PhD-FSTC-3-2007

Faculté des Sciences, de la Technologie et de la Communication, Université du Luxembourg
Institut d'Informatique, Université de Namur (FUNDP)

THÈSE

Soutenue le 20/09/2007 à Luxembourg

En vue de l'obtention des grades académiques de

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE
&
DOCTEUR DE L'UNIVERSITÉ DE NAMUR EN SCIENCES

par

Gilles PERROUIN
Né le 30 Novembre 1978 à Rennes, France

ARCHITECTING SOFTWARE SYSTEMS USING MODEL TRANSFORMATIONS AND ARCHITECTURAL FRAMEWORKS

Jury de thèse

Prof. Dr. Pascal BOUVRY, Président, *Université du Luxembourg, Luxembourg*
Prof. Dr. Jean-Luc HAINAUT, Président Suppléant, *Université de Namur (FUNDP), Belgique*
Prof. Dr. Jean-Marc JÉZÉQUEL, Rapporteur, *Université de Rennes I / IRISA, France*
Dr. Frank VAN DER LINDEN, Rapporteur, *Philips Medical Systems, Pays-Bas*
Prof. Dr. Pierre-Yves SCHOBBENS, Expert Invité, *Université de Namur (FUNDP), Belgique*
Dr. Olivier BIBERSTEIN, Expert Invité, *Université des Sciences Appliquées de Berne, Suisse*
Dr. Eric DUBOIS, Expert Invité, *CITI CRP Henri Tudor, Luxembourg*
Prof. Dr. Nicolas GUELFI, directeur de thèse, *Université du Luxembourg, Luxembourg*
Prof. Dr. Patrick HEYMANS, co-directeur de thèse, *Université de Namur (FUNDP), Belgique*

If a man does not know to what port he is steering, no wind is favorable to him.
Seneca, *Roman dramatist, philosopher, & politician (5 BC - 65 AD)*

REMERCIEMENTS [1]

Ayant été surnommé “marin de la thèse” en faisant référence tant à mes loisirs nautiques qu'à ma capacité à faire faire le gros dos face aux tempêtes qu'une université en construction ne manque pas de générer, je vais employer cette métaphore maritime pour remercier tous ceux qui ont m'ont permis de partir et de revenir à bon port. En effet, une thèse est comparable à une circumnavigation de quelques années sur le vaste océan de la recherche ; on espère y trouver un certain nombre de terres vierges ou de portions inexplorées de continents déjà connus et de rentrer au port avec un journal de bord suffisamment détaillé pour qu'un jury de navigateurs chevronnés y trouve intérêt...

Je tiens tout d'abord à remercier mon armateur, le professeur Nicolas Guelfi, qui en supervisant cette aventure m'a offert un grand choix d'océans à explorer sur la planète du génie logiciel, en perpétuelle quête de nouvelles aventures, me permettant de revenir au port avec un bon nombre d'idées pour de futures explorations. Il m'a aussi donné les moyens de ces explorations et a fait en sorte que de nouvelles soient possibles.

Qu'il me soit permis de remercier mes co-armateurs, les professeurs Eric Dubois et Patrick Heymans. Le premier pour avoir initialement suivi l'aventure et avoir validé mes différents points de route. Le second pour avoir si gentiment accepté de reprendre l'aventure en cours de route et d'avoir si fortement contribué à la qualité du journal de bord ainsi que d'avoir copieusement fourni la bibliothèque du bord (qu'il va falloir envisager d'agrandir rapidement...).

Je tiens aussi à remercier les membres jury, qui par leur commentaires et leurs conseils ont permis d'importants amendements au journal de bord. En particulier, les professeurs Jean-luc Hainaut et Pascal Bouvry pour avoir accepté de présider la défense privée à Namur et la publique à Luxembourg, le professeur Jean-Marc Jézéquel pour avoir relu et commenté le journal de bord en détails.
I would like to thank Dr. Frank van der Linden for having accepted to review this thesis.
Je voudrais aussi remercier le professeur Pierre-Yves Schobbens pour avoir gentiment accepté de participer dans le jury à la dernière minute.

Au cours d'une telle navigation, on rencontre aussi un certain nombre de marins plus aguerris qui acceptent d'échanger avec vous un peu de leur science marine. En particulier, je tiens à remercier le professeur Olivier Biberstein pour m'avoir permis de maintenir le cap, dans les calmes comme dans les tempêtes et de m'avoir si chaleureusement accueilli dans sa “taverne” de marin suisse. Je voudrais aussi remercier les docteurs Shane Sendall et Giovanna Di Marzo Serugendo

[1] SOME ACKNOWLEDGMENTS HAVE BEEN WRITTEN IN ENGLISH TO THANK NON FRENCH-SPEAKING PEOPLE.

pour leur contributions au projet FIDJI qui initia cette aventure ainsi que pour leur support.

Il y a aussi les camarades de bordée du projet FIDJI partageant le même armateur mais suivant leur route sur d'autres navires. Je tiens à remercier et à souhaiter bon vent à Benoît Ries et Cédric Pruski : je souhaite que vous aussi puissiez trouver le cap du port très bientôt. Je tiens à remercier Paul Sterges, notre ex-ingénieur en chef pour avoir assuré de manière efficace la logistique et le fonctionnement de la compagnie de navigation durant ce projet.

Je tiens à remercier les "nouveaux" membres de la compagnie de navigation Marcos Da Silveira et Jacques Klein pour leurs intéressantes discussions et leur support. Merci aussi à Jörg Kienzle pour sa "funitude" fort agréable lorsque les nuages noirs de s'amoncelaient et que la rédaction du journal de bord se faisait plus difficile.

I would like to thank my colleagues at the laboratory of advanced systems for interesting discussions and nice social events.

Je tiens aussi à remercier mes amis de mon club d'aviron (la société des régates messines) pour m'avoir appris la subtile différence existant entre galérer et ramer et avoir ainsi contribué a la santé physique et mentale nécessaire à l'accomplissement d'une telle navigation. Merci à mes camarades de train, Elizabeth et Nelly dont le crayon vert a permis de ne pas (trop) insulter Shakespeare...

Je suis redevable de mes amis rencontrés avant cette aventure. Un grand merci au club d'astronomie de Sévérac ainsi qu'a Sébastien en particulier pour m'avoir fait partagé depuis le lycée son goût de la navigation sur les océans de la recherche. Je tiens à remercier mes amis ingénieurs, Rémi et Florence, Geobert et Valérie, Pierre et DP ainsi que Josselin pour m'avoir rappelé qu'il peut parfois y avoir une vie en dehors du quotidien de marin et avoir assuré la logistique de mes escales parisiennes. Maintenant que je touche au but, j'espère avoir l'occasion de vous voir plus souvent autour d'un bon verre.
Je voudrais saluer tout spécialement Chloë qui, telle un phare dans la nuit, m'a éclairé de son humour et de sa joie de vivre. Et tout marin sait à quel point il est réconfortant d'apercevoir un phare quand le vent fraîchit et la mer déferle. Un grand merci à toi.

Enfin, toute cette aventure n'aurait pas été possible sans le soutien de mes parents qui ont toujours veillé à ce que le bateau soit bien entretenu et que je n'affale pas les voiles alors que le découragement m'entrainait vers ses brumes assassines. Un immense merci à vous.

RÉSUMÉ

Les applications logicielles sont devenues indispensables dans un grand nombre d'activités humaines et se sont répandues grâce à l'avènement des réseaux (ADSL, WIFI, GSM...) ou de l'informatique nomade (ordinateurs portables, assistant digitaux personnels, téléphones mobiles etc.) qui ont rendu l'interaction avec des systèmes informatiques possible en presque tout lieu. Cet état de fait a engendré à la fois une grande complexité pour la conception de ces systèmes distribués et de grandes attentes de la part des clients de ces systèmes, préoccupés principalement par les qualités et temps de réalisation des applications logicielles exécutées par ces systèmes ainsi que les coûts qui en découlent. Il est donc nécessaire d'améliorer nos méthodes de développement afin de faire face à ces nouveaux défis.

D'une part, l'ingénierie dirigée par les modèles (IDM), en permettant la description d'applications logicielles à divers niveaux d'abstraction et en générant certains de leurs éléments via la transformation de modèles, s'attaque à la complexité intrinsèque de ces systèmes et réduit leur temps de développement ainsi que celui de maintenance. D'autre part, l'approche de ligne de produits accélère la réutilisation logicielle en proposant le développement d'applications basées sur un ensemble de composants communs dans un domaine déterminé. Ainsi, lorsque ceux-ci sont conçus avec soin, il est possible de satisfaire simultanément des critères de qualité et de temps de développement. Les méthodes qui ont été construites sur le paradigme de ligne de produits se sont principalement attachées à décrire les points communs et les différences parmi les composants qui seront réutilisés par les applications membres de lignes de produits.

Néanmoins, un logiciel doit très souvent répondre à un besoin qui émane d'un utilisateur particulier. Il est donc nécessaire de prendre en compte ses attentes qui sont parfois spécifiques à cet utilisateur et il n'est ni possible ni souhaitable de définir et de supporter celles ci dans l'ensemble de composant réutilisables à partir desquels les applications sont dérivées. Certaines méthodes orientées lignes de produits proposent des approches pour la dérivation de produits par trop restrictives qui excluent de manière indue des produits qui, bien que pouvant être développés à partir des composants de la ligne de produits, n'ont pas été envisagés lors de sa définition, ce qui nous prive donc de la possibilité d'adresser facilement les exigences spécifiques des utilisateurs. Si quelques méthodes reconnaissent la nécessité de prendre en compte ces exigences particulières, elles ne fournissent aucune solution systématique pour intégrer ces exigences dans le cycle de développement d'un produit.
Cette thèse s'attache à la définition d'une méthode plus souple pour le développement d'applications dans le contexte de ligne de produits qui s'appuie sur une combinaison de l'approche IDM avec les "frameworks" orientés-objet. Le domaine ciblé par cette méthode est celui des applications de ventes enchères sur Internet qui est un domaine non-critique pour lequel une grande variabilité dans les scenarii d'utilisation est requise.

La première partie de cette thèse introduit les concepts de base de notre méthode nommée FIDJI. En particulier, nous définissons la notion de *framework architectural* comme un ensemble

de modèles permettant la description cohérente des divers constituants d'analyse et de concept d'une ligne de produits. Cette entité est ensuite instanciée par le biais de transformations de modèles. Nous décrivons ensuite les principes méthodologiques qui ont déterminé les choix des modèles du framework architectural ainsi que l'instanciation flexible de produits encadré par des contraintes définies sur les modèles du framework architectural.

La seconde partie de cette thèse présente en détail les phases de définition des charges, d'analyse et de conception de la méthode FIDJI. Tout d'abord, un modèle de définition de ligne de produits est introduit permettant la définition des charges de manière informelle en se basant sur des variations de cas d'utilisation et un dictionnaire de données et encadré par des règles méthodologiques simples. Nous définissons ensuite la phase d'analyse comme un raffinement de la phase de découverte des charges en proposant la modélisation des concepts du domaine à l'aide de diagrammes UML 2.0 ainsi que l'enrichissement des cas d'utilisation par des expressions OCL (Object Constraint Language). Au niveau de l'analyse, nous démontrons comment un certain degré de flexibilité peut être obtenu au niveau du cycle de vie des évenements échangés entre le système et ses acteurs via l'utilisation de variables d'états. La phase de design s'intéresse principalement à l'aspect architectural, en proposant un modèle de composants basé sur UML 2.0 permettant la description de styles architecturaux structurant le framework architectural. Des profiles UML 2.0 définissant les éléments de modèles utilisés, leurs conditions d'application ainsi que des règles de cohérence et de traceabilité pour les modèles d'analyse et de conception sont proposés. Le processus méthodologique, commun aux phases d'analyse et de conception, consiste en l'écriture d'un programme de transformation de modèles permettant d'instancier le framework architectural tout en étant guidé par des contraintes d'instanciation qui définissent de manière souple les frontières de la ligne de produits.

Enfin, la dernière partie de cette thèse s'intéresse à l'application concrète de la méthode FIDJI. Une étude de cas appartenant au domaine des applications e-commerce est détaillée, illustrant ainsi les modèles FIDJI. Nous montrons en particulier comment écrire le programme de transformations à partir d'opérations de transformation prédéfinies ainsi que la raison d'être et l'utilisation des contraintes guidant le processus d'instanciation au coeur de la méthode.

ABSTRACT

Software systems have become essential to many human activities and have proliferated thanks to various hardware innovations such as mobile computing (laptops, personal digital assistants, mobile phones) and networks (DSL, WIFI, GSM, etc.) enabling interactions between users and computer systems in virtually any place. This situation has created both a great complexity for such distributed systems to be designed and great expectations (mainly concerned with quality, time and induced costs of the software) from the users of these systems, requiring improvements in software engineering methods in order to meet these challenges.

On the one hand, Model Driven Engineering (MDE), by allowing the description of software systems through abstractions and deriving useful system artifacts, harnesses inherent complexity of software systems and reduces time-to-market via model transformations. On the other hand, software product lines foster software reuse by proposing to develop applications based on a set of common assets belonging to a particular domain. Thus, when product line assets are carefully designed, both quality and time-to-market requirements can be achieved. Development methods that have resulted from the product line paradigm generally focus on defining common and variable assets to be reused by product line members. However, they hardly address the development of applications from the product line assets in a systematic way. Furthermore, those considering it propose automated but rather inflexible approaches that unnecessarily exclude products which, although addressable by product line assets, have not been explicitly envisioned during product line definition. If in some domains — in particular, those including hardware constraints and/or critical features — it is possible to fully determine the products that are part of the software product line, in the other cases, an initial set of products can only be considered assuming that the customers' requests will be met by this set. We believe that this assumption is false in general and this thesis examines the research question which consists in proposing a set of models and a product line development method to offer more flexibility while deriving products in order to seamlessly address customers' requests. The domain we consider is that of web e-bartering systems.

This thesis strives to propose a trade-off between automated and unsupported product derivation by providing a model-driven product line development method that allows developers to define product line members by transforming a coherent and layered set of product line models. Moreover, constraints on the possible transformations have to be specified in order to determine which products cannot be derived both for functional and technical reasons.

The first part of this thesis introduces the foundational concepts of our FIDJI method. In particular, it describes the notion of *architectural framework* as a set of models defining product line assets at analysis and design levels and which is instantiated in order to obtain product line members thanks to model transformations. This part then describes key methodological principles driving the choice of architectural framework models and how flexibility in product

derivation can be achieved and controlled by constraints defined over the set of architectural framework models.

The second part of this thesis is devoted to requirements elicitation, analysis and design phases of the method. For requirements elicitation, a specific product line template is defined to allow for the description of a software product line in an informal manner via use case variants and data dictionaries. The analysis phase refines requirements elicitation by allowing the precise description of domain concepts in terms of UML models as well as functionalities in terms of use cases completed by OCL expressions. Variability is ensured through the use of state variables in OCL expressions which enable a wide variety of scenarios to be implemented in the product. Constraints indirectly define product line boundaries by preventing certain instantiations from being made. The design phase focuses on the architectural design of the architectural framework and describes it in terms of interacting components structured via architectural styles. Analysis and design models are supported by UML profiles defining the constructs offered by the FIDJI method, their usage conditions as well as traceability and consistency rules ensuring model correctness. The methodological process for both analysis and design consists in writing a transformation program, validated over the aforementioned constraints, that will instantiate the architectural framework to obtain a viable product line member.

The last part of the thesis deals with the practical application of the method. A case study belonging to the e-commerce domain illustrates the FIDJI method in detail and a simple architectural framework is defined for this purpose. In particular, we show how the transformation program is created from predefined transformation operations dedicated to FIDJI models and the rationale and usage of constraints controlling the instantiation of the architectural framework.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivations

Software engineering is about defining, specifying and resolving interesting (i.e useful) problems using computers. Almost since software engineering emerged, software engineers have had to cope with the famous "software crisis", challenging their abilities to provide satisfactory solutions (i.e. software applications) within a reasonable time. In 1972, Dijkstra gave his views [Dij72] on the problem; at the beginning, programmers' task mainly involved finding tips to take optimal advantage of the hardware then available and it was believed that their problems would be easily solved through more powerful hardware. Such hardware swiftly appeared but things got worse; indeed, due to its sophistication, hardware became more difficult to program efficiently. Thus, reasons for crisis are not only to be found in technology but are also inherent to the problems being addressed. Furthermore, hardware popularization also implies greater expectations from software customers.

Unsurprisingly, in 2007, software engineering is still going through a crisis. Technology offerings and customer needs have kept complicating the definition, specification and implementation of systems, most of which are currently distributed. To address the current version of the "software crisis", solutions are required which meet the following objectives:

- **Reducing software lifecycle time**: The fact that nowadays software is part of every business and is necessary to fulfill its productivity goals raises customer expectations to an unprecedented level. Like any product, it has to be developed on schedule and within the budget initially allocated (software projects seldom respect both schedule and budget constraints) and to fulfill its purpose,
- **Responding to Technological Changes**: Technological changes impact the IT industry more than any other industry due to the constant progress made by semiconductor industry in terms of computing power and miniaturization. Software should be engineered both in order to exploit hardware facilities during its creation and to facilitate migration from one computing platform to another during its maintenance;
- **Accommodating to requirements changes**: Partly as a consequence of the previous point and together with the global competition that forces companies to increase their agility and adaptability, requirements for a given software application constantly evolve;
- **Harnessing software complexity**: As noted by Easterbrook and Nuseibeh [EN04], there is a complex interaction between humans and the software they are using, shaping each other in a non-predictable way. Therefore capturing this interaction is inherently difficult ("complexity of purpose" [EN04]). Furthermore, technological changes have enabled various computing platforms to cooperate in the same information system thanks to network availability. Though it provides a more flexible and visible system to its users (such as

mobile access, VPN), analysis, design, implementation and maintenance are more difficult to perform. And, as stated earlier, these systems are becoming common and crucial to companies' productivity, there is hence an urgent need to improve the ways we deal with this complexity.

Fortunately, 25 years after Dijkstra's analysis, the software engineering community has developed some innovations that are enabling engineers to tame the inherent complexity of modern systems and to develop them more rapidly. Among them, this thesis identifies *model driven engineering*, *software product lines*, *object-oriented frameworks*, *software architecture* and *methods* as mainstays to define an approach capable to address the aforementioned challenges.

As we have seen, the main problem software engineers are faced with is complexity; whether accidental (i.e. related to a particular technology we are using to develop systems) or essential (related to the problem to solve). One possible approach to deal with complexity is the use of models. Models have been used in various engineering disciplines such as civil engineering or anatomy to reason about the system to be built or to be analyzed. Models provide a view of the system from a *certain perspective*, that is they concentrate on some relevant aspects of the system while abstracting others. Therefore, models are easier to read and facilitate the understanding of the system. In contrast to civil engineering, software models can also be used to actually build the system by transforming them from requirements to system implementation. This way of building systems, stemming from formal refinement techniques, is actually emerging in various areas of software under the generic denomination of *Model Driven Engineering* (MDE) thanks to standardized modeling languages such as the Unified Modeling Language (UML) and generic model transformation techniques and tools. MDE-based approaches also promote the notion of "platform independence" as a solution to perpetuate software design decisions with respect to technological changes; abstract models of a system can be freed from any information about the technology that will be actually used to develop the system. Technology-dependent concrete models can be generated thanks to model transformation and via separate models describing the target platform.

The concept of Software Product Line (SPL) is based upon the idea of "product families" introduced by Parnas [Par76]: instead of considering applications individually, we should group them in sets and reuse some of the functionality of the existing members to develop new ones. SPL approaches leverage product families by proposing to develop common assets (that may be factored from previous applications in the set) that are deliberately designed to be reused to form applications. Successful SPL approaches minimize time-to-market by providing the necessary descriptions, process and tools to systematically reuse these assets in an effective manner. Software product lines are developed along two intertwined processes: domain engineering which deals with the elaboration, design and maintenance of reusable assets and application engineering which addresses the reuse of such assets in order to build SPL members.
Since the 90s, several models and development methodologies have been devised to address software product lines. They mainly focus on defining commonalities and variabilities offered by SPL assets. This definition is often mixed with the definition of the assets themselves; methodologically, it imposes that all the variations for the possible SPL members are known beforehand which is a very difficult task and often results in the definition of a huge number of variants which are tricky to manage. Furthermore, this set of variants does not guarantee that it will cover application specific requirements, requiring a SPL evolution. As mentioned above, one of the modern challenges of software engineering is to keep up with requirements changes. Having to change a given SPL according to the needs of a particular application is neither desirable

methodologically (because a SPL must evolve to address the needs that will be shared by a great number of its members, not one) nor realizable in practice without seriously affecting development productivity.
The way a SPL member is obtained from the SPL assets, a process known as *product derivation* [ZJ06], is barely covered by existing SPL approaches. Existing approaches either configure products based on a selection of variants or generate the product via model transformations. Due to the coupling between asset and variant definition, these approaches lack flexibility by covering only the defined variants and place the application developer in a delicate position as he must address features that are not present in the SPL.

Object-Oriented Frameworks, since their definition in 1988 by Ralf Johnson [JF88], have considerably improved software development by providing proved solutions to recurrent problems (patterns) and maximizing application reuse. An object-oriented framework is composed of a number of generic classes implementing core functionalities of a given application area or domain (e.g. numerical calculus, multi-tiered J2EE applications...). These classes are then specialized (using object inheritance) by developers to implement specific functionality of an application pertaining to that domain. Frameworks are accompanied by documentation that can take various forms and usually details the technicalities of each service (its purpose and how to use it). Developing an application with a framework implies having a deep understanding of the framework to be able to specialize the framework efficiently while preserving its quality attributes (portability, performance, security, correctness with respect to its requirements...). However, existing framework documentation does not ease framework understanding because it generally provides information at the detailed design and implementation levels that cannot be exploited easily by software analysts and architects. Indeed, they need to have clear descriptions of the framework's functionalities as well as an overall view of framework's architecture (i.e. its architecture) to assess framework functionalities and qualities regarding those required by the application to be developed with the framework. Moreover, specializing a complex framework manually is a time-consuming process and may violate framework assets if developers do not have a sufficient understanding of the framework architecture.

Software architecture, although being studied since the 60s, has been gaining momentum in both academia and industry for more than 10 years. This momentum is motivated by the aforementioned complexity of modern software such as object-oriented frameworks introduced in the previous paragraph. Software architecture proposes to organize software systems in coarse-grained structures grouping computation in coherent units (components), offering and requiring services to and from other units (ports or interfaces) and together determining the overall system's behavior via well-identified links (connectors). Therefore, software architecture can be seen as an abstraction of its design, thus facilitating its understanding and allowing the early assessment and enforcement of certain qualities of the system through architectural patterns or styles.

Finally, software development methods were introduced to address the aforementioned software complexity and software development time. Indeed, by dividing the work to be accomplished in small chunks, called phases, and by defining clear transitions between them, software development methods guide software development stakeholders (analysts, architects, programmers...) from requirements to maintenance. They allow a better understanding of the job to be done in each phase and a better predictability of the whole development process. Furthermore, development methodologies systematize the development process by providing specific techniques

and tools for each phase resulting in an increased overall efficiency. The key value of a software development method resides in the embedded core paradigm(s) (i.e. SPL, MDE...) and on the way these paradigms are combined in order to provide software engineers with guidelines that are precise enough to enable them to produce quality software and flexible enough to adapt to the constantly evolving requirements issued by customers. Unfortunately, although certain development methods such as Fusion [CAB+94] put the emphasis on the explicit definition of the development phases and on the traceability between artifacts produced, they do not provide the level of adaptability required by software engineers to meet customer expectations. In addition, iterative methods such as RUP [Kru03] define general processes and supplied models require adaptation and expertise to be successful. Besides, as we mentioned above, product line development methods do not really succeed in combining flexibility and support with respect to product derivation.

1.2 Research Problem

The research problem addressed by this thesis is that of addressing complexity and flexibility in product line development. Compared to single product development which is already complex as we have seen above, SPL-based development makes the situation more complicated given that it is necessary to define and manage variability throughout the development of the domain assets and provides ways to resolve this variability and support product analysts and designers while performing application engineering.

More specifically, this thesis addresses the issue of defining a SPL development method with the following characteristics:

- **Simplicity**: The proposed method should offer simple models to facilitate core assets and products elicitation, analysis and design;
- **Flexibility**: The proposed method should be able to handle unforeseen products in an efficient way;
- **Uniformity & Integration**: In order to be supportive enough for application developers, the proposed method should be uniform in the variability mechanism it offers and provide an integrated set of models.

1.3 Development Context

The FIDJI method developed in this thesis is intended for a category of distributed software systems which have motivated the method's founding principles detailed in Chapter 3 and its modeling language detailed in Chapters 4 and 5. This category has the following characteristics:

- **Web-oriented**: The Internet has migrated from the passive presentation of static HTML pages to fully featured applications such as webmails, word-processing, spreadsheets, multimedia applications and Web 2.0 [O'R05] will prompt the development of richer web applications/services. Such kind of applications are characterized by user interfaces (which can be accessed via different devices such as computers, mobile phones...) separated from business logic and distribution at the server level (data servers, application servers, etc.);

- **Customer-oriented**: The FIDJI methodology targets applications that are developed in the context of a contract with a particular customer who will define requirements. This greatly differs from software that is supplied in "boxes" and whose requirements responsibility is totally ensured by the supplier;
- **Reactive**: As a consequence of the first point, the software we consider is reactive [HP85]; it continuously reacts to stimuli sent by actors (human or other systems);
- **Middleware-based**: Web-based applications over dedicated middleware or component platforms such as J2EE [Sun06] and .NET [Mic06a, RR02] which handle some issues related to distribution such as concurrency. Therefore, our modeling language does not take into account such issues since they may be redundant or conflict with middleware on which the architectural framework will be implemented;
- **Non Critical**: Although one of the goals of the method is to model and develop software rigorously, systems considered are not safety-critical (i.e. they cannot endanger lives in case of failure) and, in case of failure, financial consequences will be at a reasonable level (in particular, we consider e-bartering applications that is based on items exchanged rather than real currencies). This corresponds to the state of the practice in web-based application domain;
- **Software Product Line Based**: FIDJI assumes a product line based development approach. This has motivated our notion of architectural framework (see below) and some hypotheses on the development process.

1.4 Thesis Context

The concepts defined in this thesis have been derived from the results of a research project, called FIDJI (scientiFic engIneering of Distributed Java applIcations). This project was carried out at the University of Luxembourg from 2001 to 2004. The main objective of this research was to define a method as well as a dedicated tool support to perform the design and implementation of distributed Java applications. The project's core team was composed of three engineers (Benoît Ries, Paul Sterges and Gilles Perrouin) supervised by Professor Nicolas Guelfi. In addition to this core team, some external experts (Dr. Olivier Biberstein, Dr. Shane Sendall and Dr. Giovanna Di Marzo Serugendo) brought their contribution to the project and the disseminate of helped to disseminate its results. In the late stages of the project, the team was completed by trainees and one engineer assisting in the development of the tool support and case study.

The research method followed during the FIDJI project was to adapt formal refinement [Ser99, SG98] ideas applied to CO-OPN models [Bib97] to a more pragmatic development approach. On the one hand, selecting UML as the main modeling language for the method facilitates the adoption of the method by the software engineering community. On the other hand, thanks to its collaboration (from 2001 to 2003) with Rational (now IBM/Rational) the FIDJI team could evaluate research ideas with respect to real needs and had internal access of their UML case tool (XDE, now replaced by the architect tool suite [IBM06]) to develop tool support for the method.

This research project has contributed in various areas of software engineering in order to define an integrated development approach. The main contribution consists of having defined the notion of *architectural framework* as a combination of object-oriented framework with MDE concepts and

its implementation [GS02b, GR02, GRS03a]. Application development was supported by model transformations for which transformation languages were defined [GPR+03, SPGB03]. Finally, a process [GP02, GP04] was defined to guide developers while applying the FIDJI approach. Gilles Perrouin was particularly involved in transformational and methodological aspects of the research project.

1.5 Contribution

Considering the aforementioned research problem and building on the experience acquired during the FIDJI project, we were able to contribute to SPL-based development at the domain and application engineering levels.

1.5.1 Domain Engineering

This thesis makes the following contributions to the existing state-of-the-art in SPL domain engineering. First, at the requirements elicitation level, we define an informal template devoted to software product lines description, in order to organize variability information at this level; in particular the template proposes to list the variable domain concepts and to describe SPL scenarios with use case variants.

Second, by outlining the natural synergy between software product lines and object-oriented frameworks concepts, we form the notion of *architectural framework* as a layered set of models that describe SPL assets at the requirements analysis (or late requirements) and architecture design levels. The implementation level is achieved by an object-oriented framework though this work does not cover this level explicitly. By leveraging the abstraction level in which object-oriented frameworks are documented, architectural framework models allow to ease the design and to improve the understanding of a particular object-oriented framework. They are also the starting point of application engineering.

Third, we detail the architectural framework concept by providing a metamodel, mainly based on a dedicated UML 2.0 profile, that fully defines the constructs that have to be used in order to model an architectural framework at the requirements analysis and design levels of abstraction. At the analysis level, we refine domain concepts elicited in the aforementioned template in terms of UML 2.0 class diagram. SPL scenarios are detailed under the form of sequences of operations forming single units of behavior and declaratively specified in OCL 2.0. For each use case, a *use case component* is defined in order to show structural elements (domain concepts and UML 2.0 primitive types) necessary to fully specify the use case in OCL 2.0. At the design level, we offer a component-based approach allowing to model the architecture of the architectural framework with predefined architectural styles. This approach uses UML 2.0 composite structures as well as an action language to define detailed design of architectural framework components. FIDJI's modeling language elements are chosen according to the following criteria: simplicity, conciseness, precision and flexibility.

Simplicity has determined the choice of our modeling constructs which stem from widely-used notations (such as use cases to describe requirements) and well-known UML elements. This facilitates a quick adoption of the method by software engineers. Furthermore, conciseness is

achieved by rational selection of a small set of constructs.
The precision issue is tackled using the Object Constraint Language (OCL) which adds the rigor of a logic language to UML 2.0 metamodeling constructs. In particular, OCL is used to clarify analysis scenarios, declaratively specify analysis operations, define component interactions at the design level and to define rules specifying how FIDJI's metamodel shall be used.

As for flexibility, we have defined the following principles:

- **Adaptable Behavior Definition**: SPL asset behavioral definitions at the analysis and design levels are controlled by the use of *state variables* which permit to enforce mandatory behaviors while allowing a complete freedom to analysts and designers in other cases;
- **Separate variability from asset models**: Such a separation allows to focus on the structure/behavior of the SPL assets and make asset reuse in novel ways easier;
- **Defining SPL Boundaries by Restriction**: Rather than striving to specify all the possible variants of SPL assets, we only focus on avoiding those that are considered "harmful" for the SPL either for functional or technical reasons. Such restrictions are defined for each architectural framework layer in a separate model and expressed in terms of OCL constraints.

1.5.2 Application Engineering

As we outlined above, current approaches to SPL development partly fail to support the application engineering process both in a flexible and supportive way. This thesis proposes to remedy this issue via a generative mechanism.

Our approach to application engineering is founded on the instantiation of the architectural framework defined during domain engineering. The architectural framework instantiation process is supported by model transformations and have been designed to meet simplicity, flexibility and methodological support requirements during application engineering.

We achieve the simplicity requirement via two means. First, we benefit from the separation of variability description from asset modeling to allow a direct reuse of asset models. Second, model transformations supporting architectural framework instantiation, declaratively defined in OCL and packaged in libraries dedicated to the transformation (creation, destruction, update) of the model elements conforming to the FIDJI metamodel, are combined in an imperative setting to form an instantiation program. Both FIDJI tailored transformations and well-known imperative syntax of the transformation language proposed contribute to an easier writing of the instantiation program.

We ensure flexibility by letting the product engineers define their own instantiation programs and thus create the products according to product customers' needs rather than relying on rigid domain assets and decision models proposed by a number of traditional SPL approaches. This freedom of action is controlled by a set of instantiation constraints defined at the domain engineering level in order to ensure that the product to be built observes SPL boundaries. In fact, variability resolution simply consists in validating the instantiation program against instantiation constraints.

Finally, we provide a method to assist developers in the instantiation of the architectural framework. This is a waterfall process that defines, for each of the architectural framework layers, tasks that have to be completed in order to construct product models for these layers as well as how transitions can be made between the different application models (analysis and design) according to the architectural framework layers.

1.6 Document Organization

This dissertation is divided in three parts; the first (chapters 2 and 3) is devoted to the basic concepts pertaining to the FIDJI method; the second one (chapters 4 and 5) the method in a detailed manner, while the third part (chapter 6) illustrates FIDJI in practice. Finally, the last part (chapters 7 and 8) concludes the thesis and discusses some open research questions related to the FIDJI method. An overview of each chapter is given below.

Chapter 2 sets out the main concepts (such as object-oriented frameworks, MDE, software product lines) this thesis relies upon as well as development methods that inspired the FIDJI method.

Chapter 3 contains a high-level description of the contribution brought by the FIDJI method. Firstly, the notion of *architectural framework* is defined. Then, a general overview of the method is given, highlighting its objectives and the general instantiation process of an architectural framework.

Chapter 4 delves deeper into FIDJI requirements elicitation and analysis phases. It is divided into three parts:

- The first part defines a template for the description of software product lines at the requirements elicitation level. This template is based on the documentation of domain concepts used and use cases defining the functionalities of the SPL members. A mechanism for documenting commonalities and variabilities is also provided as well as a set of simple methodological tips to write and validate SPL descriptions;
- The second part defines the analysis models proposed by the method. It explains how domain concepts can be precisely defined in terms of UML classes and use cases enriched with UML component diagrams and OCL expressions for a detailed view on the scenarios available to SPL members. It also introduces state variables as the key notion to ensure flexibility in the elaboration of SPL scenarios;
- Finally, the last part is dedicated to the methodological rules governing both the elaboration and use of the FIDJI requirements elicitation and analysis models as well as the activities required in order to instantiate the analysis layer of the architectural framework thereby deriving the analysis of SPL members.

Chapter 5 delineates the FIDJI design phase. In particular, it shows how UML 2.0 constructs can be used to describe the software architecture of an architectural framework according to styles and how to its design is to be detailed in terms of internal structures and behavioral models. We also describe the architectural framework instantiation process at this level and how traceability with the analysis phase is achieved.

Chapter 6 illustrates the FIDJI method in practice. A case study belonging to the e-commerce domain details both models and method usage; in particular, we demonstrate concretely how the product is derived using model transformations and architectural framework instantiation constraints.

The final part of the thesis concludes and outlines future work; Chapter 7 concludes the dissertation and Chapter 8 presents open challenges related to the FIDJI approach and their possible solutions. This includes providing dedicated tool support for modeling and transforming FIDJI models, formalization and perspectives on software product line testing and evolution.

Part I

CONCEPTS

2. BACKGROUND

Abstract

In this chapter we explore the various innovations that have transformed the software engineering field over the last years. These innovations offer the opportunity to present both concepts and related work upon which FIDJI is based. Section 2.1 explains how models can be used to overcome complexity and to reduce development time thanks to transformations. Section 2.2 sketches the latest constructs of the Unified Modeling Language (UML), now in its version 2.0, which will be used as a base for FIDJI models. Section 2.4 describes the concepts and current approaches to product line development and Section 2.5 advocates frameworks as a natural approach to support product lines. Finally, Section 2.6 describes development methods exploiting the aforementioned innovations and which have strongly influenced the FIDJI process.

2.1 Model Driven Engineering

In this Section, we discuss the notion of "Model Driven Engineering" (MDE) [Ken02] [1] as a software development approach that is based on two main "pillars": models and transformation.

2.1.1 On the Power of Models

We mentioned in the introduction that one of the toughest challenges for the software engineering community is the inherent complexity of modern software systems. As noted in [FPB87, GSCK04], there are two types of complexity:

- **Essential Complexity**: essential complexity belongs to the problem being solved; it is related to the number of features used to analyze the problem and the interactions among them. As essential complexity is inherent to the problem, it cannot be reduced or eliminated. However, knowledge in the problem area helps to describe the problem more easily as we will see in Section 2.4;
- **Accidental Complexity**: accidental complexity is related to the solution space; it is dependent on the number of artifacts (code, configuration files, external software such databases etc.) that have to be composed together for the implemented software to solve the problem. Accidental complexity is highly dependent on the technology used to implement the solution: for instance component technologies such as J2EE [Mic05] require the

[1] We define this notion in its general acceptation and independently of any standard/trademark. See "MDE Flavors" for an overview of specific MDE approaches. Model Driven Software Development (MDSD) and Model Driven Development (MDD) are also used as synonyms even if the latter has a more industrial and "OMG-based" connotation.

creation of several interfaces in order to ensure the correct management of components within the application server or assembly languages which may turn out to be impractical to implement graphical user interfaces. Therefore, accidental complexity can be efficiently faced by making the appropriate choices concerning techniques used to develop software.

In both cases, complexity is generated by the great number of elements and their cross-cutting links existing either in the problem space or the solution space. In order to be manageable, we need to reduce this number of elements and focus on the important points rather than details. This is exactly the purpose of *abstraction* [GW92]: "abstraction is the mapping from one representation of a problem to another which preserves certain desirable properties and which reduces complexity". In the solution space, abstractions have been used in the field of programming languages [Sha84] ranging from assembly languages — which in the 1950's gave a simpler view on machine code by organizing it via mnemonic names rather than operation codes — to object-oriented languages that meaningfully help the programmer to organize both structural and behavioral information constituting software. In the problem space, abstractions have been embedded in formal specification languages; examples of such abstractions are sets (VDM [Jon86], Z [Spi92]) or abstract data types which can be described via abstract machines (B [Abr96]) or algebraic specifications (CO-OPN/2 [Bib97], CASL [Egi02]).

In software engineering, an approach, focused on providing ways to effectively represent, define and use abstractions for any part of a software system, is centered on the use of *models*. Several definitions of the notion of model have been given; in [OMG05f], a model "is an abstraction of the physical system, with a certain purpose", in [Sei03], a model "is a set of statements about some system under study (SUS)", finally Bézivin and Gerbé state that "A model is a simplification of a system built with an intended goal in mind" [BG01]. We believe that each of these definitions captures only one aspect of the notion of model and this notion should have a broader interpretation. For example, we share Favre's critical view [Fav04b] about the first definition; a system does not need to be physical to be modeled, other system sorts can be considered and, in this dissertation, we will consider distributed software systems which are non physical. The second definition interprets a statement as "some expression about the SUS that may be considered true or false" which we again find a bit too restrictive so we will consider any expression and not only boolean ones. The last definition considers "simplification" and "abstraction" as synonymous and imposes that a model answers the same questions as the system. In fact, the set of questions addressable by a model is necessarily a subset of the questions that can be asked to the actual system as the model abstracts some details; this abstraction is done with a certain intent (we will call it *purpose*), for example to describe the static organization of the system. As a result we give our view on what a model is, which is in essence a combination of the aforementioned definitions:

Definition 1 (Model) *A model is a set of statements defining an abstraction of a system (or the problem addressed by that system) under study and fulfilling a particular purpose.*

As noted by Bézivin [Béz05], models have been used for ages in various scientific disciplines including biology, economy, house building or geography in which the use of maps is common practice.
The set of statements expressed in a model typically uses a set of predefined modeling constructs and constraints restricting statement usage. It is the role of a *metamodel* to define these constructs (i.e. their abstract syntax) and constraints; without a metamodel, models cannot be built so that they can be validated or processed by tools. For example, the JAVA language

specification [GJSB05] acts as a metamodel; it describes how all JAVA programs (playing the role of models here) have to be written to be valid and transformed in bytecode. According to our definition of model, a metamodel can be defined as follows:

Definition 2 (Metamodel) *A metamodel is a model whose modeled system/problem under study is a set of models and whose purpose is the definition of their abstract syntax.*

Relationships between a system, its representation as a model and the metamodel have been studied in [Fav04a, BBB+05] and summarized in Figure 2.1:

- μ: Designates the "is a model of" relationship. This characterizes the mapping between the system ("the reality") and its abstraction defined in the model,
- $\in$: Is the classical "belongs to" relationship used in set theory,
- χ: This relationship indicates a conformity between a model and its metamodel. A model conforms to its metamodel if and only if it uses the constructs defined by the metamodel and it satisfies the guidelines and constraints applied to them. Or, to put it differently, a model conforms to to a metamodel if and only if it belongs to the set of models this metamodel is a model of.

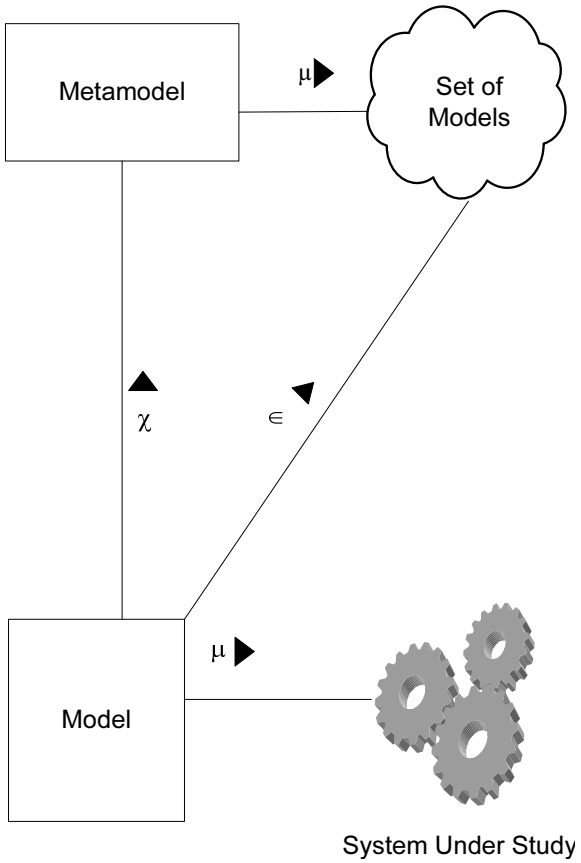

Fig. 2.1: Relationships between Model, Metamodel and System

It should be noted that since metamodels are models, they conform to metamodels (meta-metamodels, indeed) and we can recursively form a hierarchy of models in which each model

plays the role of a metamodel for a set of models and vice versa. There is no universal rule which specifies where this hierarchy should be stopped, hence modeling language designers provide the number of levels they find useful.

Simply providing a metamodel is not sufficient to use models; it also needed to describe what is the meaning of the individual metamodel constructs as well as their composition. This is the role of *semantics* to provide this information. Actually, as explained by Harel and Rumpe [HR04], models semantics is defined by providing a mapping between of the syntactical constructs into a semantic domain. For example, to provide a semantics for the following statement "2+2", it is necessary to map "2" to to the semantics domain of natural numbers and "+" as the function returning the sum (in standard arithmetics) of two naturals. We need this information to be able to interpret model statements for any useful purpose (here computing the result).

As we have already seen, both a metamodel and its associated semantics are required to fully build, validate and understand models. In the following, we will call *modeling language* descriptions that give such information. Depending on their purposes, modeling languages may propose textual or graphical constructs which are formally defined (formal syntax and semantics) or semi-formally defined (well-defined syntax but semantics defined in natural language). The first category is well adapted for reasoning (proofs, model checking etc.) and code generation. The second category is more accessible (as mathematical skills are required to understand them) and yielded general purpose modeling languages such as UML [OMG05f, OMG05d] that will be described in Section 2.2. This semi-formal positioning of the modeling approach is the one that has been chosen for the FIDJI methodology and will be further discussed in Chapter 3.

Models are useful in themselves to describe systems by abstracting them and focusing on some of their aspects. However, alone, they do not distinguish from the current practice for documenting software systems. Founding a software development approach centered on models implies that they are not only considered as description artifacts but that they are also used for its construction; models should be treated as first-class citizens [BFJ+03]. Tough, models are not software systems; they need to be processed by computer-based tools in order to derive other useful models and some of the artifacts composing a real software system. The ability to control this generative process is called *model transformation* and is at the heart of model-driven processes [SK03, GLR+02].

2.1.2 Model Transformation

According to Kleppe et al. [KBW03] the concept of model transformation can be defined as follows: "A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language." Here, it could be noticed, as stated in [MCG04], that a transformation definition may involve several source models and/or target models either to perform model merging (several source models and one target model) or generate multiple models from one (which is quite current when we generate low level artifacts from higher level ones). We give the following definitions:

Definition 3 ((model) transformation) *A transformation is a model characterizing the*

automatic generation of one or more target models from one or more source models.

Definition 4 (transformation operation) *A transformation operation consists in describing how one or more constructs in the source language can be transformed into one or more constructs in the target language.*

Definition 5 (transformation definition) *A transformation definition is a set of transformation operations, assembled together in accordance with a particular transformation language, which describe how one or more source models are transformed into one or more target models.*

Definition 6 (transformation language) *A transformation language is a modeling language for transformation definitions.*

Figure 2.2 summarizes model transformation concepts. Over the years, a number of model transformations approaches have been developed and several taxonomies have been proposed to organize them [SK03, MCG04, CH06]. Our aim here is not to detail all these taxonomies but to provide a coarse description of the various model transformation approaches.

Firstly, model transformations vary in their support for different source and target languages. *Endogenous* transformations are processing models defined with the same modeling language while *exogenous* transformations link different source and target languages. Another distinction that should be made is between *vertical* and *horizontal* transformations: the former processes models at different abstraction levels (e.g. consider a transformation that transforms a design model into code). The latter operates at the same abstraction level (refactoring is a typical example). As noted by Mens et al. [MCG04], there is no correlation between endogenous/exogenous and vertical/horizontal; we can change the abstraction level within the same language (refinement) and keep the same abstraction level while changing language (language translation).

Secondly, model transformations vary in the approach followed by the language in which these transformations are defined. The three types of approaches *declarative*, *imperative* and *hybrid*.

Declarative approaches to model transformation are based on the following principle: transformation rules are composed of two main parts; Left Hand-Side (LHS) and Right-Hand Side (RHS). The LHS describes the elements of the source language (that need to be bound to specific model elements while applying a concrete transformation) which will be processed by the rule. The RHS describes the elements that will replace in the target language the ones defined in the LHS. This category is supported by three mechanisms:

- **Functional programming**: Any transformation can be regarded as function that produces some result from some input. There are few approaches supporting this mechanism, including UMLAUT [HJGP99] which has been discontinued for more imperative transformation approaches and metamodel transformation experiments with F# [BCRP05], a functional language inspired from Caml [Ler05] and ported to the Microsoft's .NET platform [Mic06a];

- **Logic programming**: Logic programming languages have interesting functionalities with respect to model transformation, such as query mechanisms, backtracking or constraint propagation. Logic programming-based approaches define transformations in terms of formulas satisfaction. If source model elements can satisfy premises (LHS) of a set of formulas, they are matched and processed so that they comply with the logical consequences (RHS) of these formulas. General purpose logic languages such as PROLOG have been used [Whi02, GLR+02] but some approaches [PVJ02, CMSD04] uses the Object Constraint Language (OCL) [OMG05e] (see Section 2.2.6) to specify model transformation (see Section 2.2) in a declarative manner [2];
- **Graph transformation**: Graph transformation approaches follow a scheme similar to logic programming but use graphs instead of formulas; the binding of the LHS is based on graph pattern matching and matched elements are replaced by the graph defined in the RHS part of the rule. Graph transformation has a well-founded theoretical background [Roz97] and several implementations exist both in academics and industry; ATOM^3 [dLV02], VIATRA2 [BV06], FUJABA [NNZ00], BOTL [BM03], MOLA [KBC05] and GREAT [AKL03] to name but a few.

Declarative approaches have sound theoretical bases (lambda calculus for functional programming, first-order logic or graph theory) and some of them are now mature (graph transformation especially). However, two issues arise with regard to declarative approaches. First, declarative approaches are usually non-deterministic; several parts may appear to match the LHS and, by default, transformation rules can be executed in any order. Graph transformation approaches (FUJABA, MOLA, VIATRA, GREAT) have been complemented with means for defining transformation sequencing visually. However, they do not scale properly in the case when transformation rules have to be combined in nested loops. The second issue is related with the technical skills required to define transformations; because of their theoretical background, functional and logic programming are not very popular amongst software engineers who tend to prefer object-oriented and imperative languages.

Imperative approaches concentrate on building the target model (or language) rather than characterizing it. This involves writing transformations in general purpose programming languages and libraries provided by tool vendors; Compuware OptimalJ [Com06] or IBM Rational Architect [IBM06] are examples of such approaches. Other approaches include the addition of imperative constructs to OCL; MTL [VJ04] and its successor Kermeta [MFJ05] fall in this category. Imperative approaches are generally more accessible to software engineers since they use a similar syntax as the programming languages they know. However, general purpose programming languages sometimes provide too low-level constructs for model manipulation and transformation, yielding long transformation programs. Imperative OCL-based approaches cope with this problem since OCL natively supports model navigation.

Finally, recent model transformation languages recognizing the qualities and drawbacks of both declarative and imperative approaches propose to mix the above categories. Much effort has been devoted to supporting declarative and imperative styles in the Object Management Group (OMG) standard for model transformation called Query/View/Transformations (QVT being shortened for) [OMG05b] (see Section 2.2). Indeed, hybrid rules permit to add imperative code in the RHS to cope with the issue of specifying declaratively the target model completely. The

[2] Note that in this case, OCL needs to be combined with a logical language such as PROLOG or supported in a graph transformation engine to be implemented declaratively. Otherwise it should be seen as "sugar" at the syntactic level.

hybrid approach to model transformation lies at the heart of the FIDJI approach for product derivation (see Section 2.4) and will be presented in Chapter 3.

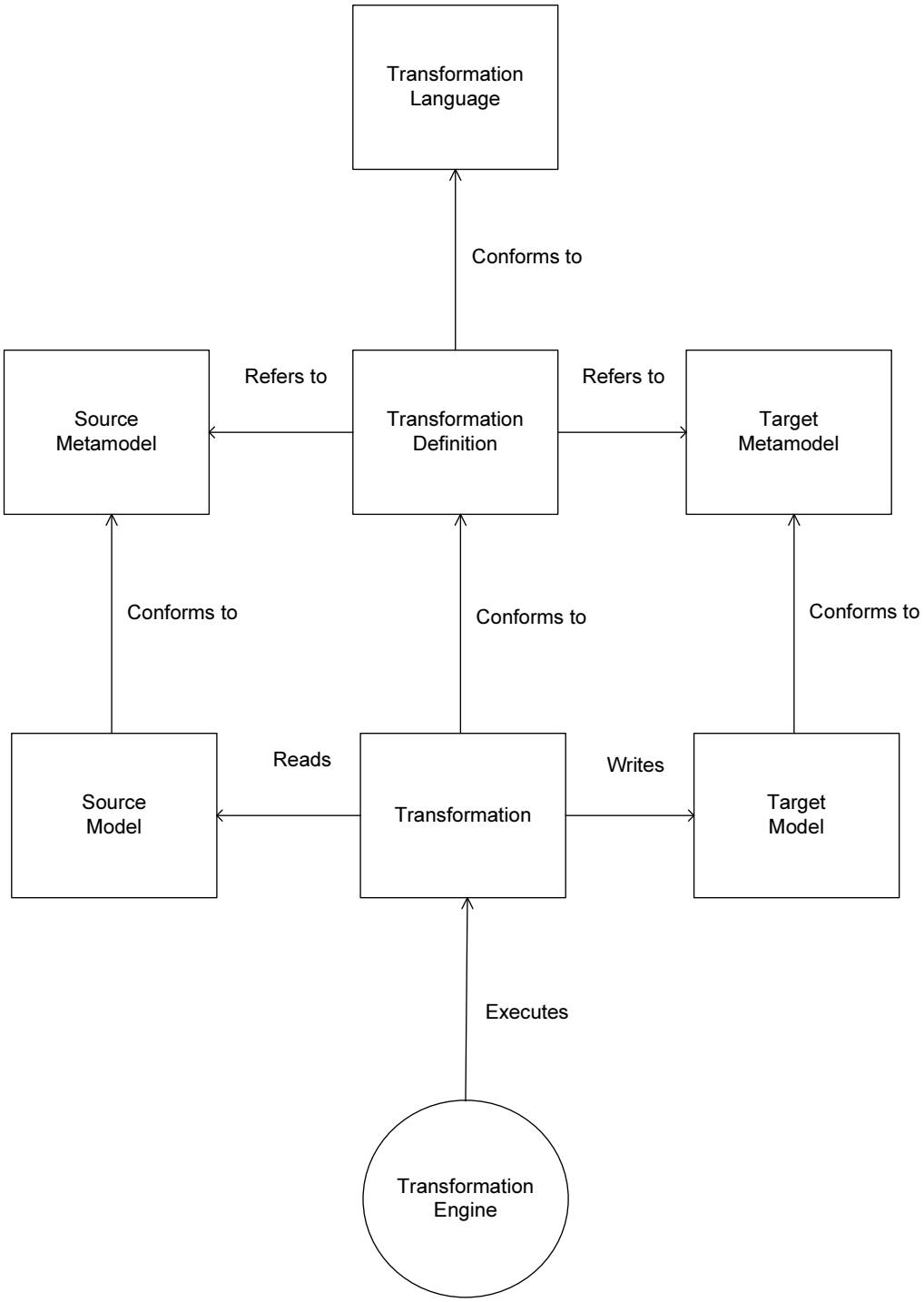

Fig. 2.2: Model Transformation Concepts (adapted from [CH06])

2.1.3 MDE Approaches

In the following we summarize some of the most popular model driven engineering approaches which have had a significant impact with regard to popularizing MDE concepts in the software engineering field.

Model Integrated Computing (MIC)

Introduced in 1997 by Sztipanovits et al [SK97], MIC is centered on the development of Domain Specific Modeling Languages (DSML); these languages define modeling constructs that are tailored to the specific needs of a particular domain. For example, a car manufacturer may be interested in modeling the assembly process in order to optimize plant throughput or military applications that must comply with hard real-time constraints. The MIC comprises the following steps:

- Designing a Domain Specific Modeling Language (DSML): this step allows engineers to create a language that is tailored to the specific needs of the application domain. Tools able to interpret instances of this language (i.e. models of the application) must also be created;
- This language is then used to construct domain models of the application;
- Lastly, the transformation tool interprets domain models to build executable models for a specific target platform,

This approach is currently supported by a modeling environment [AKL03] including a tool for designing DSMLs (GME) and a model transformation engine based on graph transformations (GREAT).
Another approach builds on the same idea: *multi-paradigm modeling*. It consists in integrating different modeling languages in order to provide an accurate description of complex systems and simulate them. The approach is supported by the ATOM^3 graph transformation engine [VdLM02].

Software Factories

"Software Factories" [GSCK04] represent the Microsoft's view on MDE. They are built upon the "factory" metaphor in which development can be industrialized using a well-organized chain of software development tools enabling product generation by adapting and assembling standardized parts. A software factory use a combination of DSMLs to model systems at various abstraction levels and provide transformations between these levels. Naturally, the languages and tools required for efficient development vary depending on the application type considered; this is why software factories adopt a development approach based on software product lines (see Section 2.4). A particular software factory will be tailored to support the development of a well-identified set of products.

All the activities of a software factory are defined with respect to what is called a *software factory schema*. A software factory schema is a directed graph where the nodes represent particular aspects (called viewpoints) of the system to be modeled and edges represent the transformations

occuring between them. In particular, viewpoints provide the definition of the DSMLs to be used to create model of the viewpoints, development processes supporting model creation, model refactoring patterns, constraints imposed by other viewpoints (that help ensuring consistency between viewpoints) and finally any artifact assisting the developer in the implementation of models based on viewpoints. The transformations between viewpoints are mostly supported in an hybrid or imperative way through templates, recipes and wizards that are integrated as extensions to the Visual Studio .NET 2005 CASE tool [Mic06b]. Though appealing, the definition of a different language to model each particular concern of a system raises problems in terms of consistency, especially with respect to semantics [BDP07].

Model Driven Architecture (MDA)

Model Driven Architecture was introduced in 2000 by the OMG [SO00] in order to cope with the so-called "middleware proliferation problem": various middleware implementations based on different component technologies (.NET [Mic06a, RR02], J2EE [Sun06] and CORBA [OMG06a]) are available today. The choice for a particular technology is not immutable and some environmental changes may require that an application be moved from one platform to another, generating high maintenance and evolution costs and affecting design decisions made for a particular *platform*. According to MDA terminology, a platform is "a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented" [OMG03b]. A platform can be generic, e.g. a platform providing object orientation, or specific, such as CORBA or .NET. MDA proposes a three-layer modeling architecture applying a strict separation of concerns in order to capitalize design decisions at a higher abstraction level:

- **CIM**: Computation Independent Model. This model is used to define the business concepts that are currently in place within a particular company and which will be supported by the software to develop. Thus, it is not a model of the software itself but it enables to elicit its requirements by providing information on its environment and explicit shared vocabulary useful in the domain;

- **PIM**: Platform Independent Model. This model gives the description of the working system while abstracting technology-related issues. This model is hence designed to be reusable amongst several platforms;

- **Platform Model**: Details a particular platform in order to implement applications with or on top of it.

- **PSM**: Platform Specific Model. The platform specific model gives details of the executable system. This is the result of one or more transformation(s) taking a PIM and a platform model as inputs.

The above models are intended to be implemented with OMG standards including UML [OMG05f] (see Chapter 2.2) that acts as general purpose modeling notation (possibly extended to describe PSMs) XMI [OMG05a] to enable model interoperability amongst various CASE tools, and the recently adopted QVT [OMG05b] for model transformations.

2.1.4 Model Traceability

The complete description of a software system may comprise numerous models, expressed in different languages, covering all the phases of its development from early requirements (natural language documents) to implementation (code). When developing a system, it is important to provide information about how these models relate to each other and means to follow one of system's features from its description as a requirement to its implementation. Such means is known as *Traceability* [Pal97]. Thus, traceability amongst models is essential to ensure that design artifacts validate their specifications to preserve consistency amongst models and to support software evolution.
Traceability approaches use three techniques: links, rules and model transformation.

Links

In [RJ01], Ramesh and Jarke performed an in-depth analysis of requirements traceability practice across several companies. In particular, they distinguish low-end users who are primarily interested in tracing requirements for compliance verification and change management from high-end ones who capture discussions, decisions and rationale relevant for requirements specifications. Based on this distinction, they provided metamodels for both kinds of users. In [Let02], Letelier proposes a metamodeling approach for requirements traceability based on UML. This metamodel allows to relate requirements (either in textual form or as UML use cases) to test specifications or other UML models. Traceability links are typed and support the dimensions of traceability identified by Ramesh and Jarke (requirements, rationale, allocation and test); `modifies` and `responsibleOf` identify the stakeholder in charge of the element, `validatedBy` attaches a requirement specification to a test specification, while `verifiedBy` relates a test specification to a UML specification that should comply to this test. Finally, `assignedTo` determines the UML model that satisfies a requirement. Other metamodels for traceability have also been proposed [Dic02, Kel05].

Pons and Kustsche [PK04] propose an approach focused on formalizing traceability links in the context of refinement of UML models. Their approach applies to UML classes and use cases. They basically consider two types of links: specialization and decomposition. Links formalization is carried out by means of OCL pre/postconditions (see Section 2.2) and supported by a case tool embedded in the Eclipse environment.

Rules

In [ZSPMK02], a rule-based approach for relating commercial texts, use case structured texts, object models is presented. Rules are described in XML [BPSM$^+$06] and are used to automatically generate traceability links that match these rules. Traceability link types permit to match overlapping elements (in relation to the same feature), to create a commercial statement via use cases or to require a particular feature in the object model.
In [PG96], a formal graph-based approach is embedded in a tool called TOOR. In contrast with commercial requirements tracing tools [Tel07, IBM07], TOOR allow the user to define an axiomatic semantics with new link types.

Model Transformation

Model transformation is another means to ensure traceability amongst evolving models. In [Jou05], Jouault distinguished internal traceability from external one. Internal traceability is established in order to perform the actual transformation and does not persist after its execution. External traceability is explicitly kept after the transformation has been completted. He then provided a mechanism to generate a traceability model (conforming to a simple metamodel) by means of additional rules expressed in the ATL [JK05] transformation language. QVT [OMG05b] (see Section 2.2) models traceability implicitly at the declarative level and in terms of trace classes at the imperative level. More complete discussions about traceability and MDE can be found in [CH06, ARNRSG06].

2.1.5 Model Consistency

As we have seen, models are used to describe a particular viewpoint (this will be discussed further in Section 2.3) or aspect of a system and in dependence thereon, constructs offered by its metamodel may vary. DSMLs have promoted this idea to its extremes but even in approaches where one single language is used, several viewpoints have to be modeled (structural, behavioral, physical etc.). In such a context, the problem of *consistency* inside and between models is unavoidable. The concept of consistency has its roots in formal methods and can be defined as the fact that all statements regarding the system are true at the same time or, more simply, there is no contradiction.

Consistency Classification

In [EB04], Elaasar and Briand provided a classification of consistency types. They identified the following categories:

- **Syntax vs. Semantics**: Syntactic rules can be expressed in a formal language relating metamodel elements. Semantic rules refer to the meaning of elements. As noted in [SC02], a syntactic rule has a meaning that is induced by the semantics of the language primitives and their composition (which may be formal if the syntactic rule is expressed in a formal language);
- **Static vs. Dynamic**: Static rules can be verified without executing or interpreting the model, otherwise they are dynamic;
- **Intra-model vs. Inter-model**: Intra-model rules refer to the syntax of model elements with respect to their metamodel definitions (conformance relationship). Inter-model syntactic rules are defined between different models and can take the form of traceability links or model transformations. Naturally, the same distinction can be made at the semantic level;
- **Multi-level**: Consistency rules can also be organized according to their source abstraction level: modeling language syntax and semantics, extension of the language, methodology, etc.;
- **Nature of Error**: Finally, this last category deals with the different issues covered by consistency rules. It can be a contradiction, an incompleteness or an ambiguity.

Detection and Resolution of Inconsistencies

Syntactic rules need to be defined to automatically detect and resolve inconsistencies in models. Three paths can be followed:

- **Formalizing Semantics**: As we have seen, semi-formal languages have a well-defined syntax but variable or imprecise semantics. In order to express consistency rules with more accuracy, (a part of) the language semantics can be formalized. However, rules are highly dependent on formalization completeness and on the semantics used;
- **Using a Constraint Language**: Another approach consists in using a formal constraint language able to express consistency rules. This allows to rely on the semantics provided by this formal constraint language for the purpose of the consistency rules and does not force a complete formalization of the modeling language to which consistency rules apply;
- **Translating Models in a Formal Language**: The last path to follow consists in translating models conforming to the semi-formal language into formal models and in expressing rules as described in the first path.

