podem possuir algumas características especificas, mas essas podem não o classificar como um PSM, já os novos PSMs possuem características específicas de uma determinada plataforma. Um PSM pode ser transformado em novos PSMs e novos PIMs. Os novos PSMs são refinamentos do PSM original, já os novos PIMs são modelos que substituem as características de uma determinada plataforma para características de uma plataforma independente.

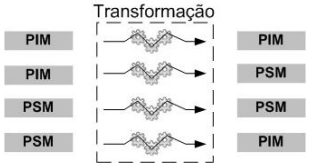


Figura 2 - Transformação de Modelos

As transformações de modelos podem ser chamadas de *Transformação Modelo-Modelo* e *Transformação Modelo-Texto*. Na categoria *Modelo-Modelo*, a transformação pode ser distinguida entre as seguintes abordagens: manipulação direta; orientada a estrutura; operacional; baseada em *template;* relacional; baseada em grafos e híbrida. Na categoria *Modelo-Texto*, as transformações podem ser distinguidas entre as abordagens baseadas no padrão *Visitor* [Gamma *et al.* 1995] e baseadas em *templates*

[Czarnecki e Helsen 2006]. As transformações *Modelo-Texto* são aplicadas para gerar códigos em uma determinada linguagem a partir de um PSM [Mda 2003].

A Figura 3 ilustra os conceitos básicos da *Transformação Modelo-Modelo*. Para que ocorra a transformação entre modelos é preciso definir as regras de transformação do Transformador. Essas regras são baseadas no conhecimento das estruturas dos elementos dos metamodelos *Origem* e *Destino*. O transformador recebe como entrada o *Origem'* Modelo e o transforma no *Destino'* Modelo.

Figura 3 - Definição de Regras de Transformação [Czarnecki e Helsen 2006]

De acordo com os domínios dos metamodelos *Origem* e *Destino*, o Transformador pode viabilizar: i) novos modelos no mesmo domínio, a fim de se obter um grau maior de especificidade em relação ao modelo original, essa transformação é chamada de *endogenous* ou *rephrasings* e; ii) novos modelos em domínios diferentes, a fim de se obter reuso de modelos para criação de novos modelos em diferentes domínios, essa transformação é chamada de *exogenous* ou *translations* [Mens e Van Gorp. 2005]. Uma aplicação pode ser gerada automaticamente de uma especificação escrita em uma linguagem textual ou gráfica de um determinado domínio de problemas [Czarnecki 2004].

Na próxima seção são apresentados detalhes da implementação do protótipo da MDA na MVCASE utilizando os conceitos abordados.

4. Protótipo da MDA na MVCASE

Baseado nos estudos e idéias apresentadas, foi implementada um protótipo da MDA na ferramenta *Multiple View* CASE (MVCASE) [Paiva *et al.* 2006]. A MVCASE é uma ferramenta que suporta a modelagem de sistemas de *software* com a notação UML [Booch *et al*. 2005]. O sistema modelado pode ser especificado segundo quatro visões: Casos de Uso; Lógica; Componentes; Visão "*Deployment*" ou Implantação.

O protótipo tem como objetivo operacionalizar os conceitos da MDA através do uso de componentes de *software*. Esses componentes transformam um sistema modelado na Visão Lógica da MVCASE (Modelo de Classes) em um sistema modelado para BDOR, e desse para códigos em SQL. Um componente é um conjunto de vários artefatos de *software* que podem ser independentemente desenvolvidos e ainda, pode ser composto para construir algo maior [D'Souza e Wills 1999].

A Figura 4 mostra os diferentes níveis de abstração da MDA e os componentes integrados à MVCASE. O componentes *MDRComp* e *UMLComp* já fazem parte da arquitetura da MVCASE. O *MDRComp* é responsável por armazenar os metadados dos componentes que dependem dele e suas instâncias correspondentes. O componente *UMLComp* prove o suporte aos modelos OO *(Modelos UML)*. O componente *CWMComp* é integrado à MVCASE com o objetivo de prover o suporte aos modelos BDOR *(Modelos CWM)*. O componente *UML2CWM* é um *Componente Transformador*

de Modelos que é integrado com o objetivo de prover o suporte a *Transformação Modelo-Modelo*, isto é, de *Modelos UML* para *Modelos CWM*. O *UML2CWM* é implementado de acordo com a abordagem de manipulação direta, na qual a lógica das regras de transformação e o escalonamento dessas regras são implementados diretamente no código do componente.

