ICESI UNIVERSITY

MASTER IN INFORMATION TECHNOLOGY AND TELECOMUNICATIONS

SCHOOL OF ENGINEERING

QUALITY-DRIVEN SOFTWARE PRODUCT LINES

MEMBERS
DAVID DURÁN GIRALDO, S. Eng.

ADVISOR
HUGO ARBOLEDA, PhD.

Santiago de Cali, January 2015

TABLE OF CONTENTS

List of Figures

ABSTRACT

In software product line engineering, the customers mostly concentrate on the functionalities of the target product during product configuration. The quality attributes of a target product, such as security and performance, are often assessed until the final product is generated. However, it might be very costly to fix the problem if it is found that the generated product cannot satisfy the customers' quality requirements. Several approaches have been proposed to deal with this issue, focusing on the assessment of a quality attribute of a product configuration to measure the impact on a quality attribute made by the set of functional variable features selected in a configuration. Nevertheless, these approaches are only interested in characterizing the relationships among quality attributes and product functionalities to provide useful information about predicting the quality of the target product, relying on the previous existence of the software components that provide such measures and values. Our approach provides a SPL that uses model-driven techniques to automate derivation of product line members, considering promotion of quality attributes during this process by means of software enterprise patterns. In concrete we provide the following contributions: *i)* a domain metamodel that enables defining functional scope of product line members, *ii)* a quality attributes variability model to handle definition of quality scope of product line members, *iii)* a Reference Architecture (characterization of software enterprise design patterns from the perspective of the quality attributes they promote or inhibit) to construct product line members that exposes explicit variation points related to quality attributes and their relationships with functional features, *iv)* and tool support based on a generation engine to automatically construct product line members, following the Reference Architecture constraints. An illustrative example based on a Project Management software product line is presented to demonstrate how the proposed approach works.

1. INTRODUCTION

Today software engineers are faced with a demand for complex and powerful software systems, which must be developed in short, time. To solve this problem, software reuse was emerging as a principal key to a successful software development because of its potential to reduce the time to market, increase quality and reduce costs [1], as it consists in creating and in assembling systems with existing components. Software Product Line Engineering (SPLE) is an expanding approach, which aims at developing a set of software systems that share common features and satisfy the requirements of a specific domain [2]. While having much in common, products derived from a SPL still differ in certain requirements, design decisions, and implementation details. The variability stems from many sources such as customer's specific needs, mutability of the environment, system maintenance and evolution, and so on. Product Lines are gaining importance in the software development field as they reduce development time, effort, cost and complexity and increase quality of products [3].

One of the challenges in today's both traditional software engineering and SPLE approaches is to deliver high-quality software on time to customers [4]. Successful companies must have a focus on customer satisfaction and software quality to ensure that the desired quality is built into the software product and that customers remain loyal to the company. This is especially true for the IT industry where customers have ever-increasing expectations of software quality. Software quality has become a major concern of software organizations [5]. A lot of research thus has been done to refine the concept of quality into a number of quality attributes (QAs), also known as quality characteristics, quality factors or non-functional requirements (NFRs), (see e.g., [6] [7] [8] [9]).

QAs of the products, such as performance, and security are usually handled until the final product is produced and tested in the system-testing phase [10]. Different members of the software product line may require different levels of quality attributes. For example, one product may require a very high security whereas in another product security is not that important. If it is found that the quality attributes of the product fail to meet the customers' requirements in a later product development stage, it is costly to fix the problems. Therefore, the QAs of a target product (level of accomplishment for each QA) should be assessed as early as possible in the product development process.

Although there are studies that mention the existence and influence of QAs on the domain analysis and design (e.g., [11] [12] [13]), they do not consider their influence on SPL assets implementation. Zhang *et al.* [14] propose a Bayesian Belief Network (BBN) based approach to explicitly modeling the impact of variants (especially design decisions) on system quality attributes. Zhang G. [15] proposes an Analytic Hierarchical Process (AHP) based approach to estimate the relative importance of each functional variable feature on a quality attribute. Bartholdt et *al.* [16] presents an integrated tool-supported approach that considers both qualitative and quantitative quality attributes without imposing hierarchical structural constraints. Even though these works explicitly consider QAs variations and their relationships with functional features, they focus on characterizing such relationships to provide useful information about predicting the quality of the target product, relying on the previous existence of the components that provide such measures and values. Thus, the design decisions and implementation details needed to construct these components in order to promote the configured quality levels are not considered.

Such condition exposes limitations on current SPL approaches regarding strategies that systematically make use of good design practices to promote quality attributes in the core assets of the line, i.e. software components. More particularly, although there are several repositories that encompass the knowledge regarding how to design concrete architectures and components of a given application domain, taking into account quality concerns, i.e. Reference Architectures, current SPL approaches do not use a systematic mechanism to take advantage of this consolidated expertise to modify software component's design structure and behavior, in order to promote different levels of quality.

Our contribution in this work is to develop a strategy and tool support that, making use of software design good practices, allows a product line engineer to automatically derive products that are configured based on a set of functional and quality constraints. Specifically, we provide the following contributions: *i)* a domain metamodel that enables defining functional scope of product line members, *ii)* a quality attributes variability model to handle definition of quality scope of product line members, *iii)* a Reference Architecture (characterization of software enterprise design patterns from the perspective of the quality attributes they promote or inhibit) to construct product line members that exposes explicit variation points related to quality attributes and their relationships with functional features, *iv)* and tool support based on a generation engine to automatically construct product line members, following the Reference Architecture constraints. An illustrative example based on a Project Management software product line is presented to demonstrate how the proposed approach works.

2. BACKGROUND

2.1 QUALITY ATTRIBUTES

Quality is the degree to which a system meets the Non-Functional Requirements (NFRs) in the context of the required functionality. Quality Attributes (QAs) are crosscutting-concerns known as nonfunctional properties of a software system such as performance, safety, and security [17], [18]. Achieving QAs must be considered throughout the development process of a software system. According to Bass *et al.* [19], there are three problems related to QAs: the definitions provided for an attribute are non-operational (lack of preciseness), there is no clarity on which quality a particular aspect belongs to (overlapping attribute concerns) and each attribute community has developed its own vocabulary. A solution to the first two of these problems is to use quality attribute scenarios [20]. A solution to the third problem is to use a standard model that specifies each attribute underlying concerns, like ISO 25000 series [21].

A quality attribute scenario (QAS) serves as a mean of characterizing quality attributes, and consists of six parts [22]: *i)* the source of stimulus, which is the entity that generates the stimulus, *ii)* the stimulus, which represents an internal or external incentive that affects a part of the system and acts as a trigger e.g. a user invokes a function, *iii)* the environment, that represent the conditions under which the stimulus occurs, e.g. at runtime, *iv)* the artifact that identifies the system or a part of it that is stimulated, *v)* the response, which is the action that's undertaken when the stimulus arises, and *vi)* the response measure, that provides numeric indicators so the quality attribute can be tested. An example of a QAS can be: *An end user requests the system to retrieve data from a particular table stored in a local database. Such retrieval must take 4 seconds tops to display the information to the user.*

2.2 SOFTWARE ARCHITECTURE

Software Architecture (SA) can be defined as the set of structures needed to reason about the software system, which comprises the software elements, the relations between them, and the properties of both elements and relations [23]. The importance of SA is that it serves as a blueprint that details the system that is going to be developed [24]. It must provide an alignment between user, business and system goals [25]. One of the most relevant contributions of SA is its role as primary carrier of system qualities such as performance, modifiability and security [22], [24]. It involves a series of decisions based on a wide range of factors that have considerable impact on QAs that decide the overall success of applications.

Authors like Hollingsworth [26] and Vogel *et al.* [27] have found that to adequately reason about specific properties, the software architecture field must be divided into micro and macro architectural levels. Macro-architecture deals with top-level/high-level design issues, for instance, the spectrum to which architecturally relevant elements are assigned. It covers aspects such as requirements, decisions and structures at a high level of abstraction, for example, decisions with regard to important system interfaces, identification of system's main building blocks and the relationships among them [27]. While macro-architecture deals with system's overall structure [28], such as viewpoints, architectural styles and patterns, micro-architecture focuses on detailed structure and behavior of system's components.

Alur *et al.* [29] highlight that micro-architectures are building blocks upon which we build applications and systems, and that they can be seen as a prescriptive solution that uses different design patterns to solve larger problems concerning macro-architecture decisions. Micro-architecture is also known as software component engineering [26], which is closely related to the notion of reuse [30]. Some of the advantages of this particular area are the focus on component's interoperability and the assurance of component's promoted properties through proper design decisions. Such decisions focus on promoting quality attributes on each developed component, which provides a well founded and proven base to deal with system's composition at macro-architectural levels, fomenting the construction of applications under a quality approach.

Enterprise Software Applications (see section 2.3) usually are developed using proven architectural styles, such as the three-layer based architecture [29], which we are going to use as the fundamental structure to derive the products of our line. An architectural style defines a set of types of elements, types of relationships and constraints between them

[22]. Software architects designing a solution using a style, must design the software using the style-specific types of elements and relationships, and respect the corresponding constraints.

The Three-layer architectural style [29] [31] defines three layers: the presentation, logic and data-access layers. An application designed using this style includes these three layers. In addition, each layer comprises components with layer-specific types and responsibilities: the presentation layer comprises GUI components; the logic layer includes the components that implement the logic behind the business transactions; and the data access layer includes data-access components that store and retrieve data from files, servers and databases.

2.2.1 TACTICS

Knowing SA's role as quality insurer, it is important to rely on mechanisms that aim to guarantee software quality on both macro and micro levels. Such mechanisms rely on design decisions and are known as tactics. Clements *et al.* [32] and Rozanski *et al.* [33] determine that a tactic is a widely used architectural approach that has proven to be useful to achieve a particular quality. For example, "rollback" is a tactic to recover from a failure aiming to increase a system's availability, or "concurrency" is a tactic to manage resource access aiming to improve performance. Among these tactics, Design Patterns [34] are a well-know mechanism to achieve QAs at a micro level, i.e. component's internal structure.

According to Gamma *et al.* [34], a design pattern is a recurrent situation that must contain four essential elements: a *pattern name*, the *problem* that determines when to apply the pattern, the *solution* that states how the elements provided by the pattern should interact and function to solve the problem, and the *consequences* that take place when the pattern is applied, such as results and trade-offs. Design patterns catalogs must contain a set of design patterns that are useful in a particular context. Every pattern contained in the patterns catalog must contain at least these four elements to describe its use. There are many patterns catalogs in the literature. Steel *et al.* [35] and Hafiz M. [36] provide catalogs that focus on improving applications security. Fowler M. [31] provides a list of design patterns used in ESAs.

Bien's book [37] provides a set of design patterns that result as an evolution from previous patterns specified in [29]. These patterns are intended to be use in the context of ESA developed using Java Enterprise Edition-JEE [38]. The book maintains tiers division provided in [29] and elaborates on the details and main changes of deprecated patterns and newly ones. Each pattern contains the four elements described by Gamma *et. al*, plus a section that highlights non-functional attributes that are affected by them. It also proposes two configurations to properly use the patterns catalog depending on application objectives: *service oriented* and *domain-driven* architectures. Each configuration summarizes a pattern language that declares some constraints to include or exclude different patterns. Bien's 2012 book [39] offers and update to his earlier catalog in terms of new discovered patterns and modifications to some previous patterns intends and uses. It provides an extensive set of enterprise patterns that meet the following characteristics:

i) are currently used in real enterprise applications, *ii)* provide the basic elements listed by Gamma *et. al.*, *iii)* each pattern briefly describes how it affects particular quality attributes and *iv)* the catalog is focused to be used in enterprise applications, which is our context of work.

Adam Bien is a well-known java developer and architect that have participated in the edition of several books related to Java technology [40]. He holds numerous titles that certificate his expertise in using Java for enterprise IT projects, between them Java Champion [41] and Java Developer of the year 2010 [42]. His contributions vary from the organization of Java related workshops [43], two books about the use of Java in the development of enterprise applications (the referred above among them), a constantly updated blog about Java topics and implementation issues of his proposals [44], and a TV channel addressing the same concerns [45]. Bien also provides a repository[1] that consolidates a collection of implemented samples and reusable templates, which demonstrates patterns, approaches and architectural ideas for the Java EE 6 platform.

Besides design patterns, there are several other tactics that attempt to ensure quality attributes in enterprise applications. For instance, Kalinsky D. [46] list a set of design patterns for high availability that can be applied to system's infrastructure, like hardware redundancy. Microsoft [47] provides server-clustering pattern to address performance, scalability and availability of enterprise applications. Specifically, load-balanced cluster allows distribution of network traffic between several physical servers to improve application's performance, while fail over cluster proposes hardware redundancy to deal with server damage and unavailability.

2.2.2 *Reference Architectures*

RAs go one step further in reuse of best practices in architectural design [48], [32]. Nakagawa *et al.* [49] define a RA as: "*an architecture that encompasses the knowledge regarding how to design concrete architectures of systems of a given application domain; therefore, it must address the business rules, architectural styles (that address quality attributes in the reference architecture), best practices of software development (for instance, architectural decisions, domain constraints, legislation, regulations, and standards), and the software elements that support the development of systems for that domain. All of these must be supported by a unified, unambiguous, and widely understood domain terminology*". RAs importance is that they provide consolidated information about a particular domain to serve as guideline to build specific products taking into account best practices and quality constraints.

RAs become a main asset to develop software applications, because they provide consolidated information of the most relevant components of a particular domain. A RA provides structure, identification and relationships of the main components of a particular software architecture that can be reused to build a concrete software application, serving as support at a macro architectural level. This information may be expressed in terms of architectural styles or patterns using a standard modeling language like UML. RAs must

[1] https://kenai.com/projects/javaee-patterns

also contain information related to quality attributes, that is, tactics. Such information may vary from infrastructural decisions to software design decisions. Software decisions include definition of guidelines, standards and templates to address common problems of the related domain. The use of design patterns to describe component's internal composition and behavior (micro-architecture) is also an important contribution that RAs provide, as they consolidate relevant information to develop components aligned to inherent domain quality attributes.

As stated before, there are several elements that compose a RA, so it is important to have means to adequately represent a RA, because they are required to be understandable for wide variety of stakeholders (such as customers, product managers, project managers, and engineers). Nakagawa [50] summarizes several works that have focused on the properly representation/description of RA; these methods include semi-formal techniques of UML (Unified Modeling Language) [51], and ADL (Architectural Descriptions Languages) and their extensions [52].

2.3 ENTERPRISE SOFTWARE APPLICATIONS

The context or domain of this work is focused on Enterprise Software Applications (ESAs). These types of applications are intended to satisfy the needs of entire organizations. In [31], Fowler M. identifies that an ESA usually involves persistent data, concurrent user access to the information and several user interfaces to handle the big amount of data requested. This type of software requires abstraction and modeling of how organizations work; besides, it requires development tools that support such model in order to build unique appropriate applications that match organization's needs. ESAs must accomplish a certain set of characteristics, for instance, [53] defines ESAs as network applications that must be large-scale, multi-tiered, scalable, reliable, and secure. These non-functional characteristics ensure ESAs quality. Bass *et al.* [22] agree that this type of applications must be oriented to a web-based environment (network apps) and that they must fulfill a minimum set of quality attribute requirements to ensure quality, between them scalability, availability/reliability, security, usability and performance. Sections below describe the domain metamodel that we use to limit the scope of particular ESAs that we are interested in.

There are standards that emphasize on providing an environment that ensures all of these characteristics, allowing developers to focus on relevant business information, such as its logic and functionality. Java Enterprise Edition [38] is one of these solutions. It provides latest technologies integration and support to maximize web based enterprise applications development and management. There are other solutions like JBoss Application Server [54] that also provides integrated tools to ease enterprise applications management. The selection of one these solutions depends on organization's specific needs and technical knowledge.

2.4 MODEL DRIVEN ENGINEERING

As technology evolves, several platforms for software development have been developed. These platforms are usually heterogeneous, that is, despite offering similar interfaces, each one has its own operational standard and a specific set of base components. This implies that it is mandatory to know the characteristics of the selected platform prior the software development process, in order to use the basic functionality it provides. Hence, once the new application components are built, this software will only run on the selected platform and won't be compatible with other technologies.

As a concrete example of the above situation, we find Java and .NET platforms, which provide a set of basic components that facilitate application development. These basic components provide functions or tasks that simplify software development. However, once an application is built using the basic components of any of the two platforms, it can only be executed on the selected platform. This implies that an application developed under Java platform cannot run on .NET platform and vice versa.

Model Driven Software Development (MDSD) represents a new approach in software engineering that deals with the inability of third generation languages to manage heterogeneous platforms, plus it provides effective expression of domain concepts [55]. MDSD does not attempt to solve the problem of heterogeneity by unifying platforms. Its goal is to support software development independently of the technology used. Thus, decisions of platform selection and implementation details are postponed to final stages of the development process. In order to achieve his goal, MDSD raises the level of abstraction, using models as first-class elements, i.e. assets that can be processed by a computer or by a tool. This is the main feature that differentiates MDSD from traditional approaches of software development where models are used exclusively for documentation purposes.

Models deal with a high degree of abstraction, representing the concepts that are relevant to a particular domain. This enables focusing in the representation of the problem rather than its implementation, reducing software development complexity through the separation of different concerns in multiple views. Thus, MDSD can express both the problem and the solution across different models, each one representing a specific point of view, thereby reducing the complexity of developing and managing heterogeneous platforms, since the software can be expressed in terms of domain concepts.

In order to increase the level of abstraction, each model must be defined in terms of a specific language. In general, these languages are defined in terms of domain particular concepts without considering implementation details. The definition of a language involves abstracting its domain concepts, a proper notation and rules that must be met. Thus, a model is constructed in terms of the language of its metamodel, implying that the metamodel is responsible for abstracting these domain concepts and rules. In conclusion, metamodels are models that represent concepts of a domain and the relationships between these concepts. The relationship between a model and the metamodel is known as a conformity relationship [56]. Thus, it is said that a model conforms to a metamodel if it meets the description and restrictions of the metamodel.

As explained above, MDSD uses models as first-class elements, i.e. assets that can be processed by a computer or a tool. In order to process these models, it is necessary to use languages that enable specifying the required inputs; the operations performed and the output elements. The specification of these operations is known as model transformation. Transformations between models are composed of transformation rules, which are defined in terms of the domain concepts (metamodel) involved in the transformation [57]. Each transformation rule defines a set of inputs, a set of operations and a set of output elements. Both inputs and outputs may be specified in source and target models of the transformation. There are different kinds of transformations classified by the nature of their source and target domains. Most common transformations are: model-to-model and model-to-text. Former implies turning a source domain model into a target domain model, latter enables using a domain model to create text. This type of transformation is generally used to generate source code and configuration files.

2.5 SOFTWARE PRODUCT LINE ENGINEERING

Product line engineering is a paradigm that places component production and the reuse strategy at the center of the development process [58], [59], intended to allow the development of several applications that share common aspects. SPLE put into practice this paradigm in software context. This discipline is divided in two main phases, named domain engineering and application engineering [60]. Former phase identifies domain applications, distinguish between their similarities and variations and structures this information. Latest phase consist of the efficient production of the applications defined in the first stage, using a production plan as a guideline to successfully configure each application.

Domain engineering phase deals with identifying and documenting SPL scope, which is known as domain, plus it must deal with variability management. The first activity is the identification of the commonalities of the SPL domain, which are the characteristics that will be shared by all the products of the product line. This stage also has to deal with the identification of variation points, which are the specific points of the SPL domain that allow derived product's customization. Each variation point requires defining all its possible variants. A variant is a characteristic that isn't necessary contained in every derived product of the line. An asset known as the *variability model* consolidates identified characteristics and defines their relationships. There are several notations to define the variability model; between them, [61], [62] and [63] propose feature models (FMs); [64] suggest enriched UML diagrams; [65] add model-engineering notations; [66] propose a variability language; [60] presents Orthogonal Variability Model (OVM).

Once the variability model is defined, a specification shared by all domain products must be set. The product line architecture (PLA) is an asset preplanned to support their common basis, variations and architectural requirements. It must capture the entire set of features of the SPL, its variability model, and must define assembly rules and constraints (using a formal notation) that allow the generation of particular products. According to Clements P. [59] and Bass *et al.* [19] the PLA is the original artifact of the reusable artifact kernel, and thus must be a main development objective.

There is some controversy on the subject of differences between the concepts of RA and PLA. Most authors, including [67], [68] see no real differences between the two, but others see differences in the abstraction level and domain type. For instance, Nakagawa *et al.* [69] establishes a relationship between this two concepts: *while reference architectures deal with the range of knowledge of an application domain, Product Line Architectures (PLA) are more specialized, focusing sometimes on a specific subset of software systems of a domain and providing standardized solutions for a smaller family of systems.* This means that a RA can be specialized into many PLAs. Angelov *et al.* [70] agree with the previous definition stating that product line architectures are less abstract than RA, but more abstract than concrete architectures.

Variability management must also allow construction of product line members. Approaches like [67] state the use of decision models [71], [72], [73], [74] to capture external variability and define the concrete resolution during the derivation of products [72]. Each decision maps each variation point defined in a variability model with its possible resolutions (i.e. components), in order to define how core assets must be adapted and assembled, aiming to realize corresponding related variation.

Application engineering phase focuses on configuration and derivation of product line members; according to the main assets produced in domain engineering stage and a production plan [67]. The configuration consists of selecting a consistent and full set of variants from the variability model. Product derivation deals with the manual or automatic activities that constructs the final product from a functional configuration, reusable elements, and in accordance with the production plan. There are several approaches that focus on the automation of product creation from decision and resolution models. Dhungana *et al.* suggest DOPLER, an approach entirely based on decision models for deriving the products in a line [75]. The AMPLE project [76], with the TENTE approach, has proposed a quasi-automatic derivation of the product line from the features model and a features-oriented language. Lastly, model-oriented approaches such as [77] and [78] consider methods for expressing the use of reusable elements in the realization of a selection of features.

Voelter *et al.* highlight in [3] that it is possible to build product lines of product line architectures. They refer to this property as Meta-Product Lines. This means that the domain models that result from a particular domain metamodel are used as variability models to configure independent SPLs i.e. each domain model enables configuration of several products by selecting/deselecting domain concepts that are wanted on a target product; instead of being used as particular products i.e. each domain model is a target product. In this work we provide a domain metamodel that is used as a Meta-Product Line for constructing ESAs, within the scope of domain concepts and relationships defined in it.

2.6 RELATED WORK

Due to there is a wide range of QAs that can be considered in SPL development, i.e. ISO's 25000 classification, addressing all of them might end up in several interrelated constraints and conditions, making the derivation of products a difficult and error-prone task to achieve. Thus, it is important to identify the most frequent QAs that the community is interested in. Works like [15] [14] [79] identify performance, usability, security and cost as the quality attributes that are of frequent interest to final users.

After identifying these QAs, a proper way to model them (contemplating their variations) is needed. The idea of variations in quality is considered in different works [11] [12] [13] and there are also some approaches that address variability modeling taking into account quality attribute variability. However, "there is a lack of thorough understanding of existing approaches to be able to integrate quality attribute variability as a part of the systematic variability management of software product lines" as proposed by [11].

A survey presented in [80] summarizes several existing methods that address quality attribute variability specification and managing. Approaches like Goal-based model [81] propose to treat and model non-functional requirements or QAs as soft goals, so they represent conditions or criteria that the system should meet. Benavides et al [82] [83] propose to extend feature model to deal with extra-functional features. They propose a notation extending feature models with attributes, characteristics of a feature that can be measured such as availability, cost, latency, bandwidth and relations among attributes. Work in [15] uses feature models to represent QAs, where the leaf nodes of such quality tree determine the quality that the SPL will assess, because they have sufficient semantics for the impact-relationships between quality attributes and functional variable features analysis. This decision is due to quality attributes that are represented at high levels in a feature model are often vague and inherently hard to measure, such as performance or security.

Work presented in [15] highlights the importance of dealing with quality issues in early stages of the product development process, such as considering impact of quality constraints to the final products from the start. Besides gathering several approaches for specifying variation in quality attributes in SPL, Etxeberria *et al.* [80] emphasize on the techniques that they use to relate functional variability with quality variability. Goal-based models use correlations to represent the links among functional and soft goals. Each correlation links is marked with an influence qualitative label (++, --) that is converted to a qualitative value to be used in a later quality analysis. Zhang et al. [14] proposed a Bayesian Belief Network (BBN) to capture the impact of functional variants on quality attributes. They link functional requirements to quality attributes using noted definitions that are relative to each domain, i.e. "high performance" definition might mean a response time lower than 0.5 seconds to one domain and less than 1 seconds to another. After all links are defined, conditional probability is used to quantify the conceptual relationships. Notice that these models assume that the quality levels are affected by the functionalities selected, and not in the opposite direction.

Several works consider QAs in SPL development. For instance, Huerta *et al.* [84] provide support, applying model-driven engineering principles, to the identification and representation of non-functional requirements (NFRs) in SPL development, also offering a mechanism for the validation of their fulfillment through the association of measures, thresholds and OCL constraints to each NFR. They provide a meta-model that allows definition of NFRs based on ISO 25000's [21] quality model to define measures for a specific quality attribute, besides the impact that such attribute might have on features or core assets (i.e. positive/negative). Their model also allows definition of variability for a particular QA, i.e., performance may have different acceptance thresholds depending on system's configuration. They determine that a NFR may be related to many QAs, for instance, reliability can be broken down into two QAs: availability and fault tolerance. Each NFR must define a restriction that determines its achievement; such restriction is defined using OCL. Once NFRs are defined and related, an automatic OCL validation is performed taking as input this configuration model to check whether the different constraints are satisfied or not. Such validation concludes whether a product/artifact promotes or inhibits the associated NFRs. This work focuses on providing means to adequately relate and measure NFRs impact on features/artifacts; thus, they rely on the specification of such measures to operate. They do not care about component's design decisions to determine how they affect QAs, nor provide mechanisms to automate their generation.

Gürses expresses NFRs in [85] through qualitative goals. This approach allows performing trade-off selections among different types of basic NFR-goals, returning the configuration that fulfills those goals. The modeling of NFRs is done using an extended feature model that is annotated with quality information. Sagardui *et al.* defined a method for capturing the quality variability and the relationships among functional variability and quality aspects through an extended feature model that allows the expression of quality attributes, their variability and the relationships (impacts) that features or feature groups have on the quality attributes [86]. Both proposal focuses on modeling variability of quality attributes and relating such variability to functional characteristics of the SPL. They also provide means to evaluate accomplishment and trade-offs between quality attributes for a particular configuration of the SPL. However, their proposals only introduce quality considerations to SPLE, hence they do not focus on how such identified and related QAs are going to be fulfilled, i.e. using design patterns for their construction.

Work in [14] proposes a Bayesian Belief Network based approach to address the problem of analysis and prediction of quality attributes for a product line. They use BBN to capture the design knowledge and experiences of domain experts. Such approach enables graphically modeling the impact of design decisions on quality attributes, by using nodes and relationships among them. After capturing the qualitative relationships among variables (denoted by nodes), a quantification process takes place, where a conditional probability is assigned to each node in the BBN. Having quantified the graphical model, a quantitative analysis can be performed (e.g., predicting the quality of the target system). The proposal in [15] uses an Analytic Hierarchical Process (AHP) based approach to estimate the relative importance of each functionality on a quality attribute. They conclude that based on the relative importance value of each functionality on a quality attribute, the level of quality attributes of a product configuration in software product lines can be assessed. To do so, they identify the relevant functionalities that impact a quality attribute; they estimate the relative importance value of functionality identified on a quality attribute, they calculate an importance value for a product configuration on a quality attribute (e.g. 1 for Equal Importance: Two elements contribute equally to the objective, and 7 for Very Strong Importance: One element is favored very strongly over another), and they define a representation scheme for quality attributes in feature models that enables measuring quality levels for a particular product. [16] provides an integrated tool-supported approach with both qualitative and quantitative quality attributes that are explicitly considered in the product derivation process without imposing structural constraints such as a hierarchical structure. This tool enables configuring a product relating each functionality to the quality attributes it promotes or inhibits. These relationships are quantified in order to enable calculating the resulting value of a quality attribute for a particular product, to determine weather the product satisfies the quality needs of the customer. Even though these works explicitly consider QAs variations and their relationships with functional features, they focus on characterizing such relationships to provide useful information about predicting the quality of the target product, relying on the previous existence of the components that provide such measures and values. Thus, the design decisions and implementation details needed to construct these components in order to promote the configured quality levels are not considered.

Dealing with code generation, there are several technologies to manage templates, e.g., Xtend2[2], Acceleo[3]. Former is designed as a successor of Xpand. However, it is not tailored specifically to code generation but as a general-purpose language that is nicely usable for code generation as well. The use of Xtend2 demands providing an engine (source code) that drives the generation, although it is possible to use Modeling Workflow Engine for this purpose. Latter is more tailored for simple code generation. Its syntax is based on an OMG specification for code generation, and provides a full-featured IDE for developing code generation. Difference between these tools is that Acceleo is limited to work only with particular models (EMF), while Xtend2 permits using other data sources, providing an easy way to call any Java code available.

3. *CASE STUDY*

To illustrate our approach, this section presents a case study on a SPL of enterprise software applications, a product line with functional and quality variability. Given that this SPL must support configuration of several product line members (SPLs) to derive different ESAs, we elaborate on the description of a SPL for Project Management systems. This description serves as an exemplification of the kind of software product lines that can be derived from the SPL of enterprise software applications.

The SPL for Project Management systems supports a variety of functionalities for management of projects, risks and users. In addition, it supports variations in the quality attributes each product must exhibit in order to offer different products to medium and big companies. Following, we will describe its functional variability, which is a set of optional and mandatory functionalities that the SPL offers. This SPL must always allow the authentication of users against the system, using a login and a password. Once the user is authenticated, the product line might provide the following super set of functionalities to users:

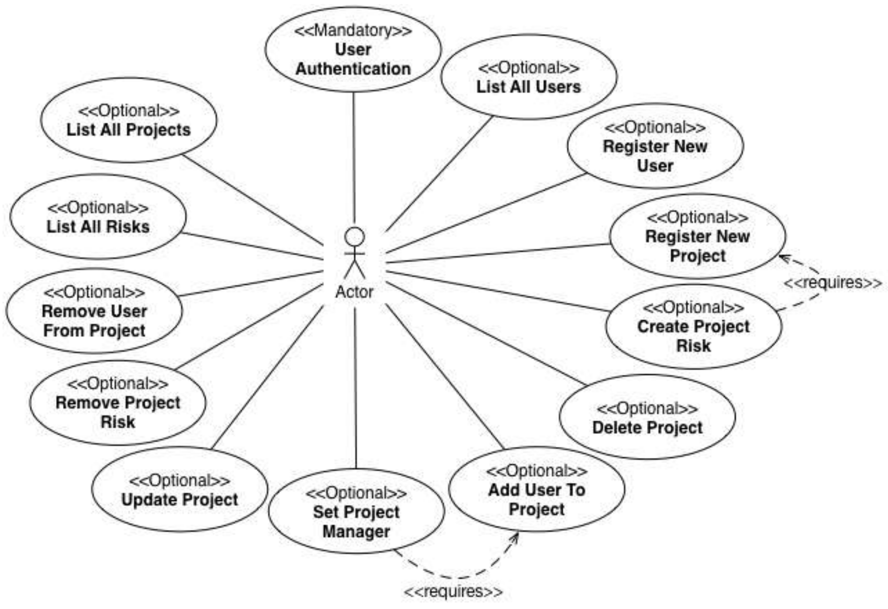
- List all the users registered in the system
- Register new users in the system

[2] http://eclipse.org/xtend/
[3] https://eclipse.org/acceleo/

- Register new projects in the system
- Update the information related to a project of the system
- Delete projects from the system
- Add users to projects, in order to indicate the members of each project
- Remove users from projects, in case a user is no longer a member of a project
- Create project risks. Notice that risks existence depends on a project's existence
- Remove risks from projects
- Set a project manager from the users related to a project. Notice that this functionality requires the addition of users to projects
- List all the risks of a project
- List all the projects registered in the system

Every project must have an id (serial number), a name, a description and a start date. Each registered user must have an identification number, a name, a cellphone number and a password. To identify risk, each one must provide an id (number), a name, a description, an impact (decimal), a probability (decimal) and the identification of the project it belongs to.

These functionalities are usually modeled as use cases that can be included in the application during application engineering. These use cases can be modeled [87] [88] including stereotypes such as <<*Mandatory*>> and <*Optional*>> to represent if the use case must be included in all the products or might be included in one particular product. We also use the <<*requires*>> stereotype to indicate that the selection of a use case demands the selection of the use case it requires. Figure 1 shows the functional variability of the SPL for Project Management systems in terms of use cases. The <<*requires*>> dependency between "*Create Project Risk*" and "*Register New User*" use cases indicates that a risk cannot be created unless the project that it belongs to is created first. The <<*requires*>> dependency between "*Set Project Manager*" and "*Add User To Project*" means that a project can only set its manager as long as it has users associated to it. Thus, the project manager has to be one of these related users. Notice that many products for Project Management might be configured, by selecting several optional use cases.



<<Mandatory>> User Authentication
<<Optional>> List All Users
<<Optional>> List All Projects
<<Optional>> Register New User
<<Optional>> List All Risks
<<Optional>> Register New Project
<<requires>>
<<Optional>> Remove User From Project
Actor
<<Optional>> Create Project Risk
<<Optional>> Remove Project Risk
<<Optional>> Delete Project
<<Optional>> Update Project
<<Optional>> Set Project Manager
<<Optional>> Add User To Project
<<requires>>

Figure 1. Use Cases of the SPL for Projects Management

Figure 1 presents the SPL for Project Management systems as an example of a software product line that can be derived from the SPL of enterprise applications. Some authors like Robert B. identify use case patterns [89], which are recurrent user intentions and system responsibilities that are present in several systems. We use this concept in order to abstract and group the functional scope of the SPL of enterprise applications. Our proposal includes the following use case patterns as part of the functional scope we can reach. Figure 2 summarizes these use case patterns, which conform the functional scope of the SPL of enterprise applications.

- **List all elements from a business entity.** This abstraction applies to the use cases: *List All Projects*, *List All Risks* and *List All Users*, where the business entity abstraction might be concretized into *Project*, *Risk* and *User*. This use case pattern refers to retrieving all instances of a particular business entity.

- **Create a master business entity.** This abstraction is used in the uses cases: *Register New User* and *Register New Project*, where master stands for an entity that is self identified and do not depends on the existence of others to be created [90] [91]. The business entity abstraction might be concretized to *Project* and *Risk*.

- **Create a detail business entity.** This abstraction applies to the use case: *Create Project Risk*, where detail stands for an entity that depends on the existence of another entity to be identified and created [90] [91], which is specified by the <<*requires*>> stereotype. The business entity abstraction in this case is concretized into *Risk*.

- **Delete a master business entity.** This abstraction applies to the use case: *Delete Project*, where the business entity abstraction is concretized into *Project*.

- **Associate two master business entities (One to Many fashion).** This abstraction is used in the use case: *Add User To Project*, given that both business entities *User* and *Project* are masters and a relationship between them is created, where one project may have many users associated.

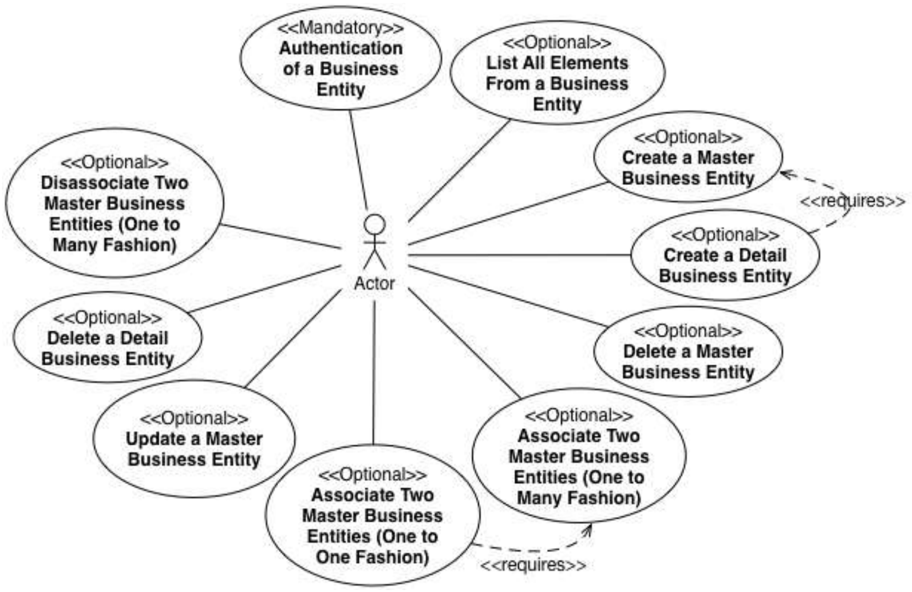
- **Associate two business entities (One to One fashion).** This abstraction applies to the use case: *Set Project Manager*, given that it relates one *User* to one *Project*. This association has the particularity that it may only be established from a previous existing "one to many" association between the same entities. This constraint is represented by the <<*requires*>> association, e.g. a project can set its manager as long as it has users related to it (the manager must be one of these users).

- **Update a master business entity.** This abstraction is used in the use case: *Update Project*, where the business entity abstraction is concretized into *Project*.

- **Delete a detail business entity.** This abstraction applies to the use case: *Remove Project Risk*, where the business entity abstraction is concretized into *Risk*.

- **Disassociate two master business entities (One to Many fashion).** This abstraction applies to the use case: *Remove User From Project*, given that both business entities *User* and *Project* are masters and an existing "one to many" relationship between them will be eliminated.