In the FIDJI method, our approach to model consistency is based on the second category for its simplicity and flexibility. Our consistency rules are syntactic (expressed in OCL) are relevant for intra/inter models, and cover various levels such as the extension of the UML metamodel on which FIDJI relies and the method. In addition, some consistency rules are also defined in order to ensure traceability between FIDJI models. We will detail them in dedicated sections of Chapters 4 and 5.

2.1.6 Model Impact Analysis

Impact analysis has been defined by Bohner et al. [BA96] as "the process of identifying the potential consequences (side-effects) of a change, and estimating what needs to be modified to accomplish a change". In [BLY06], Briand et al. introduced two categories of impact analysis: Horizontal Impact Analysis (HIA) and Vertical Impact Analysis (VIA). HIA corresponds to changes at a particular abstraction level and has been addressed by the authors in earlier works [BLO03, BLOS06] while VIA is focused on impact analysis across different abstraction levels. HIA and VIA are related to each other; a horizontal change performed on a particular abstraction level impacts elements both at the same level and at a lower level.

Naturally, to be able to estimate the consequences of a model transformation applying to an element, impacted elements must be identifiable. Consequently, model traceability is a prerequisite to model impact analysis. In order to support VIA, Briand et al. have first introduced a new metamodel for traceability in UML. This metamodel particularly defines the `Refinement` metaclass which is central to the vertical impact analysis reasoning. Refinements are conforming to the template consisting of a general description of the refinement and user intent, a list of required atomic changes, constraints and traceability links in OCL. The impact analysis strategy is as follows: it uses two models (the original and the refined one), identifies the refinements on the basis of the differences between the two models and uses the definition of traceability links in the refinement templates to generate the list of impacted model elements. A similar approach is applied to perform HIA.

However it shall be noted that both HIA and VIA depend on the taxonomy of refinement or changes; therefore thy must be applied on a reasonable number of case studies in order to obtain an efficient set of changes that can be applied to a given model. Furthermore, the set containing all the impacted elements within a model may be huge; they can be grouped according to a notion of distance [BLOS06].

The FIDJI method focuses on controlling HIA during the product derivation process via constraints. These constraints inhibit product derivations whose an impact on the models offered by the SPL is too important, or represent consistency rules that have to be satisfied by the newly derived models. We will give more details on HIA in Chapters 4 and 5.

2.2 Unified Modeling Language

In this Section, we introduce the Unified Modeling Language (UML) [OMG05f], which is the most widely used modeling language in the software engineering community. We also describe the Object Constraint Language [OMG05e] (OCL) which allows to clarify UML semantics.

2.2.1 Introduction

The UML is the result of a standardization effort that begun in 1994 when Grady Booch and Jim Rumbaugh started to unify their respective object-oriented modeling methods: Booch [Boo94] and Object Modeling Technique (OMT) [RBP+91]. Later in 1995, Ivar Jacobson added his work on Object Oriented Software Engineering (OOSE) [JCJO92] method. This standardization effort was motivated by the desire to federate the development of these notations that were independently evolving towards the same ideas and trying to stabilize the marketplace in order to improve tool support. The first unified versions — UML 0.9 and 0.91 — were released in 1996, and under the aegis of the Object Management Group (OMG), a request for proposal was issued. Finally, at the end of 1997, the first UML specification (1.0) was adopted by the OMG. Since, it has evolved along several revisions into the current 2.1.1 [OMG07b] version [3] adopted in February 2007. In this section, we will focus on this last version.

2.2.2 Metamodel

The UML specification is based on a four-layer object-oriented metamodeling hierarchy which is depicted in Figure 2.3:

- **M3 or MOF**: The M3 layer is called metametamodel. The Meta-Object Facility (MOF) [OMG06b] is a model manipulation framework designed to define modeling languages. The MOF is comprised of a set of modeling constructs shared by a variety of languages (CWM [OMG03a],UML) and a library of model manipulation operations facilitating the definition of new languages and model handling in CASE tools;
- **M2**: The M2 layer includes the UML metamodel. It is comprised of two parts, i.e. the infrastructure [OMG07a] whose constructs are directly adapted from the MOF and the superstructure embedding concrete syntax of constructs used to form actual application models;
- **M1**: The M1 layer is composed of models defined by users;
- **M0**: M0 elements are objects as present in the memory of a computer running the system.

Note `instanceOf` relationships between layers; indeed, they correspond to a particular example of the χ relationship (see Section 2.1) since an instance of an element belonging to a particular layer has to conform to its metaelement (characteristics, properties, constraints...) defined in the immediately upper layer. Throughout this dissertation, we will be particularly interested in using and extending metaclasses at the M2 level.

[3] Throughout this dissertation, we refer to "UML 2" when speaking about features that are common to all versions of the specification since the 2.0 release and to "UML 1.x" when concerning previous revisions. Referring to the language in the general, the term UML will be used. When we need to refer to a construct in a specific version, we mention its version number explicitly.

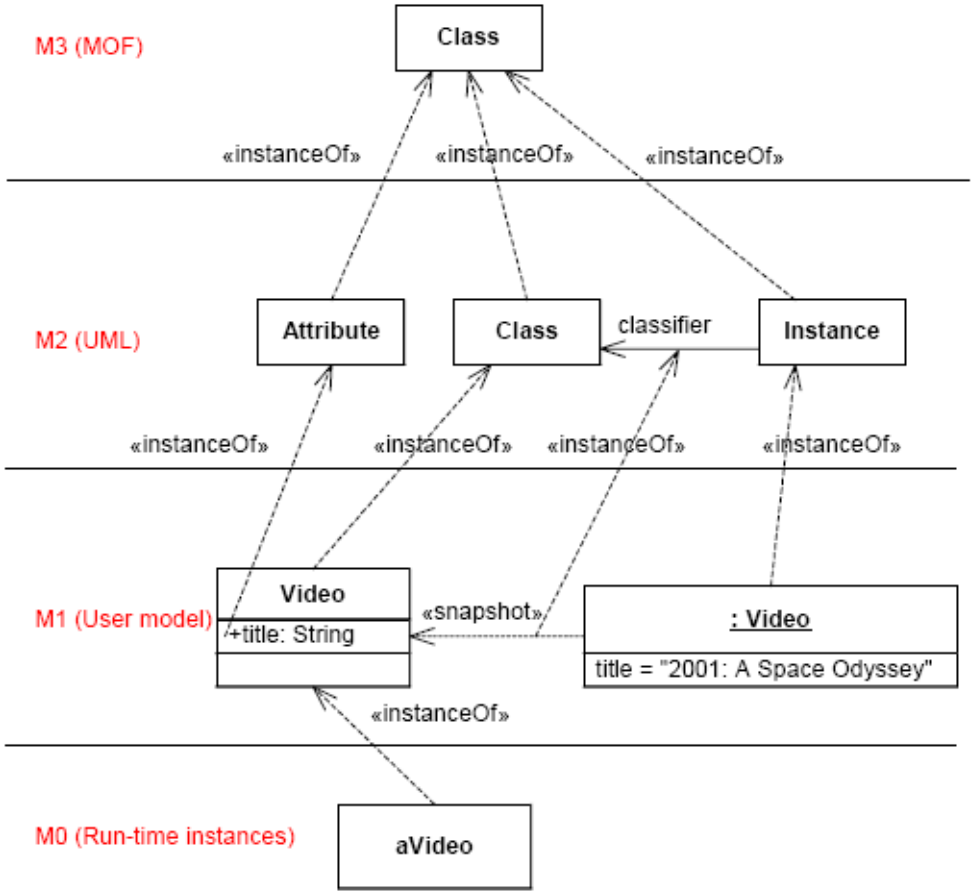

Fig. 2.3: UML Four-Layer Metamodeling Hierarchy (from [OMG07a])

2.2.3 Overview

The UML 2 specification is comprised of loosely related packages of constructs focusing on a particular aspect of the system definition. This package decomposition aims at providing modularity to modelers since they only have to focus on the constructs in one given package — and possibly to import depending packages — according to their needs. Figure 2.4 gives the top level packages of the specification. Some of the most important ones are presented below:

- **Classes**: This package contains the most basic constructs of the language. In particular, it includes `Class` and `Classifier` metaclasses as the fundamental modeling concepts in UML and all the necessary constructs to form an object-oriented hierarchy (whole/part relationship, inheritance, etc.);

- **Use Cases**: This package provides the necessary concepts to model use cases [JCJO92] in UML and their relationships (see Section 2.2.5 below);

- **Components**: The "Components" package extends and improves the UML 1.x notion of `Component`. More specifically, it permits to define systems at a high abstraction level as a "wiring" of components connected together via provided and required interfaces;

- **Composite Structures**: This package extends the above notion of components by offering them the possibility to own an internal structure detailing their structure and behavior;
- **Interactions**: This package provides constructs to model information exchange between model elements. The most used constructs are probably those allowing to define sequence diagrams which are a sophisticated version of Message Sequence Charts (MSCs) [IT04].

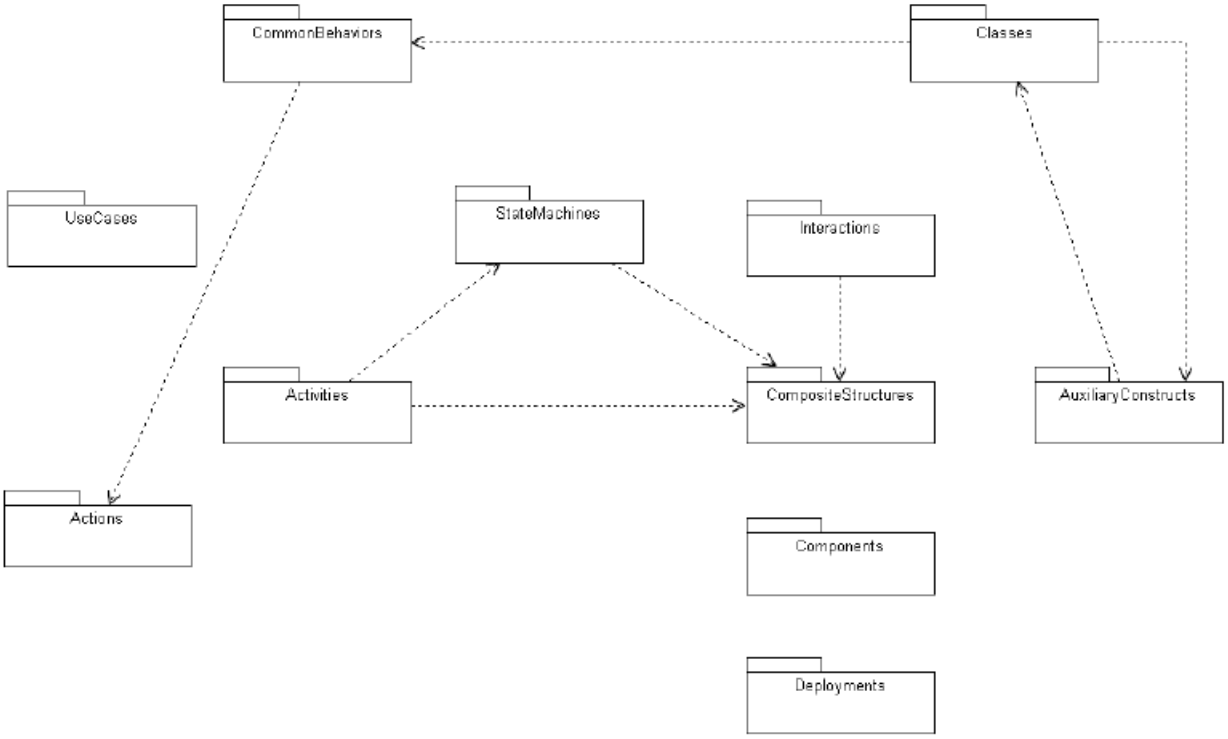

Fig. 2.4: UML 2.1.1 Packages (from [OMG07a])

In the following, we describe the UML 2 metaclasses that are of interest for the FIDJI method developed in this thesis. We will discuss them in more detail later when needed.

2.2.4 Structural Modeling

As mentioned above, constructs for structural modeling are defined in the `Classes`, `Components` and `Composite structures` packages. Constructs contained in the `Classes` package are used to define class diagrams, which are widely known and used to design systems therefore, these constructs will not be detailed here. Instead, we prefer to focus on the innovations offered by UML 2 to model software architectures and in particular to the component-and-connector viewpoint, which will be presented in Section 2.3:

- **Components**: Components are specializations of classes and therefore have attributes and operations, but are also associated with provided and required interfaces. Components are also allowed to have an internal structure comprised of **properties** that in turn describe sets of instances of particular classes. Finally, components may own ports that formalize their interaction points;

- **Connectors**: Connectors which are either *assembly connectors* that connect the required interface of a first component to the provided interface of a second component, or *delegation connectors* that link the ports of a component to its internal parts;
- **Interfaces**: Interfaces can be considered as contracts that components must comply with. An interface is either *provided* or *required*. It is provided when it describes a set of functionalities offered by a component. It is required when it describes a set of functionalities that a component expects from its environment;
- **Ports**: Ports specify a distinct interaction point between the component it belongs to and its environment or between the (behavior of the) component and its internal parts. Ports may specify required and provided interfaces for the component in which they are defined. A behavior port is a special port type that sends all the incoming requests to the classifier in which the port resides, rather than to its internal parts;
- **Classes**: Classes represent the constituents which form the internal structure of components. These are not used in general-purpose class diagrams, but in composite structure diagrams, showing how the required and provided interfaces of a component delegate to or from its internal parts via the corresponding ports. Usually, the composite structure diagrams do not contain the classes themselves, but sets regrouping their instances in the form of properties.

A significant part of the models offered by the FIDJI method is based either directly or indirectly (by extending the metamodel, as specified in Section 2.2.7) on these constructs. Chapters 4 and 5 will define their purpose and use in detail.

2.2.5 Behavioral Modeling

The UML 2 specification offer a wide range of constructs for behavioral modeling which are organized in the packages shown on Figure 2.5.

Use Cases

Use cases were introduced by Jacobson [JCJO92] in order to elicit software requirements in a way that is meaningful for non-technical stakeholders. The main idea is to describe informally the usage of the system with respect to a given goal. This usage is described in terms of interactions (or scenarios specifying information exchanged) between one or more *actors* and the system. We will describe in Section 2.4 how these scenarios can be captured in the context of software product lines.

The UML notation focuses on the definition of actors, use case goals and relationships between use cases. Figure 2.6 shows a concrete example. Use cases are represented by ovals in which use case goals are given as a short active verb phrase. Actors are represented by stylized men whose names refer to the roles they play in the use cases. Associations assign actors to multiple use cases and define their multiplicities. Finally, relationships between use cases can be of the following types:

- `Extend`: An extension is used when a behavior defined in a another use case can be added, possibly conditionally, to a given use case. Use case extension is shown as an arrow with the <<`extend`>> keyword;

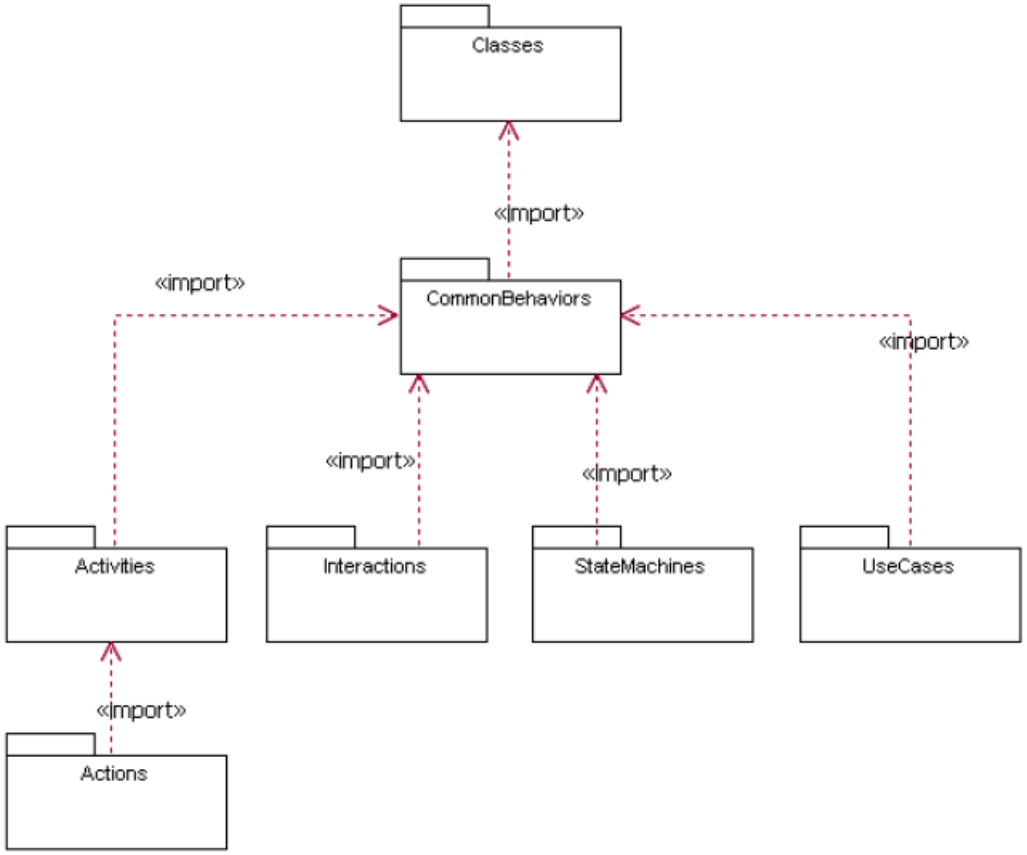

Fig. 2.5: UML Behavioral Packages

- `Include`: The inclusion relationship is used when the behavior of the included use case is embedded in the behavior of the including use case. It is shown as an arrow with the <<`include`>> keyword;
- `Generalization`: Generalization can be used between actors (with the same semantics as that defined for classes) or between use cases, where elements of the general use case are available to specify the behavior of each specific use case. It uses the same notation as for generalization defined for structural elements, represented by a line with a hollow triangle pointing on the general element.

FIDJI uses a restricted version of use case diagrams because of the unclear semantics of some of its relationships. This will be discussed in Chapter 4.

Interactions

Interactions are used to define interprocess communication between structural elements (such as classes or components). They may also be used to define scenarios for use cases. One key construct to model interactions is `Message`. A message can refer to an operation call on a class, data sent asynchronously (instances of `Signal`) or relate to the creation or destruction of elements. Messages are defined between lifelines (instances of `Lifeline` metaclass) which represent individual participants in the interactions. Within an interaction, the message sequence

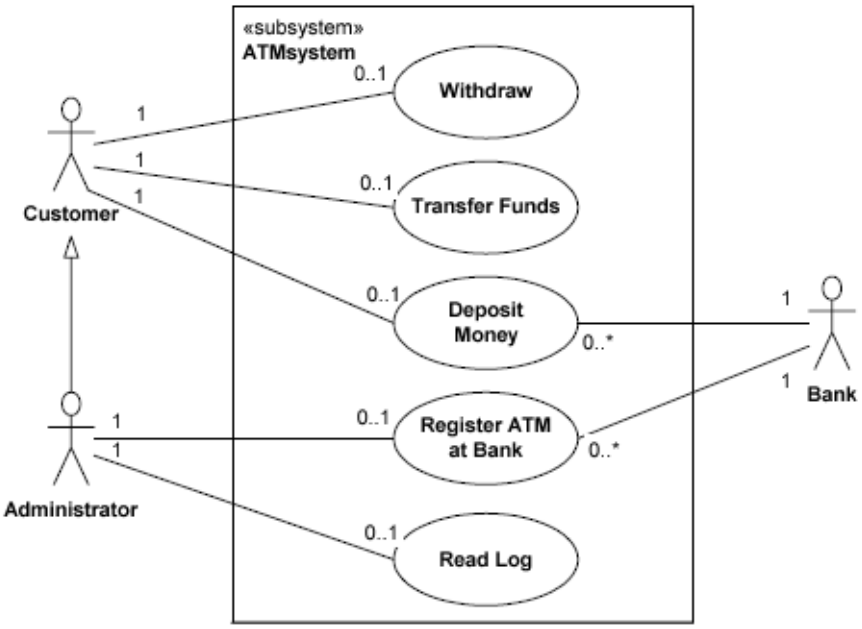

Fig. 2.6: Use Cases Diagram Example (Source: UML 2.0 Specification)

is of particular importance as it helps to understand the situation. For that reason, messages are generally numbered or laid out in such a way that the message sequence appears clearly. From the semantics perspective, UML 2 interaction can be formalized using a trace-based semantics [Har03]. From the syntactic perspective, UML 2 offers two major diagrams to define interactions: sequence diagrams and communication diagrams. Sequence diagrams are extensions of Message Sequence Charts (MSC) [IT04]; in particular, a sequence diagram allows to model variations in the message sequence via dedicated operators applying on the messages and defined within an instance of `CombinedFragment`. These operators allow to model alternatives and optional messages, loops, unauthorized messages, strict and weak sequencing and few others. An example of such a construct is shown in Figure 2.7: depending on the value of variable `x`, two message sequences are possible.

Communication diagrams (called collaboration diagrams in UML 1.x) can be thought as a simplified form of sequence diagrams that do not use combined fragments. An example is shown on Figure 2.8.

The UML 2 specification notices that, typically, interactions do not model the "complete story". Indeed, there are some legal traces that may not be contained in the interaction models. However, as stated in the specification, some projects require that all the traces be modeled, which is unrealistic from our point of view. Furthermore, even though rich graphical notations for scenarios such as those proposed by UML in sequence diagrams or Damm and Harel's Life Sequence Charts (LSCs) [DH01] are expressive enough to precisely define the major interactions (and undesired ones), provided notation is rather cumbersome in complex cases. These two points have motivated our alternative approach to model interactions, which will be introduced in Chapter 4.

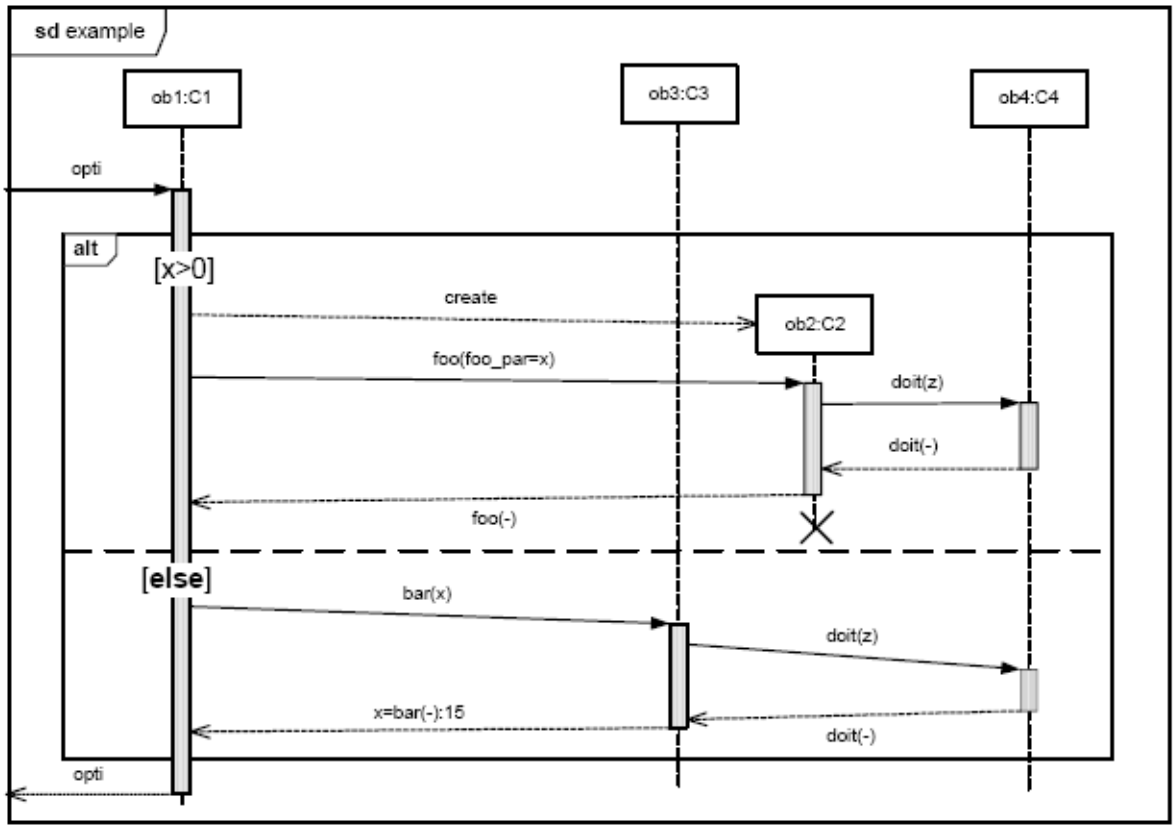

Fig. 2.7: A Sequence Diagram Example (From [OMG07b])

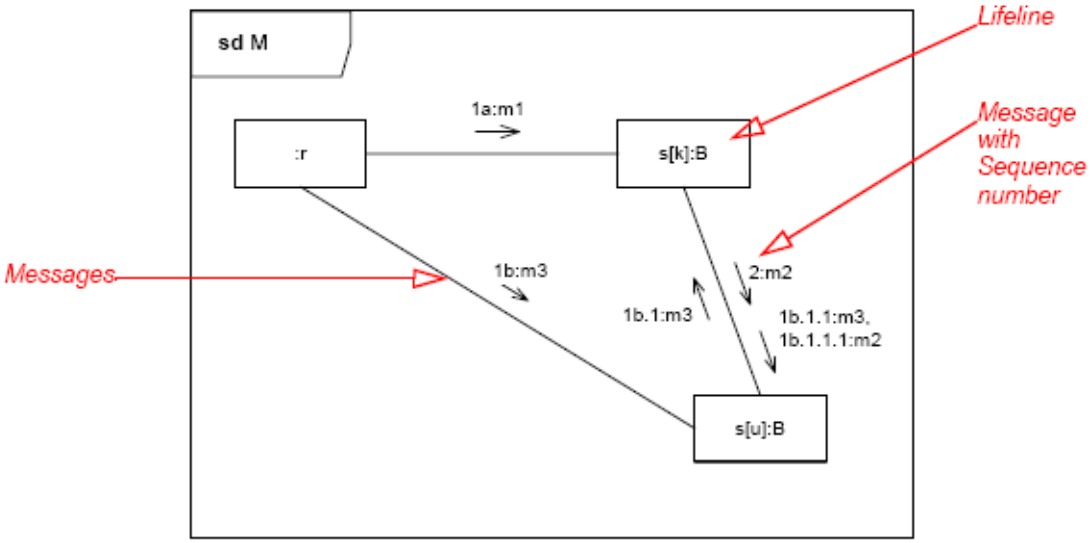

Fig. 2.8: A Communication Diagram Example (from [OMG07b])

State Machines

State machines are used to express the behavior a system part. They can also be used to define the protocol (usage) of an entity by means of modeling its interface's behavior. UML state

machines are derived from Harel's statecharts formalism [Har87]. A state machine is a graph in which nodes are called "states" (instance of the `State` metaclass) and whose directed edges are formed with "transitions" (`Transition`'s instances) explaining how to switch from one state to another. A state usually models a situation characterized by the satisfaction of a usually implicit invariant-based condition. This invariant can be static such as the value of a variable or dynamic such as the process of executing some behavior. There are two specific states: the initial state which represents a starting point of the state machine — no incoming transition — and the final state in which no more transitions are possible. The transition from one state to another is caused by an event that can be associated with a guard condition (that is evaluated before the transition is fired) and an expression defining the behavior to execute if the guard evaluates to true.
Figure 2.9 depicts an example of state machine defining the behavior of a telephone. States are shown as rounded rectangles while arrows represent transitions. The main state of the telephone object, `Active`, is started when a user lifts the receiver and proceeds to `DialTone` sub-state thanks to the transition that has been automatically fired from an initial sub-state (filled dot). `DialTone` is a dynamic state and `do/play dial tone` indicates that the dial tone should be played continuously until a digit is entered (`dial digit(n)`). There are two main ways to exit from the `Active` state. The first one consists in using the `terminate` event which proceeds to the final state of the whole state machine. The second one and the most common simply consists in hanging up, then the telephone switches to the `Idle` state.

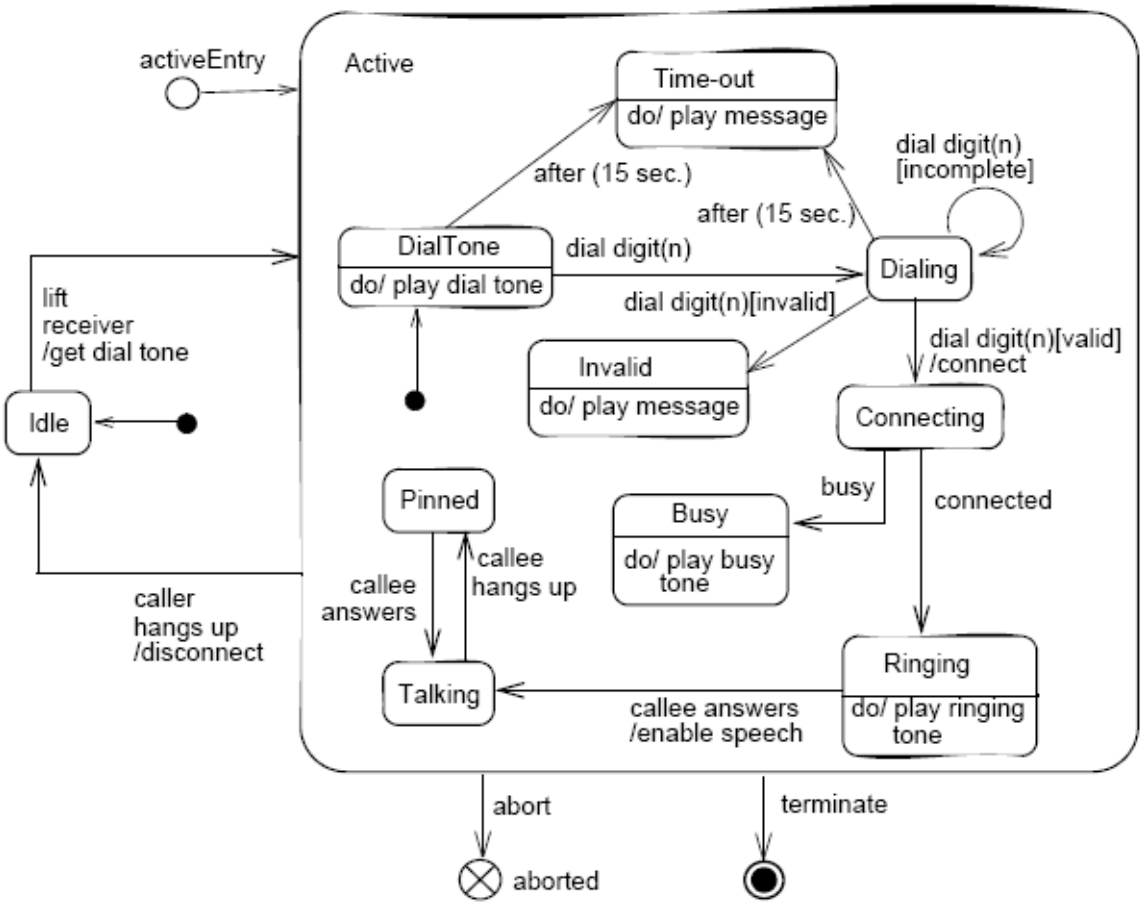

Fig. 2.9: A State Machine Diagram (from [OMG07b])

In order to model how a classifier has to be used in a high-level fashion, a special kind of state machines called *protocol state machines* have been defined. Their main difference with the classic state machines (qualified as *behavioral*) is that they do not model the internal effects when transitions are fired.

State machines allows to fully model classifier behavior in UML and have a well-founded semantics that is often defined in terms of finite state automata. However, more than communication diagrams, state machines require to model quite exhaustively all the states and transitions between them, which can be a difficult task due to the number of states and transitions to consider. They are also more demanding with regard to the skills required to carefully model them.

Actions

Actions are the most fundamental units of behavior in the UML 2 specification. An action takes a set of inputs and transform it into a set of outputs. They are embedded in behaviors which provide their executing context and inputs. Actions model either operation calls and signals sent or read/write operations on UML features (variable assignment, etc.). The `Actions` package includes the source constructs to define languages that enable UML models to be executable. We will make use of such an action language in Chapter 5.

Activities

The `Activities` package focuses on providing constructs to model business processes. Activities have, for a long time, been associated to state machines' semantics. UML 2 provides a new semantics for actions based on token passing. Still, activities and actions syntax contains some commonalities. Activity diagrams describe the internal behavior of UML classifiers as a sequence of activities (composed of lower level actions) coordinated in a procedural flow. Thus, they are suitable to define algorithms in a high-level manner.
Figure 2.10 shows an activity diagram example concerning a reimbursement workflow; the path through the workflow is determined via guard conditions on transitions. Unfilled white diamonds show choices (`if` semantics in programming languages) and the vertical bar is used to denoted forked outgoing edges (duplicate tokens).

Depicting a complex algorithm with an activity diagram can be quite fastidious. Therefore, in the FIDJI method, we have chosen a textual format to define the behavior of design elements (see "intra-component modeling" in Chapter 5).

2.2.6 Making the UML precise: OCL

Object Constraint Language [OMG05e](OCL) is a typed formal language influenced by the IBM Syntropy method [CD94]. It has been developed in order to provide a means to define constraints on UML models, which is less ambiguous than natural language and does not require any mathematical knowledge. OCL is side-effect free; its expressions can only observe model elements' values, their evolution and messages exchanged between elements. Therefore it is not a programming language that can be used to implement algorithms. However, it can be used to *specify* assumptions holding at each state of the system thus clarifying its behavior. OCL can be used for a wide range of purposes including:

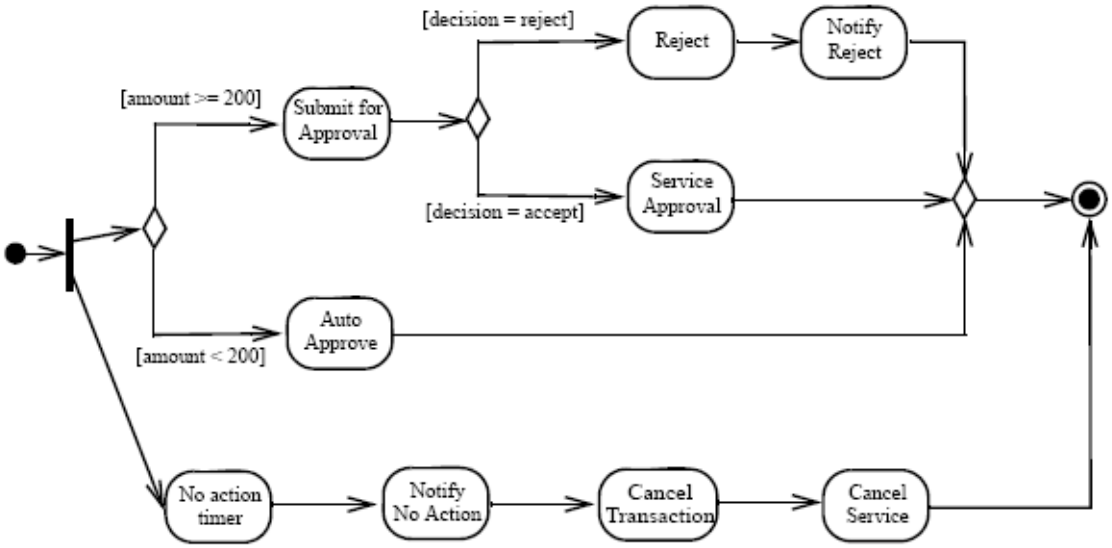

Fig. 2.10: Activity Diagram for a Workflow (from [OMG07b])

- **Structural Constraints**: This is the most common application of OCL. Such constraints are used in structural diagrams to define constraints on model elements (such as the attribute `age` of the class `Adult` that cannot be below 18) or to specify guards in behavioral diagrams and thus determine alternatives as part of a decision;
- **Model Navigation and Query**: Due to its alignment with the UML 2 metamodel and its syntax inspired from SQL [CO93], OCL is a preferred language to retrieve access to model elements in a model following paths determined by relationships;
- **Behavior Specification**: OCL allows to specify a behavior by defining a special type of constraints on UML behavioral elements that must be satisfied before (preconditions) and after (postconditions) its execution. As of the 2.0 version, it is also possible to include in these expressions the set of messages (operations calls or asynchronous data) received by a particular object. Thus, it is possible to model interactions between UML elements as we will show in Chapter 5;

It should be noted that OCL is also used for its navigation and declarative behavior specification abilities in model transformation languages such as ATL [JK05], KerMeta [MFJ05] and obviously the QVT specification [OMG05b].

Context

All OCL expressions are attached to a UML model element providing the application context of these expressions. This context is provided just after the keyword `Context` in the expression. Hence, to limit the attribute `age` of class `Adult` instances, it should be written:

```
Context MyApp::Adult inv: self.age >= 18
```

`MyApp` designating the namespace (that is either the `Package` or `Classifier` containing the context model element) in which the element is to be found. Expressions can also be defined in a UML diagram; depending on the case tool used to define diagrams, there are various possible notations. In general, expressions are bracketed (i.e. between `{` and `}`) either in a special

placeholder in the model element or in a separate comment attached to the element. In both cases, we do not have to make the context explicit since it is provided by the element to which the constraint is attached.

Structural Constraints

Structural constraints are mainly used to define invariants. Invariants must be true for all the instances of a `Classifier` during system operation. These constraints are qualified by the keyword `inv`. The preceding paragraph showed an example of such an invariant definition. The keyword `self` refers to the object designated by the context.

Model Navigation

OCL provides constructs to browse UML models by accessing to elements which follow the relationships they have with the context; associations between elements are treated the same way as element properties (such as attributes for a class), by using the keyword ".". As the navigation over a model results in a collection of elements almost systematically, OCL includes a library of operations to work with collections (`->` should precede any invocation of such an operation):

```
Context MyPackage inv:
self.ownedMember->select(e|e.oclIsTypeOf(Class))->size()<= 10
```

This invariant prevents the class-typed contents of `MyPackage` from having a cardinality greater than 10.

Behavior Specification

An element behavior can be specified in OCL by constraining the element state before and after behavior execution. The global element state can be characterized either using its variable values or with the messages received by the element. The `pre` keyword allows to list the constraints that must be satisfied before the execution of a behavior so that the resulting behavior has an interpretable meaning. The `post` keyword is following by assumptions about the changes occurred after behavior execution:

```
Context MyInteger::divide(Integer a, Integer b):Integer
pre: b <> 0
post: result = a/b
```

In this case, the precondition prevents the denominator of the division operation from being null. The postcondition states what the result should be (`result` refers to an operation result and is of the same return type as that of the operation). If the operation being defined has no-side effect — it is called a *query operation* — the keyword `body` can be used to define the algorithm of the operation.

The second mechanism allows to model messages received by a given classifier. It is comprised of two OCL operators `^` (spelled out as `hasSent`) and `^^` (spelled out as `message` and returning

instances of the special `OCLMessage` type). The first one refers to a particular message that has been sent to a classifier:

```
Context Subject::hasChanged()
post:observer^update(?: Integer,?:Integer)
```

This postcondition evaluates to true of the message `update` has been sent to `observer` with two parameters of type `Integer`. Parameters types are optional and can be used to disambiguate the message if several operations `update` exist. The operator `message` is used to retrieve en entire collection of message and to perform query on them:

```
Context Subject::hasChanged()
post:let mseq: Sequence(OCLMessage) = observer^^update(?: Integer,?:Integer) in
mseq->forAll(m|m.hasReturned())
```

This postcondition checks on all `update` messages that have been sent during the lifetime of `hasChanged()` if they have finished their execution. OCL also provides the ability to express condition on message return (such has the return value of an operation call) via the `hasReturned()` operation offered by `OCLMessage`.

2.2.7 Profiles

We mentioned above that the design of UML 2 was driven by a need for modularity enabling the modelers to focus on the relevant constructs rather than having to inherit from the whole language. However, in certain situations, and despite the huge number of metaclasses offered by the language, the latter needs to be extended. These situations typically include the description of a particular platform (in the MDA sense) with which the modeled software will interact or the specialization of the language constructs in order to support a particular method. The OMG standard provides two manners to extend the UML. The first one consists in using MOF mechanisms to add new metaclasses to the language, and is referred to as a "first-class" extension mechanism. This approach has two main drawbacks. First of all, it is necessary to understand the UML language as a whole in order to properly define the new metaclasses. Secondly, the resulting language is a derivation of the original language which does not longer adhere to the UML specification and is unlikely to be supported by CASE tools. The second approach to UML extension is more "lightweight": it consists in simply extending the existing metaclasses via *UML profiles* [FV04] and [OMG07b] (Chapter 18). In the remainder we present the main constructs used to define UML profiles.

Stereotypes

A stereotype is a metaclass that extends the existing metaclasses of the metamodel. It is a specialization of `Class` from which it borrows the notation with the keyword <<`stereotype`>> placed above its name. Moreover, a stereotype is related to the metaclass it extends via the `Extension` construct which is a kind of `Association`. Figure 2.11 shows an example of such an extension; the stereotype <<Bean>> will be applied to all the components (thanks the to the `required` constraint, when the constraint is missing, the application of an instance of this

stereotype on the instance of the extended metaclass is optional) defined at the model level (M1) for which this profile is applied.

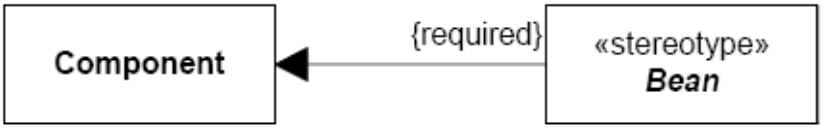

Fig. 2.11: Extension Example (from [OMG07b])

As any other kind of class, stereotypes may have properties. These properties are referred to as *tag definitions*. At the model level, these properties are instantiated and are called *tag values*.

Constraints

In addition to stereotypes and tag definitions, constraints may be added to a profile. These constraints typically enforce the restrictions to the metamodel defined by this profile. Profile constraints may be expressed either in natural language or in OCL, though the latter is preferable because of its precision and the possibility for these constraints to be enforced in OCL-aware modeling tools. Constraints can refer to the stereotypes themselves or to the extended metaclasses; in order to identify the base metaclass in an OCL expression, `Extension` instances respect a particular naming scheme for their roles: the metaclass end role name is `base_` followed by the metaclass name and the stereotype end role name is `extension_` followed by the stereotype name. Thus in the example depicted Figure 2.12, the fact that a `Home` interface shall not have attributes would be expressed as follows:

```
Context Home inv no_att:
self.baseInterface.ownedAtrributes()->size()=0
```

Profile

All the elements constituting a full profile (stereotypes, tag definitions, constraints) are gathered in a `Profile` which is a specialization of `Package`. A profile is always linked to a reference metamodel such as UML; indeed, profiles are just extensions of the base metamodel and cannot modify it differently than by defining stereotypes that have more features and constraints than their base metaclasses. A profile can be applied to a model via the `ProfileApplication` relationship. The same profile can be applied to several packages and a model package can apply several profiles (provided there is no name conflict between stereotypes). The application of a profile on a model is meant to be reversible: when the profile is removed, all the stereotypes and their features are removed from the formerly stereotyped model elements but the elements themselves are kept with the features they inherit from the metaclass of which they are instances.

Fig. 2.12: An EJB Home Interface (From [OMG07b])

An example: SPEM

A significant number of UML profiles have been developed to address various needs such as software architecture [Kas00, ZIKN01], middleware platforms [OMG04, OMG05g] or systems engineering [OMG06c]. In these paragraphs, we present the Software Engineering Process Metamodel (SPEM) [OMG05c] dedicated to the description of software processes and which will be used in Chapter 3 to define the steps of the FIDJI methodology. This profile is currently aligned on 1.x versions of UML and a revision of UML 2 is currently ongoing. Figure 2.13 depicts the base stereotypes defined in SPEM. In the following, we will focus on the most useful ones for our purposes.

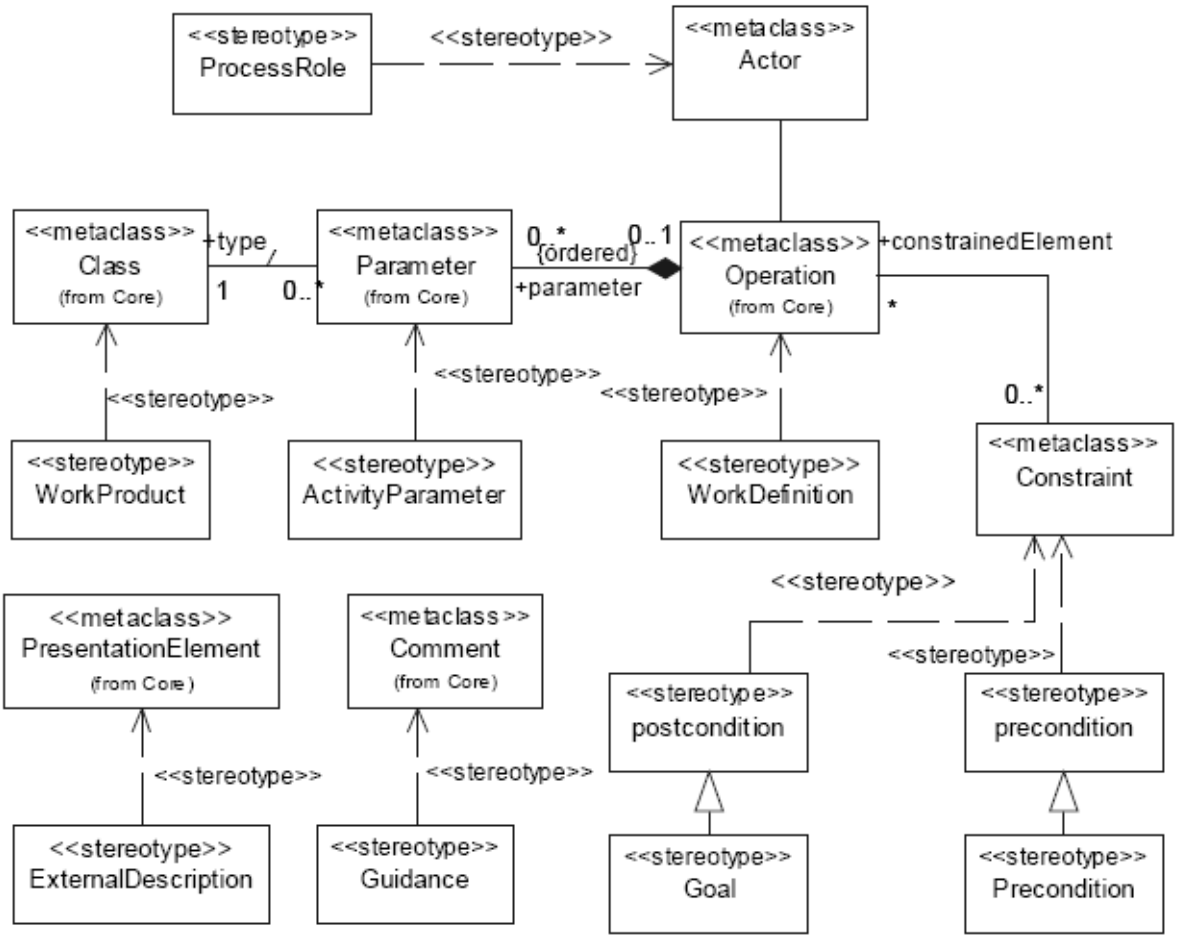

Fig. 2.13: SPEM Stereotypes

One of the key notions of the SPEM profile is <<Activity>>. An activity represents a high-level working unit of a method lifecycle. An activity can be further decomposed in steps. It has input and output parameters which respectively represent the artifacts needed to perform this activity and the artifacts produced during the execution of the activity.
Pieces of work produced and consumed by activities are all instances of the <<WorkProduct>> stereotype. Work products are associated with <<WorkProductKind>> which further defines the type. Two sub-stereotypes of <<WorkProduct>> are defined:

- <<UML Model>> refers to any sort of UML model;

- <<Document>> refers to any document which can be an informal description of a system or a low-level technical artifact such as a configuration file.

Finally, <<Guidance>> is a sort of comment that provides more information on a model element, in particular on the way this element can be derived or constrained; it can be a UML profile, a model transformation, an algorithm, etc. The graphical icons associated with these stereotypes are depicted on Figure 2.14.

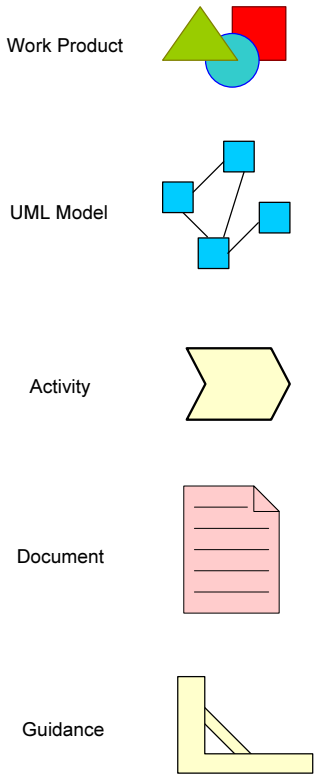

Fig. 2.14: SPEM Stereotypes Notation

2.3 Software Architecture

2.3.1 Definition

Software architecture has been an important topic in the academic software engineering research for more than 10 years — while its main concepts were defined in the 60s — but appeared as a major industrial issue only a few years ago. Why this sudden interest? Indeed, this field gained momentum when software systems became so big and complex that it became necessary to define new concepts to understand them and larger engineering teams to actually build them. In [Kaz01], Rick Kazman explains why architecture is crucial to software-intensive systems:

- "Communication among stakeholders": Due to the complexity and size of modern software systems, numerous skills are required to complete a project successfully. These skills are supported by analysts, UI designers, core programmers, performance engineers, testers including people that will set up the hardware on which the project will run and obviously the customer of the project. All stakeholders need to understand and communicate about the project in order to agree on the major decisions that impact their particular skills and interest;
- "Early design decisions": Software architecture represents the earliest artifact on the "solution" side (the "problem" side being addressed by requirements engineering techniques) of the project; as a result, it is also the first artifact on which project's numerous qualities such as flexibility, performance or portability can be assessed;
- "Transferable abstraction of a system": Software architecture represents a relatively small and abstract amount of information about how a system is structured and how its components work together. This abstraction (software architecture description of a system can be seen as part of a PIM level in an MDA approach) makes it easy to transfer a software architecture description across systems sharing similar requirements especially in a software product line context (see Section 2.4).

Now that we have analyzed the rationale behind software architecture, we can give its definition. Over the years, many definitions of software architecture have been proposed which have been gathered by the Software Engineering Institute [4] at Carnegie Mellon University. As software architecture is an abstract notion, these definitions vary greatly depending on the focus of their authors and their level of detail. In this dissertation, we will retain the following definition which, we believe, synthesizes the essence of software architecture:

Definition 7 (Software Architecture) *''The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [BCK03]*

The following remarks could be pointed out regarding this definition. Firstly, software elements may be of diverse nature: tasks, processes, functions. Secondly, there can be more than one structure for a given system. By "structure" the authors mean a particular organization of elements; depending on its purpose (run-time processes, module organization), this structure may relate the same elements differently. Structure models are called "views" [CBB+02] and

[4] `http://www.sei.cmu.edu/architecture/definitions.html`

form together the system architecture. Though, none of them can claim to represent "the" architecture of the system. Thirdly, the "externally visible properties" refer to the assumptions (services, performance, etc.) which can be formulated about the elements and every detail of these elements that is not part of the "visible properties" should be omitted, thus revealing the abstract nature of architectural descriptions. Finally, as stated in the rationale above, software architecture is about defining relationships between elements inasmuch as they participate in the visible properties of the system: choosing a particular network protocol between one server and their clients has repercussions in terms of performance, reliability and security of the whole system. Since architectures are structures of software elements, every implemented system has an architecture. However, without any architectural description either being explicitly defined during the development of the system or being recovered via reverse engineering techniques, nobody can understand the architecture of a system and assess its benefits and/or pitfalls.

2.3.2 Architectural Views

First of all, we need to explain the notion of view sketched above. The IEEE made a valuable clarification and effort in their Recommended Practice for Software Architecture description [IEE00]:

- **View**: A view is defined as "a representation of a whole system from the perspective of a related set of concerns." This definition inspired Clements et al.'s work on the documentation of architectural views [CBB+02]. It is worth mentioning here that a view is a complete model of the system with respect to a particular focus; a simple analogy is to consider a view as the result of looking at the system with filter glasses;

- **Viewpoint**: A viewpoint is a "specification of the conventions for constructing and using a view." In our modeling terminology, a viewpoint is a modeling language for a view and defines the perspective (i.e. the type of properties of the system) its associated view must exhibit and how. Viewpoints include rationales which motivate their use in a particular context.

Secondly, we have mentioned that one of the motivations for architectural description is to allow stakeholders to express their concerns about the software system, those concerns being described in terms of views conforming to their respective viewpoints. As a consequence, there is not a predetermined set of viewpoints that fits every system; viewpoints rather depend on the kind of concerns identified by stakeholders and the development context (application domain and methodology). As a result, several sets of viewpoints for architectural description have been developed [Kru95, BCK03, Zac87, IEE00]. One of the best known approaches is called "4+1" and has been developed by Kruchten [Kru95]. The viewpoints offered by this approach are the following (see Figure 2.15):

- **Logical**: The logical viewpoint addresses the functional requirements of a system by offering constructs for the major packages, components and classes that compose system functionality;

- **Process**: The process viewpoint focuses on system run-time processes and the way they interact. It offer constructs to model qualities such as parallelism, concurrency or fault tolerance;

- **Implementation**: The implementation viewpoint offers constructs for software modules such as source code files, executables or data files;
- **Deployment**: The deployment viewpoint depicts how the various components and executables are mapped to the underlying platforms (middleware, OS, etc.) or hardware nodes (servers, sensors);
- **Use Case**: Finally, the use case viewpoint, though not pertaining explicitly to the architectural level, propose to model the key scenarios used to drive and validate architectural development.

These viewpoints have been used for the architectural description of systems developed according to the Rational Unified Process (see Section 2.6) and use subsets of the Unified Modeling Language (see next section) to provide their constructs. Bass et al. propose to organize their architectural structures (modeled as views) according to the following viewpoints:

- **Module**: The module viewpoint focuses on the description of system implementation elements. It proposes to address issues such as functionality implemented by modules, inter-module dependencies as well as dependencies with other software, composition and inheritance amongst modules. This viewpoint is rather close to the logical viewpoint presented here before;
- **Component-and-Connector**: The component-and-connector viewpoint addresses system decomposition in terms of components (which are the main units of computation) and connectors (communication paths amongst components). Issues addressed here are the interactions between major components, their potential replication, shared data, etc. ;
- **Allocation**: This viewpoint focuses on the definition of deployment properties and is close to the deployment viewpoint of the 4+1 approach.

As noted by Bass et al. [BCK03], in addition to selecting the relevant viewpoints for the description of a software architecture, mappings exhibiting the relationships between views have to be provided. The reason for these mappings is that all the views model the same system and therefore may have some elements in common. It is therefore important to document these commonalities in order to increase the understanding about the architecture as an "unified conceptual whole". We may add, on a technical perspective, that ignoring these relationships may yield inconsistencies (e.g. setting distinct properties which cannot be satisfied at the same time for a single element) that will lead to implementation issues. In fact this issue is simply an instance of the model consistency issues presented in Section 2.1.

2.3.3 Architectural Styles

Though every system architecture is different, it often addresses common problems for which known and proved solutions exist. Therefore, it is valuable to reuse these solutions when designing the architecture of a system as guides for its development and to ensure that certain qualities are met. This is the purpose of *architectural styles or patterns*: "An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used" [BCK03]. Numerous styles have been developed [SC97, AAG95]. Amongst them, the most popular are the following:

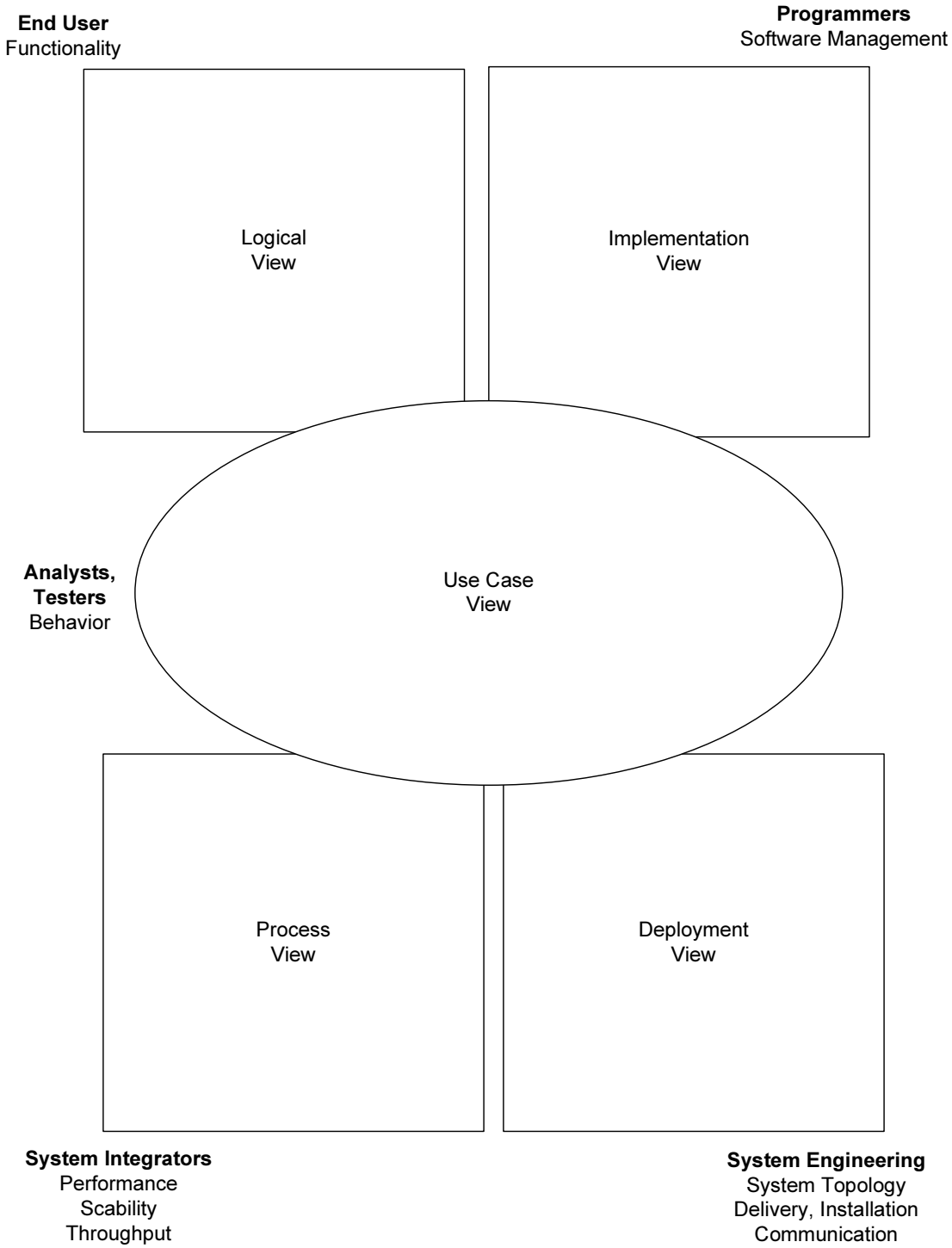

Fig. 2.15: 4+1 Views for Software Architecture (Adapted from [Kru03])

- **client-server**: This is the most commonly used style in distributed systems. It comprises

a server element whose role is to provide services to several clients connected to it using a specific communication protocol;

- **pipe-and-filter**: This style is used in data intensive systems in which information have to be processed throughout the system. This style comprises elements that process information ("filters") and send it through "pipes" to other filters. A concrete application example of such a style is the combination of several UNIX commands through pipes;

- **layered**: The layered style [FO85, LS79] is a hierarchical decomposition of elements in which the $(n+1)^{th}$ level of elements is only depending on the n^{th} level. This style is endorsed in network protocol descriptions (OSI reference model [Zim80]) or in middleware-based applications in which middleware depends on lower layers containing operating system and networks protocol implementations to provide service to the application developers;

- **N-tier**: N-tier architectural style can be seen as a generalization of the client-server one with multiple servers. Server components are grouped according to their purpose (application servers, database servers, etc.).

Note that architectural styles may be employed in more coarse-grained reusable architectural structures, i.e. *reference architectures* which are a standard organization of software elements that cooperatively solve a particular problem. Their usage is particularly fruitful in the context of product lines (see Section 2.4) in which reference architectures are used to build a set of related products.

2.3.4 Architecture Description Languages

In order to support the description of software architectures, several dedicated modeling languages were designed. Although there are great variations in their syntax, semantics and application range (distributed systems, real-time and embedded systems...), there are all based on a limited number of core concepts, which are the following:

- **Components**: "Components are the locus of computation and state." [SDK+95]. Components can be as large as an entire subsystem or as small as a method and are directly supported by code units at the implementation level. One particular characteristic of components is that their interactions with other components are clearly identified via connectors and fulfill a contract as specified by their interfaces. Thus, a component represents a reusable unit of behavior that may be replaced by another fulfilling a compatible contract;

- **Connectors**: Connectors model the relationships between components; they are the "pipes" in the pipe-and-filter style. Unlike components, there is not direct link between a connector and its implementation which may be performed via various mechanisms; remote procedure calls, publish-subscribe, etc;

- **Roles**: Roles are entities attached to connectors which represent a participant in the interaction modeled by the connector. For example, a pipe contains two roles, i.e. one for reading and one for writing;

- **Interfaces or Ports**: An interface is a set of interaction points between a connector and the other components and/or connectors attached to it. As such, they represent the

> "externally visible properties" of the software elements targeted by the software architecture definition. Interfaces allow to connect components which provide some services to other components that require them through connectors. They are hence a key notion for reasoning on software architectures.