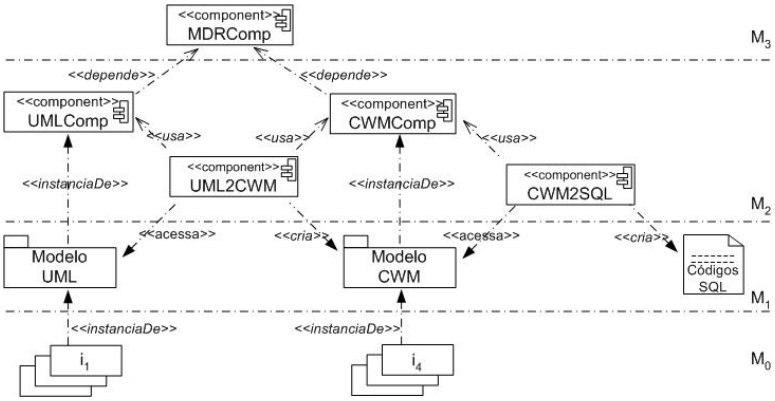


Figura 4 – Componentes que apóiam a MDA na MVCASE

O componente *CWM2SQL* é um *Componente Gerador de Códigos* que é integrado à MVCASE com o objetivo de prover o suporte a *Transformação Modelo-Texto*, isto é, de *Modelos CWM* para *Códigos SQL*. O CWM2SQL é implementado de acordo com a abordagem de manipulação direta, e suas regras de transformação consistem em navegar na estrutura interna dos elementos do *Modelo CWM*, coletando informações e as transformando em *Códigos SQL*.

4.1. Integrando Componentes

A Figura 5 mostra a integração desses componentes com a MVCASE, a qual é representada na figura por meio do pacote MVCASE. A MVCASE usa os componentes *MDRComp*, *CWMComp*, *CWM2SQL*, *CWM2UML*, *UMLComp* por meio de suas interfaces.

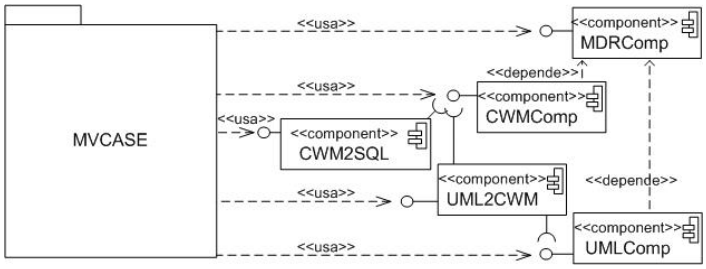


Figura 5 - Componentes integrados à MVCASE

O MDRComp é uma implementação do MOF, do JMI *(Java Metadata Interface)* e do XMI fornecida gratuitamente pela *NetBeans Community* e chamada de *Metadata Repository* (MDR) [Matula 2003]. O JMI implementado no *MDRComp* define o mapeamento de metamodelos baseados no MOF para a linguagem Java e define também um conjunto de interfaces reflexivas que podem ser usadas para acessar instâncias dos metamodelos. O XMI implementado no *MDRComp* possibilita a descrição dos metadados dos metamodelos em documentos XML. O MOF

implementado possibilita que metadados de metamodelos baseados no MOF sejam armazenados no *MDRComp*.

O *UMLComp* é baseado no *Metamodelo UML* [Omg 2004] e o *CWMComp* é baseado no *Metamodelo CWM* [Omg 2003], ambos os metamodelos são baseados no MOF, assim, esses componentes dependem das implementações do *MDRComp* para armazenar seus metadados, instâncias e ler e escrever informações em XMI.

O *UML2CWM* acessa as interfaces do *UMLComp* e do *CWMComp*, isto possibilita a ele navegar por suas estruturas e coletar informações de suas instâncias, ou seja, dos *Modelos UML* e *CWM*, a fim de prover a transformação de forma determinística, unidirecional e guiada por informações contidas nos *Modelos UML* e *CWM* [Mda 2003].

O *SQL2CWM* acessa a interface do *CWMComp*, isto também possibilita a ele navegar por sua estrutura e coletar informações do *Modelo CWM*, a fim de também prover essa transformação de forma determinística e unidirecional, guiada por informações contidas nos *Modelos CWM* e *Códigos SQL*.

A Figura 6 ilustra por meio do *Modelo de Casos de Usos de Componentes de Transformação* o comportamento dos componentes *UML2CWM* e *CWM2SQL*. No caso do *UML2CWM*, tem-se um *Modelo UML* como entrada que é submetido ao caso de uso *Transformar Modelo UML para Modelo CWM*. Esse caso de uso organiza o projeto de transformação do *Componente Transformador de Modelos* e incorpora em uma dada seqüência o comportamento dos seguintes casos de uso: *Transformar Tipo de Dado*, *Transformar Classe*, *Transformar Atributo*, *Transformar Associação* e *Transformar Generalização*, os quais são responsáveis pela transformação de instâncias UML em instâncias CWM. Tem-se como saída do caso de uso *Transformar Modelo UML para Modelo CWM* um *Modelo CWM*.