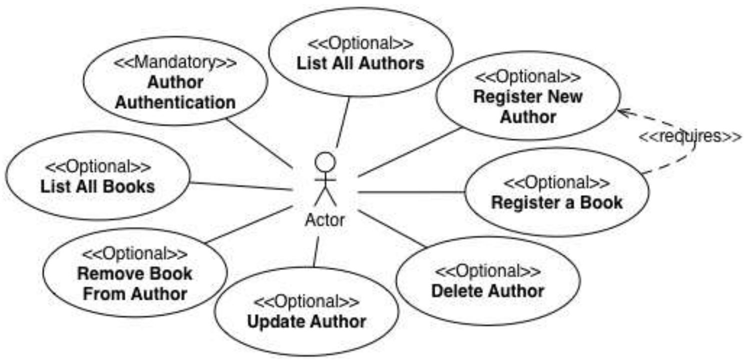
- **Authentication of a business entity.** This abstraction is used in the use case: *User Authentication*, where the *User* entity provides the information to be used in the authentication process. This use case pattern must be particularized for one and only one business entity, i.e. *User* entity only.



<<Mandatory>> Authentication of a Business Entity
<<Optional>> List All Elements From a Business Entity
<<Optional>> Create a Master Business Entity
<<requires>>
<<Optional>> Create a Detail Business Entity
<<Optional>> Delete a Master Business Entity
<<Optional>> Associate Two Master Business Entities (One to Many Fashion)
<<requires>>
<<Optional>> Associate Two Master Business Entities (One to One Fashion)
<<Optional>> Update a Master Business Entity
<<Optional>> Delete a Detail Business Entity
<<Optional>> Disassociate Two Master Business Entities (One to Many Fashion)
Actor

Figure 2. Use case patterns of the SPL of enterprise applications

Notice that this super set of use case patterns can be used to configure several SPL of enterprise applications. For instance, figure 1 shows how these use case patterns are concretized to create a SPL for Project Management systems. Another customer might be interested in configuring a SPL for Reference Management systems, where the functionalities are creation/deletion of Authors and Books, allowing relating these two concepts. Figure 3 depicts the functional variability of this SPL in terms of use cases.



<<Mandatory>> Author Authentication
<<Optional>> List All Authors
<<Optional>> Register New Author
<<requires>>
<<Optional>> List All Books
Actor
<<Optional>> Register a Book
<<Optional>> Remove Book From Author
<<Optional>> Update Author
<<Optional>> Delete Author

Figure 3. Use Cases of the SPL for Reference Management

The software product lines derived from the SPL of enterprise applications must also consider quality variability, such as levels of performance and/or security. This issue will be addressed in section 4.3.2.

4. PROPOSAL

Our contribution is a approach and tool support that, making use of software design good practices, allows a product line engineer to automatically derive products that are configured based on a set of functional and quality constraints. To do so, we provide a Domain Metamodel to deal with functional variability, we propose a features model to address non-functional (quality) variability; we mapped each variant of the quality model with enterprise Java patterns that promote the related quality attribute, and we developed a generation engine that takes a model from our domain metamodel and a configuration of our quality model as inputs to generate a product that satisfies such constraints. Following sections explain these steps in detail.

GENERAL PROCESS FOR PRODUCT DERIVATION

In order to derive product line members by using our proposal, a series of steps must be performed. Figure 4 depicts the process we designed for product line engineers to create SPLs. The first activity is concerned to define the product line scope, which we separate in functional scope and quality scope. This includes relating functionalities and quality attributes in order to determine how quality decisions impact the implementation of product functionalities. Once the scope has been set and the required relationships have been determined, the software design process of product line members begins. For that, as part of our generation engine, enterprise design patterns are selected according to functionality and quality attributes previously defined; such design patterns are used to design applications according to our controlled and pre-set Reference Architecture. Finally, code generation is performed, coupling design decision incrementally.

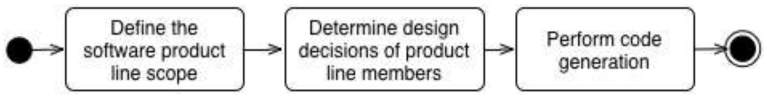

Define the software product line scope
Determine design decisions of product line members
Perform code generation

Figure 4. General Process for product derivation

4.1. DEFINE THE SOFTWARE PRODUCT LINE SCOPE

4.1.1. FUNCTIONAL SCOPE DEFINITION

The first activity that a product line engineer must perform to use our proposal is the definition of the functional scope of the line to be configured. To do so, we presented in figure 2 a super set of use case patterns to enable configuration of product line members. Parting from this, we have taken and adapted a Domain Metamodel (DMM) from our previous work [92] in order to represent the use case patterns our strategy involves. The metamodel captures the variability in terms of business entities and their relationships, enabling **managing functional variability for enterprise applications** that involve CRUD operations over business entities, considering Master-Detail (One to Many) and "One to One" relationships between them. Creating a domain model according to this domain metamodel determines the functional scope of the product line member to be configured. The concepts involved in the domain metamodel are explained below. Figure 5 shows the domain metamodel.

- **Business.** Represents the identification of the SPL that is going to be configured. This identification must be provided as the name of the Business concept, i.e. "Project Management SPL".

- **Business Entity.** Represents a main concept of the business that stores data values and expose them through properties; they contain and manage business data used by the products derived from the SPL, i.e. Project, User and Risk. The *name* property acts as a label for the Business Entity, while the *isAuthenticable* property says whether the entity will be used as the authentication of the SPL or not.

- **Attribute:** Every Business Entity has many attributes. An attribute contains particular information related to an entity, i.e. Project name and start date. Each attribute specifies its type, which can be any of the values exposed by the *DataType* concept, and it must indicate weather it is required (i.e. needs a particular value or can be null) and if it's the identification of its container (BusinessEntity).

- **Association.** Business entities might be related in two ways: "one to one" and "one to many". **SimpleAssociation** concept represents a "one to one" association, where the *relatedEntity* relationship indicates the associated entity, i.e. Project's related entity is User. **MultipleAssociation** concept represents a "one to many" association, which is a Master-Detail association, where the *entity* relationship indicates the entity that plays the Detail Role, i.e. Project's detail entity is User.

- **Contracts.** This concept specifies the operations or services that a Business Entity exposes. Such operations are CRUD related, i.e. creation, updating, deletion and retrieval of Projects. **ContractElements** are a particular type of *Contracts* that determine the operations that can be performed within a Master-Detail relationship, such as addition and deletion of details to a master, i.e. add users to a project.

The following restrictions and conditions must be taken into account, in order to properly use the Domain Metamodel:

1. There must be one and only one *Business Entity* with its *isAuthenticable* attribute set to true. This is due to the authentication of the SPL that must be realized with only one entity.
2. Authenticable *Business Entity* must define a *password Attribute*, which is required in the authentication process.
3. Master *Business Entities* are the ones that own one or many *Associations* of type Multiple.
4. Detail *Business Entities* are the ones that do not own any *Association*.
5. Each *Association* must have one and only one owner.
6. Every *Business Entity* might have at most one *Contract* of each type, i.e. one *ListAll Contract* only.

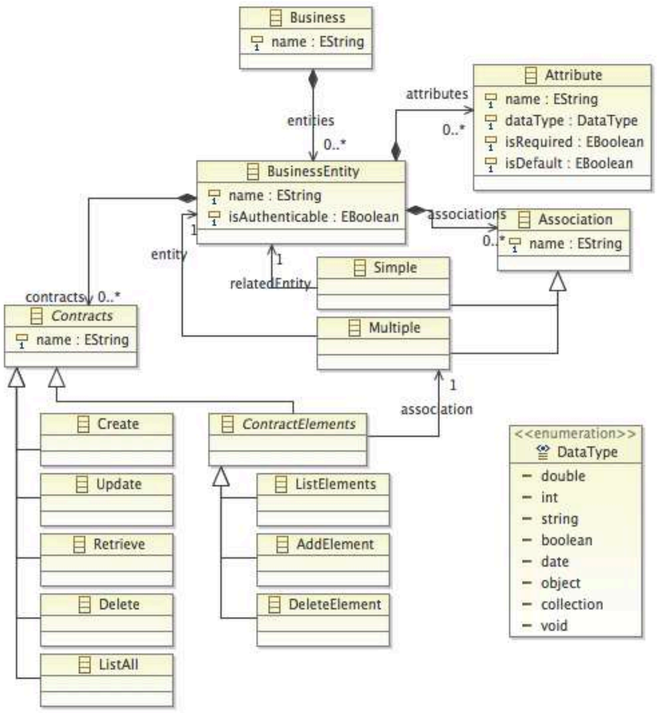


Figure 5. Domain Metamodel

Figure 6a presents a domain model that corresponds to the functional scope of the SPL for Project Management systems presented as use case diagram in figure 1. Figure 6b depicts the corresponding domain model of the functional scope of the SPL for Reference Management systems presented as use case diagram in figure 3.

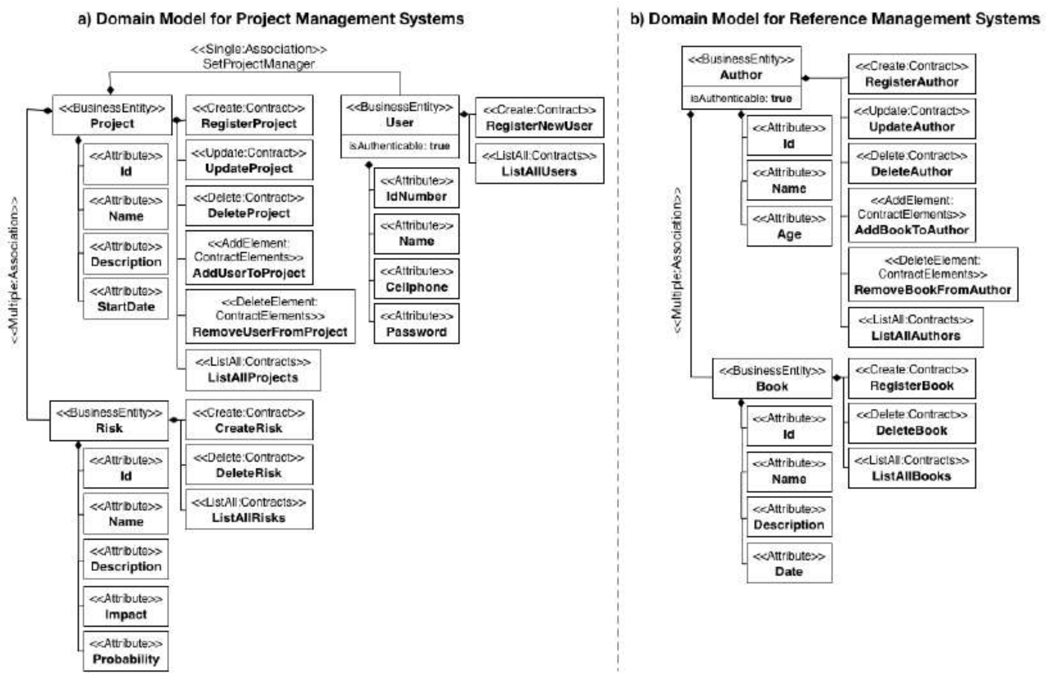

a) Domain Model for Project Management Systems
<<Single:Association>> SetProjectManager
<<Multiple:Association>>
<<BusinessEntity>> Project
<<Attribute>> Id
<<Attribute>> Name
<<Attribute>> Description
<<Attribute>> StartDate
<<Create:Contract>> RegisterProject
<<Update:Contract>> UpdateProject
<<Delete:Contract>> DeleteProject
<<AddElement: ContractElements>> AddUserToProject
<<DeleteElement: ContractElements>> RemoveUserFromProject
<<ListAll:Contracts>> ListAllProjects
<<BusinessEntity>> User
isAuthenticable: true
<<Attribute>> IdNumber
<<Attribute>> Name
<<Attribute>> Cellphone
<<Attribute>> Password
<<Create:Contract>> RegisterNewUser
<<ListAll:Contracts>> ListAllUsers
<<BusinessEntity>> Risk
<<Attribute>> Id
<<Attribute>> Name
<<Attribute>> Description
<<Attribute>> Impact
<<Attribute>> Probability
<<Create:Contract>> CreateRisk
<<Delete:Contract>> DeleteRisk
<<ListAll:Contracts>> ListAllRisks
b) Domain Model for Reference Management Systems
<<BusinessEntity>> Author
isAuthenticable: true
<<Attribute>> Id
<<Attribute>> Name
<<Attribute>> Age
<<Create:Contract>> RegisterAuthor
<<Update:Contract>> UpdateAuthor
<<Delete:Contract>> DeleteAuthor
<<AddElement: ContractElements>> AddBookToAuthor
<<DeleteElement: ContractElements>> RemoveBookFromAuthor
<<ListAll:Contracts>> ListAllAuthors
<<Multiple:Association>>
<<BusinessEntity>> Book
<<Attribute>> Id
<<Attribute>> Name
<<Attribute>> Description
<<Attribute>> Date
<<Create:Contract>> RegisterBook
<<Delete:Contract>> DeleteBook
<<ListAll:Contracts>> ListAllBooks

Figure 6. Domain Model example

If a product line engineer is interested in extending the domain metamodel to support a wider range of operations for the product line members, he/she may analyze the use cases involved in the target product line members, searching for similar characteristics among them that might be generalized or taken to a higher level of abstraction, allowing the identification and declaration of abstract use case patterns that are particularized depending on business functional requirements. Then, the newly use case patterns must be mapped into domain concepts e.g. business entities, so they can be included in the domain metamodel.

4.1.2. QUALITY SCOPE DEFINITION

The products we are able to derive using our approach must satisfy the functional needs of the customers as well as have the desired quality attributes. Thus, the product line engineer must define the quality scope of the SPL of enterprise applications. Our approach enables defining the quality scope in terms of QAs, particularly, two of them, **performance** and **security**. ISO 25010 defines performance as "*the degree to which the software product provides appropriate performance, relative to the amount of resources used, under stated conditions*". It is specialized into **Time Behavior**, Resource Utilization and Performance Efficiency Compliance. We focus on the first one, which determines the degree to which the software product provides appropriate response and processing times and throughput rates when performing its function, under stated conditions. On the other hand, security is described as "*the protection of system items from accidental or malicious access, use, modification, destruction, or disclosure*", and it is specialized into 6 attributes. We consider 3 of those 6: **Confidentiality**, **Authenticity** and **Integrity**. The first one is related to protection from unauthorized disclosure of data or information, whether accidental or deliberate, the second one deals with proving the identity of a subject or resource, the third one ensures completeness of the data by avoiding modifications in an unauthorized or undetected manner. We use feature models to represent QAs like the work presented in [15], this help us to avoid depending on specific tools for modeling. Figure 7 depicts our QAs variability model.

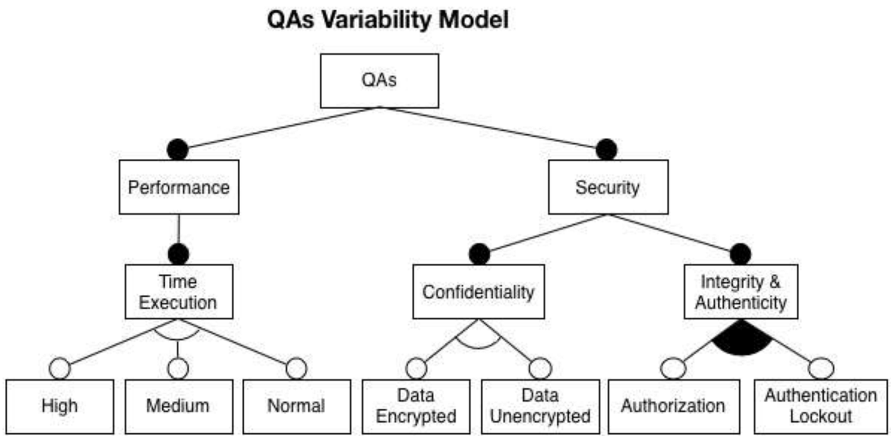


Figure 7. QAs Variability Model

- **Performance:** Related to the response time (time execution) of database operations, in particular, retrieval of several records of a table (entity). This QA provides the following levels:
    - **Normal:** Refers to achieving response times equivalent to the provided by the database system. We will call such time *X1*.
    - **Medium:** Refers to achieving lower response times than the ones offered by the database system. Let *X2* be the medium performance time; the condition $X2 < X1$ will always occur.
    - **High:** Refers to achieving lowest possible response times for data retrieval. Let *X3* be lowest possible response times; the condition $X3 < X2$ will always occur.

- **Security:** Related to protecting access, use, modification and/or disclosure of the system items. This QA is specialized into the following QAs:
    - **Confidentiality:** Provides a protection to the system data by encrypting it before it reaches the database. This QA provides two levels: one to indicate that the system data must be encrypted, and another one to indicate ignoring of data encryption.
    - **Integrity:** Refers to proving the identity of a subject that tries to access the system. This QA provides an optional level that enables blocking an account when several failed login attempts occur.
    - **Authenticity:** Provides proper authorization to users when they try to access/modify the system data. This QA provides an optional level that enables the system to provide a type of access control to its functionalities.

Notice that every QA is a mandatory feature. Such conditions occurs because as said in [93], functionalities can be definitely involved in or removed from a product of the SPL, but an QA can never be said to be involved or removed but only high or low in the degree of its effectiveness. Therefore, we can still consider QAs from a realistic viewpoint, that is to say a QA can be seen as not concerned if no special consideration is needed for it. *Time Execution* and Confidentiality QAs are exclusive grouped features, meaning that only one level of them can be selected at a time. We also decided to group *Integrity* and *Authenticity* QAs to illustrate an inclusive grouped feature. Such QAs can be modeled as separate QAs as well.

4.1.2.1. RELATE THE DMM WITH THE QAS VARIABILITY MODEL

By using our approach, product line engineers accurately model the impacts of functional variants on quality attributes and vice versa. These impact relationships are indispensable to take the most adequate decisions during design and derivation of products to promote the required quality levels. Our concern with these relationships is to make explicit how software design practices are used to promote desired quality levels. Thus, in essence, we seek to recognize which QAs affect the implementation of the functionalities of our SPL and how they do it.

We part from the premise that functionalities from the Domain Metamodel are affected by the QAs contained in the QAs variability model. That means that functionalities may vary their implementation depending on the desired quality level. We decided to use syntax similar to BBNs [14] to represent these relationships, but we do not consider quantification of them, given that we are interested in identifying and using design strategies to promote desired quality levels and not in measuring them. Figure 8 depicts these relationships.

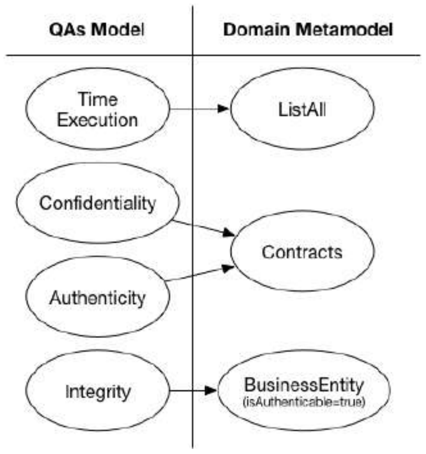


Figure 8. Relationships between DMM and QAs Variability Model

Given that *ListAll Contract* from our Domain metamodel reefers to the operation of retrieving the entire records of a *Business Entity* (which is mapped to a table in the database), and that we defined *Time Execution* as the response time of database operations, in particular, retrieval of several records of a table (entity), the selection of a *Time Execution* level directly affects the implementation of the *ListAll* operation.

Since *Confidentiality* QA deals with data encryption, every functionality of our Domain Metamodel must be adapted to support such operation. In our case, *Contracts* element represents all possible functionalities that our SPL might satisfy; therefore *Confidentiality* impacts the implementation of each *Contract* of our line. *Authenticity* QA also affects every *Contract* of the line, because it authorizes the execution of each functionality, according to the permissions provided by a role; therefore, a validation of permissions must be performed prior the execution of every *Contract*. We also spotted that *Integrity* QA impacts the implementation of the authenticable *Business Entity*, since it provides an extra action on the authentication operation (which is provided by this entity) to block an account after several failed access attempts.

4.1.2.1.1. COMPLEX NATURE OF RELATIONSHIPS

The definition of relationships among QAs and functionalities involves different kinds of relationships. First kind refers to granularity, indicating the scope of the relationships, that is, whether a relationship impacts the entire product functionalities or just some of them. Granularity is divided into two types, coarse-grained and fine-grained. Former implies that every instance of a functionality affected by a QA has to promote the same level of quality given; this means that if we part from the model depicted in figure 5, when a level of *Time Execution* is selected, "*Medium*" for example, every instance of *ListAll Contract* has to promote such level, therefore, the implementation of the *Contracts ListAllProjects* and *ListAllUsers* must be adapted to develop a medium level of time execution. Latter relationship, fine-grained, suggests that different levels of quality may be promoted by the instances of a functionality affected by a QA, thus *ListAllProjects Contract* might promote a medium level of time execution, while *ListAllUsers* could address a high level of performance. Relationships in this work are limited to coarse-grained nature.

The second kind of relationship refers to how quality levels impact on product functionalities and vice versa. Particularly, we can see that:

1. One QA may influence many functionalities. For example, the choice of using data encryption has influence on every *Contract* (functionality) of our Domain Metamodel.
2. One functionality may be influenced by many QAs. For example, levels of time execution and security have influence on *ListAll Contract* implementation.

The third kind of relationship considers how QAs interact with each other. We see that different QAs may be competing or synergic, that is, one QA may be affected by many QAs, with some contributing positively and others contributing negatively. This kind of relationship raises some conflicts when a QA influences negatively over another that must be explicitly considered:

- **Synergic Conflict**. This establishes a soft condition, which implies that although the impact of a QA over another QA lowers the promotion of the desired quality levels, they can still both coexist. For example, if *High* level of *Time Execution* is desired, a particular response time *Tr* is expected. If Confidentiality of the data is also required

(Data Encrypted), then the *Tr* time provided by *ListAll* functionalities is diminished, due to the required time of decryption of retrieved data from the database. Even when both *Time Execution* and *Confidentiality* levels are to be promoted, *High* level of time execution in the presence of *Data Encryption* will still be faster than a *Normal* level of it. Therefore, both QA may be selected. This condition must be informed when configuring the desired quality levels, enabling the user to decide whether the resulting quality levels fit his needs or not.

- **Competing Conflict**. This establishes a hard condition, which denotes that two QAs cannot coexist, because the selection of one entirely inhibits or contradicts the other, provoking a mutual exclusion among of them. For example, let's assume that promoting a *Medium* level of *Time Execution* requires a cache to avoid reprocesses for similar requests to the database. Let's also say that we are interested in promoting a *Medium* level of *Availability* (achieve a degraded operational level when one of the servers is unavailable), which uses a *spare computing* pattern to replace a failed component. These spare components require to be initialized to a persistent state before enter into operation. This pattern promotes the use of persistent storage (e.g. database) to maintain application state and avoid the use of memory. Hence, accessing the database to handle requests is mandatory and no caches are permitted. When a product is configured to promote these two quality levels a conflict arises, given that one requires using a cache structure that's forbidden by the other one. Thus, we conclude that in our example these two quality levels cannot be applied simultaneously. We do not provide a mechanism in this work to handle it.

Once the quality scope is characterized in the QAs variability model, the product line engineer must configure the quality levels for each product line member. Selecting the desired quality levels from the QAs variability model does this. Figure 9 depicts the SPL for Product Management systems presented in figure 6a configured to promote a medium level of time execution and an authentication lockout level of integrity. At this point, the product line engineer knows which functionalities the selected quality levels impact on. In order to extend the quality scope of the SPL of enterprise applications, different QAs must be considered, e.g. maintainability and usability. Once the target QAs have been selected, the interactions among these attributes and the functionalities must be set.

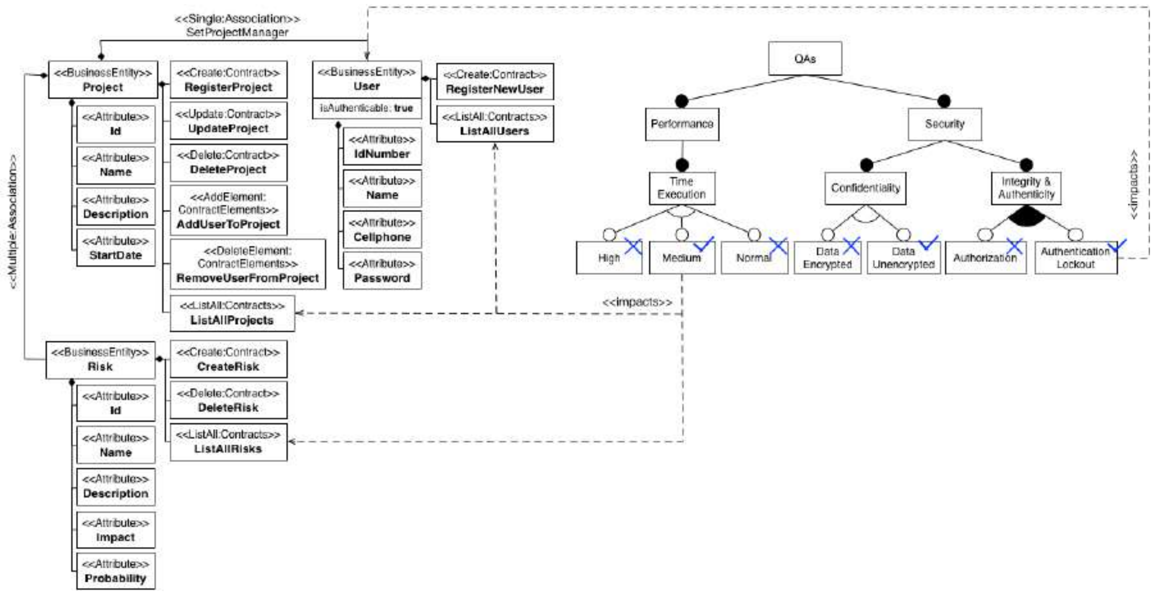

<<Single:Association>> SetProjectManager
<<BusinessEntity>> Project
<<Create:Contract>> RegisterProject
<<Update:Contract>> UpdateProject
<<Delete:Contract>> DeleteProject
<<AddElement: ContractElements>> AddUserToProject
<<DeleteElement: ContractElements>> RemoveUserFromProject
<<ListAll:Contracts>> ListAllProjects
<<Attribute>> Id
<<Attribute>> Name
<<Attribute>> Description
<<Attribute>> StartDate
<<BusinessEntity>> User
isAuthenticable: true
<<Create:Contract>> RegisterNewUser
<<ListAll:Contracts>> ListAllUsers
<<Attribute>> IdNumber
<<Attribute>> Name
<<Attribute>> Cellphone
<<Attribute>> Password
<<Multiple:Association>>
<<BusinessEntity>> Risk
<<Create:Contract>> CreateRisk
<<Delete:Contract>> DeleteRisk
<<ListAll:Contracts>> ListAllRisks
<<Attribute>> Id
<<Attribute>> Name
<<Attribute>> Description
<<Attribute>> Impact
<<Attribute>> Probability
QAs
Performance
Security
Time Execution
Confidentiality
Integrity & Authenticity
High
Medium
Normal
Data Encrypted
Data Unencrypted
Authorization
Authentication Lockout
<<impacts>>
<<impacts>>

Figure 9. Example of quality configuration impact on SPL for Project Management systems

4.2. DETERMINE DESIGN DECISIONS OF PRODUCT LINE MEMBERS

Once the SPL scope is set, our approach states that the product line engineer must take design decisions to determine how quality levels of the QAs variability model impact the implementation of functionalities from the DMM.

4.2.1. IDENTIFY ENTERPRISE DESIGN PATTERNS TO PROMOTE VARIANTS FROM THE QAS MODEL

We selected specific design tactics/strategies to promote the quality levels provided by each QA. To do so, we turned to design patterns. Usually, design patterns are organized in catalogs, according to the domain scope. Since our work is focused on managing functional variability for enterprise applications, we based on Bien's catalog, which provides a set of patterns that are intended to be use in the context of enterprise applications, using the Java Enterprise Edition (JEE) specification. We use the following patterns in order to promote quality levels for Time Execution:

- *Normal Time Execution* level refers to achieving response times equivalent to the provided by the database system, therefore, the default implementation of a JEE component (bean) that manages data retrieval is sufficient to promote this level. Bien proposes the *ECB* (Entity-Control-Boundary) approach, which acts as the base (default) structure to conceive JEE components. Thus, in order to promote the *Normal* level of *Time Execution* we will use the plain *ECB* approach.

- The Fast Lane Reader (*FLR*) pattern provides a more efficient way to access large amount of read-only data. According to Bien, there are several strategies to implement this pattern, depending on business needs and conditions. In our case, we seek to achieve lower response times than the ones offered by the database system for our *Medium Time Execution* level. The *ListAll* functionality retrieves all the records from the database of the related *Business Entity*, where a record is a set of values for the *Attributes* of that entity. We decided to use the *JDBC* strategy to implement the *FLR*, in order to promote a *Medium* level of *Time Execution*. This strategy takes advantage of the read-only access to the database, to provide direct access to the *DataSource* [94] resource, surpassing the need of a *PersistenceContext* [95] (management of instances and lifecycle of the data base entities). This ends up in retrieving a bunch of primitives (*Attributes* values) from the database for each record of the *Business Entity*, improving the response time of this operation.

- To promote a *High* level of *Time Execution*, we also rely on FLR pattern, the difference is that we use another strategy to implement it. Given that this level pursues the improvement of the response times provided by the *Medium* level, we used the *Paginator* strategy, which states that for dealing with the retrieval of several records, the entire result set can be divided into smaller chunks. Knowing this strategy, we found that dealing with several chunks of records might be performed sequentially and in parallel, so we provided both strategies to deal with *High* level of *Time Execution* as follows:
    - **Sync Strategy.** Uses one thread of execution to retrieve a several chunks of records. Each chunk is displayed to the user as is gathered. This is achieved by applying a plain *Paginator*.
    - **Async Strategy.** Uses several threads of execution to retrieve a several chunks of records. Each thread's objective is to retrieve a chunk. All chunks must be gathered (each thread must complete its task) before displaying them to the user. This is achieved by applying the *Parallelizer* pattern, which increases the throughput as it performs multiple asynchronous operations.

To deal with security levels, we evaluated how Bien's patterns might contribute to promoting them, and we found that *Data Unencrypted* level of *Confidentiality* can also be satisfied with the default implementation of a Java EE component (bean) that promotes CRUD functionalities. Thus, *ECB* approach is also used to promote this level. To deal with *Data Encryption* level, we consulted Oracle docs [96] and we decided to use the Java security API [97] to address this issue. In particular, we selected the Password Based Encryption (PBE) strategy to create a *CryptographyManager* that is responsible of providing entire encryption and decryption services, based on this cryptography method. Notice that this manager may provide different levels of encryption, by using different strategies like RSA [98] and AES [99].

Dealing with *Integrity & Authenticity* QA required us to consult catalogs of patterns specialized in security. We studied the "Security Patterns in Practice" catalog [100], which provides software patterns to design secure architectures. Taking advantage of the classification of patterns based on the security concern that is provided by this catalog, we studied the patterns related to access control, in order to find the most suitable pattern to provide our *Authorization* level (related to the *Authenticity* QA). We decided to use the *Role-Based Access Control* pattern, because it describes how to assign rights based on the functionalities of users in an environment in which control of access to computing resources like system data, is required, which accommodates to our expectations for *Authenticity*.

We use the Lockout pattern from the catalog in [101] to promote our *Authentication Lockout* level (*Integrity* QA). This pattern aligns perfectly with our *Integrity* definition, given that it protects customer accounts from automated password-guessing attacks, by implementing a limit on incorrect password attempts before further attempts are disallowed. Figure 10 depicts the QAs variability model (including the two newly *High Time Execution* strategies) and how the quality levels are realized through software design patterns.

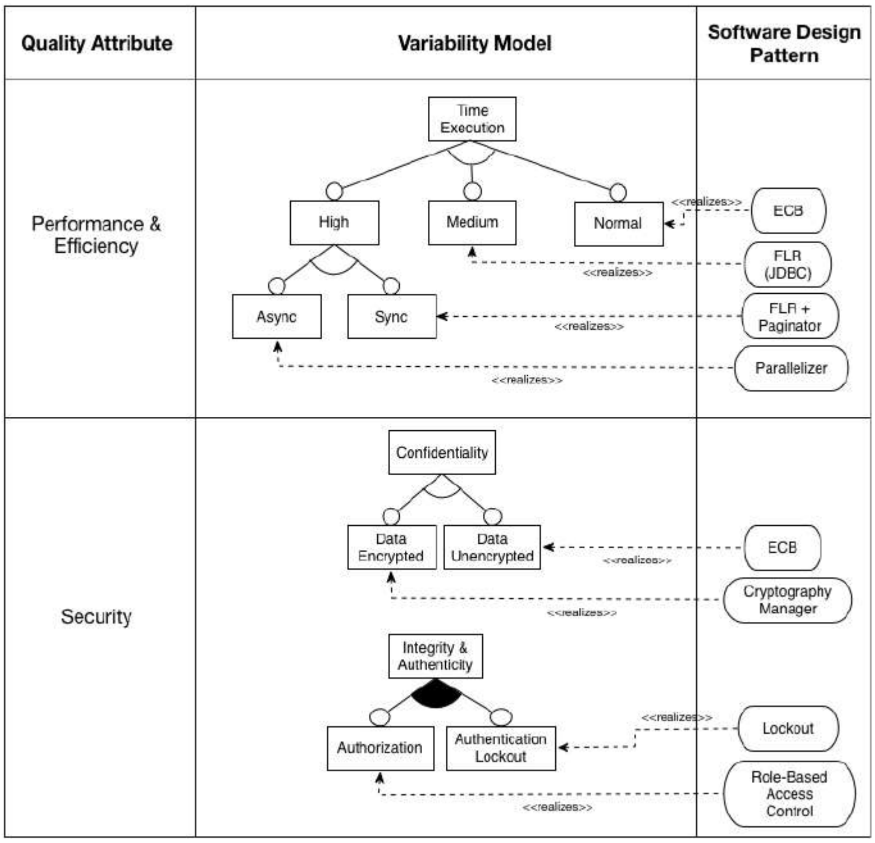


Figure 10. Software Design Patterns to promote quality levels

4.2.2. *Reference Architecture Definition*

This section consolidates specific design decisions based on identified enterprise design patterns in previous section, providing class and sequence diagrams to illustrate how each pattern must be applied, depending on the quality level and the impact relationships between functionalities and QAs.

4.2.2.1. *Macro Architecture Specification*

The macro architecture of every product line member follows the three-layer architectural style [29] [31], which defines three layers: the presentation, logic and data-access layers. An application designed using this style includes these three layers. In addition, each layer comprises components with layer-specific types and responsibilities: the presentation layer comprises GUI components; the logic layer includes the components that implement the logic behind the business transactions; and the data access layer includes data-access components that store and retrieve data from files, servers and databases. Figure 11 provides a component view representing this layer division.

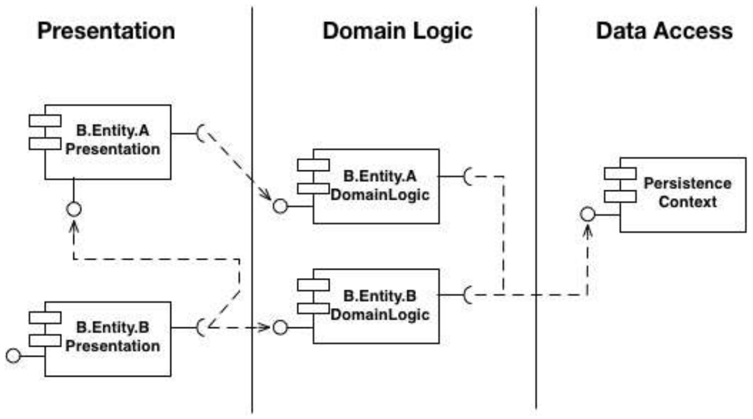


Figure 11. Macroarchitecture: Components view

4.2.2.1.1. MACRO ARCHITECTURE RESTRICTIONS

Figure 11 presents how components are arranged in the three-layer style, and how they may interact with each other. From this figure we can conclude the following restrictions that apply to every product of the line:

1. Data Access layer contain a unique component that can be identified as a generic Data Access Object (DAO), given that it provides all-encompassing services to deal with data management.

2. Every *Business Entity* of the DMM will be mapped into two components: One GUI component and one Domain component, i.e. a *BusinessEntity X* will create an *X* GUI component and an *X* domain component. Former provides user interface visualization and functioning, while latter implements the domain logic related to the business entity, which is specified by the contracts and relationships configured when a Domain Model is derived from the DMM.

3. Domain Logic components may interact with other components among the same layer, that is, provide and consume services. Presentation components may also interact with other components contained in this layer. When a GUI component needs to perform a business operation, it must consume the services provided by the corresponding Domain component.

4. Due to Business Entity relationships, it is possible that a GUI component requires a service from a domain component of a different business entity, i.e. *ProjectUI* component might want to list the users of a project, which is a service provided by *UsersDomainLogic*. In this case, the corresponding GUI component of such domain entity (*UserUI* in this case) will expose a service that invokes the required domain service. This decision enables the interested GUI component to access the required

service through a relationship on the same layer, plus it prevents mixing responsibilities, i.e. *ProjectUI* access *ProjectsDomainLogic* services only.

For developing our case study, we used different technologies to construct the components contained by each layer. To develop GUI components we used Vaadin 7[4], which is a Java framework that enables building web components using java code. The components that contain the business logic will be developed using Enterprise Java Beans (EJB 3.1) specification [102], which handles common concerns as transactional integrity, and security in a standard way, leaving programmers free to concentrate on the particular problem at hand. Components that handle data access and persistence will be developed using the Java Persistence API (JPA 2.4.2) [103] specification, which provides a POJO persistence model for object-relational mapping.