Architecture Description Languages (ADLs) have been designed to satisfy specific requirements thus inducing a great variability in the constructs (both at the syntactical and semantic level) they are offer and the significant number of ADLs that have been developed. A complete survey of ADLs has been carried out in [MT00]. In the remainder of this section, we focus on the two most relevant ones with respect to our methodological context, ACME and WRIGHT.

Even though not being explicitly designed as an ADL [MT00], one of the most popular languages is ACME [GMW97]. Its main purpose is to be an interchange language i.e. to support mapping between ADLs by factorizing software architecture concepts and to facilitate exchange of architectural specifications between tools. ACME supports the structural definition of components, connectors ports and roles. ACME provides both a textual and graphical notation for these constructs. Additionally, a set of properties can be defined for components and connectors; these properties are used to attach additional information (such as boolean specifying the maximum number for roles of a connector or a pointer to an additional specification of the component's behavior). The whole architecture description in ACME can be associated with a set of constraints expressed in first-order logic. These constraints can either be used to specify invariants that have to be enforced in the implementation or as rules which guide design but may be violated if necessary.
ACME is supported by a comprehensive tool support [CMU07].

WRIGHT [All97] is one of the rare languages that focus on describing the behavior of the architectures in addition to their structure. At the structural level, notations for components, ports, connectors and roles are provided. Indeed, these abstractions are types that can be parameterized in order to support architectural styles or product lines (see Section 2.4) more easily. But the most important contribution of WRIGHT is its formal semantics to model the behavior, which implies that WRIGHT specifications can be analyzed. The formalization used here is a subset of the process algebra CSP [Hoa85] which is used to model both component internal behavior and component interactions (called *architectural connection*) [AG97]. The aim of this approach was to define connectors behavior between components in terms of interacting protocols, that is the behavioral description of the following elements:

- **Roles**: Roles represent the interaction points of connectors to which interacting components have to be attached. Therefore, their behavior should be fulfilled by the behavior of the components involved in the interaction;

- **Glue**: The glue determines the coordination between the roles, such as the sequence of the events exchanged (that is, in the GAM, the sequence of operation calls and their returns).

WRIGHT's formal basis allows to precisely define the events exchanged through a connector both independently (i.e. being defined in a style) and in the context of a particular architectural description by matching roles to component ports.

2.3.5 Describing Architectures with UML/OCL

As we have seen above, ADLs natively support architectural concepts and often include formalized syntax, semantics and tool support. However, they tend to be so specialized and numerous that they were not an instant hit with the industrial software engineering community. From a methodological point of view, using such languages is an issue because users need first to be trained, which hampers the adoption of a method using these languages.
The increasing popularity of UML has led software architecture experts to consider this language as an opportunity to model software architectures [MRRR02]. However, UML 1.x has for long suffered from a lack of native architectural support (unsatisfactory constructs to define architectural components and connectors). To overcome this issue, several extensions to the language have been provided, generally provided in terms of profiles [Kas00, ZIKN01, KCSS02]. As presented in Section 2.2, UML 2.x has integrated some architectural concepts to the language (see "Structural Modeling"). These constructs form a reasonable basis to defined an ADL based on UML 2. However, there are still some issues to address before using UML 2 as a mainstream ADL:

- **Connector modeling**: Though present, the "connector" construct is rather poor; it does not allow to directly add some information, neither about its structure (such as the size of a buffer in a pipe) nor about its behavior;
- **Architectural styles modeling**: As noted by Zdun and Avgeriou [ZA05], there is no dedicated support for architectural styles in UML. A way to model them has to be found precisely.

Ivers et al [ICG+04] proposed three strategies to improve connectors in UML 2.0:

- **Using associations**: A UML association can be used instead of a connector. However, as such, it does not really differ from the UML 2.0 connector construct and does not allow neither to model attribute nor to attach semantic information;
- **Using UML association classes**: If the connector is modeled as an association, it is possible to attach an association class to it. This permits to model attributes and behavior associated to this connector. However, this strategy is inconvenient from a visual point of view and may yield confusion in roles modeling;
- **Using classes**: As a variant of the previous strategy, classes may be used to represent connectors. This removes the visual clutter caused by the use of association classes. However, this also removes the distinction between connectors and classes. One solution would be to define a stereotype with a specific graphical representation but it is likely to be unsupported by UML case tools.

A more precise approach to detail component interactions consists in employing OCL. In [Car03], Cariou presented such an approach in the context of component interactions in UML collaborations. The key point of this approach is the concept of *medium*. A medium can be seen as an abstract component devoted to the component interactions and is modeled as a UML collaboration. A medium is specified via views and using the following notations:

- **Collaboration Diagram**: Describes the elements involved in the interaction as well as the sequence of the operations called;

- **OCL**: OCL is used to define constraints on the medium and the behavior (via pre/post conditions) of offered and required operations. OCL is used as much as possible to maximize precision and consistency amongst views;
- **State Diagrams**: State diagrams are used to specify synchronization aspects thus completing information provided in the collaboration diagram.

In order to generalize the use of OCL for interaction modeling, Cariou proposed two extensions to the language. The first one, `caller`, allows to refer to the instance that has called a given operation and is used in pre/post conditions of operations. The second one, `oclCallOperation`, allows to specify if a given operation has been called.
The interaction approach we will propose in Chapter 5 will be completely based on OCL and will take advantage of OCL 2.0's `OCLmessage` construct, thus making Cariou's second extension obsolete.

Currently, a few efforts are under way to address the specific problem of modeling architectural styles in UML 2.0 [ZA05, PM03]. Both approaches are based on profiling to extend the UML metamodel through stereotypes and OCL constraints. We will integrate in our design profile defined in Chapter 5 some examples of well-known architectural styles, which are relevant for the FIDJI method.

2.3.6 Enterprise Architecture Frameworks

We have mentioned above that a software architecture description contains a number of different views. In order to facilitate architectural description and sharing amongst stakeholders within a particular organization, viewpoints have to be selected and related to each other, then guidelines to model software architectures according to these viewpoints have to be provided. This is the role of an *Enterprise Architecture Framework* (EAF) to provide such information:

Definition 8 (Enterprise Architecture Framework) *An enterprise architecture framework is a tool which can be used to develop a broad range of different architectures. It should describe a method for designing an information system in terms of a set of building blocks, and for showing how the building blocks fit together. It should contain a set of tools and provide a common vocabulary. It should also include a list of recommended standards and compliant products that can be used to implement the building blocks [Gro06].*

Several EAFs have been developed [5] in order to perform architectural description in various domains (defense, IT, etc.). In the following, we will focus on the three most relevant.

DODAF

The Department of Defense Architectural Framework (DoDAF) [Gro04]. DoDAF decomposes an architecture description in the the following views:

- **Operational View (OV)**: The purpose of this view is to provide a description of the tasks and activities supporting the mission addressed by the system. This view is comprised of textual and graphical elements describing tasks and activities as well as information exchanged between them;

[5] http://www.opengroup.org/architecture/togaf8-doc/arch/

- **Systems View (SV)**: The system view defines all the elements and their interconnections supporting the mission defined by the operational view;
- **Technical Standards View (TV)**: The technical standards view details how the different system parts may be arranged (giving for example a set of standards to follow) and technology evolution that may affect the architecture.

Additionally, a fourth view (called "All-Views" (AV)) provides the linkage between the other views by establishing a common vocabulary, and information on the system context (scope, purpose, intended users, etc.).

These elements are complemented by a set of principles and guidelines to document architectures with DoDAF in a stepwise manner and by information on how to integrate the different views.

TOGAF

The Open Group Architecture Framework (TOGAF) is more focused on processes than on providing detailed architecture viewpoints. Indeed, TOGAF defines a process, called Architecture Description Method (ADM), covering all architecture description phases from vision to evolution. The high-level process is shown in Figure 2.16.

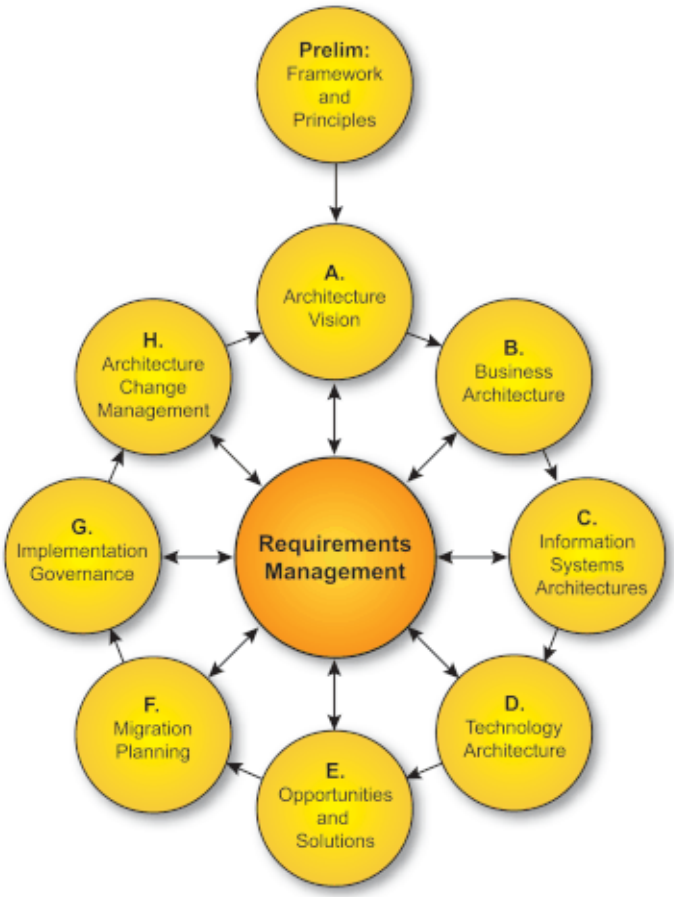

Fig. 2.16: ADM Phases (from [Gro06])

Each of the phases are decomposed in well-identified sub-steps. Artifacts composing the architecture are developed as a result of each phase and may be modified in a later phase (the

process is iterative not waterfall). TOGAF is meant to be integrated with other architecture frameworks that provide detailed architectural artifacts.

Zachman Framework

The Zachman framework was introduced by John Zachman in 1987 [Zac87] and is considered as the first and one of the most complete EAF to date. The framework is comprised of 36 views organized in a table:

- **Perspectives**: Perspectives form table rows. Identified perspectives are: Planner (which position the product in its environment), Owner (business processes to be implemented by the product), Designer (logical view on the system structure and behavior), Builder (technical description of the system's artifacts), Subcontractor (low-level artifacts: data model, configuration files etc.) and Functioning Enterprise (represents the user concern in the system);
- **Dimensions**: Dimensions form table columns and are designed to cover to a particular aspect of the system. Dimensions are the following: Data (What?), Function (How?), Network (Where?), People (Who?), Time (When?), Motivation (Why?).

In order to be general, Zachman's framework neither prescribes any specific languages to represent views in the cells nor provide any methodological guidelines to define architecture according to the framework. As noted by Pereira and Sousa [PS04], this approach has both pros and cons. Although the Zachman's framework is adaptable to any development methodology with any modeling language supporting the description of views, it provides no guidance to architects to model views.

Even though EAFs are interesting structures because they define a set of views that are relevant to a particular context, we think that their will to be modeling language independent (none of the EAFs above explicitly define the viewpoints their views should conform to) unavoidably raises two issues. At the methodological level, architecture development guidelines are condemned to be very general as they cannot rely on metamodeling elements to define how integration between the different views has to be achieved. At the technical level, there is no possibility to validate conformity pf an architectural description with its EAF since consistency rules cannot be checked against. This issue is currently being tackled by CASE tool vendors who provide specific extensions to model EAFs (for example, NoMagic MagicDraw CASE tool [NoM07] includes a plugin to model DoDAF compliant architectures).

2.4 Software Product Lines

In this section, we present the notions supporting the concept of Software Product Line (SPL) which provides the research context of the FIDJI method.

2.4.1 Introduction

The concept of software product line [6] has its origins in the program families approach introduced by Parnas [Par76]. However, it drew attention of the software engineering community when software begun to be integrated massively in families of hardware products, cellular phones [MH05] being the most known, but several other areas, such as automotive systems, aerospace or telecommunication are also targeted by software product lines [7].

As we mentioned in Chapter 1, the fundamental idea on which software product lines rely is to reuse a base of managed software artifacts to systematically define, design, build and maintain a set of related products in a given domain. This idea is captured by Clements and Northrop [CN01] and we will use their definition of SPL throughout this dissertation:

Definition 9 (Software Product Line) *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.*

Thus, adopting a SPL approach implies to perform two intertwined activities, i.e. *Domain Engineering* and *Application Engineering*. Domain engineering deals with the " common set of core assets" serving as a base to develop features "that satisfy the specific needs of a particular market segment or mission". An *asset* is "a description of a partial solution (such as a component or design document) or knowledge (such as a requirements database or test procedures) that engineers use to build or modify software products" [Wit96]. Assets need to be carefully defined, built and managed. The first step is to identify the "particular market segment or mission" which resides in a *domain* [BCD+00]:

Definition 10 (Domain) *An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.*

Domain engineering starts with a domain analysis phase that identifies *commonalities* and *variabilities* amongst SPL members. In this dissertation, we will adopt the following definitions (adapted from [CHW98, WL99]) for commonality and variability:

Definition 11 (Commonality) *A property held uniformly across all the members of the SPL.*

Definition 12 (Variability) *A property about how members of a SPL differ from each other.*

[6] For historical and geographical reasons, US terminology "product lines" was called "product families" in Europe and was corresponding to distinct yet related communities, holding different research conferences (Software Product Line Conference in the US and Product Family Engineering in Europe). Recently, these communities decided to merge these research events under the generic name "software product line conference" name. Therefore, we decided to keep this terminology.

[7] For an overview of famous software product lines, consult SEI's SPL Hall of Fame: `http://www.sei.cmu.edu/productlines/plp_hof.html`

Effective management of variability [vGBS01] is key to success for software product lines; it determines how flexibly new members of a given SPL can be obtained and defines SPL boundaries. Various mechanisms have been proposed to model and achieve it at various abstraction levels (requirements elicitation and design levels principally) [SvGB05]; we will summarize some of them in Section 2.4.2.

Clements and Northrop's definition also mentions that the "set of software-intensive systems" is developed "from a common set of core assets in a prescribed way". This specific activity is known as *application engineering* or *product derivation* [ZJ06, DSB05].

Definition 13 (Product Derivation) *A complete process of constructing products from the domain assets.*

We will discuss the available techniques for product derivation in Section 2.4.3. As we mentioned earlier, domain engineering and product derivation are intertwined, the former providing core assets that are "consumed" by the latter wheb building applications. As a result of the product derivation task, feedback on specific products can be acquired and used to improve SPL core assets as shown in Figure 2.17.

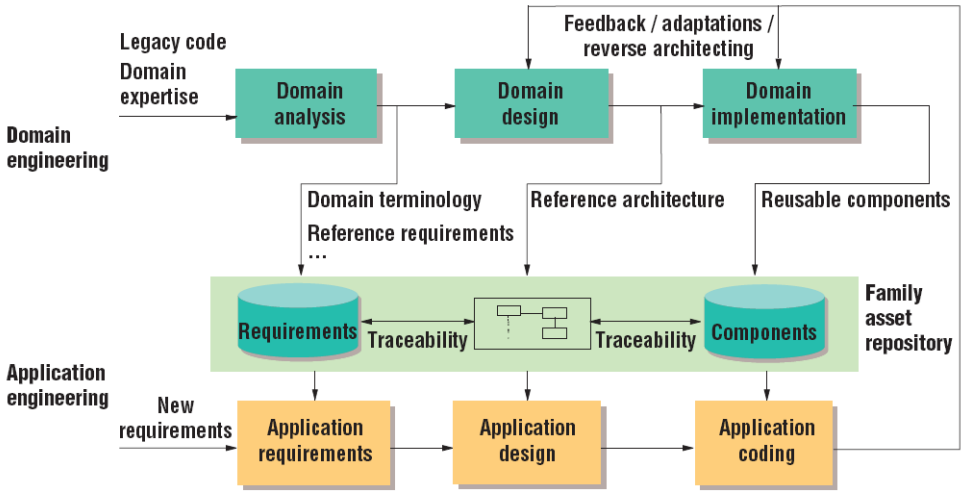

Fig. 2.17: Domain Engineering and Application Engineering Processes [vdL02]

2.4.2 Domain Engineering

In this section, we present the current approaches that support domain engineering within software product lines. We have grouped these techniques according to their purpose, i.e. focusing on defining domain assets at the requirements and design levels.

Requirements

There are two popular techniques to define software product lines requirements: *Feature Modeling* and *Use Cases*.

Feature Modeling

Several definitions of the notion of feature can be found in the research literature. The most general was given by Kang et al. in [KCH+90]: "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems". Czarnecki and Eisenecker focus on the SPL variants: "a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family" [CE00]. Finally, Bosch emphasizes on system behavior: "a logical unit of behavior that is specified by a set of functional and quality requirements" [Bos00]. We will retain the definition given by Czarnecki and Eisenecker because we believe that it is crucial to capture commonalities and variabilities at an early stage of SPL development.

Variability amongst features is typically depicted using a *feature diagram* [TH03]. According to their definition, feature diagrams often constitutes trees (though some notations allow additional relationships which formally make feature diagrams graphs) composed of nodes and directed edges. The tree root represents a feature which is progressively decomposed using mandatory, optional, alternative (exclusive-OR features) and OR-features. In feature diagrams, mandatory features are features which are always included in every product. Feature that are not necessarily included in every product are qualified as variables. Variation points are features that have at least one direct variable subfeature (i.e. as one of its children).

The first feature diagram notation was introduced by Kang et al. in the context of their "Feature-Oriented Domain Analysis" method [KCH+90]. It has a very simple syntax which makes it easy to use as shown in Figure 2.18.

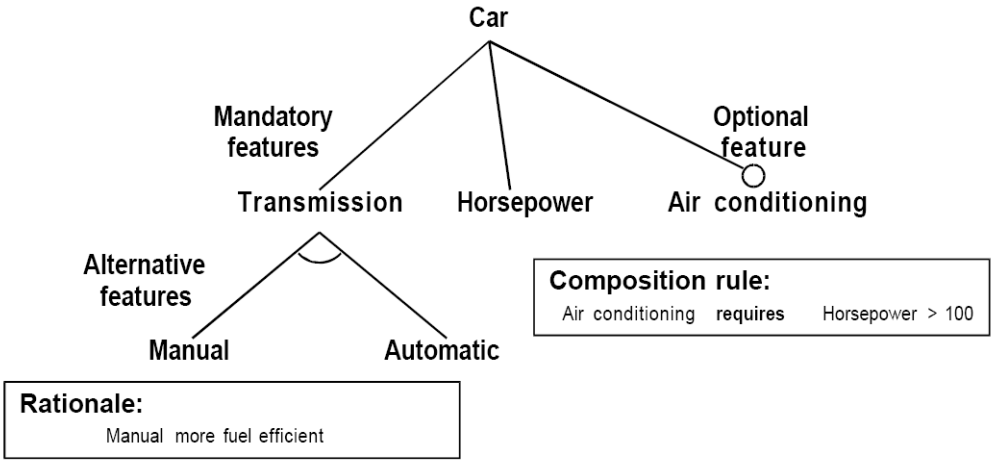

Fig. 2.18: FODA's Notation

Depending on researchers' view on the concept of feature, FODA feature diagrams were found to lack of expressiveness. Therefore several extensions (in addition to variations of the concrete syntax) were added to the original notation. In [CHE05b], Czarnecki et al. detailed these extensions which are described below:

- **Feature Cardinalities**: Features can be annotated with cardinalities, such as [1..] or

[3..3]. Mandatory and optional features can be considered as special cases of features with the cardinalities [1..1] and [0..1], respectively. Feature cardinalities were motivated by a practical application [CBUE02]. In addition, Riebisch et al. [RBSP02] proposed cardinalities (conforming to UML notation) in order to generalize the concept of feature grouping (by specifying the number of features one can select in a set);

- **Attributes**: Attributes were introduced by Czarnecki et al. [CBUE02] as a way to represent a choice of a value from a large or infinite domain such as integers or strings. An elegant way to model attributes is to allow a feature to be associated with a type, such as integer or string. A collection of attributes can be modeled as a number of subfeatures, where each is associated with the desired type;

- **Relationships**: Several authors (Griss et al [GFdA98], van Gurp et al. [vGBS01]) proposed to extend feature models with different kinds of relationships such as consists-of or is-generalization-of which are usually used in entity relationships or UML class diagrams. For example, van Gurp et al. [vGBS01] use an "or specialization" and "xor specialization" to define the binding time. Binding time occurs when this feature has to be chosen (design, compilation, run-time...);

- **Feature categories and annotations**: FORM distinguishes among context, representation and operational features. Other kinds of feature categorization exist. Griss et al [GFdA98] propose several layers of features according to their abstraction level (see Section 2.4.3) and Czarnecki et al. [CHE05b] propose to include also a fine-grained categorization amongst project stakeholders (such as analysts, architects, etc.). In addition, annotations to feature diagrams were proposed to model relationships [KCH+90] or constraints [BTRC05, GFdA98, KKL+98];

- **Modularization**: A feature diagram may contain one or more special leaf nodes, each representing a separate feature diagram. This mechanism allow to break up large diagrams into smaller ones and to reuse common parts in several places. This is an important mechanism because, in practice, feature diagrams often become too large to be considered in their entirety [CHE05b].

In addition to these conceptual extensions, feature diagrams vary in their concrete syntax; most of them are specific but, for instance Gomaa [GS02a, Gom04] uses UML. In consequence, understanding the characteristics of all feature modeling languages is not easy and interoperability between different notations (which are not always precisely defined) may be error-prone. In [SHT06], Schobbens et al. proposed a generic approach, called Free Feature Diagram (FFD), able to express abstract syntax and semantics for most feature modeling languages in a formal way. This approach was initially designed to compare the expressiveness of various feature modeling languages but may also be used as a basis for tool support or, as illustrated in [GP06], use its abstract syntax and semantics to be independent of any particular feature modeling language. FFD proposes the following operators to represent variants:

- and_n: Evaluates to true if all the n ($n \in N*$ n represents the operator's arity) subfeatures for a feature exist in the final product (mandatory features). or_n and xor_n are defined the same way: at least one (resp. exactly one) of their n subfeature(s) is selected in the final product;

- opt_n: Always evaluates to true (optional semantics);

- $vp_n(i..j)$: $(n, i, j \in N*$ and $j > i)$ evaluates to true if at least i and at most j subfeatures are selected for the product. Indeed, this last operator conveys the semantics of all the aforementioned operators [SHT06], however, in practice, "classic" operators may be easier to use;
- Two binary operators are defined to express dependencies between features: mutex for mutually exclusive features (|) and requires ($\Longrightarrow$) for compulsory dependencies between features.

Use Cases

Use Cases [JCJO92, Coc01] are a popular notation to describe requirements of a software system in such a way that the final user can understand and modify these descriptions. Use Cases capture functional and non-functional requirements in terms of scenarios relating how the actors interact with the system. They are hence excellent candidates to perform the elicitation of a SPLbehavior. In addition to UML notation for use cases presented in Chapter 2.1, use cases are mainly described using textual descriptions which typically comprise use case name, actors involved, pre/post-conditions, flow of events, exceptions or possible alternatives in the flow. These textual descriptions are written in natural language but may be completed with more formal languages as shown in [Rus97, GL00]. We will detail our approach to rigorous use cases in Chapter 4, Section 4.2), which is based on a template. Various templates have been given ([Coc01, Lar02] just to name a few) organizing the above information differently.
Standard use case constructs can be used to represent variability amongst use cases:

- **UML Use Cases Modeling Elements**: As explained in [Gom04], Gomaa illustrates the use of `extend` relationship, extensions points and `include` relationship. The extend relationship can be used to simplify the expression of complex alternative behaviors. Moreover, an extension point [OMG05f] allows to differentiate variants by enabling to specify guard conditions. The `include` relationship can be used to specify optional use cases: for example the optional use case "print receipt" may include the mandatory one "Withdraw Cash" in the context of an ATM system;
- **Textual Templates**: Textual templates propose fields to document alternatives and extensions (which can be used to document optional scenarios).

However, use cases were initially meant to describe scenarios of a single system, not a collection of systems differentiated by variants. As noted by Halmans and Pohl [HP03] as well as Trigaux and Heymans [TH03], there are major shortcomings with this approach. In the UML representation it is difficult to distinguish between an alternative behavior which represents a variant in the SPL and a behavior which is part of the use case (it can be a list of choices given to the user part of common behavior of the SPL) not related to any variability identified by the SPL requirements process. In textual representations, the ad-hoc description of variability (by just annotating use case descriptions) may become unclear. Therefore, several extensions were proposed to extend both textual and UML-based use case notations in order to improve variability support amongst software product lines. These extensions are summarized in the following.

In [BNT02], Biddle et al. introduced a first approach to facilitate use case reuse: the possibility to embed parameters anywhere within the textual description of the use case. They also designed

a tool, called Ukase, acting as repository for use case description and facilitating their edition. This approach is not specific to product lines.
In [FGJ+03, FGLN04], Fantechi et al. define an approach, called Product Lines Use Cases (PLUC), to extend use cases with the explicit description of variability. In particular, they propose to add tags to model variants in the use case text which have to be substituted by the actual value for a given product. A special clause at the end of the use case completely define the tag (i.e its type and possible values). Figure 2.19 illustrates the use of these tags. There are three types of tags:

- Alternative: indicates that it is possible to choose the actual value for the tag from a predefined set of values, each of them depending on a condition (exclusive-OR semantics);
- Optional: indicates that the actual value is chosen indifferently from a set of values;
- Parametric: indicates that the actual value depends on a condition specified with `if then else` constructs related to the other use case tags values. This type differs from the alternative type since it allows more than one variant to be chosen [FGLN04].

There are also extensions of the UML notation for variability support. In [HP03], Halmans and Pohl extend the standard UML notation for use case diagrams and define new minimal graphical notations, depicted in Figure 2.20. These constructs allow to specify the following:

- Variation points: variation points are represented using the `include` relationship followed by a triangle. A variation point is said mandatory if at least one of its variants is required (shown in this case with a filled triangle) and optional if the variants can be selected or not without any restriction.
- Variants: variants are represented with the stereotype <<`variant`>> and they are always associated with a variation point. Relationships between a variation point and its variants are are depicted like trees and annotated with cardinalities indicating the minimum number of variants which have to be selected for this relationship and the maximum number of variants which can be selected for it.

Gomaa [Gom04, GS02a] also defined graphical extensions to use case diagrams using stereotypes:

- <<`kernel`>>: distinguish the use cases that must be supported in all products (mandatory);
- <<`alternative`>>: distinguish the use case for which a choice must be made;
- <<`optional`>>: define use cases required by some but not all members; of the family. This distinction is given by a specific condition associated with optional use cases.

These stereotypes are used in combination with the relationships that can be defined to relate UML use cases as shown in Figure 2.21.

Some authors promote a hybrid approach to use case modeling in which both textual and UML use case diagrams are combined. In [JM02], John and Muthig define generic use case diagrams and generic textual use cases:

- Concerning the generic use case diagrams (Figure 2.22), they are divided into two parts: a first part that is mandatory for all products and a second part that vary amongst products;

```
USE CASE: CallAnswer
Goal:  Answer an incoming call on a [V0] Mobile Phone
Scope: The [V0] Mobile Phone
Precondition: Signal is available;Mobile Phone is switched on
Trigger: Incoming call
Primary actor : The user
Secondary actors: The { [V0] Mobile Phone} (the system)
                                  The Mobile Phone Company
Main success scenario
 1. The user accepts the call by pressing the Accept button
 2. The system establishes the connection by following the {
    [V1] appropriate } procedure.
Extensions
 1a. The call is not accepted:
     1a.1. the user presses the Reject button
     1a.2. scenario terminates

PL Variability Features
V0: Alternative:
       0. Model 0
       1. Model 1
       2. Model 2

V1: Parametric:
case (V0) of
0: Procedure A:
   2.1 Connect Caller and callee
1 or 2:  if V5= a then Procedure B:
            2.1 Interrupt the game
            2.2 Connect Caller and callee
else if V5= b then Procedure C:
           2.1 Save current game status
            2.2 Interrupt the game
            2.3 Connect Caller and callee

 V5  Alternative:
     a. games available, but if interrupted status is not
        saved
     b. games available, and if interrupted status is saved
```

Fig. 2.19: An Example of a PLUC (From [FGLN04])

- Concerning Generic Textual Use Cases (Figure 2.23), they describe variant text fragments and variant in the scenario using XML-like tags `<variant>` and `</variant>`;
- Variant use cases are instantiated during application engineering. The instantiation process is guided by a decision model, which captures the motivation and interdependencies of variation points. The decision model can also describe whether a use case is optional or whether it is alternative.

In addition to his graphical extensions, Gomaa [Gom04] also proposes an additional textual template for use cases as shown in Table 2.1.

Integrating Features with Use Cases

It may be interesting to combine feature diagrams and use cases, the former providing a clear overview of the variations handled within the SPL while the latter provides a more fine-grained

Use case name	Each use case is given a name. The name should be the goal as a short active verb phrase.
Reuse Category	Specifies whether the use case is kernel, optional, or alternative.
Summary	Describes the use case in one or two sentences.(i.e. a longer statement of the use case goal).
Actors	Names all the actors that participate in the use case, starts with the primary actor and is followed by the secondary ones. • Primary actor: name (actor who initiates the use case); • Secondary actor: name (actor who may participate in the use case).
Dependency	Describes whether the use case depends on other use cases, e.g. whether it includes or extends another use case.
Preconditions	Specifies one or more conditions that must be true at the start of the use case.
Description	Textual description taking the form of actor inputs, followed by system responses. The system is treated as a black box, that is, interest is taken in system responses not in the internals or the processes used. The main scenario defines a partial order for the set of operations related to the possible products.
Alternatives	This section provides a description for the alternative branches of the main sequence.
Variation points	Description of the variation points which can be handled directly in the use case by indicating the name, the functionality type (optional, mandatory alternative, optional alternative), the lines of the use case description concerned and a description of the variation point in natural English. If the variation point is too large then one should consider extending the use case with the mechanisms presented in the beginning of this section.
Postcondition	Identifies the condition that is always true at the end of the use case if the main sequence has been followed.
Outstanding Questions	This section documents questions about the use cases for discussion with the users.

Tab. 2.1: Gomaa Use Case Template [Gom04]

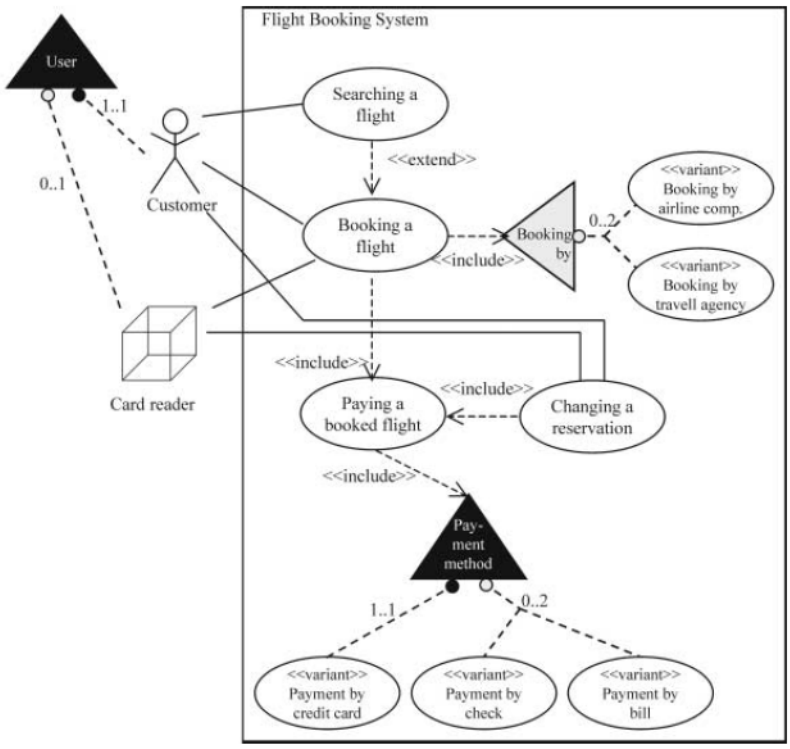

Fig. 2.20: Halmans and Pohl Notation [HP03]

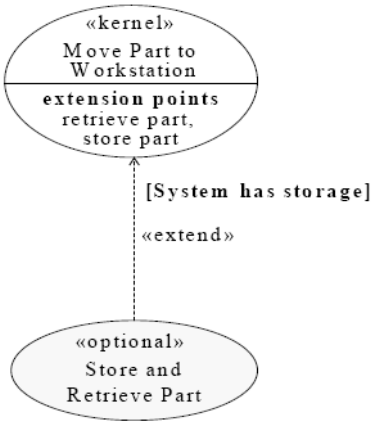

Fig. 2.21: Gomma Notation for Use Cases

description of product behavior via scenarios. In [GFdA98], Griss et al. integrate both approaches in order to extend their method with a domain analysis phase (see also Section 2.4.3). Use cases are simple text descriptions (with no information on variability) that are linked to features of the feature model.
In [vdML02], Van der Maßen and Lichter present an extension to the UML 1.4 metamodel for modeling variability within use case diagrams:

- `VariationPoint` is a metaclass specializing `ExtensionPoint` and is associated with a use case. It contains a list of relationships stereotyped as <<`alternative`>>. `Alternative` has a parameter `choice` which consists of pairs of conditions and use cases;

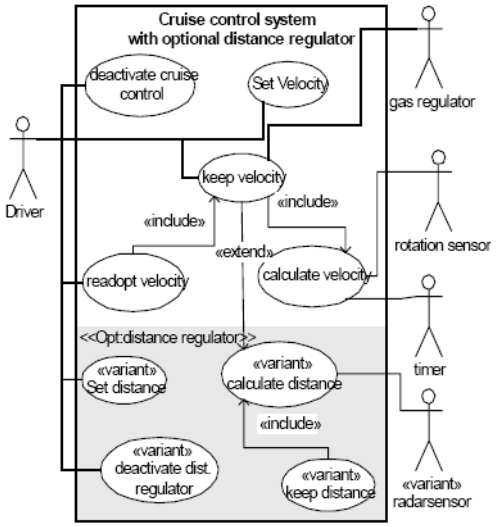

Fig. 2.22: Generic Use Case Diagram [JM02]

Use Case Name: keep velocity
Short Description: keep the actual velocity value over gas regulator
<variant> by controlling the distance to cars in front </variant>
Actors: driver, gas regulator
Trigger: actor driver, <variant> actor distance regulator </variant>
Precondition: --
Input: starting signal, velocity value vtarget
Output: infinit
Postcondition: vactual = vtarget
Success guarantee: vactual = vtarget
Minimal guarantee: The car keeps driving
Main Sucess Scenario:

1.) <keep velocity> is selected by actor driver
2.)get vactual, vtarget (<Calculate Velocity>)
3.) Does a distance regulator exist?
<variant OPT> get dactual, dtarget (<Calculate Distance>) </variant>
4.) Does a distance regulator exist?
<variant ALT 1: no; only cruise control>
- compare vactual and vtarget
If vactual < vtarget : gas regulator increase velocity
- restart <keep velocity>
If vactual < vtarget : gas regulator decrease velocity
- restart <keep velocity>
else restart <keep velocity>
</variant>
<variant ALT 2: yes, cruise control + distance regulator>
- - compare vactual and vtarget
If vactual < vtarget : gas regulator increase velocity
- restart <keep velocity>
If vactual < vtarget and atarget £ aactual: gas regulator decrease velocity
- restart <keep velocity>
If vactual < vtarget and atarget > aactual: gas regulator increase velocity
- restart <keep velocity>
else restart<keep velocity>
</variant>

Fig. 2.23: Generic Use Case Template [JM02]

- Options are modeled via an <<`option`>> stereotyped relationship. If the option requires

a condition to be chosen, the authors recommend to use the standard `Extend` relationship.

In addition, they use feature diagrams (according to the FODA notation) in a second step (after use case modeling) to capture the common and variable structural characteristics of the domain.

The approach for integrating feature models with use case diagrams has been generalized in the concept of *Orthogonal Variability Modeling* (OVM) [PBvdL05]. OVM promotes the principle that variability should be modeled as a first class artifact in a specific feature modeling language and related to domain engineering models via traceability links. This allows a consistent definition of variability at any abstraction level and a reduced size and complexity of domain engineering models. The FIDJI method also builds on that principle although it uses a notation and an application engineering process which are different from those.

SPL Architecture and Design

Even though the research community has carried out in-depth studies on addressing SPL requirements resulting in the approaches and models described above, architecture and design for SPL have not been studied as intensively. Some authors [GSCK04, BCK03, Bos00] notice that a SPL software architecture has some similarities with a single product software architecture and relies on component modeling techniques and patterns presented in Section 2.3. Concerning variable parts of the architecture, they propose models and methodological guidelines in order to identify variations points and support them via object oriented approaches.

Some authors go beyond these general considerations by providing detailing architecture and design models for software product lines. In an approach called FORM [KKL+98], Kang et al. have extended their FODA approach [KCH+90] to support other phases of SPL development. The authors propose to organize feature models in several layers: capabilities (functional and non-functional requirements), operating environment (platform attributes such as operating system or network context), domain technologies and implementation techniques (low-level details) each corresponding to a different perspective stakeholders may have on the system. These feature models are then mapped onto architectural models which comprises a "subsystem model" (static architecture) mapping capabilities, a "process model" mapping behavioral aspects, and a "module model" providing a detailed view on each of the system components and in which all feature model types are mapped to. General engineering principles (such as separation of concerns, information hiding or layering) are given to facilitate mapping between feature models and architecture in the domain engineering phase.

Gomaa [Gom04] proposes to apply <<`kernel`>>, <<`optional`>> and <<`variant`>> stereotypes to UML class diagrams in order to design the static architecture. The dynamic part of the architecture is mainly modeled via communication diagrams and state machines. Concerning communication diagrams, they directly depend on variability types (kernel, optional or variant) of the use cases they detail; kernel and optional are modeled the same way as for single product development (one communication diagram per kernel/optional use case) while variant communication diagrams are depicted either as extensions of the kernel ones or as a communication diagrams showing only new optional/variant features. In both cases, variants depend on guard conditions attached to the message defined between objects of the communication diagram. Concerning state machines, variability is modeled either via inheritance (one inherited

state machine for each variant class) or via parameterization, where variability is modeled via boolean guards in state machine transitions.

Atkinson and Muthig [MA02] propose a metamodel for variability management. It is mainly based on the definition of the stereotype <<variant>> that is used both for structural diagrams (class diagrams) and behavioral diagrams (communication and activity). In addition, a textual decision model defines the rules and relationships between variants to support product derivation. These models have been integrated in the KoBra [ABB+02] methodology that we will present in Section 2.6.

Ziadi et al. [ZHJ03] have defined a UML 2.0 profile to model variability in class and sequence diagrams. Concerning class diagrams, the following stereotypes have been defined: <<optional>> which refers to classes that can be omitted in products, <<variation>> which is defined on an abstract class to model a variation point, and <<variant>> which is used to annotate the class that can be chosen at a given variation point (variants are related to variation points via generalization relationships). Similar stereotypes for optionality and variation points are defined in order to define optional/variant lifeline and interactions (which contain a sequence of messages). In addition, <<virtual>> is used to annotate interactions that can be redefined in each product of the SPL. The profile is supported by a set of well-formedness rules that are expressed in OCL. In [ZJ06], Ziadi and Jézéquel propose a formalization (in terms of algebraic specifications) of their approach concerning sequence diagrams.

We believe that documenting variability within domain engineering models (both at the requirements analysis or late requirements and design levels) complicates the development of domain engineering models and limits flexibility in application engineering. We will elaborate on this issue in Chapter 3 and will define an original way to describe variability that support a more flexible product derivation process.

2.4.3 Product Derivation

Currently available approaches to support product derivation can roughly be organized in two main categories, according to the derivation technique used: configuration and transformation.

Product Derivation by Configuration

Product Configuration or software mass customization [Kru02, Kru06] originates from the idea that product derivation activities should be based on the parameterization of SPL core assets rather than focusing on how individual products can be obtained. Indeed, the final goal of product configuration is to avoid any application engineering activity, considered as harmful. When all SPL members can be completely characterized — which is both difficult to achieve and overly restrictive in the scope of the FIDJI method — an automated derivation process can be devised. In this context, product derivation relies on selecting product features according to the variants offered by the product line requirements description. Then, a configuration tool (called configurator) selects and assembles core assets automatically according to a decision model. This decision model contains the necessary constraints and traceability information in order for the configuration tool to make the right decision to generate a viable particular product.

Several configuration-based approaches [CHE05b, GFdA98, KKL+98] for product derivation based their decision models on feature models. We have seen above that Kang et al. [KKL+98] extended the FODA approach in order to support the description of domain assets at the design level. Their derivation process starts with requirements phase in which similar features to the features desired by a given customer are selected in the layers proposed (capabilities, operating environment, domain technologies and implementation techniques). Due to mapping between feature models and architecture as well as design artifacts, selecting on feature diagram results in an already configured design model for the product.

This idea of configuring feature models at various abstraction levels is also explored in [CHE05b] through the concept of "staged configuration": every time the user makes a choice in the feature model (such as selecting a particular feature or refining a cardinality), a new feature model is computed according to user choice at a lower stage. For a single product, the final stage will be such that once the variability being resolved, only one configuration corresponds to that feature model. Similarly to Kang et al.'s approach, feature models can also be separated according to some user-defined criteria but Czarnecki et al. go further by giving a complete formalization of their approach [CHE05a]. In order to perform the actual configuration of the product, Czarnecki and Antkiewicz [CA05] map feature models to UML activity and class diagrams via annotations. Annotations are represented visually on classes and activities with a color scheme (which can be kept tractable by carefully splitting diagrams) together with constraints mapped from original feature models and driving the configuration. In this approach, the staged configuration of feature models presented above results in the corresponding configuration of activity and class diagrams thus completing the product specification.
Product configuration based on feature modeling has also received commercial tool support such as Pure::Variants [Pur06] which provide a complete feature modeling environment integrated with IBM Eclipse IDE or BigLever GEARS [Big06] whose particularity is to act as a "bridge" between several product lines to configure a particular product.

There are also configuration approaches that do not base their decision model on feature model (or at least not in the usual FODA/FFD sense.
In [KWH05], Krebs et al. propose a decision model (called "Common Applicable Model" abbreviated to CAM) that relates features to their realizing software and hardware assets and the contextual information which provides additional constraints in a single model. This model is processed by tools [HWG00, G95] capable to infer (using a structure-based approach to configuration [WKHM04] to plan configuration tasks and logical reasoning techniques to process constraints) some of the software artifacts required to build the product. In order to support changes in SPL scope or core assets, the CAM needs to evolve. In [KWH04], change operations have been introduced in order to transform the CAM; these operations can be basic, such as adding a parameter value or deleting a concept or complex, i.e. composed of several other operations in a predefined order. These operations are useful to make the SPL evolve: adding new artifacts (and relationships between them), removing existing ones, defining new constraints, etc. This decision model is part of the ConIPF [HWK+06] methodology, stemming from a European research project aiming at validating product configuration in an industrial context.

Van Ommering [vO02] has designed an architecture description language called Koala to define the product architecture based on a pool of components that may be reused from different product lines (this approach, called "product population", is used with Philips TVs, DVD players/recorders etc.). Koala models offer, at the domain engineering level, the possibility to define expressions to connect components ("switches") or to define parameters that will be grouped

in the required interfaces ("diversity interfaces") of a given component. Product Derivation is performed by assigning values for parameters and switches, then a compiler will automatically configure the product according to these values.

PuLSE-I [BGMW00] uses decision models realized during a domain analysis phase called PuLSE-CDA [BMW99]; the "product line" model contains multiple workproducts and the relationships among them as well as commonality and variability information. The "Domain Decision Model" attaches decisions to variants using meta-attributes on workproducts; decisions act as rules (that may depend on other decisions already made) which guide in selecting variants. Therefore, when all the decisions have been made, the configured domain model corresponds to the specification of a single product. The same mechanism is used at the architectural level (which also owns "architecture" and " architecture decision" models). It should be noted that PuLSE-I has envisioned the fact that some of the SPL assets might not correspond or fail to fully implement the product. In this case, additional elements have to be modeled manually and integrated with the configured model of the product.

Product Derivation by Transformation

We believe that MDE techniques, by providing models as useful abstractions to understand assets and transformations able to use them as first-class artifacts for product generation, has clearly a major role to play in SPL engineering [GP04] and product derivation, especially through model transformation. This point of view is shared by several researchers [GSCK04, ZJ06, HMPOS04, KMHC05]. As we have shown in Section 2.1, MDE is still — and, to a lesser extent, academic research on software product lines — in its infancy in numerous domains. Therefore, most of the work currently addressing the synergy between SPL and MDE is currently ongoing. We examine the current transformation-based approaches in the remaining of this section.

In [HMPOS04], a conceptual model for SPL engineering aligned with MDA models is presented. A requirements view (CIM level, called "product line model") of the product line is described using UML 2.0 use cases. Individual products are specified in the "product model" (CIM level and subset of the product line model), which takes the form of an actor having association relationships with some of the product line uses cases and may be related to non-functional properties described as stereotyped classes. Core assets (PIM level, called "system family model") are described in terms of UML 2.0 composite structures diagrams extended with variations points described via stereotypes. A QVT transformation relates elements of the product line model to those of the system family model. The actual product derivation takes the form of a partly automated transformation that takes the product and system family model as inputs to output an instantiated version of the system family model, called "Product/System Model" which is also a PIM. Finally product implementation is obtained after several refinements at the PSM level.

In [KMHC05], Kim et al. analyze the respective shortcomings of SPL engineering and MDA, and propose an overview of a complete method integrating both approaches. In particular, for product derivation, the authors propose to instantiate, via MDA transformation mechanisms, a framework embodying core assets on the basis of a decision model and according to the variants selected for a specific product. Then, for application parts which are not implemented in the framework, an integration phase takes place, resulting in a single product model at the PIM

level. This model may be refined to obtain the PSM and ultimately application in a similar way to Haugen et al. [HMPOS04].

Finally, the most comprehensive transformation approach to product derivation can be found in [ZJ06]. Unlike the other approaches that cover the whole SPL engineering activities, Ziadi and Jézéquel put the emphasis on product derivation at the design level both for static and behavioral aspects. Static models are described in terms of UML class diagrams extended with the UML profile we have presented above (see "SPL Architecture and Design"). The derivation process uses a decision model taking the form of a design pattern to display the variants available for each product. In a first derivation step, variants are selected in this model and relevant classes are automatically selected. In the second and third steps, unused variants are removed and the model is optimized. The derivation process is described in an imperative pseudo-code format supported by the MTL transformation language. Behavioral derivation is based on the synthesis of state machines from scenarios that have been algebraically formalized at the domain engineering level. First, variability is resolved by interpreting the algebraic expressions on a decision model that defines the choice made for each variation point. The synthesis process generates "flat" state machines from individual sequences and then combines them so that they map sequence diagrams with combined fragments.

Both automated transformation and configuration approaches require that a decision model be fully defined, implying that all variation points are fully known, which may not meet customers' expectations. PuLSE-I [BGMW00] has envisioned the possibility that additional features might be developed but provides no support for them. The FIDJI product derivation process enables product engineers to define their own variation points by directly transforming the domain engineering artifacts in a controlled way. We will describe this approach in Chapter 3.

2.5 Object-Oriented Frameworks

In this section, we describe the notion of object-oriented framework which is one of the pillars of modern software development. We specifically insist on relating object-oriented frameworks concepts to those of software product lines. We then introduce the more recent concept of enterprise architecture framework as a way to structure object-oriented framework artifacts.

2.5.1 Definition

Object-Oriented Frameworks have emerged from the need to provide better reuse artifacts than class libraries initially provided with object-oriented languages [Dah68]. In particular, the objective was to reuse a more coarse-grained solution than a particular functionality embodied into a class. Early examples of object-oriented frameworks are related to the Smalltalk-80 language [GR83]; one of the most famous examples is the Model View Controller (MVC) that smalltalk provided for user interface management.

Several definitions exist for object-oriented frameworks [JF88, FS97, Szy98]. In this thesis, we will use the definition of object-oriented framework given by Johnson and Foote [JF88]:

Definition 14 (Object-Oriented Framework) *An object-oriented framework is a set of classes that embodies an abstract and reusable design for solutions to a family of related problems in a particular domain.*

Considering this definition, it is obvious that object-oriented frameworks and software product lines are related to each other: "solutions to a family of related problems" can be understood as different members of a SPL and "abstract and reusable design" may easily be related to the notion of SPL core assets. As discussed by Tourwé (see [Tou02], Chapter 2), an object-oriented framework design captures structural and behavioral relationships amongst its classes. In order to achieve reusable and sound structural organization of classes, object-oriented frameworks heavily rely on design patterns [GHJV95] which are proven solutions to recurrent problems. However, object-oriented frameworks are more concrete than design patterns, and combine several instances of different patterns to form a highly specialized solution. Concerning behavior, a distinction have to be made between object-oriented frameworks and class libraries; indeed, object-oriented frameworks own a specific thread of control while class libraries do not. Therefore, an object-oriented framework can be seen as a partial application that will be extended in the context of a particular development as defined by Fayad and Schmidt [FS97]. In [Pre95, Pre00], Pree introduced the notion of "spots" to conceptualize object-oriented framework flexibility; frozen spots represent parts of an object-oriented framework that are fixed and cannot be adapted while developing an application with it. Hot spots, for their part, describe parts of a framework that can be extended for a specific application as needed; this is the process of *instantiation* that will be discussed below. Hot spots are typically defined in terms of abstract classes and interfaces while frozen spots may involve concrete classes as well. Drawing an analogy with product line concepts, frozen spots correspond to commonalities while hot spots correspond to variabilities in the domain addressed by the SPL.

Over the years, various object-oriented frameworks have been developed in several domains. For instance, Fayad and Johnson gathered in [FJ00] object-oriented frameworks belonging to

the following domains: manufacturing, distributed systems, networks and telecommunication. Object-oriented frameworks can also be categorized according to the abstraction level in which they provide their functionality. For example, Fayad and Johnson [FJ00] defined the following levels:

- **System Infrastructure**: object-oriented frameworks in this category typically provide solutions for accessing hardware resources, implementing user interfaces (JAVA SWING [HWL+02]), supporting network protocols or are provided with programming languages (Microsoft .NET framework [RR02]...);
- **Middleware Integration**: object-oriented frameworks in this category address interactions between components within a distributed setting. This includes CORBA related frameworks (Borland Visibroker [Bor], IONA Orbix [ION]), J2EE containers (IBM WebSphere [IBM], BEA Weblogic [BEA]) or web-services (Apache Axis [Fou], etc.);
- **Enterprise Application**: object-oriented frameworks in this category provide core business functionality in a broad domain range. In the e-business domain, examples include the discontinued IBM San Francisco project [RCB98] or commercial frameworks for enterprise resource planning such as SAP [Her05].

Naturally, these levels are connected: a middleware integration object-oriented framework may use a system infrastructure level providing low-level network functionality and an enterprise application level may need the service of distributed component technologies to provide distributed functionality on heterogeneous platforms and networks.

2.5.2 Instantiation

An object-oriented framework only offers an abstract and/or partial design to an application. We need to complete it with specific code in order to complete to the final application. This activity is called *instantiation* [Tou02]:

Definition 15 (Object-Oriented Framework Instantiation) *Deriving an application from an existing framework is called instantiating (or specializing) that framework. A concrete application derived from a given framework is called an instance, an instantiation or a specialization of that framework.*

In accordance with the analogy drawn earlier, if an object-oriented framework implements core assets for a SPL, then object-oriented framework instantiation represents the actual process of product derivation. The instantiation process is realized by completing hot spots, which can be done in two ways [Pre00]:

- **Inheritance-based instantiation**: in this approach, we use the inheritance mechanism of object-oriented languages to extend classes (which are most of the time abstract in hot spots) and override methods (which are often called *hook* methods because they form the interaction points between object-oriented frameworks and application behaviors). Doing so requires that developers have access and understand the object-oriented framework implementation. This is why object-oriented frameworks providing inheritance-based hot spots are said to be *white-box* [JF88]. A white box object-oriented framework is proactive [Bos00]; it calls the various parts of the application according to its thread of control. From the application viewpoint, it realizes an *inversion of control* [JF88] also known as the *Hollywood Principle*: “Don’t call us we will call you”;

- **Composition-based instantiation**: in this approach, the final application behavior is obtained by different class compositions. Thus, in this case, the application developer does not need to know all object-oriented framework implementation details but just the way classes can be combined (through the knowledge of their interfaces, for example). Therefore, object-oriented frameworks providing composition-based hot spots are said to be *black-box* [JF88]. Indeed, these different class combinations can bee seen as parameterizations of the object-oriented framework requiring less application development effort since the work amounts to providing sound combinations of object-oriented framework classes instead of extending them. Black-box object-oriented frameworks can be thought as white-box ones which have matured so that class extensions are forming default and concrete behavior and are integrated in the object-oriented framework.

As noted in [Bos00, Pre00], no real object-oriented framework is entirely white-box or black-box; in general, object-oriented frameworks combine both approaches where most stable parts are reused via composition and some are specialized via inheritance. The same applies for control, the object-oriented framework calls some parts of the application while the application controls some classes of the object-oriented framework. With respect to SPL concepts, we can relate the compositional object-oriented framework instantiation to the configuration approach (Van Ommering product population approach [vO02] is an example) while the inheritance-based instantiation is more in alignment with transformation approaches (Ziadi and Jézéquel [ZJ06]).

2.5.3 Documentation

Depending on abstraction level (system, middleware and enterprise application) and how their domain is scoped, object-oriented frameworks can form quite large class infrastructures. However, as noted in Chapter 2 of Tom Tourwé thesis [Tou02], design patterns typically introduce extra abstractions and indirections between object-oriented framework artifacts that are difficult to understand. Therefore, application developers need a sound documentation providing them with design guidelines on how to instantiate the object-oriented framework. It should be noted that this documentation can be *active*: in addition to providing guidelines for application developers, they may also provide technique to actually instantiate the object-oriented framework. In the remaining paragraphs we sketch some of the most common approaches to object-oriented framework documentation.

Documentation approaches can be divided in black-box and white-box ones, addressing different purposes. Black-box approaches provides guidance on how to instantiate the object-oriented framework while ignoring design and implementation details:

- **Examples**: Examples are the simplest way to document an object-oriented framework. They are predefined instantiations of an object-oriented framework the developer can review to understand object-oriented framework design and possibly adapt those examples as needed;
- **Cookbooks**: A cookbook [Joh92, Pre95, FPR00] is a collection of recipes to instantiate the various hot spots of an object-oriented framework. Recipes usually give an intent (e.g. adapting the behavior of a particular class), a description of the classes involved ("ingredients" to apply the cooking metaphor) and steps to follow in order to instantiate the hot spots. Recipes can be systematically structured using a use case-like format called reuse case [BGK98];

- **Task-based**: Task-based approaches can be seen as an active version of recipes. Recipes are embedded in integrated environments that automatically guide the developer when he instantiates the object-oriented framework by showing which tasks have been completed and which remain to complete to finalize instantiation. Examples of this kind of environment includes FRED [HHK+01a, HHK+01b] or Ortigosa and Campo HiFi tool [OC00];

- **Constraints**: Another way to help developers instantiating an object-oriented framework is to provide them with logical constraints avoiding misuse. In [HHR04], Hou et al. describe a language called Framework Constraint Language (FCL) that is able to ensure that the properties governing the instantiation of an object-oriented framework are verified. FCL is based on first-order logic and has a type system that covers object-oriented model basics (classes, methods and variables as well as inheritance relationships). This language was initially designed to operate with C++ based object-oriented frameworks but has recently been extended and renamed to Structural Constraint Language (SCL) to also support JAVA programs [HH06]. Moreover, tool support has been designed in the eclipse environment. The toolset works as follows: from source code, it extracts a base of facts through syntactic and semantic passes of the compilers. Then, it checks whether the boolean formulas composing the FCL specification are satisfied. We will extend this approach at the requirements and design levels as a support to control product derivation and document SPL variability (see Chapter 3).

On the contrary, white-box-based documentation is concerned with the implementation details of object-oriented frameworks and includes:

- **Patterns**: As we mentioned above, object-oriented frameworks embody a great number of design patterns that need to be understood in order to perform object-oriented framework instantiation. In [BJ94], Beck and Johnson propose a template for documenting design patterns comprising preconditions to be fulfilled before the pattern application, description of the problem, constraints on its application (i.e. trade-offs to be made between performance and flexibility) and a brief summary of the solution. This template is then exemplified in the HotDraw object-oriented framework. This format was accompanied with diagrams showing class structures and their mutual interactions with each other (ancestors of UML class and sequence diagrams) and code realizing their usage as illustrated in [GHJV95]. In [Tou02], Tom Tourwé proposes to use "metapatterns" (which can be thought as abstraction of design patterns) to define transformations in the object-oriented framework that can both be used for its instantiation or evolution. These transformation are specified in a logic programming language before being implemented in an object programming language;

- **Interfaces**: Interface contracts [Mey92] provide the specification of obligations for a class as well as invariants in isolation [BD99]. A careful documentation of class interfaces is one step towards instantiation support as shown in [Vil03];

- **Exemplars**: In [GM95], the authors propose to document object-oriented frameworks in a top-down fashion by providing instantiation of their abstract classes through *exemplars*; for each abstract class, at least one concrete class has to be provided. This documentation is provided by means of a visual object-oriented modeling language supported by a modeling tool. Once the user has selected the desired exemplar, he can then adapt it to its application by changing the concrete classes;

- **UML Profiles**: Several UML profiles have been elaborated to [FTV02, FPR00, SA02]. The most exhaustive work in this field is the UML-F profile [FPR00]. This profile allows to specify hot spots and frozen spots. UML-F is based on the extension of two UML 1.4 diagrams: class diagrams and sequence diagrams. For class diagrams, the notation provides constructs for specifying which classes and methods can be adapted (and the kind of adaptation statically via inheritance or dynamically via class-loading) and those that are fixed, as well as overriding and inheritance relationships between modeling elements. For sequence diagrams, UML-F supports the definition of alternative scenarios, loops amongst methods and objects (such a construct is now available in the last UML 2.0 specification). In summary, the UML-F profile provides a notation to model commonalities and variabilities in object-oriented frameworks. Some constructs of these profiles can be related to the work realized in product line modeling with UML [ZHJ03, Zia04].

As a remainder of analogy with software product lines, white-box documentation refers to the problem of modeling commonalities and variabilities in domain engineering models while black-box techniques focus on assisting product derivation. Therefore, similarly to instantiation techniques, a combination of both approaches have to be used in order to inform developers on the object-oriented framework's design and to help them develop products efficiently. As we have seen, MDE provides means to describe systems at various abstraction levels and transformation techniques to generate those systems. Thus, object-oriented framework documentation and instantiation can be treated uniformly, resulting in increased understanding and productivity of developers, if we combine the concept of object-oriented framework with MDE. Achieving this combination is one of the main contributions of this thesis and will be presented in Chapter 3.

2.6 Development methods

In this section, we explore the development methods that are of interest for the FIDJI method described in this thesis. We will particularly emphasize on the analysis and design phases of the methods considered since they lie within to the scope addressed by FIDJI.

2.6.1 Fusion/Fondue

Fusion

Fusion [CAB+94] is an object-oriented development method having its origins in the OMT [RBP+91] and Booch [Boo94] methods. It consists of a waterfall process organized in three phases: analysis, design and implementation. Each phase is supported by graphical and textual notational elements and guidelines to construct models with those elements and transition to the next phase. In addition to the notational elements provided for each phase, Fusion emphasizes on the necessity to build and use a *data dictionary*: it is a central repository of definitions listing the terms and concepts used for the development of the system.

Analysis

The analysis phase treats the system under study as a black-box focusing on its interaction with the environment. Realizing the analysis phase of a system with Fusion involves building the *object model*, *system object model* and *interface model*.

The role of the *object model* is to define the domain of the problem addressed by the system. It is composed of classes (which instances are objects) and relationships between them. Classes have attributes of primitive types such as Integer, Boolean, String and Enumeration but cannot be a class. These classes do not have any operation; indeed, at the analysis level, we are interested in the data and processing required to model the problem at the system level. For example, a requirement for an e-banking system can be formulated as follows: "the system should be able to update my balance". At this point, it is not relevant to determine which class will be responsible for handling this requirement. This responsibility will be attributed during the design phase. The object model is represented graphically in an extended form of the Entity-Relationship model [Che76] and allows a class to have an internal structure representation.

The *system object model* is a subset of the object model that defines the boundary between the system and its environment. This boundary is shown as a dashed curve on an object model.

The *interface model* focuses on describing the system behavior. This behavior is described in terms of input/output events and the changes they may cause in the system state. A system is modeled as a reactive entity that interacts with other reactive entities called *agents*, which can be either human users or hardware or software systems. Communication between a system and the set of agents forming it environment is enabled through *events* which are instantaneous and atomic units of communication. Input (from an agent to the system) and output (from the system to an agent) events are sent asynchronously i.e. the sender does not wait for a response

to the event it has sent. The event can be accompanied with data values and objects. When a system receives an event, it can cause a state change and event outputs. An input event and its effect on a system is called a *system operation*. At any point in time, only one system operation can be active. In order to determine the interface of the system, i.e. the set of system operations it can respond to and the set of events it can output, Fusion offers, as an option, to model sequence of events between agents and the system with timeline diagrams (a simplified form of sequence diagrams without alternatives, hence several timeline diagrams are required to detail a scenario) completed with textual descriptions (but not as structured as use cases).
The interface model is comprised of two models: *operation model* and *life-cycle model*.