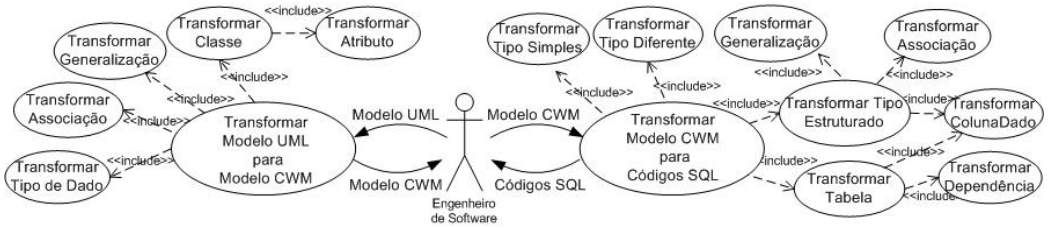


Figura 6 - Modelo de Casos de Usos de Componentes de Transformação

No caso do *CWM2SQL*, tem-se um *Modelo CWM* como entrada que é submetido ao caso de uso *Transformar Modelo CWM para Códigos SQL*. Esse caso de uso organiza o projeto de transformação do *Componente Gerador de Códigos* e incorpora em uma dada seqüência às operações *Transformar SQLSimpleType, Transformar SQLDistinctType, Transformar SQLStructuredType, Transformar Generalização, Transformar Associação, Transformar ColunaDado, Transformar Tabela* e *Transformar Dependência,* os quais são responsáveis pelo mapeamento de instâncias CWM para códigos SQL. Tem-se como saída do caso de uso *Transformar Modelo CWM para Códigos SQL* os *Códigos SQL* correspondentes ao *Modelo CWM* de entrada.

A Tabela 1 e a Tabela 2 sumarizam as transformações executadas pelos Casos de Uso da Figura 6.

Tabela 1 – Transformações executadas pelo UML2CWM

Instância UML	**Casos de Uso**	**Instância CWM**
DataType	Tipo de Dado	<cwmDataTypeList> <cwmDataTypeList> = SqlSimpleType | SqlDistinctType | SqlStructuredType
Class	Classe e Atributo	SqlStructuredType, Column, Table e Dependency
Association	Associação	Association
Generalization	Generalização	Generalization

Tabela 2 – Transformações executadas pelo CWM2SQL

Instância CWM	**Casos de Uso**	**Código SQL**
SqlSimpleType	Tipo Simples	<sqlSimpleTypeList> <sqlSimpleTypeList> = char | number | date | etc
SqlDistinctType	Tipo Distinto	Create Distinct Type *sqlDistinctTypeInstance* As <sqlSimpleTypeList>
SqlStructuredType, Column, Association e Generalization	Tipo Estruturado, ColunaDado, Associação e Generalização	Create Type *SqlStructuredTypeInstance* <elementList> <columnsList> <elementList> = As Object | As Table Of *sqlStructuredTypeInstance* | Varray [1..9] Of *sqlStructuredTypeInstance* | Under *sqlStructuredTypInstance* <columnsList> = *columnInstance* <dataTypeList> <dataTypeList> = *sqlSimpleTypeList* | *sqlDistinctTypeInstance* | <sqlStructuredTypeList> <sqlStructuredTypeList> = [*Ref*] *sqlStructuredTypeInstance*
Table e Dependency	Tabela, Dependência e ColunaDado	Create Table *tableInstance* Of *dependencyInstance* [*columnInstance* <typeList>] <typeList> = *PrimaryKey* | Nested Table Store As

A Tabela 3 mostra os estereótipos utilizados para modelar *Instâncias CWM*. Esses estereótipos ajudam a caracterizar a semântica de modelos BDOR na notação UML [Zendulka 2005].

Tabela 3 - Estereótipos de Instâncias CWM

Instância CWM	**Estereótipo**
SqlStructuredType	*ObjectType*
Table	<stereotypeList> <stereotypeList> = *ObjectTable* | *NestedTable*

4.2.Usando a MDA na MVCASE

Uma aplicação simples do domínio de vendas é utilizada para ilustrar a operacionalização dos componentes dos níveis M_3 e M_2 sobre modelos do nível M_1, isto é, constrói-se um modelo OO e a partir desse obtêm-se um modelo BDOR e por fim, seus códigos em SQL Essa aplicação é chamada de *Sale* e seu *Modelo UML* foi especificado na MVCASE usando um editor gráfico orientado pela sintaxe e semântica da UML.

A Figura 7 ilustra o *Modelo UML* da *Sale*, esse modelo mostra que uma *Pessoa* pode ser um *Consumidor*, o qual pode estar associado a um ou mais *Pedido*. Cada *Pedido* é composto por um ou mais *ItemPedido* e pode estar associado a apenas um *Consumidor*. Cada *ItemPedido* está associado a um único *Produto* e faz parte de apenas um *Pedido*.