4.2.2.2. Micro Architecture Specification

After identifying the components that will be part of the product members of the SPL of enterprise applications and the relationships between them, we will analyze micro architectural concerns, which are related to specifying the internal structure of each component. Depending on the layer of the component, several design patterns may be used to implement them [31] [104]. Thus, implementation of the components for each layer will be detailed in the following sections.

4.2.2.2.1. Presentation Components

The implementation of GUI components commonly involves the use of *UI Controllers* [105]. The *UI Controller* is the entity that processes the requests it receives from a related form or *view*. To implement a *UI controller* in the presentation layer of a web-based application, a designer can use the *Application controller*, the *Front controller* or the *Page controller* pattern [104]. In addition, to implement a *view* controller, a designer can use a *Template view*, a *Transform view* or a *Two-phase view* pattern [31][9]. Software architects decide one pattern or the other depending on the development platform and the requirements for the application. In our case, we decided that each GUI component will have an UI controller and a view associated to it. Figure 12 depicts this structure.

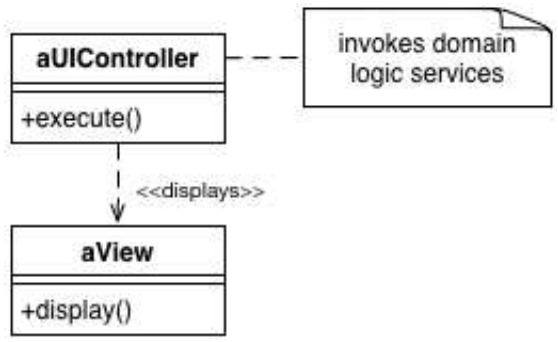


Figure 12. Microarchitecture of a Presentation Component

[4] https://vaadin.com/home

In Figure 12, UI controller is responsible for display the view and handling its events. It also invokes domain logic services depending on the functionalities provided by the view. The view is in charge of providing the user interface layout to manage the functionalities associated to the Business Entity that it represents.

4.2.2.2.2. DOMAIN LOGIC COMPONENTS

We use the *ECB* approach proposed by Bien [39] to define the base (abstract) structure of our domain components. This approach uses the basic elements involved in robustness diagrams [106] (Entity-Control-Boundary) to structure the internal composition of components. Thus, each domain logic component will have these three elements in form of layers (packages within the component's implementation). Bien also provides a pattern for each layer, identified with the same name:

- *Boundary* is a façade, which exposes a component's services in a convenient way. A client only has to know a Boundary's method in order to access the component's functionality.

- *Control* is a reusable, fine-grained service behind a Boundary. It is optional and usually created during Boundary refactorings. Uncohesive functionality is extracted from Boundaries into focused Controls.

- *Entity* refers to object-oriented or procedural domain objects. This conceptual entity is persisted and mapped to a single JPA entity.

Every Business Entity of our DMM will always provide a *Boundary* and an *Entity*. Besides these patterns, we determined the following conditions. Figure 13 presents the base structure of the domain components.

- *Control* pattern will be used in the following scenarios:
    - The presence of a Master-Detail relationship between two *Business Entities*. For example, listing the users related to a project requires retrieving data from User and Project entities. To prevent *UserDomainLogic* from accessing *Project* entity (breaking responsibilities and increasing coupling) to execute the retrieval query, a *UserDAO* is generated. Such *DAO* provides the implementation of this query using only the id of the project and not the entire entity. Thus, *Project* entity manipulation is properly managed. **Notice that the *DAO* will always be located in the component of the detail entity**. *DAOs* will be treated as *Controls* in this work.
    - Selection of particular quality levels. Depending on the quality level selected of the QAs in our model, different controls might be needed to implement the associated design pattern. These variations will be detailed in section 4.2.2.2.3.

- We intend to follow SOA principles; therefore, we must provide the must convenient level of encapsulation of our components. Parting from this assumption, Bien recommends using the Transfer Object (*TO*) pattern (which encapsulates an carries data between components) to avoid direct exposure of Entity instances to the client. Hence, every Entity will be mapped to its representing *TO*. *TOs* will be located in an independent utilities java project to be used by any component.

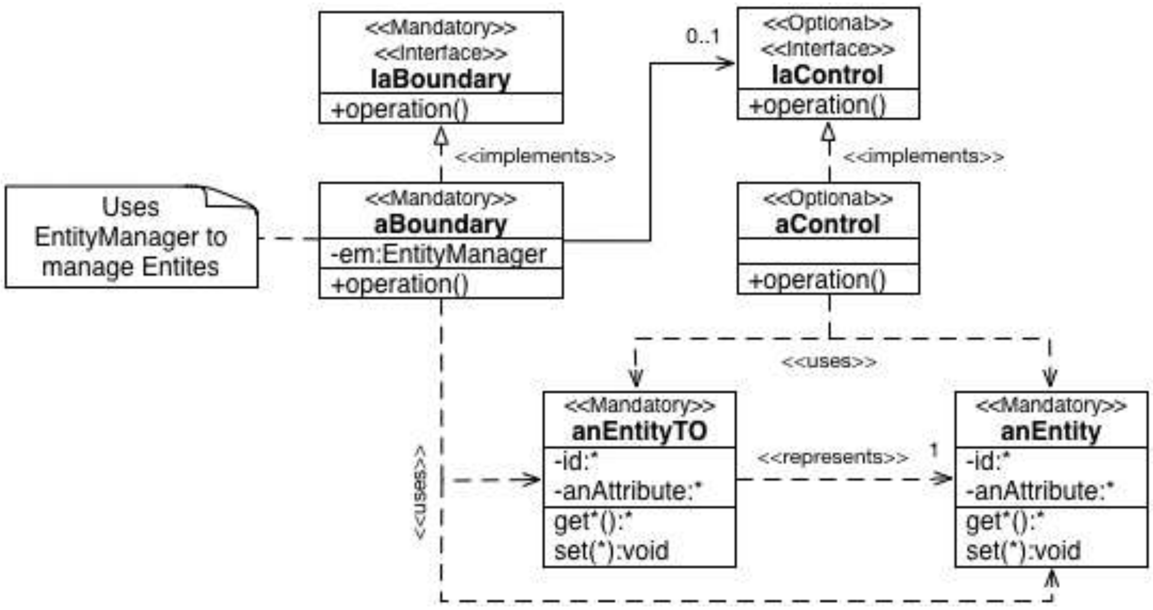


Figure 13. General Microarchitecture of Domain Components

Data Access Components won't be discussed given that we decided to use JPA specification to manage persistence issues. Thus, implementation details and pattern in this layer are omitted.

4.2.2.2.3. Considering Quality Variations in the Reference Architecture

To implement different levels of quality, QAs such as performance or security, component's internal structure must vary according to the selected design patterns shown in figure 10. Following sections detail domain logic component's adaptations for each quality level of the QAs variability model.

4.2.2.2.3.1. Time Execution

According to figure 8, time execution modifies the implementation of *ListAll* operations. Therefore, the following diagrams and examples will be focused on how the components must accommodate their internal structure according to the level of time execution, in order to perform this type of operations.

Normal Level

Given that we decided to achieve this level using the *ECB* approach, the internal structure of a domain component that is to promote this level must look like the one shown in figure 13. Notice that this structure is the abstract implementation of the *ECB* approach, thus, we will elaborate the following case to illustrate how to concretize the *ECB*. Parting from the SPL for Project Management systems presented in the case study section, a *Business Entity* for managing *Projects* is needed, and such entity must consider time execution levels due to its interest in listing all projects. Keeping in mind restriction two from section 4.2.2.1.1, a domain component for projects has to be created. This component concretization of *ECB* approach is shown in figure 14.

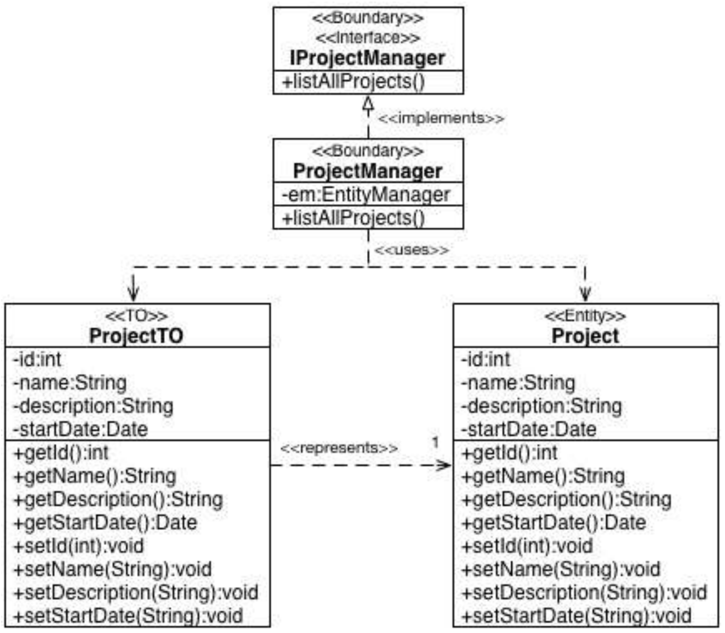
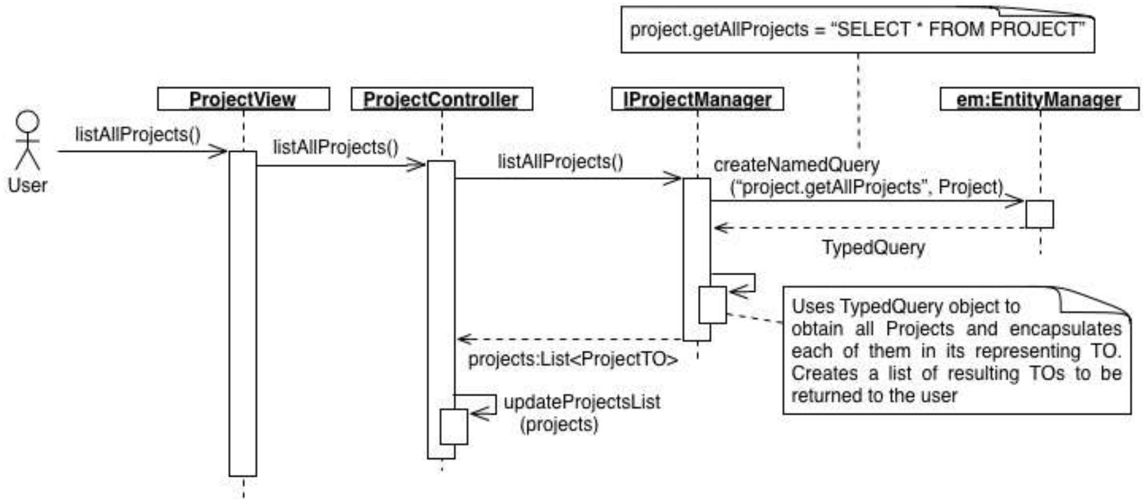


Figure 14. Normal Time Execution Concrete Design

Notice that no Control is needed. Therefore, it is no used by the projects domain logic component. To show how these elements operate with each other when a user requires listing all projects, Figure 15 presents a sequence diagram to illustrate the interaction between classes in figure 14.



project.getAllProjects = "SELECT * FROM PROJECT"
ProjectView
ProjectController
IProjectManager
em:EntityManager
User
listAllProjects()
listAllProjects()
listAllProjects()
createNamedQuery
("project.getAllProjects", Project)
TypedQuery
Uses TypedQuery object to obtain all Projects and encapsulates each of them in its representing TO. Creates a list of resulting TOs to be returned to the user
projects:List<ProjectTO>
updateProjectsList
(projects)

Figure 15. *ListAll* projects use case sequence diagram (Normal Time Execution)

Figure 15 illustrates how to use the *EntityManager* (provided by the JPA specification) to retrieve all projects. Notice that *ProjectController* interacts with project's component interface (*IProjectManager*), enabling its implementation to be client independent. The method `createNamedQuery` takes two parameters: one to specify the SQL query to be executed, and another one to identify the *Entity* who owns the query declaration; hence, this method can be used to execute any desired query. Annex 1 illustrates how to manage query declarations. Annex 2 shows an example implementation of data retrieval using the *EntityManager*.

MEDIUM LEVEL

This particular level requires the use of the *FLR* pattern with the *JDBC* strategy. This implies the need for a *Control* to manage retrieval of all records using the *DataSource* resource instead of the *EntityManager*. This condition is an exception to our macro architecture restrictions, given that the database is accessed using the *JDBC* API, by injecting the *DataSource* resource. **It is important to clarify that this access mechanism will only be used for *ListAll* operations**. The rest of them will be managed using the *EntityManager*. Figure 16 depicts how component's internal structure must be adapted. Notice how *aFLR* concrete class uses the *DataSource* to execute retrieval queries.

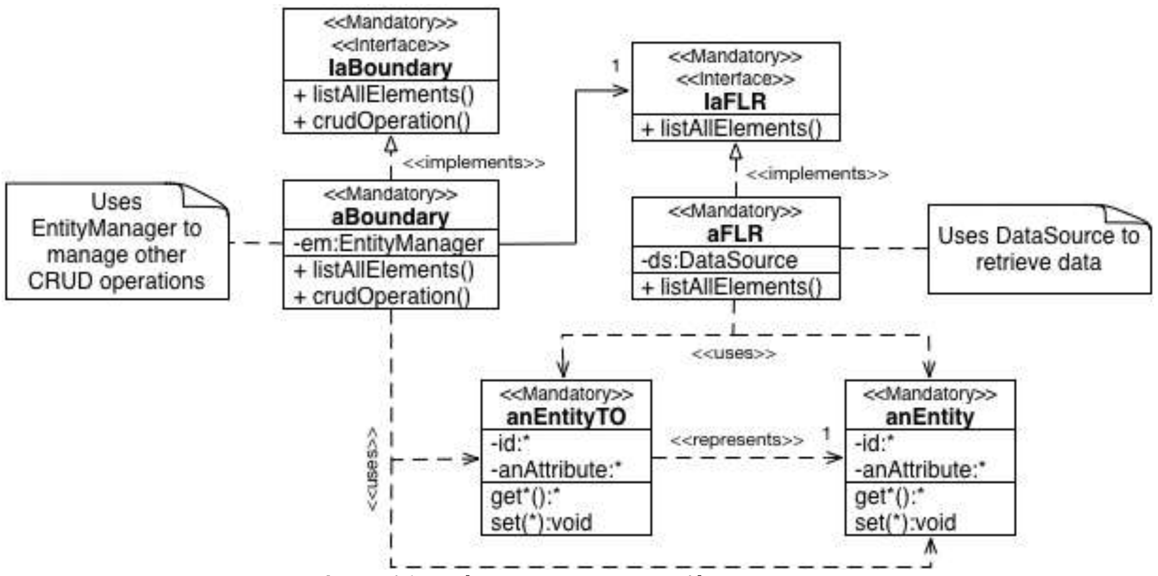


Figure 16. Medium Time Execution Abstract Design

We will continue using our previous example of the "list all projects" requirement presented in the case study section, to illustrate how to concretize this diagram. Figure 17 depicts the concretization of diagram in figure 16; figure 18 shows its sequence diagram.

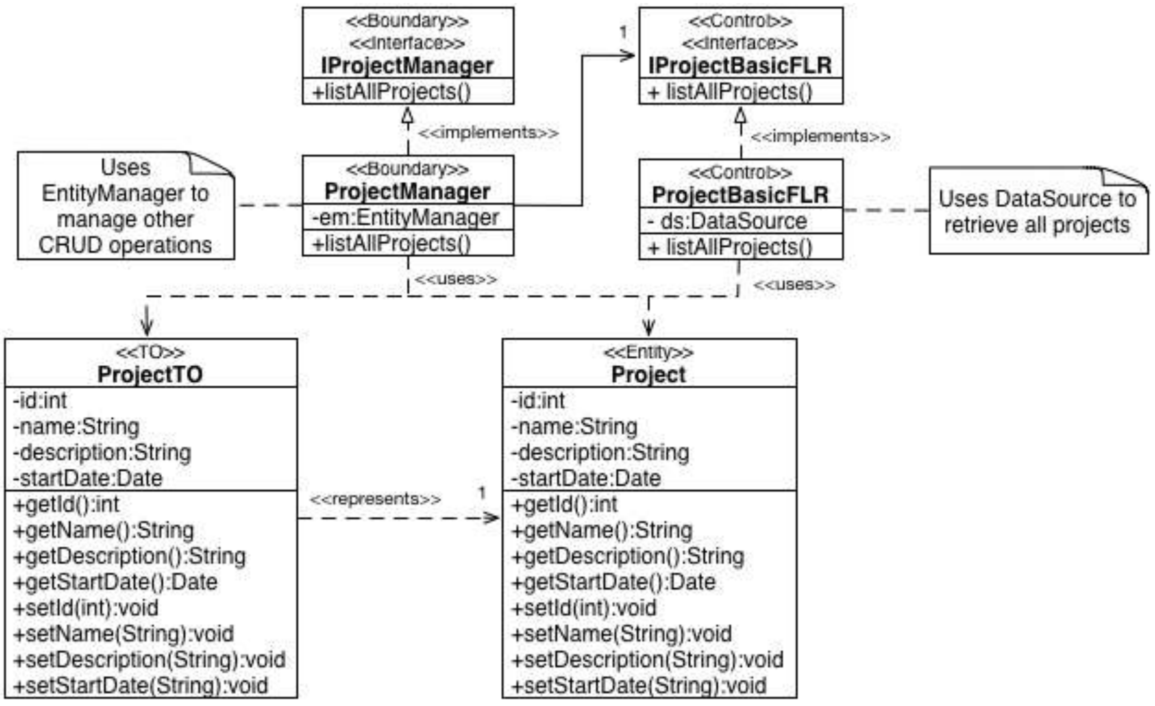
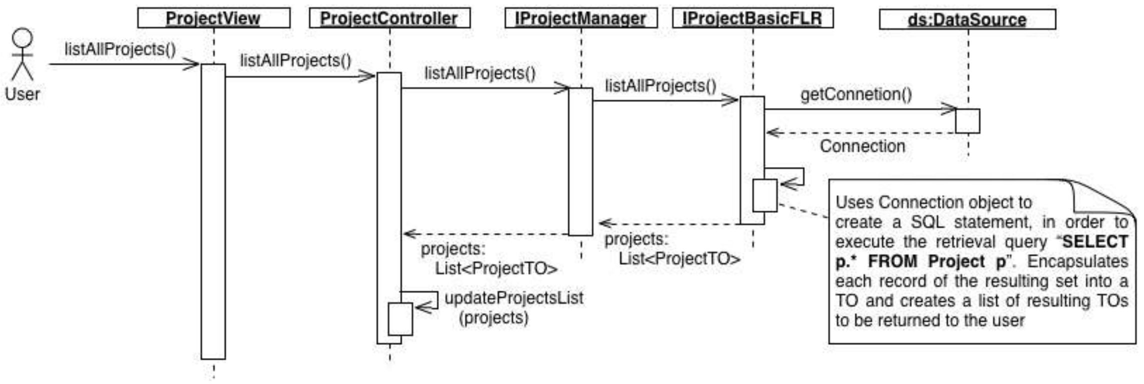


Figure 17. Medium Time Execution Concrete Design



ProjectView
ProjectController
IProjectManager
IProjectBasicFLR
ds:DataSource
User
listAllProjects()
listAllProjects()
listAllProjects()
listAllProjects()
getConnetion()
Connection
projects: List<ProjectTO>
projects: List<ProjectTO>
updateProjectsList (projects)
Uses Connection object to create a SQL statement, in order to execute the retrieval query "SELECT p.* FROM Project p". Encapsulates each record of the resulting set into a TO and creates a list of resulting TOs to be returned to the user

Figure 18. *ListAll* projects use case sequence diagram (Medium Time Execution)

Notice how the *IProjectManager* delegates the retrieval of projects to *IProjectBasicFLR*. This interface uses its concretization (*ProjectBasicFLR*) to access the *DataSource* resource, create and perform the retrieval query. Annex 3 provides an implementation of data retrieval using this pattern.

HIGH SYNC LEVEL

This particular level requires the use of the *FLR* pattern with the *Paginator* strategy. This implies the need for a *Control* to manage retrieval of all records using the *DataSource* resource, plus a mechanism to indicate the current page and the chunk size to be retrieved. The *JDBC* API is also needed to implement this pattern. **The use of this pattern will only affect *ListAll* operations**. The rest of them will be managed using the *EntityManager*. Figure 19 depicts how component's internal structure must be adapted.

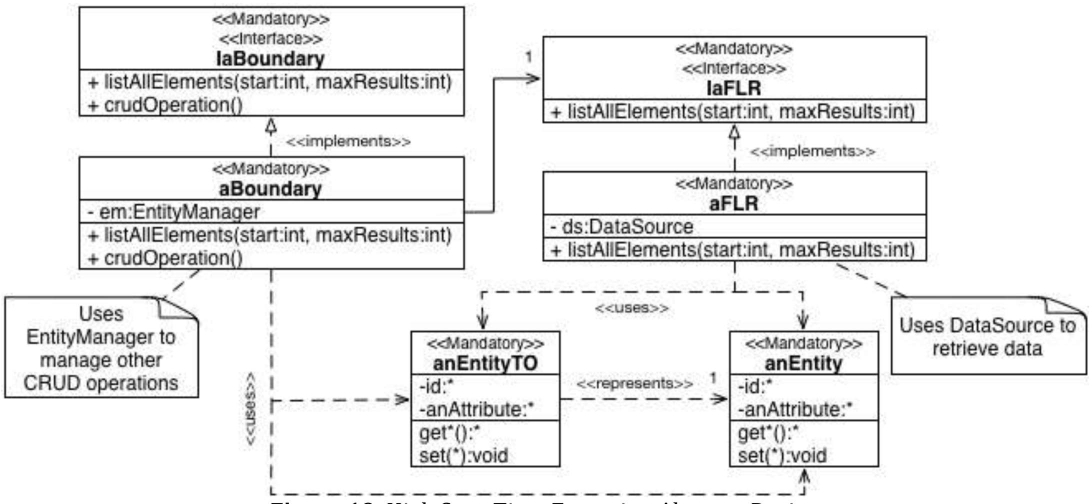
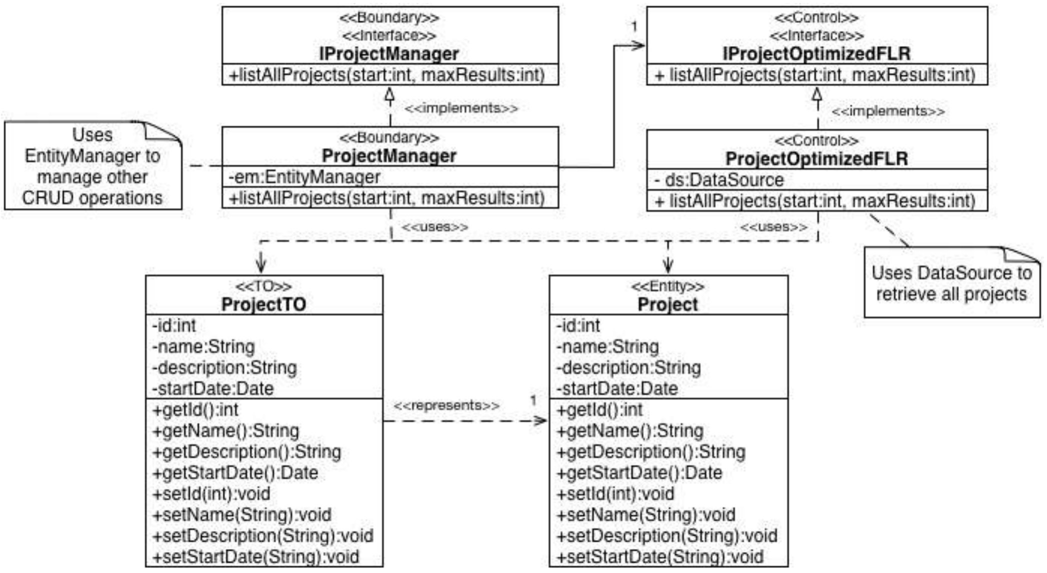


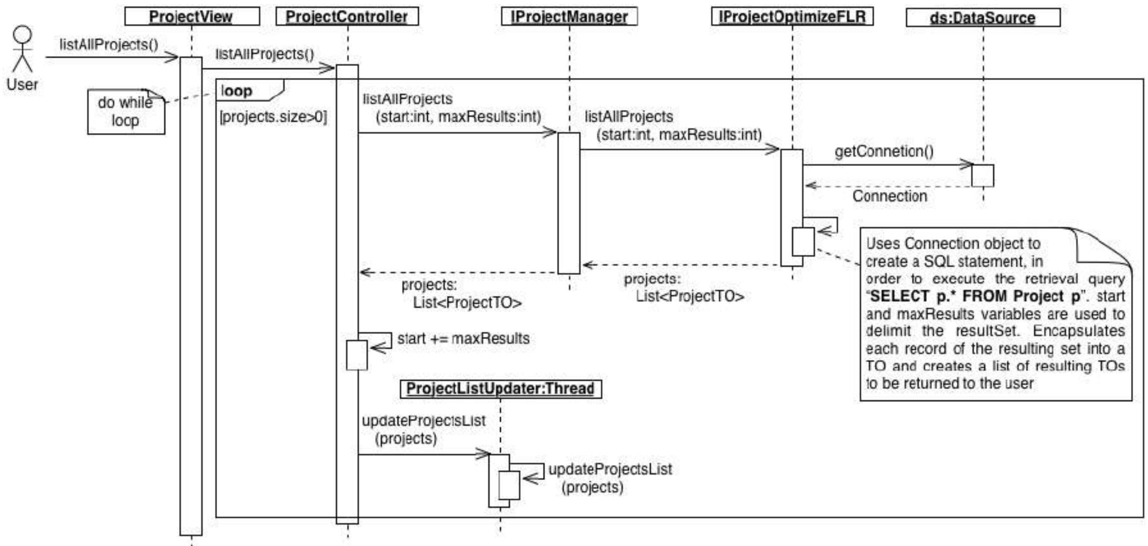
Figure 19. High Sync Time Execution Abstract Design

Bien proposes to use this pattern using a cache (*Stateful* bean [107]) to handle iteration logic, but given that we are following SOA principles, caches cannot be used. To deal with this issue, we delegated the iteration logic to the UI controller. Thus, the UI controller must indicate the current page and the chunk size to the *FLR*, so it retrieves the proper data. This adaptation of the pattern implies modifying the *listAllElements* service to take two parameters: one to indicate the current page (*start*) and another indicating the chunk size (*maxResults*). We also decided that displaying of pages does not have to be linked to the user interface (as proposed by Bien), e.g. a "Next button" to retrieve the following page of records. Therefore, the entire set of records will be displayed (as in previous time execution levels) but its retrieval will occur sequentially in several chunks. Following our previous example of the "list all projects" requirement presented in the case study section, we illustrate how to concretize the previous diagram. Figure 20 depicts this action. The corresponding sequence diagram to this retrieval process is shown in figure 21.



<<Boundary>> <<Interface>> IProjectManager
+listAllProjects(start:int, maxResults:int)
<<implements>>
<<Boundary>> ProjectManager
-em:EntityManager
+listAllProjects(start:int, maxResults:int)
Uses EntityManager to manage other CRUD operations
1
<<Control>> <<Interface>> IProjectOptimizedFLR
+ listAllProjects(start:int, maxResults:int)
<<implements>>
<<Control>> ProjectOptimizedFLR
- ds:DataSource
+ listAllProjects(start:int, maxResults:int)
<<uses>>
<<uses>>
Uses DataSource to retrieve all projects
<<TO>> ProjectTO
-id:int
-name:String
-description:String
-startDate:Date
+getId():int
+getName():String
+getDescription():String
+getStartDate():Date
+setId(int):void
+setName(String):void
+setDescription(String):void
+setStartDate(String):void
<<represents>>
1
<<Entity>> Project
-id:int
-name:String
-description:String
-startDate:Date
+getId():int
+getName():String
+getDescription():String
+getStartDate():Date
+setId(int):void
+setName(String):void
+setDescription(String):void
+setStartDate(String):void

Figure 20. High Sync Time Execution Concrete Design



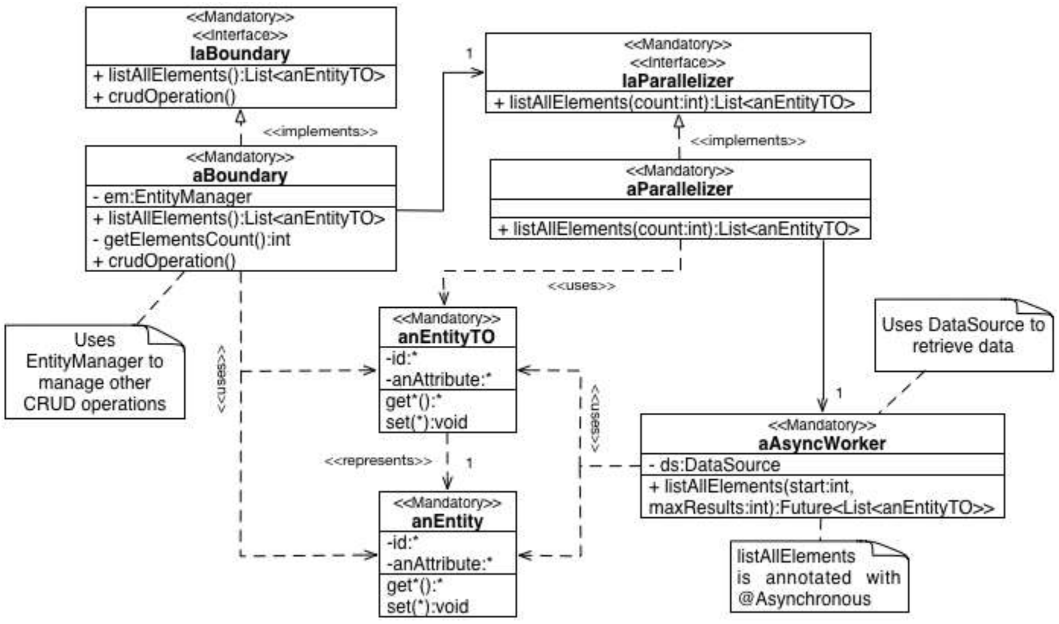
ProjectView
ProjectController
IProjectManager
IProjectOptimizeFLR
ds:DataSource
User
listAllProjects()
listAllProjects()
do while loop
loop
[projects.size>0]
listAllProjects (start:int, maxResults:int)
listAllProjects (start:int, maxResults:int)
getConnetion()
Connection
Uses Connection object to create a SQL statement, in order to execute the retrieval query "SELECT p.* FROM Project p". start and maxResults variables are used to delimit the resultSet. Encapsulates each record of the resulting set into a TO and creates a list of resulting TOs to be returned to the user
projects: List<ProjectTO>
projects: List<ProjectTO>
start += maxResults
ProjectListUpdater:Thread
updateProjectsList (projects)
updateProjectsList (projects)

Figure 21. *ListAll* projects use case sequence diagram (High Sync Time Execution)

Implementation of this pattern is similar to the one shown for the medium level of time execution, however, this one considers *start* and *maxResults* parameters to limit the projects retrieval query. Observing the sequence diagram in figure 21, *ProjecController* has to delegate the updates of the project list to a specialized class (*ProjectListUpdater*) that executes the updates in an independent thread. This is needed to provide a fluent visualization of the project list, because the retrieval of several chunks of projects is performed in the same thread, therefore, displaying the results will only occur when this thread finishes its execution. Using the *ProjectListUpdater* enables displaying every chunk right after it reaches the *ProjectController*, so the user can perceive the optimization to the retrieval time (chunks are displayed incrementally as they arrive). This additional class will be part of every GUI component that involves a *ListAll* operation, only if this level of time execution (high sync) is desired. Annex 4 provides an implementation of data retrieval using this pattern.

HIGH ASYNC LEVEL

We identified the *Parallelizer* pattern to promote a *High Time Execution* level. This pattern uses several threads of execution to retrieve several chunks of records. Each thread's objective is to retrieve a chunk. All chunks must be gathered (each thread must complete its task) before displaying them to the user. Thus, this pattern serves as a coordinator of several parallel requests of data retrieval. The general structure of this pattern is shown in figure 22.



<<Mandatory>> <<Interface>> IaBoundary
+ listAllElements():List<anEntityTO>
+ crudOperation()
<<implements>>
<<Mandatory>> aBoundary
- em:EntityManager
+ listAllElements():List<anEntityTO>
- getElementsCount():int
+ crudOperation()
1
<<Mandatory>> <<Interface>> IaParallelizer
+ listAllElements(count:int):List<anEntityTO>
<<implements>>
<<Mandatory>> aParallelizer
+ listAllElements(count:int):List<anEntityTO>
<<uses>>
Uses EntityManager to manage other CRUD operations
<<uses>>
<<Mandatory>> anEntityTO
-id:*
-anAttribute:*
get*():*
set(*):void
<<represents>>
1
<<Mandatory>> anEntity
-id:*
-anAttribute:*
get*():*
set(*):void
<<uses>>
Uses DataSource to retrieve data
1
<<Mandatory>> aAsyncWorker
- ds:DataSource
+ listAllElements(start:int, maxResults:int):Future<List<anEntityTO>>
listAllElements is annotated with @Asynchronous

Figure 22. High Async Time Execution Abstract Design

aParallelizer class is in charge of launching the retrieval threads, and assembling their results into one list of records. *aAsyncWorker* provides the implementation of the task that each thread is going to execute. Such task must be of asynchronous nature, given that it is going to be executed by many threads at the same time. Figure 23 illustrates how to concretize this diagram. The interactions of the classes involved in this pattern are shown in figure 24.

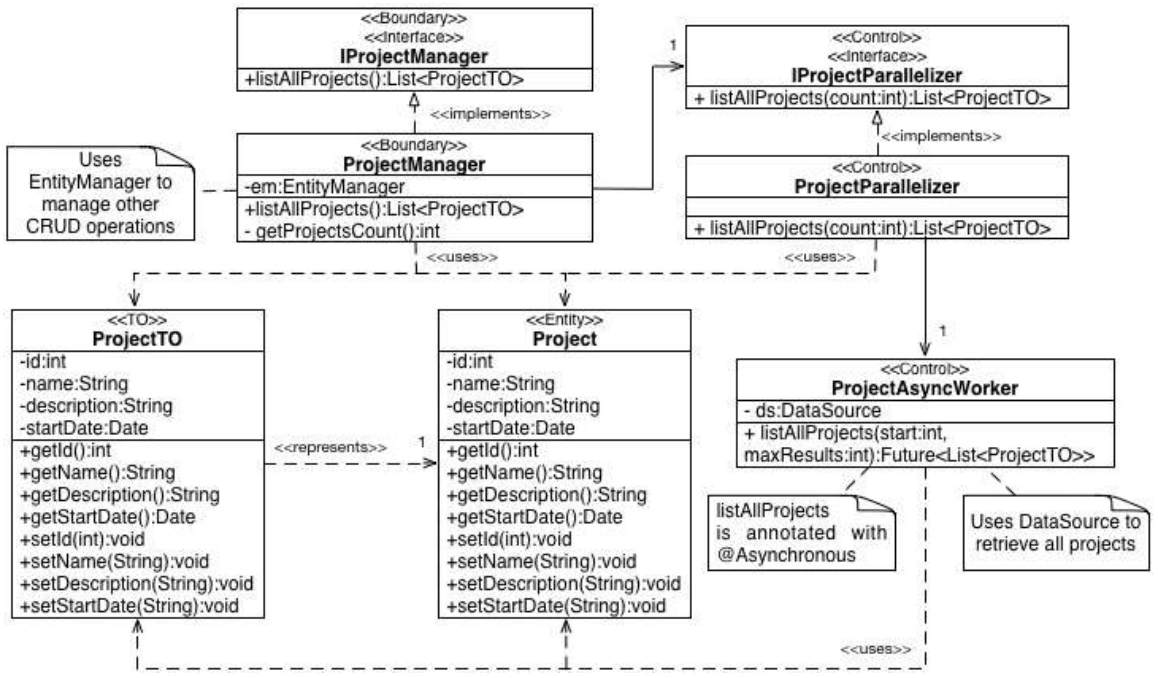
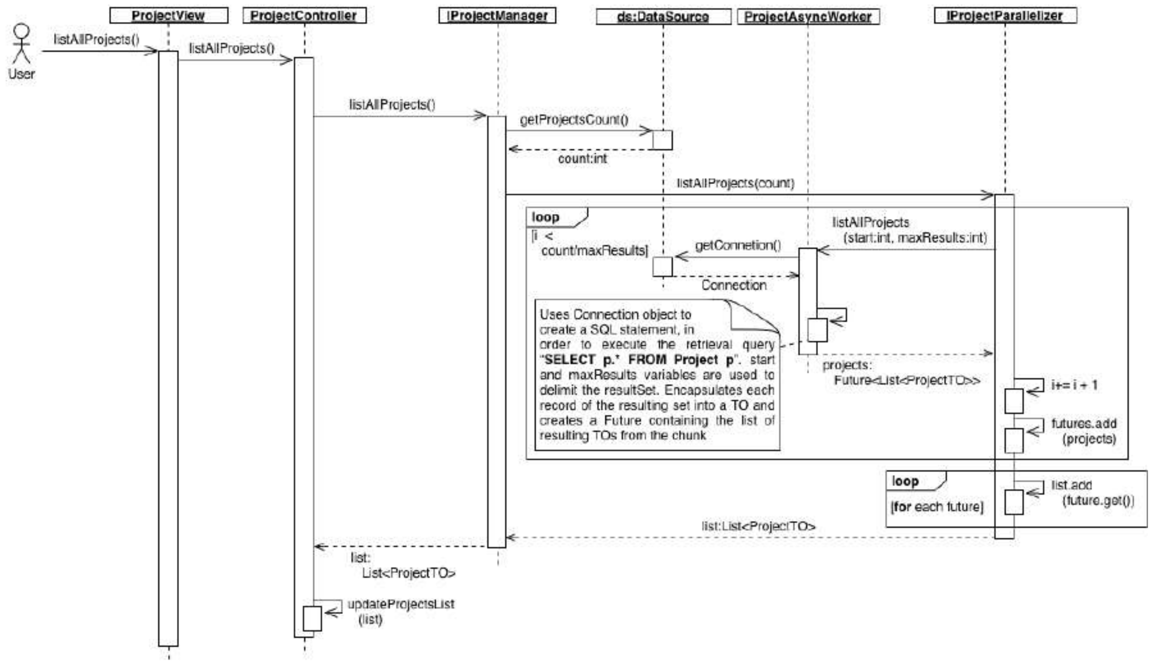


Figure 23. High Async Time Execution Concrete Design



ProjectView
ProjectController
IProjectManager
ds:DataSource
ProjectAsyncWorker
IProjectParallelizer
User
listAllProjects()
listAllProjects()
listAllProjects()
getProjectsCount()
count:int
listAllProjects(count)
loop
[i < count/maxResults]
getConnetion()
Connection
listAllProjects (start:int, maxResults:int)
Uses Connection object to create a SQL statement, in order to execute the retrieval query "SELECT p.* FROM Project p". start and maxResults variables are used to delimit the resultSet. Encapsulates each record of the resulting set into a TO and creates a Future containing the list of resulting TOs from the chunk
projects: Future<List<ProjectTO>>
i+=i+1
futures.add (projects)
loop
[for each future]
list.add (future.get())
list:List<ProjectTO>
list: List<ProjectTO>
updateProjectsList (list)

Figure 24. *ListAll* projects use case sequence diagram (High Async Time Execution)

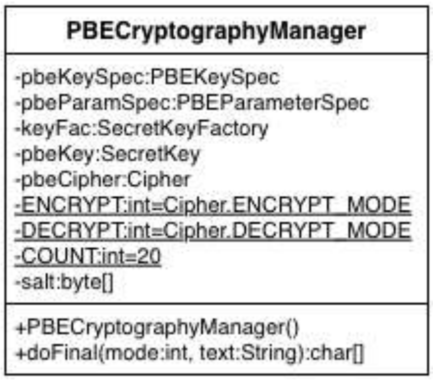
In figure 24, *ProjectParallelizer* has two main loops, one to launch the execution of all instantiated threads, and another one to gather results from all of them. It is important to have two different loops because the use of the `get()` method of a Future freezes the current execution Thread, thus, if it is invoked within the same loop, each thread must be fully executed to launch the next one, causing a sequential retrieval of data. Implementation of `listAllProjects(start, maxResults)` in *ProjectAsyncWorker* class is the same as the one shown in "high sync level" section. The only difference is that the resulting list is wrapped into a Future to enable asynchronous operations. Annex 5 provides an implementation of data retrieval using this pattern.