The operation model "specifies system operations declaratively by defining their effect in terms of change of state and events that are output" [CAB+94]. In fact, system operations are modeled as black boxes with preconditions and postconditions. The operation model is constituted of textual descriptions following an operation schemata composed of the following fields:

- **Operation**: A unique identifier for a system operation;
- **Description**: Description of the operation in natural language;
- **Reads**: List displaying variables the operation may access but not change. They can refer to an object, attribute or relationship belonging to the system object model;
- **Changes**: List showing variables the operation may change. They can come from an object, an attribute or a relationship belonging to the system object model. The keyword **new** indicates that the system operation introduces a new object in the system state;
- **Sends**: All the agents and the events the operation may send to them. For each agent, the list of events that can be sent to it is enclosed in curly brackets;
- **Assumes**: A boolean precondition formulated in natural language on the system state;
- **Result**: Postcondition of the operation. The keyword **initial** (respectively **final**) indicates a value before operation invocation (respectively on completion). This postcondition is also composed of the description the events that are output.

The life-cycle model is composed of expressions that "describe the allowable sequences of interactions that a system may participate in over its lifetime" [CAB+94]. These expressions follow a regular-expression approach [DDQ78].

Models offered by Fusion at the analysis level cover static and behavioral aspects of a system and are detailed enough to provide a precise and concise description of system functionality while maintaining an acceptable level of accessibility with a mix of simple graphical and textual notations. However, two main remarks can be made.

First, Fusion does neither have any requirements elicitation phase (requirements are assumed to be written by a customer and provided to the software supplier. And nowadays, requirements elicitation is more the result of an interactive process between a software supplier and its customer) nor provide any way to link analysis models to software requirements. This is an issue especially for the determination of the system object model which defines the system boundaries. We believe that the link between requirements elicitation and analysis should be made

explicit in order to assess system functionality with respect to customer expectations. FIDJI places scenarios (described as extended use cases) at the center of the analysis process and link them to a general description of the product line commonalities and variabilities.

Second, we argue that, for a non-trivial system, capturing all possible sequences of system operations is difficult if not impossible because it forces the analyst to reason on all the states in which the system can be in order to define Fusion life-cycle model. FIDJI partitions life-cycle reasoning within scenarios so that the set of states to consider is greatly reduced. Furthermore, we do not ultimately seek to synthesize the global life-cycle of operations for the SPL products but focus on the mandatory and unauthorized scenarios.

Design

The aim of the Fusion design phase is to provide a concrete object-oriented structure of the abstract specifications produced during analysis of the system. In particular, it states 'how' the system is implemented, that is, it provides the algorithmic information necessary to fulfill the goals declaratively described in the operation model and assigns this information to concrete classes satisfying the abstract description of the system object model. The design phase is based on building the following elements: *object interaction graphs*, *visibility graphs*, *class descriptions* and *inheritance graphs*.

The first step of the design phase is to describe how the abstract behavior of the operation model is supported. This is done by describing the messages exchanged between objects. For each system operation, an *object interaction graph* is constructed defining the objects involved and how they communicate. The notation for object interaction graphs is composed of boxes (representing design objects involved) and arrows for message passing (these graphs are the ancestors of UML 2 communication diagrams). Amongst the objects involved in an object interaction graph, one is responsible for handling the request to invoke a system operation: the *controller*. Therefore, in order to be complete, each object interaction graph should have an identified controller responsible for performing each system operation defined in the operation model.

The second step of the design phase precises the accessibility between objects in order for them to satisfy the communication described in the object interaction graph. This is the purpose of the visibility graphs to provide such descriptions. In particular, a visibility graph identifies the following elements for each class:

- Objects the class needs to reference;
- Appropriate kinds of references to these objects. These references can be permanent or dynamic (depicted by plain or dashed arrows between boxes), defining exclusive or shared access for client objects, bound (i.e. deleted if the related object is deleted) or not, and lastly mutable over the time or not.

The final step of the design phase is to fully describe classes with their attributes and methods as well as inheritance relationships. Class description follows a textual template, while the inheritance graph distinguishes, via filled and empty triangle, generalizations that are incomplete (they may be new classes that specialize a particular class) or complete (the set of subclasses

partition the superclass). This mechanism is also found in UML via the `GeneralizationSet` construct [OMG05f].

Fusion does not provide any means to model system architecture, the method directly proceeds to detailed design. As we have seen, architecture modeling is crucial to understand large and complex systems and to ensure that they will achieve certain qualities using architectural styles. In a discontinued effort, HP proposed an enhancement to Fusion, called Fusion 2.0 [HP98], which includes an architecture development phase based on the definition of component architecture and architectural styles usage. However, they do not provide detailed modeling constructs to define such elements.

Fondue

Fondue [SBS04, Sen02] is a derivation of Fusion which uses UML (2.0) models. Figure 2.24 shows the models offered by the method.

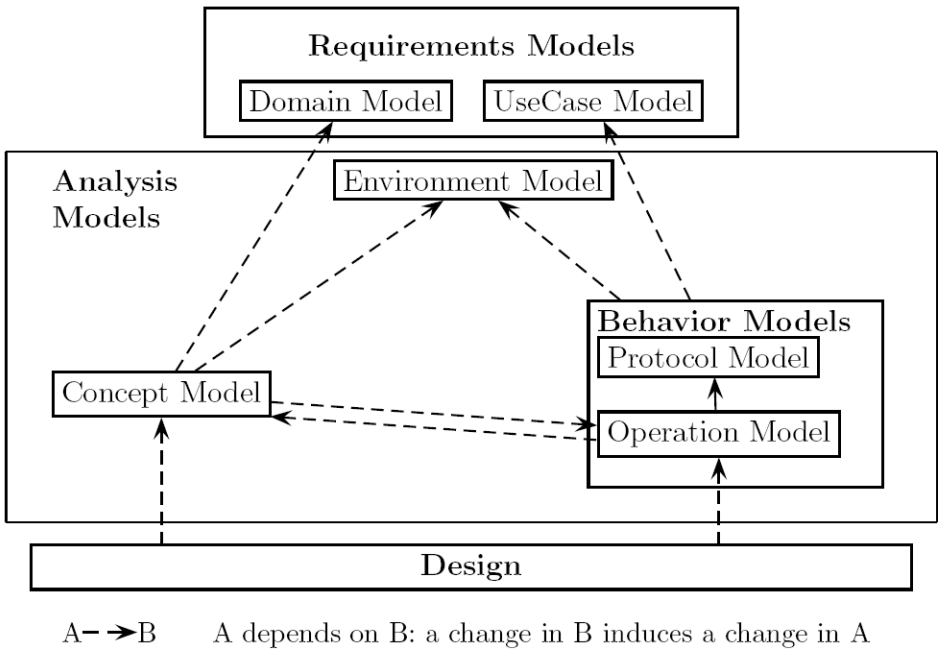

Fig. 2.24: Fondue Models and their Inter-Relationships (From [SBS04])

Table 2.2 shows the correspondence between Fusion and Fondue models as well as the UML elements used. Fondue explicitly links analysis models to use case description which helps to drive the determination of the environment model. Moreover, Fondue has particularly contributed to behavior precision thanks to OCL. Preconditions and postconditions of the operation model are specified with OCL which gives a more formal description of the conditions on the system state while still being easy to read [Sen02]. FIDJI also uses OCL for system operations and proposes for the fusion life-cycle model to formalize use case descriptions via OCL pre/postconditions and state variables. This has the advantage, over the protocol model, not to be exhaustive about the chronological event order.
Partial tool support for Fondue was developed: firstly, a compiler to validate syntax and check OCL expressions for operation schemas was proposed [Gup01]. Secondly, a metamodel for Fon-

due integrated to the UML CASE tool Borland Together in order to assist the edition of Fondue models was built more recently [LGL05].

Fusion Model	**Fondue Model**	**Fondue UML Notation**
Object Model	Domain Model	Class Diagram
System Object Model	Concept Model	Class Diagram
(Scenarios: text)	Use Case Model	Use Case Diagram & Descriptions
Interface Model	Environment Model	Collaboration Diagram
(Scenarios: Timeline Diagrams)	Scenarios	Sequence Diagram
Life Cycle Model	Protocol Model	Protocol State Machines
Operation Model	Operation Model	Operation Schema + OCL
Object Interaction Graph	Interaction Model	Collaboration Diagram
Visibility Graph	Dependency Model	Class Diagram
Inheritance Graph	Inheritance Model	Class Diagram
Class Description	Design Class Model	Class Diagram

Tab. 2.2: Fusion & Fondue Models (adapted from [SBS04, LGL02])

Fondue has also been extended in a method, called "Enterprise Fondue" [SS03], that integrates separation of concerns [TOHSMS99], aspect oriented programming [Kic96], component-based software engineering [Szy98] and model-driven architecture. Enterprise Fondue is organized in five layers:

- **Component-based layer**: Describes an organization of the system in terms of business components and relationships among them;
- **Concern-based layer**: Once business components have been identified, the fondue models are used to actually define them completely;
- **Technology-dependent layer**: Contains business components models refined for a particular technology (CORBA,J2EE) thanks to UML profiles;
- **Platform-dependent layer**: Models of the technology-dependent layers are enriched with specific information on the platform offered by a given vendor (e.g. BEA and IBM for J2EE application servers);
- **Language-dependent layer**: Contains the application code in which the various concerns covered in the second layer are weaved using aspect-oriented programming techniques.

Enterprise Fondue follows a waterfall process and all the layers described above are related through of model transformations.

Fondue and Enterprise Fondue make contributions to the Fusion method at the analysis and design levels. These contributions use UML as a standard modeling notation, OCL for precision, and recent achievements in component engineering to address the complexity inherent to the architecture and design of distributed systems. However, these works having been carried out independently, there is no clear integration of these contributions in the fondue process. Furthermore, Enterprise Fondue does not detail the modeling language provided with the method and transformations proposed to generate the successive layers.

2.6.2 Rational Unified Process

The Rational Unified Process (RUP) [Kru03] is a derivation of a generic process called the Unified Software Development Process [JBR99] and uses UML as its main modeling language. RUP stems from the idea that it is risky to get "everything right" before going to the next phase and that the cost of an early error is so high when tackled at the end of the development process that it should be minimized. Therefore, RUP proposes to divide the development process in *iterations*. Each iteration can been seen as a waterfall model passing through all software engineering activities (requirements, design...) thus allowing to accommodate a change in any of these activities in the next iteration. Iterations are grouped in phases which characterize the emphasis that should be put on the various activities depending on the overall progress of the development project (see Figure 2.25). RUP identifies four phases:

- **Inception**: This phase focuses on establishing a business case of the system: i.e. all the entities that interact with the system and the nature of these interactions. The outcome of this phase includes informal description of the system functionality (summary use cases), financial and risk assessment documents, project glossary and initial development plan of the system;
- **Elaboration**: The role of the elaboration phase is to have a complete understanding of the software. This entails the creation of a requirement model (80 % of all use cases should be present) a software architecture description, prototypes, a development plan, revised risk and business case. In particular, the architectural description of the system may be built with UML according to the renowned "4+1" views [Kru95];
- **Construction**: The construction phase concerns the main development of the product together with its careful testing and its accompanying user manual;
- **Transition**: Finally, the transition phase involves deployment, maintenance activities for the product.

It should to be noted that in addition to iterating within a particular phase, the four phases can be enacted incrementally as noted by Sommerville [Som04]. Each run through the four phases is called a *cycle* and produces a *software generation*. Kruchten [Kru03] also mentions that these cycles may overlap (the transition phase of one cycle may include the inception and elaboration phases of the next cycle).

RUP is not prescriptive in the models it offers. For example, while the 4+1 approach led Kurchten to define RUP as an architecture-centric processs [Kru03], no specific notation — if we except the fact that RUP as a whole is based on UML as it primary modeling language — is provided to natively support these viewpoints. In his vision of RUP, Larman [Lar02] uses UML packages to organize system architecture in terms of layers and partitions. Other diagrams

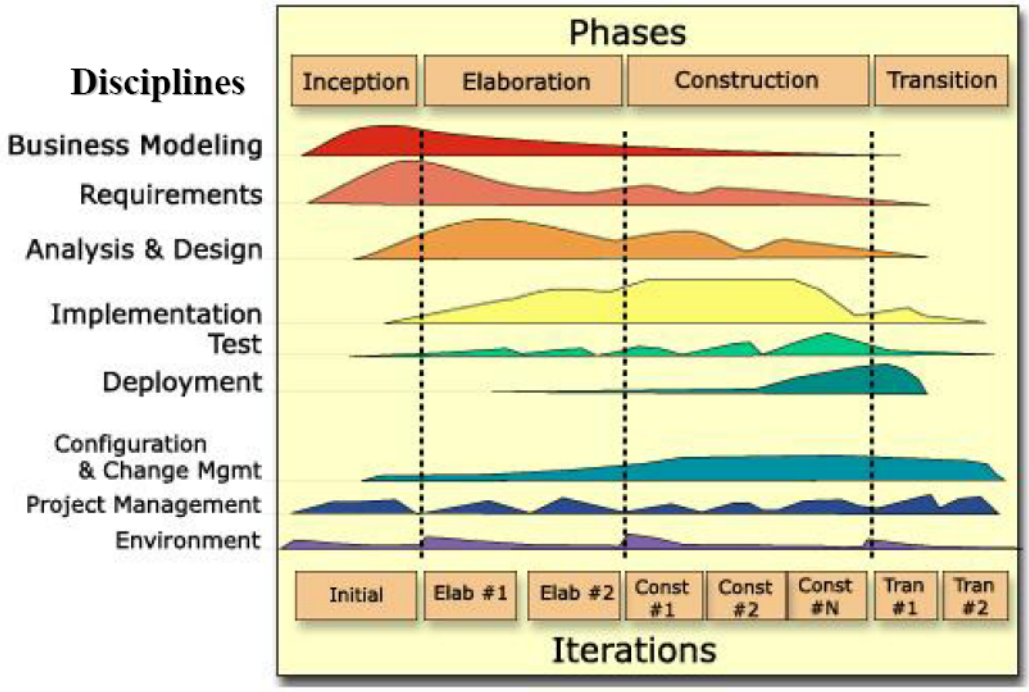

Fig. 2.25: RUP Phases & Iterations ([Kru03])

are used to provide their internal structure and behavior (class and interactions), and major decisions concerning architecture rationale are captured in the Software Architecture Document (SAD). However this notation fails to describe relationships between components and hence does not support the modeling of component-and-connector views.
Furthermore, RUP is a very general process framework and needs to be adapted to a particular organization; depending on the size and type of the project all artifacts do not need to be built (see chapter 17 in [Kru03] and [Lar02]). Furthermore, although the RUP process gives a set of good software engineering practices, it does not transform them in useful modeling elements accompanied with concrete building guidelines. Therefore, quality of the method depends on the adaptation done to apply it in concrete developments.

2.6.3 Catalysis

Catalysis [DW99] is an component-based development method that uses UML and relies on the following three principles:

- **Abstraction**: In Catalysis, abstraction is used in two ways. The first one aims at providing descriptions of the system uncluttered of any details that would not be relevant at a given stage of the development of a system. The second one strives at prioritizing choices made about the system throughout its development; thus, it is possible to make design decisions before coding and to reuse some of them without considering low-level issues such as the particular technology in which the system will be implemented. The various Catalysis models corresponding to a particular development phase are then obtained via refinement techniques;

- **Precision**: Abstraction, however, does not necessarily means unclarity. Catalysis' authors

claim that precision is related with the capability of a model to be refutable and to provide traceability amongst artifacts;

- **Pluggable Parts**: Catalysis emphasizes on building systems via the assembly or adaptation of components with well-defined interfaces. Component reuse should occur at each phase of the system development, i.e. from requirements to code.

Catalysis does not propose a particular development process but rather a set of patterns to apply the method depending on a particular context. Patterns are ordered according to the development phase of the system they apply to. Catalysis identifies the following major development phases:

- **Business**: Understanding the terms used by the stakeholders of the system (dictionary), capturing business rules and constraints independently of the software solution to be provided. A business model is generally composed of class diagrams for concepts, OCL invariants for constraints and use cases (specified using OCL pre/postconditions) detailing business actions. All these elements are informally described in a dictionary;
- **Component Specification**: The goal of this phase is to declaratively describe the behavior of each system component. It typically involves building a "type model" that is refined from the business model. This type model hence contains objects and actions but is complemented with a statechart detailing the acceptable flow of events the component can handle, use case descriptions and user interface descriptions in the form of storyboards [LM96];
- **Component Design**: A detailed design of the system is made in order to satisfy the type model. This involves building or reusing existing components and assembling them in a well defined architecture. Catalysis proposes a "Component-Port-Connector" model which is quite similar to the component and connector viewpoint; ports define the provided and required interfaces of components, while connectors are dedicated to message passing or operation calls. Connectors are typed and it is possible to specify their behavior in terms of OCL pre/post-conditions connecting two or more ports.

A specific combination of patterns composes a "route". Catalysis provides basic routes to develop a system form scratch, to reengineer one or to handle legacy components. These routes are just examples; depending on the context, every organization can define its own route through the method by wisely combining individual patterns amongst phases. In addition, Catalysis authors state that the process is nonlinear, iterative and parallel. For example, a reengineering context may involve building type models form existing components design while defining the business model for the new system. The last step would be to combine type models in a single type model and use it to refine the business model.

Therefore, Catalysis is as flexible as the Rational unified process but provides, thanks to refinement clearer relationships between the artifacts of the systems. However, Catalysis is not an integrated method if we compare it to Fusion; we have to choose amongst the different routes in order to form a complete development process. We believe that, in order to be practical, a development method should offer a unique process ("route") adapted to the state of the practice in a given domain guiding software engineers from early requirements to design, as well as a modeling language, which elements are tailored to describe the information required for each step of the method. Furthermore relationships between these elements have to be clearly mentioned.

2.6.4 KoBra

KoBra [ABM00, ABB+02] is a product line development method that is an object-oriented derivation of the PuLSE approach presented in Section 2.4. Korbra builds on the same synergy that we outlined in Section 2.5 between product lines and object-oriented frameworks concepts. The fundamental concept of the KoBra approach is the Kobra Component or "Komponent" [8] that is assembled with other komponents to form a framework on which individual products will be developed. A komponent is described at two abstraction levels:

- **Specification**: Specification models define the externally visible properties of the komponent. They include a structural model in the form of an UML class digram that exposes the class and operations available at the interface of the komponent, a functional model described as Fusion operation schemata specifying individual operation behavior and a behavioral model using UML statechart notation describing how the komponent react to external stimuli. In addition, a decision model adopting a tabular notation describes the effect a particular choice has on the aforementioned models;
- **Realization**: Realization models detail komponent internal design in terms of interaction models, structural model and activity model. Interaction models describe the concrete realization of each operation specified in the functional model and use either UML collaboration diagrams or sequence diagrams to do so. A structural model describes the internal komponent architecture as a refinement of the structural specification model: new classes may be introduced and some specification classes may be detailed. The activity model gives a process-oriented view on each of the komponent's operations and may also be used to elaborate interaction models. Finally, the decision model at the realization level is an extension of the specification level.

The domain engineering activity (called framework engineering in KoBra) firstly establishes a "context realization" model of the framework. This model is composed of a class diagram showing the komponents involved in the framework, a list of processes supported by the framework and further detailed with activity diagrams, and a decision model that is relating questions to be asked to the product customer to variability proposed in the business processes. This model can be thought as a realization model of the "root" komponent of the framework used to initiate the KoBra process. From this realization model, the specification model of each of the subkomponents is defined, then refined into a new realization model which forms the basis for new subkomponents in a recursive manner. The recursion stops when there is no new subkomponent to define. Thus, the development of the framework can be compared to the development of a tree whose child komponents participate in the realization of their parents.
In addition to this top-down approach, a bottom-up approach, called "extreme harvesting" has been proposed [HA04]. It is based on the retrieval of components (available in a given source such as Internet) based on method signature for syntax and test cases for semantics. Extreme harvesting and KoBra integration are shown in [AH04].

The application engineering activity basically consists in assembling an application (or parts of it) from the framework. Application models follow the same language as the framework models but, in the former, variability is completely resolved. Application models may also contain specific features that are not provided by the framework but required for a particular product.

[8] In the remainder of the thesis, we will write komponents to refer to KoBra Component

Application engineering comprises two steps: *application context realization* and *framework instantiation*. The realization of the application context takes place between a customer who is interested in a type of application provided by a software vendor and a consultant of this vendor. During his interaction with the customer, the consultant progressively makes decisions that meet the customer requirements in the framework realization models according to the decision model. If a choice cannot be made with he help of the customer, the framework models are left partially instantiated. If none of the proposed alternatives for that decision point suit the customer, the specific requirements for that decision point are modeled explicitly. The alternative that is the closest to this requirement will be used as input for that modeling activity so that we only need to model differences between what is provided, what is required, and the corresponding realization for that alternative can act as a guide for the design activity. When all the decisions are made, the customer carefully checks the application context model and validates it.

The framework instantiation activity takes the application context model as input and subsequently performs all the decisions in the subkomponents that are connected to the context realization of the framework. Unresolved points are fed back to the consultant who will resolve them either with the customer or with the developers. In addition, customer-specific requirements will be implemented and integrated into the application model. They can also be integrated to the framework if it is believed that these requirements could be useful for several customers.

Thanks to its recursive komponent structure, Kobra provides an integrated metamodel and a well defined process covering SPL domain engineering. Concerning application engineering, no automatable support is defined, especially regarding features that are not provided by the framework and derivable with the help of the decision model. FIDJI contributes to this point by defining a flexible instantiation process (see Chapter 3). Furthermore, FIDJI offers more accuracy in the modeling process through the systematic use of OCL.

3. FIDJI CONCEPTS

Abstract

This chapter performs a critical analysis of the software engineering innovations presented in the previous chapter and introduces the solutions provided by the FIDJI method in a high-level manner. First, we forge our notion of architectural framework by combining models and existing OO frameworks. We then explain its instantiation mechanism by means of model transformations controlled via instantiation constraints. Section 3.1.1 presents the current issues in existing object-oriented framework development practices that have motivated the introduction of the architectural framework concept. Section 3.1.2 defines this concept and presents its constituents that will be detailed in Part II. Section 3.1.3 presents the approach for architectural framework instantiation. Second, Section 3.2 introduces the FIDJI methodology which activities are organized around the architectural framework instantiation. Finally, Section 3.3 describes the research method followed to develop this thesis.

3.1 Architectural Frameworks

3.1.1 Motivations

The Framework Reuse Problem

As we showed in Chapter 2, object-oriented frameworks are very popular in software engineering and are the key to successful code reuse. In particular, object-oriented framework-based development which proposes the systematic reuse of a code infrastructure is appropriate to product line development approaches where core assets are reused according to their commonalities and variabilities [BCS00]. Conversely, gains in application development time and quality generated by the reuse of the same infrastructure of assets make the cost of building an object-oriented framework acceptable. Hence, we believe that any SPL-based development approach should be based on the reuse of an object-oriented framework that is extended by application developers. However, non-trivial object-oriented frameworks contain a huge number of classes that are generally difficult to understand. Moreover, application developers have some difficulties to identify the appropriate classes to extend first, which may lead to mis(re)use of object-oriented framework assets thus violating framework design benefits (such as encapsulation or use of a particular design pattern [GHJV95]). Experimental evidence [KRW05] gathered from university student projects showed that framework reuse problems can be divided in four major categories (209 issues were analyzed but 46 were left out because of irrelevance or lack of information):

- **Mapping (38 issues)**: This category is about finding the appropriate implementation modules in the framework that come up with an abstract solution to the problem. Typical concerns include: “What should I use to represent...?” or “How do I express...?”;

- **Understanding functionality (60 issues)**: This category contains the highest number of issues. These issues are related to the specific understanding of the inner workings of the framework classes. Questions include: "How does...work?" or "Where is...defined/ created/ called?";
- **Understanding interactions (48 issues)**: This category relates to the communication between classes of the framework: "Where should I put...?". This category is important because a misunderstanding of interactions between classes can lead to issues somewhere else in the framework that are really difficult to trace and solve;
- **Understanding the framework architecture (17 issues)**: This category is about instantiating the framework in a particular application with disregard for its high-level architectural properties (which are enforced by design patterns for example). Although the authors claim that it may have no short-term effect, we believe that the essence of a framework being in the particular architecture it provides (which distinguishes it from a library), it is hence crucial that the application developer be aware of the impact an architectural change implies on his application.

This study is interesting because, as the authors point it, it shows what are the current challenges in framework-based development and that they are not all addressed by current techniques for framework documentation exposed in Chapter 2. In fact, each of these techniques is focused on one particular issue. For example, the use of patterns tackles the mapping problem and architecture-oriented programming [HHV+01] helps in understanding functionality by providing tasks to follow in order to specialize framework hot spots. However, the authors claim that understanding key interactions in the framework deserves further research. Indeed, if we look at the software engineering phases in which these categories of problems reside we observe that the following ones are involved:

- **Analysis**: Mapping problems relate to matching requirements analysis of an application to the solutions a framework provides;
- **Architecture**: It is necessary to have a global view of what framework components are doing and how they interact to understand its key interactions;
- **Design**: Details of a specific object-oriented framework functionality have to be known to use it satisfactorily.

Therefore, we are convinced that it is necessary to have a broad vision on framework reuse problems because they refer to different phases of software development and a framework documentation technique that will be able to solve all problems does not exists. As the authors of the above study mentioned, it is important to combine different techniques in order to solve these problems as a whole and not individually. This reasoning constitutes the starting point of our reflections on enhancing the notion of object-oriented framework to build the one of *architectural framework* by combining an object-oriented framework with models and transformations. In the following sections, we explain to what extent models and transformations can be of interest to address the aforementioned framework reuse issue before giving our definition of architectural framework.

Models for Framework Understanding

Within the categories identified by Kirk et al. in the preceding section, we can notice that three of them are related to framework understanding at various levels of detail. For example,

"mapping" and "understanding the framework architecture" are symptomatic of abstraction level mismatches: in both cases, developers try to find high-level information such as abstract description of the functionality offered by the framework (that has to be mapped with the general description of the solution they require for their application) or have a quick overview of the framework architecture in low-level artifacts such as code or accompanying documentation that typically do not provide this information. Hence, application developers are forced to infer this information from these low-level artifacts which is a time-consuming and error-prone task. In fact, this inferring activity may result in different conclusions about framework design than those originally drawn and implemented by framework developers. Several reverse-engineering techniques address understanding of an object-oriented framework by exploiting its interfaces [Vil03] or by recognizing the design patterns [KP03, GAK99] used. In fact, these techniques raise the abstraction level to ease understanding.
In Chapter 2, we promoted the use of models to describe systems at various abstraction levels. Therefore, quite naturally, we advocate that models should be employed at any abstraction level to ease understanding and reuse of object-oriented frameworks and improve the quality of their documentation. It should be noted that it is a general statement, in this thesis we will concentrate on requirements analysis, and architecture levels.

Transformations for Framework Instantiation

Understanding an object-oriented framework is a prerequisite to the development of an application based on it. The next step is to extend object-oriented framework assets and to complete them with application-specific code. This completion may involve a significant number of classes to create depending on the technology chosen (for example, J2EE EJB technology involves the creation of several interfaces for each EJB) and the design patterns used (which sometimes makes necessary to create additional classes to achieve reuse and flexibility). Automated techniques for framework instantiation provided by active cookbooks, tasks or metapattern transformations provide assistance only for the implementation phase of the application, not for its requirements analysis. We believe that model transformation is of paramount importance to automate part of the object-oriented framework instantiation.
In particular, we support the reuse of SPL assets at a particular abstraction level by means of horizontal transformations. In a SPL context, this implies using these transformations to implement variability mechanisms and select what particular assets of the product line will be present in the product i.e performing product derivation. In this thesis, we will particularly focus on them as a mean to support a flexible product line development process as initially presented in [GP06].

Object-Oriented Frameworks and Software Product Lines

In a part of Chapter 2, we explained the natural relationship between object-oriented frameworks and SPL concepts, both focusing on building and reusing common assets and providing variability mechanisms to support the development of different products. Although these concepts are similar, they do not address the same abstraction levels; object-oriented framework-based variability mechanisms (such as inheritance or composition) are focused on the actual implementation of variability while a SPL also requires to be identified and modeled at requirements and design levels. For example, feature models and use case variants are used for requirements elicitation while configuration scripts support product deployment. Various mechanisms have been proposed to cope with variability at all stages of SPL development [HP03, SvGB05] and

which have to be selected carefully carefully when performing domain engineering.

However, industrial research [BFG+01, DSB04] has shown that selecting variability mechanisms, although having a great impact in terms of flexibility and performance (at implementation stage), is frequently done arbitrarily. Furthermore, the same research states that the interaction between such mechanisms at various levels has not been sufficiently studied. Therefore, just like there is an abstraction mismatch between information present in an object-oriented framework and that required to readily understand it, there is also a mismatch between variability mechanisms hindering their smooth combination.
We believe that constraints enabling the definition of restrictions on product derivation amongst abstraction levels represent a way to integrate variability mechanisms in a coherent manner [GP06].

3.1.2 Definition

The preceding paragraphs motivated the need to compose models, transformations and object-oriented frameworks to facilitate SPL-based development. This need led us to define the term *architectural framework* as follows:

Definition 16 (Architectural Framework) *A layered set of reusable models characterizing core assets devoted to the specification and realization of a specific SPL. Layers include core assets definition, design and implementation as an object-oriented framework.*

The "architectural" qualifier refers to early work on the FIDJI method [GP02, GP04] (see Section 3.3 below).
We also assign a generic meaning to these "reusable models"; they normally include UML models, but also use case textual descriptions or code implementation etc.

Figure 3.1 illustrates a general overview of the layers composing an architectural framework. Each layer corresponds to a particular abstraction level and contains elements designed to be reused in an identified phase of product development. Each layer provides models and appropriate instantiation constraints to support the instantiation of the architectural framework models into product models. In addition, layers are connected by traceability links (expressed as dependencies in Figure 3.1).

The purpose of the analysis layer is to specify the functionalities offered as well as the concepts handled by the architectural framework. It refines SPL requirement elicitation documents and provides the necessary predefined transformations to build the products initially defined at the requirement elicitation level as well as constraints preventing invalid products from being developed. Chapter 4 describes in details the modeling language defined by FIDJI for this layer as well as transformation operations supporting product derivation available at this level.

The role of the design layer is to expose how the architectural framework achieves functionalities specified at the analysis level. This includes the explicit definition of the architectural framework structure as well as the algorithms supporting the behavior defined declaratively at the analysis level. Similarly to the analysis layer, transformation operations support product derivation and constraints ensuring that the intrinsic qualities of the architecture are kept. These elements will be detailed in Chapter 5.

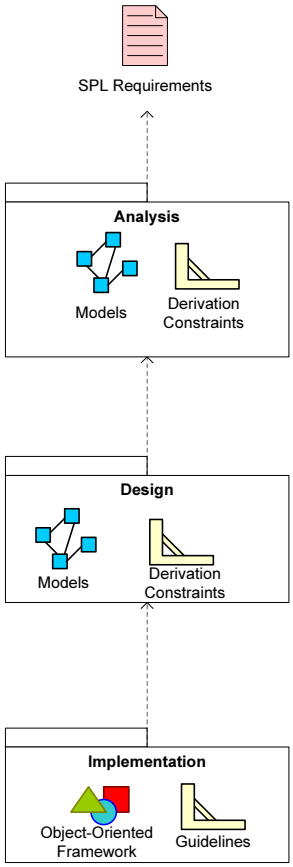

Fig. 3.1: Architectural Framework Layers

The implementation layer is the translation of the design layer in a particular programming language. It comprises an object-oriented framework along with the necessary documentation to complete product implementation.

3.1.3 Architectural Framework Instantiation

Overview

Architectural framework instantiation has the same goal as that of an object-oriented framework: to support the actual derivation of a product. The main difference is the instantiation scope: an object-oriented framework supports only product implementation whereas an architectural framework supports other artifacts (requirements analysis, design...) depending on the layers it provides.

Definition 17 (Architectural Framework Instantiation) *Process which consists in deriving product models by reusing architectural framework layers. Layer reuse is supported via*

model transformations and is controlled via constraints keeping some of the architectural framework's important qualities and ensuring adequacy of the derived product with respect to the SPL's definition.

If an architectural framework provides all the layers from requirements to implementation and deployment, then the process of architectural framework instantiation is synonymous with "application engineering" or "product derivation" concepts in SPL engineering (see Chapter 2, Section 2.4). Figure 3.2 depicts the relationships between architectural framework layers and application models:

- **Requirements Elicitation**: The requirement elicitation document of a product is a combination of both SPL features and informal description gathered from the product customer (FIDJI addresses SPL-based development in which products are requested and validated by customers. See Section 1.3 below);
- **Analysis**: Product analysis is obtained by reusing the architectural framework analysis layer by means of model transformation and validated against architectural framework analysis constraints;
- **Design**: The same applies for the design layer. Analysis models, by selecting or not particular architectural framework assets imply that their design should or should not be made available at the design level. Constraints provided at the design level take into account possible choices by means of conditional constraints,
- **Implementation**: Similarly, instantiation of an object-oriented framework depends on design choices. For example, in a task-based documented object-oriented framework, some of the tasks may be invalidated because they are corresponding to unused portions of the object-oriented framework for that particular application.

Instantiation by Transformation

As we already mentioned, we use model transformations to instantiate product models from architectural framework layers. Indeed, architectural framework instantiation is actually performed by the execution of an instantiation program.

Definition 18 (Instantiation Program) *An instantiation program is a combination of model transformation operations applying to a given architectural framework and supporting its instantiation.*

Our approach to model transformation is to define a library of transformation operations tailored to work closely with the architectural framework layers: for each layer, a dedicated package containing appropriate transformation operations to operate on layer models is provided, thus facilitating their use by the product developer. Each transformation operation is defined in a declarative way by means of OCL pre/postconditions. These operations are then combined imperatively in an *instantiation program* using basic programming language constructs. This instantiation program uses OCL capabilities to query the FIDJI models (which are mainly based on UML) and imperatively construct the target model using operations of the library.

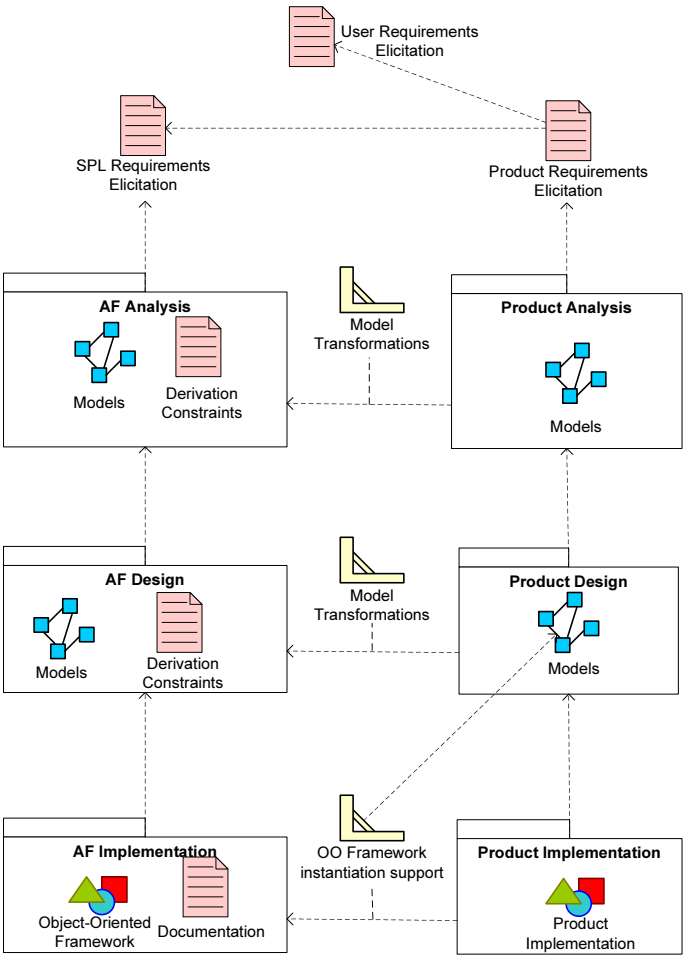

Fig. 3.2: Architectural Framework instantiation

In addition to simplifying the query mechanism on UML models, such an approach is also easily interoperable with other OCL-based tools which are likely to rapidly increase as the QVT standard is becoming more popular. The library of transformation operations will be illustrated for analysis and design phases of the FIDJI methodology in chapters 4 and 5. The concrete use of the language will be demonstrated in Chapter 6.

Instantiation Constraints

The purpose of constraints is to place restrictions on the products that can be defined via an instantiation program. They can be seen as a generalization to the analysis and design layers of the Structural Constraint Language proposed by Hou et al. [HHR04]. Constraints are expressed in OCL and state which parts of the architectural framework must be reused for all the products of a given SPL or which parts cannot be adapted by the instantiation program. Constraints are of two kinds:

- **Invariants on model elements**: This category of constraints specifies which model

elements must (or must not) be part of the instantiated product model;

- **Invariants on transformation operations**: Alternatively, it is also possible to express constraints using operations on a given set of elements. For example, one it is possible to prevent adding elements in a given asset. Although this category is semantically equivalent to the first since they can be rewritten as standard invariants using the postconditions specifying their behavior, it offers a more concise way to express architectural framework constraints.

We will exemplify the usage of constraints in our case study in Chapter 6.

Instantiation Validation

For each layer of the architectural framework, an instantiation program is developed. The first step is to validate program syntax according to the rules given by the instantiation language. The main step consists in generating the product model by executing the instantiation program. Then, we have to check the newly instantiated product model against instantiation constraints. If they are satisfied, an impact analysis step takes place in order to evaluate changes to be undertaken in elements depending on those concerned involved in the architectural framework instantiation. If the constraints are not satisfied, the newly product model is discarded and the instantiation program has to be rewritten.

3.2 Methodological Overview

3.2.1 Scope

FIDJI intends to be a general product line development methodology targeting the kind of application characterized in Chapter 1. However, there are three points that the method does not cover:

- **Scoping**: FIDJI does not provide any process for identifying the initial boundaries of the system and assumes that this preliminary activity has been completed using goal models [Lam01] and/or product line scoping techniques [DS99, CN01] in order to define this initial set in an abstract way and prioritize system goals;
- **User Interface**: Web applications have particular characteristics concerning their user interfaces which may greatly vary depending on the technology used to implement them and the hardware constraints affecting their display: nowadays, it is often required that web applications be displayed on various computing devices such as mobile phones and PDAs. FIDJI does not explicitly consider the specification and design of such user interfaces although our analysis model allows to capture information to be displayed by the user interface of a product;
- **Implementation**: We mentioned above that the implementation of an architectural framework is supported by an object-oriented framework. FIDJI does not define a specific approach for object-oriented framework instantiation but rather relies on existing techniques. We will give a few considerations about their suitability with respect to the FIDJI process in Chapter 8.

3.2.2 Driving Principles

Syntactic Reduction & Semantic Precision

In [AR03], Astesiano and Reggio report their experience with applying formal methods on industrial-sized projects. They argue that a development method should be distinguished from the formalism being usied and the relationships between methods, formalisms and their engineering contexts should be made explicit. In particular, they have come up with the idea of "well-founded software development method" consisting in analyzing current software engineering practices and their problems, providing some formal foundations if needed and improve these practices on this base while hiding formalization details to users. They have exemplified this approach on the RUP methodology, first remarking that the formalism used here (UML) was not rigorous enough, and that this methodology gives so many choices to their users that inexperienced ones may be confused about these options. Then, they have devised a "tight and precise" approach with the following goals in mind:

- To use only semantically sound constructs (i.e. that can be formalized easily);
- To have better means for making the modeling decisions;
- To restrict produced artifacts as to allow consistency management in the construction and checking phase.

While the FIDJI models presented in this thesis are not given a full semantics, FIDJI pursues the preceding goals by selecting relevant UML constructs with respect to the method's scope. These constructs are extended through a UML profile that also clarifies their use via OCL constraints. The question of formalizing FIDJI models will be discussed in Chapter 8.

Flexible Product Derivation

In Chapter 2, Section 2.4.3 we presented the current product derivation techniques which, from our point of view, lack flexiblity whatever their approach is (configuration or transformation). By "flexibility", we mean the possibility to address products that were not explicitly planned by product line designers. This situation is summarized in Figure 3.3: "Green" applications like A fall within the scope of the product line and are targeted by current techniques. "Orange" applications do not fall strictly within the product line scope; i.e they are not directly derivable from the decision model though sharing many features with green applications. For example, consider an order form in an e-commerce application; on the domain engineering side designers have specified that the price of an item would be represented by an integer number. An "orange" application may require that the price be entered in full for improved security. Finally "red" applications represent products that go far beyond the scope of the product line.

We claim that "orange" applications should be supported by a product line development approach because even if they do not observe the original product line definition and implementation, they are believed (by SPL designers) to respect SPL intent and can benefit from most of the existing assets offered by this SPL. However, current techniques show quite substantial approaches to support these applications:

- **Configuration approaches**: These approaches have no other choice than re-engineering part of the product line, i.e. introducing the new assets to support the variant and to modify the decision model accordingly. In ConIPF [HWK+06], a specific support via change operations is provided to update the decision model;
- **Transformation approaches**: None of the surveyed transformation approaches provides specific support on this point. In a similar way, the assets and/or the decision model would have to be modified.

In both cases, it is necessary to modify part of the product line to support product-specific assets. This is detrimental, especially due to the fact that the SPL has to evolve even though the changes required for a product does not correspond to a large number of customer requests. Over a long time, the SPL may contain a lot of useless features thus complicating its management. As we noted in Chapter 2, several SPL methods based on configuration (Pulse-I, Van Ommering's product population) cope with the problem of providing methodological support for product-specific features. However, this is a kind of textual methodological support and there is no technical means to assist the product developer in the derivation process.

This issue motivated our architectural framework instantiation mechanism; instead of explicitly modeling the variability in product line models, we allow the architectural framework to be instantiated via transformations defined by the product analyst/designer, who can accommodate application-specific requirements provided he respects the constraints — which play the role of a decision model — that ensure that the derivation is resulting in an "orange" application. In fact, these constraints indirectly specify the boundary (shown as a dashed oval in Figure 3.3) between applications that can be efficiently built with this this SPL and applications that would require either a modification of the product line or a specific single product development.

Traceability Management

FIDJI acknowledges the importance of tracing model artifacts during the software development process. It plays an important role both for architectural framework modeling and instantiation. We use the following techniques to ensure vertical and horizontal traceability:

- **Links**: The definition of vertical traceability amongst architectural framework layers is provided via links. To trace UML elements, we use constructs which are already available in the UML specification (see Chapter 5, Section 5.3). Concerning non-UML elements, we have defined a special textual template to document them (see Chapter 4, Section 4.3),
- **Instantiation programs**: As we have seen in Chapter 2, model transformation is one of the mechanisms allowing traceability construction. Therefore, horizontal traceability is a by-product of the application of the instantiation programs (as this thesis does not prescribe a particular transformation language/toolset at the technical level, we will not explain traceability bindings).

3.2.3 Process Overview

FIDJI bases its process on the instantiation of an architectural framework as presented in Chapter 3.1. It is a "stairs" (Figure 3.4) process which is indeed a "waterfall" [Roy70] development

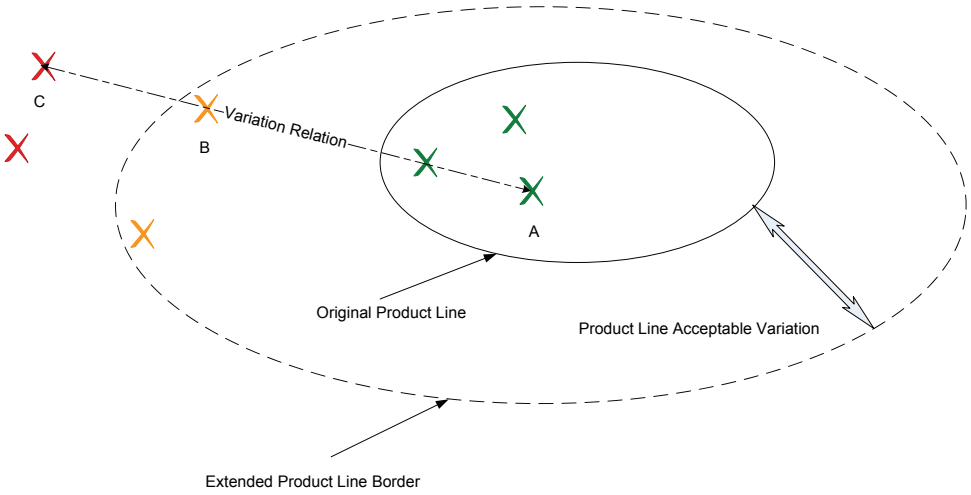

Fig. 3.3: Product Line Variation

model. Many criticisms have been formulated about this process model so we need to state our rationale clearly here. The waterfall process model is composed of phases that cover the whole development scope and are validated before the next phase can begin. If problems are found in the next phase with artifacts of the previous one, an iteration is made in order to go back to the problematic phase, to fix the problems, to revalidate it etc. Sommerville [Som04] states that the main disadvantage of the waterfall process model is the premature freezing of some parts of the product that occurs after a small number of iterations (due to the cost associated with producing and validating documents as well as reworking these iterations involve). This may result in a product that does not correspond to customer expectations since his voice is discarded early in the process. As we saw in Chapter 2, RUP designers insist on the risk of freezing a phase before proceeding to another; the later a fault is found, the higher the price is to fix it. Therefore, as noted by Sommerville [Som04], this kind of process should only be used when requirements of the software to be produced are well-understood as well as when the risks incurred as a result of designing and implementing such a system are well-known.

Surprisingly, the FIDJI method's requirements and scope correspond to the application context of the waterfall approach.
First, the waterfall process is the simplest of all the process models which satisfies our simplicity requirement. Furthermore, the waterfall process model allows model integration in a seamless way which is also one of the characteristics FIDJI has to support.
Second, in a SPL approach, most product requirements are well-understood since they have been carefully defined and analyzed according to a particular market and to be valuable enough to develop reusable assets supporting them. Moreover, design and implementation risks are largely reduced by the (re)use of an architectural framework since it provides a core architecture and its

implementation (in the form of an object-oriented framework) on which all products are based. The flexibility concern may seem more problematic to handle with such a process since it induces some risks concerning customer-specific requirements. However, these risks can be reduced by defining instantiation constraints that would prevent high-risk products (i.e. products that need substantial transformations of the key architectural framework assets that would threaten architectural framework qualities) to be obtained from the architectural framework. Additionally, customer-specific requirements should represent a small part of the set of requirements pertaining to the product. In a different situation, the product to be developed would not match the architectural framework's scope and consequently, a strategic decision would have to be made (update of the architectural framework, use of another architectural framework or consider specific implementation, etc.). However, some reasons may motivate evolutions of the method process. They will be discussed in Chapter 8.

Figure 3.4 presents the FIDJI process as a SPEM activity diagram. The process is composed of the following activities: *Define Product, Write the analysis derivation program, Write the design derivation program* and *Build product*:

- *Define Product*: In this activity, requirements elicitation of the product is performed with respect to the product line definition expressed following a template called REQET [GP06, GGMP06]. It consists in resolving the variability offered by the SPL by selecting the required scenarios and concepts for the product and complementing them if necessary. We describe this activity as well as the template used in Chapter 4, Section 4.1;

- *Instantiate the architectural framework Analysis Layer*: This activity consists in performing the analysis of the product, by reusing the analysis models of the architectural framework that will be detailed in Chapter 4. This includes the identification of concerned architectural framework analysis elements, writing and validation of the instantiation program and impact evaluation;

- *Instantiate the architectural framework Design Layer*: In a similar manner, product design relies on the same steps applied to design elements of the architectural framework. The architectural framework design model as well as the steps required to perform instantiation of the architectural framework will be provided in Chapter 5;

- *Build Product*: Once the design has been performed, the product needs to be built reusing the object-oriented framework supporting the implementation layer of the architectural framework. The FIDJI process provides no specific methodological support for this activity, instead it builds on existing object-oriented framework documentation techniques presented in Chapter 2, Section 2.5.3.

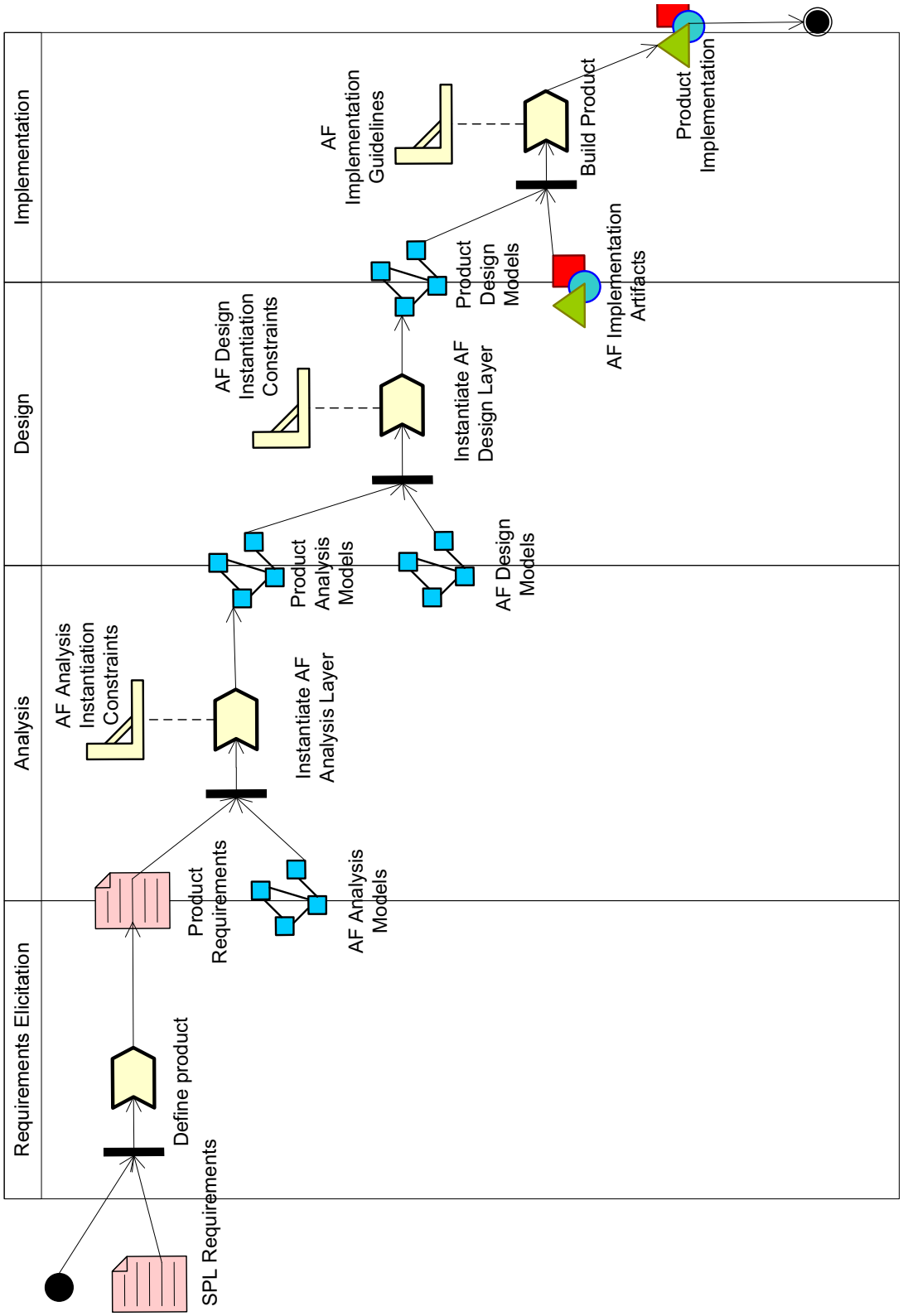

Fig. 3.4: FIDJI "Stairs" Process

3.3 Research Method

The ideas presented in this chapter result from the experience acquired during the FIDJI project and literature review performed in chapter 2. In this section, we explain how these ideas are born by recalling and analyzing the contributions of the FIDJI project. These contributions pertain to three research domains: framework-based development, model transformation and methodology.

3.3.1 Framework-based Development

Our early notion of architectural framework erected during the FIDJI project [GP02]. We wanted to emphasize on documenting object-oriented framework architectures (hence the "architectural" qualifier) in order to understand the framework and to instantiate applications from it. This notion was concretely implemented in the J2EE Architectural FrAmewoRk (JAFAR) [GS02b, GR02, GRS03a] which was dedicated to the development of e-barter applications. This experience has allowed us to validate the suitability of using such an entity in the development process of SPL members (see Chapter 6, Section 6.6).
Due to its design and implementation focus, JAFAR did not include detailed models for requirements description (this was limited to UML use cases, which could be omitted when performing application development). However, as mentioned previously by Kirk et al. [KRW05], it is necessary to understand the framework at various abstraction levels. Based on that remark and on the lack of requirements models in JAFAR, we decided to extend the research scope to requirements engineering.

3.3.2 Model Transformation

In order to support application development with JAFAR, we used model transformation to generate detailed design and implementation from high-level design models. We defined a visual model transformation language [GPR+03] and implemented it as a tool called MEDAL [GRS03b] on top of the Rational XDE platform, thus improving its already existing transformation mechanism. An individual transformation was modeled as a UML class diagram connecting the XDE model elements actually performing the transformation and its parameters calculated by evaluating OCL queries. Transformations were combined using activity diagrams. The language was supported by a UML profile. Thanks to the OCL facilities of Rational XDE, we were able to define constraints to check conformance of transformation definitions with respect to the UML profile.

Research work on model transformation then focused on improving the transformation language that was heavily dependent on the XDE transformation mechanism. Our new proposition was called Visual Model Transformation (VMT) [SPGB03] and was based on a graph transformation approach thus allowing to separate the UML elements to match (specified at the M2 level and completed with OCL constraints) and the definition of the transformation. The right hand side could be completed with Java statements in order to ease transformation definition.

However, there is one reason that led us to reconsider our transformation approach in the context of this thesis. As a pragmatic approach, FIDJI tries to rely upon standards to promote

and support its adoption in the industrial context. The QVT [OMG05b] specification, which is now the OMG standard for model transformation, is essentially based on a textual and OCL-based language. Several other model transformation languages such as ATL [JK05] and Kermeta [FDVF06] follow a similar approach.

3.3.3 Method

As we have mentioned in the introduction of this dissertation, the initial FIDJI process originated from formal refinement approaches and from the Fusion method [CAB+94]. This motivated our initial framework instantiation approach in which it was only possible to add information (either by specializing classes or creating new elements in predefined placeholders). As this approach did not allow any other variation than those explicitly defined and supported in JAFAR, it suffered from the same inflexibility as the approaches discussed in Section 3.2.

Our approach to supporting flexible product derivation via horizontal model transformation has its roots in *architectural reconciliation* [AGP05]. This approach was based on the fact that when a software application evolves, it is usually only its code that changes and not its higher-level models (requirements and architecture). This situation has generated the "architectural erosion" [MEG03] issue; models do not reflect the actual software implementation anymore and the implementation does not enforce architectural qualities anymore. Architectural reconciliation first consists in recovering architectural models from the actual code (using reverse engineering techniques). Then the recovered model is compared with respect to the "ideal" architectural model derived from requirements. On the basis of that comparison, the software architect makes some decisions (he "reconciles" the two models) which result in a third model. Reconciliation decisions were recorded in terms of OCL constraints defined over UML 2.0 architectural elements. These constraints can then be used to derive horizontal model transformations generating the reconciled model.
From a methodological perspective, product derivation at a given abstraction level can be seen as a reconciliation between an "ideal" model stemming from customer requirements and the existing architectural framework model (which is not recovered in this case). The reconciliation takes place in writing an instantiation program that takes into account the "ideal" model and instantiation constraints in order to obtain the model of the actual SPL member.

As we have mentioned in Chapter 2, object-oriented frameworks impose the operation life-cycle to the applications that are based on them ("Hollywood" principle). JAFAR was no exception to this principle. Thus, the operation life-cycle of a particular SPL member was primarily determined by the architectural framework and the various approaches discussed to describe operation life-cycle could have been used in the FIDJI project (this issue has not been explored, though). Allowing flexible product derivation therefore implies that operation life-cycle can be modified in unpredictable ways in order to support one particular product. This situations makes the existing approaches for life-cycle descriptions (regular expressions proposed by Fusion in particular) intractable. This issue has motivated the elaboration of state variables.

Part II

FIDJI: A METHODOLOGY FOR DISTRIBUTED SYSTEMS

4. REQUIREMENTS ELICITATION AND ANALYSIS

Abstract

In this Chapter, we define the requirements elicitation and analysis phases of the FIDJI method. As in any SPL-based development method, a distinction can be made between domain engineering and application engineering. Sections 4.1 and 4.2 present domain engineering models supporting the description of a SPL and the architectural framework analysis layer, respectively. Section 4.4 is devoted to the definition of FIDJI application engineering steps yielding the final analysis model of a SPL member.

4.1 FIDJI Prescriptions for SPL Requirements Elicitation

4.1.1 REQET Overview

In this Section, we describe a template, initially defined in [GP06, GGMP06] and called REQuirements Eliciation Template (REQET), to elicit informally SPL requirements. The objective of this template is to share knowledge about software product lines and prepare a detailed analysis with the FIDJI method. This template puts the emphasis on documenting variants in order to have a clear idea of the structural and behavioral variations amongst the different products. This information will then be used to define the analysis constraints guiding the derivation of a product as we mentioned in Chapters 3.1 and 3.2. The REQET template provides two sections: DOMain Elicitation Table (DOMET) used to define concepts of the domain and Use Case Elicitation Template (UCET) describing product behavior.

4.1.2 DOMET

The role of the DOMET is to provide the necessary information to understand all the possible variants concerning data (domain concepts) amongst SPL members. It takes the form of a "data dictionary" *à la* Fusion depicted using a tabular notation as shown on Table 4.1 below. An example is given chapter 6, Table 6.1.

4.1.3 UCET

As we mentioned in Chapter 2, use cases are an interesting approach to elicit software product lines requirements. The use case elicitation template builds on the popular template given by Cockburn [Coc01, Coc] and extends it with the "reuse category" approach [Gom04] applied to

Concept Name	**Var Type**	**Description**	**Dependencies**
Give the name of a Concept	Give the Variation Type. It represents the level of variation the concept can have: • *Mand*: represents a concept that is mandatory in all members of the SPL, • *Alt*: represents one of the alternative concepts that has to be chosen in a given product. • *Opt*: represents an optional concept that may be omitted.	Give a textual description of the concept	Describe all the dependencies that can show up within this concept.

Tab. 4.1: DOMET Contents

use cases and the use case variants notations proposed by Fantechi *et al* [FGJ+03, FGLN04]. In particular, two levels of variability for use cases are proposed:

- The availability of the use case itself may vary depending on the product. The "variation type" field, which is equivalent to the variation type for uses cases mentioned above, states whether a use case is mandatory *Mand*, optional *Opt* or alternative *Alt*. In the alternative case, other possibilities to choose from are given as use case identifiers within braces,
- Within a particular use case, some steps of the scenarios may vary from one product to another. To describe these variants, we employ Fantechi et al.'s notation and fully describe them (their types and if they concern data or behavior) in a dedicated field of the UCET.

All the UCET fields are defined in Table 4.2. Examples of UCET descriptions are given in Chapter 6.

4.1.4 REQET Usage and Validation

In the following, we give some advice on how to efficiently fill the template and some informal rules to check consistency of REQET-based descriptions.

When willing to use REQET for a SPL, one question raises up spontaneously; shall we describe use cases first and then fill the domain table or the opposite? Traditionally SPL description begins with a "domain engineering" phase that identifies SPL concepts and then requirements are formed on the basis of those concepts. In REQET the situation is a bit different since our domain table includes the type of variation for each concept that can be devised from use cases variants (see next paragraph). Moreover, it is likely that use case writing yields new SPL concepts that were not identified beforehand (unless domain engineering forces to use only the concepts it identified: it is a bit too inflexible to follow such an approach at the requirements

ID	An identification tag of the form UCXX (where X is a digit), useful for referencing UCs within variants.
Use case name	Each use case is given a name. The name should be the goal as a short active verb phrase
Var. Type	Specify whether the use case is mandatory (Mand), optional (Opt), or alternative (Alt) add the alternatives here.
Description	Describe the use case, one or two sentences (i.e. a longer statement of the use case goal).
Actors	Name all the actors that participate in the use case. Start with the primary actor and continue with secondary actors. • Primary actor: name (It is the actor that initiates the use case) • Secondary actor: name (is the actor that may participate in the use case)
Dependency	Describe whether the use case depends on other use cases; e.g. whether it includes another use case.
Preconditions	Specify one or more conditions that must be true at the start of the use case.
Postconditions	Identify the condition that is always true at the end of the use case if the main sequence has been followed.
Main scenario	Textual description taking the form of the input from the actor, followed by the response of the system. The system is treated as a black box, that is, dealing with what the system does in response to the actors inputs, not the internals or how it does it. The main scenario defines a partial order over the set of operations of the possible products.
Alternatives of the main scenario	This section provides the description of the alternative branches of the main sequence. Each branch must state if the main goal is achieved (that is the postcondition satisfied)
Non-Functional	Specify non-functional properties (like security, efficiency, reliability, scalability, etc.) related to the use case.
Variation points description	Describe here the introduced variation points and their dependencies in the use case V1,..., Vn. Variants have a type (Mand,Alt,Opt) and a concern: data or behavior. Moreover when using a variant parentheses may be use to indicate a default value. For optional variants, a special value, `null`, is defined when the option has not been selected.

Tab. 4.2: UCET Contents

elicitation level). Hence we suggest a path where the two activities (filling the domain table and writing use cases) overlap greatly:

1. List preliminary SPL domain concepts and define them in the table,
2. Write use cases on the basis of the table and update them if needed,
3. Adjust concept variation types according to the consistency rules given in the next paragraphs.

Although it is possible to omit some common functionality in the SPL (use cases are not exhaustive about the behavior of a system) we impose that all identified variable concepts and behaviors are described. Following other SPL development phases (analysis, design) will help to discover details of the products that already belong to the SPL and to handle variants but not to find new variants that may be omitted at the requirements elicitation level. Furthermore it is important when working with customers of the SPL members that we figure out useful variants using a particular scoping technique. Naturally, products will not be limited to this set of identified variation points, new variation points can be obtained via our product instantiation mechanism.