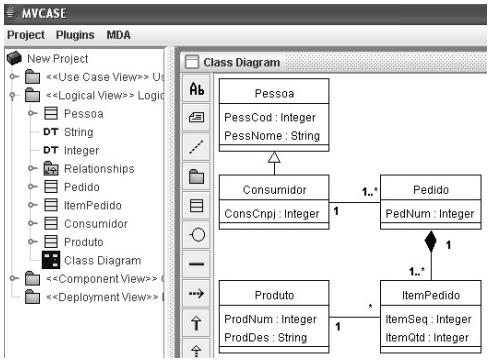


Figura 7 - Modelo UML da *Sale*

A Figura 8 ilustra o *Modelo CWM* obtido a partir do *Modelo UML* da Figura 7 por meio das transformações automáticas executadas pelo *UML2CWM*. O *Modelo CWM* obtido é um modelo BDOR da *Sale*, o qual está de acordo com as características OO contidas no padrão SQL [Sql 1999].

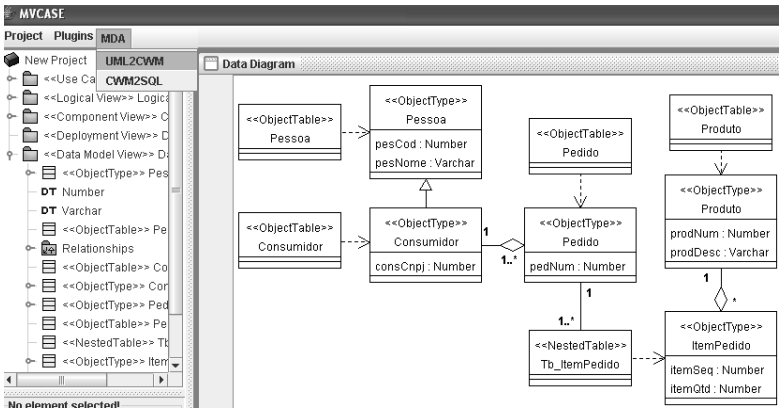


Figura 8 - Modelo CWM da *Sale*

Para obter o *Modelo CWM* a partir do *Modelo UML* da *Sale*, o *UML2CWM* recebe como entrada *Instâncias UML* e as transformam em *Instâncias CWM* de acordo com as transformações descritas na Tabela 1 adicionadas de estereótipos descritos na Tabela 3. Para o caso do *Modelo UML* da *Sale*, as seguintes regras foram utilizadas:

i) Tipo de Dado: instâncias *DataType* transformadas em *SqlSimpleType*, isto é, as instâncias Integer e String do tipo *DataType* no *Modelo UML* são transformadas nas correspondentes instâncias *Number* e *Varchar* do tipo *SqlSimpleType* no *Modelo CWM;*

ii) Classe e Atributo: instâncias *Class* transformadas em *Table*, *SqlStructuredType* e *Dependency*, isto é, as instâncias *Pessoa*, *Consumidor*, *Pedido*, *ItemPedido* e *Produto* do tipo *Class* no *Modelo UML* são transformadas em instâncias

no *Modelo CWM* do tipo *SqlStructuredType* com estereótipo *ObjectType*, *Table* com estereótipo *ObjectTable* e *Dependency;*

iii) Associação: instâncias *Association* transformadas em *Association*, isto é, as instâncias do tipo *Association* no *Modelo UML* que possuem cardinalidade "*1xN*" são transformadas em instâncias do tipo *Association* no *Modelo CWM* com o valor de agregação para o lado com cardinalidade "*N*". Para o caso da associação que possui em um dos lados a composição *(composite)* no *Modelo UML*, são modeladas com um tipo estereótipo do tipo *Nested Table*, executando ainda a atualização das instâncias *Association* e *Table* de *CWMComp;* e

iv) Generalização: instâncias *Generalization* transformadas em *Generalization*, isto é, as instâncias do tipo *Generalization* no *Modelo UML* são transformadas em instâncias do tipo *Generalization* no *Modelo CWM*.

A Figura 9 mostra os códigos em SQL obtidos a partir do *Modelo CWM* da Figura 8 com o auxílio dos *Componentes Geradores de Códigos*.