4.2.2.2.3.2. CONFIDENTIALITY

We determined in figure 8 that this QA affects the implementation of every *Contracts* of the DMM. Given that *Data Unencrypted* level is satisfied using the *ECB* approach, no changes on the internal structure of components will appear. Hence, we will focus on *Data Encrypted* level and its implications.

DATA ENCRYPTED

We selected the Password Based Encryption (PBE) strategy to create a *CryptographyManager* that is responsible of providing entire encryption and decryption services, based on this cryptography method. Figure 25 shows the *CryptographyManager.*



PBECryptographyManager
-pbeKeySpec:PBEKeySpec
-pbeParamSpec:PBEParameterSpec
-keyFac:SecretKeyFactory
-pbeKey:SecretKey
-pbeCipher:Cipher
-ENCRYPT:int=Cipher.ENCRYPT_MODE
-DECRYPT:int=Cipher.DECRYPT_MODE
-COUNT:int=20
-salt:byte[]
+PBECryptographyManager()
+doFinal(mode:int, text:String):char[]

Figure 25. Security-Confidentiality Manager Concrete Design

This manager has to be used as a *Control* by every domain logic component, in order to manage encryption tasks with the data involved in the *Contracts* of each component. To do so, every component must have an instance of this manager (a *Control* per component), causing code duplication. To overcome this issue, we decided to provide this manager as an independent component, which is located in the domain logic layer. This decision enables centralizing confidentiality concerns in one place, plus it promotes reuse of encryption services. Implementing this strategy implies that every domain component must consume the `doFinal` service provided by the *CryptographyManager* every time a contract (component service) is executed, considering the following guidelines:

- Contracts related to *Insert* and *Update* operations must encrypt the data before its storage on the database.
- Rest of the contracts (*Retrieve* and *Delete*) must decrypt the data before sending it to the final user (displaying it on the corresponding GUI).
- Encryption and decryption operations will only be carried out on String data types. This is necessary due to data types used on the database. For example, if a number related to the id of a project is encrypted, an error will occur when and insertion or

deletion is performed, because the encrypted id is represented by a String and the database type related to id's storage is a number.

Keeping in mind the SPL of Project Management systems presented in the case study section, we provide the following diagrams to illustrate how the projects domain component consumes the services of the security component (see figure 26), and to detail encryption and decryption operations. The encryption diagram (see figure 27) is related to the creation of a new project. The decryption diagram (see figure 28) is related to listing all registered projects.

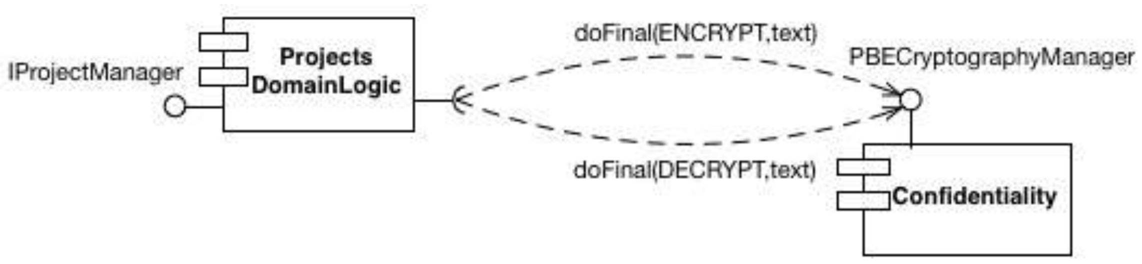


Figure 26. Concrete Interactions with Confidentiality Component

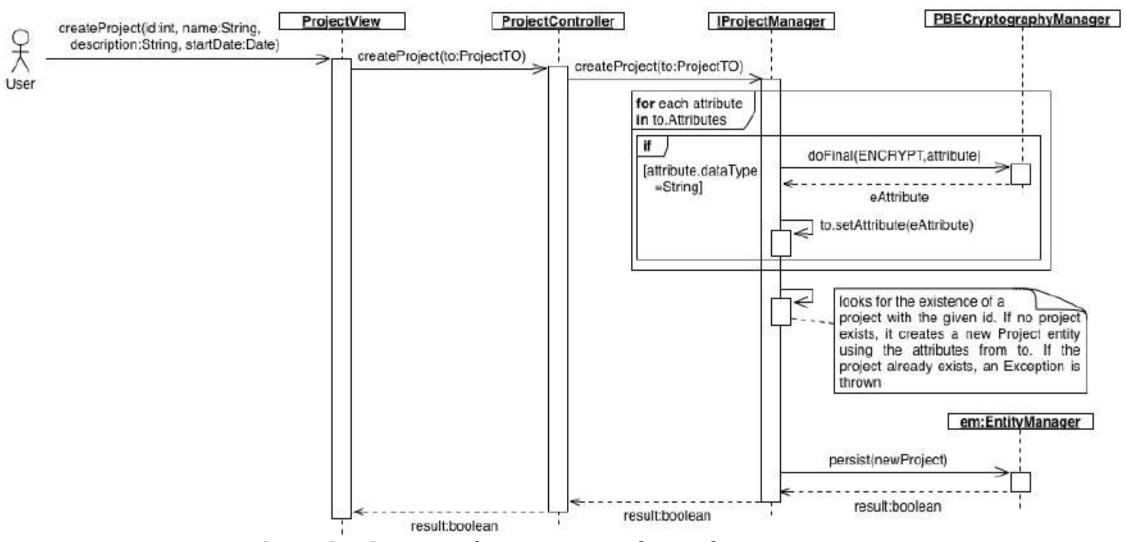

User
createProject(id:int, name:String, description:String, startDate:Date)
ProjectView
createProject(to:ProjectTO)
ProjectController
createProject(to:ProjectTO)
IProjectManager
PBECryptographyManager
for each attribute in to.Attributes
if
[attribute.dataType =String]
doFinal(ENCRYPT,attribute)
eAttribute
to.setAttribute(eAttribute)
looks for the existence of a project with the given id. If no project exists, it creates a new Project entity using the attributes from to. If the project already exists, an Exception is thrown
em:EntityManager
persist(newProject)
result:boolean
result:boolean
result:boolean

Figure 27. Creation of a project considering data encryption

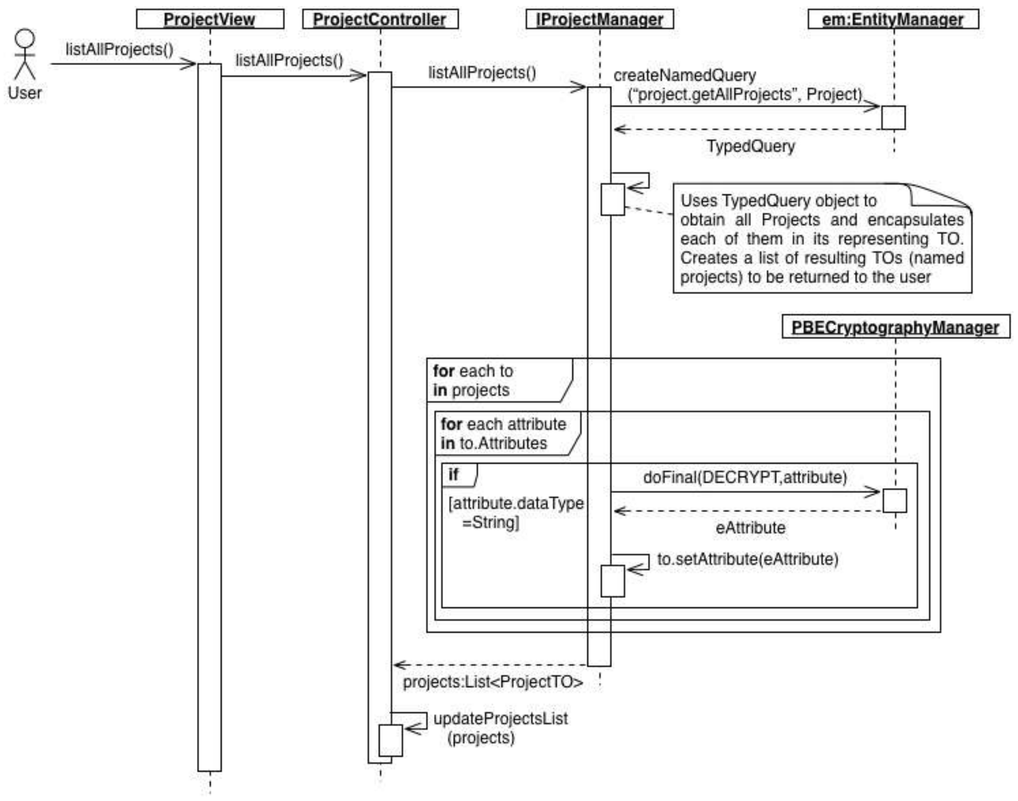

ProjectView
ProjectController
IProjectManager
em:EntityManager
User
listAllProjects()
listAllProjects()
listAllProjects()
createNamedQuery ("project.getAllProjects", Project)
TypedQuery
Uses TypedQuery object to obtain all Projects and encapsulates each of them in its representing TO. Creates a list of resulting TOs (named projects) to be returned to the user
PBECryptographyManager
for each to in projects
for each attribute in to.Attributes
if
[attribute.dataType =String]
doFinal(DECRYPT,attribute)
eAttribute
to.setAttribute(eAttribute)
projects:List<ProjectTO>
updateProjectsList (projects)

Figure 28. Retrieving all projects considering data decryption

Notice that the appearance of confidentiality in the previous diagram affects the retrieval process of a normal time execution level (shown in figure 15) by introducing the interaction with the *PBECryptographyManager*. Such interactions must be taken into account when deriving products of the line, and are discussed in section 4.3.3.

4.2.2.2.3.3. INTEGRITY

According to figure 8, this QA affects the implementation of the authenticable *Business Entity*. To promote *Authentication Lockout* level of this QA, we decided to apply the *Account Lockout* pattern. This pattern uses a *LockoutManager* to protect customer accounts from automated password-guessing attacks, by implementing a limit on incorrect password attempts before further attempts are disallowed. Figure 29 shows the structure of this pattern.

This pattern uses an additional *Entity* named *Attempt*, which is in charge of persist the login attempts for each user of the system. This implies modifying the database of the derived products when this level of QA is desired. `getAttempts` service retrieves the remaining attempts of a given user. `setAttempts` service updates the remaining attempts of a given user. `AuthEntity` refers to the *BusinessEntity* with its *isAuthenticable* attribute set to true. To concretize the previous diagram we will use the authenticable *BusinessEntity* from our case study, *User*. Figure 30 shows how the concretized elements interact with each other when an authentication is required.

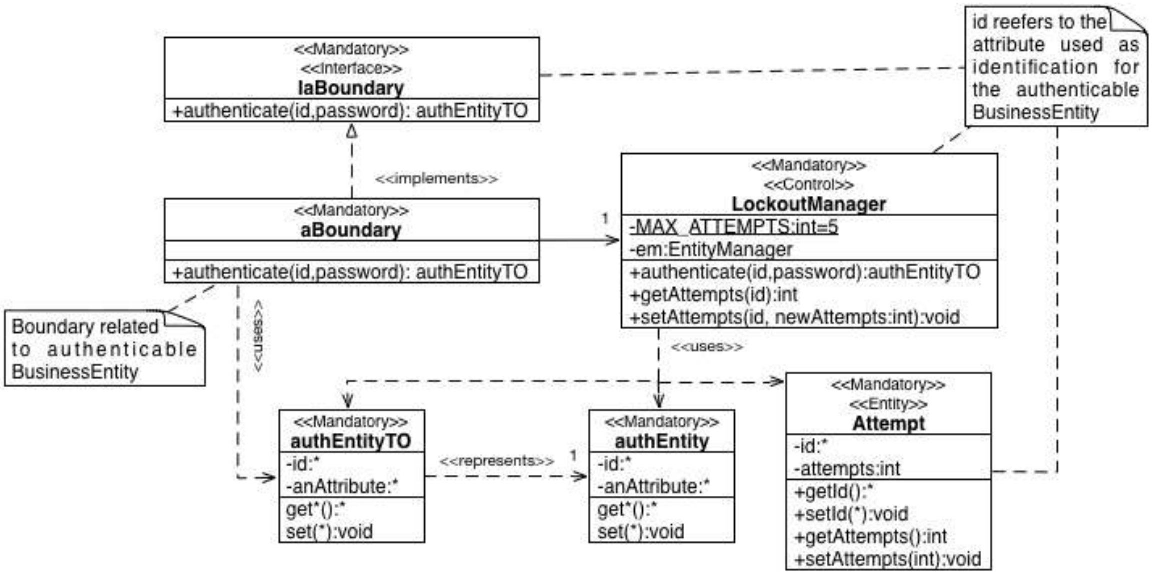


Figure 29. Security-Integrity Abstract Design

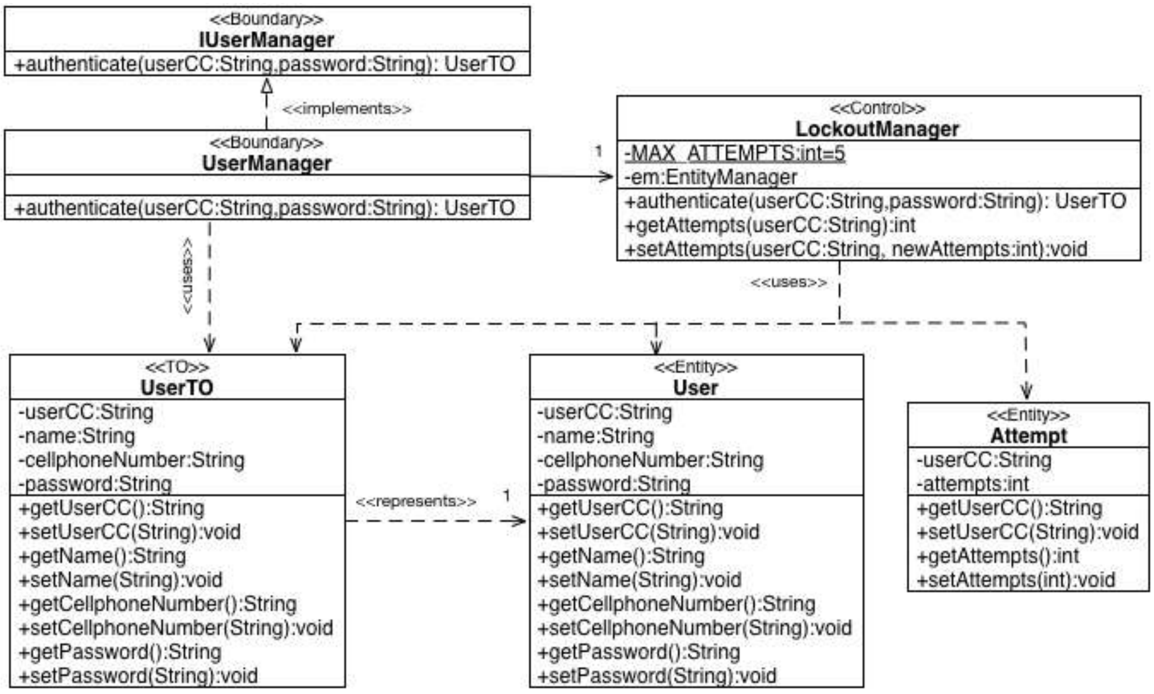


Figure 30. Security-Integrity Concrete Design

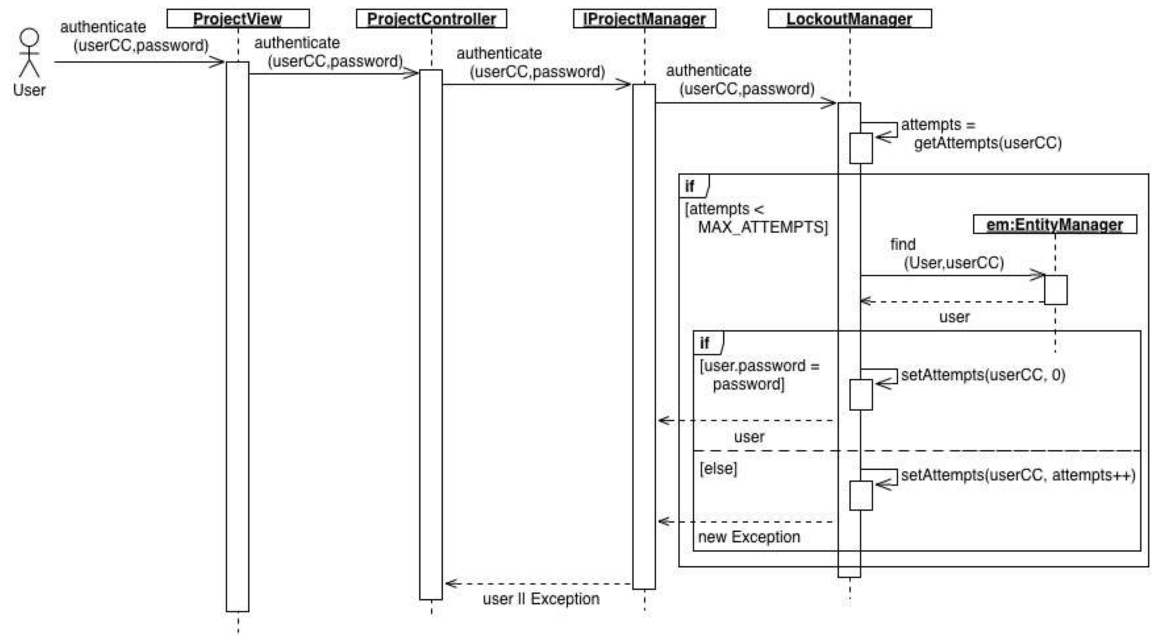

ProjectView
ProjectController
IProjectManager
LockoutManager
User
authenticate (userCC,password)
authenticate (userCC,password)
authenticate (userCC,password)
authenticate (userCC,password)
attempts = getAttempts(userCC)
if
[attempts < MAX_ATTEMPTS]
em:EntityManager
find (User,userCC)
user
if
[user.password = password]
setAttempts(userCC, 0)
user
[else]
setAttempts(userCC, attempts++)
new Exception
user II Exception

Figure 31. Authentication process considering Integrity (Account Lockout)

Notice how *LockoutManager* uses the *EntityManager* to retrieve the user from the database. Once the user is in memory, then a password match is performed. If both passwords (retrieved and provided) match, the user attempts are reset and the authentication process finishes. In case the match fails, the remaining login attempts of the user are reduced and the authentication process fails.

4.2.2.2.3.4. AUTHENTICITY

In order to promote this QA, we defined the *Authorization* level, which enables the system to provide a type of access control to its functionalities. We decided to use the *Role-Based Access Control* pattern, because it describes how to assign rights based on the functionalities of users in an environment in which control of access to computing resources like system data, is required, which accommodates to our expectations of *Authenticity*. The implementation of this pattern implies modifying the database of the system, in order to persist roles and rights involved in the configured product. Fernandez in [100] provides the following structure (see figure 32) to implement the *Role-Based Access Control* pattern.

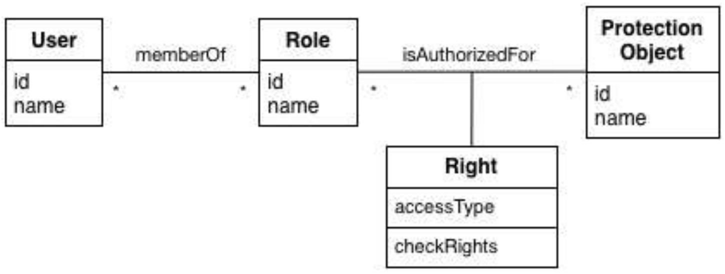


Figure 32. Role-Based Access Control Pattern

The *User* and *Role* classes describe registered users and their predefined roles respectively. *Users* are assigned to roles; roles are given rights according to their functions. The association class *Right* defines the access types that a user within a role is authorized to apply to the protection object. We decided to apply this pattern at database level, and we provided a domain logic component to access and manage these tables. The following diagrams depict how we adapted this pattern to be applied in our SPL (general structure in figure 33), and an example of how to concretize this pattern (see figure 34) with our case study. Notice that our protection object is *Service*, which represents each functionality of the product, i.e. *Create Project* and *List All Projects*. Thus, every functionality of a product has a corresponding Service object. This particular quality level is different from the previous, given that it requires providing a user interface to manage roles, services and their relationships to users of the system. For this reason we designed *IAuthorizationManager* interface, in order to provide proper encapsulation of the services needed to accomplish these tasks.

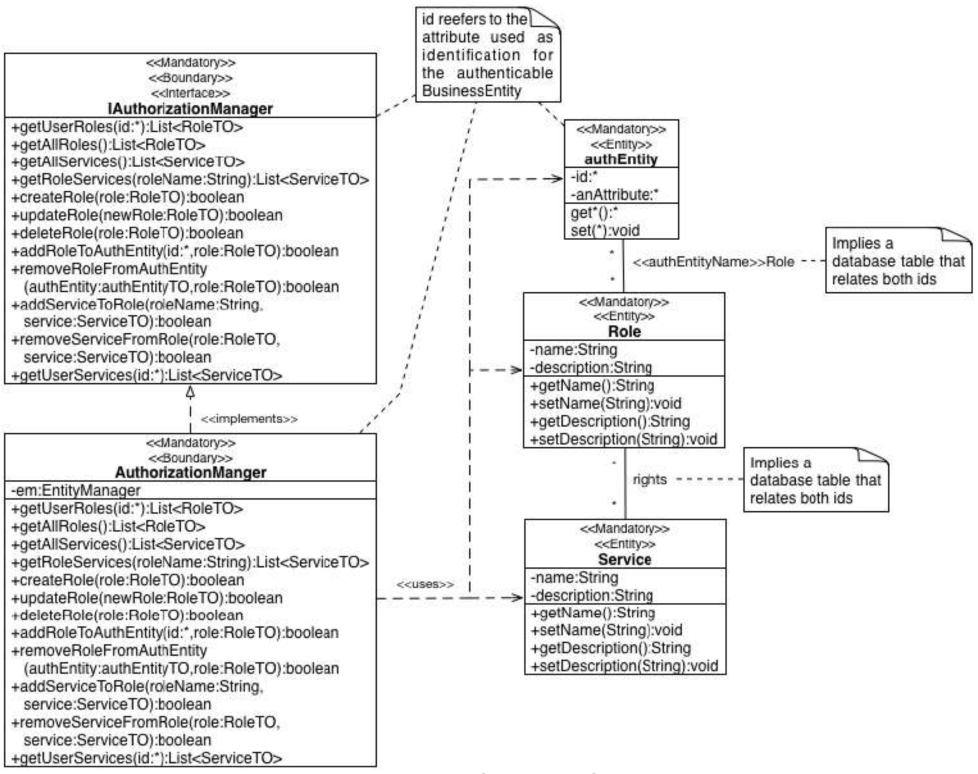
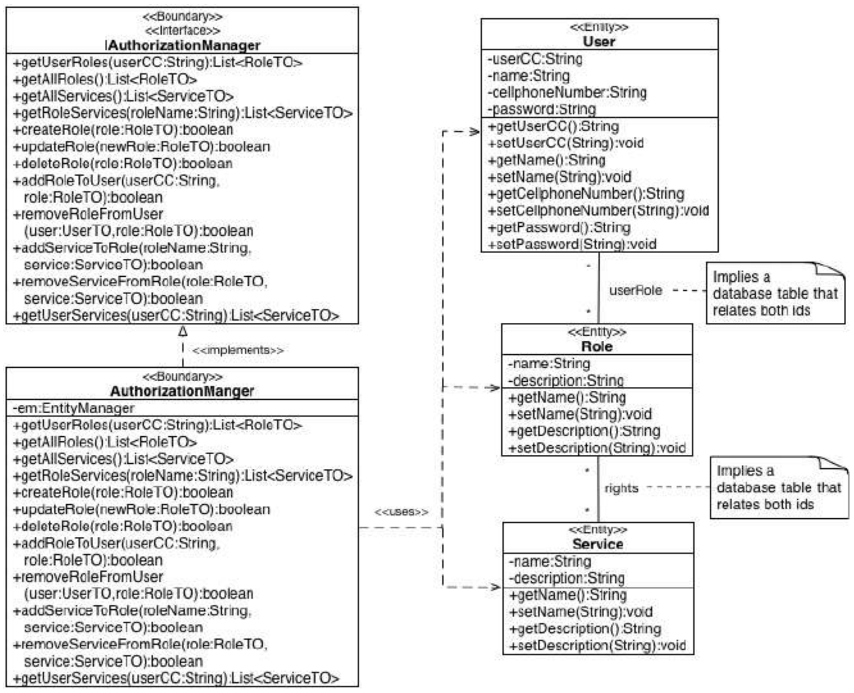

id reefers to the attribute used as identification for the authenticable BusinessEntity
<<Mandatory>> <<Boundary>> <<Interface>> IAuthorizationManager
+getUserRoles(id:*):List<RoleTO>
+getAllRoles():List<RoleTO>
+getAllServices():List<ServiceTO>
+getRoleServices(roleName:String):List<ServiceTO>
+createRole(role:RoleTO):boolean
+updateRole(newRole:RoleTO):boolean
+deleteRole(role:RoleTO):boolean
+addRoleToAuthEntity(id:*,role:RoleTO):boolean
+removeRoleFromAuthEntity (authEntity:authEntityTO,role:RoleTO):boolean
+addServiceToRole(roleName:String, service:ServiceTO):boolean
+removeServiceFromRole(role:RoleTO, service:ServiceTO):boolean
+getUserServices(id:*):List<ServiceTO>
<<implements>>
<<Mandatory>> <<Boundary>> AuthorizationManger
-em:EntityManager
+getUserRoles(id:*):List<RoleTO>
+getAllRoles():List<RoleTO>
+getAllServices():List<ServiceTO>
+getRoleServices(roleName:String):List<ServiceTO>
+createRole(role:RoleTO):boolean
+updateRole(newRole:RoleTO):boolean
+deleteRole(role:RoleTO):boolean
+addRoleToAuthEntity(id:*,role:RoleTO):boolean
+removeRoleFromAuthEntity (authEntity:authEntityTO,role:RoleTO):boolean
+addServiceToRole(roleName:String, service:ServiceTO):boolean
+removeServiceFromRole(role:RoleTO, service:ServiceTO):boolean
+getUserServices(id:*):List<ServiceTO>
<<uses>>
<<Mandatory>> <<Entity>> authEntity
-id:*
-anAttribute:*
get*():*
set(*):void
*
<<authEntityName>>Role
Implies a database table that relates both ids
*
<<Mandatory>> <<Entity>> Role
-name:String
-description:String
+getName():String
+setName(String):void
+getDescription():String
+setDescription(String):void
*
rights
Implies a database table that relates both ids
*
<<Mandatory>> <<Entity>> Service
-name:String
-description:String
+getName():String
+setName(String):void
+getDescription():String
+setDescription(String):void

Figure 33. Security-Authenticity Abstract Design



<<Boundary>>
<<Interface>>
IAuthorizationManager
+getUserRoles(userCC:String):List<RoleTO>
+getAllRoles():List<RoleTO>
+getAllServices():List<ServiceTO>
+getRoleServices(roleName:String):List<ServiceTO>
+createRole(role:RoleTO):boolean
+updateRole(newRole:RoleTO):boolean
+deleteRole(role:RoleTO):boolean
+addRoleToUser(userCC:String, role:RoleTO):boolean
+removeRoleFromUser (user:UserTO,role:RoleTO):boolean
+addServiceToRole(roleName:String, service:ServiceTO):boolean
+removeServiceFromRole(role:RoleTO, service:ServiceTO):boolean
+getUserServices(userCC:String):List<ServiceTO>
<<implements>>
<<Boundary>>
AuthorizationManger
-em:EntityManager
+getUserRoles(userCC:String):List<RoleTO>
+getAllRoles():List<RoleTO>
+getAllServices():List<ServiceTO>
+getRoleServices(roleName:String):List<ServiceTO>
+createRole(role:RoleTO):boolean
+updateRole(newRole:RoleTO):boolean
+deleteRole(role:RoleTO):boolean
+addRoleToUser(userCC:String, role:RoleTO):boolean
+removeRoleFromUser (user:UserTO,role:RoleTO):boolean
+addServiceToRole(roleName:String, service:ServiceTO):boolean
+removeServiceFromRole(role:RoleTO, service:ServiceTO):boolean
+getUserServices(userCC:String):List<ServiceTO>
<<uses>>
<<Entity>>
User
-userCC:String
-name:String
-cellphoneNumber:String
-password:String
+getUserCC():String
+setUserCC(String):void
+getName():String
+setName(String):void
+getCellphoneNumber():String
+setCellphoneNumber(String):void
+getPassword():String
+setPassword(String):void
userRole
Implies a database table that relates both ids
<<Entity>>
Role
-name:String
-description:String
+getName():String
+setName(String):void
+getDescription():String
+setDescription(String):void
rights
Implies a database table that relates both ids
<<Entity>>
Service
-name:String
-description:String
+getName():String
+setName(String):void
+getDescription():String
+setDescription(String):void

Figure 34. Security-Authenticity Concrete Design

Figure 35 illustrates how we use the *Role-Based Access Control* pattern to authorize the execution of a particular functionality. To do so, we recline on `authorize` service provided by *AuthorizerController*, which is the UI controller of authenticity matters. This service consumes `getUserServices` service provided by *AuthorizationManager* to check if a user is authorized to use a particular functionality (service). This diagram is an example of interaction among presentation components (*Authenticity* and *ProjectPresentation*).

Once we have detailed the specification of our Reference Architecture on both macro and micro levels, we exhibit how to use it to construct the SPL of Product Management systems presented in section 3. We assume that the Project Management SPL is configured to promote a *Medium* level of *Time Execution*, a *Data Encrypted* level of *Confidentiality*, a *Lockout* level of *Integrity* and an *Authorization* level of *Authenticity*. The macro architecture of the Project Management SPL is shown in figures 36-38. It is important to clarify that *AuthenticityPresentation*, *Confidentiality*, *PersistenceContext* and *JDBC* components are the same for the entire product, but they are replicated due to space limitations and ease of visualization.