While writing a DOMET description, we should be particularly careful about the following points:

- Name Conflict: one should ensure that two different concepts do not have the same name as it may yield mismatches in the products. This error is easy to avoid in short DOMET descriptions but may be more subtle in long ones,
- Dependencies Problems: When we rate concepts via dependencies in the fourth column of the DOMET, we need to think about the impact they have on the SPL as a whole. For example, assume that a concept A requires a concept B and cannot be used together with concept C. If B requires C, all products needing to have the concept A can not satisfy the dependencies defined in the DOMET. This kind of problems may be difficult to find and solve if there are many inter-related concepts.

We also have to consider the relationship that exists between a use case variant (concerning concepts or behavior) and the variation type of the owning use case. For instance, a concept that is used obligatorily in an optional use case will not become mandatory for the whole SPL. Table 4.3 determines how variable elements should be handled with respect to the reuse category for the use case.

Variant type \ **UC Variation Type**	Mand	Alt	Opt
Mand	Mand	Alt	Opt
Alt	Alt	Alt	Opt
Opt	Opt	Opt	Opt

Tab. 4.3: Relationships between Use Case Variation Type and Variant Type

Finally, one should also ensure consistency between concepts in the domain table and their usage in the use cases variants. For instance the same concept can be used in a use case with an optional meaning and in another with an alternative one. This raises an issue in the DOMET because there is only one variation type allowed in this table. In order to build such a table we should take into account the "strongest usage" of the concept. Strongest usage means for us having the following (natural) order: Mand > Alt > Opt, so that a concept that is obligatorily used in a use case and optionally in another will get a mandatory one in the DOMET. Thus, the DOMET provides useful information for core assets designers who will quickly identify the concepts they have to provide in their design. Concerning alternative concepts used in several use cases, one has to mention the various alternatives prefixed by the use case id for a better traceability in the dependencies column.

4.2 FIDJI Analysis Model

The FIDJI analysis model is used both to fully describe the analysis layer of an architectural framework and to describe the analysis of a product that has been instantiated from this architectural framework. This section is devoted to the domain engineering usage of that model. The FIDJI analysis model is composed of the following sub-models:

- Domain Model: defines precisely concepts manipulated by use cases and operations,
- Use Case Model: extends and refines the use cases described at requirements elicitation time. Its purposes is to express the architectural framework's behavior in terms of sequence of operations,
- Operation Model: specifies in detail each operation: informal description, parameters, return values and pre/postconditions.

4.2.1 Domain Model

FIDJI domain model extends the DOMET by providing more detailed descriptions of the concepts manipulated at the analysis level. In particular it considers two kinds of entities:

- Concept: At the FIDJI analysis level, a "concept" represents a fundamental piece of data in the system, e.g. a user account storing name, surname and password to log into an e-commerce system.
- Signal: Here a signal is used to define information exchanged while the system interacts with its actors. It can represents the return of a value to the user after the execution of an operation or it can be used to "emulate" user interfaces (FIDJI does not provide any specific model to describe user interfaces, interested reader is referred to [MRP95]) by simply defining the content (fields, forms etc.) a screen may show as a response to an actor's action.

We use a class diagram notation to represent concepts and signals. Concepts are mapped to classes and may show attributes but excludes any possibility to attach behavioral information such as operations (which are detailed in Section 4.2.3). We believe that separating data and behavior at the analysis level helps the analyst focusing on the "what" of these elements rather

than delving too quickly in preliminary design. Some analysis concepts are representation of an external actor in the system; for instance the `UserAccount` concept is the representation in the system of a human actor. In order to model such a relationship (that will be useful to precisely model interaction between the considered the system and its actors), we make use of a technique, initially proposed by Sendall and Strohmeier [SS01], which consists in defining a special association stereotyped as <<`id`>> connecting the concept with the actor it maps to. Signals follow the UML 2.0 notation for the `Signal` metaclass (see section 13 "Common Behavior" of the specification [OMG05f]); they are represented as classes stereotyped with <<`Signal`>>. Figure 4.1 presents the domain diagram of the ***LuxDeal*** case study detailed in Chapter 6.

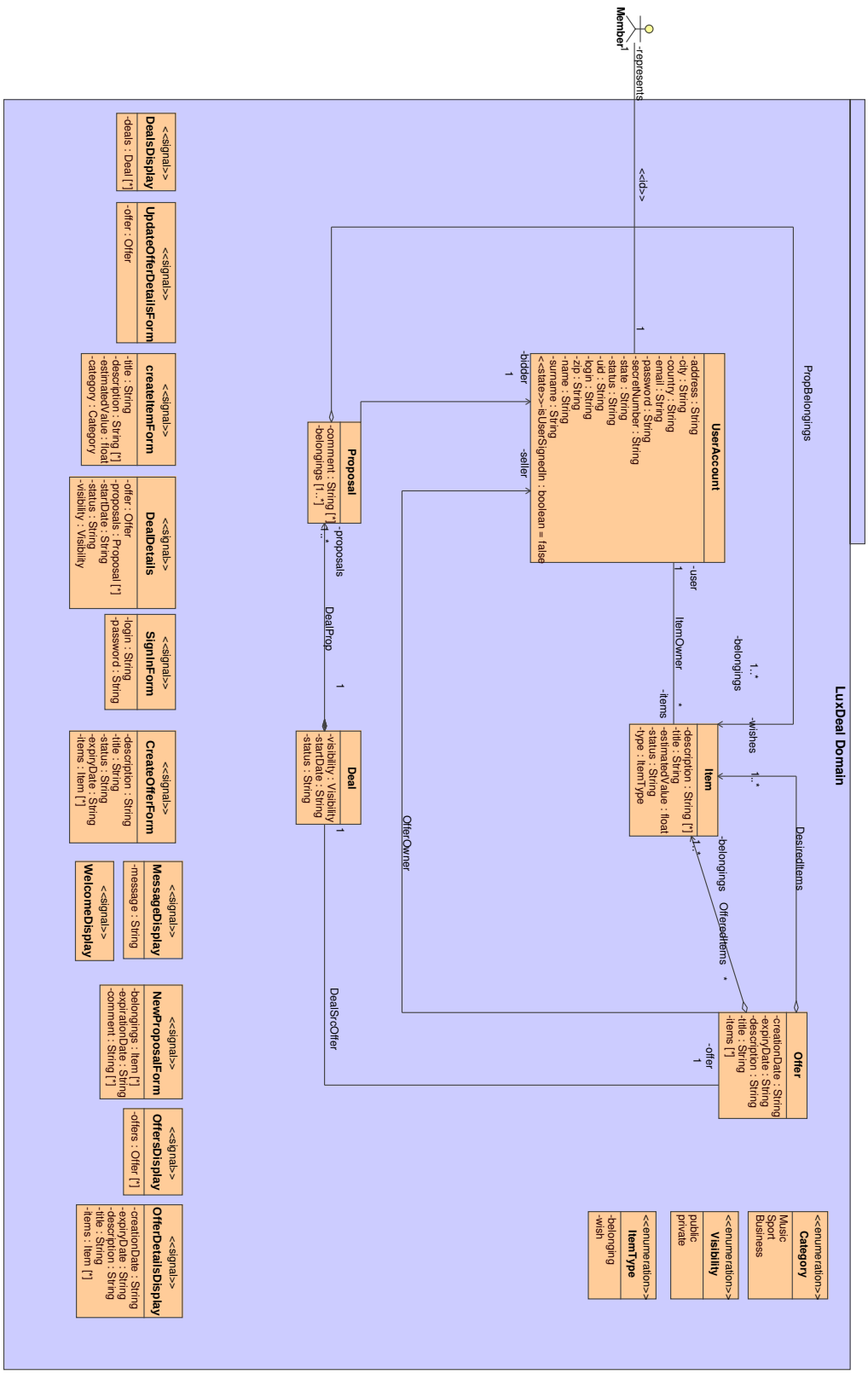

Fig. 4.1: Domain Diagram

The domain diagram gives a "flat" description of the data manipulated by the operations of the analysis model. However it does not give the rationale of these concepts and signals. We propose to define a *domain dictionary* which is inspired from Fusion data dictionary (while Fusion uses a single dictionary and extends its columns upon need, we prefer to use several dedicated dictionaries for concision) to capture data rationale. The purpose of the domain dictionary is to explain informally the meaning of the concepts and data elements which are used in the analysis specification of the system. We use a tabular notation as shown in Table 4.4.

Name	**Kind**	**Use Cases**	**Description**
Concept or Signal name	[Concept/Signal]; states whether the element is used to refer a concept of the application such as a user having name, surname and address or whether it is used for the specification of a communication between an actor and the system	list of the use case(s) manipulating the signal (not available for Concepts)	Description of the element in natural language

Tab. 4.4: Domain Dictionary

4.2.2 Use Case Model

Motivations

The role of the use case model is to refine the UCET-based description by precisely modeling interactions between actors and the system. Although giving a partial view on the allowable scenarios of the system (all allowable interactions can be be derived from the operation model introduced in Section 4.2.3) it helps understanding the important scenarios offered by the product line. From a methodological perspective there are also useful to facilitate the discovery and elaboration of the system operations. Our use case model is based on textual use cases extended with UML and OCL to achieve the following goals:

- To state the conditions that must hold before the execution of the use case scenario an the conditions that must hold after its execution,
- To describe the behavior of each step of the scenario thus complementing precisely textual behavioral description. We model these steps by means of OCL postconditions and by using the new OCL 2.0 [OMG05e] `message` operator,
- To define *state variables* that can be used to define guard conditions over the course of events without requiring to explicitly model it,
- Finally, we gather in a *use case component diagram* the definition of the signature of all the operations supporting the use case behavior as well as the depiction of UML elements on which OCL expressions are defined.

Template

We will base our use case descriptions on the following template which is a subset of the one given by Cockburn [Coc01, Coc]; in our context, we will focus on system functionality first and

leaving details about project management issues and use cases relationships (that can be shown more easily in a graphical manner as we will illustrate it on the use case diagram later in this section).

Use Case: UCNN (NN: two digits) : `<The name should be the goal as a short active verb phrase>`

CHARACTERISTIC INFORMATION
Goal in Context: `<longer statement of the goal if needed>`
Scope: `<System considered as black box under design>`
Preconditions: `<(Partial) State of the system required before goal execution>`
`<text and OCL 2.0 precondition>`
Success End Condition: `<(Partial) State of the system if goal successful>`
`<text and OCL 2.0 postcondition>`
Failed End Condition: `<(Partial) State of the system if goal abandoned>`
`<text and OCL 2.0 postcondition>`
Primary Actor: `<primary actor name>`
Secondary Actors: `<secondary actor names or descriptions>`
Trigger: `<event enabling the use case;`
`<text and OCL 2.0 postcondition>`

MAIN SUCCESS SCENARIO
`<Describe here the main steps, using text and OCL 2.0 pre/postconditions>`
`<text have the following form: <step#>.<action description>`

ALTERNATIVES
```
<Describe here alternative steps (prefixed by the altered step number followed by
 the action description) of the main scenario, using text
and OCL 2.0 pre/postconditions>
```

OCL Expressions

We use OCL 2.0 expressions in order to detail more precisely use case pre/postconditions and scenarios. Both OCL constraints and pre/postcondition are used. Warmer and Kleppe [WK03] documented the usage of OCL 2.0 in use cases in showing how pre/postconditions can be employed to precise the global pre/postconditions of a use case. However, at the analysis level it is important to detail the interactions between the system considered and its actors. As we do not consider any specific interaction language to describe such interactions, we need another way to specify them. The answer is to be found in the OCL 2.0 specification itself; as we mentioned in Chapter 2, the `isSent` and `message` operators are used to specify either the call to an operation of the system or the sending of a signal. Consequently we use this construct at each step of the main scenario as well as for the alternative scenarios and the triggering condition. These expressions provide a precise description of the usage of the system operations that are declaratively specified in the operation model (see 4.2.3).

All the interactions (modeled via `isSent` and `message` operators in OCL) defined in the context of a use case are asynchronous; both system and actors do not wait for the other part to respond.

In fact the interaction sequences are not based on time but on conditions: the next step can occur if and only if its precondition is satisfied (see "Instantaneous and Asynchronous Execution of Operations" below).

State Variables

OCL Expressions presented in the previous paragraph are useful to define the interactions between actors(s) of use case and the considered system. We often need having to refer to a previous state of the system to define the current behavior to be specified for a step, e.g. checking if an user has successfully logged in the system prior to show him his account details. As shown in Chapter 2, OCL 2.0 does not support the expression of fine-grained temporal constraints and researchers extended the language to support temporal logic constructs. However, these approaches requires change in the OCL specification both at the syntactic and semantic level. In order to cope with this issue we introduce *state variables* as a lightweight means to store information that could affect the event sequence (branching between the main scenario and alternative(s)) or the precondition of an individual scenario step. From a methodological perspective, defining state variables locally for a scenario is easier than defining a state machine for the whole use case. State machines can be constructed by using the information provided in the operation model (see Section 4.2.3).

Use case may need to share state variables in order to specify dependencies between their sequences or to access to a state information that is needed by several use cases (e.g. if a user is logged in). Therefore, we both need to define state variables globally or locally. Global state variables are defined as concept attributes in the domain diagram. In order to distinguish them from the other attributes of the concept there are stereotypes as <<`state`>> within the domain diagram. Local state variables are gathered in a dedicated classifier within the specification of a use case component (see below). For example, state variable `isUserAuthenticated`, which evaluates to true if the user has successfully entered his credentials in the system, is likely to be defined as an attribute of the `UserAccount` concept while `loginAttempts` which is a variable restricting the number of times a user can fail to enter his credentials before its account get deactivated, will probably only be defined in the context of the `Sign In` use case.

The value of a state variable may be changed via a system operation (see Section 4.2.3) which also used them to express guards on its preconditions and thus restrict the possible sequences of interactions within the system. Hence, state variables represent a concise means for the analyst to specify a wide variety of scenarios (all those that are tolerated by operation preconditions) without having to explicit them.

It has to be noted that though state variables may be reused at the design and implementation level to actually control methods (operations implementations), they do not exclude another approach (such as the raising of exceptions in a given programming language) to be used.
We will illustrate the possible usages of state variable in the FIDJI analysis model in Chapter 6.

Use Case Components

We introduced above our extension of textual use cases with OCL 2.0 expressions. However, as stated in chapter 2, OCL expressions need to have a context to be meaningful. Textual use

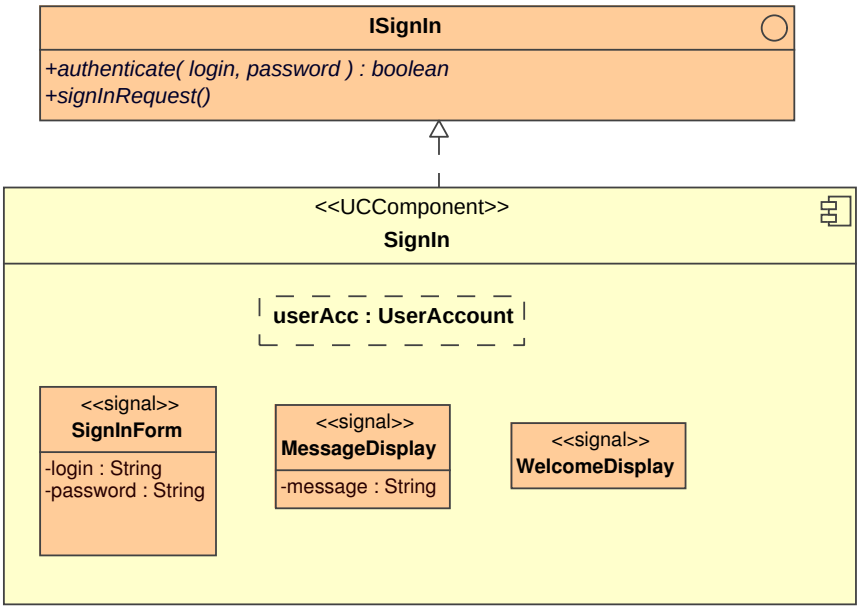

Fig. 4.2: Use Case Component Sign In

cases are not part of the UML 2.0 specification and therefore are not directly related to any UML element that could provide a suitable context for the OCL expressions. This problem has been outlined by Warmer et al. [WK03]: the context of OCL expressions is not clear since it is difficult to consider the whole system as a type on which we can base expressions. Moreover in the case of a large system including several nested packages this would induce quite huge OCL expressions that would browse the complete system, which is also discouraged by the authors. To overcome this problem, Warmer and Kleppe [WK03] suggest to define the UML types used in expressions at the use case level and provide a class diagram fully describing these elements. We extend this approach by associating an UML 2.0 component, called *use case component*, to every use case defined in the use case model:

Definition 19 (Use Case Component) *A use case component is a UML 2.0 component that contains all the elements (concepts, signals and operation signatures) necessary for the complete specification of OCL expressions defined within a use case. It also provides their evaluation context.*

As we mentioned in Chapter 2, UML 2.0 component notation is principally used to model architecture and design artifacts. However, similarly to domain concepts which are not design classes, use case components are abstract; they are not executable as is and are not prescribing any design choice. In order to distinguish them from more common usages of the notation, use case components are all stereotyped with <<`UCComponent`>>. UML 2.0 components have a packaging ability [OMG05f]; this allows, from a practical perspective, to embed the textual use case description within its associated use case component and ease its management in a UML case tool. Figure 4.2 depicts a use case component associated to a use case performing users sign in on an e-commerce website.

Parts

Parts in use case components represent the following information:

- Analysis concepts handled in the use case; e.g. `userAcc` in Figure 4.2,
- Data exchange between actors and subject of a use case. Data asynchronously output by the system (see also Section 4.2.3) is modeled as UML 2.0 `Signal` metaclass instances, e.g. `MessageDisplay` and `WelcomeDisplay` in Figure 4.2,
- State variables defined for this use case (here `loginAttempts`).

As a graphical convention, we use the standard notation for parts (i.e. a solid rectangle with the name of the property and its type) that are shared amongst several use cases and we exposes the full definition using the class notation for parts that are defined specifically in the context of this use case. Only domain concepts can be shared (e.g. `UserAccount`) whereas signals are defined only for a single use case. The complete specification of all concepts and signals is to be found in the domain model (Section 4.2.1).

State variables defined in the context of a specific use case are gathered in a special class stereotyped as <<`UCControl`>>. They are accessible to OCL expressions through this class. As we already mentioned use case components provide the context for OCL expression to evaluate; by default, the context of OCL expressions defined in the textual use cases is the one of the use ace component that is associated to this particular use case.

Interfaces

Use case components provides interfaces. These interfaces give the signatures of all the system operations that are employed by the use case to provide its functionality. Operations signatures shown on the use case component diagram must be compatible with their full specification in the operation model (see Section 4.2.3).

Use Case Diagram

As pointed out by Cockburn and Larman [Lar02], writing use cases means writing text, which is the heart of use-case descriptions. However relating dependencies between use cases and actors in a textual form as proposed in existing use case templates does not give a concise view on them. As we presented it in Chapter 2, UML 2.0 offers a diagrammatic notation to give a high-level view on use cases and their relationships to actors and other use-cases. Recall that according to the UML specification, the following metaclasses expressing relationships between use cases are available: `extends`, `includes`, `generalize`.
The UML specification states the rationale of `extends` relationship as: "This relationship is intended to be used when there is some additional behavior that should be added, possibly conditionally, to the behavior defined in another use case (which is meaningful independently of the extending use case)." [OMG05f]. In our context, there are two usages in which `Extends` could be used:

- Documenting alternative Scenarios: the "additional behavior" can be considered as an alternative scenario possibly triggered by the value of a condition. Furthermore it is added that "the extending use case typically defines behavior that may not necessarily

be meaningful by itself." Hence it could be as well included in the extended use case. Our textual template already provides means to express alternative scenarios as well as triggering conditions by means of OCL expressions,

- Handling SPL variants: the other possible usage is to express SPL variants as proposed in [Gom04, HP03]. As we adopt a generative approach, the handling of such variants is realized through sub-transformation programs and controlled by constraints.

The `generalize` relationship is not specific to use case and defines inheritance between all UML classifiers. Between use cases in our context the semantics of this relationship is not clear; is it an overriding of the system operations sketched in the use case component diagrams and fully defined in the operation model (Section 4.2.3) ? Or as proposed in [Iso04], is it part of the behavior of the generalized use case that is simulated by the generalizing use case ? Consequently in alignment with our approach to use only semantically clear subset of the UML metamodel, we decided not to allow generalization relationships between use cases. The `includes` relationship has a more clear semantics; it imposes that the included use case behavior is inserted in the behavior of the including use case. This is useful to decompose some behavior in smaller pieces thus minimizing the size of some use case descriptions. In this case, it is mandatory that a reference to the included use case component is present in the including use case component; this allows OCL expressions of the included uses cases to benefit from the context of the including one and in the including one to refer the system operations offered by the included use case.

Actor Multiplicity & Identification

As we have seen, actor multiplicities can be specified in the use-case diagram. The use case diagram shows the actor(s) who interact with this specific use case. This actor may send or receive events as specified in the OCL statements accompanying use case steps. As a shortcut we will use the names of the actors directly because it is more significant in an use case description. However if one needs to checks this OCL constraints formally, the name of an actor must be replaced by `attribute.represents` (see Section 4.2.1) where `attribute` is an attribute of the use case component of the type of the class that pertain to an <<`id`>>-stereotyped association with the concerned actor type in the domain model.

The following use case is excerpt of the case use case model of our case study detailed in Chapter 6. It exemplifies the various model elements we defined above for use-case modeling:

Use Case: `UC01: Sign In`

CHARACTERISTIC INFORMATION
Goal in Context: A member signs in the application in order to access to the application main's services.
Scope: `LuxDeal`
Preconditions: Member is not signed in:
pre: `userAcc.isUserSignedIn=false`
Success End Condition: Member is signed in:+
post: `userAcc.isUserSignedIn=true`
Failed End Condition: Member is not signed in:
post: `userAcc.isUserSignedIn=false`
Primary Actor: `Member`
Secondary Actors: None.+

Trigger: User requests to sign in;
post: `self^signInRequest()`

MAIN SUCCESS SCENARIO

1. Member is presented a sign in form:
   post: `sender^SignInForm(login,password)`
2. Member enters his credentials in the form,
3. Member authenticates himself to the system;
   post: `self^authenticate(login,password)`
4. If member's credentials are correct, Member is displayed a welcome screen:
   post: `Let m:OCLMessage = SignIn^^authenticate(login,password)->first()` **in**
   `m.hasReturned() and (m.result()=true) implies`
   `sender^WelcomeDisplay and userAcc.isUserSignedIn=true`

ALTERNATIVES
4a. if user credentials are incorrect and the session have not expired the user is invited to retry:

- Member is informed that his credentials are incorrect:
  pre: `Let m:OCLMessage = self^^authenticate(login,password)->first()` **in**
  `m.hasReturned() and m.result()=false`
  post: `sender^SignInDisplay(login,password)`
- user retries:
  post: `self^authenticate(login,password)`
- the main sequence continues then normally starting from step 4

4.2.3 Operation Model

The purpose of the operation model is to define the effects of the *system operations*:

Definition 20 (System Operation) *A system operation represents a single unit of behavior offered by a use case and called by an actor.*

The operation model takes the form of a textual template declaratively specifying operation behavior in terms of OCL pre/postconditions.

Template

Operation Name: `<name of operation>`

Related Use Case: `<the use case the operation relates to>`

Description: `<natural language description of the operation>`

Parameters:

1. `<parameter name>` **typeOf:** `<parameter type>`

Sends:

1. `<signal name>` **to** `<actor>`

Preconditions: `Natural language and OCL statements`,

Postconditions: `Natural language and OCL statements`.

Instantaneous and Asynchronous Execution of Operations

FIDJI operations follows an *atomic* execution scheme as proposed in the Fusion [CAB+94] method; there is no way to see the sequence of changes made to system states starting from the satisfaction of the preconditions to the satisfaction of the postconditions, we consider that the operation executes instantaneously and that, at any point in time there is only one operation that is active. Hence, there is no issue in concurrent access to shared variables problem within operations since they are processed in "zero-time" sequentially. Though unrealistic, this semantics allows to focus on the individual behavior of operation and avoids concurrency issues thus simplifying the analysis model, which is in line with the FIDJI requirements. Furthermore, the technology targeted by the FIDJI method at the implementation level, J2EE EJBs [Mic05], relies on container implementations or third party software (such as relational databases) to support concurrency.
However, as noted by Sendall [Sen02], this semantics does not help to design the system so that these conditions can be actually satisfied in a distributed manner. Hence, he extended the operation model to support shared access to resources for concurrent executions of operations and proposed means to ensure synchronization. The approach consists in defining special tags to be added on OCL pre/postconditions as well the definition of clauses allowing to state which variables are shared. In this approach, atomicity is confined to variable updates rather than on the operation as a whole. Such an extension can be considered in future work on the FIDJI method. However, to be fully valuable, the semantics of this extension has to be formalized (possible formalization of the FIDJI models will be discussed in Chapter 8).

We mentioned in Section 4.2.2, that interactions between actors and the system were asynchronous. Hence an operation does not have a return type (which would includes that the actor actually "waits" for the return value), but may send signals to the calling actor (as well as to other actors) as specified by the "Sends" clause in the operation template.

Pre/Postconditions

The OCL standard [OMG05e] states that "the purpose of a precondition is to specify the conditions that must hold before the operation executes". It is the responsibility of the analyst to define scenarios so that the operation cannot be called if its precondition is not satisfied. At the design level, one can devise a fault tolerance mechanism (such as exception raising) that prevent the execution of the operation behavior if its precondition is not met.
The situation is different for postconditions: "The purpose of a postcondition is to specify the

conditions that must hold after the operation executes". This means that it is the responsibility of the operation to ensure the that the conditions hold and hence part of its contract.

It is sometimes useful to refer to the sender of a message to, for instance, send a signal back to him. However, in OCL the `OCLMessage` construct do not provide this functionality (at least at the M1 layer in the UML metamodeling hierarchy). In [SS01], Sendall and Strohmeier suggest to employ the keyword *sender* to alleviate this problem, in our operation specifications we use it.

Operation Model and Life-cycle

We have defined the use case model in order to provide a partial yet practical on view on the system's life-cycle (allowable sequence of operations). But reasoning about the life-cycle can also be done in the operation model, since state variables are also used to express operation preconditions and postconditions. Examining these conditions would allow to reconstruct all the possible sequences of operations of the system. However this number is huge and it is not the intent of the FIDJI method to model them explicitly. But reconstructing only a partial sequence related to the specific behavior of an operation; this helps analysts to discover dependencies between operations and designers to implement them. Naturally, reconstructing these sequences "by hand", though possible for small sequences may become unmanageable for larger sequences. Providing a systematic approach to life-cycle reconstruction is linked with the formalization of the FIDJI models which is one of the important perspectives envisioned in Chapter 8.

4.2.4 Analysis Instantiation Constraints

As instantiation constraints are defined in OCL, they can only apply to the elements of the analysis model that are defined in UML:

- Domain Diagram: constraints on this diagram allow to specify which concept(s) should be obligatorily reused or if their attributes may be altered,
- Use Case Diagram: constraints defined on this diagram are interesting to control the reuse of use cases in their entirety. For example, architectural framework developers may wish that a particular use case is present in all the products instantiated form the architectural framework (mandatory or "kernel" use case),
- Use Case Component Diagram: constraints defined on this kind of diagrams allow for a fine tuning of the use case reuse: operation(s) considered as mandatory, alteration of some parameters etc.

We will give concrete examples of such constraints in Chapter 6. The scope of application of constraints go beyond the diagram on which they are defined: for example defining a concept as mandatory via an invariant also impacts the use case component (and operations) that use this concept and hence should satisfy the invariant. Furthermore, the application of the well-formedness rules of the analysis profile (Section 4.2.5), implicitly generates new constraints form the original ones: e.g. the satisfaction of the rules which states that all operations present in the use case components should be further detailed in the operation model imposes that an operation defined as mandatory at the use case component level should also be reused at the operation model level. This situation is one example of impact analysis that we will address in Section 4.4.

4.2.5 FIDJI Analysis Profile

In this section we gather the specific modeling elements used in our analysis model in the form of an UML 2.0 profile, and give their associated well-formedness rules.

Profile

Figure 4.3 sums up the stereotypes used in FIDJI analysis models and table 4.5 describes the purpose of each stereotype.

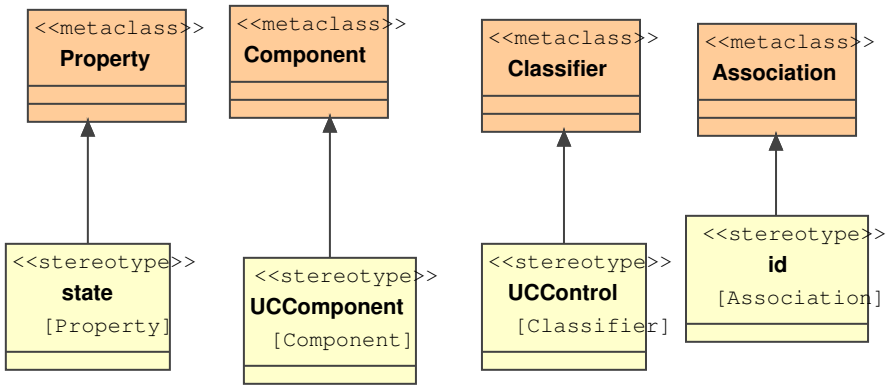

Fig. 4.3: The FIDJI Analysis profile

Stereotype Name	Base Metaclass	Description
<<UCComponent>>	Component	Use Case Component
<<state>>	Property	Used in Domain concept and use case components to store information about the state of the system to be used in OCL expressions
<<UCControl>>	Classifier	Classifier gathering state variables for a given use case component
<<id>>	Association	Relates an external actor to its system representation

Tab. 4.5: FIDJI Analysis Stereotypes

Well Formedness Rules

In this section we explicit the well formedness rules that apply to the FIDJI analysis modeling elements. These rules are given both in English and OCL when possible. They are also organized according to the sub-model (domain, use case and operation) to which they apply. In Figure 4.4, we recall here the structure of the FIDJI Analysis Model in order to give OCL expressions context:

In order to ease writing of OCL expressions concerning stereotyped model elements, we define the following query operation that retrieves all the stereotypes associated to a given metaclass:

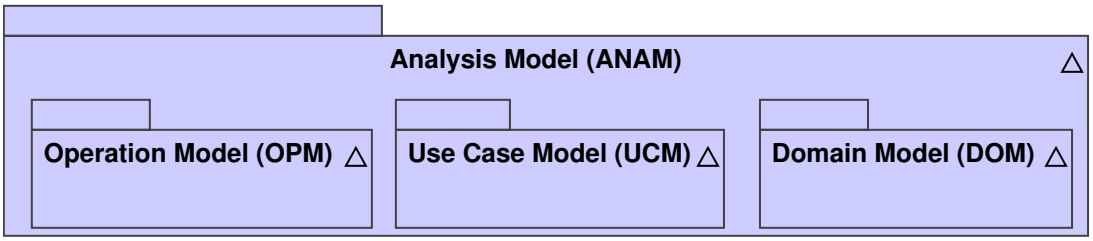

Fig. 4.4: FIDJI Analysis Model Structure

```
Class::allStereotypes():Set{Stereotypes}
pre: --none
post: result = self.extension.ownedEnd.type
```

Global Rules

WFR-1 All the operations signatures present in the interfaces of the use components in the use case model must be fully defined as operations in the operation model.

WFR-2 All the parameters and signals sent by system operations must have been defined in the domain model (not expressible in OCL).

Domain Model

WFR-3 Concepts are not allowed to own any kind of behavior:

```
context ANAM::DOM inv:
self.ownedType->select(c|c.oclIsKindOf(Class)).Feature
->select(f|f.oclIsKindOf(BehavioralFeature))->isEmpty()
```

WFR-4 All the concepts and signals depicted in the domain diagram must be present in the domain dictionary (not expressible in OCL).

WFR-5 All the properties of concepts and signals must have a type:

```
context ANAM::DOM inv:
self.ownedType->select(c|c.oclIsKindOf(Classifier))->forAll(c|
c.attribute->forAll(p|p.type->notEmpty()))
```

Use Case Model

WFR-6 The only relationship that is tolerated between use cases is inclusion:

```
context ANAM::UCM inv:
let rel: self.ownedElement->select(r|r.oclIsKindOf(Relationship)
and r.relatedElement->forAll(e|e.oclIsTypeOf(UseCase))) in
rel->forAll(r|r.oclIsTypeOf(Include))
```

WFR-7 There must not be more than one classifier with stereotype "UCControl" per use case component:

```
context ANAM::UCM inv:
self.ownedType->select(c|c.oclIsTypeOf(Component))->
forAll(c|c.ownedType.oclAsType(Class)->allStereotypes()->
select(s|s.name='UCControl')->size()<=1)
```

WFR-8 Within an <<UCControl>> stereotyped classifier, the only allowed model element are properties stereotyped by <<state>>:

```
context UCControl inv:
self.base_Classifier.attribute->forAll(p|p.oclAsType(Class)->allStereotypes()->
forAll(s|s.name='state'))
```

4.3 Transitioning from Requirements Elicitation to Analysis

In this section, we explain how can REQET-based descriptions described in Section 4.1 be linked with FIDJI analysis models. Establishing these traceability links is of primary interest for domain engineers who need to build a FIDJI analysis models satisfying a SPL elicited with the help of the REQET template.

4.3.1 Traceability

Before addressing traceability amongst REQET and FIDJI analysis model elements, we need to assume a well-formed template description according to the hints given in Section 4.1.4. Because REQET-based descriptions are more abstract than FIDJI analysis descriptions, there are one-to-many relationships amongst them. In this context, we cannot apply UML-based traceability approaches presented in Chapter 2 since REQET is not based on UML. Furthermore, as we work at the domain engineering level here, traceability links only need to be established once for a given SPL. Hence we propose to document traceability information manually via tuples of the form:

$$< e_{REQET}, \{e_{FIDJI_1} \dots e_{FIDJI_n}\} >$$

Where e_{REQET} is designating a REQET-based element (use case or domain concept) and $e_{FIDJI_1} \dots e_{FIDJI_n}$ are elements belonging to the FIDJI analysis model of an architectural framework. We do not provide any way to define types for these traceability relationships; indeed, two types are implicitly defined:

- Use Cases: This kind of tuples relates UCET-based descriptions with FIDJI analysis use cases. In this case, members of the tuples are use case identifiers,
- Concepts: These tuples relate concepts described in the DOMET with those explicated in the domain dictionary at the analysis level. Members of these tuples are simply concept names.

These tuples are provided in a textual file. Furthermore, in order to validate traceability information the following well-formedness rules are defined:

WFR-9 All UCET-based use cases must be involved in a use case traceability tuple with FIDJI analysis use cases.

WFR-10 All the members of the DOMET must be part of a concept traceability tuple.

Note that these rules are not bijective; there are usually more than one analysis element in the tuple. Two reasons explain this matter of fact. Firstly, as we highlighted in Section 4.1, REQET-based description do not strive to be exhaustive but rather concentrate on documenting variability in the SPL elicitation requirements. Thus, some common elements may be omitted in the description and discovered at the analysis level. Secondly, the abstraction gap between requirements elicitation and analysis implies that some elements may be required at the analysis level to provide detailed specification of a particular functionality that are irrelevant at the elicitation level.

4.3.2 Relating Variability Information

As we have mentioned above, the FIDJI method specifies variability in terms of instantiation constraints in the analysis and design layers of the architectural framework supporting a given SPL. Therefore completing traceability between REQET-based descriptions and FIDJI analysis models means also encoding variability information of the DOMET and UCET in terms of instantiation constraints. We can use the traceability format given above to identify analysis elements that are concerned by a given variation. First, we define instantiation constraints for analysis concepts corresponding to mandatory and alternative DOMET concepts. Optional concepts do not deserve instantiation constraints unless they are depending on other element that are not optional (in that case the dependencies column of the DOMET should help in the elaboration of such constraints). Use case variations are processed the same way. Second variation point descriptions defined within a particular use case have to be treated specifically; for example an optional sub-scenario in a mandatory use case will imply that at the analysis level that if the use case should be present in the product some of its operations may be omitted.

As we illustrated in the previous section, the implementation of variants of the SPL is done via the instantiation program. Hence, in order to assist application engineers in the writing of the instantiation program, domain engineers may provide sub-transformation programs realizing the implementation of the variants identified in the REQET-based description and traced in the analysis layer via the above mechanism. However, this is an optional practice; application engineers are free to use the transformation operations they wish to instantiate the architectural framework's analysis layer. In both cases, the instantiation will be controlled by the consistency rules defined between REQET variants and their implementation via transformation primitives at the analysis level.

4.4 Product Elicitation and Analysis

In this section, we provide the necessary methodological rules to perform the elicitation and analysis on the basis of an architectural framework conforming to the models presented above.

4.4.1 Define Product

As illustrated on the SPEM model depicted in Figure 3.4, the FIDJI process begins with the "Define Product" task at the requirements elicitation level. The definition of a product involves two stakeholders; the customer of the product line member to be developed and the product analyst. In first place, the product analyst resolves the appropriate variation points while producing the product description (which follows a similar template) in accordance with the client. Then, he adds new concepts and use cases which correspond to product specific requirements that are not present in the SPL description. Finally, the product requirements are checked against REQET validation rules presented in Section 4.1 and validated by the customer.

4.4.2 Instantiate architectural framework Analysis Layer

Identifying Concerned Elements

The first step to accomplish in the architectural framework analysis layer instantiation activity is the identification of the architectural framework analysis layer elements that are necessary to build the product analysis model. Concept names and use case identifiers present in the product description are compared for matching with the first element of the traceability tuples introduced in the previous section. When both the product element and the tuple match, thus meaning that this element was initially present in the REQET-based description of the SPL, the set of analysis elements concerned by this product requirement is obtained via the second member of the tuple. By performing the union of these sets we can identify the analysis elements that will be used as source of the model transformation operations in the instantiation program. Note that the matching procedure can be skipped if the product analyst is realizing both requirements elicitation and analysis of the product because he may already know which elements have been directly copied from the REQET-based description to elicit the product.

Writing and Validating the Analysis Instantiation Program

The second step is the writing of the instantiation program in itself. It takes the following inputs:

- The product description,
- The set of architectural framework analysis elements identified trough the traceability links,
- The optional (sub)-instantiation program(s) performing the instantiation of analysis elements according to the variability information found in the REQET-based description of the SPL (see previous section),

- The analysis instantiation constraints which informs the analyst on the restrictions applying to the architectural framework analysis layer and on the transformation operations he may not use.

There is no specific order to write the analysis instantiation program, though the program will be evaluated imperatively. The product description is used to create at the analysis level elements corresponding to requirements that are specific to the product and therefore not present in the architectural framework. Depending on the availability of predefined transformation operations, the product analyst may use them to perform the instantiation of analysis elements identified via the traceability links. Throughout the whole writing of the program, the product analyst should consult the instantiation constraints as a guide to help him instantiate the architectural framework correctly.

Once the program is written (examples of program syntax and transformation operations are given in Chapter 6), it has to be validated over the instantiation constraints presented in Section 4.2.4.

Assessing and Resolving Impact

Once the analysis instantiation program has been validated, the actual generation of the product analysis model can be performed. The last step is to evaluate and resolve impact on elements which depend on the transformed architectural framework analysis layer ones. Impacted elements can be discovered by examining the following issues:

- UML metamodel and FIDJI profile unconformity: At the analysis level, derived use case component diagram may violate either UML or FIDJI analysis profile well-formedness rules; therefore the derived model may need be updated consequently,
- Inconsistencies with other depending analysis models: The removal of an operation from a derived use case component involves that its description in the operation model is no longer valid; FIDJI analysis profile constraints cover this kind of issues,
- Impacted Elements: In addition to the preceding point, a change on a particular attribute of a concept or parameter of an operation may have consequences on the other depending elements (even if the whole model is conforming to UML metamodel and FIDJI analysis profile).

5. ARCHITECTURE & DESIGN

Abstract

In this chapter, we present the architecture and design phase of the FIDJI method. This chapter is structured in a similar manner as the previous one; Section 5.1 presents the requirements of the design models with respect to the analysis phase and the FIDJI process. Section 5.2 presents the models used for the static and behavioral modeling of the architectural framework design layer. Section 5.4 presents the architecture and design profile as well as rules for validating design models. Section 5.3 gives the traceability information with respect to analysis. Finally, Section 5.5 gives the methodological guidelines to perform architectural framework instantiation at the design level.

5.1 Requirements for Design Models

The role of the design phase is to define the "how" of a particular system; that is, the decomposition the system in structures (i.e. its architecture as we seen in Chapter 2, Section 2.3) as well as the detailed behavior of these structures. Naturally this "how" is related to the "what" of the analysis phase; one needs to understand how the requirements defined in the analysis are mapped to software elements that will constitute the software system. With respect to our analysis phase, we have identified the following characteristics for the design phase to provide:

- **Flexible Product Derivation**: At the design level, some mechanisms must be provided to ensure the same level of controlled flexibility to reuse models as did state variables and use cases at the analysis level and provide adequate support to guide product developers in the instantiation of the architectural framework,
- **Architectural Support**: These models should be able to expose the system architecture clearly. In particular, it should be able to describe architectural styles implemented by the object-oriented framework at the code level,
- **Behavior**: Design models should be able to describe the algorithms realizing operations that have been defined in the operation model.

5.2 Design Models

In this section we present the design models offered by the FIDJI method. We illustrate them through an example which is a sophisticated form of "Hello World" that was implemented as a sample application of the JAFAR architectural framework [GR02, GS02b, GRS03a].

5.2.1 Structural Modeling

The structural model consists of three sub-models: the global architecture model, the internal structure model, and the parts specification model.

Global Architecture Model (GAM)

The role of the GAM is to give a high level description of the structural architecture of the architectural framework according to the component-and-connector viewpoint. Methodologically, the elaboration of this model corresponds to the architectural phase of the design process, while the subsequent models are dedicated to lower level description of individual components at the design level.

Components

In the GAM, we focus on the "externally visible" properties of components and their interaction with other components through connectors, ports and interfaces. Similarly to KoBra komponents [ABB+02], FIDJI components form a tree-like containment hierarchy in which the whole system is seen as a root component which is decomposed in more fine grained components. We consider three kinds of component:

- **Primitive Components**: primitive components are simple components that do not own other components and can be designed directly in terms of classes, operations and relationships among them. They are the leaves of the containment tree,
- **Compound Components**: Compound components are larger structures that own a sub-component decomposition; they are forming intermediate nodes of the containment hierarchy which own decomposition in sub-components. Such a decomposition can either result from the application of a particular architectural styles or the will to modularize the internals of component so that its architecture description, design and implementation is facilitated. In order to distinguish them from primitive components they are annotated with <<compound>> stereotype,
- **Root Component**: The root component is a special type of compound component which is a at the top of the containment hierarchy. The root component plays the role of the context realization model in KoBra. The boundaries of the root component defines the boundaries of the system and its interfaces are formed with the interfaces of use case component defined at the analysis level. This helps ensuring traceability between analysis and design layers of the architectural framework as we will discuss it in Section 5.3. The root component is annotated with the stereotype <<root>>.

As noted by Ivers et al. [ICG+04], there are two ways of representing components in UML 2.0: using instances of the metaclass `Class` (from the `StructuredClasses` package) and using instances of the `Component` metaclass. The only significant differences between these constructs is that components can own additional elements (as we have seen it for use case components in Chapter 4) such as deployment descriptors and have the ability to be supported via technology-specific profiles such as the ones for EJB. As the FIDJI method targets applications developed with the help of such technologies we chose to model our components thanks to the `Component` metaclass. As an additional incentive, Ivers et al. [ICG+04] also remark that connectors may be documented with classes which may induce some confusion.
In the GAM we do not model the internal structures of individual components explicitly, (it is the role of the ISM and PSM); internal structures are reserved for compound and root components and are represented in separate component diagrams showing their decomposition in sub-components.

Ports

As we saw in Chapter 2, Section 2.2, an UML 2.0 ports specify distinct interaction points between a classifier and its environment or between the behavior of the classifier and its internal parts. The main use of ports in the GAM is to to encapsulate lower-level design patterns characterizing the interaction of the component with its parts/interfaces and whose details are not relevant in this model. Consequently the types of ports (which are classes responsible for the application of the design pattern) are not shown in the GAM; they will be given in the ISM and PSM. Ports are optional in this model.

Connectors

Since we do not show the internal structure of the components in the GAM, the only kind of UML connectors is the assembly connectors linking provided interfaces/ports with required interfaces/ports. It is important to carefully model the connectors at this level for the following two reasons. First, connectors participates in the architectural qualities and the decisions that have been made for their definition strongly influence design. Second, precise modeling of assembly connectors can be done only within the GAM, because ISM and PSM are focusing on detailing the components internal details and not interactions with their environment.

Interfaces

Interfaces in the GAM are modeled differently depending on the kind of components they are attached to:

- **Compound/primitive components**: In these components interfaces are modeled within an architectural connection [AG97] (supported in UML as an assembly connector) using our OCL-based approach that will be detailed in Section 5.2.2,
- **Root component**: Provided interfaces of the root component correspond to the interfaces of the use case components modeled at the analysis level. They can be modeled either using the complete UML notation for interfaces or in terms of "lollipops" to reduce visual

clutter. Required interfaces (corresponding to services provided by external systems) need to be defined completely since they will not be detailed in the other design models.

Modeling architectural styles with the GAM

As the GAM provides a high-level overview of an architectural framework architecture and design, it can describe the architectural styles which affect its overall structure and guaranties that certain qualities are met in instantiated SPL members. We will integrate constructs allowing to model architectural styles in our design profile (see Section 5.4).

We have selected the following architectural styles that we found relevant with respect to the component-and-connector viewpoint, GAM's purpose and FIDJI method: pipe-and-filter, layered style, N-tier.

The layered style, which have been already mentioned several times, divides the system in communicating layers. In the *strict* variant of the pattern, a component of the layer N is only allowed to communicate with adjacent layers (N-1 or N+1). The *flexible* variant allows layers to communicate arbitrarily which may possibly lead to a too loose hierarchy that eventually will limit the interest of decomposing the system in layers. Therefore we propose to model only the strict variant. In order to do so, we suggest to use the <<layer>> architectural primitive defined in [ZA05]; it is an extension of the UML `Package` metaclass which has a tag definition indicating the number of the layer in the hierarchy. An OCL constraint enforce the strict variant rule. In middleware-based architecture descriptions, there can be a huge number of assembly connectors and interfaces characterizing the interactions between two components. In order to keep this situation tractable within the GAM, we introduce the notion of <<abstractConnector>> as an extension of the the `Connector` metaclass that gathers all the connections in a single one. The <<abstractConnector>> connects two <<abstractInterface>> which can be understood as the union of the interfaces participating in the connection. The abstract connector is also useful when we do not want to detail a complex path of components and connectors; in this usage it has the same semantics as the "virtual connector" defined in [ZA05].

The N-tiered style is typical of web-based architectures and can be seen as a generalization of the client-server one. It relies on the separation of the GUI, business logic and data. For example J2EE-based applications are a good example of N-tiered (here N=3) architectures using JSP and HTML for the GUI tier, EJB session beans for business logic and Relational Database Management Systems (RDBMS) to store data defined in the entity beans. Additional tiers can be added to provide caching mechanisms between business logic and RDBMS. The N-tiered style differs from the "layer" style in the sense that the decomposition is not hierarchical and based on the abstraction level of components but rather functional or physical; tiers help to identify the physical nodes on which will be running the components they own. We model this style similarly to the "layer" style via the stereotype <<Tier>>.
We will give concrete examples of modeling architectural styles in Section 5.2.2.

Internal Structure Model (ISM)

The role of the internal structure model is to exhibit the internals of each individual component that has been abstractly modeled in the GAM. It represents the first level of structural design of

components. The second will be covered in the parts structure model. The ISM is mainly based on the UML 2.0 notational elements that are found in the `Composite Structures` package. We detail the ISM constructs in the following paragraphs.

Ports

Ports are the first elements to deal with when elaborating the ISM because they represents the "entry point" to the component's internal elements. There are two cases:

1. Port Refinement. In this case, ports for this component have already been defined in the GAM. The refinement consists in attributing a type (i.e. a class) to all ports of the component,
2. Port addition. We have mentioned that ports are optional in the GAM. In the ISM, it may be necessary to create some new ports to detail the interaction between provided and required interfaces and the internal parts of the component.

In both cases, we require that at least one port is present for each component and that each port has a type. Although ports can be refined, no port defined in the GAM can be removed in the ISM. Port types are completely described in the parts structure model.

Parts

Parts refer to classifiers' instances which are participating in the design of the component and attached via connector the component's ports. There are fully detailed in the parts structure model.

Connectors

Connectors have two roles in the ISM:

1. Call or provide methods to components' interfaces. We use delegation connectors to connect parts to ports and therefore to describe how provided interfaces are designed across the component parts or to require some methods form other components,
2. Links between parts. Parts may require the participation of other parts to fulfill their functionality; we use instances of `Connector` (from `InternalStructures` package) in order to relate parts.

Figure 5.1 shows the internal structure of the `HelloWorld` component:

- Interface `IHelloWorld` is the main provided interface to clients of this component; it offers `sayHello` and `sayBonjour` methods (which respectively add `Hello` and `Bonjour` before user entered name). Interface `IDelegate` is defined within the framework and should be implemented by "delegate" classes (see below) in order to perform initialization tasks,

- Port `pHDel` is providing `IHelloWorld` interface. Its type, `HelloWorldDelegate`, is a class that handle requests to `HelloWorld` service according to the "business delegate" pattern [ACM01]. This pattern suggests to create an intermediate class that handles clients requests and perform various tasks not directly related to the service functionality which is implemented in the beans. Delegate classes minimize the coupling between client requests and EJB design thus facilitating changes. `HelloWorldDelegate` needs access to several other classes to implement its functionality; connectors to anonymous instances of `ServiceInvocation`, `ServiceInvocationResult` and `DelegateHelper`. Finally the business part of the interface (i.e. displaying "hello" or "bonjour" before an user given name) is delegated to `HelloWorldBean` which is a session bean that actually realizes the service behavior.

Parts Structure Model (PSM)

The role of the parts structure model is to fully define all the parts that are sketched in the internal structure model. Hence, it completes the static description of components in a fine-grained way. Similarly to the ISM, it is attached to a single component. The PSM plays the roles of the inheritance, dependency and design class models in Fondue (however as Fondue is not component-based, their models concerns the whole system ad not only a component) and the structural realization model for KoBra komponents.

The graphical notation offered by the PSM is based on UML 2.0 class diagrams. We detail the particular usage of these UML constructs in the following paragraphs.

Classifiers

Classifiers include the full specification of their attributes (name, type and visibility) as well as method signatures (name return type and parameters). Each class in the PSM should correspond to one part abstractly defined in the ISM. There are some classes (such as the architectural framework ones) that are not owned by the component itself; they are also fully specified but we require that their explicit owning package be given.

Relationships

Relationships amongst classifiers are important since they determine the features that have to be provided for a classifier and the features it needs to access via other classifiers and how. These relationships can be of the following kinds:

- **Inheritance**: Inheritance relationships allow the determination of a classifier type and therefore the attributes and methods it gets from its parent(s). In addition, it may override methods (or implement them in the case of the realization of an interface) and/or add attributes and methods in order to fulfill its parent(s) contract and/or add new features. Graphically, inheritance relationships can take various notations: they can be instances of `Generalization` (classical "is-a" relationships), `InterfaceRealization` or can be modeled in a indirect way via stereotypes; e.g. in the EJB component model, the application

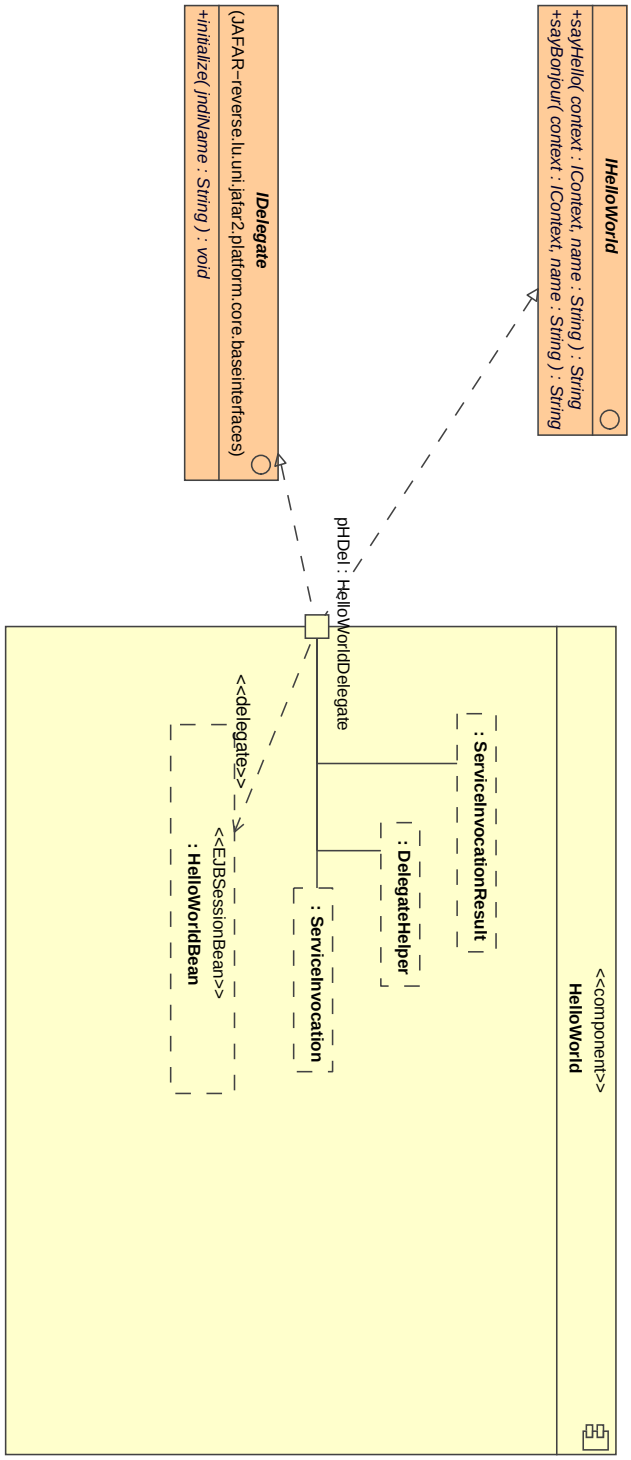

Fig. 5.1: HelloWorld Internal Structure Model

of the <<EJBSessionBean>> stereotype will inform the designer that the class will have to implement session bean methods and therefore realize the `SessionBean` interface. If J2EE stereotypes are supported within a UML case tool, this link may even be defined automatically at the code level. The role of such relationships is equivalent to Fusion inheritance graphs,

- **Association**: Association relationships identify the classifiers that are needed by a given classifier to perform its contract. We require that the relevant classifiers should be modeled as associations rather than attributes. Classifier attributes should primarily concern primitive types. In some cases, to avoid visual clutter, classifiers belonging to a well identified library (such as J2EE) can be modeled as attributes,
- **Dependency**: Dependency relationships concern indirect links between model elements. For example, they may involve interface realizations that are handled by the J2EE container rather than by a statement in the implementation code.

Figure 5.2 shows the parts structure model for `HelloWorld` component. It illustrates the various kinds of relationships discussed above: This diagram shows how port `pHDel` type, represented by class `HelloWorldDelegate`, indirectly provides the `HelloWorld` services (`sayBonjour` and `SayHello`) via the interface `HelloWorld`. This interface is indirectly realized (a J2EE container will ensure the mapping at run-time) by `HelloWorldBean` (hence the dependency relationship rather than a direct inheritance).

5.2.2 Behavioral Modeling

The role of the behavioral models is to describe how the various elements defined in the structural models interact and therefore support the behavior of the operations defined in the analysis models. More specifically, we distinguish two cases for behavioral modeling: *inter-component modeling*, whose purpose is to detail the behavior of the assembly connectors defined within the GAM and *intra-component modeling* which consists in describing the internal behavior of an individual component.

Inter-Component Model

We mentioned above that the main motivation for the GAM was to describe the overall architecture of the system and the architectural styles used to form it. In particular, the emphasis is put on the interactions between components, the other models presenting their internal details. As presented in Chapter 2, Section 2.4, existing approaches to model component interactions in the the context of SPL are using collaboration diagrams (Fusion [CAB+94], Fondue [SBS04], KoBra [ABM00], Gomaa's PLUS method [Gom04]) or sequence diagrams (Ziadi and Jézéquel [ZJ06]). However, as noted by Zdun and Avgeriou [ZA05] there are two issues in modeling architectural styles with UML.
Firstly, the UML language does not support natively architectural styles such as the pipe-and-filter style. Yet, we have provided solutions to alleviate this problem based on UML extensions. The second and most important point is related to the flexibility issue we dealt with at the analysis level; architectural styles do not define unique solutions but implicitly induce a solution space for recurring problems. Even if UML provides some mechanisms (e.g. the fragments that can be employed in sequence diagrams) can be used as parameters to the style to describe several solutions, the fact that architectural styles are often described informally hinders a direct translation in a parametric solution.
This motivates our will to provide an interaction mechanism between components that provide the necessary flexibility to describe architectural styles and interactions to be reused by the application developer to define the architecture of its product. In alignment with our analysis phase of Chapter 4, our goal is to provide a OCL-based approach similar to the one we proposed to complement use case descriptions (which are also interactions but between an actor and the

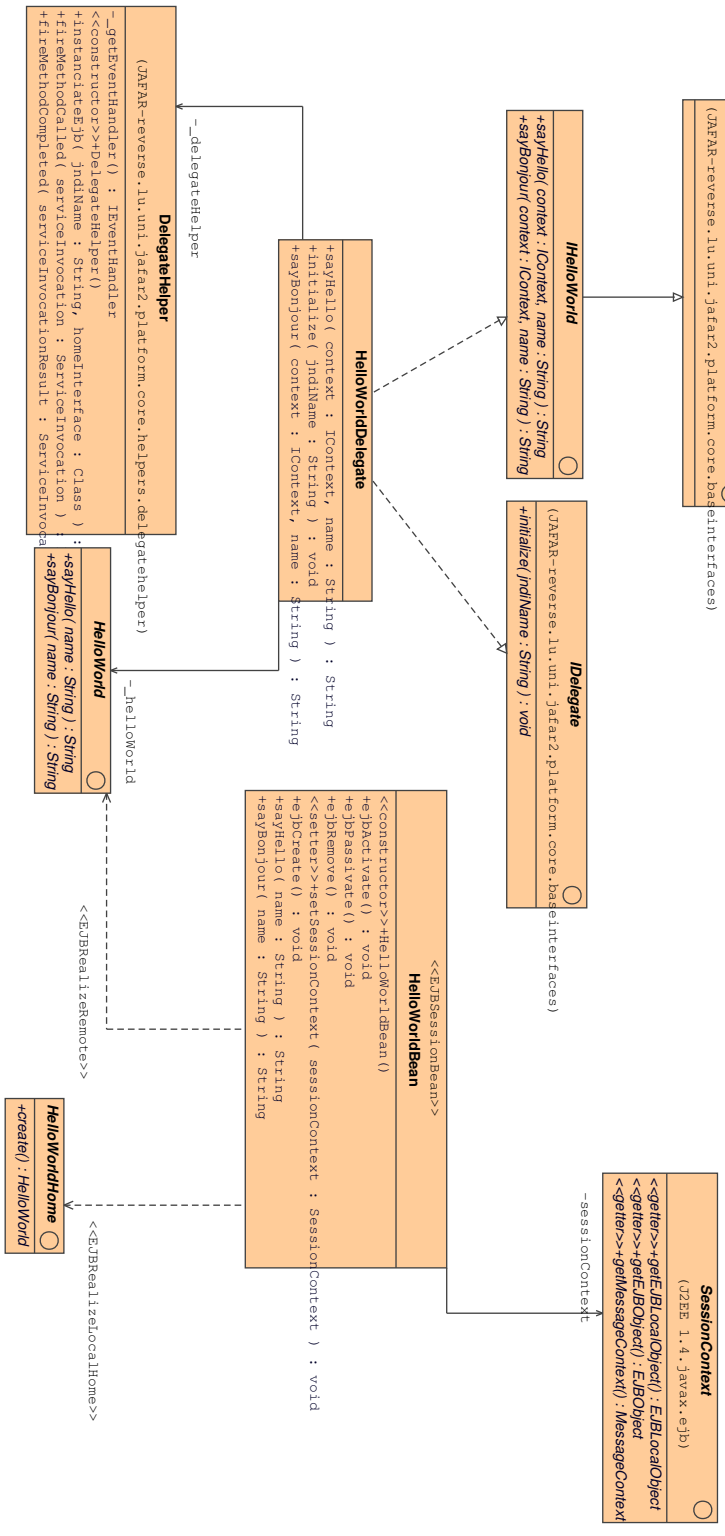

Fig. 5.2: HelloWorld Parts Structure Model

system). This goal is pursued for methodological grounds; use of a similar notation at the analysis and design layers reduces the learning curve of the method and facilitate the transition between FIDJI models.

Our general approach to interactions modeling is to adapt Allen and Garlan's architectural con-

nection [AG97] to OCL-based descriptions. In the context of assembly connectors used in the GAM, the roles are played respectively by provided and required interfaces or ports, while the "glue" refers to a particular interaction between components. The remaining paragraphs explain how OCL can be used in place of WRIGHT in order to model role and glue behavior.

Role Modeling

Modeling role behavior corresponds to the way provided or required operations of components have to be called (i.e. their protocol). Individual behavior of operations is being ensured by the internal structure of the component providing these operations. Roles are modeled as operations having no parameter and which are attached to connector behavior (see "Connector Modeling"). The definition of a role protocol is based on the elaboration of an OCL postcondition describing non-deterministically the sequence of UML messages (operation calls and signals) that have to be satisfied in the interaction with another component. As for the analysis phase, the main technique used to constrain temporally message exchange is via the definition of state variables that are defined at the interface level. Yet the way to handle them differs; in the analysis phase we used state variable as guards in the preconditions of each use case step. This is not possible here so we rely on the `if ... then ... else ... endif` OCL block to specify such guards. Note that it is also possible to define local state variables in the pre/postconditions via the `let ... in` construct.