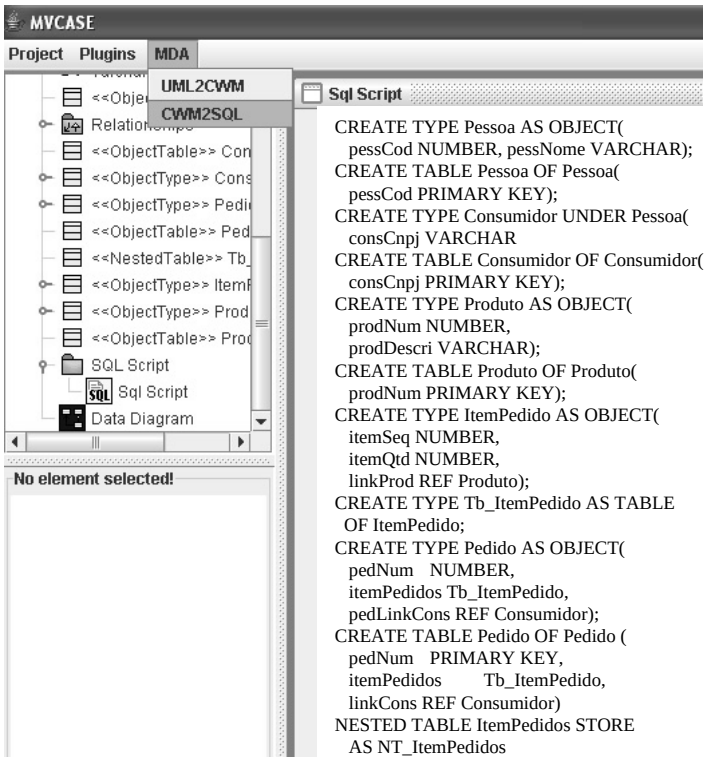


Figura 9 – Código em SQL

Para obter *Códigos SQL* a partir do *Modelo CWM* da *Sale*, o *CWM2SQL* recupera instâncias no *Modelo CWM* e as transformam, em *Códigos SQL* de acordo com as características descritas na Tabela 2. Para o caso do *Modelo CWM* da *Sale*, as seguintes regras foram utilizadas:

i) Tipo Simples: instâncias *SqlSimpleType* e *SqlDistinctType* são mapeadas para o código *"instancia.nome() instancia.type()";*

ii) Tipo Estruturado, ColunaDado, Associação e Generalização: instâncias *SqlStructuredType* que possuem estereótipos com o valor *ObjectType* e que não fazem parte como *"filha"* em um dos lados do relacionamento de instâncias *Generalization* no *Modelo CWM,* são mapeadas para o código *"Create Type instância.nome() As Object"*. Já as instâncias que fazem parte como *"filhas"* em um dos lados do relacionamento de uma instância *Generalization,* são mapeadas para o código *"Create Type instância.filha() Under instância.pai()"*. Por exemplo, a instância *Pessoa* origina o seguinte código *"Create Type Pessoa As Object"*. É o caso também das instâncias *Produto, ItemPedido e Pedido.* A instância *Consumidor* por fazer parte como *"filha"* em um relacionamento de uma instância *Generalization,* é mapeada para o seguinte código "*Cre*ate *Type Consumidor UNDER Pessoa*", indicando que o *Consumidor* é uma especialização de *Pessoa;*

iii) Tabela, Dependência e ColunaDado: instâncias *Table* que possuem estereótipos com o valor *ObjectTable* e que possuem uma instância *Dependency* para uma outra instância *SqlStructuredType* cujo valor de estereótipo seja *ObjectType,* são mapeadas para o seguinte código *"Create Table instância.nome() Of dependênciaInstância.nome()",* essa sintaxe SQL cria uma tabela com a mesma estrutura de um objeto no banco de dados. Por exemplo, a instância *Consumidor* origina o código *"Create Table Consumidor Of Consumidor",* como é também o caso das instâncias *Pessoa, Pedido* e *Produto;*

iv) Tipo Estruturado, ColunaDadoTabela, Associação e Generalização: instâncias do tipo *SqlStructuredType* que possuem estereótipos com o valor *NestedTable* e que possuem uma dependência para uma outra instância do tipo *ObjectType* cujo estereótipo seja *ObjectType,* são mapeadas para o seguinte código *"Create Type instância.nome() As Table Of dependênciaInstância.nome()",* essa sintaxe SQL cria uma tabela aninhada que pode ser referenciada por um atributo em uma outra tabela no banco de dados. Por exemplo, a instância *Tb_ItemPedido* origina o código *"Create Type Tb_ItemPedido As Table Of ItemPedido";*

v) Tipo Estruturado, ColunaDadoTabela, Associação e Generalização: instâncias *Association* que possuem em um dos lados o valor *aggregate* são mapeadas para um atributo que faz referência ao outro lado da associação de acordo com o seguinte código *"'link'+instanciaLadoSemAggregate.nome() Ref instânciaLadoSemAggregate.nome()",* essa sintaxe SQL cria um atributo do tipo referência de outros objetos no banco de dados. Por exemplo, na instância do tipo *Association* que possui em um lado a instância *Produto* e no outro lado a instância *ItemPedido* com o valor *aggregate,* é criado um atributo em *ItemPedido* que referência a instância *Produto,* de acordo com o código *"linkProduto REF Produto";* e

vi) Tipo Estruturado, ColunaDadoTabela, Associação e Generalização: instâncias do tipo *Association* que relacionam duas outras instâncias dos tipos *SqlStructuredType* e *Table,* cujos estereótipos são *ObjectType* e *NestedTable,* são mapeadas para um atributo na instância de estereótipo *ObjectType* que faz referência a instância do lado de estereótipo *NestedTable* de acordo com o código *"instânciaTb.nomeSemTb()+'s' instânciaTb.nome() Nested Table instânciaTb.nomeSemTb()+s Store As Nt_+instânciaTb.nomeSemsTb()",* essa sintaxe SQL cria uma atributo do tipo tabela aninhada no banco de dados. Por exemplo, na instância *Association* que relaciona as instâncias *Pedido* e *Tb_ItemPedido,* um atributo

que referencia *Tb_ItemPedido* é criado no lado *Pedido* com o seguinte código SQL *"ItemPedidos Tb_ItemPedido Nested Table ItemPedidos Store As Nt_ItemPedido"*.