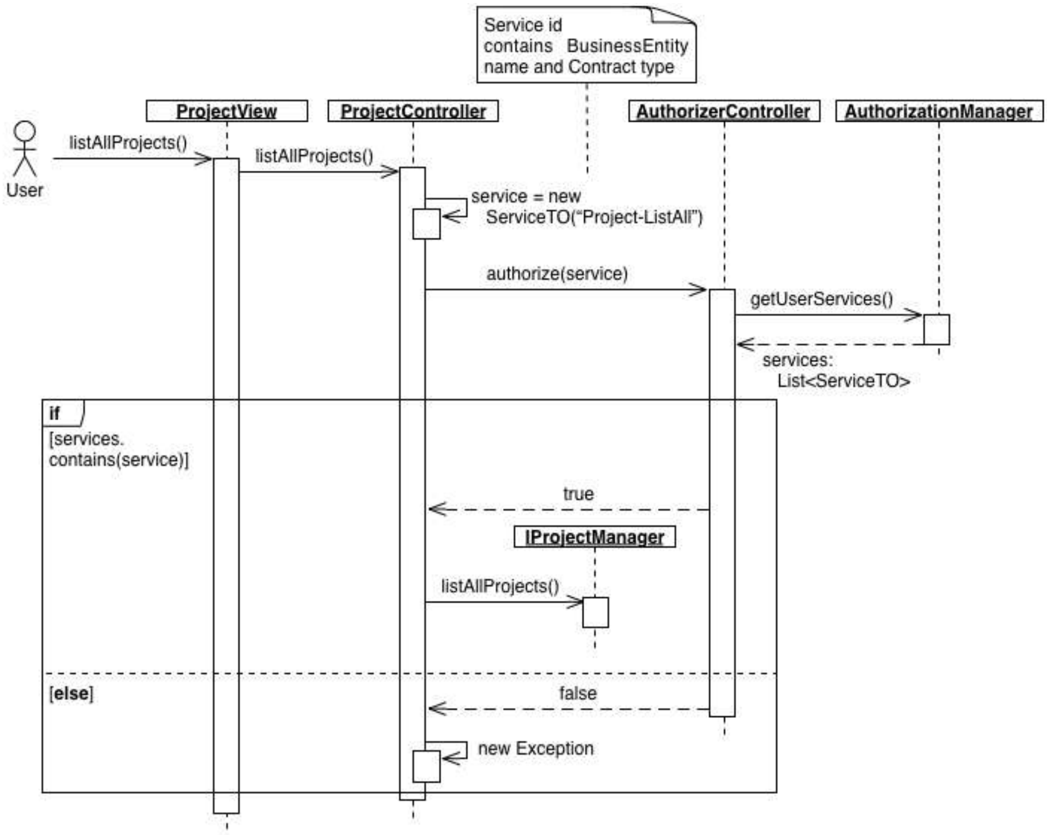


Figure 35. Authorization sequence diagram

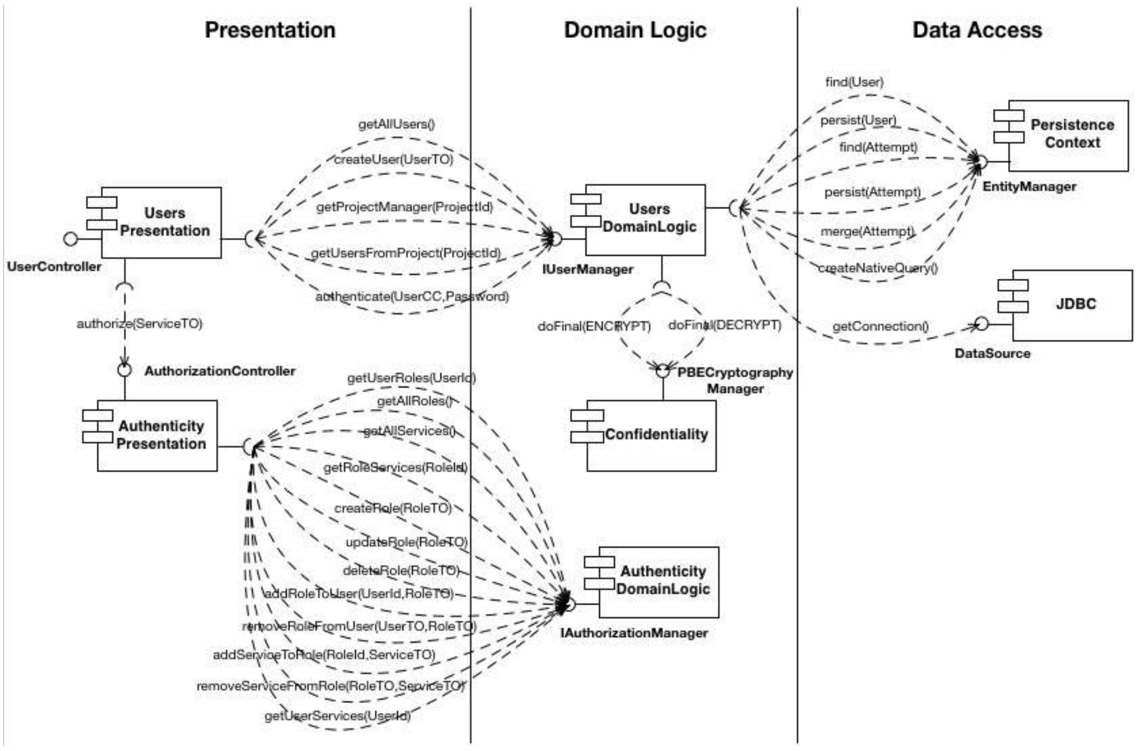


Figure 36. Case Study Macroarchitecture - Users

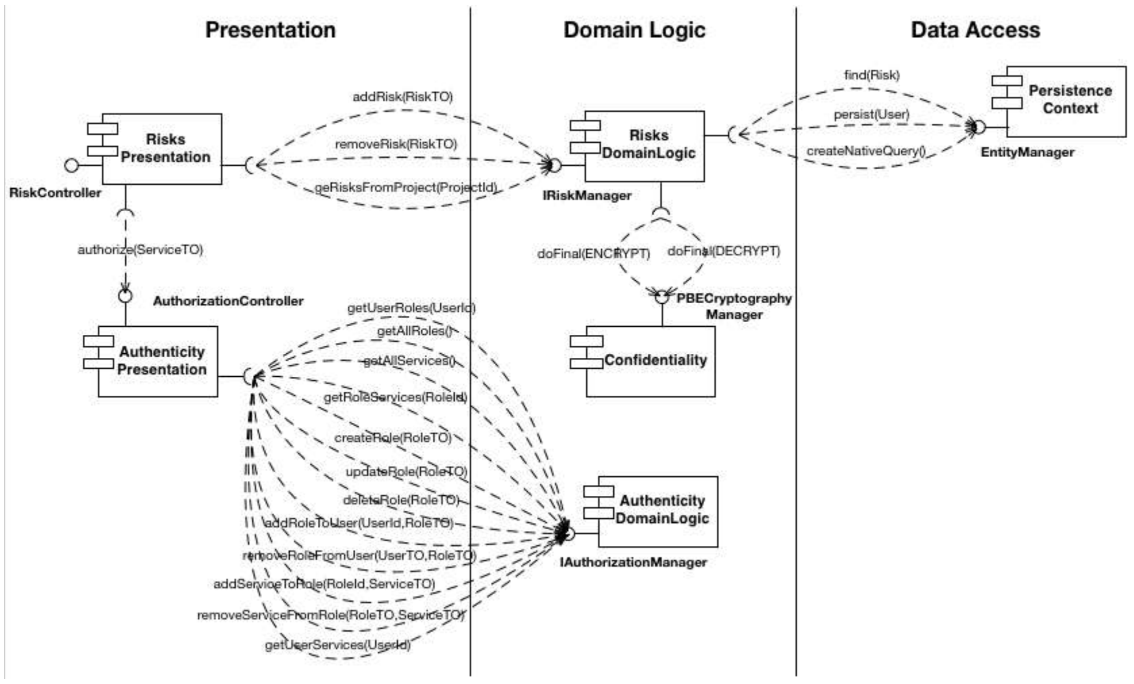


Figure 37. Case Study Macroarchitecture - Risks

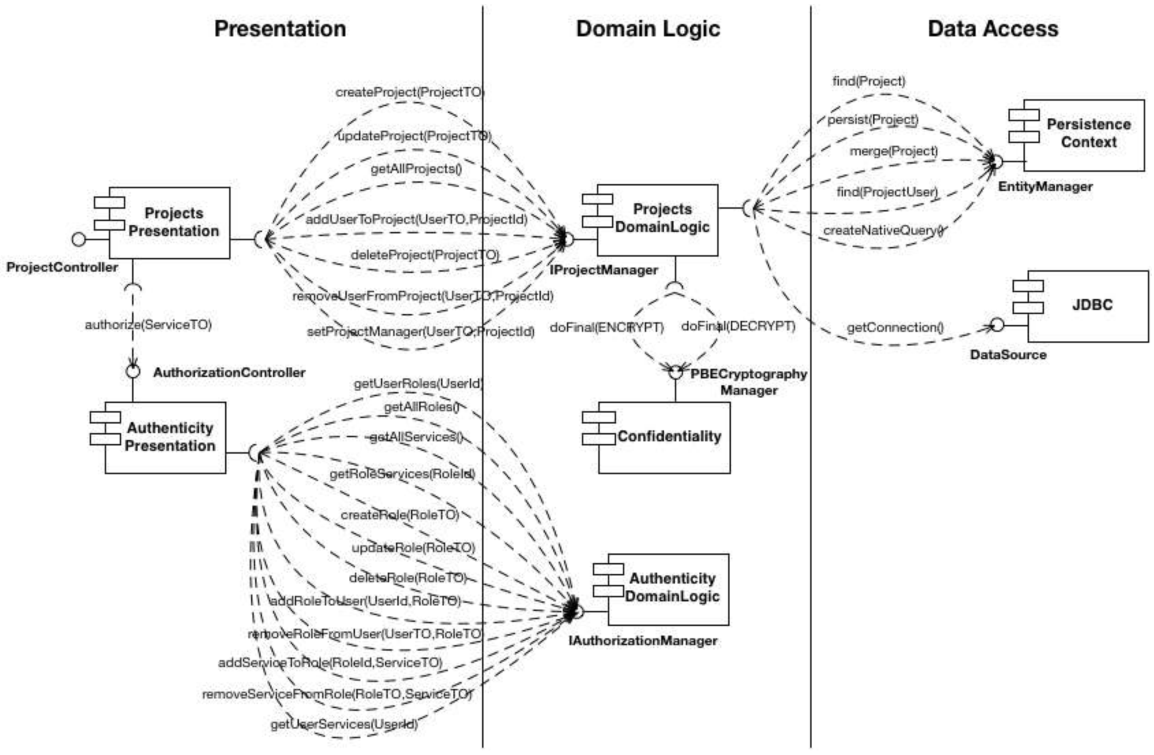
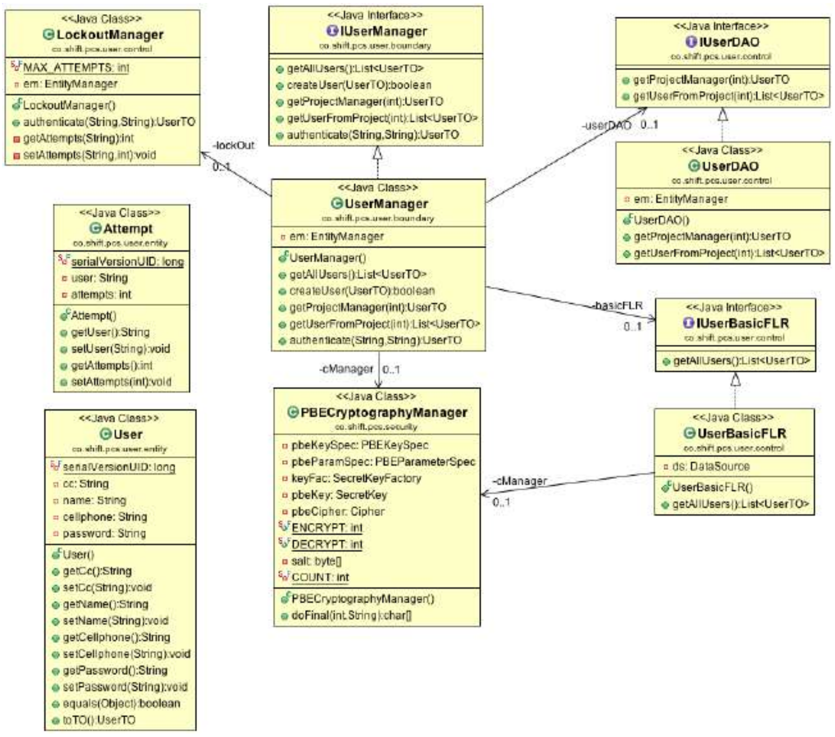


Figure 38. Case Study Macroarchitecture - Projects

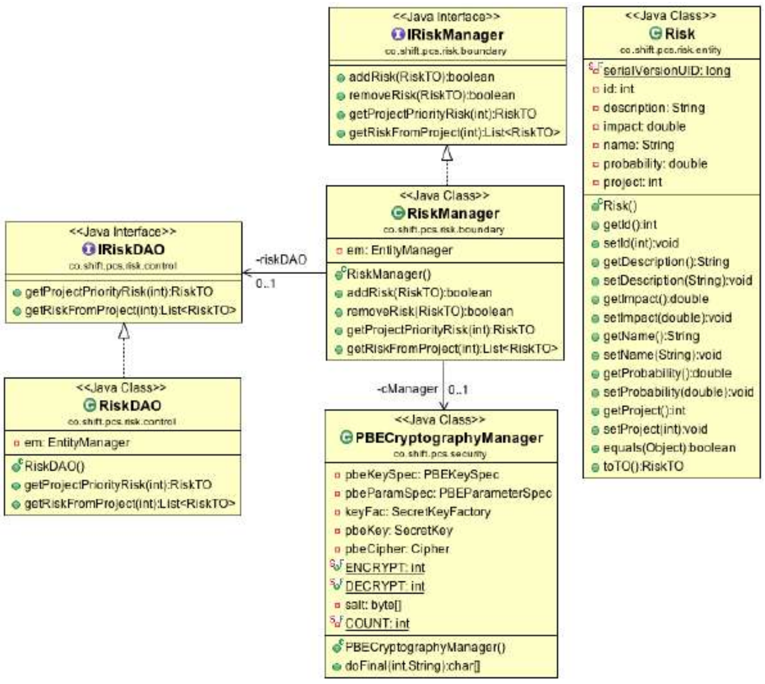
Notice that risks components do not involve the *JDBC* component due to *Risk BusinessEntity* nature as a detail of *Projects* (doesn't involve *ListAll* contracts). The *createNamedQuery* service provided by the *EntityManager* is used to build up queries related to deletion operations and retrieval of related entities through the corresponding

DAOs. To depict the microarchitecture of our case study, we only provide the class diagrams (see figures 39 to 43) of the domain components (business logic). We omit sequence diagrams because all of them are based on the ones shown in section 4.2.2.2; they are adapted according to business entities characteristics implementing functionalities.



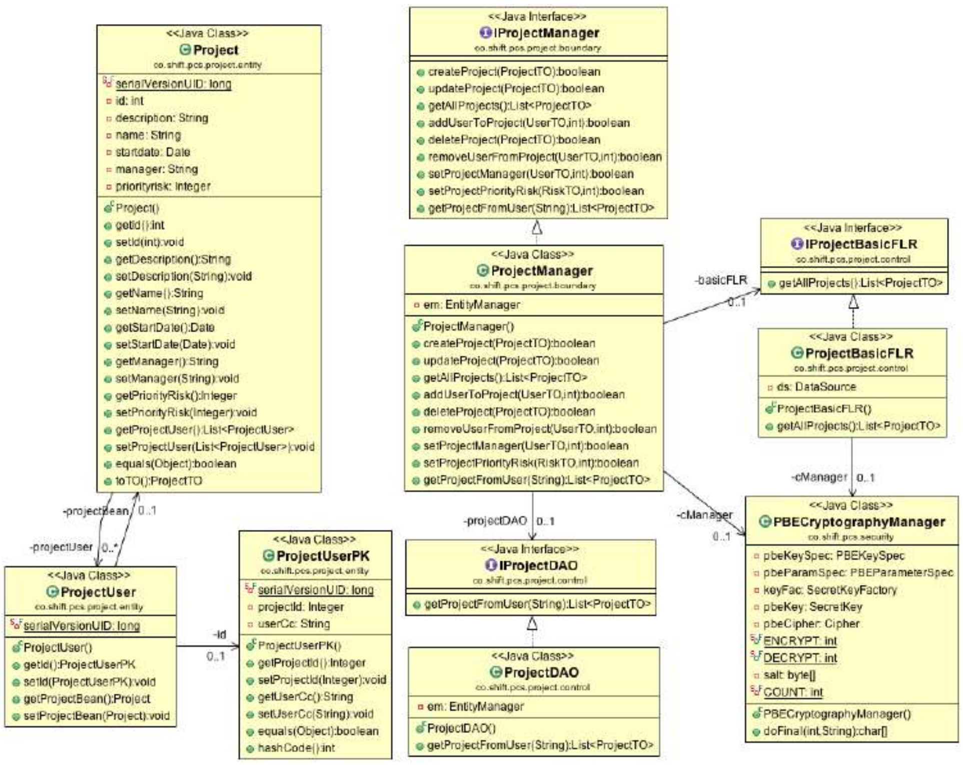
<<Java Class>> LockoutManager co.shift.pcs.user.control
MAX_ATTEMPTS: int
em: EntityManager
LockoutManager()
authenticate(String,String):UserTO
getAttempts(String):int
setAttempts(String,int):void
<<Java Interface>> IUserManager co.shift.pcs.user.boundary
getAllUsers():List<UserTO>
createUser(UserTO):boolean
getProjectManager(int):UserTO
getUserFromProject(int):List<UserTO>
authenticate(String,String):UserTO
<<Java Interface>> IUserDAO co.shift.pcs.user.control
getProjectManager(int):UserTO
getUserFromProject(int):List<UserTO>
-lockOut
0..1
-userDAO 0..1
<<Java Class>> UserDAO co.shift.pcs.user.control
em: EntityManager
UserDAO()
getProjectManager(int):UserTO
getUserFromProject(int):List<UserTO>
<<Java Class>> UserManager co.shift.pcs.user.boundary
em: EntityManager
UserManager()
getAllUsers():List<UserTO>
createUser(UserTO):boolean
getProjectManager(int):UserTO
getUserFromProject(int):List<UserTO>
authenticate(String,String):UserTO
<<Java Class>> Attempt co.shift.pcs.user.entity
serialVersionUID: long
user: String
attempts: int
Attempt()
getUser():String
setUser(String):void
getAttempts():int
setAttempts(int):void
-basicFLR
0..1
<<Java Interface>> IUserBasicFLR co.shift.pcs.user.control
getAllUsers():List<UserTO>
-cManager 0..1
<<Java Class>> User co.shift.pcs.user.entity
serialVersionUID: long
cc: String
name: String
cellphone: String
password: String
User()
getCc():String
setCc(String):void
getName():String
setName(String):void
getCellphone():String
setCellphone(String):void
getPassword():String
setPassword(String):void
equals(Object):boolean
toTO():UserTO
<<Java Class>> PBECryptographyManager co.shift.pcs.security
pbeKeySpec: PBEKeySpec
pbeParamSpec: PBEParameterSpec
keyFac: SecretKeyFactory
pbeKey: SecretKey
pbeCipher: Cipher
ENCRYPT: int
DECRYPT: int
salt: byte[]
COUNT: int
PBECryptographyManager()
doFinal(int,String):char[]
<<Java Class>> UserBasicFLR co.shift.pcs.user.control
ds: DataSource
UserBasicFLR()
getAllUsers():List<UserTO>
-cManager
0..1

Figure 39. Case Study Microarchitecture - Users Component



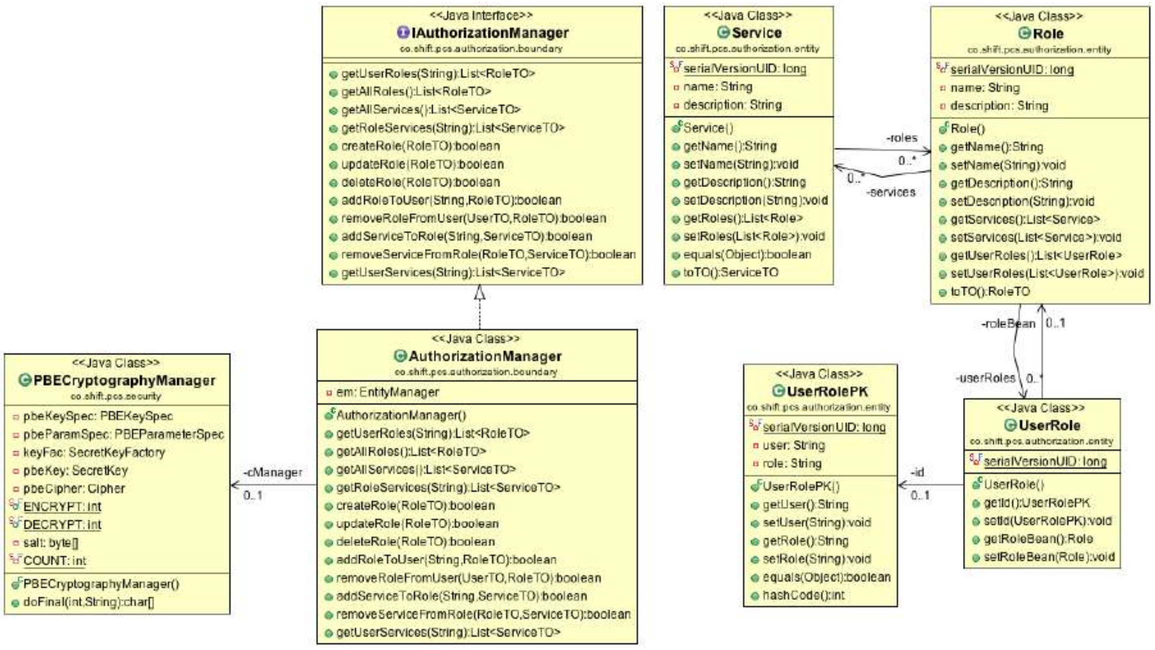
<<Java Interface>> IRiskManager co.shift.pcs.risk.boundary
addRisk(RiskTO):boolean
removeRisk(RiskTO):boolean
getProjectPriorityRisk(int):RiskTO
getRiskFromProject(int):List<RiskTO>
<<Java Class>> Risk co.shift.pcs.risk.entity
serialVersionUID: long
id: int
description: String
impact: double
name: String
probability: double
project: int
Risk()
getId():int
setId(int):void
getDescription():String
setDescription(String):void
getImpact():double
setImpact(double):void
getName():String
setName(String):void
getProbability():double
setProbability(double):void
getProject():int
setProject(int):void
equals(Object):boolean
toTO():RiskTO
<<Java Class>> RiskManager co.shift.pcs.risk.boundary
em: EntityManager
RiskManager()
addRisk(RiskTO):boolean
removeRisk(RiskTO):boolean
getProjectPriorityRisk(int):RiskTO
getRiskFromProject(int):List<RiskTO>
<<Java Interface>> IRiskDAO co.shift.pcs.risk.control
getProjectPriorityRisk(int):RiskTO
getRiskFromProject(int):List<RiskTO>
-riskDAO
0..1
<<Java Class>> RiskDAO co.shift.pcs.risk.control
em: EntityManager
RiskDAO()
getProjectPriorityRisk(int):RiskTO
getRiskFromProject(int):List<RiskTO>
-cManager 0..1
<<Java Class>> PBECryptographyManager co.shift.pcs.security
pbeKeySpec: PBEKeySpec
pbeParamSpec: PBEParameterSpec
keyFac: SecretKeyFactory
pbeKey: SecretKey
pbeCipher: Cipher
ENCRYPT: int
DECRYPT: int
salt: byte[]
COUNT: int
PBECryptographyManager()
doFinal(int,String):char[]

Figure 40. Case Study Microarchitecture - Risks Component



<<Java Class>>
Project
co.shift.pcs.project.entity
serialVersionUID: long
id: int
description: String
name: String
startdate: Date
manager: String
priorityrisk: Integer
Project()
getId():int
setId(int):void
getDescription():String
setDescription(String):void
getName():String
setName(String):void
getStartDate():Date
setStartDate(Date):void
getManager():String
setManager(String):void
getPriorityRisk():Integer
setPriorityRisk(Integer):void
getProjectUser():List<ProjectUser>
setProjectUser(List<ProjectUser>):void
equals(Object):boolean
toTO():ProjectTO
<<Java Interface>>
IProjectManager
co.shift.pcs.project.boundary
createProject(ProjectTO):boolean
updateProject(ProjectTO):boolean
getAllProjects():List<ProjectTO>
addUserToProject(UserTO,int):boolean
deleteProject(ProjectTO):boolean
removeUserFromProject(UserTO,int):boolean
setProjectManager(UserTO,int):boolean
setProjectPriorityRisk(RiskTO,int):boolean
getProjectFromUser(String):List<ProjectTO>
<<Java Class>>
ProjectManager
co.shift.pcs.project.boundary
em: EntityManager
ProjectManager()
createProject(ProjectTO):boolean
updateProject(ProjectTO):boolean
getAllProjects():List<ProjectTO>
addUserToProject(UserTO,int):boolean
deleteProject(ProjectTO):boolean
removeUserFromProject(UserTO,int):boolean
setProjectManager(UserTO,int):boolean
setProjectPriorityRisk(RiskTO,int):boolean
getProjectFromUser(String):List<ProjectTO>
-basicFLR
0..1
<<Java Interface>>
IProjectBasicFLR
co.shift.pcs.project.control
getAllProjects():List<ProjectTO>
<<Java Class>>
ProjectBasicFLR
co.shift.pcs.project.control
ds: DataSource
ProjectBasicFLR()
getAllProjects():List<ProjectTO>
-cManager
0..1
-projectBean
0..1
-projectUser
0..*
<<Java Class>>
ProjectUser
co.shift.pcs.project.entity
serialVersionUID: long
ProjectUser()
getId():ProjectUserPK
setId(ProjectUserPK):void
getProjectBean():Project
setProjectBean(Project):void
-id
0..1
<<Java Class>>
ProjectUserPK
co.shift.pcs.project.entity
serialVersionUID: long
projectId: Integer
userCc: String
ProjectUserPK()
getProjectId():Integer
setProjectId(Integer):void
getUserCc():String
setUserCc(String):void
equals(Object):boolean
hashCode():int
-projectDAO
0..1
<<Java Interface>>
IProjectDAO
co.shift.pcs.project.control
getProjectFromUser(String):List<ProjectTO>
<<Java Class>>
ProjectDAO
co.shift.pcs.project.control
em: EntityManager
ProjectDAO()
getProjectFromUser(String):List<ProjectTO>
-cManager
0..1
<<Java Class>>
PBECryptographyManager
co.shift.pcs.security
pbeKeySpec: PBEKeySpec
pbeParamSpec: PBEParameterSpec
keyFac: SecretKeyFactory
pbeKey: SecretKey
pbeCipher: Cipher
ENCRYPT: int
DECRYPT: int
salt: byte[]
COUNT: int
PBECryptographyManager()
doFinal(int,String):char[]

Figure 41. Case Study Microarchitecture - Projects Component



<<Java Interface>>
IAuthorizationManager
co.shift.pcs.authorization.boundary
getUserRoles(String):List<RoleTO>
getAllRoles():List<RoleTO>
getAllServices():List<ServiceTO>
getRoleServices(String):List<ServiceTO>
createRole(RoleTO):boolean
updateRole(RoleTO):boolean
deleteRole(RoleTO):boolean
addRoleToUser(String,RoleTO):boolean
removeRoleFromUser(UserTO,RoleTO):boolean
addServiceToRole(String,ServiceTO):boolean
removeServiceFromRole(RoleTO,ServiceTO):boolean
getUserServices(String):List<ServiceTO>
<<Java Class>>
Service
co.shift.pcs.authorization.entity
serialVersionUID: long
name: String
description: String
Service()
getName():String
setName(String):void
getDescription():String
setDescription(String):void
getRoles():List<Role>
setRoles(List<Role>):void
equals(Object):boolean
toTO():ServiceTO
-roles
0..*
0..*
-services
<<Java Class>>
Role
co.shift.pcs.authorization.entity
serialVersionUID: long
name: String
description: String
Role()
getName():String
setName(String):void
getDescription():String
setDescription(String):void
getServices():List<Service>
setServices(List<Service>):void
getUserRoles():List<UserRole>
setUserRoles(List<UserRole>):void
toTO():RoleTO
-roleBean
0..1
-userRoles
0..*
<<Java Class>>
PBECryptographyManager
co.shift.pcs.security
pbeKeySpec: PBEKeySpec
pbeParamSpec: PBEParameterSpec
keyFac: SecretKeyFactory
pbeKey: SecretKey
pbeCipher: Cipher
ENCRYPT: int
DECRYPT: int
salt: byte[]
COUNT: int
PBECryptographyManager()
doFinal(int,String):char[]
-cManager
0..1
<<Java Class>>
AuthorizationManager
co.shift.pcs.authorization.boundary
em: EntityManager
AuthorizationManager()
getUserRoles(String):List<RoleTO>
getAllRoles():List<RoleTO>
getAllServices():List<ServiceTO>
getRoleServices(String):List<ServiceTO>
createRole(RoleTO):boolean
updateRole(RoleTO):boolean
deleteRole(RoleTO):boolean
addRoleToUser(String,RoleTO):boolean
removeRoleFromUser(UserTO,RoleTO):boolean
addServiceToRole(String,ServiceTO):boolean
removeServiceFromRole(RoleTO,ServiceTO):boolean
getUserServices(String):List<ServiceTO>
<<Java Class>>
UserRolePK
co.shift.pcs.authorization.entity
serialVersionUID: long
user: String
role: String
UserRolePK()
getUser():String
setUser(String):void
getRole():String
setRole(String):void
equals(Object):boolean
hashCode():int
-id
0..1
<<Java Class>>
UserRole
co.shift.pcs.authorization.entity
serialVersionUID: long
UserRole()
getId():UserRolePK
setId(UserRolePK):void
getRoleBean():Role
setRoleBean(Role):void

Figure 42. Case Study Microarchitecture - Authenticity Component

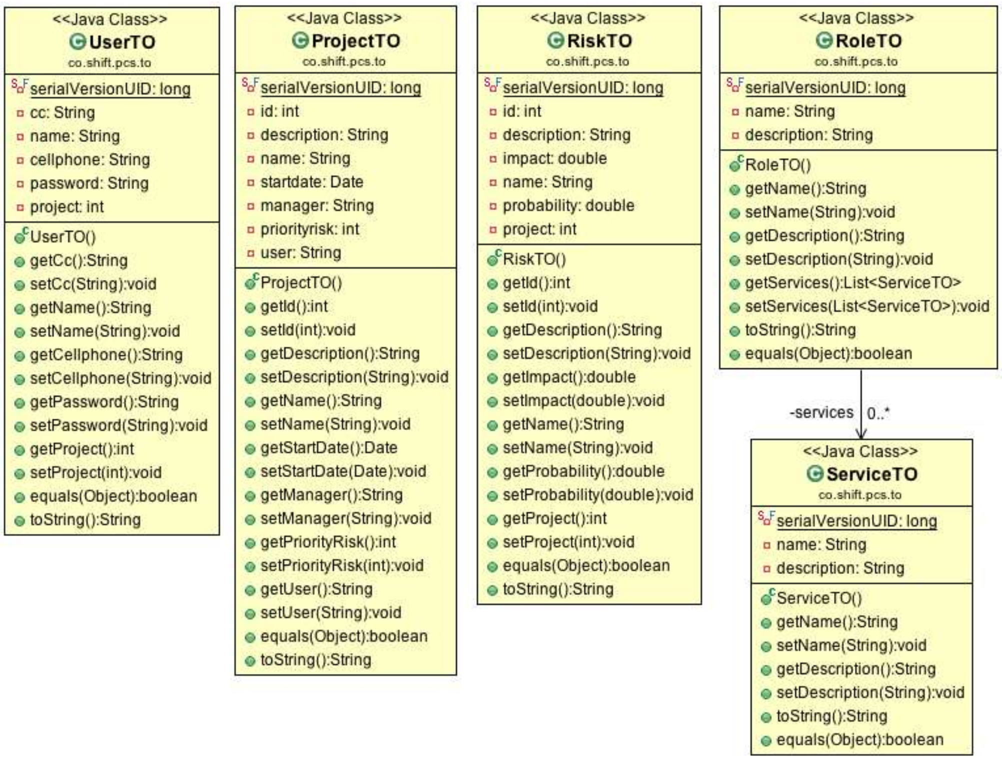


Figure 43. Case Study Microarchitecture - Transfer Objects

These diagrams entirely depend on functionalities and quality levels configured for a particular product, thus, we provide a mechanism to generate the class diagrams of the domain components of a product, parting from a domain model and a quality configuration. Such mechanism is one of our generation artifacts that are going to be described in the following section.

To extend our approach, the product line engineer might select different catalogs from the ones shown in this work, in order to identify varied software design patterns that leverage the promotion of quality levels consigned in the QAs variability model. For each newly design pattern, the product line engineer must provide the necessary design decisions (class and sequence diagrams) to proper implement the pattern when constructing product line members. Thus, the reference architecture must be adapted to support these newly patterns.

4.3. Perform Code Generation

A product line member is the result of transforming a set of functionalities contained in a domain model along with a configuration of quality levels into source code, following the constraints and conditions dictated by our RA. To assist the product line engineer in this process, we provide a mechanism to automate the product generation from the SPL of enterprise applications. To do so, we developed a Model-to-Model (M2M) transformation that takes a domain model and a quality configuration as inputs to select the proper designs for the product line member, according to the constraints of the RA. We also developed several Model-to-Text (M2T) transformations [108] that use the selected designs to generate the proper source code. Following sections explain these transformations.

4.3.1. *Constructing the Product Line Member Concrete Architecture*

In sections above we mentioned that creation of macro and micro architecture diagrams entirely depends on functionalities and quality levels configured for a particular product. Thus, we provide a mechanism to generate the concrete class diagrams of the domain components of a product line member, parting from a domain model and a quality configuration. These concrete diagrams represent the product line member architecture. Such mechanism is a Model-To-Model (M2M) transformation that produces a UML diagram based on the models taken as inputs. The domain model is configured according to the DMM, the quality configuration is specified using the QAs variability model and the resulting UML model is created according to UML2[5], which is an EMF-based implementation of the Unified Modeling Language (UML) 2.x[6] OMG metamodel for the Eclipse platform [109]. Figure 44 depicts this transformation.

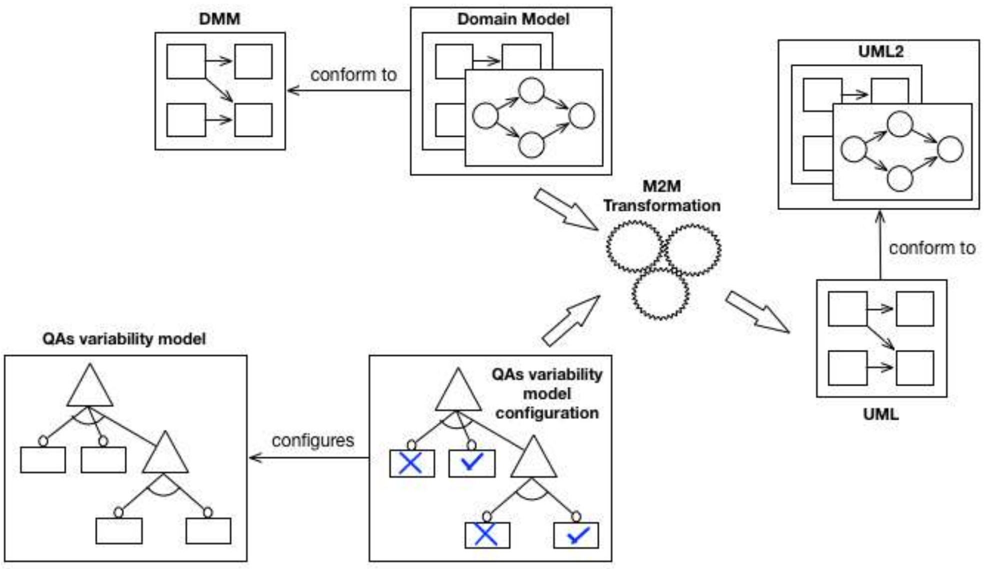
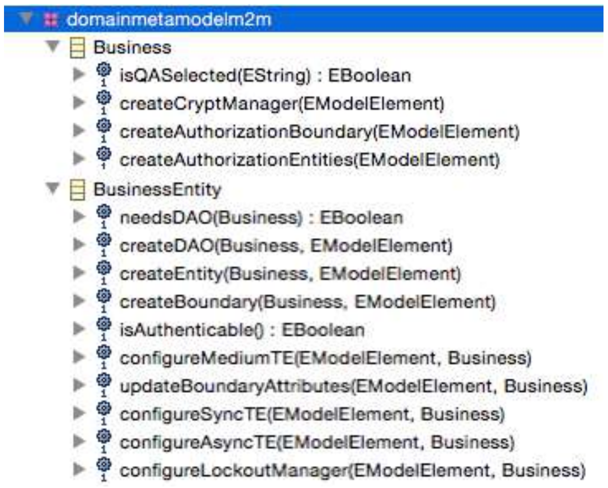


Figure 44. Case Study Microarchitecture

In order to perform the M2M transformation, we used the ATL Transformation Language (ATL)[7]. ATL provides a way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. Transformations created using this language are domain restrictive. This means that every rule can only access information (e.g. attributes) and operations (java methods that must be overridden) defined in the metamodels involved in the transformation, hence, no external libraries can be used. In order to use the utilities of an

[5] http://www.eclipse.org/modeling/mdt/?project=uml2
[6] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML
[7] http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.xtext.doc%2Fcontents%2F118-mwe-in-depth.html

API we provide to manipulate the DMM (explained in section 4.3.3) and the QAs configuration parser (XML reader also explained in section 4.3.3), we need to provide proper operations for each object in the DMM. Figure 45 shows the operations added to the DMM. Table 1 details the purpose of each operation.



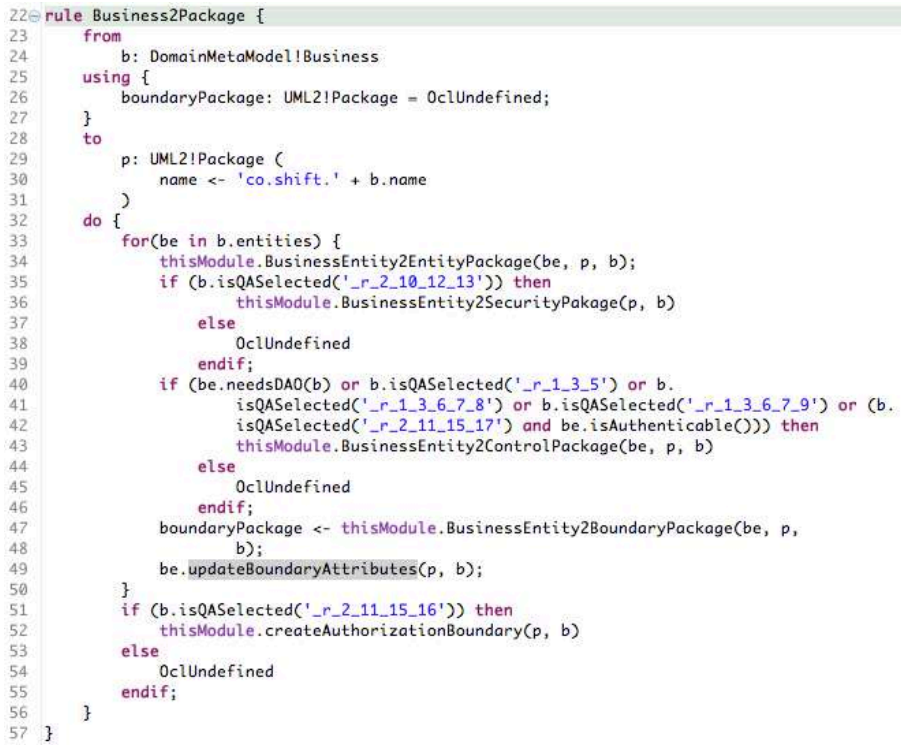
```
domainmetamodelm2m
  Business
    isQASelected(EString) : EBoolean
    createCryptManager(EModelElement)
    createAuthorizationBoundary(EModelElement)
    createAuthorizationEntities(EModelElement)
  BusinessEntity
    needsDAO(Business) : EBoolean
    createDAO(Business, EModelElement)
    createEntity(Business, EModelElement)
    createBoundary(Business, EModelElement)
    isAuthenticable() : EBoolean
    configureMediumTE(EModelElement, Business)
    updateBoundaryAttributes(EModelElement, Business)
    configureSyncTE(EModelElement, Business)
    configureAsyncTE(EModelElement, Business)
    configureLockoutManager(EModelElement, Business)
```

Figure 45. Modifications to our DMM to handle UML transformation

Business	
isQASelected	Validates whether the provided QA id (String) is selected or not in the quality configuration
createCryptManager	Creates the PBECryptographyManager class shown in figure 25
createAuthorizationBoundary	Creates the AuthorizationManager class shown in figure 33
createAuthorizationEntities	Creates the Entities involved in the RBAC pattern (see figure 35)
BusinessEntity	
needsDAO	Validates whether the current BusinessEntity needs a DAO or not (see section 4.2.2.2.2)
createDAO	Creates a DAO class when needed
createEntity	Creates the JPA Entity that represents the current BusinessEntity
createBoundary	Creates the EJB Bean that represents the current BusinessEntity
isAuthenticable	Validates whether a BusinessEntity is authenticable or not
configureMediumTE	Creates all the classes involved in the medium level of time execution (see figure 16)
updateBoundaryAttributes	Updates the relationships of the Boundary of the current BusinessEntity
configureSyncTE	Creates all the classes involved in the high sync level of time execution (see figure 19)
configureAsyncTE	Creates all the classes involved in the high async level of time execution (see figure 22)
configureLockoutManager	Creates the classes involved in the lockout level of integrity (see figure 29)

Table 1. Operations description

ATL defines two different kinds of transformation rules: the matched and the called rules. A matched rule enables to match some of the model elements of a source model, and to generate from them a number of distinct target model elements. As opposed to matched rules, a called rule has to be invoked from an ATL imperative block in order to be executed. The code fragment shown in figure 46 depicts the implementation of a matched rule in our transformation that takes a *Business* object as input and generates all UML classes for the domain components involved, e.g. *Boundaries*, *Controls*, *and Entities*.