The `hasSent` OCL operator requires that the recipient of the message is identified in order to be defined properly. While modeling of a single role there is always one role that is not directly identified. Indeed, two symmetrical situations occur depending on the kind of role being modeled:

- **provided**: Specification of operation calls to a provided interface are straightforward. However, we need a way to identify the interface or port that issued the call in order to specify a response (either synchronously via a return message or asynchronously via a `Signal`). We propose to introduce a special variable, `requirer`, to map the requiring interface or port in the interface behavioral specification,
- **required**: Similarly for required interfaces or ports, the variable `provider` designates the interface or port that provide services to this interface.

While modeling a connector role, we do not know the exact operations that will be provided or required by components through their ports and interfaces. There are two reasons to not having already such information:

1. **Method**: Our approach to architecture and design is top-down; we are going from the most abstract (the internal structure of the <<root>> component modeled as a GAM) to the most concrete (detailed design of individual components). Therefore we model component interactions without having specified their interfaces. In fact roles will act as guides for their design (see "Intra-Component Model");
2. **Architectural Styles Modeling**: In order to support the definition of architectural styles, we need to model roles independently of a particular architectural connection.

However, we need to make assumptions about the provided and required ports or interfaces (as identified by `provider` and `requirer` keywords) in order to model role behavior correctly. Hence we propose to define *virtual operations*. These operations define the minimum behavior that actual ports or interfaces should support in order to take part in an architectural connection. The context owning virtual operations is provided by the `provider` and `requirer` keywords. Hence virtual operations can be defined in the connector behavior via OCL pre/postconditions using such context. Virtual operations may either be directly used as part of component interfaces while they are designed or as guides to choose an existing component (in the case where the architectural framework design layer is built from pre-existing components). We will elaborate further on this point (see "Intra-Component Model").

Connector Modeling

Assembly connectors in the GAM may convey a specific information about the connection. For example, in the pipe and filter architectural style, one may specify if the interaction between two interfaces has to be secured or at a lower level, the nature of information exchanged (binary/ASCII streams) or the pipe buffer behaves if it is full. Zdun and Avgeriou [ZA05] proposed to attach OCL constraints (invariants) to model connector semantics. However they do not consider complex connector in their approach and modeling a sophisticated "pipe" with a buffer can be difficult using just invariants and may not be informative enough for designers and developers. Therefore we wish to reuse our interaction mechanism to model the connector's detail.

Until very recently (February 2007) and as opposed to numerous architecture description languages that treat connector behavior the same way as components, it was not possible to attach behaviors to UML connectors. Strategies such as the use of structured classes whose interfaces would match the provided and required interfaces of components have been proposed by Ivers et al. [ICG+04] to overcome this lack. The latest revision of the UML 2 specification (version 2.1.1) [OMG07b], now includes the possibility to describe connector behavior through an association called `contract` referencing any subtype of `Behavior` defined in the `Connector` metaclass. As a consequence, we have to define a `Behavior` instance defining the inner features (both structural and behavioral) of a connector and use OCL the same way we did for interfaces, i.e. as a postcondition defining the protocol of the messages exchanged. The `Behavior` metaclass factorizes all the common behaviors that can be modeled by the UML 2 specification such as state machines, sequence diagrams etc. Given the concerns we expressed about those above we chose to use `OpaqueBehavior` instances instead. Such instances are indeed modeled as UML classes which have pre/postconditions. We can thus apply our OCL based interaction approach through this mechanism.

"Gluing"

"Gluing" represents the actual formation of the connection in a particular model, that is how the separately modeled interfaces of components can be glued via an assembly connector. There are two options to perform the gluing process, applying a predefined connector type or defining a specific connector.

Applying a Predefined Connector Type

We devised above a mechanism to model assembly connector semantics without explicitly referring to components' interfaces. This allows to define connector types that are crucial when dealing with architectural styles. Gluing a connector type to actual components' interfaces/ports requires first that roles defined in the connector description match with components' interfaces/ports descriptions. Then, ports/interface protocols and connector "glue" have to be verified for consistency. Concerning the first point, one has to find a matching interface operation for each virtual operation involved in the role specification and determine their *compatibility*. At the syntactical level operations should be matched according to parameters and return types in their signatures. It does not mean that operation signatures are exactly the same; the set of parameter types of the virtual operation has to be a subset of the set of the actual interface operation parameters and return types. At the semantic level, compatibility means that the sate defined by the postcondition of the virtual operation should be one of the states reachable by the actual operation. This deliberately loose notion of compatibility between operations fosters flexibility and reuse by allowing already existing operations to fulfill several roles.

The second step is to check if the newly matched operations can be combined according to the connector glue. The key point to examine is the preconditions of the operations of the interfaces which may exhibit certain state variables requiring or preventing some operation sequences from being defined. Here also, the connection of the interfaces/ports may allow many more operation sequences than the ones defined by the connector glue; this is not a problem provided that the connector glue sequences are a subset of the interfaces connection possible sequences.
Once these two steps completed, one should obtain a complete description of the connector with all the virtual operations as well as `provider` and `requirer` replaced by the actual values of the interfaces/ports of components.

Defining a Specific Connector

Not all connectors are part of a style, sometimes one needs to create a specific connector to attach components. In this case, roles are directly modeled with the actual interfaces operations of the components. Hence, there is no matching of individual operations. Definition of the connector glue is similar to the predefined connector case and focuses on the elaboration of a postcondition that respects the authorized combination of interface operations.

Note that in both cases, the usage of state variables and the fact that OCL does not provide any means to specify operation sequences allow for a great number of possibility for architectural connections between components, which is a good thing at the domain engineering level. Naturally, there it is still possible to be more restrictive at the application engineering level by defining instantiation constraints inhibiting some connections.

An Example

Figure 5.3 exemplifies our approach on a pipe-and-filter system defined by the `Pf-Sample` root component. It consists of a simple "pipe" connector that use a buffer and two filters; `Filter1` send data through the connector while `Filter2` process data. The assembly connector is realizing

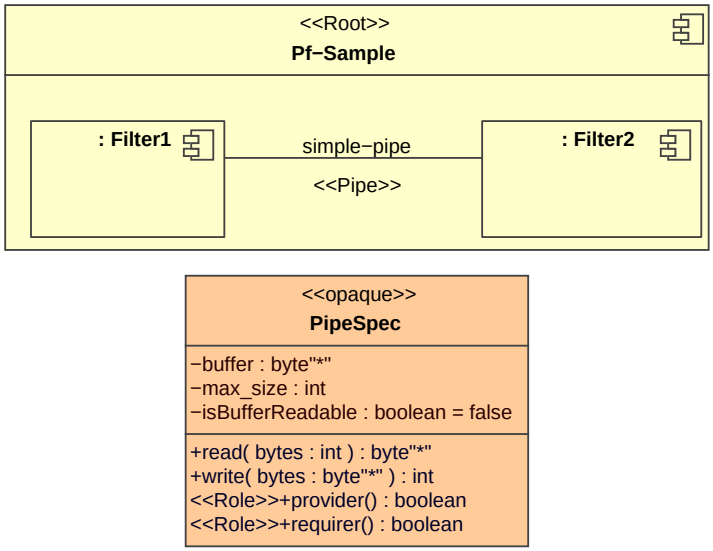

Fig. 5.3: A Pipe and Filter Example

the pipe via `PipeSpec`. This class consists of a data buffer whose size is constrained by the `max_size`, and two operations for reading and writing on the buffer. We impose that `write` have been called at least once prior to read on the buffer (using the "state variable" approach). We do not handle concurrent read/write in this example. The following constraints define more precisely this behavior.

```
context PipeSpec inv:
buffer->size() <= max_size

context PipeSpec::provider(): boolean -- provider role
post: let m: OCLMessage = provider^^getData(length)-> first() in
m.hasReturned() and m.result().oclIsTypeOf(Sequence(Byte))
and m.result()->size() = length
and self^write(m.result())

context PipeSpec::requirer() : boolean -- requirer role
post: let data : OCLMessage = self^^read(length) in
data.hasReturned()
and requirer^processData(data.result())

context PipeSpec::read(length : Integer): Sequence(Byte)
post: if buffer->size() <= length and isBufferReadable then
buffer = buffer@pre->excludesAll(buffer->subSequence(buffer->size()-length,
buffer->size()))
and result = buffer else
buffer->isEmpty() and result = buffer
endif

context PipeSpec::write(bytes: Sequence(Byte)): Integer
post: if bytes->size() <= max_size- buffer->size() then
```

```
buffer = buffer@pre->prepend(bytes) and result = bytes->size()
and isBufferReadable = true
else
buffer = buffer@pre->prepend(bytes->subSequence(buffer->size(),max_size))
and result = max_size- buffer->size()
and isBufferReadable = true
```

Here we also assume two virtual operations. One is `getData`, which returns an amount of raw data whose length in bytes is given as a parameter. The other is `processData` which refers to particular data processing operated by `Filter2`:

```
context provider::getData(size): Sequence(Byte)
post: result->size() = size

context requirer::processData(data: Sequence(Byte))
post: -- none
```

Finally, the following OCL postcondition details the pipe connector interactions.

```
context PipeSpec
post: provider^getData(length)
and let mpro : OCLMessage = self^^provider()->first() in
let mreq : OCLMessage = self^^requirer()->first() in
mpro.hasReturned() and mreq.HasReturned() and
mpro.result() = mreq.result = true
```

With respect to the approach of Allen and Garlan [AG97], the above postconditions may be seen as the "glue" for an individual connector.

Intra-Component Model

The role of the intra-component model is to detail the behavior of the individual components parts and interfaces. Therefore it is related at the structural level with the ISM and PSM. We distinguish two cases depending on the element being modeled:

- **Ports/Interfaces**: Ports and interfaces represent the interaction points of components with their environment and are used to form the actual connection between components in the GAM. Therefore, we use the same OCL dialect for modeling the behavior of ports and interfaces as the one used for connectors. This eases matching between component interfaces and connector roles (see "Gluing" above). More specifically, we use pre/postconditions to describe individually each operation of the interface while a postcondition defined on the whole interface specifies its protocol,

- **Parts**: Parts contribute together to the realization of the behavior specified by ports and interfaces. We put the emphasis on the definition of algorithms that will be implemented in the implementation layer.

As we have seen in Chapter 2, the UML specification provides the action semantics approach proposing an abstract syntax for defining behaviors either by using the predefined notation for activity diagrams and state machines or by using a specific action language. Our approach is to use an action language that has a textual syntax so that it is accessible to programmers performing component implementation but that is not a full-fledged programming language in order to abstract irrelevant details and to focus on method algorithms.

We chose KerMeta [MFJ05, FDVF06] to give a high-level description of the behavior of the methods present in classes composing the internal structure of a component. This choice is motivated by its ability to describe and manipulate UML models (based on a restriction of MOF), its operations over collections (which are close to the ones proposed by the standard OCL library) and its comprehensive tool support which provides syntax highlighting, model visualization and a compiler to actually run KerMeta programs and make models executable. As an example, Figure 5.4 depicts the KerMeta code for the `HelloWorldBean` as entered in its dedicated editor. `HelloWorldBean` implements the business logic of the component (i.e. returning a string of characters consisting of a greetings sentence followed by the name of the user inputted as parameter).

Note that KerMeta, though executable, is not suitable to implement the final code of the classes and methods composing components' internals. Indeed, it is not equipped to address the accidental complexity of the JAVA/J2EE platform; KerMeta environment does not provide access to the standard J2EE libraries and therefore cannot realize the indirect relationships between beans and their interfaces or the communication mechanisms between beans. This is not a problem, however. From a methodological point of view we believe that we need to address essential complexity and resolve accidental complexity at the implementation level (which is currently not supported by the FIDJI method). Additionally, it is possible to call JAVA code from KerMeta programs [FDVF06].

```
/* $Id: $
 * Creation date: August 14, 2007
 * License:
 * Copyright:
 * Authors:
 */
@mainClass "hello::"
@mainOperation ""


package hello;


require kermeta
require "SessionContext.kmt"
using kermeta::standard
//using SessionContext

class HelloWorldBean
{
    reference _sessionContext:SessionContext

    operation ejbCreate() : Void is do
    end

    operation ejbRemove() : Void is do
    end

    operation ejbPassivate() : Void is do
    end

    operation setSessionContext(sessionContext: SessionContext) : Void is do
    self._sessionContext := sessionContext
    end
    operation sayHello(name:String) : String is do
        result := "Hello, " + name + "!"
    end
    operation sayBonjour(name:String): String is do
    result := "Bonjour, " + name + "!"
    end
}
```

Fig. 5.4: KerMeta Code for HelloWorldBean

5.3 Transitioning From Analysis to Design

In this section we explain how we can trace models developed during the analysis with the design models. Indeed, the main purpose of the design phase is to distribute the behavior modeled in terms of use case and operation descriptions to the actual components of the system. A naive approach would be to use the use case components as the basis to derive architecture and design components on a one-to-one relationship. Such an approach is problematic mainly because preoccupations differ at the analysis and design phase: analysis is concerned with the overall functionality of the system while the architecture and design phase strives to find the best local arrangement of the physical components of the system. Thus, it is straightforward that a one-to-one relationship cannot be established between use case component and architecture and design ones.

Since the behavior developed at the analysis level concerns the overall system and its interactions with the outside (use case actors), it appears that the root component is an excellent candidate to ensure traceability between requirements and architecture. The root component's provided interfaces have to be the same than the ones provided by the use case component at the analysis level. Ports providing root component's interfaces have to ensure that the interactions defined at the analysis level are satisfied at the design level. Note that root component's ports do not implement interfaces directly (there cannot be behavior port since the root component role is to define the boundaries of the system not to implement its behavior) but delegate it to parts. This delegation, which characterizes traceability of system operations throughout the design, is modeled at the syntactical and semantic levels. At the syntactical level, We use the UML 2.0 delegation connector notation to relate ports of the root component to parts realizing part or whole of the provided interface behavior. At the semantic level, as ports can be typed by classes it is possible to give their operational semantics in terms of KerMeta code.
But ensuring traceability between analysis and design does not only concern system operations and their realization at the design level; concepts have also to be traced. In order to trace an analysis concept as one or more classes at the design level, we propose to use the special stereotype <<refine>> defined by the UML 2.1.1 specification [OMG07b]. This construct is a stereotype based on the <<abstraction>> dependency. This construct is more focused on relating elements at different abstraction levels than the more general <<trace>>, therefore it has been preferred in FIDJI.

5.4 Design Profile

This section consists of two parts; the core profile which defines the stereotypes and well-formedness rules for the FIDJI design models and the "architectural styles" part which illustrates how styles discussed in this chapter can be supported via UML 2.0 extension mechanisms.

5.4.1 Core Profile

The core profile describes the particular component types that are defined to perform FIDJI-based designs and the well-formedness rules to validate design models and to ensure transition between analysis and design. Similarly to the analysis profile, we provide the structure of the design model in order to ease the context definition of well-formedness rules. This structure is depicted on Figure 5.5.

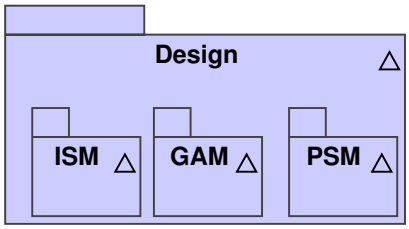

Fig. 5.5: FIDJI Design Models Structure

The `GAM` package contains all the components of the system as well as a class diagram representing their interaction and textual description related to assembly connector behavior. The `ISM` and `PSM` packages defines the internal structure and parts structure models, respectively. As we have mentioned in Section 5.2.1, the `ISM` consists of components that expose the same externally visible properties as the GAM components; therefore some well-formedness rules are shared by the two packages (for the sake of concision we will not repeat the whole OCL expressions if only their contexts change, we will only mention the contexts concerned by this rule sharing). The `PSM` contains the full definition (in terms of classes) of the types of properties exhibited as parts in the ISM components.

Stereotypes

Table 5.1 summarizes the core design profile stereotypes:

Well-Formedness Rules

WFR-11 Every Component defined in the `GAM` must have a corresponding component in the `ISM` that has the same externally visible properties (port names and interfaces):

```
Context Design::GAM inv:
let comp: Set(Component) = self.ownedType->
select(c|c.oclIsTypeOf(Component)) in
comp->forAll(c|self.nestingPackage.ISM.ownedType->
exists(ci|c.name = ci.name and c.provided = ci.provided
```

Stereotype Name	Base Metaclass	Description
<<compound>>	Component	Compound components have an internal structure whose (some or all) parts are typed by components.
<<role>>	Operation	Identifies an operation that represents a role in the behavioral description of an assembly connector.
<<root>>	Component	The root component represents the system as a whole and is the top-level component of the design hierarchy

Tab. 5.1: FIDJI Core Design Profile Stereotypes

```
and c.required = ci.required and
ci.ownedPort->forAll(p|c.ownedPort->exists(pi|pi.name =
p.name))))
```

WFR-12 There is only one root component (same rule for ISM):

```
Context Design::GAM inv:
self.ownedType->select(c|c.oclIsTypeOf(Component)
and c.allStereotypes()->
includes(s|s.name='Root'))->size()=1
```

WFR-13 The root component cannot directly own any behavior (same rule for ISM):

```
Context Design::GAM inv:
let root: Component = self.ownedType->
select(c|c.oclIsTypeOf(Component) and c.allStereotypes()->
includes(s|s.name='Root'))->first()
in
not root.feature->exists(f|f.oclIsKindOf(BehavioralFeature))
```

WFR-14 Being the top-level element of the design component hierarchy, the root component cannot be owned by another component (same rule for ISM):

```
Context Design::GAM inv:
let root: Component = self.ownedType->
select(c|c.allStereotypes()->includes(s|s.name='Root'))
in
not root.owner.oclIsKindOf(Component)
```

WFR-15 Each component of the ISM must have at least one typed port:

```
Context Design::ISM inv:
self.ownedType->select(c|c.oclIsTypeOf(Component)->forAll(c|c.ownedPort->
exists(p|p.type->notEmpty()))
```

WFR-16 All parts in the ISM must be defined in the PSM:

```
Context Design::PSM inv:
self.ownedType->select(c|c.oclIsTypeOf(Component))->forAll(c|c.ownedAttribute->
reject(att|att.type.oclIsKindOf(PrimitiveType)) =
self.nestingPackage.ISM.ownedType->select(c|c.oclIsTypeOf(Component))->
forAll(c|c.ownedAttribute->reject(att|att.type.oclIsKindOf(PrimitiveType))
```

The following rule covers the relationships discussed in Section 5.3 between analysis and design models.

WFR-17 All the interfaces designed at the analysis level should be provided by the root component:

```
Context Design::GAM inv:
let root: Component = self.ownedType->
select(c|c.oclIsTypeOf(Component) and c.allStereotypes()->
includes(s|s.name='Root'))->first()
in
self.nestingPackage.nestingPackage.ANAM.UCM->select(c|
c.oclIsTypeOf(Component))->provided = root.provided
```

5.4.2 Architectural Styles

In the remainder, we present how the styles illustrated in this chapter can be modeled via UML 2.0 extension mechanisms. Note that these architectural styles are not part of the FIDJI method in itself; they may be defined depending on functional and non-functional requirements of the SPL being modeled.

Layer

Table 5.2 presents the stereotypes of the layer style.

Stereotype Name	Base Metaclass	Description
<<abstractConnector>>	Connector	Connector gathering interactions between components located in two different layers.
<<abstractInterface>>	Interface	An abstract interface represents the union of a component's interfaces in order to support a grouped interaction with another component.
<<layer>>	Package	Represents a layer. The tagged value `layerNum` of type `Integer` identifies the layer number.

Tab. 5.2: Stereotypes for the Layer Architectural Style

WFR-18 An abstract connector has only two ends:

```
Context abstractConnector inv:
self.baseConnector.end->size() = 2
```

WFR-19 An abstract connector connects element located in the same layer or in adjacent ones:

```
Context abstractConnector inv:
let c1 : ConnectorEnd = self.baseConnector.end->first() in
let c2 : ConnectorEnd = self.baseConnector.end->last()  in
(c1.role.type.package.layerNum - c2.role.type.package.layerNum).abs() < 2
```

WFR-20 An abstract connector connects to abstract interfaces:

```
Context abstractConnector inv:
self.baseConnector.end->forAll(e|e.type.oclIsTypeOf(Interface) and
e.type.allStereotypes()->includes(s|s.name = 'abstractInterface'))
```

N-Tiered

The N-Tiered architectural style is composed of only one stereotype as defined in Table 5.3.

Stereotype Name	Base Metaclass	Description
<<tier>>	Package	A tier provides a namespace in order to group elements.

Tab. 5.3: Stereotypes for the N-Tiered Architectural Style

A tier is only a logical grouping of element and may have no existence in the physical system. Therefore if some elements are directly owned by a tier rather than imported, they may be destroyed.

WFR-21 A tier cannot own directly elements:

```
Context tier inv:
self.basePackage.packagedElement->isEmpty()
```

Pipe and Filter

We discussed this style in Chapter 2.3 and in Section 5.2.2 of this chapter in which we used this style to illustrate architectural connection in the GAM. There are several variants of the pipe and filter style, therefore here we focus on the commonalities between them. As a consequence we do not give any rule covering behavior because it is difficult to find rules that are general enough for this style. Table 5.4 presents pipe-and-filter style stereotypes.

WFR-22 A pipe connects only components that are instances of filters.

Stereotype Name	Base Metaclass	Description
<<pipe>>	Connector	Models a pipe through which data can be exchanged.
<<filter>>	Component	Filters are components that process data (such as conversion) and send them through pipes.

Tab. 5.4: Stereotypes for the Pipe and Filter Architectural Style

5.5 Design Process

In this section we explain how the above defined models constituting the design layer of the architectural framework can be instantiated to perform the application design. In the general FIDJI process depicted on Figure 3.4, these steps are composing the "instantiate architectural framework design layer" activity.

5.5.1 Identifying Concerned Design Elements

The first step is to identify the architectural framework design elements that will be concerned by the instantiation program. This is is done by examining use case components interfaces of the product at the analysis level; by comparing them with the actual interfaces provided by the root component in the GAM, it is possible to infer which interfaces need to be transformed by the instantiation program. Furthermore, depending on the way delegation connectors have been modeled between root components interfaces and its internal parts, the designer may also determine which internal components are concerned by the transformation. This further motivate the need to carefully model traceability while designing the architectural framework as explained in Section 5.3.

5.5.2 Writing Design Instantiation Program

Once the design elements to be transformed have been identified the instantiation program is written according to the previous step. As opposed to the analysis models instantiation in which no particular order is recommended, here we focus on the GAM first. GAM related instantiation sub-steps are as follows:

- **Update Root Component Interfaces**: The first instruction of the design instantiation program concerns interfaces update so that they match the interfaces provided by the use case components of the product at the analysis level. Note that part of the analysis instantiation program can be reused to perform this task,
- **Add/Remove/Update Components**: Depending on the defined delegation connectors between the root component's provided interfaces/ports, several components may require to be transformed in the application models as well as their interfaces,

- **Resolve/Update Architectural Connections**: Architectural connections may also be updated to take into account new component functionality or "dangling connections" in which some participating components has been removed.

At this point, some evaluation of the instantiation program against the architectural framework's instantiation constraints should be performed; this enables to validate the transformations in the component hierarchy of the product with respect to architectural framework's ones prior to work on the components internals. Indeed, if an instantiation constraint impedes a component to be added in the GAM, it would be a waste of effort to model its internals. Once this task completed, additional transformations can be specified to deal with component internals of the ISM and types defined in the PSM and the design instantiation program can be checked again against design instantiation constraints (in the case fine-grained constraints would concern components' internals, which is unlikely however), and the actual GAM, ISM and PSM of the product can be created.

5.5.3 Updating Behaviors

Once the the structural design models have been created, the application designer needs to update the behavior of these new models. This behavioral update follows the same order as models creation; first, interface definitions and architectural connections are updated to accommodate changes in components' interactions with their environments. Then, the internal behavior of components are given in terms of KerMeta descriptions.

5.5.4 Assessing Impact

As for the analysis phase, model transformation changing product components with respect to the architectural framework's components induces some impact on the untouched components. At the structural level, the GAM is a good starting point for impact assessment because it gives an overall view of the architecture in an abstract way therefore simplifying the identification of impacted components. As the GAM is fully defined in UML, an horizontal impact analysis approach as defined by Briand and Labiche [BLO03, BLOS06] can be employed; specific impact analysis rules can be defined as specific transformation operations specified in OCL. However has noted by Briand and Labiche [BLOS06], the fact that a given element is considered as a result of the application of an impact analysis rule does not necessarily mean that it needs to be changed, especially if the impact is indirect (that is depending on elements that are related to the element being changed rather than on the changed element itself). For example in the GAM, the modification of an assembly connector directly impacts the connected components. Concerning indirect impact, there are two possibilities; either the structural change in the GAM is inducing a localized behavioral change of the concerned components which is not affecting any of their other interfaces/ports (e.g. by removing a provided operation that is used only in the interface/port which is connected to the updated connector). The second possibility implies that the connector update is due to a global change of the components at the behavioral level and therefore may have a great subsequent impact. This information is not available at the GAM level. One needs to manually check the behavior of the components involved in the change to determine further impact. Note, that domain engineers being aware of component behavior may define specific instantiation constraints in order to inhibit updates in the GAM that would yield to global behavioral changes.

Part III

FIDJI IN PRACTICE

6. CASE STUDY

Abstract

In this chapter, we illustrate the FIDJI modeling language and product derivation process through a case study. This case study belongs to the e-barter domain in which people can exchange goods and services without necessarily using money. Section 6.1 illustrates the considered SPL called ***LuxDeal*** *and exhibits its commonalities an variabilities. Section 6.2 gives the FIDJI analysis models specifying this product line as well as their relationships with the REQET-based description by tracing requirement elicitation elements and documenting variability through instantiation constraints. Section 6.3 presents an excerpt of the architectural framework design layer and also explains its relationship with the architectural framework analysis layer. Section 6.4 illustrates architectural framework instantiation process by describing how a product can be obtained. Finally, Section 6.5 explains how, based on the previous product derivation, a new product can be quickly derived.*

6.1 Product Line Requirements Elicitation

6.1.1 Overview

The SPL considered in this dissertation is called ***LuxDeal*** and belongs to the general domain of the web e-bartering applications and whose some members have been developed as proofs of concept for the original FIDJI methodology [GPR04, GP02]. This kind of applications differs from traditional auctioneering applications in that items are not paid with money but rather exchanged against other items. Indeed a deal is initiated by a seller who defined an offer composed of a wish list and a belonging list. Bidders make proposals which include a list of belongings matching sellers' wishes and a wish list. During a deal, an offer can be amended and several proposals can be made until both parties agree. At this point the deal is concluded and other pending proposals are discarded. Note that deal agreement differs form traditional e-auctioneering applications in the fact that it is not the application that can determine automatically offer winner on the basis of the highest bid. In ***LuxDeal*** even if the estimated value of the items of particular proposal is higher than another the offerer may prefer the lowest one on the basis of the items proposed. Therefore, the attribution of the winning proposal is transfered to the offerer. Naturally, account management features such as registration, log in/logout and wish/belonging item management, are required to support the core functionalities of a ***LuxDeal*** member. In the following we will not be exhaustive about all these use cases; we will define the most relevant ones with respect to functional importance and variability.

6.1.2 REQET-based SPL Description

DOMET

As we described in Chapter 4, Section 4.1, the DOMET is dedicated to the description of the SPL concepts and their variants. Table 6.1 defines the concepts of the ***LuxDeal*** product line.

Concept Name	Var Type	Description	Dependencies
`Deal`	Mand	Deal is the core concept of the auctioneering process in ***LuxDeal*** applications. It gathers an initial offer and several propositions.	
`Item`	Mand	Subject of an exchange/sell/buy in ***LuxDeal*** applications. Items have title and description, an estimated value, a reserved price (price under which a seller will not accept to deal). Items can represent wishes and belonging. Within an offer they can be validated or not when the seller and the bidder deal.	
`Member`	Mand	Represents all the information about a user of a ***LuxDeal***-based application.	
`Offer`	Mand	An offer is the starting point of any deal, it comprises items (both wishes and belongings) and creation and expiration date.	
`Proposal`	Mand	A proposal represents a response to an offer within a deal. Proposals are composed exclusively of belongings (since wishes are assumed to match with the offer) and are ordered.	
`Category`	Opt	Category allows for gathering items in coherent sets. A category may own several subcategories (i.e. "Musical Instruments" includes "Pianos", "Guitars" and "Drums") and an item may belong to several (sub) categories.	

Tab. 6.1: DOMET for ***LuxDeal***

UCET

The UCET focuses on describing the behavior of the SPL members and their variants in an abstract way via high level or summary use cases. We give below the list of the most relevant use cases characterizing the main functionality of these members.

ID: `UC01`
Use case name: `Sign In`
Var. Type: Mand.
Description: A member signs in the application in order to access to the application's main services.
Actors: Member **Dependency**: None.
Preconditions: Member has created an account and has log in credentials (we assume a mandatory registration use case which is not detailed here). Member is not signed in.
Postconditions: Member is signed in.
Main scenario:

1. Member requests the application to show him a "Sign In" page,
2. The application shows a page comprised of a login form,
3. Member enters his credentials,
4. The application checks if they are valid,
5. Member is successfully signed in and the application presents the member his personalized home page.

Alternatives to the main scenario:
3a. If the member credentials are invalid, member is invited to retry (if this alternative is unsuccessful the main use case postcondition is not satisfied):

- Member is shown the "Sign In" page again,
- Member enters his credentials,
- The application checks if they are valid,
- If they are valid the main sequence continue starting from step 4.

Non-Functional: N/A.
Variation points description: None.

ID: `UC02`
Use case name: `Create Item`
Var. Type: Mand.
Description: A member wishes to define a new item.
Actors: Member.
Dependency: None.
Preconditions: Member is signed in.
Postconditions: A new item has been added to the member's account.
Main scenario:

1. Member requests to create an item,
2. The application displays a “create item” page comprised of a form,
3. Member enters title, description, [V1](category), estimated value, optional reserve price and states whether the item is a belonging or a wish,
4. The application stores the item definition in the member portfolio.

Alternatives of the main scenario: None.
Non-Functional: N/A.
Variation points description: V1: Type: Opt Concerns: Data, this variation point concerns an optional category which has been defined by the application owners (we do not treat category management here) and to which an item may belong.

ID: `UC03`
Use case name: `Create Offer`
Var. Type: Mand.
Description: Member defines a new offer in the system.
Actors: Member **Dependency**: None.
Preconditions: Member is signed in.
Postconditions: A new offer has been created.
Main scenario:

1. Member requests the application to show him a “Create Offer” page,
2. The application shows a page comprised of a form allowing him to define an offer,
3. Member enters the offer’s title, description and expiry date,
4. Member selects a set of items representing his wishes,
5. Member selects a set of items representing his belongings,
6. Member validates,
7. The application creates the offer.

Alternatives to the main scenario: None.
Non-Functional: N/A.
Variation points description: None.

ID: `UC04`
Use case name: `Browse Offers`
Var. Type: Mand.
Description: Member wishes to consult the offers already defined in the application.
Actors: Member **Dependency**: None.
Preconditions: Member is signed in.
Postconditions: Member has consulted/updated offers.
Main scenario:

1. Member requests the application to browse offers,

2. The application shows the list offers defined in the application,

3. Member selects one offer,

4. The application shows offer details.

Alternatives to the main scenario: 4a. If the offer belongs to the user (this alternative has no effect on the main use case postcondition):

V1 (The application offers the member to update offer details),

V2 (Member selects to update offer details),

V3 (Application shows a filled form with current offer information),

V4 (Member changes details and validates)

4b. If the offer does not belong to the user, the application offers the possibility to make a proposal (see "UC05: Make a Proposal").
Non-Functional: N/A.
Variation points description: V1: Type: Opt, Concerns: Behavior, values=The application offers the member to update offer details
V2: Type: Opt, Concerns: Behavior. If UC04.V1 = not `null` then values=Member selects to update offer details else values=`null`
V3: Type: Opt, Concerns: Behavior. If UC04.V2 = not `null` then values=Application shows a filled form with current offer information else values=`null`
V4: Type: Opt, Concerns: Behavior. If UC04.V3 = not `null` then values=Member changes details and validates else values=null.

ID: `UC05`
Use case name: `Make a Proposal`
Var. Type: Mand.
Description: Member responds to an offer by making a proposal.
Actors: Member **Dependency**: Includes `UC04`.
Preconditions: Member is signed in. At least one offer exists in the system. Member is consulting the details of an offer.
Postconditions: A new proposal has been made. If this proposal is the first, a new deal has been created.
Main scenario:

1. Member selects to make a new proposal,

2. The application shows a new proposal form,

3. Member defines proposal details (belongings) and validates,

4. The new proposal is stored in the application,

5. If the proposal is the first, a new deal is created.

Alternatives of the main scenario: None.
Non-Functional: N/A.
Variation points description: None.

ID: `UC06`
Use case name: `Validate a Deal`
Var. Type: Mand.
Description: A member wishes to accept a proposal that has been made regarding one of its offers.
Actors: Member **Dependency**: None.
Preconditions: Member is signed in. He owns at least one offer.
Postconditions: Deal has been validated.
Main scenario:

1. The Member is browsing the list of current deals concerning the offers he owns,
2. Member selects a particular deal,
3. The application is showing him deal details and full descriptions proposals that have been made for this offer,
4. Member decides to validate one,
5. Application informs the successful bidder as well as the other participants.

Alternatives of the main scenario: None.
Non-Functional: N/A.
Variation points description: None.

6.2 Architectural Framework Analysis Layer

As mentioned in Chapter 4, the analysis layer of the architectural framework refines REQET-based descriptions. We will illustrate in this section how this refinement is made and how variability is achieved via state varaibles.

6.2.1 Domain Model

Figure 6.1 depicts the domain diagram for the ***LuxDeal*** product line.

The data dictionary for the ***LuxDeal*** product line is shown on Table 6.2.

Name	Kind	Use Cases	Description
`Category`	Concept	UC02	Category allows the classification of Items.
`CreateItemForm`	Signal	UC02	Form allowing the member to enter information regarding a particular item.
`CreateOfferForm`	Signal	UC03	Form allowing the member to enter information regarding a particular offer.
`Deal`	Concept	UC05	Deals can be public or private (via `visibility` property) and their `status` indicate if their are still open or being closed to proposals.
`DealDetails`	Signal	UC05	This signal is intended to be processed by the user interface to show the details of a deal to the member.
`DealsDisplay`	Signal	UC05	This signal is intended to be processed by the user interface to depict all the deals regarding the offer he owns.
`Item`	Concept	UC02	Subject of an exchange/sell/buy in ***LuxDeal*** applications. Items have title and description, an estimated value, a reserved price (price under which a seller will not accept to deal). Items can represents wishes and belonging. Within an offer they can be validated or not when the seller and the bidder deal.
`ItemType`	Concept	UC02	Enumeration used to specify if the item is used as a wish or as a belonging.

Name	Kind	Use Cases	Description
`MessageDisplay`	Signal	UC02,UC05	This signal witnesses a general kind of message sent to a particular actor transmitted through various ways (direct displaying, e-mail, SMS etc.).
`NewProposalForm`	Signal	UC04	From allowing members to enter information regarding the creation of a proposal.
`Offer`	Concept	UC03,UC04	An offer is comprised of wishes, belongings and is uniquely related to a deal.
`OfferDetailsDisplay`	Signal	UC04	Signal representing the display of the details of an offer to the member.
`OffersDisplay`	Signal	UC04	Signal representing the display of the list of offers valid in the ***LuxDeal*** product.
`Proposal`	Concept	UC04	Proposals are representing the response of bidders to an offer within a `Deal`.
`SignInForm`	Signal	UC01	Form allowing a member of a ***LuxDeal*** product to enter his credentials.
`UpdateOfferDetailsForm`	Signal	UC04	Form allowing a member to update offer details.
`UserAccount`	Concept	UC01-UC05	This concept is used to store all the information regarding a member in the ***LuxDeal*** products. Additionally, state variable `isUserSignedIn` tells whether is signed in or not.
`Visibility`	Concept	UC05	Enumeration used to state if a deal is public or private.
`WelcomeDisplay`	Signal	UC01	This signal corresponds to the display of a welcome page when the member has successfully signed in the ***LuxDeal*** product.

Tab. 6.2: Domain Data Dictionary for ***LuxDeal***

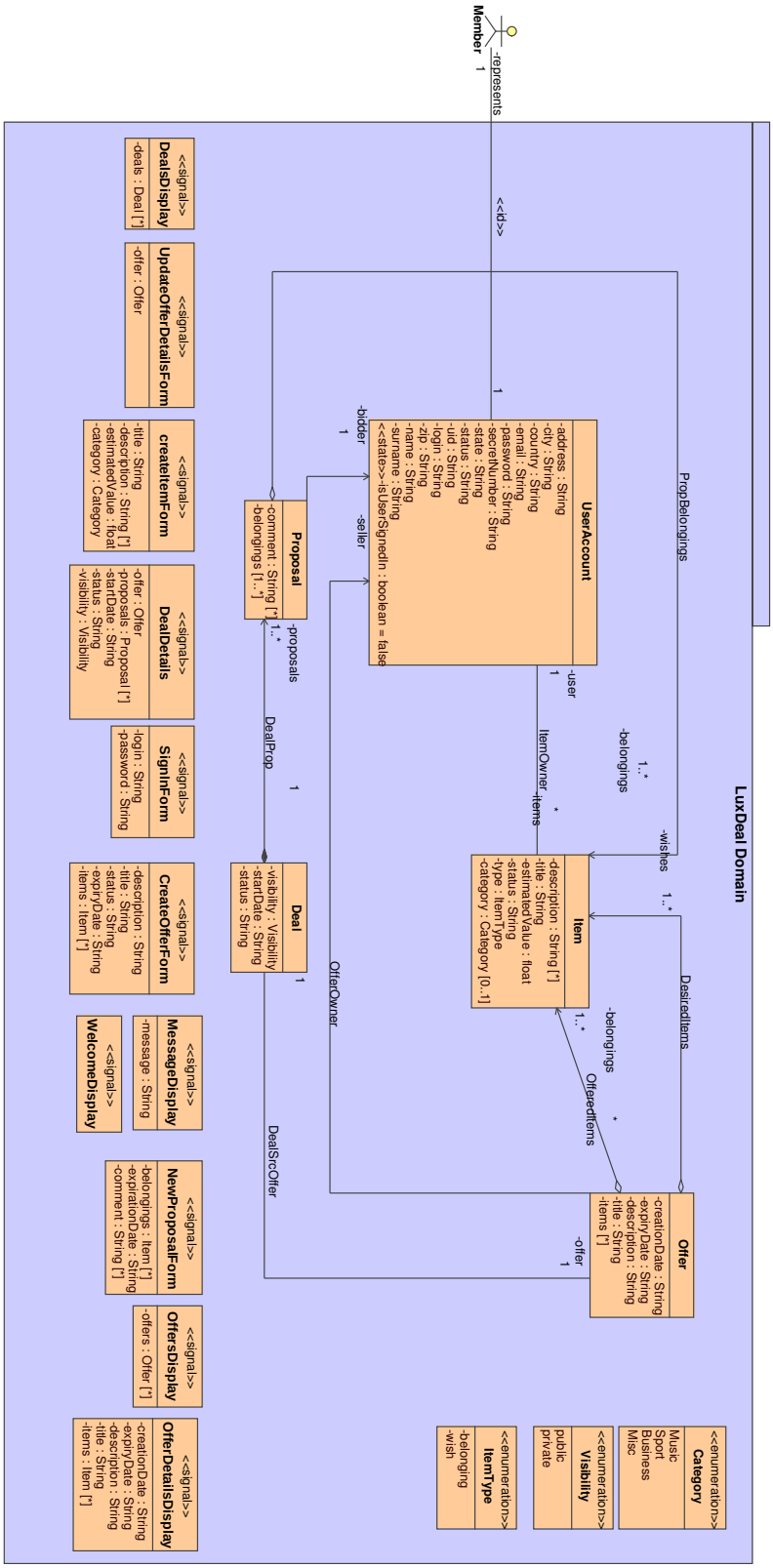

Fig. 6.1: ***LuxDeal*** Domain Diagram

6.2.2 Use Case Model

Figure 6.2 summarizes all the use cases defined for the analysis layer.

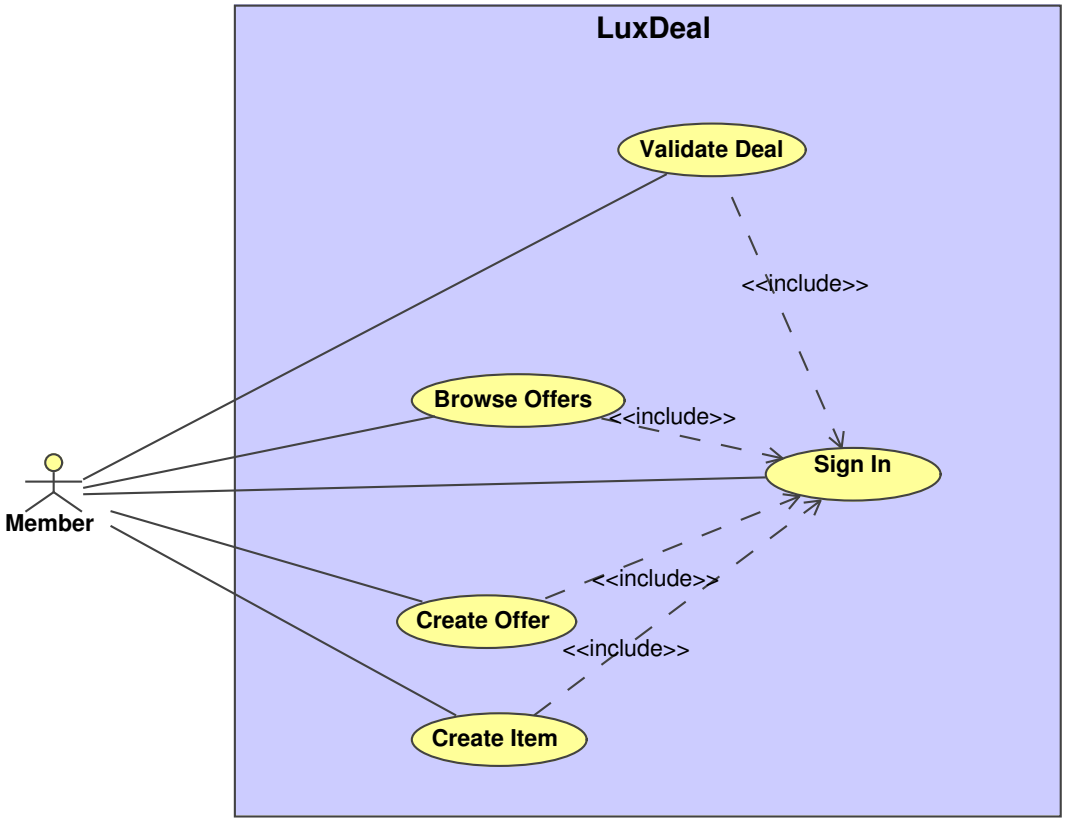

Fig. 6.2: The ***LuxDeal*** Use Case Diagram

Use Case: `UC01: Sign In`

CHARACTERISTIC INFORMATION
Goal in Context: A member signs in the application in order to access to the application main's services.
Scope: `LuxDeal`
Preconditions: Member is not signed in:
pre: `userAcc.isUserSignedIn=false`
Success End Condition: Member is signed in:
post: `userAcc.isUserSignedIn=true`
Failed End Condition: Member is not signed in:
post: `userAcc.isUserSignedIn=false`
Primary Actor: `Member`
Secondary Actors: None.
Trigger: User requests to sign in;
post: `self^signInRequest()`

MAIN SUCCESS SCENARIO

1. Member is presented a sign in form:
   post: `sender^SignInForm(login,password)`
2. Member enters his credentials in the form,
3. Member authenticates himself to the system;
   post: `self^authenticate(login,password)`
4. If member's credentials are correct, member is displayed a welcome screen:
   post: `Let m:OCLMessage = SignIn^^authenticate(login,password)->first()` **in**
   `m.hasReturned() and (m.result()=true) implies`
   `sender^WelcomeDisplay and userAcc.isUserSignedIn=true`

ALTERNATIVES
4a. if user credentials are incorrect and the session have not expired the user is invited to retry:

- Member is informed that his credentials are incorrect:
  pre: `Let m:OCLMessage = self^^authenticate(login,password)->first()` **in**
  `m.hasReturned() and m.result()=false`
  post: `sender^SignInDisplay(login,password)`
- user retries:
  post: `self^authenticate(login,password)`
- the main sequence continues then normally starting from step 4

Figure 6.3 shows the services and objects managed by this use case.

Use Case: `UC02: Create Item`

CHARACTERISTIC INFORMATION
Goal in Context: A member wishes to create an item.
Scope: `LuxDeal`.
Primary Actor: `Member`.
Preconditions: Member is signed in:
pre: `userAcc.isUserSignedIn=true`
Success End Condition: a new item has been created:
post: `LuxDeal.Item.allInstances()->exists(i|i.user=userAcc and i.oclIsNew())`
Failed End Condition: No new item has been created.
post: `LuxDeal.Item.allInstances() = LuxDeal.Item.allInstances()@pre`
Trigger: Member requests to create a new item;
post: `self^createItemRequest()`

MAIN SUCCESS SCENARIO

1. `LuxDeal` sends a create item form for display:
   post: `sender^createItemDisplay(description,title,estimatedValue)`
2. Member fills in the form and creates the new item:
   post: `self^createItem(description,title,estimatedValue)`

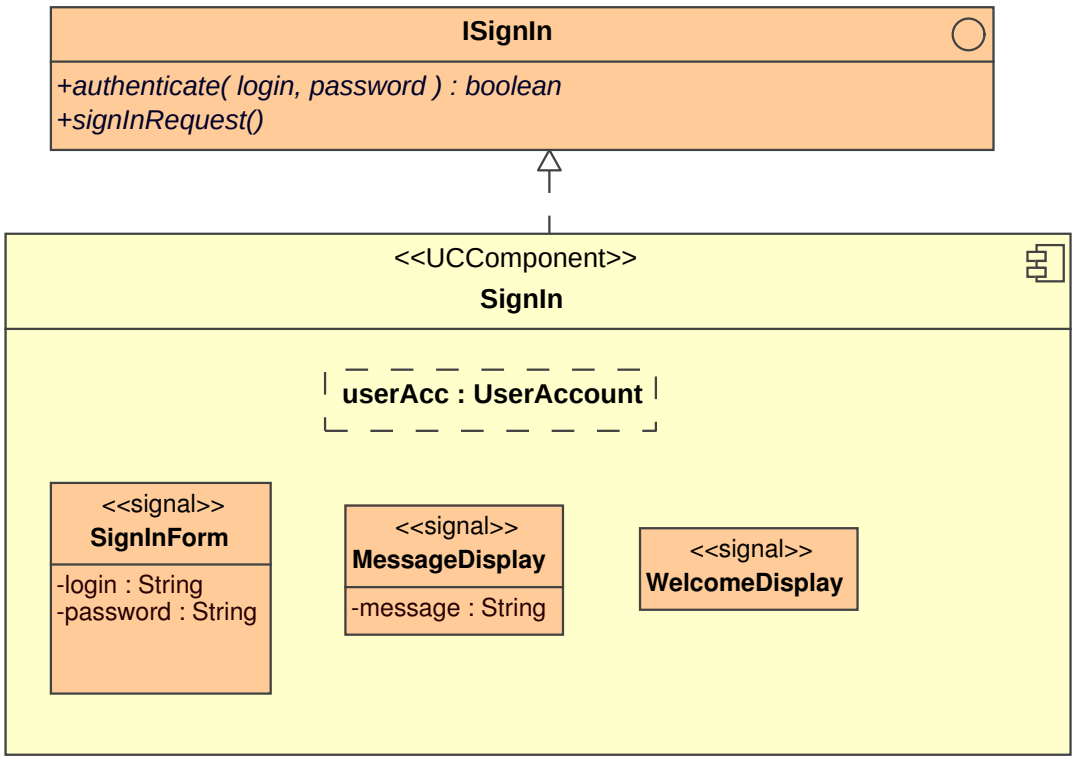

Fig. 6.3: Use Case Component SignIn

ALTERNATIVES
None.

Figure 6.4 shows the services and objects managed by this use case. By default, the ***LuxDeal*** architectural framework analysis layer provides the possibility to store category information for items. As we have seen in Section 6.1, category is an optional feature of the ***LuxDeal*** SPL; indeed, product engineers will be free to keep or remove it while instantiating the architectural framework.

Use Case: `UC03: Create an Offer`

CHARACTERISTIC INFORMATION
Goal in Context: A member wishes to create an offer.
Scope: `LuxDeal`.
Primary Actor: `Member`.
Preconditions: Member is signed in:
pre: `userAcc.isUserSignedIn=true`
Success End Condition: A new offer has been created:
post: `LuxDeal.Offer.allInstances()->exists(o|o.user=userAcc and o.oclIsNew())`
Failed End Condition: No new item has been created.
post: `LuxDeal.Offer.allInstances()=LuxDeal.Offer.allInstances()@pre`
Trigger: Member requests to create a new offer
post: `self^createOfferRequest()`

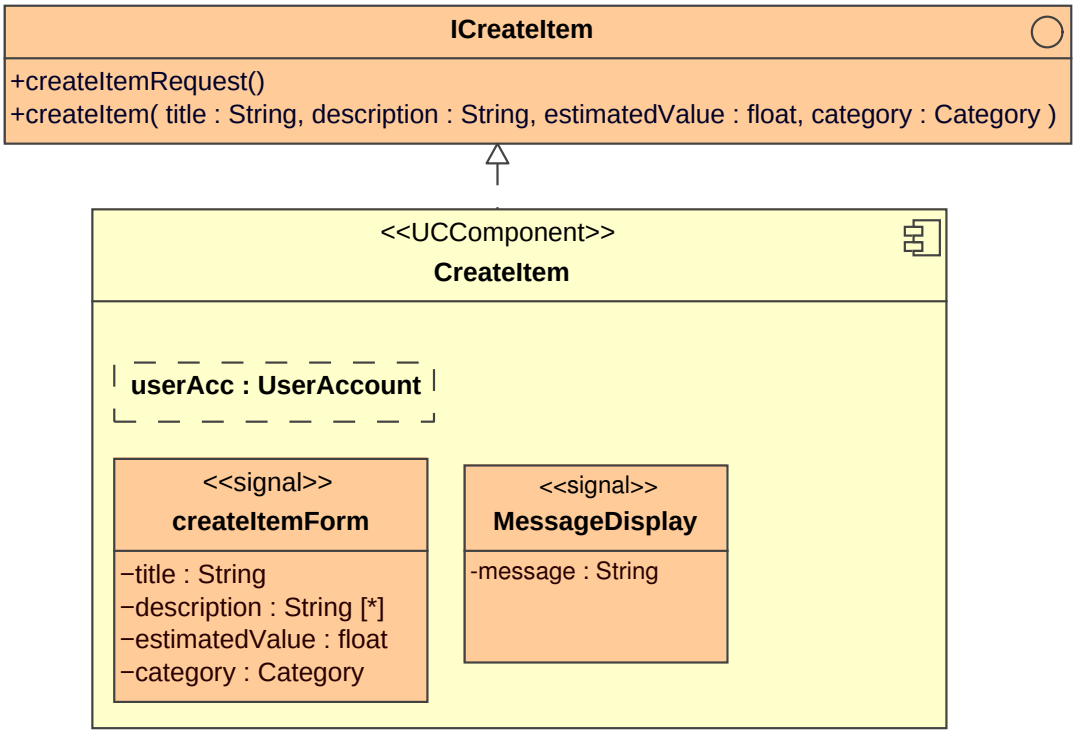

Fig. 6.4: Use Case Component CreateItem

MAIN SUCCESS SCENARIO

1. Application displays an offer form to the member:
   post: `sender^CreateOfferDisplay(description,title,expiryDate,wishes,belongings)`
2. Member enters offer title, description, expiryDate and selects wishes and belongings,
3. Member validates his offer:
   post: `self^createOffer(description,title,creationDate,expiryDate,wishes,belongings)`

ALTERNATIVES
None.

Figure 6.5 presents the use case component associated to “create offer”.

Use Case: `UC04: Browse Offers`

CHARACTERISTIC INFORMATION
Goal in Context: A member wishes to browse offers.
Scope: `LuxDeal`.
Primary Actor: `Member`.
Preconditions: Member is signed in:

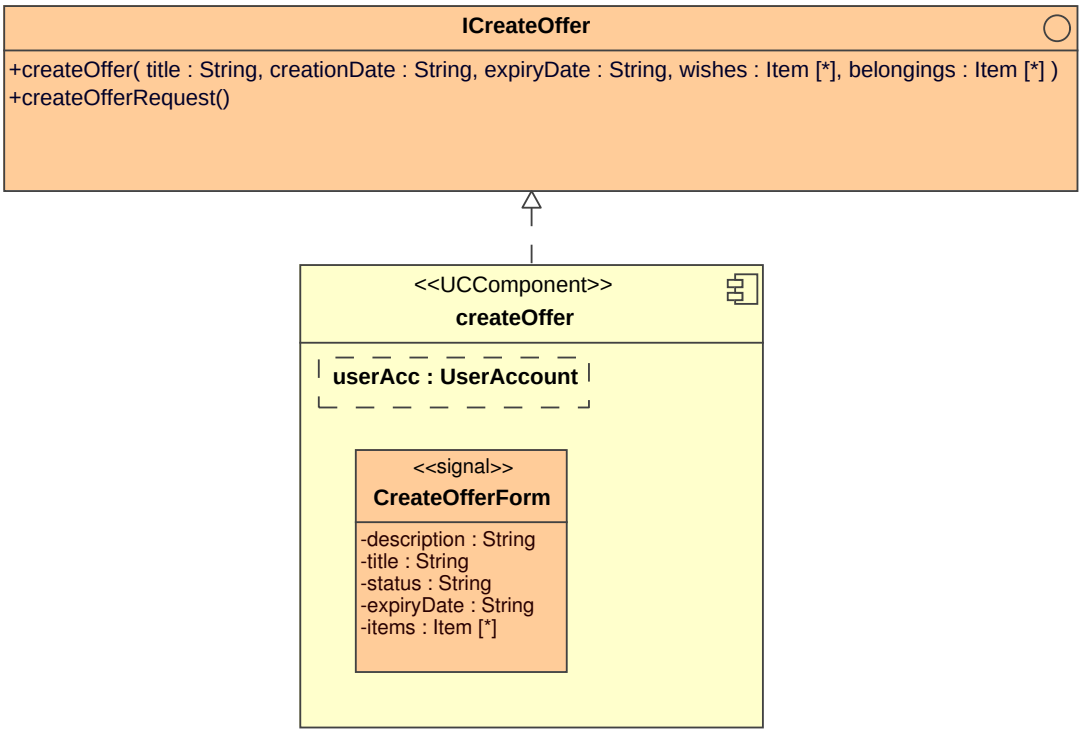

Fig. 6.5: Use Case Component CreateItem

pre: `userAcc.isUserSignedIn=true`
Success End Condition: None.
Failed End Condition: None.
Trigger: Member requests to browse available offers:
post: `self^browseOffersRequest()`

MAIN SUCCESS SCENARIO

1. The application displays the list of available offers:
   post: `sender^OffersDisplay(offers)`
2. Member selects one particular offer and requests to see its details:
   post: `self^seeOfferDetails(offer)`
3. The application displays the details of one offer:
   post: `sender^OfferDetailsDiplay(offer)`

ALTERNATIVES
4a. If the offer belongs to the member, the application enables offer update:

- Member requests to update the offer:
  pre: `BrowseOffersControl.isOfferOwned = true`
  post: `self^updateOfferRequest(offer)`

- The application displays an editable screen comprised of offer information:
  post: `sender^UpdateOfferDetails(offer)`
- Member changes details and validates:
  post: `self^updateOffer(offer)`

4b. If the offer does not belong to the user, the application offers the possibility to make a proposal:

- Member selects to make a new proposal:
  pre: `BrowseOffersControl.isOfferOwned = false`
  post: `self^newProposalRequest()`
- The application shows a new proposal form:
  post: `sender^NewProposalForm(belongings,epxirationDate,comment)`
- Member defines proposal details (belongings) and validates:
  post: `self^newProposal(belongings,epxirationDate,comment)`
- If the proposal is the first, a new deal is created:
  post: `offer.deal->isEmpty()@pre implies offer.deal->first().oclIsNew()`

Figure 6.6 presents the use case component associated to "Browse Offers". Note that we have merged in the analysis layer UC04 and UC05 defined in the REQET-based description. This is one of the two ways to handle use case inclusion at the analysis level; the other is to define a state variable in one of the concepts of the domain whose value is related to the output of the use case to be included. For instance, `isUserSignedIn` testifies that the member is authenticated by the system as he completed the "Sign In" use case. Choosing one of these techniques is mainly a matter of convenience.
In the same vein, there are two ways to model information returned by an operation in terms of signals. Either we detail all the properties that have to be viewed/filled by the actor of the use case (see `NewProposalForm`) or we pass the unique concept as a property of the signal (see `UpdateOfferDetails`).

Use Case: `UC05: Validate a Deal`

CHARACTERISTIC INFORMATION
Goal in Context: A member
Scope: `LuxDeal`.
Primary Actor: `Member`.
Preconditions: Member is signed in:
pre: `userAcc.isUserSignedIn=true`
Success End Condition: One deal has been validated:
`deal.status = 'validated'`
Failed End Condition: No deal has been validated:
`not deal.status = 'validated'`
Trigger: The Member is requesting to see the list of deals regarding the offer he owns:
post: `self^seeCurrentDeals()`

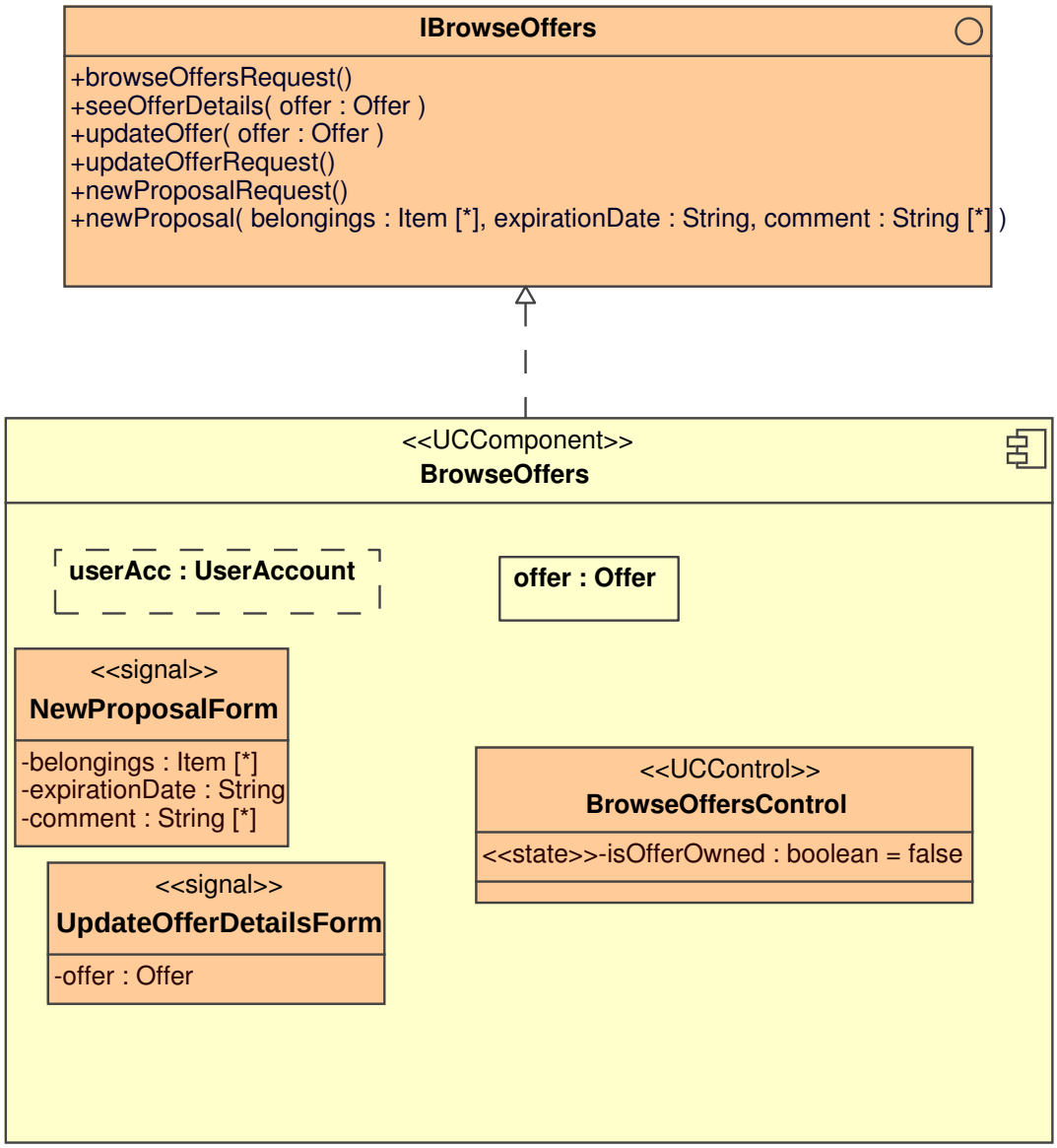

Fig. 6.6: Use Case Component BrowseOffers

MAIN SUCCESS SCENARIO

1. Application displays the list of current deals:
   post: `sender^DealsDisplay(deals)`
2. Member selects a particular deal:
   post: `self^seeDealDetails(deal)`
3. The application is showing him deal details and full descriptions proposals that have been made for this offer:
   post: `sender^DealDetails(offer,startDate,status,visibility,proposals)`
4. Member decides to validate one proposal:

post: `self^acceptProposal(proposal)`

5. Application informs the successful bidder as well as the other participants:
   post: `acceptedProposal.bidder.represents^MessageDisplay('Your proposal has been accepted')`
   post: `deal.proposals.bidder->excludes(acceptedProposal.bidder)->forAll(b| b.represents^MessageDisplay('Your proposal has not been accepted'))`

ALTERNATIVES
None.

Figure 6.7 presents the use case component associated to "Validate a Deal" use case. State variable `deal` is initialized when the member selects a deal on to see its details.