5. Trabalhos Correlatos

Diversas abordagens têm sido propostas para a transformação de modelos, dentre elas encontra-se: a *Query/Views/Transformations* (QVT) [Qvt 2005], que visa permitir a definição das regras de transformações unidirecionais por meio da abordagem declarativa e imperativa; a *MOF to Text Transformation* [M2T], que define uma abordagem baseada em *Templates* para especificar as regras de transformação entre modelos baseados no MOF e texto; a *Visual Automated Model* (VIATRA) [Varró *et al*. 2004], que visa criar os relacionamentos entre os elementos dos *Modelos Origem* e *Destino* por meio de grafos que os relacionam, aplicando regras de transformação de forma não determinística e a MT *Model Transformation Language* [Tratt 2006], que é uma derivação da QVT e permite transformações unidirecionais por meio da abordagem declarativa permitindo que códigos sejam embutidos as regras de transformação. Além dessas abordagens, foram encontradas diversas implementações em ferramentas existentes que se aproximam deste trabalho, dentre elas: o projeto *Generative Modeling Technologies* (GMT) [Gmt 2006]; e o *framework* AndroMDA [Andromda 2006].

O GMT é um dos projetos da *Eclipse Community* que produz um conjunto de ferramentas para a *Model-Driven Engineering* (MDE). Uma dessas ferramentas faz uso da *Atlas Transformation Language* (ATL). A ATL é uma linguagem baseada no MOF e baseada numa sintaxe concreta para *Transformação Modelo-Modelo* através da combinação de linguagem declarativa e imperativa. As regras de transformações são descritas em ATL [M2m 2006] e são processadas pelo ATL *Development Tooling* (ADT), o qual funciona com o suporte da *Integrated Development Environment* (IDE) Eclipse. Os componentes de transformação apresentados neste artigo possuem dependência da máquina virtual Java e de outros componentes de metamodelos.

O AndroMDA destaca-se pela sua estabilidade e relativa maturidade. Ele é um *framework* extensível para a MDA que recebe como entrada um XMI e gera uma saída utilizando *templates* configuráveis específicos para plataformas pré-determinadas. Além dos *templates* prontos, como por exemplo, Hibernate, Struts, JSF e outros, é possível criar novos *templates* escritos em *Velocity Template Language* (VTL) de acordo com cada necessidade. No AndroMDA, a geração de códigos é dirigida por estereótipos especificados no modelo UML, servindo como guias para os *templates.* Portanto, os modelos UML especificados precisam ser mais elaborados e completos. Já o protótipo proposto, necessita apenas de um modelo UML simples, sem a necessidade de estereótipos para guiar a transformação. Assim, o Engenheiro de *Software* precisa se concentrar apenas nos aspectos relacionados ao domínio do problema, deixando que questões relacionadas as características particulares de cada plataforma sejam adicionadas automaticamente na transformações.

A implementação proposta diferencia-se ainda dos trabalhos citados porque permite que os *Componentes de Transformação de Modelos* e de *Geração de Códigos*, isto é, os componentes que possibilitam a transformação de modelos OO em modelos BDOR e códigos SQL, sejam reutilizados por outras ferramentas que utilizam o MDR [Matula 2003] como repositório de metadados e que utilizam os metamodelos UML [Omg 2004] e CWM [Omg 2003].

6. Conclusão

A principal contribuição deste trabalho compreende a integração das diferentes linguagens MOF, UML, CWM e XMI da OMG com os componentes que provêem a Transformação Modelo-Modelo e Modelo-Texto, isto é, de modelos OO para modelos BDOR e posteriormente, códigos em SQL. Essas transformações são viabilizadas por meio de componentes que podem ser reutilizados em outras ferramentas de desenvolvimento de *software*. A transformação de modelos é uma contribuição importante, visto que possibilita o reuso de informações entre plataformas diferentes, proporcionando maior rapidez e qualidade no desenvolvimento de *software* além de não exigir do Engenheiro de *Software* nenhum conhecimento específico na área de modelagem de BDOR.

Os resultados da implementação do protótipo estão direcionando os próximos passos. Assim, a continuidade do trabalho compreende: um estudo maior de cada abstração que envolve a manutenção e refino dos modelos BDOR de forma gráfica; o desenvolvimento de um transformador genérico o suficiente para permitir que o Engenheiro de *Software* possa fazer alterações nas regras de transformação através do uso de um editor gráfico orientado pela sintaxe e semântica; e a criação de um metamodelo para o domínio do rastreamento das instâncias dos Metamodelos *Origem* para o *Destino.*

Referências

Andromda (2006). AndroMDA. Disponível em http://www.andromda.org, Dezembro.