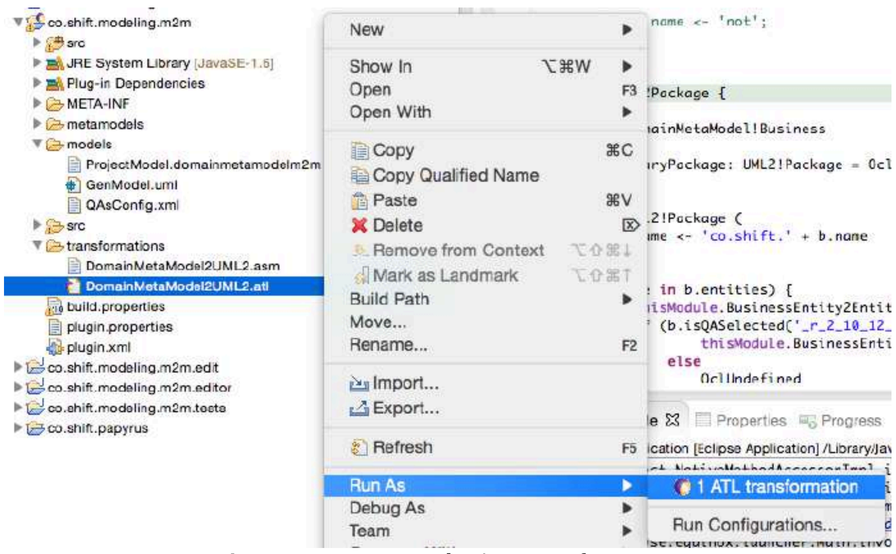


```
22 rule Business2Package {
23     from
24         b: DomainMetaModel!Business
25     using {
26         boundaryPackage: UML2!Package = OclUndefined;
27     }
28     to
29         p: UML2!Package (
30             name <- 'co.shift.' + b.name
31         )
32     do {
33         for(be in b.entities) {
34             thisModule.BusinessEntity2EntityPackage(be, p, b);
35             if (b.isQASelected('_r_2_10_12_13')) then
36                     thisModule.BusinessEntity2SecurityPakage(p, b)
37                 else
38                     OclUndefined
39                 endif;
40             if (be.needsDAO(b) or b.isQASelected('_r_1_3_5') or b.
41                     isQASelected('_r_1_3_6_7_8') or b.isQASelected('_r_1_3_6_7_9') or (b.
42                     isQASelected('_r_2_11_15_17') and be.isAuthenticable())) then
43                     thisModule.BusinessEntity2ControlPackage(be, p, b)
44                 else
45                     OclUndefined
46                 endif;
47             boundaryPackage <- thisModule.BusinessEntity2BoundaryPackage(be, p,
48                     b);
49             be.updateBoundaryAttributes(p, b);
50         }
51         if (b.isQASelected('_r_2_11_15_16')) then
52             thisModule.createAuthorizationBoundary(p, b)
53         else
54             OclUndefined
55         endif;
56     }
57 }
```

Figure 46. ATL transformation (code fragment)

Lines 23 and 24 specify the rule input, which is a *Business* object of our DMM. Line 25 and 26 declare a variable of type *Package* from UML2 metamodel. Lines 28 to 31 declare the target model element (a *Package* from UML2 metamodel in this case). Lines 32 to 56 conform a declarative block, indicating what to do with each entity contained in the *Business* input. Line 34 creates an entity package, containing all Entities related to the current *BusinessEntity*. This is an example of how a called rule is invoked. Lines 35 to 39 validate whether the *Confidentiality* level of *Security* is selected or not, in order to create the corresponding UML classes. Same behavior occurs in lines 40 to 46, to validate the existence of a *Control*. Lines 47 and 48 create the corresponding *Boundary* of the current *BusinessEntity*. Line 49 uses one of the operations created in figure 58 to update the relationships of the previously created *Boundary*, based on the *Controls* created before. Lines 51 to 55 validate the selection of *Authorization* level of *Authenticity* to create the appropriate UML classes. The rest of the transformation can be found in [110], under `co.shift.modeling.m2m/transformations/DomainMetaModel2UML2.atl`.

Figure 47 depicts how to run the transformation. Notice that *models* folder must contain both domain model (*ProjectModel.domainmetamodelm2m*) and QAs configuration (*QAsConfig.xml*) before executing the transformation. Section 5 details how to create these files. *GenModel.uml* file is the one that's generated after executing the transformation. This file can be visualized as a tree using the eclipse default editor. The guide in [111] shows how to construct a UML diagram parting from an `.uml` model, using Papyrus plugin.

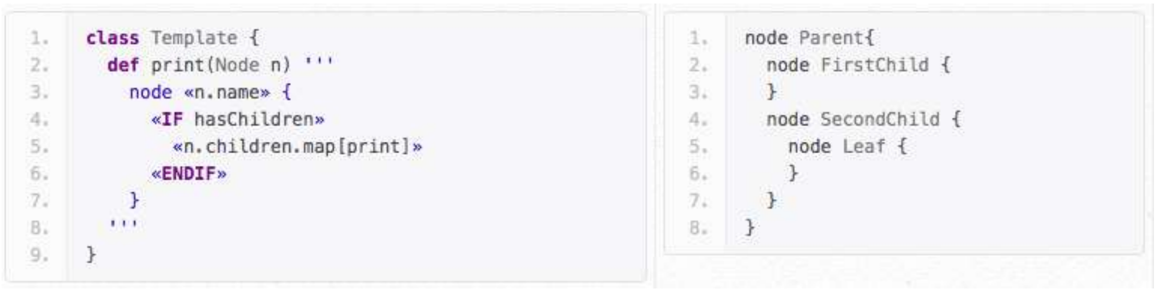


```
name <- 'not';
!Package {
lainMetaModel!Business
ryPackage: UML2!Package = Ocl
.2!Package (
ame <- 'co.shift.' + b.name
: in b.entities) {
isModule.BusinessEntity2Entit
 (b.isQASelected('_r_2_10_12_
   thisModule.BusinessEnti
else
   OclUndefined
```

Figure 47. Executing the ATL transformation

4.3.2. *General Delegation Strategy*

The concrete architecture of a product line member determines the specific design constraints to construct develop the source code. We generate through templates, using Model-to-Text (M2T) transformations. Templates are files that allow for readable string concatenation [112]. Thus, any line written on a template will be transformed into source code. In this work, we use Xtend2 as our template manager because it provides terminals for interpolated expressions that are called guillemets `«expression»,` enabling dynamic construction of source code. These expressions support the use of conditionals, loops and declaring variables. An example of how a template is declared is shown in figure 48. Code on the left is the template declaration, which is identified by triple single quotes ('''). Notice that a class can define more than one template using the word `def`. Code on the right side is the result of executing the template on the left, which uses recursion to create a `Parent` class and its child. More information about using templates with Xtend2 can be found in [112]. It is important to say that each template provides the definition for a particular class, resulting in a one to one relationship.



```
class Template {
  def print(Node n) '''
    node «n.name» {
      «IF hasChildren»
        «n.children.map[print]»
      «ENDIF»
    }
  '''
}
```

```
node Parent{
  node FirstChild {
  }
  node SecondChild {
    node Leaf {
    }
  }
}
```

Figure 48. Xtend2 Template example

We defined templates for our SPL. First, we identified two main groups of templates: kernel and contributed. Former contains the templates that generate common code, that is, code that every product of our product line must have, i.e. *Boundaries* and *Entities* for each *Business Entity* of our DMM. Latter groups the templates that are particular to each quality level, for instance, *Medium* level of *Time Execution* requires a template to generate the *FLR* interface and another one for its implementation. Figure 49 illustrates these groups.

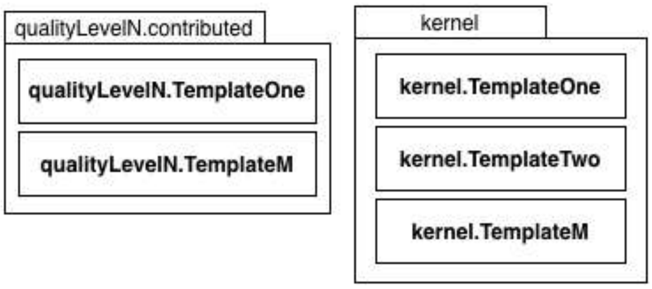
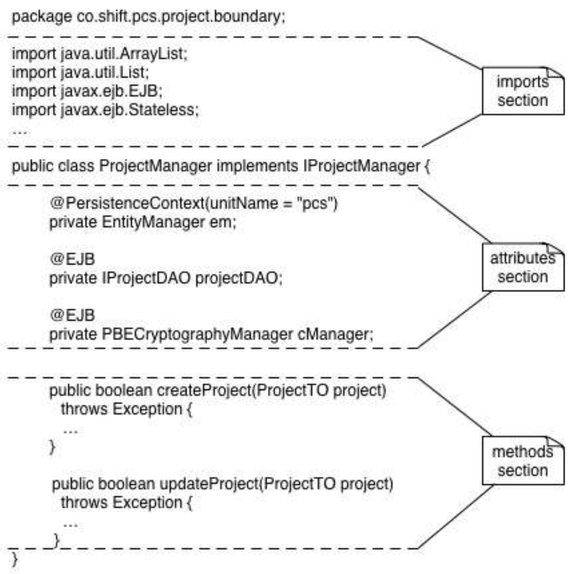


Figure 49. Template groups

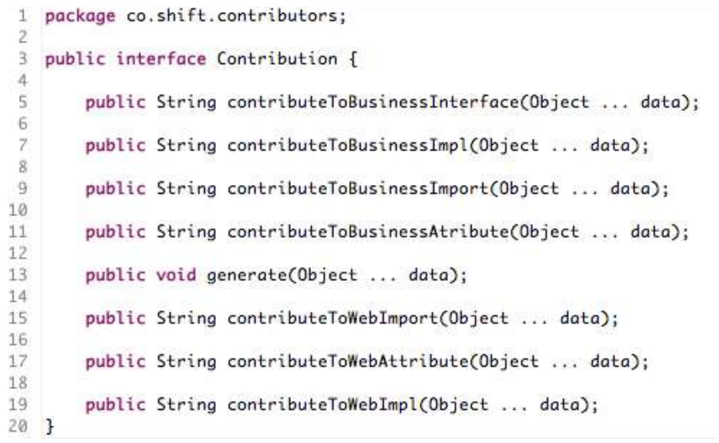
We refer to the code contained in a template as *rules*, given that such code determines how the source code of a class will be organized and generated. Parting from our Reference Architecture (Section 4.2.2) we can conclude that the generated code is modified according to the level of quality selected for each quality attribute of the QAs variability model. For example, the boundary implementation of the Project's domain logic component must define a relationship with *IProjectBasicFLR* interface when a *Medium* level of *Time Execution* is selected (see figure 17). Such relationship changes when *Time Execution* level is different. These modifications represent changes to the generated source code; hence, it is important to detect the specific points of the classes (source code) that might be affected. Figure 50 presents the structure of a Java Class, which in our case is the representation of the source code.



```
package co.shift.pcs.project.boundary;

import java.util.ArrayList;
import java.util.List;
import javax.ejb.EJB;
import javax.ejb.Stateless;
...

public class ProjectManager implements IProjectManager {

    @PersistenceContext(unitName = "pcs")
    private EntityManager em;

    @EJB
    private IProjectDAO projectDAO;

    @EJB
    private PBECryptographyManager cManager;

    public boolean createProject(ProjectTO project)
      throws Exception {
      ...
    }

    public boolean updateProject(ProjectTO project)
      throws Exception {
      ...
    }
}
```

Figure 50. Java class sections

The sections highlighted in the figure 50 are the points that might be modified by the selection of a particular level. For example, relating *ProjectManager* with *IProjectBasicFLR* interface (see figure 17) implies importing the package where the interface is located, declaring the attribute of type *IProjectBasicFLR* to access its services, and modifying the methods that require the services provided by this interface. Notice that these sections are present in every Java class, so modifications of a quality level might affect *DomainLogic* classes as well as *WEB* classes, i.e. *UIController*. To handle the quality modifications (contributions from now on) we provide the interface shown in figure 51.



```
1  package co.shift.contributors;
2
3  public interface Contribution {
4
5      public String contributeToBusinessInterface(Object ... data);
6
7      public String contributeToBusinessImpl(Object ... data);
8
9      public String contributeToBusinessImport(Object ... data);
10
11     public String contributeToBusinessAtribute(Object ... data);
12
13     public void generate(Object ... data);
14
15     public String contributeToWebImport(Object ... data);
16
17     public String contributeToWebAttribute(Object ... data);
18
19     public String contributeToWebImpl(Object ... data);
20 }
```

Figure 51. Contribution interface

The services provided by this interface refer to the sections where a quality level might contribute to the source code. Service in line 5 represents a contribution to the contract declaration of a domain logic component, i.e. *listAllProjects* contract has two parameters when *High Sync* level of *Time Execution* is selected and none when any other level of this

QA is selected. Service in line 7 represents a contribution to the implementation of a contract, i.e. implementing the retrieval of all Projects or delegating the retrieval handling to a *FLR*. Line 9 is a contribution to the imports section of a domain component, i.e. importing the package where *IProjectBasicFLR* is located. Line 11 represents a contribution to the attributes section of a domain component, i.e. declaring an attribute of type *IProjectBasicFLR* to access its services. Line 13 is a particular contribution that executes the templates related to a particular quality level, i.e. executing *PBECryptographyTemplate* when *Confidentiality* level is selected. Lines 15, 17 and 19 are similar to contributions of line 7, 9 and 11, but they modify web classes.

Our approach states that every quality level (variant) of the QAs variability model has to create a class to concretize the contribution interface, in order to determine its contributions to the source code. Notice that the services provided by this interface take a variable array of parameters, enabling each quality level to override them as needed. Each service returns the code fragment (represented as a string) that is going to be concatenated to the corresponding template. Figure 52 illustrates how the QAs variability model is related to the contributions (concrete classes) to source code. Code fragment shown in figure 53 illustrates how *Normal* level of *Time Execution* implements the `contributeToBusinessInterface` of the Contribution interface.

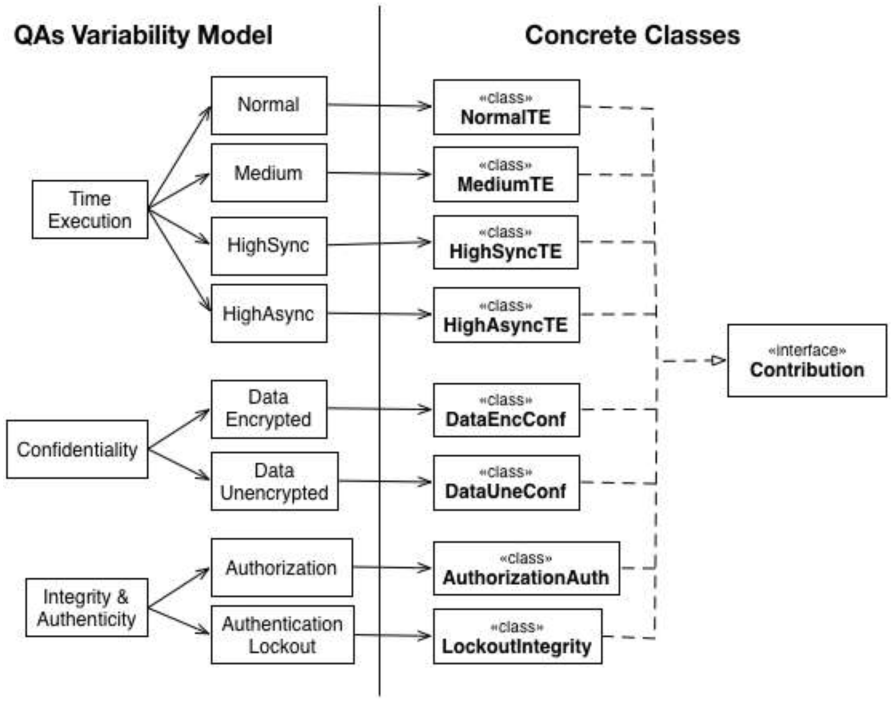
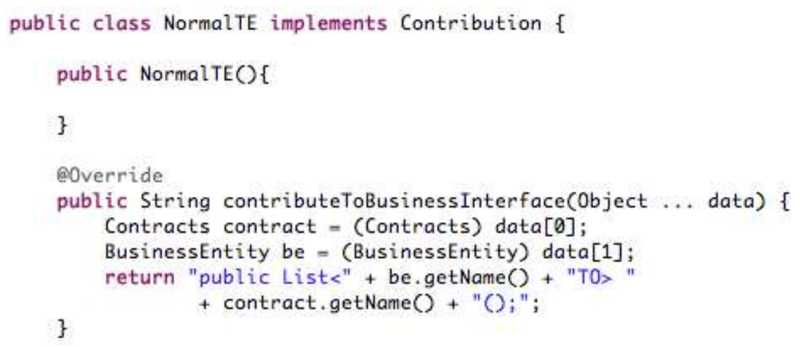


Figure 52. Contribution



```java
public class NormalTE implements Contribution {

    public NormalTE(){

    }

    @Override
    public String contributeToBusinessInterface(Object ... data) {
        Contracts contract = (Contracts) data[0];
        BusinessEntity be = (BusinessEntity) data[1];
        return "public List<" + be.getName() + "TO> "
                + contract.getName() + "();";
    }
```

Figure 53. Concretization example of Contribution interface

Figure 53 shows the concrete contribution of normal time execution variant to the source code. It casts the parameters data to its proper type and uses it to build the service contract of the related *ListAll* operation. It is important to say that not all services of the *Contributor* interface must be overridden, only the ones needed. Parting from *Contributor* interface we detected two types of contributions:

- **Extension.** Consists in contributing a fragment of source code to a Template, which is necessary to the code that's going to be generated. Services in lines 5, 7, 9, 11, 15, 17 and 19 of figure 53 are of this type.
- **Creation.** Consist in executing different templates to generate source code related to a particular quality level, involving templates outside the kernel. Service in line 13 of figure 53 is a creation contribution.

4.3.2.1. CHALLENGES OF CONSIDERING QUALITY LEVELS IN CODE GENERATION

A template implementation consists of the inherent code of the class it represents, e.g. parameterized query of data retrieval to use a *FLR*, and the code contributions provided by the concrete *Contribution* classes of each quality level. In order to include these contributions in the templates declaration, we have to deal with the following challenges:

1. Define a strategy that enables considering quality variations when implementing a template.
2. Dealing with conflicting situations among quality levels.

To deal with the first issue, the straightforward strategy is to use as many conditionals as needed to evaluate the presence of a particular level of quality when implementing a particular section of a template. This strategy has the following drawbacks: Including a conditional to validate the presence of each quality level that needs to be evaluated in a particular section results in several `IF ELSE` sentences, causing high coupling and code difficult to understand and maintain. Thus, our strategy relies on delegations. This requires the templates to know the specific point where a contribution is needed. Once the points are identified, the template delegates the code declaration to the required concrete class, indicating the specific contribution to be added (see figure 50). The concrete class is then responsible for including its contribution (code fragment) to the point specified by the template. Figure 54 illustrates this process.

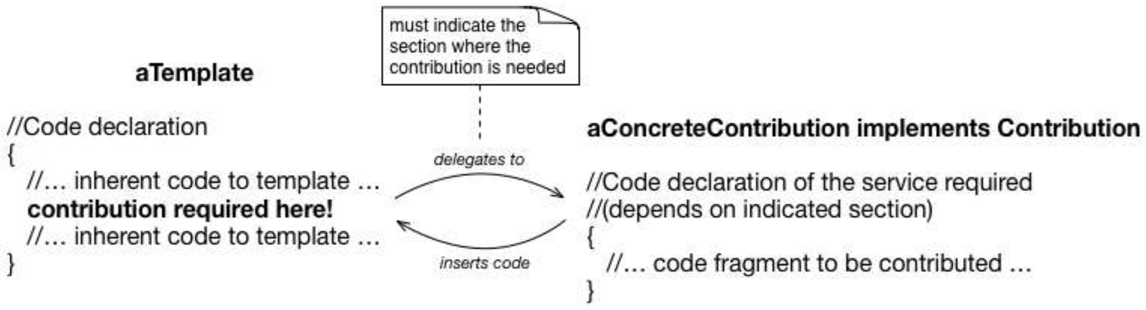

aTemplate
//Code declaration
{
//... inherent code to template ...
contribution required here!
//... inherent code to template ...
}
must indicate the section where the contribution is needed
delegates to
inserts code
aConcreteContribution implements Contribution
//Code declaration of the service required
//(depends on indicated section)
{
//... code fragment to be contributed ...
}

Figure 54. Delegation strategy

Delegation strategy shown in figure 54 suffers from several conflicting situations among quality levels (second challenge), given that configuring a product with several quality levels might cause the following tradeoffs:

1. The selection of two quality levels might cause and exclusion relationship (exposed in section 4.1.2.1.1), which denotes that two QAs cannot coexist, because the selection of one entirely inhibits or contradicts the other. This particular tradeoff is not considered in our implementation due to its absence in our QAs model.
2. Selecting two quality levels might impact the same section of a template. For example, both *Time Execution* and *Confidentiality* levels modify the implementation of a *ListAll* operation, given that all data retrieved must be unencrypted before displaying it to the final user. This implies that the contribution required on a template is composed of several contributions provided by different concrete *Contributions* (classes implementing *Contribution* interface). Thus, it is important to develop a strategy that enables these two levels to provide their contribution in a synergic manner. Figure 55 illustrates this conflict.

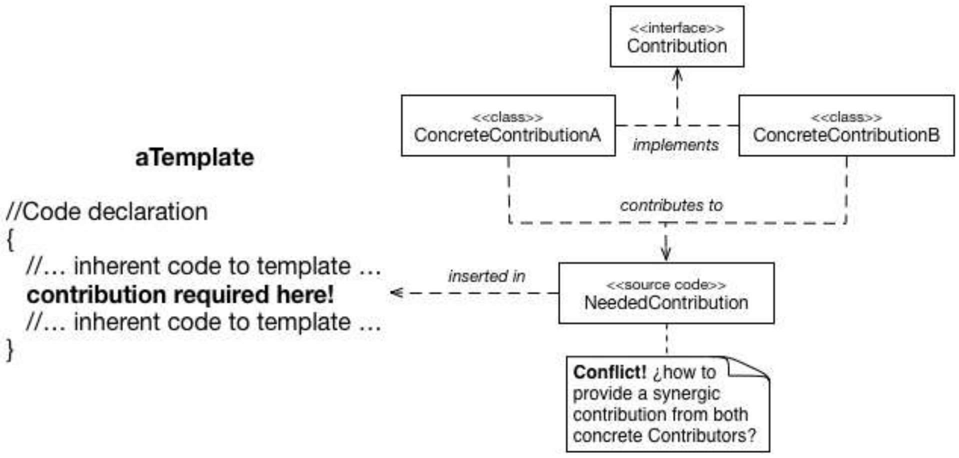
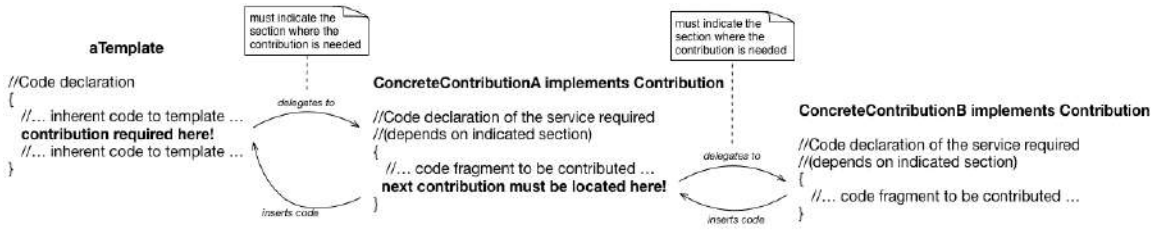

<<interface>> Contribution
<<class>> ConcreteContributionA
<<class>> ConcreteContributionB
implements
contributes to
aTemplate
//Code declaration
{
//... inherent code to template ...
contribution required here!
//... inherent code to template ...
}
inserted in
<<source code>> NeededContribution
Conflict! ¿how to provide a synergic contribution from both concrete Contributors?

Figure 55. Conflict resulted from two or more contributions to a same template section

Dealing with the conflict shown in figure 55 can be solved in two ways. The first one considers ordering the contributions, i.e. a contribution must be performed prior the other one. This order depends on the context of the contributions. For example, both *Time Execution* and *Confidentiality* levels modify the implementation of a *ListAll* operation.

Hence, the contribution related to time execution (retrieval of data) must be executed prior the confidentiality contribution (data unencrypting), given that retrieved data is needed to be unencrypted. Once the order is set, the delegation strategy shown in figure 54 must be extended, in order to be use by templates and concrete *Contributions* (classes). Thus, the template that requires the contribution delegates the code declaration to the concrete *Contribution* (class) whose contribution must be performed first. Then, this concrete class delegates the code declaration to the following concrete *Contribution* (class), according to the stated order. This pattern may be replied as many times as needed. Figure 56 depicts this solution.



aTemplate
//Code declaration
{
//... inherent code to template ...
contribution required here!
//... inherent code to template ...
}
must indicate the section where the contribution is needed
delegates to
inserts code
ConcreteContributionA implements Contribution
//Code declaration of the service required
//(depends on indicated section)
{
//... code fragment to be contributed ...
next contribution must be located here!
}
must indicate the section where the contribution is needed
delegates to
inserts code
ConcreteContributionB implements Contribution
//Code declaration of the service required
//(depends on indicated section)
{
//... code fragment to be contributed ...
}

Figure 56. Delegation strategy based on contributions ordering

Second solution is needed when ordering the contributions of concrete *Contribution* (classes) doesn't result in a coherent code fragment, but the resulting product line member must contain both quality levels. This solution demands creating a new concrete *Contribution* class. This new class must provide an implementation that considers both quality levels, in order to relate them in a synergic manner. This decision requires a design stage of the interactions between related quality levels prior its implementation, plus a validation process executed within the generation that triggers the use of the newly *Contribution* when these quality levels are present.

4.3.3. Concrete Generation Strategy

Following sections describe our approach to concretize this delegation strategy. We created three groups of kernel templates, one for the web layer, one for the domain logic and another one for the database. Based on the Reference Architecture in section 4.2.2, there are several contributed templates, depending on the quality attribute and level. Figure 57 depicts the group of templates we created for generating our SPL product members. Table 2 describes each one of the templates.

4.3.3.1. CONSIDERING QUALITY CONTRIBUTIONS TO CODE GENERATION

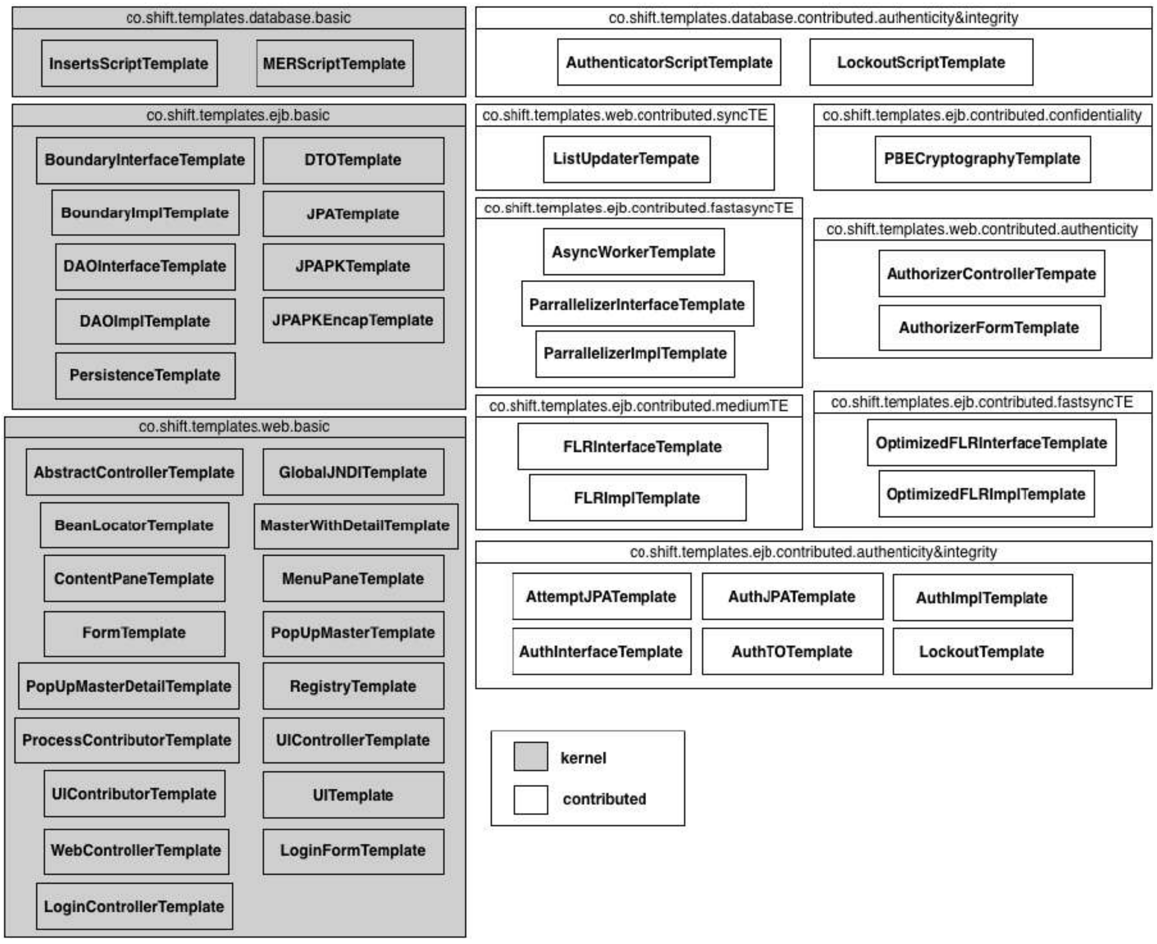


Figure 57. Templates involved in our generation process and their grouping

co.shift.tempates.database.basic	
InsertsScriptTemplate	Declares all the default inserts to the database that are needed when a product is configured, that is, a default user. If the Authorization level is selected, it creates an admin role and the entire set of services based on the selected functionalities of a product.
MERScriptTemplate	Creates the database script to persist the information related to the derived product. The declared tables and relationships entirely depend on the functionalities configured in the domain model of a product.
co.shift.templates.ejb.basic	
BoundaryInterfaceTemplate	Specifies the rules for creating the boundary interface of each *BusinessEntity*. Services names, params and return types depend on the configuration given for a particular domain model.
BoundaryImplInterfaceTemplate	Provides the rules for implementing the boundary interface of each BusinessEntity. It also evaluates selection of particular quality levels, in order to properly adapt to the designs provided in our RA.

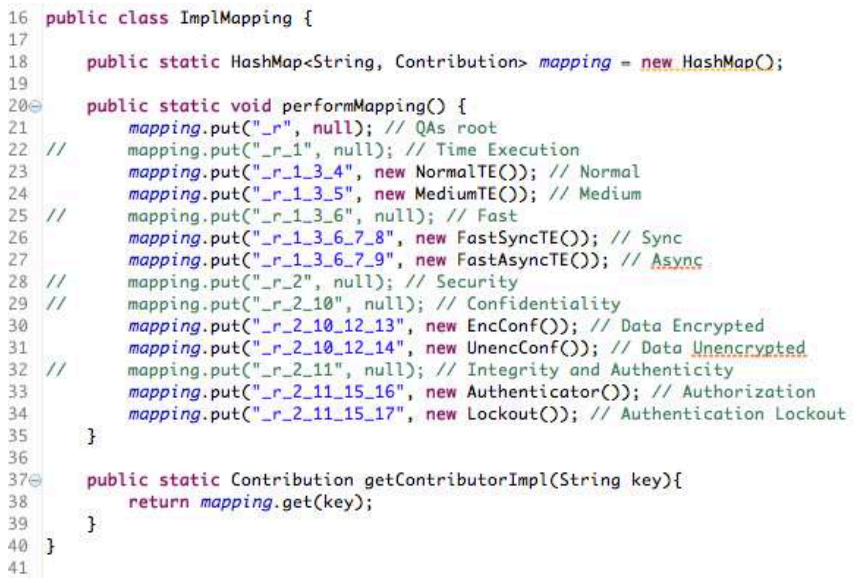
DAOInterfaceTemplate	Specifies the rules for implementing *DAO* interfaces in case they are needed (See section 4.2.2.2.2). Services names, params and return types depend on the configuration given for a particular domain model.
DAOImplInterfaceTemplate	Provides the rules to handle the implementation of the declared *DAO* interfaces. Evaluates the selection of Confidentiality level when implementing each service.
DTOTemplate	Provides the rules to implement the corresponding *DTO* for each *BusinessEntity* configured in a particular domain model.
JPATemplate	Provides the rules to implement the corresponding *Entity* for each *BusinessEntity* configured in a particular domain model.
JPAPKTemplate	Needed to handle addition and deletion of detail *BusinessEntities* to a master, it declares the rules to manage relationships among *JPA* Entities without breaking *GRASP* patterns (high cohesion and low coupling).
JPAPKEncapTemplate	Contains the rules to encapsulate the relationship of two Entities into a *JPA* Entity, so it can be used as a primary key to perform addition and deletion of detail *BusinessEntities*.
PersistenceTemplate	Provides the declaration of all the involved *JPA* entities in a particular product, so they can be manipulated by the created *EJB* components.
co.shift.templates.web.basic	
ProcessContributorTemplate	Specifies the *ProcessContributor* interface, which provides a service to be used when the implementing Controller contributes to a particular process of another Controller, for instance, *ConfidentialityController* contributes to *ProjectController* by encrypting the data when a project is created.
UIContributorTemplate	Specifies the *UIContributor* interface, which provides a service to be used when the implementing Controller contributes to the *View* managed by another Controller, for instance, *ProjectController* contributes to *MainController* by adding a menu item to access project management.
AbstractControllerTemplate	Provides the default attributes and behavior for the *UI Controllers*. Every Controller must have a collection of *UI Contributors* and another one of *Process Contributors*.
BeanLocatorTemplate	Specifies the implementation of the BeanLocator pattern [39], to deal with *EJBs* location.
GlobalJNDITemplate	Provides the implementation of *GlobalJNDI* class, which uses a *Builder* pattern [34] to make easier the construction of location strings used by the *BeanLocator*.
ContentPaneTemplate	Contains the implementation of the main view panel to display the Views of a product.
FormTemplate	Provides the rules for implementing the *View* of each *BusinessEntity* configured in a particular domain model. **This template is only used by master *BusinessEntities.***
PopUpMasterDetailTemplate	Provides the rules for implementing a *View* to relate a detail *BusinessEntity* to a master *BusinessEntity*.
WebControllerTemplate	Provides the rules for implementing the *UIController* of each *BusinessEntity* configured in a particular domain model. It

	must consider selection of time execution levels to properly implement *ListAll* operations.
LoginControllerTemplate	Provides the implementation of the Controller that handles login events of a product.
MenuPaneTemplate	Contains the implementation of the navigation menu that is used by the product to browse functionalities.
PopUpMasterTemplate	Provides the rules for implementing the *View* of the **detail** *BusinessEntities* configured in a particular domain model.
RegistryTemplate	Provides the implementation of the *Registry* class, which is used to store the session objects of a particular product when a user is logged in, i.e. user name.
UIControllerTemplate	Provides the rules for implementing the main controller of a particular product. This controller configures the contributors (UI and/or Process) of each Controller of the product, initializes the navigation menu and displays the login *View*.
UITemplate	Contains the implementation of the entry point of the application. This class initializes the product's content pane and menu pane and invokes the *UIController*.
LoginFormTemplate	Provides the implementation of the View that displays the login form of a product.
co.shift.templates.database.contributed.authenticity&integrity	
AuthenticatorScriptTemplate	Provides the implementation of the database tables needed when the *Authorization* level of *Authenticity* is selected (Roles, Services, Rights).
LockoutScriptTemplate	Provides the implementation of the database table "Attempts" when the *Lockout* level of *Integrity* is selected.
co.shift.templates.web.contributed.syncTE	
ListUpdaterTemplate	Contains the implementation of the *ListUpdater* required when the *High Sync* level of *Time Execution* is selected. See figure 19.
co.shift.templates.ejb.contributed.confidentiality	
PBECryptographyTemplate	Provides the implementation of the *PBECryptographyManager*, which is needed to handle encryption/decryption of data when *Confidentiality* level of *Security* is selected.
co.shift.templates.ejb.contributed.fastayncTE	
AsynWorkerTemplate	Provides the rules for implementing the *AsynWorker* class, which defines the task that is going to be executed in parallel when the *High Async* level of *Time Execution* is selected.
ParallelizerInterfaceTemplate	Specifies the rules for creating the *Parallelizer* interface of each *BusinessEntity* when the *High Async* level of *Time Execution* is selected.
ParallelizerImplTemplate	Provides the rules for implementing the *Parallelizer* interface of each *BusinessEntity*.
co.shift.templates.web.contributed.authenticity	
AuthorizerControllerTemplate	Provides the rules for implementing the *Authorizer* controller of a particular product. This controller manages the tables created when implementing the *RBAC* pattern, including relating users with roles.
AuthorizerFormTemplate	Provides the rules for implementing the *View* that enables

	managing roles and services and relating them to users.
co.shift.templates.ejb.contributed.mediumTE	
FLRInterfaceTemplate	Specifies the rules for creating the *FLR* interface of each *BusinessEntity* when the *Medium* level of *Time Execution* is selected.
FLRImplInterface	Provides the rules for implementing the *FLR* interface of each *BusinessEntity*.
co.shift.templates.ejb.contributed.fastsyncTE	
OptimizedFLRInterfaceTemplate	Specifies the rules for creating the *OptimizedFLR* interface of each *BusinessEntity* when the *High Sync* level of *Time Execution* is selected.
OptimizedFLRImplInterface	Provides the rules for implementing the *OptimizedFLR* interface of each *BusinessEntity*.
co.shift.templates.ejb.contributed.authenticity&integrity	
AttemptJPATemplate	Provides the mapping of the database table "Attempts" to a *JPA Entity* when the *Lockout* level of *Integrity* is selected.
AuthJPATemplate	This class contains all the templates needed to map the database tables involved in the *RBAC* pattern to *JPA Entities*.
AuthTOTemplate	This class contains all the templates needed to map the database tables involved in the *RBAC* pattern to *DTOs*.
LockoutTemplate	Provides the rules needed to implement the *AccountManager*, which is needed to control the number of login attempts performed by each user account when the *Lockout* level of *Integrity* is selected.
AuthInterfaceTemplate	Specifies the rules for creating the *IAuthorizationManager* that provides the operations for managing Roles, Rights and Services when the *Authorization* level of *Authenticity* is selected.
AuthImplTemplate	Provides the rules for implementing the *IAuthorizationManager* interface.