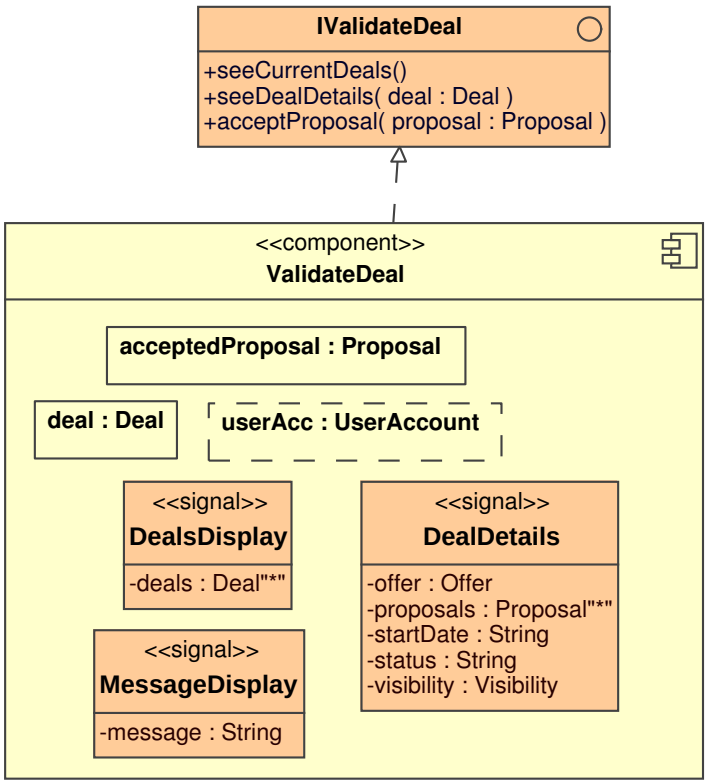

Fig. 6.7: Use Case Component ValidateDeal

6.2.3 Operation Model

It would be too tedious to provide the full operation model here. Rather we summarize all the operations defined at the analysis level in Table 6.3 (serving as a data dictionary for the operation model) and give the full description of some of them in the following.

Name	Related Use Case(s)	Description

Name	Related Use Case(s)	Description
`authenticate`	UC01	Authenticates a member in the system.
`signInRequest`	UC01	Represents a member's request to sign in in a particular ***LuxDeal*** product. This operation results in the sending of `SignInDisplay`.
`createItemRequest`	UC02	Represents a member's request to create an item. This call result in the display of the `CreateItemForm`.
`createItem`	UC02	This operation creates an item according to the values passed as parameter by the member of a ***LuxDeal*** application.
`createOfferRequest`	UC03	This operation represents a request from a member to create an offer.
`createOffer`	UC03	This operation actually creates an offer according to its parameter values.
`browseOffersRequest`	UC04	This operation represents the request of a member willing to see the offer available in the ***LuxDeal*** product he is using.
`seeOfferDetails`	UC04	This operation shows the details of an offer.
`udpdateOfferRequest`	UC04	This operation represents the request of a member willing to update an offer.
`updateOffer`	UC04	This operation actually updates an offer according to the details provided by a member of the ***LuxDeal*** product.
`newProposalRequest`	UC04	This operation represents the request of a member willing to define a new proposal.
`newProposal`	UC04	This operation perform the actual creation of a proposal in the context of a deal (or creates it if it does not exists yet) based on its parameters.
`seeCurrentDeals`	UC05	This operation represents the request of a member willing to see deals in progress.

Name	Related Use Case(s)	Description
`seeDealDetails`	UC05	This operation results in the display of the details of a deal selected by a member.
`acceptProposal`	UC05	This operation validates a proposal an therefore concludes the deal.

Tab. 6.3: ***LuxDeal*** Data Dictionary

Operation Name: `signInRequest`

Related Use Case: UC01

Description: Represents a member's request to sign in a particular ***LuxDeal*** product

Parameters: None.

Sends:

1. `SignInForm` **to** `Member`

Preconditions: Member is not signed in:
pre: `userAcc.isUserSignedIn = false`

Postconditions: Sender has been sent a "sign in" form:
post: `sender^SignInForm(login,password)`

Operation Name: `createItem`

Related Use Case: UC02

Description: This operation creates an item according to its parameters.

Parameters:

- `title` **TypeOf** `String`,
- `description` **TypeOf** `String`,
- `estimatedValue` **TypeOf** `Float`
- `category` **TypeOf** `Category`

Sends: None.

Preconditions: Member is signed in:
pre: `userAcc.isUserSignedIn = true`
Member has received an `CreateItemForm`:
`pre: userAcc.represents^CreateOfferForm`

Postconditions: A new item has been created (the collection of items is assumed to be order by creation date):
post: `userAcc.items->last().oclIsNew()`
The new item has been created conforming the parameters of the operations:
post:
```
let it:Item = userAcc.items->last() in
it.title = title and it.description = description
it.estimatedValue =  estimatedValue
and it.category = category
```

Operation Name: `newProposal`

Related Use Case: UC04

Description: This operation perform the actual creation of a proposal in the context of a deal (or creates it if it does not exists yet) based on its parameters.

Parameters:

- `belongings` **TypeOf** `String [*]`
- `expirationDate` **TypeOf** `String`
- `comment` **TypeOf** `String [*]`

Sends: None.

Preconditions: Member is signed in:
pre: `userAcc.isUserSignedIn = true`

Postconditions: A new proposal has been created:
post:
```
if offer.deal->isEmpty()@pre then offer.deal->first().oclIsNew()
else true endif
```
post: `and offer.deal.proposals->size() = offer.deal.proposals->size()@pre + 1`

6.2.4 Traceability between *LuxDeal* Elicitation and Analysis

In the following we illustrate our traceability document format between the REQET description of the ***LuxDeal*** product line and its FIDJI analysis model. Concerning domain concepts (DOMET and domain model), tuples are the following:
`<Deal,{Deal}>`
`<Item,{Item,ItemType}>`
`<Member,{UserAccount}>`
`<Proposal,{Proposal}>`
`<Category,{Category}>`
Use case tuples are the following:
`<UC01,{UC01}>`
`<UC02,{UC02}>`
`<UC03,{UC03}>`

```
<UC04,{UC04}>
<UC05,{UC04}>
<UC06,{UC05}>
```

6.2.5 Instantiation Constraints

As we have mentioned in Chapter 4, Section 4.3 we use textual traceability illustrated above to define analysis instantiation constraints. First, optional concept `Category` and `UpdateDetailsOfferForm` excluded (there is no need to define a constraint in this case), all concepts have to be kept while deriving the domain model of a product:

```
context ANAM::DOM inv:
not removeConcept(self,'CreateItem') and not removeConcept(self,'CreateOfferForm')
... and not removeConcept(self,'WelcomeDisplay')
```

`removeConcept` is one of the transformation operations that can be used in an instantiation program. Its specification his as follows:

```
removeConcept(src:Package,conceptName:String):boolean
pre: src->select(c|c.oclIsKinfOf(Classifier))->exists(co|co.name=conceptName)
post: not src->select(c|c.oclIsKinfOf(Classifier))->exists(co|co.name=conceptName)
```

One might find cumbersome having to specify explicitly which concepts cannot be removed. We have to do so because in order to validate the instantiation program these constraints are checked on the product model; if one has defined a constraint using `removeConcept()` iterating over the collection of concepts, in the product model the link with the original domain model of the architectural framework would have been lost and the constraint would be satisfied whether a concept was differing or not with respect to the architectural framework analysis layer.
We can observe in the UCET description of ***LuxDeal*** that all the use cases are mandatory. Therefore all use case components have to be present in any of the products obtained via ***LuxDeal*** architectural framework instantiation:

```
context ANAM::UCM inv:
not removeUCComponent(self,'SignIn') and not removeUCComponent(self,'CreateItem')
... and not removeUCComponent(self,'ValidateDeal')
```

We might have defined this constraint by checking the existence of the use case components without using `removeUCComponent` (which is defined in a similar way than `removeConcept`). Actually, the choice of using a transformation operation or not depends on the conciseness of the constraint and ease to write it.
Concerning the operation model, operations `updateOfferRequest` and `updateOffer` are optional since they support the optional variant defined in `UC04` of the UCET. As we have already mentioned optional variants do not deserve particular constraint unless they are dependencies between them which is the case here:

```
context ANAM::UCM::BrowseOffers inv:
let it: Interface = self.provided->first() in
not removeConcept(self,'UpdateDetailsOfferForm') implies
not removeOp(it,'updateOfferRequest')
and not removeOp(it,'updateOffer')
```

Instantiation constraints for the operation model are defined on the interfaces of the use case components since the operation model is not constituted by UML model elements.

6.3 Architectural Framework Design Layer

6.3.1 GAM

The (partial) GAM for the ***LuxDeal*** SPL is depicted Figure 6.8. Ports directly owned by <<root>> component `LuxDeal`, such as `pSignIn`, are supporting the traceability with the analysis models by providing interaction points between analysis interfaces and the components realizing `LuxDeal`. They also act as "delegates" (following the business delegate pattern sketched in Chapter 5) which allow to separate the presentation tier form the business logic.
Note that there is no one-to-one dependency between the use case components defined at the analysis level and the components realizing them at the design level; both `BrowseOffers` and `CreateOffers`. The use of architectural styles (not shown here) may also contribute to break this relationship by adding additional components and indirections. In the following, we will focus on the `ItemManager` component.

6.3.2 ISM

Figure 6.9 presents the internal structure model of the `ItemManager` component. Interface `IItemManager` is the main entry point of this component and is actually realized by class (modeled as an anonymous part) `ItemManager` to handle member requests and item creation. `ItemSelection` part realizes the `ISelectItem` interface which is required by `OfferManager` for offer creation. Finally, `MemberService` is a "helper class" which provides useful methods to access the details of the account of a member and therefore associate items to it. This class indeed delegates its functionality to the `UserAccountManager` component through `IMemberService` interface.

6.3.3 PSM

Figure 6.10 shows the part structure model for `ItemManager` component. `Item` is a refinement at the design level of the eponymous concept at the analysis level. A more detailed design using the J2EE platform would see `Item` refined as an EJB entity bean while `ItemManager`,

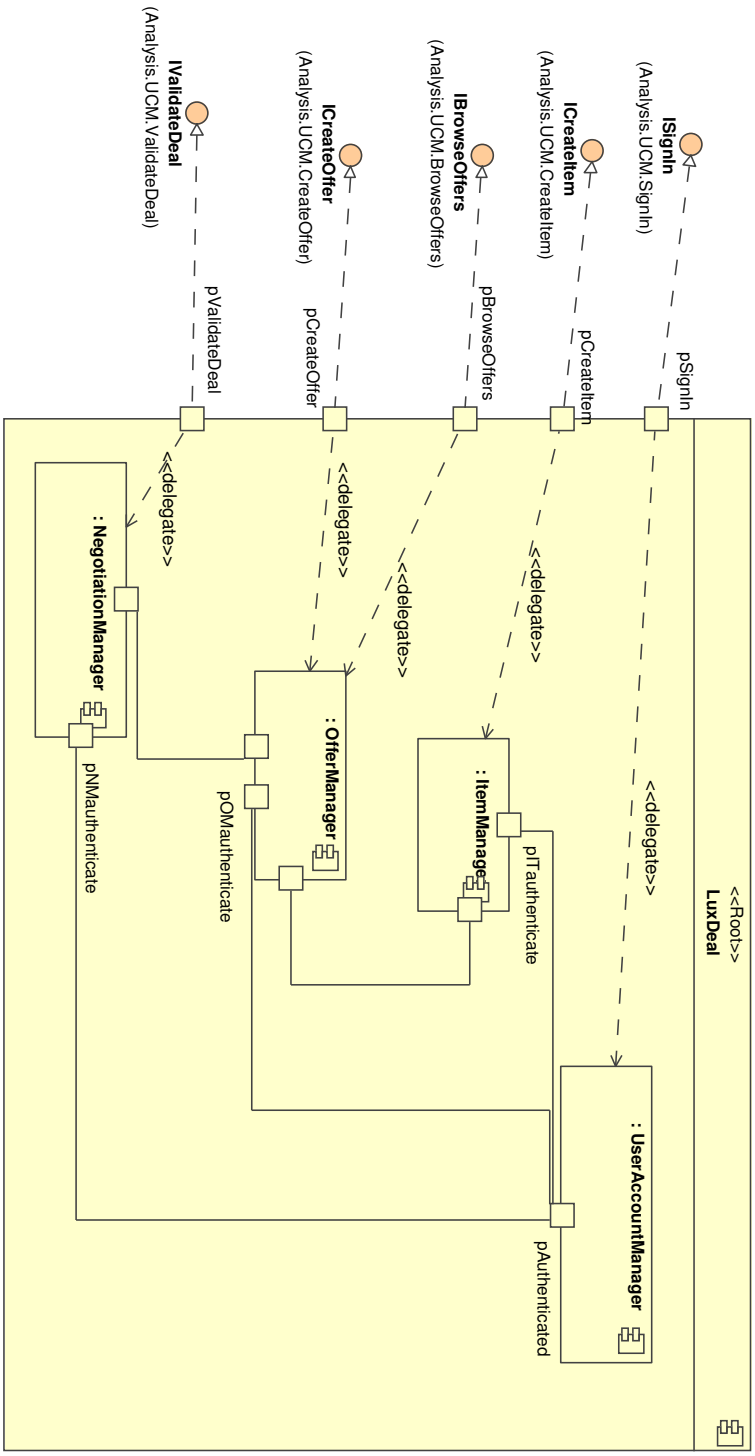

Fig. 6.8: GAM for ***LuxDeal*** SPL

`ItemSelection` and `MemberService` would detailed as session beans.

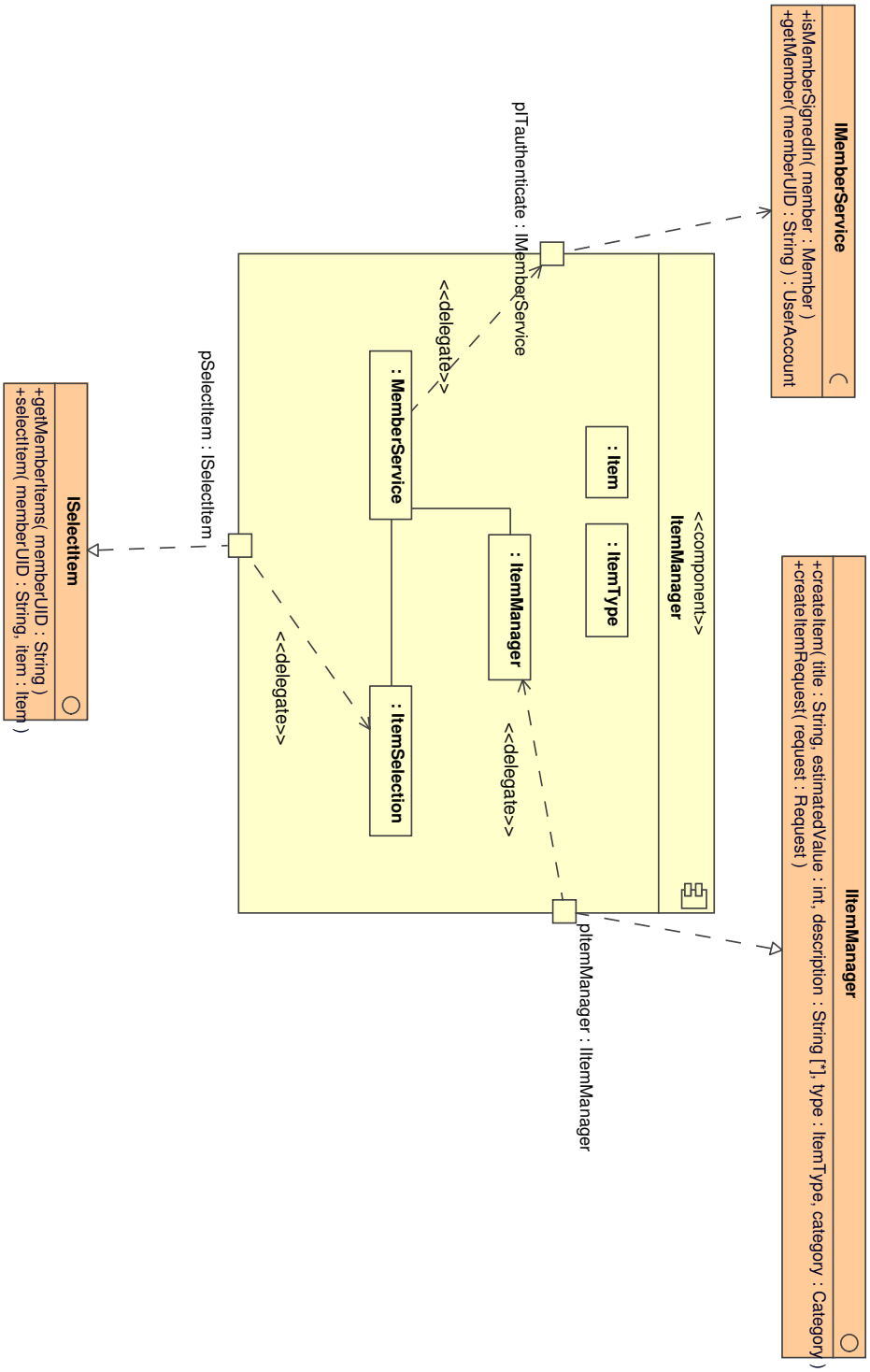

Fig. 6.9: ItemManager ISM

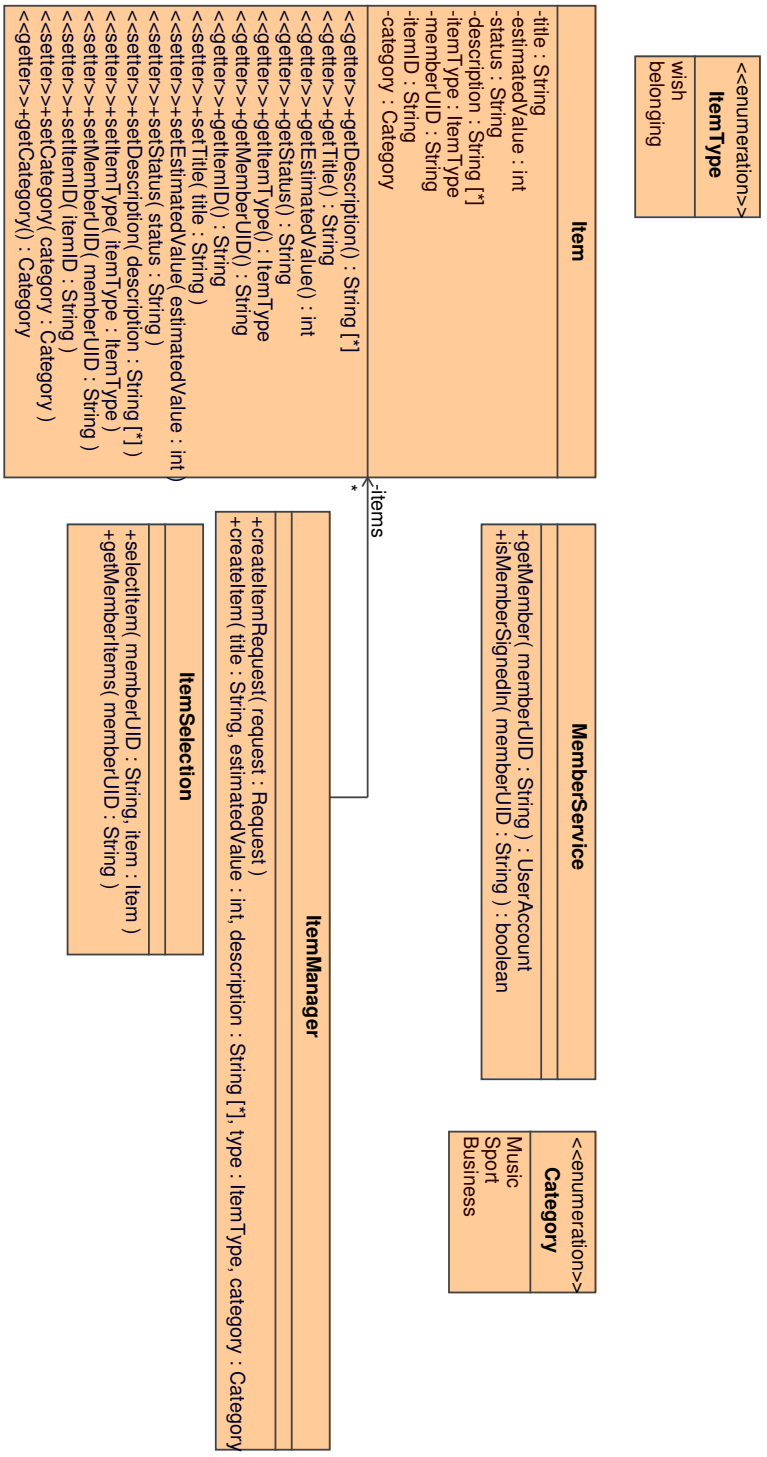

Fig. 6.10: ItemManager PSM

6.3.4 Traceability between Analysis and Design

As mentioned in Chapter 5, Section 5.3, analysis to design traceability is ensured via two mechanisms. The first one is realized via the interfaces of the use case components that have to be provided by the <<root>> component. This mechanism is illustrated on the GAM shown

on Figure 6.8. This approach allows to understand how the `CreateItem` system operation is realized at the desingn level; by observing the GAM we can notice that `ICreateItem` behavior is delegated to port `pItemManager` of `ItemManager`. In the internal structure model (Figure 6.9), it appears that this port itself delegates this behavior to the `ItemManager` class fully described in the PSM (Figure 6.10).
We do not show <<refine>> relationships between analysis concept and design classes graphically, but they can be easily documented in any UML 2.1.1 compliant CASE tool. Such relationships are defined between classes `Item` and `Category` of the PSM and their eponymous analysis concepts.

6.3.5 Design Instantiation Constraints

The determination of the design instantiation constraints is based on analysis instantiation constraints and traceability information illustrated above. Traceability allows to determine which elements of the design layer of the architectural framework are concerned by instantiation constraints. For example, by remarking the <<refine>> relationship between `Item` elements the following constraint is defined:

```
context Design:PSM::ItemManager inv:
not removeClass(self,'Item')
```

Transformation operation `removeClass` is defined in a very similar way as `removeConcept` (see above). However not all design instantiation constraints are directly derivable from the analysis constraints. There exist design level elements that are supporting non-functional services such as a load-balancing mechanism or a relational database. These elements may have to be kept in all architectural framework instantiations so that products can be viable. It is the role of architectural framework designers to define such constraints with respect to technical issues related to architectural framework design and implementation.

6.4 Product Derivation

In this section, we illustrate how a particular product can be obtained via ***LuxDeal*** architectural framework instantiation. We consider a customer who is interested in a product that support all the features supported by ***LuxDeal*** SPL. He makes the following additional requirements:

- He does not want that items be organized in categories,
- He needs a more fine grained way to describe items. Indeed, he needs to distinguish between material items (goods, which usually have a brand name) and services that are provided by people (e.g. a piano lesson or computer trouble-shooting).

6.4.1 Analysis

Instantiation Program

The first step is to create a new UML package for the product `proLux`, and to copy domain and use case models of the architectural framework in the product:

```
createProduct('proLux');
copyModel(LuxDeal::DOM,proLux::DOM);
copyModel(LuxDeal::DOM,proLux::UCM);
```

`createProduct` creates a package whose name is given as parameter and whose sub-packages organization reflects FIDJI models. `copyModel` specification is given below:

```
copyModel(src:Model,tgt:Model):boolean
pre: none.
post: src.ownedElement->forAll(e | tgt.ownedElement->includes(e))
```

We have then to update the domain model of the product by removing the `Category` concept which is useless in this product and add two new concepts, `Good` and `Service`:

```
deleteConcept(proLux::Domain,'Category');
newConcept(proLux::Domain,'Good');
newConcept(proLux::Domain,'Service');
addPropToConcept(proLux::Domain,'Good','brand',String);
addPropToConcept(proLux::Domain,'Service','provider',String);
```

`newConcept` creates a concept whose name is given by the second parameter in the model given by the first. The `deleteConcept` transformation operation has the same parameters and delete the concept identifies by the second parameter. Finally `addPropToConcept` specification is given below:

```
addPropToConcept(model:Model,conceptName:String,propName:String,type:Type):boolean
pre: not model->select(co|co.oclIsTypeOf(class) and co.name=conceptName)->first()
.attribute->exists(att|att.name=propName and att.datatype = type)
post: model->select(co|co.oclIsTypeOf(class) and co.name=conceptName)->first()
.attribute->exists(att|att.name=propName and att.datatype = type)
```

These concepts are indeed specializations of an `Item`:
`inherit(proLux::Domain,Item,Good);`
`inherit(proLux::Domain,Item,Service);`
`inherit` transformation operation creates a generalization relationship so that the second parameter inherits from the first.

One also has to modify `createItemForm` so that members can enter good or service information:
`addPropToConcept(proLux::Domain,'createItemForm','provider',String);`
`addPropToConcept(proLux::Domain,'createItemForm','brand',String);`
`addOpParameter(proLux::Domain,'createItemForm','isService',Boolean);`
The last parameter, `isService`, allows a member to state if the item to be created is a service or not. Figure 6.11 shows the resulting domain diagram. New/updated elements are shown in light yellow.

The use case model also has to be updated to take into account changes in the way items are created:
`addOpParameter(proLux::UCM::ICreateItem,'createItem','brand',String);`
`addOpParameter(proLux::UCM::ICreateItem,'createItem','provider',String);`
`addOpParameter(proLux::UCM::ICreateItem,'createItem','isService',Boolean);`
Figure 6.12 depicts the updated use case component.

Finally, the specification of `createItem` has to be updated in the operation model:

Operation Name: `createItem`

Related Use Case: UC02

Description: This operation creates an item according to its parameters.

Parameters:

- `title` **TypeOf** `String`,
- `description` **TypeOf** `String`,
- `estimatedValue` **TypeOf** `Float`,
- `category` **TypeOf** `Category`,
- `brand` **TypeOf** `String`,
- `provider` **TypeOf** `String`,
- `isService` **TypeOf** `boolean`,

Sends: None.

Preconditions: Member is signed in:
pre: `userAcc.isUserSignedIn = true`
Member has received an `CreateItemForm`:
`pre: userAcc.represents^CreateOfferForm`

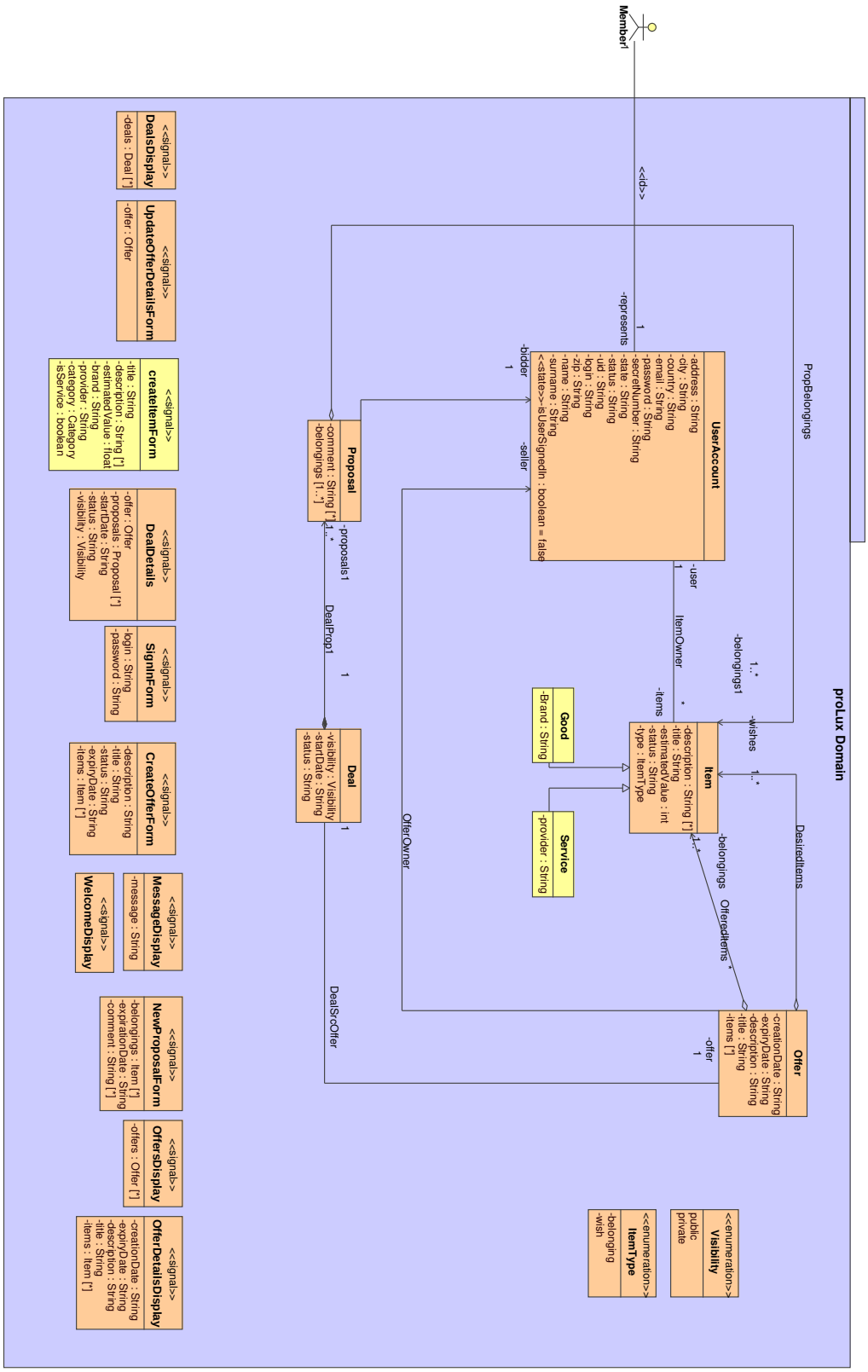

Fig. 6.11: proLux Domain Diagram

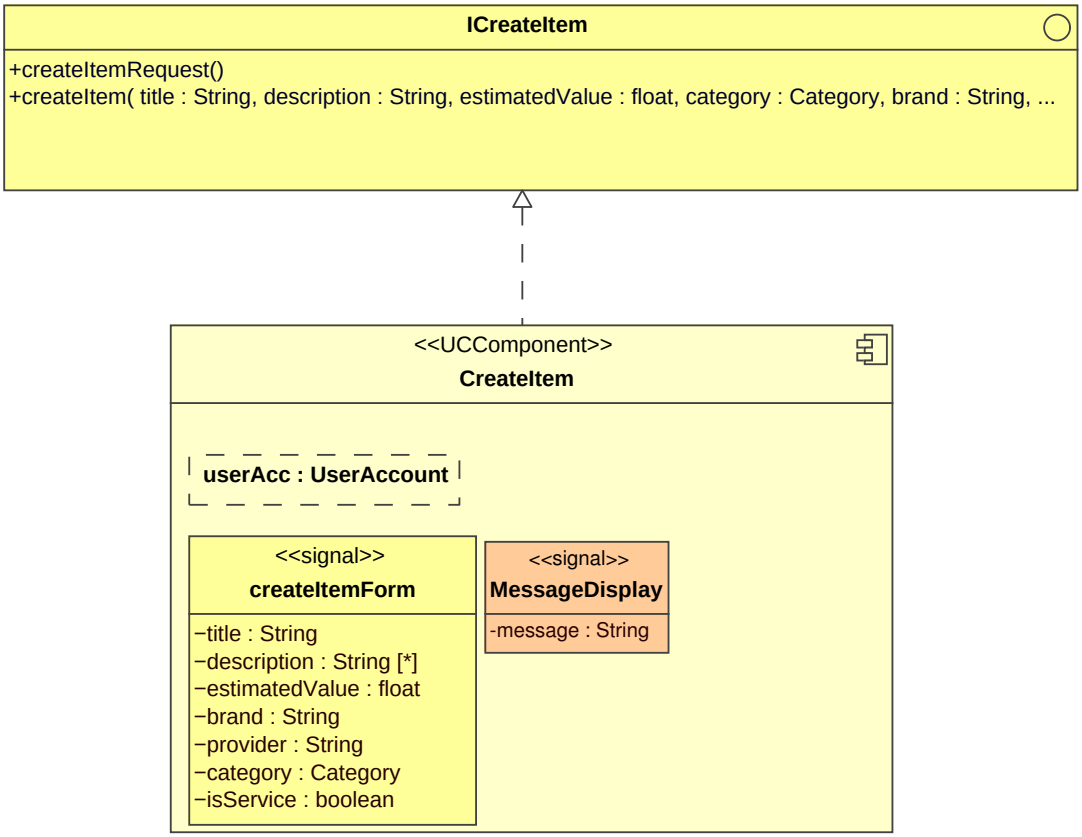

Fig. 6.12: ProLux Create Item Use Case Component

Postconditions: A new item has been created (the collection of items is assumed to be order by creation date):
post: `userAcc.items->last().oclIsNew()`
The new item has been created conforming to the parameters of the operations:
post:
```
let it:Item = userAcc.items->last() in
it.title = title and it.description = description
it.estimatedValue =  estimatedValue
and it.category = category
and if isService = true then
it.oclIsTypeOf(Service) and it.provider = provider
else it.oclIsTypeOf(Good) and it.brand = brand
endif
```

Impact Analysis and Resolution

The above paragraphs showed how the instantiation program can be defined to build the analysis model of `proLux`, in removing the `Category` concept and adding new concepts behavior for items creation. This program fulfills the previously defined instantiation constraints. Therefore, the analysis model can be actually actually built for the product.
However, this model is not conform to the FIDJI profile. The product analyst has not taken all the impact into account in his analysis program; signal `createItemForm` and operation `createItem` still have properties whose type is `Category`. This model violates rule WFR-5 (see Chapter 4) stating that all properties of signals and concepts must be typed. Furthermore, it also violates rule WFR-1 that all concepts and signals manipulated by operations have to be defined in the domain model. .

6.4.2 Design

The design of `proLux` is very similar to its analysis. Therefore we will not detail the architectural framework design instantiation process. The first step is to copy the design layer of the architectural framework in `proLux`. Then, the product designer has to write transformations on the basis of the traceability information between analysis and design: since the <<root>> component provides the interfaces defined in the use case components one can evaluate which elements are concerned by following delegation connectors in the GAM. In `proLux`, it is the `ItemManager` component that is impacted. Similarly to the analysis instantiation program, transformations to be made are dealing with the removal of the `Category` design class (refining the eponymous analysis concept) and the modification of `createItem` method. Additionally, one has to update the Kermeta description of this method.

6.5 Deriving another Product

In this section, we illustrate how we can reuse knowledge gained on the derivation of one ***LuxDeal*** member to derive a new product. We consider another customer who, after having reviewed `proLux` requirements in the software vendor products' portfolio, would like to acquire a similar product. He however introduces the following differences with `proLux` for his product (called `goodLux`):

- **Category management**: This customer needs to have categories to organize the items' catalogue. Additionally he requires that offer browsing can be made according to the categories the items offered belongs to. If an offer's items pertain to several categories, the offer will be listed in all these categories. Finally, if all items are un-categorized, this functionality should be disabled,
- **Goods and services**: This customer is interested in having only goods as a specialization of item. He does not want his website to support services for legal reasons.

As for any product, the instantiation program's first instructions set up the analysis models:

```
createProduct('goodLux');
copyModel(LuxDeal::DOM,goodLux::DOM);
copyModel(LuxDeal::DOM,goodLux::UCM);
```

Since the `Category` concept is already provided by the architectural framework, we do not have to update the domain model to define this concept in `goodLux`. But we will need a special screen to display offers by categories:

```
newSignal(goodLux::Domain,'CatOfferDisplay');
addPropToSignal(goodLux::Domain,'CatOfferDisplay','offers',Offer);
```

We reuse part of the instantiation program for `proLux` to create the concept of "good":

```
newConcept(goodLux::Domain,'Good');
addPropToConcept(goodLux::Domain,'Good','brand',String);
inherit(proLux::Domain,Item,Good);
```

Similarly, part of the instantiation program for `proLux` concerned `ItemForm` modification is reused:

```
addPropToConcept(goodLux::Domain,'createItemForm','brand',String);
```

Figure 6.13 depicts the modified domain model. Note that we are not always aware of these changes while updating the domain model. We can realize the need for a new concept when updating use cases or adding a new operation.

Concerning use cases, there are two groups of transformation operations to write. The first one consists in reusing the `proLux` instantiation program in order to support good creation in the "create Item" use case:

```
addOpParameter(goodLux::UCM::ICreateItem,'createItem','brand',String);
```

The second group is composed of the transformation operations performing changes in "browse offers" to support display of offers sorted by category. First, we create a new operation

```
newOperation(goodLux::UCM::IBrowseOffer,'browseOffersByCategoryRequest');
```

We also need to define a new state variable that will be used to enable/disable category viewing represented by `browseOffersByCategoryRequest` operation. The value of this boolean is based on the fact that there should be at least one item in one offer that pertains to a category. Therefore this is a derived state variable:

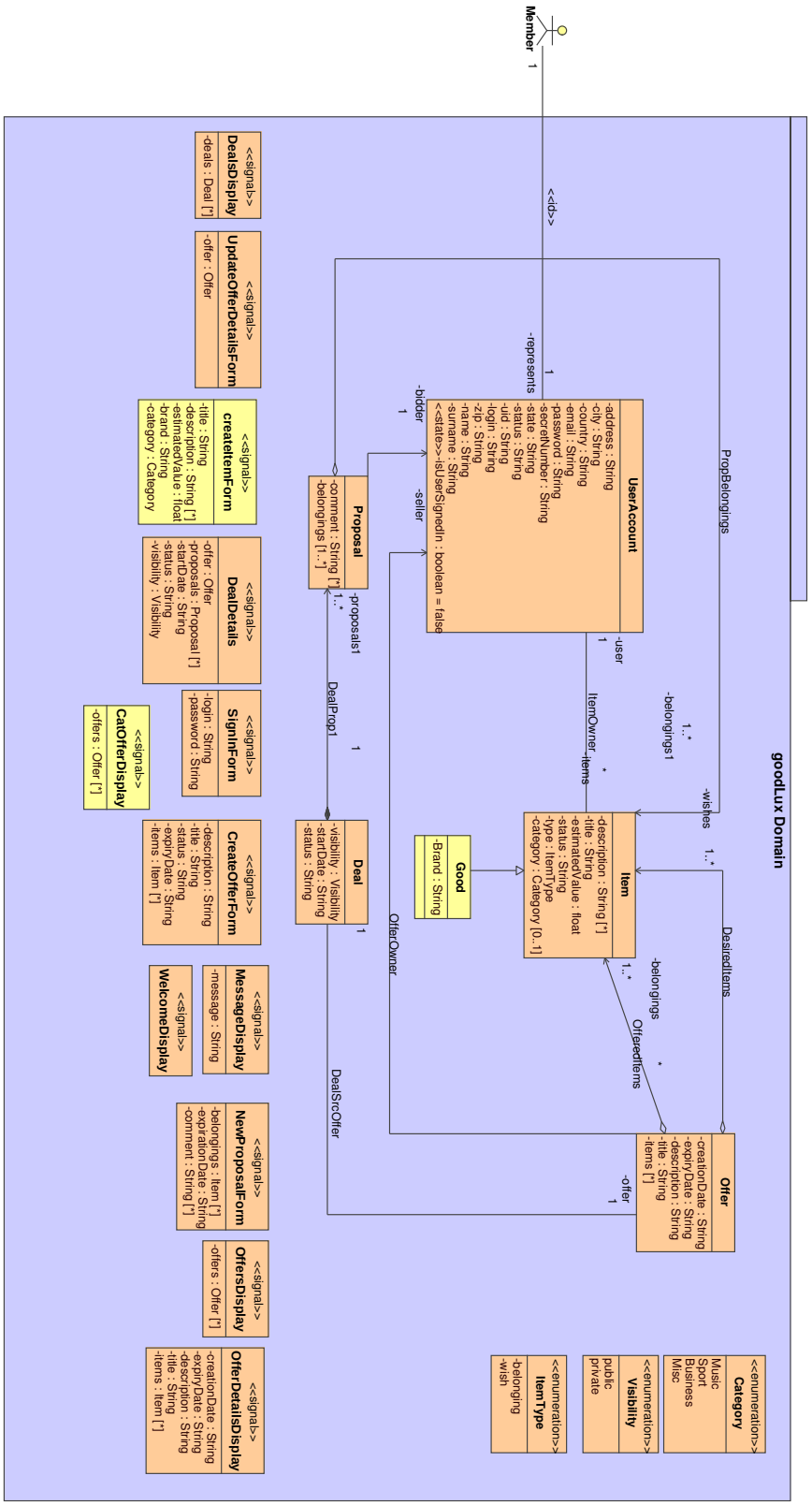

Fig. 6.13: goodLux Domain Diagram

```
Context BrowseOffers::BrowseOffersControl::itemsByCat: boolean
derive: Offer.offeredItems->exists(i|not i.category.isUndefined())
```

The specification of `newStateVarUCC` is a specialization of `addPropToConcept` in which the created property is stereotyped by <<state>>. We can also provide a default value:

```
newStateVarUCC(goodLux::UCM::BrowseOffers::BrowseOffersControl,
'itemsByCat', boolean, false);
```

Figure 6.14 shows the updated version of `BrowseOffers` use case component.

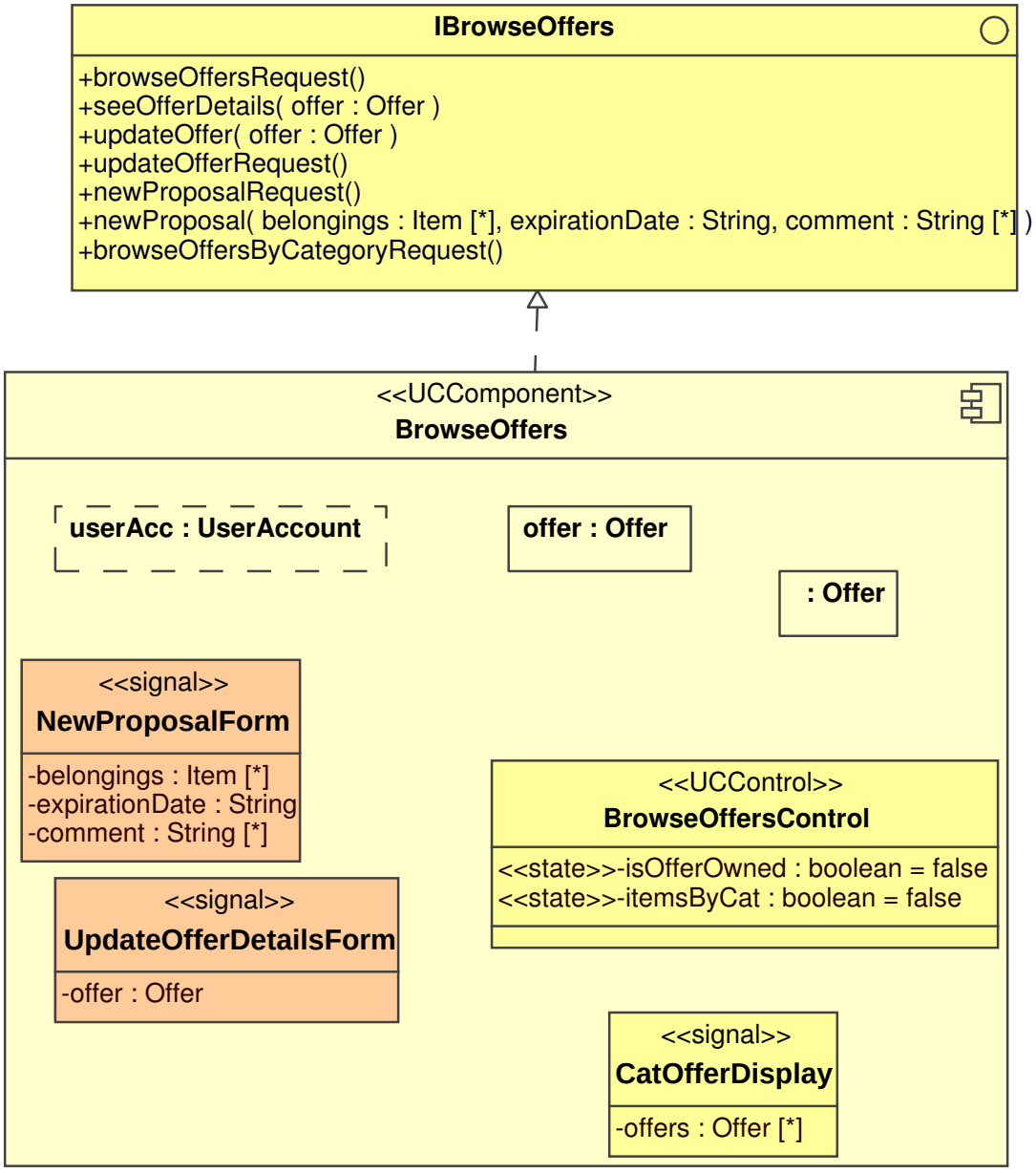

Fig. 6.14: goodLux BrowseOffers Use Case Component

Naturally we also have to update the textual description of the modified use case component. Here follows, the updated textual use case for "Browse Offers" and the specification of

`browseOffersByCategoryRequest`.

Use Case: `UC04: Browse Offers`

CHARACTERISTIC INFORMATION
Goal in Context: A member wishes to browse offers.
Scope: `LuxDeal`.
Primary Actor: `Member`.
Preconditions: Member is signed in:
pre: `userAcc.isUserSignedIn=true`
Success End Condition: None.
Failed End Condition: None.
Trigger: Member requests to browse available offers:
post: `if BrowseOffersControl.itemsByCat = false then self^browseOffersRequest() else self^browseOffersRequest() xor self^browseOffersByCategoryRequest()`

MAIN SUCCESS SCENARIO

1. The application displays the list of available offers:
   post: `sender^OffersDisplay(offers) xor sender^OffersCatDisplay(offers)`
2. Member selects one particular offer and requests to see its details:
   post: `self^seeOfferDetails(offer)`
3. The application displays the details of one offer:
   post: `sender^OfferDetailsDiplay(offer)`

ALTERNATIVES
4a. If the offer belongs to the member, the application enables offer update:

- Member requests to update the offer:
  pre: `BrowseOffersControl.isOfferOwned = true`
  post: `self^updateOfferRequest(offer)`
- The application displays an editable screen comprised of offer information:
  post: `sender^UpdateOfferDetails(offer)`
- Member changes details and validates:
  post: `self^updateOffer(offer)`

4b. If the offer does not belong to the user, the application offers the possibility to make a proposal:

- Member selects to make a new proposal:
  pre: `BrowseOffersControl.isOfferOwned = false`
  post: `self^newProposalRequest()`

- The application shows a new proposal form:
  post: `sender^NewProposalForm(belongings,expirationDate,comment)`
- Member defines proposal details (belongings) and validates:
  post: `self^newProposal(belongings,epxirationDate,comment)`
- If the proposal is the first, a new deal is created:
  post: `offer.deal->isEmpty()@pre implies offer.deal->first().oclIsNew()`

Operation Name: `browseOffersByCategoryRequest()`

Related Use Case: UC04

Description: Represents a member's request to see offer(s) by category

Parameters: None.

Sends:

1. `CatOfferDisplay` **to** `Member`

Preconditions: Member is signed in:
pre: `userAcc.isUserSignedIn = false`
Items must be organized by categories:
`BrowseOffersControl.itemsByCat = true`

Postconditions: Sender has been sent a "sign in" form:
post: `sender^CatOfferDisplay(offers)`

As for `proLux`, impact analysis has to be performed in order to complete the product's analysis models. As this task is very similar in the two products we do not illustrate it here. This section ends the illustration of the FIDJI approach, in the next section we will discuss its validation.

6.6 Towards FIDJI validation

In this section, we give some insights on the validation of the method presented in this thesis. We describe first the details of an experiment carried out during the FIDJI project and then discuss validation criteria and protocol.

6.6.1 Initial Experiment

In chapter 3, we mentioned that the FIDJI research project by implementing a concrete architectural framework and its accompanying modeling and transformation tools contributed to the methodological thinking resulting in the approach defended by this thesis. This concrete implementation allowed us to carry out an experiment (which was actually performed in may 2002) to perform an early validation of the general approach. Though the FIDJI process supported by JAFAR [GRS03a, GS02b] and MEDAL [GRS03b] was different (in particular in terms of flexibility for architectural framework instantiation), we believe that some of the findings of this experiment are applicable to the method described in this dissertation.

The validation issues investigated by this experiment were the following:

- **Assessing the learning curve of the approach**: One needs to get familiar with the architectural framework and its instantiation process in order to develop new product line members,
- **Assessing the suitability of the architectural framework functionality and technology**: One needs to validate that choices made concerning architectural framework functionality and technology are meaningful with respect to the SPL and foster development time for trained developers.

To answer to such questions, we designed an experiment centered on the development of a ***LuxDeal*** member. Categories of requirements considered for this experiment were the following:

- **Member management**: The application should be able to support member registration and sign in/out. Additionally, a member can enter a list of keywords describing items he is interested in. Periodically, the application checks if there are offers that match members' keywords and send a messages to possible bidders when it is the case,
- **Offer management**: The application should permit its members to create offers and browse them,
- **Message management**: Messages are used within the negotiation process; while browsing messages they have received, members can define a counter-proposal by composing a reply message linking their own offers with the one contained in the original message.

The experiment was conducted in parallel by three different pairs of people over a period of 9 working days. The first team were composed of the JAFAR core developers (R&D engineers Benoît Ries and Paul Sterges) who were developing the application with JAFAR. They were also responsible of the supervision of the experiment as well as troubleshooting (each issue encountered and tasks performed were described in a daily report). The second team (R&D engineers

Christian Glodt and Gilles Perrouin) did not know JAFAR and had to develop the application with it. Finally the third team (PhD student Catalin Amza and Dr. Alain Karsenty) had to develop the application without using JAFAR, based on the technology of their choice.

For the two first teams, the process was dictated by the FIDJI approach supported by JAFAR. It comprised the design of the application in UML 1.x and the implementation and deployment using the FIDJI tool set (JAFAR + MEDAL). The last team was free of using or not any software engineering process. However, they were also required to provide the design of their application.

At the end of the experiment, the first and third teams completed the application. In the second team, one requirement could not have been implemented (automatic matching of keywords with offers) due to time. Problems encountered in the second team were related to a documentation that did not give an overview of the overall instantiation process (so that it was difficult to understand which step came next) and to the team relative inexperience in J2EE as well as deployment bugs due to Rational XDE CASE tool (now replaced by the Software Architect tool suite [IBM06]) on which the FIDJI toolset was relying. JAFAR experts did not encounter any particular problem developing the ***LuxDeal*** member but collected a few suggestions to improve the architectural framework and discovered some minor bugs.

As the functionalities of the applications were identical, their comparison was realized on quality criteria. In particular the two first teams obtained a much more scalable and upgradeable application than the third group: this is due for one part to the component architecture of JAFAR and for the other part to the capability of J2EE containers embedding JAFAR. We also confirmed empirically that model transformations helped to fight accidental complexity by generating the EJB's interfaces as well as additional classes required by J2EE design patterns [ACM01].
With respect to our validation issues, the following conclusions can be drawn:

- **Learning curve**: Model transformations play an important role on this point. Indeed, they contributed to simplify the instantiation of JAFAR without requiring knowledge of the full JAFAR's design (black-box instantiation). The development method used to instantiate the architectural framework also has a great impact on the learning curve: an absence of a clear methodological guidance slows down the development process as the second team experienced it,
- **Architectural framework suitability**: JAFAR's functionality has been found suitable to the kind of applications to be developed with. Modules such as account management and predefined user interface templates saved the two first teams a lot of time. However, some more specialized modules (such as offer/item management) would have improved JAFAR's reusable potential and made it a better match with respect to its domain (e-bartering).

This experiment provided an initial validation of the general approach (that is developing SPL-based software using an architectural framework and model transformations) and did motivate the continuation of the work presented in this PhD dissertation. The results of the aforementioned experiment are to tome extent applicable to the current FIDJI method, if we take into account the following points. First, the FIDJI process supported by JAFAR did not include any supported requirements engineering activity (UML use cases could be defined for the application but were not processed by JAFAR transformations) whereas the current version provides

detailed guidance for requirements elicitation and analysis which has not yet been evaluated through a similar experiment. Second, in order to support the flexibility that was lacking in the original version of the method, architectural framework instantiation is white-box and supported through horizontal transformations while JAFAR proposes a black-box instantiation based on vertical transformations. As we have mentioned, this may have an impact on the learning curve of the architectural framework considered since one needs to know the architectural framework layer considered before writing the instantiation program. It is a necessary price to pay to achieve a convenient level of flexibility.

6.6.2 Validation Criteria

To the best of our knowledge there is no process improvement approach such as ISO 15504 [ISO06] or CMMI [SEI06] that provides specific criteria for validating SPL-based development methods. Furthermore, we believe that the most important issue is not to validate the FIDJI process in itself (the waterfall process mode on which FIDJI relies has been deeply investigated) but the impact it has on the quality of the derived products. Therefore, we will focus on seeking criteria related to product quality.

The preceding paragraphs pinpointed two particular criteria that were investigated during our initial experiment. Indeed, they are part of the ISO 9126 standard [ISO91] which gives a list of interesting quality attributes (and their sub-characteristics) for product validation. They are summarized below:

- **Functionality**: The sub-characteristics considered for this attribute are: *suitability, accuracy, interoperability* and *security*;
- **Reliability**: Relates to the ability of the software to deliver a guaranteed service provided that certain conditions are met. Sub-characteristics for this attributes are *maturity, fault tolerance, recoverability*;
- **Usability**: This attribute refers to the ability of its user to learn and use the product: *understandability, learnability, operability, attractiveness*;
- **Efficiency**: This attribute is linked with performance aspects of the software. Sub-characteristics: *time behavior, resource utilization*;
- **Maintainability**: Deals with evolution aspects. Sub-characteristics: *analyzability, changeability, stability, testability*;
- **Portability**: This attribute refers to the relationships of the product with its environment. Sub-characteristics: *adaptability, installability, co-existence, replaceability.*

Thus, the "learning curve" attribute is associated to usability while "suitability of the architectural framework" is a direct sub-characteristic of functionality. As noted by Atkinson et al. [ABB+02], there are two important attributes for SPL engineering missing in this taxonomy:

- **Reusability**: This attribute refers to the extent of changes required so that it fits to its new context in the product;

- **Cost**: Cost is of paramount importance and can be combined with any of the aforementioned attributes. As we mentioned in Chapter 1, its minimization (in time) is the key motivation of software reuse techniques.

It can be argued, however, that these criteria are not primary but derived: reusability is connected to maintainability and portability attributes whereas cost is related to efficiency (resource utilization in particular).

If we recall from Chapter 1 that FIDJI is focusing on simplicity, flexibility, uniformity and integration, we can derive the criteria suited to FIDJI's validation. In the following we define them as a combination of the quality attributes discussed and how the FIDJI approach influences them:

- **Reusability**: There are two factors that impact this attribute. The first one is linked to the way the architectural framework is engineered, which directly influences its changeability. The second one is related to the definition of the model transformations allowing to support architectural framework changeability;
- **Usability**: This attribute and its sub-characteristics apply to the models offered by the method, its process and transformation approach. It enables simplicity criterion validation;
- **Flexibility**: Flexibility can be evaluated by the impact it has on reusability. Indeed, state variables and instantiation constraints will greatly influence the changeability of the architectural framework;
- **Cost**: For FIDJI it is important to evaluate the cost of reusing one particular architectural framework element with respect to developing a new one for the product. This cost is heavily dependent on the suitability of the architectural framework for the product considered and the suitability of the transformation approach.

6.6.3 Validation Protocol

In the following, we give a few hints on how to assess the validation criteria we have discussed above. The first step to accomplish when designing a validation protocol is to define metrics suitable to the criteria we would like to validate. Unfortunately, cost excepted, it is difficult to select metrics directly. A well-known approach to identify relevant metrics is to use the "Goal-Question-Metric" (GQM) approach [BCR94] proposed Victor Basili in the mid eighties. Basili et al.'s approach is based on a three step refinement process:

1. **Goal elicitation**: The first step consists in defining a set of goals defining the purpose and the viewpoint of the measurements to be made. Goals involve objects (artifacts, processes, resources etc);
2. **Questioning**: The second step consists in defining questions whose answers will allow to know whether a goal is achieved or not;
3. **Metrics selection**: The final step consists in choosing the appropriate metrics with respect to the questions defined in the context of a particular goal.

For example considering flexibility attribute would result in the following GQM model:

- **Goal**: Assess the flexibility of a given architectural framework at the analysis level;
- **Question**: To what extent is an architectural framework's analysis model changeable ?;
- **Metric**: This part can be measured as follows: number of analysis elements for which one or more instantiation constraints have been defined / total number of elements forming the analysis layer. A high value of this metric indicates that it may not be easy to deviate from the architectural framework since many elements are constrained. However, a low value for this metric, although showing that the architectural framework is readily changeable, can also be an indicator of lack of maturity in the SPL definition resulting in very general architectural framework's assets which may require significant transformations to be useful.

Naturally, there can be several questions and metrics for a goal, especially if this goal is complex. In the following, we focus on the actual measurement of the aforementioned FIDJI validation criteria.

To validate usability, we can use an empirical validation technique called *ethnography* [Mye99]. This technique consists in observing a group of people performing a particular task for a certain period of time and collecting their feelings and behaviors. In addition to measuring attractiveness of the method, ethnographic studies (combined with interviews) can also provide interesting feedback on the skills required to use the method.
We can exploit the relationships between the other criteria to proceed to their validation indirectly. For instance, evaluating the cost of reusability (that is measuring the time required to reuse a specific architectural framework asset so that it fits the product) will give insights both on the changeability of the architectural framework and flexibility of the approach. In order to evaluate this criteria, an experiment involving FIDJI, KoBra and ConIFP can be considered. This experiment has three steps:

1. **Initiation**: In this step, a product (e.g. a ***LuxDeal*** member) is developed according to the three approaches. Though the asset base may differ in some respects (e.g. models) we impose that it is functionally equivalent in the three approaches. This phase will not be monitored.
2. **Construction**: Slight variations in the requirement of the product member are introduced. The new product is then redeveloped according to the three approaches. Time taken to perform this new derivation in the three approaches will be particularly monitored,
3. **Evaluation**: This step proceeds to an assessment of the second phase by comparing the time taken to derive the new product, the amount of reuse of the core assets and for FIDJI, if the instantiation program written in the first step was useful or not.

When devising such an experiment, we must be aware of the potential biases that may result in meaningless data. The first source of biases is human. We need to form homogeneous groups for the three methods considered; they should have the same skills (which can be evaluated according to their degrees) but also the same experience on the method they will work with. Another issue is related to the conflicts of interest that can exist between roles: e.g. in the initial FIDJI experiment, people in charge of carrying out the experiment were also contributing to the FIDJI method inducing a high risk of bias if some measures (such as requiring that all data is actually published) are not thoroughly observed. Particular attention should be also drawn on quality of the documentation offered to three groups. Different methods imply variations for each group. It is the role of the experiment designers to come up with these problems so that the experiment can be performed in fair conditions.

Part IV

CONCLUDING CHAPTERS

7. CONCLUSION

Abstract

This chapter outlines the main contributions of the work presented in this dissertation and draws some conclusions about it. Research perspectives will be discussed in the next chapter.

This thesis investigates how recent software engineering breakthroughs such as Model Driven Engineering [Ken02], Object-Oriented Frameworks [JF88] and Software Product lines [CN01] can be combined to devise a product line development method. More specifically, our research was driven by two needs: addressing *complexity* and improving *flexibility*. The first need has been addressed by providing a coherent set of models which can be used both for domain engineering and application engineering as the documentation of SPL variability has been removed from models to reduce the complexity of their definition and use. The application engineering process, based on the use of FIDJI-dedicated model transformation operations is both simple and supportive.
The second need has been addressed by an open notion of product line boundary that is not rigidly defined by the set of products directly derivable from the product line assets but is simply defined by a set of constraints on the domain assets that prevents some products to be derived for functional and/or technical reasons. Furthermore, our application engineering approach enables the product engineers to derive products that suit the needs of their customers without requiring evolution of the whole SPL evolve and provides guidance in their task.

In Chapter 3, we introduced the notion of architectural framework as a unifying concept that is based on the analogy between object-oriented frameworks and software product lines concepts discussed in Chapter 2. Architectural frameworks link SPL domain engineering asset models consistently at any abstraction level and support product derivation via model transformation and constraints. This approach allows to separate variability description from domain asset models, thus favouring reuse, and also to implicitly determine product-line boundaries which spare domain engineers from the task of documenting variability exhaustively and application engineers to be disturbed by too strict variability decision models when specifying and designing products.
We then defined a sequence of methodological steps in order to guide the product developer from requirements elicitation to design and explained the motivations underlying the choices of our models used at the domain engineering level.

Chapter 4 is dedicated to the requirements elicitation and analysis phase of the methodology. In particular, the following contributions were made:

- **Requirements Elicitation Template**: We defined a template to elicit software product lines at a high abstraction level based on use case variants and an informal description of

the domain concepts as well as their dependencies and variabilities. We also gave a set of methodological rules in order to manage variability information along REQET descriptions coherently;

- **Analysis Model**: The analysis model we proposed extends the ones offered by Fusion and Fondue by using the latest UML 2.0 notational elements. Moreover, it provides a fine-grained and localized approach to life-cycle/protocol specification to the system via use cases augmented with OCL expressions and UML 2 diagram. All these elements together model the interactions of the system with its environment. The analysis model is supported by an UML 2 profile that describes the elements defined in the methodology and specifies OCL well-formedness rules for FIDJI models;
- **State Variables**: We introduced state variables as a way to both control system life-cycle within a use case scenario (by defining guards expressed as OCL constraints on these variables) and to ensure flexibility in domain engineering models (guards permit to enforce some sub-scenarios while leaving open the others so as to facilitate requirements accommodation);
- **Product Derivation**: Product derivation is ensured by architectural framework instantiation of analysis models. Moreover, we showed how variability described at the requirements elicitation level can be mapped in terms of instantiation constraints at the analysis level.