Booch, G., Rumbaugh, J., Jacobson, I. (2005) The Unified Modeling Language User Guide (Object Technology). Addison Wesley. 2th Rev Edition.

Czarnecki, K. (2004) "Overview of Generative Software Development". *Unconventional Programming Paradigms (UPP)*, Mont Saint-Michel, France. pp. 313–328.

Czarnecki, K., Antkiewicz, M. Kim, C.H.P., Lau, S., Pietroszek, K. (2005). "Model-Driven Software Product Lines". *ACM SIGPLAN - Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) – Poster Session.* pp. 126-127.

Czarnecki, K., Helsen, S. (2006) "Feature-Based Survey of Model Transformation Approaches". *IBM Systems Journal.* pp. 621-645.

D'Souza, D. F., Wills, A. C., (1999). Objects, Components and Frameworks with UML, *The Cataliysis Approach. Addison-Wesley. USA.*

Gamma, E., Helm, R., Johnson, R. (1995). *Design Patterns: Reusable Object-Oriented Software.* Addison-Wesley.

Gmt (2006). Generative Modeling Technologies (GMT). Eclipse Modeling Project (EMP). Disponível em http://www.eclipse.org/gmt, Novembro.

Kleppe, A., Warmer, J., Bast, W. (2003) MDA Explained, The Model-Driven Architecture: Practice and Promise. Addison Wesley.

Matula, M. (2003). "NetBeans Metadata Repository". *NetBeans Community*. Disponível em http://mdr.netbeans.org/docs.html, Setembro/2006.

Mda (2003). The Model-Driven Architecture - Guide Version 1.0.1, OMG Document: omg/2003-06-01.

Mens, T., Gorp, Van Gorp, P. (2005) "A Taxonomy of Model Transformation and Its Application to Graph Transformation" *Proccedings of the International Workshop on Graph and Model Transformation.* Estônia. pp. 7-23.

Mof (2002). Meta Object Facility 1.4 (MOF) Specification. Object Management Group (OMG). Document formal/2002-04-03.

M2m (2006). Model-to-Model Transformation (M2M). Eclipse Modeling Project (EMP). Disponível em http://www.eclipse.org/proposals/m2m, Novembro.

M2t (2006). Mof to Text Transformation. Specification. Object Management Group (OMG). Document: omg/2006-11-01.

Omg (2004). Unified Modeling Language (UML) Specification, version 1.4. *Object Management Group* (OMG). Document formal/04-07-02.

Omg (2003). Common Warehouse Metamodel (CWM) Version 1.1. *Object Management Group* (OMG). Document formal/2003-03-02.

Paiva, D.M.B., Lucrédio, D., Fortes, R.P.M. (2006) "MVCASE - including design rationale to help modeling in research projects." *XX - Simpósio Brasileiro de Engenharia de Software (XX SBES) - Sessão de Ferramentas.* Florianópolis - SC - Brasil.

Poole, J., Chang, D., Tolbert, D., Mellor, D. (2003) Common Warehouse Metamodel - Developer's Guide. Willey Publishing, Inc.

Qvt (2005). Query/Views/Transformations (QVT). OMG Document: omg/2005-11-01.

Sql (1999). International Organization for Standardization (ISO) & American National Standards Institute (ANSI) - ISO/IEC JTC1/SC32 - ANSI ISO/IEC 9075-2:1999. ISO International Standard. Database Language - SQL - Parte 2: Fundation (SQL-Foundation), 1999.

Stahl, T., Völter, M. (2006) "Model-Driven Software Development – Technology, Enginnering, Management". John Willey and Sons ltda., England.

Tratt, L. (2006). "The MT Model Transformation Language", *Proceedings of ACM Special Interest Group on Applied Computing (SIGAC) - Session: Model transformation*, Dijon, France. pp. 1296-1303.

Varró, D., Varró G., Pataricza, A. (2004) "Generic and Meta-Transformation for Model Transformation Engineering", *Proceedings of the 7th International Conference on Unified Modeling Language,* Lisboa, Portugal. pp. 290-304.

Xmi (2003). XML Metadata Interchange 1.3 (XMI) Specification. Object Management Group (OMG). Document formal/03-05-01.

Xml (2006). Extensible Markup Language (XML). W3C Architecture Domain. Disponível em http://www.w3.org/XML, Dezembro.