Table 2. Templates Description

Figure 58 show the static class *ImplMapping,* which assigns a unique id for each quality level and relates each of them with their corresponding *Contributor* implementation (`performMapping` method). For example, the id `r_1_3_4` identifies the *Normal* level of *Time Execution*, and its *Contributor* implementation is the *NormalTE* class. *ImplMapping* class also provides a static method that enables obtaining the *Contributor* of a particular quality level given its id (lines 37 to 39).

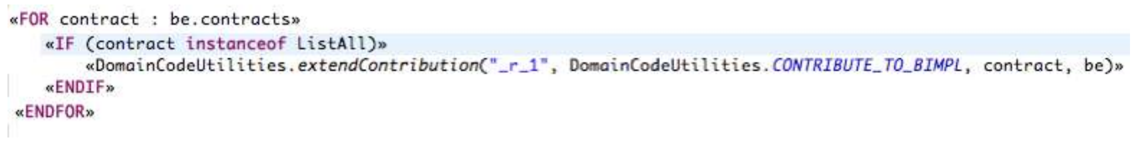


```
public class ImplMapping {

    public static HashMap<String, Contribution> mapping = new HashMap();

    public static void performMapping() {
        mapping.put("_r", null); // QAs root
//      mapping.put("_r_1", null); // Time Execution
        mapping.put("_r_1_3_4", new NormalTE()); // Normal
        mapping.put("_r_1_3_5", new MediumTE()); // Medium
//      mapping.put("_r_1_3_6", null); // Fast
        mapping.put("_r_1_3_6_7_8", new FastSyncTE()); // Sync
        mapping.put("_r_1_3_6_7_9", new FastAsyncTE()); // Async
//      mapping.put("_r_2", null); // Security
//      mapping.put("_r_2_10", null); // Confidentiality
        mapping.put("_r_2_10_12_13", new EncConf()); // Data Encrypted
        mapping.put("_r_2_10_12_14", new UnencConf()); // Data Unencrypted
//      mapping.put("_r_2_11", null); // Integrity and Authenticity
        mapping.put("_r_2_11_15_16", new Authenticator()); // Authorization
        mapping.put("_r_2_11_15_17", new Lockout()); // Authentication Lockout
    }

    public static Contribution getContributorImpl(String key){
        return mapping.get(key);
    }
}
```

Figure 58. Ids for each quality level concretization

Once we have identified each quality level, we need to know which of them are selected in a particular configuration of the QAs variability model. To manage configurations of our quality model we use a tool that enables creating and configuring features models online (S.P.L.O.T.). The use of this tool is explained in section 5.2, but we are going to anticipate that an XML containing the quality configuration is generated. Hence, we developed a parser that takes this XML as input and populates a *HashMap* with the *Contributors* of the selected quality levels. This is possible because the ids that we used in figure 58 are the same as the ones contained in the generated XML, thus, every time a quality level is selected, its id can be used to get the related *Contributor* using the static method `getContributorImpl` provided by *ImplMapping*.

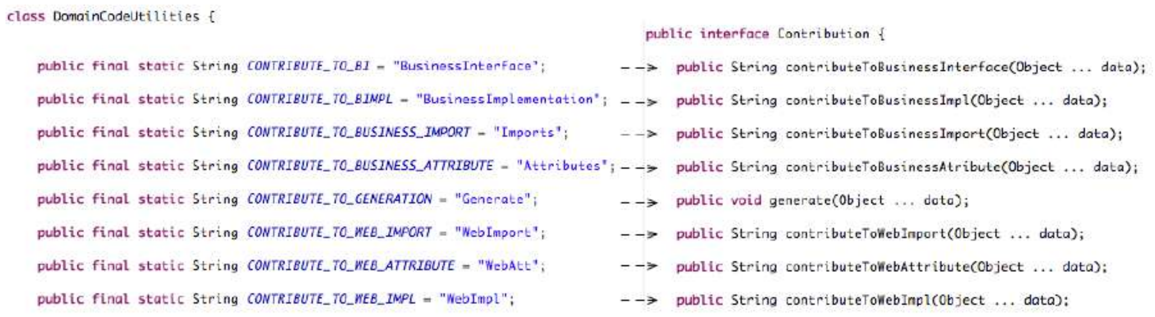
Next step after identifying the selected *Contributors* of a particular quality configuration is to develop a mechanism that allows accessing these objects (*Contributors*) to use their overridden functionalities (implementation of *Contributor* contacts) on demand. We developed an API (*DomainCodeUtilities*) that provides a particular function (among others) named *extendContribution* that takes the quality level id, the identifier of the section where the contribution is needed and additional data (variable parameters) to generate a contribution to the invoking template. Such function has a static modifier, thus, no instantiation is required to use it. Code fragment shown in figure 59 illustrates how the *BoundaryImplTemplate* uses this function to include the contribution generated by a *Time Execution* level.



```
«FOR contract : be.contracts»
    «IF (contract instanceof ListAll)»
        «DomainCodeUtilities.extendContribution("_r_1", DomainCodeUtilities.CONTRIBUTE_TO_BIMPL, contract, be)»
    «ENDIF»
«ENDFOR»
```

Figure 59. Example of using *extendContribution* function

Code in figure 59 illustrates an extension contribution with the following particularities: The first parameter contains the id of the *Time Execution* attribute (see figure 59) and not the id of a particular level (variant). When this method finds this expression, it will execute the contributions from all quality levels of the given QA. In this case, only one contribution will be executed (the selected level in the QA configuration) because *Time Execution* levels are exclusive (see figure 7). The second parameter identifies the section where the contribution is needed; in this case, the contribution will be included in the methods implementation section. There is one id for each of the eight contribution sections (see figure 60). The remaining parameters are information that the particular contribution needs to be executed. In this case, the contribution needs the current *contract* and *BusinessEntity* to work.



```
class DomainCodeUtilities {
                                                                               public interface Contribution {

   public final static String CONTRIBUTE_TO_BI = "BusinessInterface";           -->  public String contributeToBusinessInterface(Object ... data);

   public final static String CONTRIBUTE_TO_BIMPL = "BusinessImplementation";   -->  public String contributeToBusinessImpl(Object ... data);

   public final static String CONTRIBUTE_TO_BUSINESS_IMPORT = "Imports";        -->  public String contributeToBusinessImport(Object ... data);

   public final static String CONTRIBUTE_TO_BUSINESS_ATTRIBUTE = "Attributes";  -->  public String contributeToBusinessAtribute(Object ... data);

   public final static String CONTRIBUTE_TO_GENERATION = "Generate";            -->  public void generate(Object ... data);

   public final static String CONTRIBUTE_TO_WEB_IMPORT = "WebImport";           -->  public String contributeToWebImport(Object ... data);

   public final static String CONTRIBUTE_TO_WEB_ATTRIBUTE = "WebAtt";           -->  public String contributeToWebAttribute(Object ... data);

   public final static String CONTRIBUTE_TO_WEB_IMPL = "WebImpl";               -->  public String contributeToWebImpl(Object ... data);
```

Figure 60. Identifications for each contribution section

To illustrate how this strategy overcomes the challenges presented in section 4.3.2.1, lets see how we managed the contributions generated by a *Normal Time Execution* level and a *Data Encrypted* level of *Confidentiality* when a *ListAll* contract appears. There are two possible outcomes when two quality levels contribute to the same section of a template: the first one is the possibility of ordering the execution of the contributions involved, the second one is to create a new *Contributor* mixing both quality levels due to their inability to coexist. Considering that the *Normal* level provides the implementation of the *ListAll* contract and that *Data Encrypted* level provides decryption attributes retrieved of *String* type, we can establish an order where decryption occurs within the implementation of data retrieval. To clarify this, figure 61 shows both contributions to Project's domain component, individually.

If *Confidentiality* contribution is located before *Time Execution* contribution, an error will occur because the *projectTO* object used by the former has not been declared. Conversely, if these contributions are inverted (*Time Execution* first and *Confidentiality* second) an error will still be present cause the code generated by the latter is not inside a method declaration, so it cannot be compiled. To make this work, we decided to include *Confidentiality* contribution within *Time Execution* contribution, particularly between lines 10 and 11 of figure 61. This way both codes are properly connected and can be compiled as one. Figure 62 shows the mixed contributions.

Data Encrypted - Confidentiality Contribution

```
char eNameChars = cManager.doFinal(
  PBECryptographyManager.DECRYPT, projectTO.getName());
String eName = new String(eNameChars);
projectTO.setName(eName);

char eDescriptionChars = cManager.doFinal(
  PBECryptographyManager.DECRYPT, projectTO.getDescription());
String eDescription = new String(eDescriptionChars);
projectTO.setDescription(eDescription);
```

Normal - Time Execution Contribution

```
public List<ProjectTO> getAllProjects() {
        List<ProjectTO> projects = new ArrayList<>();
        TypedQuery<Project> query = em.createNamedQuery(
                "project.getAllProjects", Project.class);
        List<Project> foundProjects = query.getResultList();
        for (Project project : foundProjects) {
            ProjectTO projectTO = new ProjectTO();
            projectTO.setId(project.getId());
            projectTO.setDescription(project.getDescription());
            projectTO.setName(project.getName());
            projectTO.setStartDate(project.getStartDate());
            projects.add(projectTO);
        }
        return projects;
}
```

Figure 61. Contributions to Project's domain component (Data Encrypted and Normal Time Execution)

Mixed Contributions

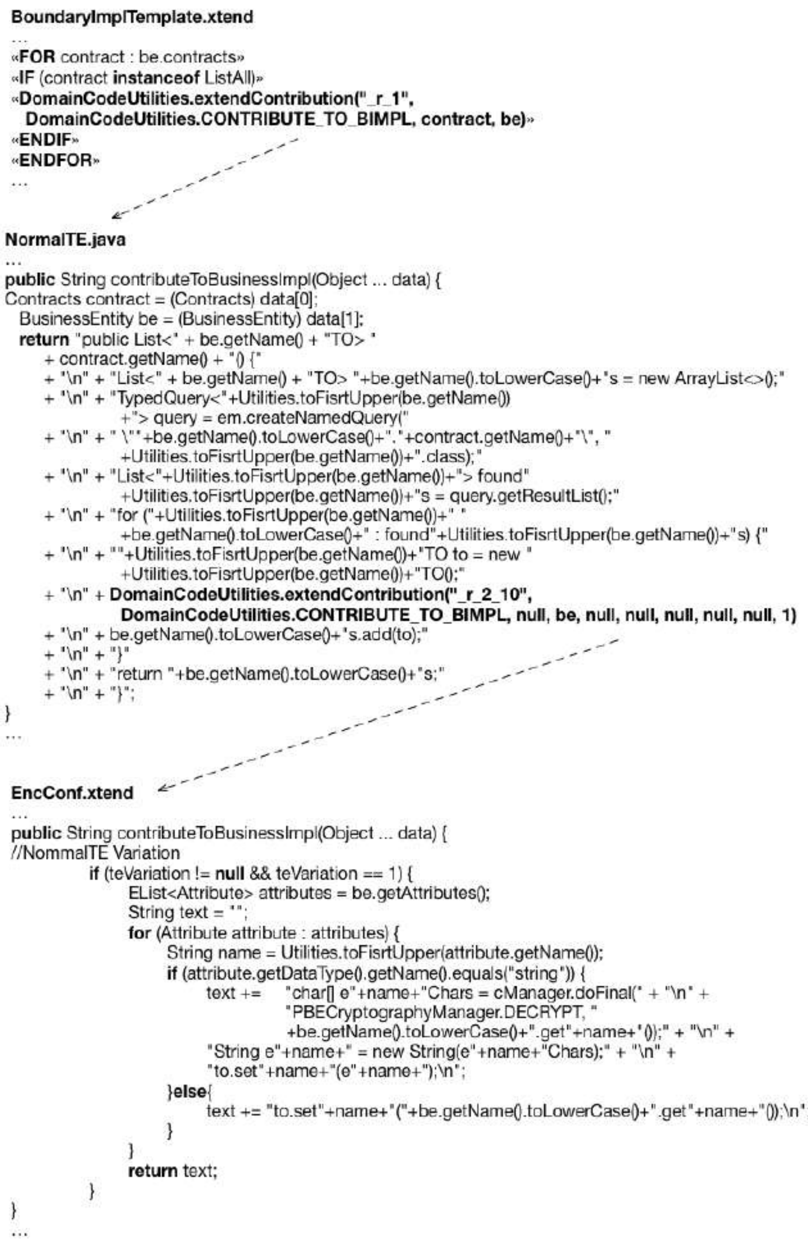
```
public List<ProjectTO> getAllProjects() {
        List<ProjectTO> projects = new ArrayList<>();
        TypedQuery<Project> query = em.createNamedQuery(
                "project.getAllProjects", Project.class);
        List<Project> foundProjects = query.getResultList();
        for (Project project : foundProjects) {
            ProjectTO projectTO = new ProjectTO();
            projectTO.setId(project.getId());
            projectTO.setDescription(project.getDescription());
            projectTO.setName(project.getName());
            projectTO.setStartDate(project.getStartDate());

            char eNameChars = cManager.doFinal(
            PBECryptographyManager.DECRYPT, projectTO.getName());
            String eName = new String(eNameChars);
            projectTO.setName(eName);

            char eDescriptionChars = cManager.doFinal(
            PBECryptographyManager.DECRYPT, projectTO.getDescription());
            String eDescription = new String(eDescriptionChars);
            projectTO.setDescription(eDescription);

            projects.add(projectTO);
        }
        return projects;
}
```

Figure 62. Mixed Contributions to Project's domain component (Data Encrypted and Normal Time Execution)

Solution shown in figure 62 implies using the *extendContribution* method in a kernel template, as well as in a particular *Contribution*. Figure 63 illustrates the implementation required in templates and concrete *Contributions* classes to obtain the result shown in figure 62. Notice how the *extendContribution* method is called inside the `for` declaration of *NormalTE*. This strategy avoids declaring several conditionals to evaluate selected quality levels (for example declaring one `IF` to check the existence of every level of *Time Execution* and then include a nested `IF` inside each of them to validate the selected *Confidentiality* level). Hence, it prevents high coupling and simplifies the code needed to consider quality variations, which promotes its maintainability and scalability.

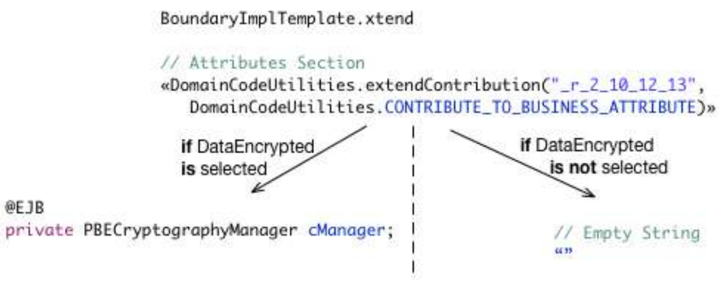
Using the *extendContribution* method with and id of a quality level that has not been selected produces no changes on the invoking template. Thus, this property makes considering quality variation a little bit easier, for example, *Data Encrypted* level of *Confidentiality* requires every *Boundary* to always have a relationship with the

PBECryptographyManager. This is translated into a contribution to the *BoundaryImplTemplate* in its attributes section, where the relationships are declared. So, we can easily use the following code to handle this situation, knowing that if *Data Encrypted* is selected the proper contribution will occur, and if its omitted from the QAs configuration, no code related to the *PBECryptographyManager* will appear in the resulting source code. Figure 64 illustrates this situation.



```
BoundaryImplTemplate.xtend
...
«FOR contract : be.contracts»
«IF (contract instanceof ListAll)»
«DomainCodeUtilities.extendContribution("_r_1",
  DomainCodeUtilities.CONTRIBUTE_TO_BIMPL, contract, be)»
«ENDIF»
«ENDFOR»
...

NormalTE.java
...
public String contributeToBusinessImpl(Object ... data) {
Contracts contract = (Contracts) data[0];
  BusinessEntity be = (BusinessEntity) data[1];
  return "public List<" + be.getName() + "TO> "
      + contract.getName() + "() {"
      + "\n" + "List<" + be.getName() + "TO> "+be.getName().toLowerCase()+"s = new ArrayList<>();"
      + "\n" + "TypedQuery<"+Utilities.toFisrtUpper(be.getName())
                +"> query = em.createNamedQuery("
      + "\n" + " \""+be.getName().toLowerCase()+"."+contract.getName()+"\", "
                +Utilities.toFisrtUpper(be.getName())+".class);"
      + "\n" + "List<"+Utilities.toFisrtUpper(be.getName())+"> found"
                +Utilities.toFisrtUpper(be.getName())+"s = query.getResultList();"
      + "\n" + "for ("+Utilities.toFisrtUpper(be.getName())+" "
                +be.getName().toLowerCase()+" : found"+Utilities.toFisrtUpper(be.getName())+"s) {"
      + "\n" + ""+Utilities.toFisrtUpper(be.getName())+"TO to = new "
                +Utilities.toFisrtUpper(be.getName())+"TO();"
      + "\n" + DomainCodeUtilities.extendContribution("_r_2_10",
                DomainCodeUtilities.CONTRIBUTE_TO_BIMPL, null, be, null, null, null, null, null, 1)
      + "\n" + be.getName().toLowerCase()+"s.add(to);"
      + "\n" + "}"
      + "\n" + "return "+be.getName().toLowerCase()+"s;"
      + "\n" + "}";
}
...

EncConf.xtend
...
public String contributeToBusinessImpl(Object ... data) {
//NommalTE Variation
        if (teVariation != null && teVariation == 1) {
            EList<Attribute> attributes = be.getAttributes();
            String text = "";
            for (Attribute attribute : attributes) {
                String name = Utilities.toFisrtUpper(attribute.getName());
                if (attribute.getDataType().getName().equals("string")) {
                    text +=     "char[] e"+name+"Chars = cManager.doFinal(" + "\n" +
                                "PBECryptographyManager.DECRYPT, "
                                +be.getName().toLowerCase()+".get"+name+"());" + "\n" +
                    "String e"+name+" = new String(e"+name+"Chars);" + "\n" +
                    "to.set"+name+"(e"+name+");\n";
                }else{
                    text += "to.set"+name+"("+be.getName().toLowerCase()+".get"+name+"());\n";
                }
            }
            return text;
        }
}
...
```

Figure 63. Example of delegating contributions among Templates and Contributors



```
BoundaryImplTemplate.xtend

// Attributes Section
«DomainCodeUtilities.extendContribution("_r_2_10_12_13",
   DomainCodeUtilities.CONTRIBUTE_TO_BUSINESS_ATTRIBUTE)»
```

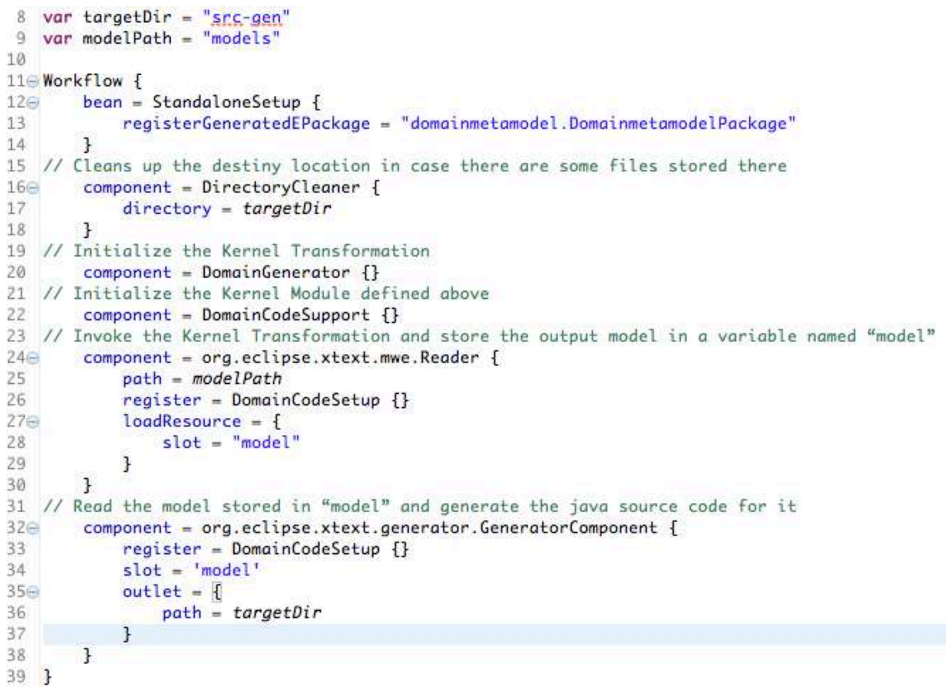
```
@EJB
private PBECryptographyManager cManager;
```

```
// Empty String
“”
```

Figure 64. Resulting contributions when an attribute is selected and when it isn’t

4.3.3.2. CODE GENERATION ENGINE

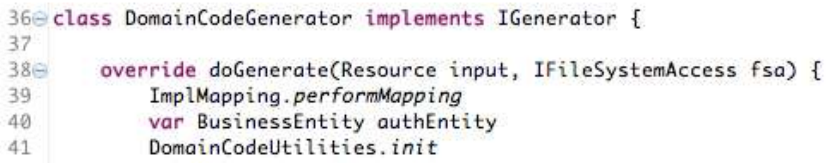
We developed an engine that takes as input the templates of section 4.3.3.1, a domain model and a quality configuration to produce the source code of a particular product line member. To do so, we decided to use the Modeling Workflow Engine 2 (MWE2), which is a declarative, externally configurable generator engine. This technology enable us to declare a workflow that takes a domain model (according to our DMM) as input, to take it through a series of transformations (M2T in this case) provided by our templates (Xtend2 classes) to produce source code. Such workflow works as director, where decisions about which templates must be executed are taken, based on the information provided in the domain model. We have developed a tutorial showing how to create a MWE2 workflow to perform model-to-model (M2M) transformations, as well as model-to-text (M2T) transformations. Thus, in this work we only provide the workflow declaration (see figure 65) and some details on how the transformations are performed.



```
 8 var targetDir = "src-gen"
 9 var modelPath = "models"
10
11 Workflow {
12     bean = StandaloneSetup {
13         registerGeneratedEPackage = "domainmetamodel.DomainmetamodelPackage"
14     }
15 // Cleans up the destiny location in case there are some files stored there
16     component = DirectoryCleaner {
17         directory = targetDir
18     }
19 // Initialize the Kernel Transformation
20     component = DomainGenerator {}
21 // Initialize the Kernel Module defined above
22     component = DomainCodeSupport {}
23 // Invoke the Kernel Transformation and store the output model in a variable named “model”
24     component = org.eclipse.xtext.mwe.Reader {
25         path = modelPath
26         register = DomainCodeSetup {}
27         loadResource = {
28             slot = "model"
29         }
30     }
31 // Read the model stored in “model” and generate the java source code for it
32     component = org.eclipse.xtext.generator.GeneratorComponent {
33         register = DomainCodeSetup {}
34         slot = 'model'
35         outlet = {
36             path = targetDir
37         }
38     }
39 }
```

Figure 65. Generation Workflow

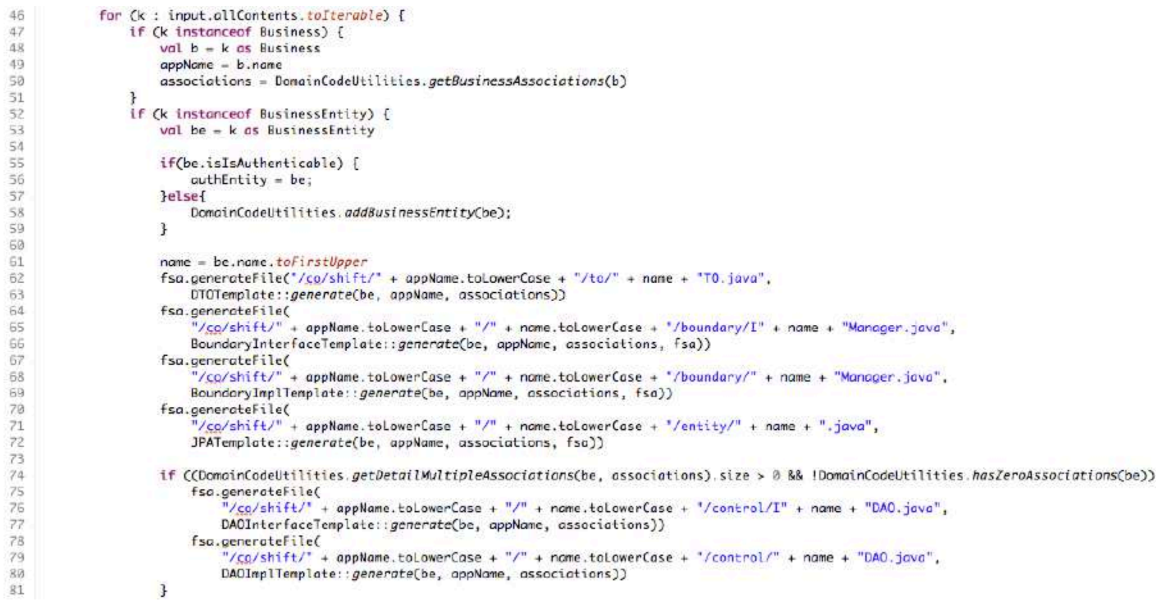
Line 8 in figure 65 contains the path where the generated source code will be stored. Line 9 provides the path where the models (domain and QA) are located. Lines 16 to 18 clean the target path before generating the code. Lines 20 and 22 initialize some auxiliary classes that are needed for executing the workflow. Lines 24 to 30 declare a *Reader* object that seeks for a domain model in the specified path (*modelPath*) and uses the *DomaCodeSetup* class to read its content and load it into memory, using the *model* id. Lines 32 to 38 use the *model* id to read the content from memory and to perform the necessary transformations to generate the source code, which is going to be stored in the *targetDir* path. The *DomainCodeSetup* class used in line 33 internally invokes the *DomainCodeGenerator* class, which is the one that overrides the behavior of the code generation, acting as the main thread that organizes the invocation of the *Templates*. This class has access to the domain model that was loaded into memory, and to the services provided by an object of type *IFileSystemAccess*, which enables executing the templates to generate the source code. Code fragments shown in figures 66 and 67 depict the implementation of the *DomainCodeGenerator*



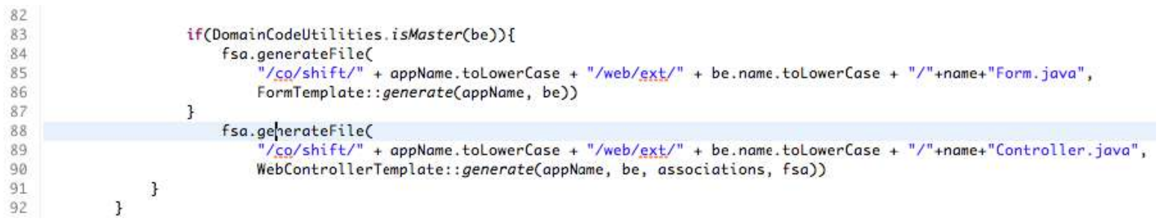
```
36 class DomainCodeGenerator implements IGenerator {
37
38     override doGenerate(Resource input, IFileSystemAccess fsa) {
39         ImplMapping.performMapping
40         var BusinessEntity authEntity
41         DomainCodeUtilities.init
```

Figure 66. Using our API to enable quality handling

First thing to do is perform the mapping discussed in figure 58. Line 39 in figure 66 accomplishes this task. Line 41 uses our provided API to initialize the map that contains the selected *Contributions* provided in the configuration of QAs. Notice that this line invokes the methods that are in charge of parsing the XML that contains the configuration of quality levels (generated by S.P.L.O.T), and loading this info into memory, so it can be accessed to perform proper rules using the strategy explained in previous the section. The *Resource* parameter in line 38 represents the domain model that was loaded into memory by the workflow.



```
        for (k : input.allContents.toIterable) {
            if (k instanceof Business) {
                val b = k as Business
                appName = b.name
                associations = DomainCodeUtilities.getBusinessAssociations(b)
            }
            if (k instanceof BusinessEntity) {
                val be = k as BusinessEntity

                if(be.isIsAuthenticable) {
                    authEntity = be;
                }else{
                    DomainCodeUtilities.addBusinessEntity(be);
                }

                name = be.name.toFirstUpper
                fsa.generateFile("/co/shift/" + appName.toLowerCase + "/to/" + name + "TO.java",
                    DTOTemplate::generate(be, appName, associations))
                fsa.generateFile(
                    "/co/shift/" + appName.toLowerCase + "/" + name.toLowerCase + "/boundary/I" + name + "Manager.java",
                    BoundaryInterfaceTemplate::generate(be, appName, associations, fsa))
                fsa.generateFile(
                    "/co/shift/" + appName.toLowerCase + "/" + name.toLowerCase + "/boundary/" + name + "Manager.java",
                    BoundaryImplTemplate::generate(be, appName, associations, fsa))
                fsa.generateFile(
                    "/co/shift/" + appName.toLowerCase + "/" + name.toLowerCase + "/entity/" + name + ".java",
                    JPATemplate::generate(be, appName, associations, fsa))

                if ((DomainCodeUtilities.getDetailMultipleAssociations(be, associations).size > 0 && !DomainCodeUtilities.hasZeroAssociations(be))
                    fsa.generateFile(
                        "/co/shift/" + appName.toLowerCase + "/" + name.toLowerCase + "/control/I" + name + "DAO.java",
                        DAOInterfaceTemplate::generate(be, appName, associations))
                    fsa.generateFile(
                        "/co/shift/" + appName.toLowerCase + "/" + name.toLowerCase + "/control/" + name + "DAO.java",
                        DAOImplTemplate::generate(be, appName, associations))
                }
```

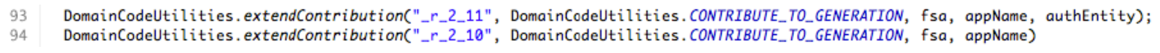


```
82
83                if(DomainCodeUtilities.isMaster(be)){
84                    fsa.generateFile(
85                        "/co/shift/" + appName.toLowerCase + "/web/ext/" + be.name.toLowerCase + "/"+name+"Form.java",
86                        FormTemplate::generate(appName, be))
87                }
88                    fsa.generateFile(
89                        "/co/shift/" + appName.toLowerCase + "/web/ext/" + be.name.toLowerCase + "/"+name+"Controller.java",
90                        WebControllerTemplate::generate(appName, be, associations, fsa))
91            }
92        }
```

Figure 67. DomainCodeGenerator implementation

Line 46 traverses all contents from the domain model in memory (input Resource). Lines 47 to 51 identify the current content being traversed as a *Business*, so its name is stored in the *appName* variable, which is used in the remaining lines. Lines 52 to 92 deal with the appearance of a *BusinessEntity*. According to rule 2 in section 4.2.2.1.1, every *BusinessEntity* must have a domain component and a GUI one. Thus, lines 62 to 81 execute the templates that every domain component must have, that is, a *DTO*, a *Boundary* Interface and its implementation, an *Entity* and the possible appearance of a *DAO*. Notice how the *fsa* object of type *IFileSystemAccess* is used to execute the corresponding template. For example, lines 62 and 63 execute the *DTOTemplate*, where the first parameter of *generateFile* operation is the path where the generated class will be located, and the second one invokes the template named *generate* in the *DTOTemplate* class (remember that an Xtend2 class can define multiple templates). Lines 83 to 91 execute the templates related to the web component. Thus, *FormTemplate* is only executed when the current *BusinessEntity* is a master (see table 2), while every *BusinessEntity* must provide a *UIController*, which is done by lines 88 to 90.

The rest of this class executes the kernel templates for the web layer, as well as the kernel templates related to the database generation. Hence, the *DomainCodeGenerator* is in charge of executing all kernel templates (see figure 57). The execution of the contributed templates is performed through the delegation strategy of section 4.3.2, using the *CONTRIBUTE_TO_GENERATION* modifier. The following code provides an example of a contribution (creation type) using this modifier. Notice that the *fsa* is passed as a parameter to handle the templates execution in the corresponding *Contribution* class, e.g. *NormalTE*.



```
93  DomainCodeUtilities.extendContribution("_r_2_11", DomainCodeUtilities.CONTRIBUTE_TO_GENERATION, fsa, appName, authEntity);
94  DomainCodeUtilities.extendContribution("_r_2_10", DomainCodeUtilities.CONTRIBUTE_TO_GENERATION, fsa, appName)
```

At this point, we are able to produce the source code for a product based on a domain model and a QAs configuration. However, this code cannot be executed as an application without the proper packaging. To handle this issue we decided to use Maven, which is a software project management and comprehension tool that enables managing project's build [113]. We decided to package the generated source code in the following projects:

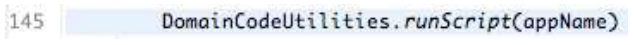
- **co.shift.root** – A wrapper project to encapsulate all generated projects related to a product.
- **co.shift.ear** – A project that encapsulates all domain logic components, so they can be used by the corresponding web components.
- **co.shift.ejb.api** – A module that contains the interface declarations of all domain logic components. It also packages the DTOs definitions to be used as utilities by the domain components.

- **co.shift.ejb** – A module that contains the entire implementation of the domain logic components (Boundaries, Entities, DAOs).
- **co.shift.web** – A project that contains the entire implementation of the web (GUI) components of the product.

We followed the tutorial presented in [114] to create each maven project using the corresponding maven command, and we encapsulated all these tasks into a single operation in our *DomainCodeUtilities* API named *runScript*. This method executes the following steps, after the source code generation has been completed:

1. Creates an .sh file (executable bash file) for each project, which contains the corresponding maven command to generate the desired project.
2. Composes a generation script that executes the generated files in the previous step.
3. Executes the created script in step 2. This execution is performed using the Terminal in mac systems; thus, it isn't compatible with Windows PCs.
4. Waits for step 3 until it finishes its execution.
5. Modifies the .pom file of each generated project to include the corresponding dependencies, e.g. eclipselink for JPA, vaadin for web components.
6. Copies the generated source codes into their corresponding maven project, i.e. boundary interfaces go into co.shift.ejb.api project.
7. Mixes all generated database scripts (kernel script, *Lockout* script and *Authenticator* script) into a single script. **This script is intended to be executing on a MySQL database engine.**

It is very important to invoke *runScript* method in the implementation of *DomainCodeGenerator* class, so the process above can be executed. This line declaration is show in the figure below. Following sections provide an example of how to run the workflow (see figure 55) to generate the code of a product.



```
DomainCodeUtilities.runScript(appName)
```

Note: *runScript* operation uses the constant *GENERATION_DIR* located in *DomainCodeUtilities* class to indicate the path where the generated maven projects will be located.

5. Technologies Involved in Product Line Members Derivation

In order to build a product line member, we developed a configuration process that follows an MDE generation strategy, which consists of three steps: 1) a Functional Configuration describing the functionalities (use cases) to include in the product line member, 2) a Quality Configuration describing the expected levels of QAs, and 3) the execution of the Generation Engine that takes both configurations as inputs to derive the desired product. Figure 68 shows this process. In the following we provide more detail of these steps.

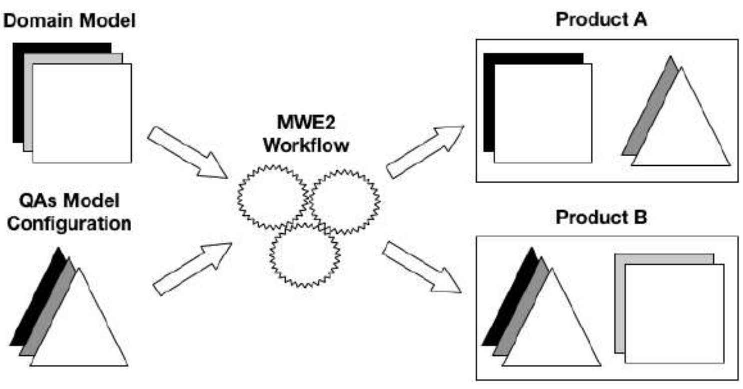


Figure 68. Product Derivation Process

5.1 Create a Model Based on the DMM

To handle model creations based on the DMM we used the Eclipse Modeling Framework (EMF)[8], which is a framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

We created an ECore metamodel to describe the DMM (see figure 5). To be able to use this metamodel as a base for creating models, we developed an eclipse plugin that can be downloaded from [110]. Such plugin contains the DMM Ecore file; the MWE2 workflow explained in section 65; all templates exposed in section 4.3.3.1 and our *DomainCodeUtilities* API. In order to run our plugin an eclipse IDE with all modeling plugins is required (can be downloaded from http://eclipse.org/modeling/downloads/).