Chapter 5 is devoted to the design phase of the methodology. We demonstrated the suitability of UML 2 as an architecture description language and shown how connector behavior can be modeled using OCL pre/postconditions over roles. Next, this approach was then applied to define a set of architectural styles that can be used when designing an architectural framework. This notation is used in the first step of the design process and is linked to analysis models through methodological rules and UML elements. We then gave additional views refining architecture components and integrating them in a tree-like structure.

Finally, Chapter 6 exemplifies the FIDJI methodology. Through an e-commerce case study, we gave concrete examples of the FIDJI models as well as instantiation constraints and model transformations both for domain engineering and application engineering at the requirements elicitation, analysis and design levels.

8. PERSPECTIVES

Abstract

This chapter explores some of the extensions that can be made to the FIDJI method and identifies some potential research areas in the field. This chapter is divided in two main sections. Section 8.1 discusses crucial points of the whole method such as formalization of the metamodel, tool support and actual validation. Section 8.2 delves into specific aspects of the method such as expressiveness of the use case model at the analysis level and formalization of the connector "gluing" process. Finally, Section 8.3 discusses a few research directions over the long term such as SPL-based testing.

8.1 General Considerations

8.1.1 Formalization

We have mentioned in Chapter 3 that FIDJI models were motivated by precision requirements in choosing UML 2.0 constructs that can be formalized. There are two main reasons motivating this need:

- **Analysis**: FIDJI determines the allowed and unauthorized products via instantiation constraints. For large product lines, there can be a significant number of constraints and it is difficult to determine whether these constraints conflict (thus preventing product derivation) or to know if a product is derivable before writing the instantiation program. A complete formalization of FIDJI analysis models could help in answering such questions both at the domain engineering and application engineering levels;
- **Model Generation**: Currently, the instantiation program only covers structural instantiation of architectural framework models, their behavior requiring to be modified by the product engineers. Giving a formal semantics to FIDJI models would be a first step towards taking into account models' behavior and would therefore complete the automated support can be provided to the instantiation program.

At the structural level, FIDJI models are based on UML structural constructs such as `Class` and `Components`. Instead of developing a whole new formalization of these constructs from scratch, one possible approach is to reuse the structure proposed by Richters in his PhD thesis [Ric02]. This structure, based on set theory and first-order logic, was initially defined to provide a solid foundation to OCL. The fundamental construct is the type signature which consists of type names and operations defined for types. Then, elements considered in this structure are those of a "classic" class diagram, that is classes (set of names determining types), attributes (functions between two types), operations (n-ary functions between types), associations (set of association

names and a function "associates" which gives a list of related class names for a given association name), and lastly a partial order defined over the set of classes modeling inheritance relationships between classes. Naturally, this structure does not take into account UML 2.0 specific constructs such as components, ports or connectors. However, thanks to the object-oriented nature of the UML 2.0 metamodel, this structure can not be applied to classes of a given model but can be applied to metaclasses of the UML specification at the M2 level.

As we mentioned above, one of the motivations for formalizing behavior is to be able to determine if a certain combination of operations (either at the analysis or design level) is derivable with respect to the instantiation constraints for the level considered. Obviously, the use of state variables in operation definitions determines if an operation can be called but it may be difficult to answer such a question if several operations are involved in modifying the same state variable. Ideally, it would be desirable to check the possibility of such a scenario using a temporal logic formula. Unfortunately, the OCL specification which is used to specify operation sequencing does not allow to check for temporal logic formulas. Indeed, as stated by Kyas and De Boer in [KdB04], the `LocalSnapshot` construct, defining the notion of object history in OCL (e.g. at the analysis level, we would be interested in capturing the history of a use case component in order to check for a particular instantiation) is only part of the language semantics and no construct is provided to access such an history. This motivated the authors to define an explicit construct to manage the history of an object. Other authors have also noticed the need for extending OCL in order to support more fine-grained temporal management. Gogolla and Ziemann [ZG03a] propose the addition of well-known temporal logic operators such as "previous", "next", "always" and their derivations for past events. The whole approach is formalized in [ZG03b] by means of Linear Temporal Logic. In [BFS02], Bradfield et al. propose a template for OCL (by adding new kinds of conditions such as "after" and "eventually") which acts as an upper language layer, the lower layer being the OCL standard itself. Flake and Mueller [FM04] define a UML profile supporting the definition of past and future properties and base its semantics on a clocked linear temporal logic. This approach is interesting since it enables a rapid support within CASE tools and is consistent with the OCL specification. A more detailed discussion on these approaches can be found in [Fla03]. Cengarle and Knapp also extended OCL using temporal operators and gave it an operational semantics in the context of real-time applications [CK02, CK05]. All these approaches need to be evaluated with respect to our use of OCL in order to find which is the most appropriate. Since our use of OCL which is mainly "interaction-oriented", an approach that would provide both temporal operators with a trace-based semantics (such as the one presented in [CK04]) can be a solution to investigate first. Then, we should evaluate the suitability of this semantics domain for the operation model in order to generate UML state machines.

8.1.2 Model Transformation

Language Definition

The first point in this aspect is to provide a complete definition of the transformation language that is only sketched in this dissertation. This concerns the syntax and semantics of the transformation language. The definition of an horizontal transformation language for product derivation is currently being studied in the context of post-doctoral studies [GK07].

Transformation Framework

In order for FIDJI to be efficient, it is important to provide an adequate library of transformation operations that will be reused in instantiation programs. A proper balance should be found between operations that are too general (in that case, the library of transformation operations would be reduced to create/update/delete operations working on any model element) or too specific (in that case, instantiation program writers would be puzzled by a huge operation library).

Vertical Transformation Support

Our model transformation language, designed to assist product analysts, architects and developers to instantiate the architectural framework, works at the horizontal level. As we have seen in Chapter 2, it is also possible to define vertical transformations to relate and generate model elements across different abstraction levels. In our context, the FIDJI methodology could benefit from the definition and support of vertical transformations on the following points:

- **Artifact Generation**: The definition of vertical transformations between analysis and design layers of the architectural framework (due to the textual nature of our requirements elicitation approach, vertical transformations defined for this layer are, though possible, less likely) may be applied on the product analysis model to generate its design model. Vertical transformations may also be used as the main technique to define an implementation phase for the methodology;
- **Traceability**: Currently, vertical traceability (horizontal traceability may be already realized by a transformation platform supporting our transformation language) is defined manually through textual document and UML <<`trace`>> links. We mentioned that for the architectural framework it is only a minor issue since it changes only in case of the evolution of the SPL. An individual product evolution may occur more frequently. Vertical transformations while generating design elements can also define traceability links automatically.

8.1.3 Towards Tool Support

In order for a methodology to be applied efficiently, a convenient tool support is necessary. Currently, there is no tool support for the FIDJI approach as presented in this dissertation (tool support [GRS03b, GRS03a] was developed for preliminary versions of the approach [GP02, GP04]). This section gives some information to define such an environment which amounts to addressing two activities; architectural framework modeling and architectural framework instantiation support.

Architectural Framework Modeling

Models proposed by the FIDJI methodology to define architectural frameworks ease tool support mainly because of their compliance to the latest OMG-related modeling standards such as UML and OCL. Thus, any CASE tool supporting these standards might be used to define architectural frameworks. Due to the maturing process of the UML 2 specification and its size, supporting

tools have long been awaited. Now, the specification is almost stable, some tools support the metamodel and concrete syntax of the specification in its entirety. In particular, in order to define an architectural framework according to the FIDJI models, the following constructs need to be supported:

- **Components and Composite Structures**: These constructs are central to the FIDJI models. They are found in the analysis models for the definition of use case components, in the design models in the GAM and ISM. Most of the CASE tools now propose these constructs. However, it should to be noted that they differ in the way they model these visually, for example some tools do model the internal structure of components in separate diagrams and therefore would fail to depict use case components and ISM-based models;
- **Profiling**: FIDJI models are mainly defined thanks to UML profiles. With the development of standardized UML profiles (promoted by the MDA initiative to describe PSMs), CASE tool vendors have significantly improved their offer for UML profiling support;
- **OCL**: We make intensive use of OCL either to improve descriptions (use cases in the analysis models, connectors and role in design models, profiles) or to define and constrain architectural framework instantiation (see below). We therefore require a certain level of support for OCL. Depending on the CASE tool, this support varies from syntax validation to evaluation [Bor07, IBM06, NoM07];
- **Action Language**: We have chosen KerMeta [MFJ05] as an action language for its simple syntax and its object-oriented basis which make it interesting to define the behavior of PSM classes. Therefore, tool support must be compliant to the eclipse-based KerMeta environment.

The UML diagrams which appear in this dissertation have been realized with NoMagic's MagicDraw [NoM07] CASE tool which has the advantage to be very close to the standard in its metamodel implementation and to provide an integrated support for profiles. Concerning OCL, MagicDraw allows the specification of rules defined both at the meta-model level (we encoded some of the well-formedness rules for our profiles in the tool) and model level (in this case rules will be evaluated on model element instances) and their evaluation. Therefore, FIDJI analysis and design profiles can be defined in MagicDraw and can be used to validate user-defined architectural framework models.

There are some FIDJI model elements that are not based on UML. This is especially true at the analysis level: use cases and data dictionaries (domain concepts, operations...). We already mentioned that textual use case descriptions can be embedded in the UML use case components which they are associated to via its "packaging" facility: thus any component can own instances of `Artifact` refering to any file on a physical system. Concerning data dictionaries, tabular notations can be easily managed using a relational database whose schemas are deduced from table columns. The traceability file linking REQET elements and analysis can also be implemented via a table in such a database.

Architectural Framework Instantiation Support

Our approach to model transformation supporting the architectural framework instantiation has been designed to facilitate its translation/implementation in model transformation languages/tools:

- **Language Syntax**: The imperative syntax of the language used to define an instantiation program is borrowed from general purpose programming languages such as JAVA [GJSB05]. As we have seen in Chapter 2, there are transformation environments that support such an imperative approach;
- **Transformation Operations**: As we have seen, transformation operations are specified declaratively in terms of OCL pre/postcondtions. These conditions can be used to define a library of transformations supporting the instantiation of FIDJI models. For example, in the ATL language and engine [JK05], preconditions and postconditions can be adapted to the ATL language (whose syntax is actually very close to OCL) to form "matched rules" which can be called by other rules.

There are several possibilities to support the FIDJI transformation language (which is currently seen more as a front-end enabling the seamless definition of instantiation programs but may evolve as a transformation language on its own). Yet, we believe that in parallel with the development of QVT [OMG05b], OCL-based hybrid languages such as ATL [JK05] will develop and be integrated with UML modeling tools (so far only Borland's Together has provided both UML and QVT support) to ease architectural framework instantiation.

8.1.4 Method Process Model

We outlined in Chapter 3 that the waterfall process model was well-suited for the FIDJI method. However, it may be interesting to consider a more incremental development process (such as Boehm's spiral [Boe88]). In particular, this is worthwhile for the customer-specific part of the application; an incremental process would allow to validate several series of prototypes with the customers before the product is finished. In order to do so we should handle the following issues:

- **Partial Instantiation**: The capacity to cope with a partial instantiation of the architectural framework at any of its abstraction layers is a necessity;
- **Partial Consistency**: Since partial instantiation will induce partial product models, partial consistency checking has to be supported.

The first issue deals with instantiation constraints; they should be designed so that partial instantiation is possible. One option is to organize instantiation constraints in several categories depending on the completion degree of the product. With such priorities, product engineers would know which instantiation constraints have to be ensured at any time during the development and which ones can be safely ignored in the context of a prototype.

Addressing the second issue requires a flexible approach to consistency management. Sendall and Küster have discussed this point in [SK04]. In particular, in the context of an approach called "model round-trip engineering", they have identified the following steps:

- defining under what circumstances the models in question are consistent and inconsistent;
- deciding when inconsistent models should be made consistent again;
- devising a plan for reconciling the models according to the intent and expectations of the user;

- applying the devised plan by reconciling the models.

Assuming that the first step has been covered by consistency rules to be applied on FIDJI models, we need to identify consistency rules that can be omitted for prototypes and those that must be true for any partial product. In our context, the reconciliation plan will be based on the priority of the consistency rules violated. The last step can be performed via the definition of a few new transformations in the instantiation program.
There is an alternative for checking all the consistency rules and then deciding which of them can be ignored. It is possible to check only the relevant rules (as determined by the priority of consistency rules). If this is not an issue for specific rules that may be defined in the FIDJI profile, this can be problematic for UML consistency rules that are often automatically checked by CASE tools. In this perspective, following an approach such as the one described in [BRpG07] regarding differed consistency rule application is interesting.

8.2 Specific Issues

In this section, we describe some interesting issues related to specific points of the FIDJI method.

8.2.1 Requirements Elicitation and Analysis

In this section, we discuss limitations and future research directions related to the requirements elicitation and analysis phases of the methodology.

8.2.2 REQET

We have identified two points which deserve future work:

- **Variability Global View**: Currently, variants are scattered across the sub-templates which is a tricky situation. It would be desirable to have a global view on these variants and their relationships. Moreover, such a view would simplify internals consistency checks;
- **Dependencies Nature**: The nature of dependencies (inheritance, composition) in the DOMET is deliberately left open.

Concerning the first point, we propose to add an additional viewpoint for the template to focus on showing variants globally. As we have seen in Chapter 2, feature models provide the necessary constructs to deal with variability at the requirements elicitation level. Since all the variability information of a REQET-based SPL description could be gathered in this view, consistency checking amongst variants can be easier. Furthermore, thanks to the well-founded semantics [SHTB06] of feature models, we have given in [GP06] a set of consistency rules that can be used as guides to define SPL instantiation constraints at the analysis level.

The second point is methodological; though we believe that we should not restrict the types of dependencies offered by REQET, a taxonomy of useful types and their impact on the whole description may be useful to guide SPL analysts. Such a taxonomy has to be defined and validated over case studies in order to assess its usefulness.

8.2.3 FIDJI Analysis

State Variables Expressiveness at the Analysis Level

Our motivation to specify operation sequencing in the use case and operation models through state variable has arisen from the observation that it is impossible to define all the possible operation sequences for the system in a global manner. This led us to work at the operation and use case step levels and to focus on mandatory and unauthorized sequences, leaving more freedom to instantiate an architectural framework scenario for a given product. It may be interesting to compare the expressiveness of this formalism with other existing approaches such as UML 2.0 sequence diagrams or Fusion regular expressions (which are derivable into state machines [CAB+94]). To do so, the semantics of the formalisms to be compared has to be formalized first. Then expressiveness can be assessed by translating one formalism into another.

Answering to such a question can have important consequences at the methodological level since it may influence the type of variability the SPL may support and as a consequence the way it is designed.
Another interesting consequence of the answer is the possibility to translate FIDJI use cases in terms of UML 2.0 sequence diagrams; this is maybe useful to communicate more easily with stakeholders or to generate test cases.

Transactional Composition of Operations

In the first paragraph of Section 4.2.3, we motivated our choice for a simple view on the functionality of the system based on asynchronous events and instantaneous operations *à la* Fusion. However, in some cases, it may be useful to consider the instantaneous execution of a sub-scenario resulting from the execution of an operation sequence. For example, consider a registration process in an university; a prospective student submits to a dean a file describing his previous studies as well as administrative and financial information. The dean will request the university to register this student. The university will process the request and check with the student's bank whether the current amount on his account is greater than the university registration fee. Finally, the university will inform the dean whether the registration is possible or not, and the dean will notify the prospective student.
This process when implemented will probably comprise both synchronous (since the dean may wait for the university to check with the bank whether the student is solvent or not) and asynchronous execution of operations resulting in the intermediate system states.
Specifying this process in such a way is not permitted in the execution model we chose. One possible workaround to this issue is to introduce a transactional concept for composing operations. The registration process shown above can be seen as a transaction embedding several operations and executed atomically. For our execution model, a transaction will behave as if it were a single operation whose preconditions and postconditions are derived from those associated with the operations embedded in the transaction. It may be argued that, if for the execution model a transaction behaves as an operation and corresponds to a sub-scenario in a use case, why not simplify the scenario and define a simple operation instead ? We think that by using too coarse-grained operations, we would fail to capture the interactions between the actors and the system in a useful manner; for example, in our registration process, this would imply that there is only one interaction between the student and the dean, therefore we would miss all the interactions between the system and the bank.

The schema for transactions is inspired by the one for operations is depicted below:

Transaction Name: `<name of operation>`

Description: `<natural language description of the operation>`

Attributes/Parameters: Union of all parameters defined for the embedded operations

Sends: Union of all signals sent by the embedded operations

Preconditions: Conjunction of all preconditions of the embedded operations

Postconditions: Conjunction of all postconditions of the embedded operations

Embedded Operations: List of the participating operations ordered as dictated by the use case steps of the transaction

8.2.4 FIDJI Design

Concerning the design phase of the FIDJI methodology, a few points deserve further investigation.

Gluing

As we have seen in Chapter 5, our design models support the independent modeling of architectural components and connectors (thus allowing architectural style modeling) that have to be "glued" in the actual models of the architectural framework. Currently, this gluing process is based on matching the virtual operations with interface operations and verifying that this matching is compatible with the connector behavior. We have also underlined that we wanted to have a flexible notion of compatibility in order to have a wide range of situations in which architectural styles may be used. It might be useful to precisely define how this compatibility can be assessed at the semantic level in order to guide SPL developers to determine the applicability of a given architectural style. We think that formalization has clearly a role to play to achieve such guidance. This would allow us to give precise constraints on virtual operation matching. The first approach would be to think that these conditions are those of a formal refinement relationship between the virtual operation and the matched component's interface operation. However, as demonstrated by Allen and Garlan [AG97], this is unnecessarily restrictive: there are cases in which incompatible behavior will never be reached by a given port thus making matching possible. Allen and Garlan described their compatibility notion by reasoning on the traces of roles and ports. However they did not formalize their approach (they illustrate it with a few examples) which makes it difficult to systematize in a methodological perspective.

Selection of Architectural Styles

We provided a few architectural styles that are useful for the kind of applications targeted by the method, i.e. multitier web applications. More experience is required to actually assess these styles with respect to the methodological context and application domain. As opposed to design patterns which have been specialized for a particular platform such as J2EE (Alur et al. [ACM01]), architectural styles are very general building blocks. The good side is that this generality makes architectural styles reusable but this turns out to have a disadvantage in that it is more difficult to choose the architectural style that is appropriate to the architectural framework being modeled. As noted by Shaw and Clements [SC06] the relationships that exist between a given style and the quality attributes it is supporting should be expanded. Since quality attributes reflect a particular domain (e.g. "performance" will not have the same priority in a real-time embedded context as in the web-applications one), linking architectural styles with the domain will also force architectural style writers to reason about the quality attributes these styles enforce.
If we relate this to vertical transformation support, one can imagine that modeling tools will assist architectural framework designers in modeling the design layer by providing transformations that create the architecture of the architectural framework in accordance with a particular style. This approach is already used at the design pattern level; for example, the MagicDraw CASE tool [NoM07] supports the application of the "Gang of Four" Design patterns [GHJV95] and we illustrated in [GP04] how a vertical transformation supporting the value object pattern [ACM01] can be defined.

Implementation

Naturally, design is not the ultimate phase of a software system development, therefore, an implementation phase should be provided in order to complete the methodology. The first point is to provide a detailed design phase that takes into account the specificities of a given platform (a platform-specific model in the MDA meaning). This can be done using of dedicated UML profiles such as the metamodel and profile for EJB [OMG04] distributed as part of the OMG's Enterprise Distributed Object Computing (EDOC) profile. The second point is related to the translation of this detailed model into the actual code of the architectural framework. One interesting research direction is to study how architectural elements are mapped into code elements; for example, a UML component may be mapped either as a JAVA class or as a package which consists of JAVA files implementing the component. Here again, vertical transformations have to be defined in order to assist the development process.

We have mentioned in Chapter 3 that the implementation layer of an architectural framework contains an object-oriented framework. Concerning instantiation constraints, OCL constraints are not suitable at this level. Moreover, constraints have to be specified directly in the implementation language. In Chapter 2, we presented Hou et al.'s [HHR04] approach for defining instantiation constraints at the source code level. An interesting topic would be to see how design instantiation constraints can be rewritten in the target implementation language and possibly performed while generating architectural framework source code via vertical transformations.
Performing product derivation at this level means instantiating the object-oriented framework. Research issues at this level should investigate whether examples of product derivations (which correspond to the set of all previous applications developed on the basis of this architectural framework) are sufficient to guide developers. We should determine whether it is practical to develop the implementation directly or if a dedicated transformation language (which cannot be based on OCL for the same reason as above) working on source code is more convenient for an efficient and reliable object-oriented framework implementation.

8.3 Long-term Perspectives

8.3.1 SPL-Based Testing

As noted in [MvdH03], testing software product lines is different from testing "normal" software since all its constituents do not have the same importance (e.g. when performing unit testing, more attention should be paid to a mandatory component than to an optional one) and the inherent variability of products makes conformance testing difficult. It is also necessary to determine which assets do not need to be tested for a particular product. In the context of the FIDJI method, two points have to be explored.

First, new techniques for testing an architectural framework have to be defined. Indeed, current techniques [KKT06] are based on the explicit documentation of asset variability in order to derive test scenarios. As FIDJI does not document variability in the same way, an approach to "test by restriction" should be devised in order to ensure that the implementation of the architectural framework does not support requirements prohibited by instantiation constraints. Therefore, we have to explore how instantiation constraints can be combined with standard test techniques in order to effectively test an architectural framework.

The second point refers to the reuse of test scenarios. The key idea is to define how the instantiation program can be used to transform test cases of the architectural framework at the domain engineering level into "single product test cases" at the application engineering level. Such a mechanism would have the advantage over template-based methods [OG06] to take into account application specific features and not only those of the domain assets.

8.3.2 Architectural Framework Life-cycle

In this dissertation, we focused on providing a metamodel for architectural framework description, but not on its development. In the following, we give a few research directions to manage the whole architectural framework life-cycle.

Creation

FIDJI essentially endorses a "top down" approach to SPL-based development and the metamodel we have given may support the definition of a SPL from requirements elicitation to design using this approach. But not all product lines are initiated in a new company or in a fresh branch of a company; in many cases, software product lines are identified after several successful developments which have common features and on which the company may wish to emphasize. At the technical level, this means that some implementation classes may have already been factorized and reused for the development of these products thus forming the basis for an object-oriented framework built "bottom up". Our design model being component-based, it facilitates the construction of the architectural framework design layer from heterogeneous components and the use of state variables at the connector level allows to "glue" components with more flexibility. However, more research is required to define such a reverse engineering process and how design instantiation constraints could be derived from it.

Evolution

We mentioned that our instantiation approach was promoting SPL evolution as the instantiation program for one product may be used to evolve the architectural framework itself. This is interesting on account of the fact that the need for new features for the SPL generally arises when developing a new product. Note that this evolution may be partial; indeed, only some of the instantiation program instructions may be relevant to evolve the architectural framework while the other ones are specializations that are useful only for one product. To do so, a feature has to be traced from its description at requirements elicitation level to its realization via transformation instructions in the analysis and design instantiation programs. In order to manage SPL evolution more efficiently, a repository containing features and their realization may be defined and maintained (which may also be useful to foster instantiation program reuse during product derivation). This approach shares some similarities with the Fireworks approach proposed by Schobbens et al. [RS04]. More research has to be done to assess the feasibility of such an approach in practice.

Some of the evolutions performed on the architectural framework, such as bug fixes, need to be propagated to the products. Symmetrically, the instantiation program is used to make such a propagation. However, the instantiation program which was used to perform architectural framework evolution may not involve the same elements are those which have been reused by the instantiation program to derive the product. Therefore, in a first step, it is necessary to identify the product's model elements that are subject to evolution. This can be done either by examining instantiation programs or by comparing horizontal traceability links generated by them. More research is needed to define traceability approaches facilitating this comparison and methodological ways to perform it efficiently.

BIBLIOGRAPHY

[AAG95] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4(4):319–364, 1995.

[ABB+02] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[ABM00] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based Product Line Development: the KobrA approach. In *Proceedings of the first conference on Software product lines : experience and research directions*, pages 289–309, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[ACM01] D. Alur, J. Crupi, and D. Malks. *Core J2EE patterns*. Prentice Hall PTR Upper Saddle River, NJ, 2001.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[AGP05] Paris Avgeriou, Nicolas Guelfi, and Gilles Perrouin. Evolution Through Architectural Reconciliation. *Electr. Notes Theor. Comput. Sci.*, 127(3):165–181, 2005.

[AH04] C. Atkinson and O. Hummel. Towards a Methodology for Component-Driven Design. In N. Guelfi, editor, *RISE : rapid integration of software engineering techniques*, number 3475 in LNCS, pages 23–33, Luxembourg-Kirchberg, Luxembourg, November 2004. Springer.

[AKL03] Aditya Agrawal, Gabor Karsai, and Akos Ledeczi. An end-to-end domain-driven software development framework. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 8–15, New York, NY, USA, 2003. ACM Press.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, CMU School of Computer Science, January 1997. CMU-SCS-97-144.

[AR03] E. Astesiano and G. Reggio. Towards a well-founded UML-based development method. In *Proceedings of the First International Conference on Software Engineering and Formal Methods*, pages 102–115, 2003.

[ARNRSG06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model Traceability. *IBM Systems Journal*, 45(3):515–525, 2006.

[BA96] S.A. Bohner and R.S. Arnold. An introduction to software change impact analysis. In *Software Change Impact Analysis*, pages 1–26. IEEE Computer Society, 1996.

[BBB+05] Jean Bézivin, Mireille Blay, Mokrane Bouzeghoub, Jacky Estublier, and Jean Marie Favre. As mda: Ingénierie dirigée par les modèles; rapport de synthèse. Technical report, CNRS, Janvier 2005.

[BCD+00] Len Bass, Paul Clements, Patrick Donohoe, John McGregor, and Linda Northrop. Fourth product line practice workshop report. Technical Report CMU/SEI-2000-TR-002, Software Engineering Institute, 2000.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison Wesley, 2nd edition, 2003.

[BCR94] V. Basili, G. Caldiera, and H.D. Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[BCRP05] A. Boronat, JÁ Carsí, I. Ramos, and J. Pedrós. An approach for cross-model semantic transformation on the .net framework. In Vaclav Skala and Piotr Nienaltowski, editors, *.NET conference Technologies*, University of Western Bohemia, Czech Republic, 2005.

[BCS00] Don Batory, Rich Cardone, and Yannis Smaragdakis. Object-oriented frameworks and product lines. In P. Donohoe, editor, *Proceedings of the First Software Product Line Conference*, pages 227–247, 2000.

[BD99] Greg Butler and Pierre Denommée. Documenting frameworks. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Building Application Frameworks*, chapter 21. Wiley & Sons, 1999.

[BDP07] Frédéric Boniol, Philippe Dhaussy, and Claire Pagetti. Points de Vue et Sémantiques Ad Hoc. In *SéMO' 07 at IDM*, Toulouse, France, 2007.

[BEA] BEA. Weblogic website,. `http://www.bea.com/framework.jsp?CNT=index.htm\&FP=/content/products/weblogic/`.

[Béz05] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[BFG+01] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *PFE4*, pages 11–19, 2001.

[BFJ+03] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective model driven engineering. In *UML 2003*, 2003.

[BFS02] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching ocl using observational mu-calculus. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 203–217, London, UK, 2002. Springer-Verlag.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 273, Washington, DC, USA, 2001. IEEE Computer Society.

[BGK98] G. Butler, P. Grogono, and F. Khendek. A reuse case perspective on documenting frameworks. In *APSEC '98: Proceedings of the Fifth Asia Pacific Software Engineering Conference*, page 94, Washington, DC, USA, 1998. IEEE Computer Society.

[BGMW00] Joachim Bayer, Cristina Gacek, Dirk Muthig, and Tanya Widen. Pulse-i: Deriving instances from a product line infrastructure. In *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, page 237, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[Bib97] Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems, PhD thesis no 2919*. PhD thesis, University of Geneva, 1997.

[Big06] BigLever. GEARS Website `http://www.biglever.com/index.html`, 2006.

[BJ94] Kent Beck and Ralph E. Johnson. Patterns generate architectures. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 139–149, London, UK, 1994. Springer-Verlag.

[BLO03] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact Analysis and Change Management of UML Models. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, Washington, DC, USA, 2003. IEEE Computer Society.

[BLOS06] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sówka. Automated impact analysis of uml models. *Journal of Systems and Software*, 79(3):339–352, 2006.

[BLY06] L. C. Briand, Y. Labiche, and T. Yue. Vertical impact analysis of uml models. Technical Report SCE-06-06, Carleton University, April 2006.

[BM03] P. Braun and F. Marschall. Transforming Object Oriented Models with BOTL. *Electronic Notes in Theoretical Computer Science*, 72(3), 2003.

[BMW99] Joachim Bayer, Dirk Muthig, and Tanya Widen. Customizable domain analysis. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 178–194, London, UK, 1999. Springer-Verlag.

[BNT02] Robert Biddle, James Noble, and Ewan D. Tempero. Supporting reusable use cases. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 210–226, London, UK, 2002. Springer-Verlag.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[Boo94] Grady Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[Bor] Borland. Visibroker website. `http://www.borland.com/us/products/visibroker/index.html`.

[Bor07] Borland. Together website. `http://www.borland.com/us/products/together/index.html`, February 2007.

[Bos00] Jan Bosch. *Design and Use of Software Architectures.* Addison-Wesley, 2000.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible markup language (xml) 1.0. Technical report, World Wide Web Consortium, 2006.

[BRpG07] Xavier Blanc, Laurent Rioux, and Marie pierre Gervais. Gestion de la cohérence des modèles au cours de la construction. In *IDM*, Toulouse, France, 2007.

[BTRC05] D. BENAVIDES, P. TRINIDAD, and A. RUIZ-CORTES. Automated reasoning on feature models. In *17th Conference on Advanced Information Systems Engineering*, Lecture notes in computer science, pages 491–503. Springer, 2005.

[BV06] András Balogh and Dániel Varró. Advanced model transformation language constructs in the viatra2 framework. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1280–1287, New York, NY, USA, 2006. ACM Press.

[CA05] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach based on Superimposed Variants. In *4th international conference Generative programming and component engineering*, volume 3676 of *LNCS*, pages 422–437. Springer-Verlag, 2005.

[CAB+94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: the Fusion Method.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[Car03] Eric Cariou. *Contribution un processus de rification d'abstractions de communication.* PhD thesis, Ecole Doctorale Matisse, Université de Rennes 1, 2003.

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley, 2002.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 156–172, London, UK, 2002. Springer-Verlag.

[CD94] Steve Cook and John Daniels. *Designing object systems: object-oriented modelling with Syntropy.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[CE00] Kryztof Czarnecki and Ulrich Eisenecker. *Generative programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing, 2000.

[CH06] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[Che76] Peter Pin-Shan Chen. The entity-relationship model–toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[CHE05a] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998.

[CK02] María Victoria Cengarle and Alexander Knapp. Towards OCL/RT. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 390–409, London, UK, 2002. Springer-Verlag.

[CK04] María Victoria Cengarle and Alexander Knapp. UML 2.0 Interactions: Semantics and Refinement. In Jan Jürjens, Eduardo B. Fernandez, Robert France, and Bernhard Rumpe, editors, *3rd International. Workshop on Critical Systems Development with UML (CSDUML '04, Proceedings)*, pages 85–99. Technische Universität München, 2004.

[CK05] María Victoria Cengarle and Alexander Knapp. Operational Semantics of UML 2.0 Interactions. TUM-Report TUM-I0505, Technische Universität München, München, 2005.

[CMSD04] Eric Cariou, Raphael Marvie, Lionel Seinturier, and Laurence Duchien. Ocl for the specification of model transformation contracts. In *OCL and Model Driven Engineering (at UML 2004)*, 2004.

[CMU07] CMU. Acmestudio website. `http://www.cs.cmu.edu/~acme/AcmeStudio/index.html`, 2007.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, Reading, MA, USA,, 2001.

[CO93] Stephen J. Cannan and Gerard A.M. Otten. *SQL–the Standard Handbook: Based on the New SQL Standard (ISO 9075: 1992 (E))*. McGraw-Hill, 1993.

[Coc] Alistair Cockburn. `http://www.usecases.org`.

[Coc01] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2001.

[Com06] Compuware. Optimalj website. `http://www.compuware.com/products/optimalj/default.htm`, 2006.

[Dah68] Ole-Johan Dahl. *SIMULA 67 common base language*. Norwegian Computing Center, 1968.

[DDQ78] P.J. Denning, J.B. Dennis, and J.E. Qualitz. *Machines, Languages and Computation*. Prentice Hall Professional Technical Reference, 1978.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[Dic02] J. Dick. Rich traceability. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering*, Edinburgh, Scotland, 2002.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[dLV02] Juan de Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 174–188, London, UK, 2002. Springer-Verlag.

[DS99] Jean-Marc DeBaud and Klaus Schmid. A systematic approach to derive the scope of software product lines. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 34–43, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[DSB04] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Experiences in Software Product Families: Problems and Issues during Product Derivation. In *SPLC3*, pages 165–182, September 2004.

[DSB05] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *J. Syst. Softw.*, 74(2):173–194, 2005.

[DW99] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and frameworks with UML: the catalysis approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[EB04] M. Elaasar and Lionel C. Briand. An overview of uml consistency management. Technical Report SCE-04-18, Carleton University, August 2004.

[Egi02] Egidio Astesiano and Michel Bidoit and Hélène Kirchner and Bernd Krieg-Brückner and Peter Mosses and Donald Sannella and Andrzej Tarlecki. CASL: the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

[EN04] S.M. Easterbrook and B.A. Nuseibeh. What is requirements engineering? In *Fundamentals of Requirements Engineering.* 2004.

[Fav04a] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, UML 2004 Satellite Event*, 2004.

[Fav04b] J.M. Favre. Foundations of Model (Driven)(Reverse) Engineering: Models. In *Dagsthul Seminar on Language Engineering for Model Driven Development, DROPS, http://drops. dagstuhl. de/portals/04101*, 2004.

[FDVF06] Franck Fleurey, Zoé Drey, Didier Vojtisek, and Cyril Faucher. Kermeta language reference manual. Technical report, IRISA, 2006.

[FGJ+03] Alessandro Fantechi, Stefania Gnesi, Isabel John, Giuseppe Lami, and Jörg Dörr. Elicitation of Use Cases for Product Lines. In *PFE2003: 5th International Workshop on Software Product-Family Engineering*, Lecture Notes in Computer Science, pages 152–167, Siena, Italy, 2003. Springer.

[FGLN04] Alessandro Fantechi, Stefania Gnesi, Giuseppe Lami, and E. Nesti. A methodology for the derivation and verification of use cases for product lines. In *SPLC*, pages 255–265, 2004.

[FJ00] Mohamed E. Fayad and Ralph E. Johnson. *Domain-specific application frameworks: framework experience by industry.* John Wiley & Sons, Inc., New York, NY, USA, 2000.

[Fla03] Stephan Flake. Temporal ocl extensions for specification of real- time constraints. In *Workshop Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS'03) at UML 2003*, San Fransico, CA, USA, 2003.

[FM04] Stephan Flake and Wolfgang Mueller. Past- and future-oriented time-bounded temporal properties with ocl. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*, pages 154–163, Washington, DC, USA, 2004. IEEE Computer Society.

[FO85] M. Fridrich and W. Older. Helix: The architecture of the XMS distributed file system. *IEEE Software*, 2(3):21–29, 1985.

[Fou] Apache Software Foundation. Axis project. `http://ws.apache.org/axis/`.

[FPB87] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[FPR00] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The Uml Profile for Framework Architectures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[FS97] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, 1997.

[FTV02] Lidia Fuentes, José M. Troya, and Antonio Vallecillo. Using uml profiles for documenting web-based application frameworks. *Ann. Softw. Eng.*, 13(1-4):249–264, 2002.

[FV04] Lidia Fuentes and Antonio Vallecillo. An introduction to uml profiles. *UPGRADE (European Journal for the Informatics Professionals)*, V(2):6–13, April 2004.

[GÖ95] Andreas Günter. *Wissensbasiertes Konfigurieren–Ergebnisse aus dem Projekt PROKON.* Infix Sankt Augustin, 1995.

[GAK99] George Yanbing Guo, Joanne M. Atlee, and Rick Kazman. A Software Architecture Reconstruction Method. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 15–34, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[GFdA98] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating Feature Modeling with the RSEB. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, Washington, DC, USA, 1998. IEEE Computer Society.

[GGMP06] Barbara Gallina, Nicolas Guelfi, Andreea Monnat, and Gilles Perrouin. A Template for Product Line Requirement Elicitation. Technical Report TR-LASSY-06-08, Laboratory for Advanced Software Systems, University of Luxembourg, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[GK07] Nicolas Guelfi and Jacques Klein. SPLIT - A Software Product Line Transformation Language. `http://se2c.uni.lu/tiki/tiki-index.php?page=SplitOverview`, 2007.

[GL00] Wolfgang Grieskamp and Markus Lepper. Using use cases in executable z. In *ICFEM '00: Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods*, page 111, Washington, DC, USA, 2000. IEEE Computer Society.

[GLR+02] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of mda. In *First International Conference on Graph Transformation, ICGT*, 2002.

[GM95] Dipayan Gangopadhyay and Subrata Mitra. Understanding frameworks by exploration of exemplars. In *CASE '95: Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering*, page 90, Washington, DC, USA, 1995. IEEE Computer Society.

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[GP02] Nicolas Guelfi and Gilles Perrouin. Rigourous Engineering of Software Architectures: Integrating ADLs, UML and Development Methodologies. In *6th Annual IASTED International Conference on Software Engineering and Applications, ACTA Press*, pages 523–529, 2002.

[GP04] Nicolas Guelfi and Gilles Perrouin. Using Model Transformation and Architectural Frameworks to Support the Software Development Process: the FIDJI Approach. In *2004 Midwest Software Engineering Conference*, pages 13–22, 2004.

[GP06] Nicolas Guelfi and Gilles Perrouin. Coherent Integration of Variability Mechanisms at the Requirements Elicitation and Analysis Levels. In Dirk Muthig and Paul Clements, editors, *Workshop on Managing Variability for Software Product Lines: Working With Variability Mechanisms at 10th Software Product Line Conference*, Baltimore, MD, USA, 2006.

[GPR+03] Nicolas Guelfi, Gilles Perrouin, Benoît Ries, Paul Sterges, and Shane Sendall. MEDAL 1.0 Reference. Technical Report TR-CST-03-01, Luxembourg University of Applied Sciences, December 2003.

[GPR04] Nicolas Guelfi, Cedric Pruski, and Benoit Ries. A Study of Mobile Internet Technologies for Secure e-commerce Applications Development. In *Techniques and Applications for Mobile Commerce (TAMOCO) part of Multi-Konferenz Wirtschaftsinformatik 2004*, pages 194–203, 2004.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GR02] Nicolas Guelfi and Benoit Ries. Using and Specializing a Pattern-Based E-business Framework: An Auction Case Study. In *6th Annual IASTED International Conference on Software Engineering and Applications, ACTA Press*, pages 512–522, 2002.

[Gro04] DoD Architecture Framework Working Group. Dod architecture framework version 1.0 (volume i: Definitions and guidelines). Technical report, Department of Defense, 2004.

[Gro06] The Open Group. Togaf v8.1.1. `http://www.opengroup.org/togaf/`, 2006.

[GRS03a] Nicolas Guelfi, Benoît Ries, and Paul Sterges. JAFAR2: an Extensible J2EE Architectural Framework for Web Applications. Technical Report TR-DIA-03-05, Luxembourg University of Applied Sciences, December 2003.

[GRS03b] Nicolas Guelfi, Benoît Ries, and Paul Sterges. MEDAL: A CASE Tool Extension for Model-driven Software Engineering. In *SwSTE'03 International Conference on Software - Science, Technology & Engineering*. IEEE Computer Society, 2003.

[GS02a] Hassan Gomaa and Michael Eonsuk Shin. Multiple-view meta-modeling of software product lines. In *ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, page 238, Washington, DC, USA, 2002. IEEE Computer Society.

[GS02b] Nicolas Guelfi and Paul Sterges. JAFAR: Detailed Design of a Pattern-based J2EE Framework. In *6th Annual IASTED International Conference on Software Engineering and Applications, ACTA Press*, pages 331–337, 2002.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[Gup01] Ankur Gupta. Operation schemas syntactic and semantic analysis using sablecc. `http://lgl.epfl.ch/research/operation-schemas/report/index.html`, 2001.

[GW92] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artif. Intell.*, 57(2-3):323–389, 1992.

[HA04] O. Hummel and C. Atkinson. Extreme Harvesting: Test Driven Discovery and Reuse of Software Components. In *IEEE International Conference on Information Reuse and Integration*, pages 66–72, 2004.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sciences of Computer Programming*, 8:231–274, 1987.

[Har03] Harald Störrle. Semantics of Interactions in UML 2.0. In *International. Workshop on Visual Languages and Formal Methods, at HCC'03*, 2003.

[Her05] Jose Antonio Hernandez. *SAP R/3 Handbook, Third Edition*. McGraw-Hill Osborne Media, 2005.

[HH06] Daqing Hou and H. James Hoover. Using scl to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[HHK+01a] Markku Hakala, Juha Hautamaki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Annotating reusable software architectures with specialization patterns. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 171, Washington, DC, USA, 2001. IEEE Computer Society.

[HHK+01b] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Generating application development environments for java frameworks. In *GCSE '01: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, pages 163–176, London, UK, 2001. Springer-Verlag.

[HHR04] Daqing Hou, H. James Hoover, and Piotr Rudnicki. Specifying framework constraints with fcl. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 96–110. IBM Press, 2004.

[HHV+01] M. Hakal, J. Hautamaki, J. Viljamaa, K. Koskimies, J. Paakki, and A. Viljamaa. Architecture-oriented programming using fred. In *23rd International Conference on Software Engineering (ICSE'01)*. IEEE Computer Society, 2001.

[HJGP99] Wai Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. Umlaut: An extendible uml transformation framework. In *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 275, Washington, DC, USA, 1999. IEEE Computer Society.

[HMPOS04] Øystein Haugen, B. Møller-Pedersen, J. Oldevik, and A. Solberg. An MDA-based Framework for Model-Driven Product Derivation. In *Software Engineering and Applications*, pages 709–714. ACTA Press, 2004.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[HP98] HP. Engineering process summary. Unpublished Technical Report, January 1998.

[HP03] Günter Halmans and Klaus Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 2(1):15–36, 2003.

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.

[HWG00] O. Hollmann, T. Wagner, and A. Günter. EngCon: A Flexible Domain-Independent Configuration Engine. In *Proc. ECAI-Workshop Configuration*, page 94, 2000.

[HWK+06] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor. *Configuration in Industrial Product Families, The ConIPF Methodology.* IOS Press, 2006.

[HWL+02] Marc Hoy, Dave Wood, Marc Loy, James Elliot, and Robert Eckstein. *Java Swing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[IBM] IBM. Websphere website. `http://www-306.ibm.com/software/websphere/`.

[IBM06] IBM. Rational software architect website. `http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html`, 2006.

[IBM07] IBM. Requisite pro website. `http://www-306.ibm.com/software/awdtools/reqpro/`, 2007.

[ICG+04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Rodrigo Oviedo Silva. Documenting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, CMU Software Engineering Institute, April 2004.

[IEE00] IEEE. Recommended Practice for Architectural Description of Software Intensive Systems. Technical Report IEEE-std-1471-2000, IEEE, 2000.

[ION] IONA. Orbix website. `http://www.iona.com/products/orbix/`.

[ISO91] ISO. Software product evaluation: Quality characteristics and guidelines for their use. Technical Report ISO/IEC 9126, ISO/IEC, Geneva, Switzerland, 1991.

[Iso04] S. Isoda. On UML2. 0s Abandonment of the Actors-Call-Use-Cases Conjecture. *Journal of Object Technology*, 4(6), 2004.

[ISO06] ISO/IEC. international standard for Software Process Assessment. Technical Report ISO/IEC TR 15504, ISO, 2006.

[IT04] ITU-T. Message sequence chart. Technical Report Z.120, International Telcommunication Union, April 2004.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, 1992.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *The Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Model Transformations in Practice Workshop at MoDELS*, Montego Bay, Jamaica, 2005.

[JM02] Isabel John and Dirk Muthig. Tailoring Use Cases for Product Line Modeling. In *REPL02*, pages 26–32, September 2002.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76, New York, NY, USA, 1992. ACM Press.

[Jon86] C.B. Jones. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1986.

[Jou05] F. Jouault. Loosely coupled traceability for atl. In *Proceedings of the European Conference on Model Driven Architecture Workshop on Traceability*, Nuremberg, Germany, 2005.

[Kas00] Mohamed Mancona Kande and alfred strohmeier. Towards a UML profile for Software Architecture Descriptions. In *Third UML Conference*, volume 1939, pages 513–527, York, UK, October 2000. LNCS.

[Kaz01] Rick Kazman. Software architecture. In *Handbook of Software Engineering and Knowledge Engineering*, pages 47–68. World Scientific Publishing, 2001.

[KBC05] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language mola. In *Model Driven Architecture*, number 3599 in LNCS, pages 62–76. Springer, 2005.

[KBW03] A. Kleppe, W. Bast, and J. Warmer. *Mda Explained, the Model Driven Architecture: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Professional, 2003.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[KCSS02] M. Kandé, V. Crettaz, A. Strohmeier, and S. Sendall. Bridging the gap between IEEE 1471, an architecture description language, and UML. *Software and Systems Modeling*, 1(2):113–129, 2002.

[KdB04] Marcel Kyas and Frank S. de Boer. On message specification in OCL. In Frank S. de Boer and Marcello Bonsangue, editors, *Compositional Verification in UML*, volume 101 of *entcs*, pages 73–93. elsevier, 2004.

[Kel05] Justin Kelleher. A reusable traceability framework using patterns. In *TEFSE '05: Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, pages 50–55, New York, NY, USA, 2005. ACM Press.

[Ken02] Stuart Kent. Model Driven Engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 286–298, London, UK, 2002. Springer-Verlag.

[Kic96] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[KKT06] Peter Kanuber, Charles Krueger, and Tim Trew, editors. *Third International Workshop on Software Product Line Testing*, volume CSR 003.06. Mannheim University of Applied Sicences - Computer Science Departement, August 2006.

[KMHC05] Soo Dong Kim, Hyun Gi Min, Jin Sun Her, and Soo Ho Chang. DREAM: A Practical Product Line Engineering Using Model Driven Architecture. In *ICITA '05: Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05) Volume 2*, pages 70–75, Washington, DC, USA, 2005. IEEE Computer Society.

[KP03] Jens Knodel and Martin Pinzger. Improving fact extraction of framework-based software systems. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 186, Washington, DC, USA, 2003. IEEE Computer Society.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[Kru02] Charles W. Krueger. Easing the Transition to Software Mass Customization. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 282–293, London, UK, 2002. Springer-Verlag.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Kru06] Charles W. Krueger. New Methods in Software Product Line Development. In *10th International Software Product Line Conference (SPLC'06)*, pages 95–102. IEEE, 2006.

[KRW05] Douglas Kirk, Marc Roper, and Murray Wood. Identifying and Addressing Problems in Framework Reuse. In *13th International Workshop on Program Comprehension (IWPC05)*, 2005.

[KWH04] T. Krebs, K. Wolzter, and L. Hotz. Mass Customization for Evolving Product Families. In *Proc. of International Conference on Economic, Technical and Organizational Aspects of Product Configuration Systems*, pages 79–86, Copenhagen, Denmark, June 28-29 2004.

[KWH05] T. Krebs, K. Wolter, and L. Hotz. Model-based Configuration Support for Product Derivation in Software Product Families. In *Mass Customization, Concepts - Tools - Realization*, pages 279–292. GITO-Verlag, 2005.

[Lam01] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE '01: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE '01)*, page 249, Washington, DC, USA, 2001. IEEE Computer Society.

[Lar02] Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development (2nd Edition).* Prentice Hall PTR, 2002.

[Ler05] Xavier Leroy. The objective caml system release 3.09. Technical report, Institut National de Recherche en Informatique et en Automatique, 2005.

[Let02] Patricio Letelier. A framework for requirements traceability in uml-based projects. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 30–41, Edinburgh, Scotland, 2002.

[LGL02] LGL. The fondue method. `http://lgl.epfl.ch/research/fondue/index.html`, October 2002.

[LGL05] LGL. Fondue builder website. `http://fondue.epfl.ch/`, 2005.

[LM96] James A. Landay and Brad A. Myers. Sketching storyboards to illustrate interface behaviors. In *CHI '96: Conference companion on Human factors in computing systems*, pages 193–194, New York, NY, USA, 1996. ACM Press.

[LS79] Hugh C. Lauer and Edwin H. Satterthwaite. The impact of mesa on system design. In *ICSE '79: Proceedings of the 4th international conference on Software engineering*, pages 174–182, Piscataway, NJ, USA, 1979. IEEE Press.

[MA02] Dirk Muthig and Colin Atkinson. Model-Driven Product Line Architectures. In *2nd Software Product Line Conference*, pages 110–129, San Diego, CA, USA, 2002.

[MCG04] Tom Mens, Krysztof Czarnecki, and Pieter Van Gorp. A Taxonomy of Model Transformations. In *Dagstuhl Seminar Proceedings 04101*, 2004.

[MEG03] Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery. In *STRAW'03 : Second International SofTware Requirements to Architectures Workshop at ICSE'03*, pages 61–68, Portland, OR, USA, May 2003.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[MFJ05] P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS*, Lecture notes in computer science, pages 264–278. Springer, 2005.

[MH05] Alessandro Maccari and Anders Heie. Managing infinite variability in mobile terminal software. *Software: Practice and Experience*, 35(6):513–537, February 2005.

[Mic05] Sun Microsystems. Jsr 220: Enterprise javabeanstm,version 3.0. Technical report, Sun Microsystems, 2005.

[Mic06a] Microsoft. Microsoft .net webpage. `http://www.microsoft.com/net/default.mspx`, 2006.

[Mic06b] Microsoft. Visual studio website. `http://msdn.microsoft.com/vstudio/`, 2006.

[MRP95] Alan Moore and David Redmond-Pyle. *Graphical User Interface Design and Evaluation: A Practical Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.

[MRRR02] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[MvdH03] H. Muccini and A. van der Hoek. Towards Testing Product Line Architectures. *Electronic Notes in Theoretical Computer Science*, 82(6), 2003.

[Mye99] Michael Myers. Investigating information systems with ethnographic research. *Commun. AIS*, 2(4es):1, 1999.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The fujaba environment. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 742–745, New York, NY, USA, 2000. ACM Press.

[NoM07] NoMagic. Magicdraw website. `http://www.magicdraw.com/`, March 2007.

[OC00] A. Ortigosa and M. Campo. Using incremental planning to foster application frameworks reuse. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):433–448, 2000.

[OG06] Erika Mir Olimpiew and Hassan Gomaa. Customizable requirements-based test models for software product lines. In *Third International Workshop on Software Product Line Testing*, pages 17–22. Mannheim University of Applied Sciences, 2006.

[OMG03a] OMG. Common warehouse metamodel (cwm). Technical Report 2003-03-02, OMG, March 2003.

[OMG03b] OMG. MDA Guide Version 1.01. Technical Report omg/2003-06-01, OMG, June 2003.

[OMG04] OMG. Metamodel and uml profile for java and ejb specification. Technical Report formal/04-02-02, OMG, February 2004.

[OMG05a] OMG. MOF 2.0/XMI Mapping Specification, v2.1. Technical Report formal/05-09-01, OMG, 2005.

[OMG05b] OMG. MOF QVT Final Adopted Specification. Technical Report ptc/05-11-01, OMG, 2005.

[OMG05c] OMG. Software process engineering metamodel specification. Technical Report formal/05-01-06, OMG, January 2005.

[OMG05d] OMG. UML 2.0 Infrastructure Specification. Technical Report ptc/05-07-05, Object Management Group, July 2005.

[OMG05e] OMG. UML 2.0 OCL 2.0 specification. Technical Report ptc/05-06-06, Object Management Group, June 2005.

[OMG05f] OMG. UML 2.0 Superstructure Specification. Technical Report formal/05-07-04, Object Management Group, July 2005.

[OMG05g] OMG. Uml profile for corba components. Technical Report formal/05-07-06, OMG, 2005.

[OMG06a] OMG. CORBA Component Model Specification. Technical Report formal/06-04-01, OMG, 2006.

[OMG06b] OMG. Meta object facility (mof) core specification. Technical Report 06-01-01, OMG, January 2006.

[OMG06c] OMG. Omg sysml specification. Technical Report ptc/06-05-04, OMG, May 2006.

[OMG07a] OMG. Unified Modeling Language Infrastructure (version 2.1.1). Technical Report formal/2007-02-04, Object Management Group, February 2007.

[OMG07b] OMG. Unified Modeling Language Superstructure (version 2.1.1). Technical Report formal/2007-02-03, Object Management Group, February 2007.

[O'R05] Tim O'Reilly. What is web 2.0. `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html?page=1`, September 2005.

[Pal97] JD Palmer. Traceability. *Software Requirements Engineering*, 1997.

[Par76] D.L. Parnas. On the Design and Development of Program Families. *TSE*, 2(1):1–9, 1976.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PG96] Francisco A. C. Pinheiro and Joseph A. Goguen. An object-oriented tool for tracing requirements. *IEEE Softw.*, 13(2):52–64, 1996.

[PK04] Claudia Pons and R-D Kutsche. Traceability across refinement steps in uml modeling. In *3rd Workshop in Software Model Engineering at UML*, 2004.

[PM03] Jorge Enrique Pérez-Martínez. Heavyweight extensions to the uml metamodel to describe the c3 architectural style. *SIGSOFT Softw. Eng. Notes*, 28(3):5–5, 2003.

[Pre95] Wolfgang Pree. *Design patterns for object-oriented software development.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Pre00] Wolfang Pree. Hot-spot-driven framework development. In R. J. M. Fayad and D. Schmidt, editors, *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* Wiley & Sons, 2000.

[PS04] Carla Marques Pereira and Pedro Sousa. A method to define an enterprise architecture using the zachman framework. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1366–1371, New York, NY, USA, 2004. ACM Press.

[Pur06] PureSystems. Pure::Variants Website `http://www.pure-systems.com/`, 2006.

[PVJ02] D. Pollet, D. Vojtisek, and J-M. Jézéquel. Ocl as a core uml transformation language. In *WITUML: Workshop on Integration and Transformation of UML models (held at ECOOP 2002)*, Malaga, Spain, 2002.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[RBSP02] Matthias Riebisch, Kai Bllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. In *6th Conference on Integrated Design & Process Technology*, Pasadena, California, USA, 2002.

[RCB98] B. S. Rubin, A. R. Christ, and K. A. Bohrer. Java and the ibm san francisco project. *IBM Syst. J.*, 37(3):365–371, 1998.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints.* PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[RJ01] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, January 2001.

[Roy70] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *IEEE WESTCON*, Los Angeles, CA, USA, 1970. IEEE Computer Society Press.

[Roz97] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations.* World Scientific Publishing Co., Inc. River Edge, NJ, USA, 1997.

[RR02] Jeffrey Richter and Jeffrey Richter. *Applied Microsoft .NET Framework Programming.* Microsoft Press, Redmond, WA, USA, 2002.

[RS04] M. Ryan and P.Y. Schobbens. Fireworks: A formal transformation-based model-driven approach to features in product lines. In *Proc. WS. on Software Variability Management for Product Derivation-Towards Tool Support*, 2004.

[Rus97] Russell R. Hurlbut. A Survey of Approaches for Describing and Formalizing Use Cases. Technical Report XPT-TR-97-03, Expertech Ltd., 1997.

[SA02] Yasunobu Sanada and Rolf Adams. Representing design patterns and frameworks in uml - towards a comprehensive approach. *Journal of Object Technology*, 1(2):143–154, July-August 2002.

[SBS04] Alfred Strohmeier, Thomas Baar, and Shane Sendall. Applying FONDUE to Specify a Drink Vending Machine. *Electr. Notes Theor. Comput. Sci.*, 102:155–173, 2004.

[SC97] Mary Shaw and Paul C. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13, Washington, DC, USA, 1997. IEEE Computer Society.

[SC02] Jean Louis Sourrouille and Guy Caplat. Constraint checking in uml modeling. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 217–224, New York, NY, USA, 2002. ACM Press.

[SC06] Mary Shaw and Paul Clements. The golden age of software architecture. *IEEE Softw.*, 23(2):31–39, 2006.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.

[Sei03] Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, 2003.

[SEI06] SEI. Cmmi®for development, version 1.2. Technical report, Software Engineering Institute, 2006.

[Sen02] Shane Sendall. *Specifying Reactive System Behavior, PhD thesis no 2588.* PhD thesis, Swiss Federal Institute of Technology, 2002.

[Ser99] Giovanna Di Marzo Serugendo. *Stepwise Refinement of Formal Specifications Based on Logical Formulae: from COOPN/2 Specifications to Java Programs*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Département d'Informatique, 1999.

[SG98] Giovanna Di Marzo Serugendo and Nicolas Guelfi. Formal Development of Java Based Web Parallel Applications. In *31st Annual Hawaii International Conference on System Sciences, Software Technology Track*, pages 604–613, 1998.

[Sha84] Mary Shaw. Abstraction techniques in modern programmming languages. *IEEE Software*, 1(4):10–26, 1984.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 136–145, 2006.

[SHTB06] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Feature Diagrams: A Survey and A Formal Semantics (in press). In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Minneapolis, Minnesota, USA, September 2006.

[SK97] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003.

[SK04] S. Sendall and J. Küster. Taming model round-trip engineering. In *Best Practices for Model-Driven Software Development (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications)*, Vancouver, Canada, October 2004.

[SO00] Richard Soley and OMG. Model Driven Architecture. Technical Report omg/00-11-05, OMG, November 2000.

[Som04] Ian Sommerville. *Software Engineering (7th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[SPGB03] Shane Sendall, Gilles Perrouin, Nicolas Guelfi, and Olivier Biberstein. Supporting Model-to-Model Transformations: the VMT approach. In *MDAFA'03*, 2003.

[Spi92] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[SS01] Shane Sendall and Alfred Strohmeier. Specifying concurrent system behavior and timing constraints using ocl and uml. In *UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 391–405, London, UK, 2001. Springer-Verlag.

[SS03] Raul Silaghi and Alfred Strohmeier. Integrating cbse, soc, mda, and aop in a software development method. In *EDOC '03: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 136, Washington, DC, USA, 2003. IEEE Computer Society.

[Sun06] SunMicrosystems. Java Platform, Entreprise Edition (Java EE). `http://java.sun.com/javaee/index.jsp`, 2006.

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[Szy98] Clemens Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[Tel07] Telelogic. Doors website. `http://www.telelogic.com/products/doors/index.cfm`, 2007.

[TH03] Jean-Christophe Trigaux and Patrick Heymans. Modelling variability requirements in Software Product Lines: a comparative survey. Technical Report EPH3310300R0462 / 215315, FUNDP, November 2003.

[TOHSMS99] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[Tou02] T. Tourwé. *Automated Support for Framework-Based Software Evolution*. PhD thesis, Vrije Universiteit Brussel, 2002.

[vdL02] Frank van der Linden. Software product families in europe: The esaps & caf projects. *IEEE Software*, 19(04):41–49, 2002.

[VdLM02] H. Vangheluwe, J. de Lara, and P.J. Mosterman. An Introduction to Multi-Paradigm Modelling and Simulation. In *AIS2002*, pages 9–20, 2002.

[vdML02] Thomas van der Maßen and Horst Lichter. Modeling Variability by UML Use Case Diagrams. In *International Workshop on Requirements Engineering for Product Lines*, pages 19–25, September 2002.

[vGBS01] Jilles van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.

[Vil03] Jukka Viljamaa. Reverse engineering framework reuse interfaces. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 217–226, New York, NY, USA, 2003. ACM Press.

[VJ04] Didier Vojtisek and Jean-Marc Jézéquel. MTL and Umlaut NG: Engine and Framework for Model Transformation. *ERCIM News*, 58, 2004.

[vO02] Rob van Ommering. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 255–265, New York, NY, USA, 2002. ACM Press.

[Whi02] Jon Whittle. Transformations and software modeling languages: Automating transformations in uml. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 227–242, London, UK, 2002. Springer-Verlag.

[Wit96] James Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, ADA 315653, Software Engineering Institute, 1996.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language (Second Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[WKHM04] K. Wolter, T. Krebs, L. Hotz, and T.D. Meijler. Knowledge-based Product Derivation Process. In *Proc. of the IFIP 18th World Computer Congress TC12 First International Conf. on AI Applications and Innovations (AIAI2004/WCC2004)*, pages 323–332, 2004.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[ZA05] Uwe Zdun and Paris Avgeriou. Modeling architectural patterns using architectural primitives. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 133–146, New York, NY, USA, 2005. ACM Press.

[Zac87] John A. Zachman. A framework for information systems architecture. *IBM Syst. J.*, 26(3):276–292, 1987.

[ZG03a] P. Ziemann and M. Gogolla. Ocl extended with temporal logic. In M. Broy and A. V. Zamulin, editors, *Perspectives of System Informatics, Novosibirsk, July 9-12*, 2003.

[ZG03b] Paul Ziemann and Martin Gogolla. An OCL extension for formulating temporal constraints. Technical Report 1/03, Universität Bremen, 2003.

[ZHJ03] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a UML Profile for Software Product Lines. In *PFE2003: 5th International Workshop on Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 129–139, Siena, Italy, November 2003. Springer.

[Zia04] T. Ziadi. *Manipulation de Lignes de Produits en UML*. PhD thesis, IFSIC, Université de Rennes 1/IRISA, 2004.

[ZIKN01] A. Zarras, V. Issarny, C. Kloukinas, and K. Nguyen. Towards a Base UML Profile for Architecture Description. In *1st ICSE Workshop on Describing Software Architecture with UML*, pages 22–26. IEEE/ACM, 2001.

[Zim80] H. Zimmermann. Osi reference model – the iso model of architecture for open systems interconnection. *IEEE TRANSACTIONS ON COMMUNICATIONS*, 28(4):425–432, 1980.

[ZJ06] T. Ziadi and Jean-Marc Jézéquel. Product Line Engineering with the UML: Deriving Products. In *Families Research Book*. Springer, 2006.

[ZSPMK02] A. Zisman, G. Spanoudakis, E. Perez-Minana, and P. Krause. Towards a Traceability Approach for Product Families Requirements. In *Proceedings of 3rd ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications*, Orlando, USA, May 2002.