Zendulka, J. (2005). "Object-Relational Modeling in UML", Encyclopedia of Database Technologies and Applications, Idea Group Publishing, Hershey, US. pp. 421-426

Palestras convidadas / Invited talks

Moving Architectural Description from Under the Technology Lamppost

Nenad Medvidović, University of Southern California

Abstract

Software architecture description languages (ADLs) were a particularly active research area in the 1990s. In 2000, I co-authored an extensive study of existing ADLs, which has served as a useful reference to software architecture researchers and practitioners. However, the field of software architecture and our understanding of it have undergone a number of changes in the past several years. In particular, the Unified Modeling Language (UML) has gained a lot of popularity and wide adoption, and as a result many of the ADLs I had studied have been pushed into obscurity. In this talk, I will argue that the main reason behind this is that the early ADLs focused almost exclusively on the technological aspects of architecture, and mostly ignored the application domain and business contexts within which software systems, and development organizations, exist. Together, these three concerns - technology, domain, and business - constitute the three lampposts needed to appropriately il luminate software architecture and architectural description. I will use this new framework to evaluate both the languages from my original study, as well as several more recent ADLs (including UML 2.0).

Software Product Lines: Past, Present, and Future

Paul Clements, Software Engineering Institute

Abstract

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. This talk will give a brief introduction to software product lines and highlight some major success stories. Then it will turn to the state of product line research and pose some challenge problems for the future.

Tutoriais convidados / Invited tutorials

Improving a Distributed Software System's Quality of Service via Architecture-Driven Dynamic Redeployment

Nenad Medvidović, University of Southern California

Abstract

The quality of service (QoS) provided by a distributed software system depends on many system parameters, such as network bandwidth, reliability of links, frequencies of software component interactions, and so on. A distributed system's deployment architecture (i.e., the mapping of software components onto hardware hosts) can have a significant impact on its QoS. Furthermore, the deployment architecture will influence user satisfaction, as users typically have varying QoS preferences for the system services they access. Finding a deployment architecture that will maximize the users' overall satisfaction is a challenging, multi-faceted problem. In this talk, I will present: (i) an extensible model of a software system's deployment architecture; (ii) a suite of tailorable algorithms for estimating an improved redeployment; (iii) a visual environment for automatically exploring large numbers of deployment options; and (iv) a runtime infrastructure for effecting the preferred deployments during system runtime. I will discuss the evaluation of this approach on a large number of representative scenarios, including two industrial settings.

Software Product Lines: Essential Practices for Success

Paul Clements, Software Engineering Institute

Abstract

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Companies of all sizes and domains are using the software product line approach to achieve astonishing improvements in time to market, productivity, cost, and quality. This tutorial will introduce the essential activities and practices for success. We will discuss what product lines are (and are not), and the three essential activities for achieving a software product line: core asset development, product development, and management. The tutorial will cover some of the essential practices unique to software product line development, and present some major industrial product line success stories.

Evaluating a Service-Oriented Architecture

Paulo Merson, Software Engineering Institute

Abstract

Are you involved in the development of a service-oriented architecture (SOA), or are you considering this approach in your next project? Do you believe architecture evaluation is an important step in the software life cycle because, among other benefits, it detects problems that are hard to fix once the implementation is in place? If you answered "yes" to both questions, we invite you to attend this tutorial where we'll provide practical information for the architecture evaluation of an SOA system. We'll discuss how to describe the architecture, what architectural approaches are applicable, what questions and design considerations can be used to probe the architecture, and other aspects of architecture analysis-always looking at the specifics of SOA solutions. In this tutorial, the SEI Architecture Tradeoff Analysis Method (ATAM) is used as a reference for the steps, inputs, and outputs of an architecture evaluation technique.

Mini-cursos convidados / Invited short courses

Managing Software Reuse

Cláudia Werner, UFRJ

Abstract

Software Reuse is the discipline responsible for creating new software systems from existing software. This concept goes beyond simple reuse of source code, as other products, such as specifications, projects and test plans may also de reused. The main motivation for software reuse is the inscrease in productivity and quality levels in the software development process. The search for improvements in software quality and the rise of productivity have been widely explored by the software engineering community, in Brazil and abroad. Many organizations have seen in the last years cases of success when using a strategy for reuse in their software development process. The potencial for software reuse may be reached from a well-established Reuse Program, which aims to introduce reuse in a company as a means to make it more competitive, by making its production more agile and allowing the development of quality software products. This short course aims to present Reuse Management, including economic, organizational and personal aspects which may lead a company to reuse sofware in an effective form. We also present existing norms and the reference model MPS.BR which address this subject.

MDA - Patterns, Technologies and Challenges

Glêdson Elias, UFPB

Abstract

This short course makes a critical analysis of MDA (Model-Driven Architecture) and aims to clearly evidence what can be adopted in current software development practices and which are the promises and visions for the future. Initially, this short course approaches basic MDA concepts, such as models, metamodels, model transformations, PIM, PSM and CIM, pointing out the importance of these for developing software based on models. Next, this short course discusses the main technologies currently available, including MOF, UML, XMI, CWM, SPEM and QVT, enumerating available tools, when it is possible. At last, this short course identifies problems and challenges, suggesting evolution and research tendencies in this field.