After downloading the eclipse for modeling and our plugin, the following projects must be imported: `co.shift.modeling`, `co.shift.modeling.edit` and `co.shift.modeling.editor`. Now, a new instance of eclipse that recognizes our plugin must be launched. This can be done by pressing right click over `co.shift.modeling.edit` project and selecting Run As/Eclipse Application option. Once the new instance starts, we must import the `co.shift.modeling` project into the new workspace. After performing these steps, the recently launched instance of eclipse should look like figure 69.

[8] http://eclipse.org/modeling/emf/

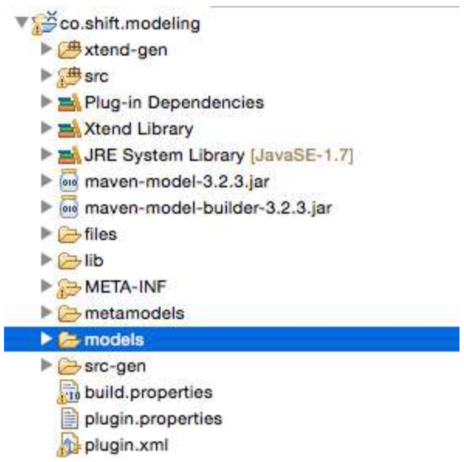


Figure 69. Generation Plugin content

At this point we are ready to configure models from the DMM. All we need to do is right click the models folder and select New/Other option (see figure 70). A wizard will popup, thus, we need to browse the displayed options until we find a folder named "Example EMF Model Creation Wizards". This folder should contain an item named "Domainmetamodel Model", which represents our DMM.

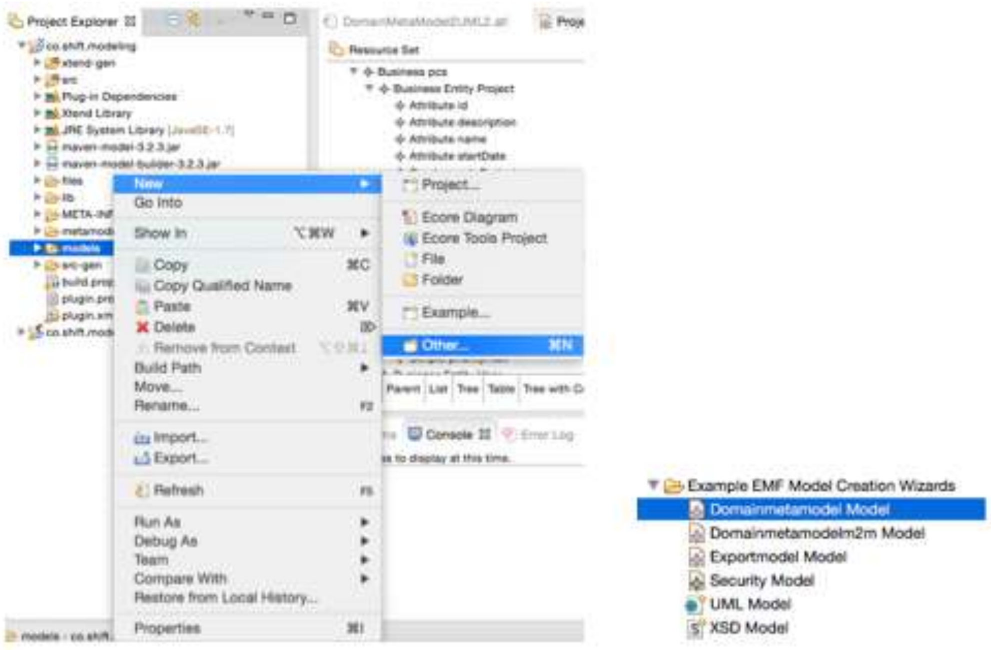
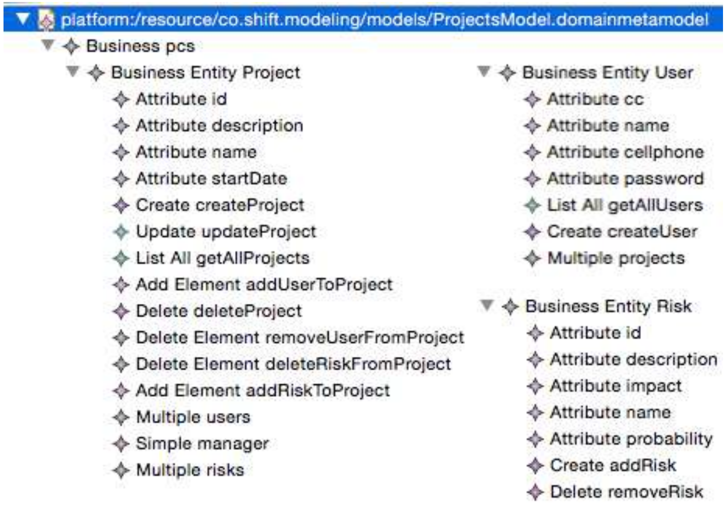


Figure 70. Using our DMM to create Domain Models

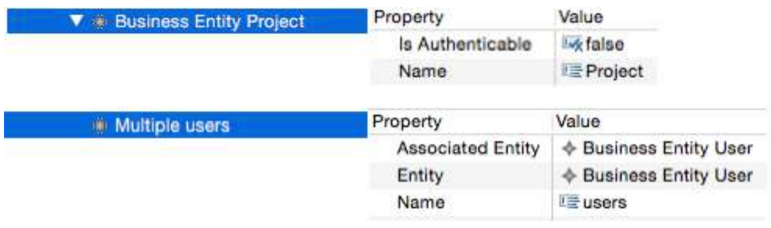
After selecting this option, we provide the model name and press "Next". Here we select Business as the model object (indicating the root element of the model) and press "Finish". A tree view editor will display the *Business* object we selected. This view enables creating a model according to the constraints of the DMM. Following our case study in section 3, we provide the domain model in figure 71 to meet the functionalities of the SPL for Project Management systems.



- platform:/resource/co.shift.modeling/models/ProjectsModel.domainmetamodel
  - Business pcs
    - Business Entity Project
      - Attribute id
      - Attribute description
      - Attribute name
      - Attribute startDate
      - Create createProject
      - Update updateProject
      - List All getAllProjects
      - Add Element addUserToProject
      - Delete deleteProject
      - Delete Element removeUserFromProject
      - Delete Element deleteRiskFromProject
      - Add Element addRiskToProject
      - Multiple users
      - Simple manager
      - Multiple risks
    - Business Entity User
      - Attribute cc
      - Attribute name
      - Attribute cellphone
      - Attribute password
      - List All getAllUsers
      - Create createUser
      - Multiple projects
    - Business Entity Risk
      - Attribute id
      - Attribute description
      - Attribute impact
      - Attribute name
      - Attribute probability
      - Create addRisk
      - Delete removeRisk

Figure 71. Domain Model of our Case Study

Each item of the figure 71 provides some properties (according to its type in the DMM) that must be set. For example, each *BusinessEntity* must indicate its name and weather it is authenticable or not; each *Multiple Association* requires setting its name and its related *BusinessEntity*. Figure 72 shows an example of these properties.



Business Entity Project

Property	Value
Is Authenticable	false
Name	Project

Multiple users

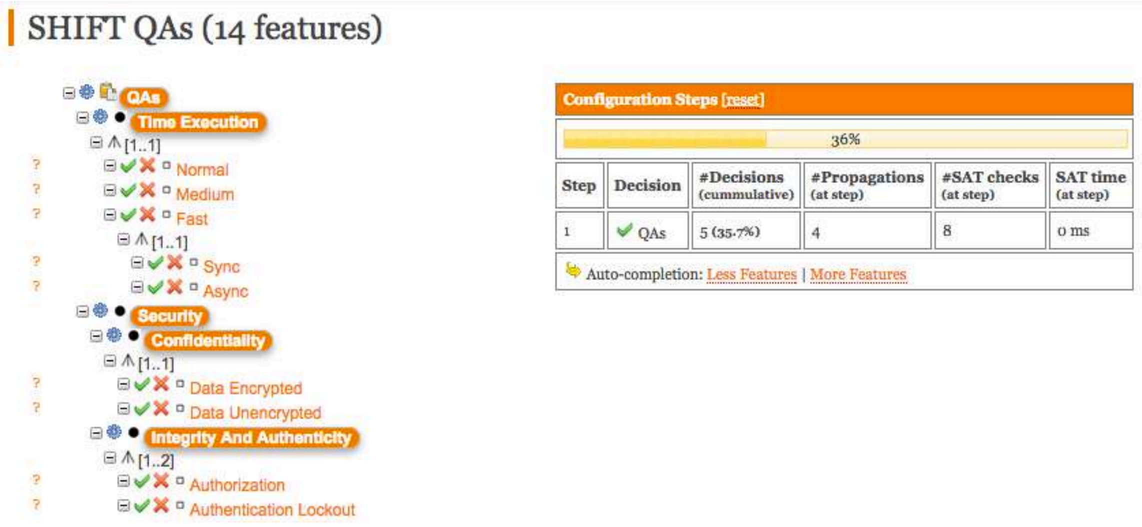
Property	Value
Associated Entity	Business Entity User
Entity	Business Entity User
Name	users

Figure 72. Properties of Domain Model elements

Once these steps are done, a domain model has been created successfully. An important restriction to highlight is that in order to use the created model in the workflow execution, the newly model must be named "ProjectsModel.domainmetamodel". This is due to workflow's implementation, which uses this name to identify the domain model that is going to be used as an input in the generation process.

5.2 SELECT A CONFIGURATION FROM THE QAS VARIABILITY MODEL

To manage the creation and configuration of the QAs variability model, we decided to use the online tool S.P.L.O.T. [115]. This tool enables editing, debugging, analyzing, configuring, sharing and downloading feature models instantly. They provide an online feature model repository where every created model using this tool is saved. We followed the QAs variability model exposed (see figure 7) to create the QAs variability model using S.P.L.O.T. We named our model *SHIFT QAs*. One of the most interesting facilities that S.P.L.O.T. provides is the ability of generating feature model configurations parting from the feature models stored in the online repository. Figure 73 depicts how the QAs variability model looks when it is about to be configured.



SHIFT QAs (14 features)

- QAs
  - Time Execution
    - [1..1]
      - Normal
      - Medium
      - Fast
        - [1..1]
          - Sync
          - Async
  - Security
    - Confidentiality
      - [1..1]
        - Data Encrypted
        - Data Unencrypted
    - Integrity And Authenticity
      - [1..2]
        - Authorization
        - Authentication Lockout

Configuration Steps [reset]

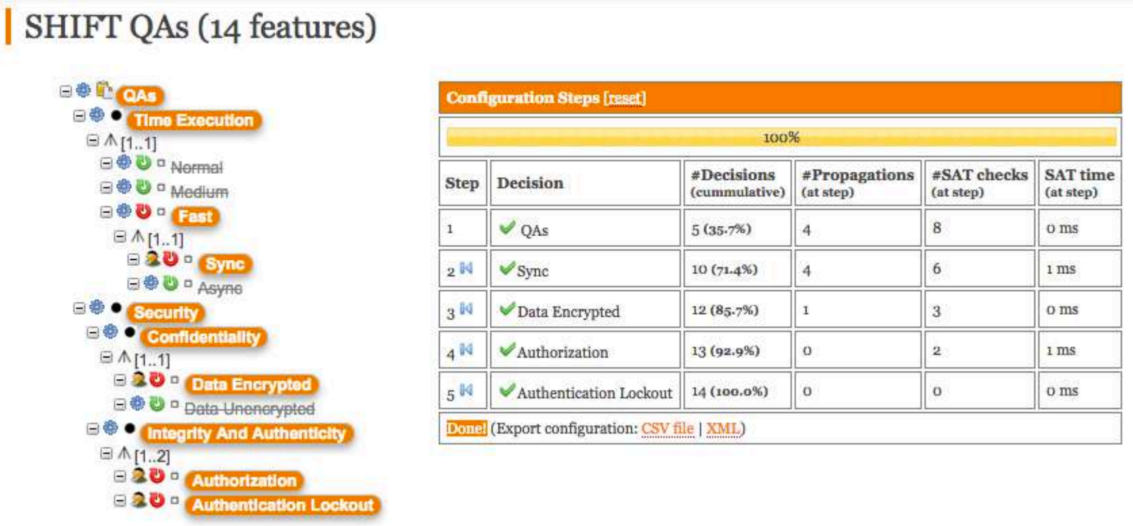
36%

Step	Decision	#Decisions (cummulative)	#Propagations (at step)	#SAT checks (at step)	SAT time (at step)
1	✓ QAs	5 (35.7%)	4	8	0 ms

Auto-completion: Less Features | More Features

Figure 73. QAs model using S.P.L.O.T.

One of the utilities provided by S.P.L.O.T. is displaying a percentage of completion each time a feature is selected/deselected. A valid configuration is one that reaches 100%. Entire management of the constraints inherent to feature models e.g. inclusive/exclusive groups, optional/mandatory features, is handled by the tool. The green icon is used to select a feature; the red x is used to do the opposite. Assuming that we want a product with a *High Sync* level of *Time Execution*, a *Data Encrypted* level of *Confidentiality* and with both *Authorization* and *Authentication Lockout* levels selected, the quality configuration has to be the one shown in figure 74.



SHIFT QAs (14 features)

- QAs
  - Time Execution
    - [1..1]
      - ~~Normal~~
      - ~~Medium~~
      - Fast
        - [1..1]
          - Sync
          - ~~Async~~
  - Security
    - Confidentiality
      - [1..1]
        - Data Encrypted
        - ~~Data Unencrypted~~
    - Integrity And Authenticity
      - [1..2]
        - Authorization
        - Authentication Lockout

Configuration Steps [reset]

100%

Step	Decision	#Decisions (cummulative)	#Propagations (at step)	#SAT checks (at step)	SAT time (at step)
1	✓ QAs	5 (35.7%)	4	8	0 ms
2	✓ Sync	10 (71.4%)	4	6	1 ms
3	✓ Data Encrypted	12 (85.7%)	1	3	0 ms
4	✓ Authorization	13 (92.9%)	0	2	1 ms
5	✓ Authentication Lockout	14 (100.0%)	0	0	0 ms

Done! (Export configuration: CSV file | XML)

Figure 74. QAs model Configuration

Notice that selected features are highlighted in orange, while the remaining ones (unselected) have a strikethrough in their names. There is also a table displaying the selected features only. Once a valid configuration is selected (100%), it can be exported to a CSV file or an XML. In our case, we use the second option. To use the exported XML configuration in our generation engine, this file must be located in the *models* folder of our plugin (see figure 69), and its name must be set to "QAsConfig.xml".

5.3 EXECUTE GENERATION WORKFLOW

After having placed both quality configuration and domain model in the *models* folder of our plugin (see figure 69), the remaining step is to run the MWE2 workflow (see figure 65). To do so, locate the workflow in `src/co.shift.generators.workflows` package, right click the `WF.mwe2` file and select "Run As/MWE2 Workflow" option (see figure 75). This will generate the packages explained in section 4.3.3.2.

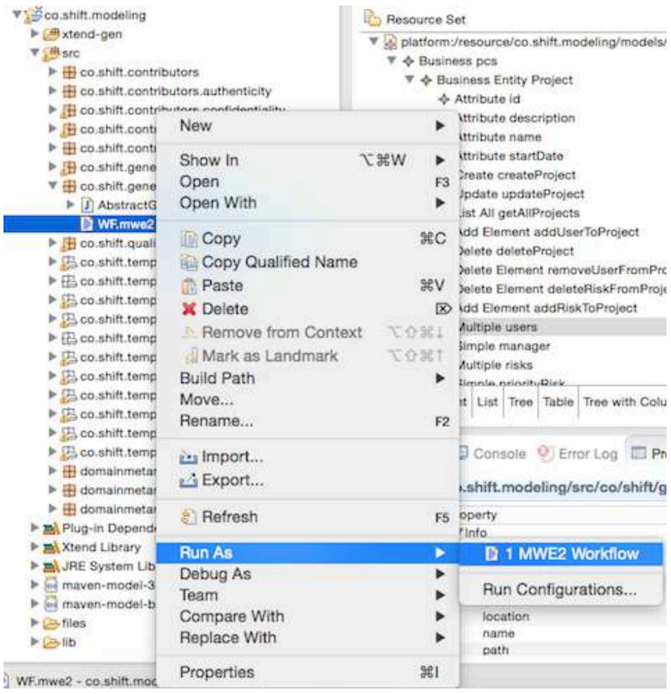


Figure 75. Generation Workflow Execution

Once the workflow finishes its execution, the product is ready to be deployed in an application server like Glassfish[9] or JBoss[10]. Two things must be taken into account when deploying the application: first, the generated database script must be run (in a MySQL database engine) before performing any further actions. The script is located in:

```
~/co.shift.root/co.shift.web/src/co.shift.<<appName>>.web.database.
```

Second, the connection pool and JDBC resources must be configured using the exact same name assigned to the *Business* in the domain model when deploying the app to a server.

6. CONCLUSIONS

This work proposes an approach to support quality attributes variability and enables relating functionalities with the quality levels they promote. Instead of designing and developing additional architectural elements to support multiple quality attributes, it is

9 https://glassfish.java.net
10 http://jbossas.jboss.org

focused on creating an initial Reference Architecture and adapting that architecture according to a selection of quality levels.

We provided a domain metamodel to define the functional scope of product line members. We also created a QAs variability model to indicate the quality attributes (performance and security) that are going to be addressed when configuring products, specifying the quality scope of product line members. We related both domain metamodel and QAs variability model using a coarse-grained approach, indicating that each desired quality level on a target product has to be promoted by every component of such product. Software design patterns promote the configured quality levels and are documented through the Reference Architecture. Finally, we provide tool support to automate derivation of products once a domain model and a configuration of quality attributes have been created. Such tool maps the reference architecture into several templates that contain the logic needed to implement possible configuration decision. Quality domain metamodel and transformations to derive products can be reused in several SPLs that share the same interests (e.g. enterprise applications).

As future work, we intend to extend our quality model to consider the *excludes* relationship to capture conflicting quality attributes, in order to provide a proper mechanism to handle these situations in our tool. We will also adapt our packaging script (see section 4.3.3.2) to be supported on Windows PCs.

7. *BIBLIOGRAPHY*

[1] K M Krandthi, B M Konda, K T Reddy, B R Kiran, and A Vindhya, "A Systematic Mapping Study on Value of Reuse," *International Journal of Computer Applications*, vol. 34, 2011.

[2] K C Kang, S G Cohen, J A Hess, W E Novak, and A S Peterson, "Feature-oriented domain analysis (FODA) feasibility study," *CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST*, 1990.

[3] M Voleter and I Grohen, "Product line implementation using aspect- oriented and model-driven software development," *SPLC*, pp. 233-242, 2007.

[4] O Regan, "A Practical Approach to Software Quality," *Springer-Verlag*, 2002.

[5] M Wieczorek and D Meyerhoff, "Software Quality: State of the Art in Management, Testing, and Tools," *Springer*, 2001.

[6] B B Anderson, A Bajaj, and W Gorr, "An estimation of the decision models of senior IS managers when evaluating the external quality of organizational software," *The Journal of Systems and Software*, pp. 59-75, 2002.

[7] B W Boehm et al., "Characteristics of Software Quality," *American Elsevier*, 1978.

[8] A C Gillies, "Software Quality: Theory and Management," *Chapman & Hall*, 1992.

[9] W E Perry, "Effective Methods of EDP Quality Assurance," *QED Information Sciences, Wellesley, Mass*, 1987.

[10] Sonia Montagud and Silvia Abrahao, "Gathering Current Knowledge about Quality Evaluation in Software Product Lines," *SPLC*, pp. 266-283, 2009.

[11] V Myllärniemi, T Männistö, and M Raatikainen, "Quality Attribute Variability within a

Software Product Family Architecture," *Second Inernational conference on Quality of Software Architecture QoSA*, 2006.

[12] G Halmans and K Pohl, "Communicating the variability of a software-product family to customers," *Journal on Software and Systems Modeling*, pp. 15-36, 2003.

[13] E Niemelä, "Architecture Centric Software Family Engineering," *Product Family Engineering Seminar*, 2005.

[14] H Zhang, S Jarzabek, and B Yang, "Quality Prediction and Assessment for Product Lines," *Springer*, 2003.

[15] G Zhang, "Quality Attributes Assessment for Feature-Based Product Configuration in Software Product Line," *Software Engineering Conference (APSEC)*, 2010.

[16] J Bartholdt, M Medak, and R Oberhauser, "Integrating Quality Modeling with Feature Modeling in Software Product Lines Joerg," *Fourth International Conference on Software Engineering Advances Integrating*, 2009.

[17] Mario Barbacci, Thomas H Longstaff, Mark H Klein, and Charles B Weinstock, "Quality Attributes, Technical Report," *CMU/SEI-95-TR-021*, 1995.

[18] Microsoft. (2008, Jan.) patterns and practices: Application Architecture Guide 2.0. [Online]. http://apparchguide.codeplex.com/wikipage?title=Chapter%207%20-%20Quality%20Attributes

[19] L Bass, P Clements, and R Kazman, *Software Architecture in Practice*. Boston: Addison-Wesley Professional, 2012.

[20] M Barbacci, M Klein, T Longstaff, and S Weinstock, "Quality Attribute Workshops," *CMU/SEI-95-TR-21* , 1995.

[21] ISO/IEC. (2011) ISO/IEC 25010:2011. [Online]. http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733

[22] L Bass, P Clements, and R Kazman, *Software Architecture in Practice*, 2nd ed.: Addison-Wesley, 2003.

[23] Paul Clements et al., *Documenting Software Architectures: Views and Beyond*, 2nd ed. Boston: Addison Wesley, 2010.

[24] SEI. (2014) Sotware Architecture Overview. [Online]. http://www.sei.cmu.edu/architecture/

[25] Microsoft. (2014) What is Software Architecture? [Online]. http://msdn.microsoft.com/en-us/library/ee658098.aspx#GoalsofArchitecture

[26] Joseph E Hollingsworth, "One Architecture Does Not Fit All: Micro-Architecture Is As Important As Macro-Architecture," *Proceedings of the Seventh Workshop on Institutionalizing Software Reuse*, 1995.

[27] Oliver Vogel, Ingo Arnold, Arif Chughtai, and Timo Kehrer, *Software Architecture: A Comprehensive Framework and Guide for Practitioners*.: Springer Science & Business Media, 2011.

[28] Steve Edwards, "Micro-Architecture of Software Components and The Need For Good Mental Models of Software Subsystems," *ACM SIGSOFT Software Engineering Notes*, pp. 46-50, 1996.

[29] Deepak Alur, Dan Malks, and John Crupi, *Core J2EE Patterns: Best Practices and Design Strategies*.: Prentice Hall, 2003.

[30] Ivica Crnkovic and Magnus Larsson, "Challenges of Component-based Development," *Journal of Software Systems*, 2001.

[31] Martin Fowler, *Patterns of Enterprise Application Architecture*.: Addison Wesley, 2002.

[32] P Clements, R Kazman, and M Klein, *Software Architecture in Practice*.: Addison Wesley Publishing Comp. , 2003.

[33] N Rozanski and E Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*.: Addison Wesley Publishing Comp. , 2005.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and Jhon Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co, 1995.

[35] Christopher Steel, Ramesh Nagappan, and Ray Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*.: Prentice Hall, 2006.

[36] Munawar Hafiz. (2013, Mar.) Security Pattern Catalog. [Online]. http://www.munawarhafiz.com/securitypatterncatalog/

[37] Adam Bien, *Real World Java EE Patterns Rethinking Best Practices*.: lulu.com, 2009.

[38] Oracle and/or its affiliates. Java Platform, Enterprise Edition (Java EE). [Online]. http://www.oracle.com/technetwork/java/javaee/overview/index.html

[39] Adam Bien, *Real World Java EE Patterns-Rethinking Best Practices*.: lulu.com, 2012.

[40] Adam Bien. (2012) about.adam-bien. [Online]. http://about.adam-bien.com

[41] Oracle and/or its affiliates. (2011) Java Champion Bios. [Online]. https://java.net/website/java-champions/bios.html#Bien

[42] Oracle Magazine. (2010, Nov.) Editors' Choice Awards 2010: Delivering Innovation. [Online]. http://www.oracle.com/technetwork/issue-archive/2010/10-nov/o60eca-176293.html#bien

[43] Adam Bien. (2014) Adam Bien's Workshops. [Online]. http://workshops.adam-bien.com

[44] Adam Bien. (2014, Aug.) Adam Bien's Weblog. [Online]. http://www.adam-bien.com/roller/abien/

[45] Adam Bien. (2012) tv.adam-bien.com. [Online]. http://tv.adam-bien.com

[46] David Kalinsky. (2002, July) Design Patterns for High Availability. [Online]. http://www.embedded.com/design/prototyping-and-development/4024434/Design-Patterns-for-High-Availability

[47] Microsoft. (2014) Performance and Reliability Patterns. [Online]. http://msdn.microsoft.com/en-us/library/ff648802.aspx

[48] P Reed. (2002) The Rational Edge. [Online]. www-128.ibm.com/developerworks/rational/library/2774.html

[49] E Y Nakagawa, P O Antonino, and M Becker, "Reference architecture and product line architecture: A subtle but critical difference," *ECSA'2011*, pp. 207-211, 2011.

[50] E Nakagawa, "Reference Architectures and Variability: Current Status and Future Perspectives," *WICSA/ECSA*, 2012.

[51] OMG. Unified modeling language (UML). [Online]. http://www.omg.org/spec/UML/

[52] Paul C Clements, "A Survey of Architecture Description Languages," *Proceeding IWSSD '96 Proceedings of the 8th International Workshop on Software Specification and Design*, 1996.

[53] Oracle and/or its affiliates. (2012) Overview of Enterprise Applications - Your First Cup: An Introduction to the Java EE Platform. [Online]. http://docs.oracle.com/javaee/6/firstcup/doc/gcrky.html

[54] Red Hat. JBoss Application Server 7. [Online]. http://jbossas.jboss.org

[55] T Stahl, M Voelter, and K Czarnecki, *Model-Driven Software Development: Technology, Engineering, Magagement.*: Wiley, 2006.

[56] J Bézivin, "On the Unification Power of Models. Software and Systems Modeling," *Software and Systems Modeling*, vol. 4, pp. 171-188, 2005.

[57] J Oldevik, "Transformation composition modelling framework," in *Distributed Applications and Interoperable Systems*, 2005, pp. 108-114.

[58] J Bosch, *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*. Boston: Addison-Wesley, 2000.

[59] P Clements and L Northrop, *Software Product Lines : Practices and Patterns*. Boston: Addison-Wesley Professional, 2002.

[60] K Pohl, G Bockle, and F Van Der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. Berlin: Springer, 2005.

[61] K Czarnecki and U W Eisenecker, *Generative Programming: Methods, Tools, and Applications*. New York: ACM Press/Addison-Wesley Publishing Co, 2000.

[62] K Czarnecki, S Helsen, and U W Eisenecker, "Staged Configuration Using Feature Models," *Proceedings of the 3th Software Product Line Conference 2004*, pp. 266-282, 2004.

[63] K Kang, S Cohen, J Hess, W Nowak, and S Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study, Report.*: Software Engineering Institute and Carnegie Mellon University, 1990.

[64] H Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. 2004: Addison Wesley Longman Publishing Co, Redwood City.

[65] H Arboleda and J C Royer, *Model-Driven and Software Product Line Engineering*. Londres: ISTE Ltd and J. Wiley & Sons, 2012.

[66] Ø Haugen, A Wasowski, and K Czarnecki, "CVL: common variability language," *SPLC*, pp. 266-267, 2012.

[67] Hugo Arboleda, Ruby Casallas, Jaime Chavarriaga, and Jean Claud Royer, "Software architecture for product lines," in *Architectures logicielles : Principes, techniques et outils.*: Lavoisier, 2014.

[68] ISO/IEC. (2013, Aug.) ISO/IEC 26550. [Online]. http://www.iso.org/iso/catalogue_detail.htm?csnumber=43075

[69] E Y Nakagawa, P O Antonino, and M Becker, "Exploring the Use of Reference Architectures in the Development of Product Line Artifacts," *SPLC*, 2011.

[70] S Angelov, J Trienekens, and P Grefen, "Towards a Method for the Evaluation of Reference Architectures: Experiences from a Case," *Proceedings of the Second European Conference on Software Architecture*, 2008.

[71] C Atkinson, J Bayer, and D Muthig, "Component-Based Product Line Devel- opment: The KobrA Approach," *Proceedings of 1st Software Product Line Conference*, pp. 289-309, 2000.

[72] J Bayer, O Flege, and C Gacek, "Creating Product Line Architectures," *IW- SAPF-3: Proceedings of the International Workshop on Software Architectures for Product Families*, pp. 210-216, 2000.

[73] D Dhungana, P Grunbacher, and R Rabiser, "DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling," *Proceedings of the 2nd Int. Workshop on Variability Modelling of Software-intensive Systems*, 2008.

[74] T Forster, D Muthig, and D Pech, "Understanding Decision Models. Visualiza- tion and Complexity reduction of Software Variability," *Proceedings of the 2nd Int. Work- shop on Variability Modeling of Software-intensive Systems*, 2008.

[75] D Dhungana, P Grunbacher, and R Rabiser, "The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study," *Automated Software Engg*, vol. 18, pp. 77-114, 2011.

[76] A Rashid, J C Royer, and A Rummler, "Aspect-Oriented, Model-Driven Software Product Lines, The AMPLE Way," *Cambridge University Press*, 2011.

[77] H Arboleda, H Romero, R Casallas, and J C Royer, "Product Derivation in a Model-Driven Software Product Line using Decision Models," *Proceedings of the 12th Iberoamerican Conference on Requirements Engineering and Software Environments (IDEAS'09)*, pp. 59-72, 2009.

[78] D Wageelar, "Context-Driven Model Refinement," *vol. 3599 of Lecture Notes in Computer Science*, pp. 189-2003, 2005.

[79] L Tan, Y Lin, and H Ye, "Modeling Quality Attributes in Software Product Line Architecture," *Engineering and Technology (S-CET)*, 2012.

[80] L Etxeberria, G Sagardui, and L Belategi, "Quality aware software product line engineering," *Comp. Soc.*, pp. 57-69, 2008.

[81] B González-Baixauli, J Leite, and J Mylopoulos, "Visual Variability Analysis for Goal Models," *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04)*, pp. 198-207, 2004.

[82] D Benavides, P Trinidad, and A Ruiz-Cortés, "Automated Reasoning on Feature Models," *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, pp. 491-503, 2005.

[83] D Benavides, S Segura, P Trinidad, and A Ruiz-Cortés, "A first step towards a framework for the automated analysis of feature models," *Managing Variability for Software Product Lines: Working With Variability Mechanisms workshop (SPLC'06)*, 2006.

[84] Javier G Huerta, Emilio Insafran, Silvia Abrahão, and John D McGregor, "Non-Functional Requirements in Model-Driven Software Product Line Engineering," *NFSP-DSML'12*, 2012.

[85] Z Gürses, "Non-Functional V ariability Management by Complementary Quality Modeling in a Software Product Line. Master of Science in Electrical and Electronics Engineering," 2010,.

[86] G Sagardui , L Belategui, L Etxeberria, and A Noguero. (2010) Quality Aware Product Line Domain Engineering. E-Diana Technical Report. [Online]. http://s15723044.onlinehome-server.info/artemise/documents/D2_1_E_QUALITY_AWARE_PRODUCT_LINE_DOMAIN_ENGINEERING_METHOD_M15_MULE.pdf

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

[88] I John and D Muthig, "Tailoring use cases for product line modeling," *Proceedings of the International Workshop on Requirements Engineering for product lines*, pp. 26-32, 2002.

[89] Robert Biddle. (2001, Apr.) Patterns of Use Cases. [Online]. http://www.mcs.vuw.ac.nz/research/object/Papers/euc-html/node12.html

[90] Oracle. (2014) Master Detail. [Online]. http://www.oracle.com/webfolder/ux/applications/fusiongps/patterns/content/recordnavigat

ion/masterdetail/index.htm

[91] Apple Inc. (2014) Creating a Master-Detail Interface. [Online]. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaBindings/Tasks/masterdetail.html

[92] F Ceballos, H Arboleda, and R Casallas, "Un Enfoque para Desarrollar Aplicaciones WEB Basado en Líneas de Producto Dirigidas por Modelos," *Paradigma – Revista Electrónica en Construcción de Software*, Nov. 2008.

[93] X Peng, S Won Lee, and W Yun Zhao, "Feature-Oriented Nonfunctional Requirement Analysis for Software Product Line," *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY*, pp. 319-338, 2009.

[94] Oracle. (2014) Interface DataSource. [Online]. http://docs.oracle.com/javase/7/docs/api/javax/sql/DataSource.html

[95] Oracle. (2014) Annotation Type PersistenceContext. [Online]. http://docs.oracle.com/javaee/6/api/javax/persistence/PersistenceContext.html

[96] Oracle. (2014) Oracle® Security Developer Tools Reference. [Online]. https://docs.oracle.com/cd/B14099_19/idmanage.1012/b15975/crypto.htm

[97] Oracle. (2014) Java Security API. [Online]. http://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html

[98] Di Management. (2014) RSA Algorithm. [Online]. http://www.di-mgt.com.au/rsa_alg.html

[99] Wikipedia. (2014) Advanced Encryption Standard. [Online]. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[100] E Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*.: Wiley, 2013.

[101] M D Elder, D Tyree, and J Edwards-Hewitt. (2014) Security Patterns Repository Version 1.0. [Online]. http://www.scrypt.net/~celer/securitypatterns/template%20and%20tutorial.pdf

[102] Oracle. (2014) Enterprise JavaBeans Technology. [Online]. http://www.oracle.com/technetwork/java/javaee/ejb/index.html

[103] Oracle. (2014) Java Persistence API. [Online]. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[104] D Trowbridge et al., "Enterprise Solutions Patterns Using Microsoft.Net," *Microsoft Press*, 2003.

[105] Martin Fowler. (2006) GUI Architectures. [Online]. http://martinfowler.com/eaaDev/uiArchs.html

[106] Agile Modeling. (2014) Robustness Diagrams: An Agile Introduction. [Online]. http://www.agilemodeling.com/artifacts/robustnessDiagram.htm

[107] Oracle. (2013) The Java EE 6 Tutorial. [Online]. http://docs.oracle.com/javaee/6/tutorial/doc/gipjg.html

[108] OMG. (2008) Mof Model To Text Transformation Language (Mofm2T), 1.0. [Online]. http://www.omg.org/spec/MOFM2T/1.0/

[109] Eclipse.org. (2014) MDT/UML2. [Online]. http://wiki.eclipse.org/MDT-UML2

[110] David Duran. (2014) SHIFT Project Repository. [Online]. https://github.com/unicesi/SHIFT

[111] Eclipse.org. (2014) Papyrus User Guide. [Online]. http://wiki.eclipse.org/Papyrus_User_Guide#Create_a_diagram_from_an_existing_uml_fil

e
[112] Eclipse.org. (2014) Xtend - Modernized Java. [Online]. http://eclipse.org/xtend/
[113] The Apache Software Foundation. (2014) Welcome to Apache Maven. [Online]. http://maven.apache.org
[114] Max Lam. (2014) Building and Deploying Java EE EAR with Maven to Java EE Application Server. [Online]. http://www.developerscrappad.com/1177/java/java-ee/maven/building-and-deploying-java-ee-ear-with-maven-to-java-ee-application-server-part-1-project-directory-structure-amp-module-generation-through-archetype-generate/
[115] Marcilio Mendonca. (2010) S.P.L.O.T. - Software Product Lines Online Tools. [Online]. http://www.splot-research.org
[116] Oracle. (2014) Interface Future. [Online]. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

ANNEX I. QUERY DECLARATION EXAMPLE

```
@Entity
@Table(name = "Project")
@NamedQueries({
      @NamedQuery(name  =  "project.getAllProjects",  query  =  "SELECT  p
FROM Project p")
})
public class Project implements Serializable {
      ...
}
```

ANNEX II. NORMAL TIME EXECUTION IMPLEMENTATION

Following code fragment illustrates how *ProjectManager* uses the *EntityManager* to retrieve all projects from database:

```
1. public List<ProjectTO> getAllProjects() {
2.          List<ProjectTO> projects = new ArrayList<>();
3.          TypedQuery<Project> query = em.createNamedQuery(
4.                      "project.getAllProjects", Project.class);
5.          List<Project> foundProjects = query.getResultList();
6.          for (Project project : foundProjects) {
7.                ProjectTO to = new ProjectTO();
8.                to.setId(project.getId());
9.                to.setDescription(project.getDescription());
10.               to.setName(project.getName());
11.               to.setStartDate(project.getStartDate());
12                projects.add(to);
13.         }
14.         return projects;
15.   }
```

Lines 2 to 5 use the *EntityManager* to create and execute the retrieval query. The results are saved into a list of project entities. Lines 6 to 13 encapsulate each found project in its representing *TO* and save it into the project *TOs* list, which is return to *ProjectController* in line 14.

ANNEX III. MEDIUM TIME EXECUTION IMPLEMENTATION

The use of *DataSource* resource requires injecting it in the realization of *IProjectManager*, which is *ProjectManager* in our case (see figure 17). This injection must be performed as follows.

```
@Resource(name=”DataSourceName”)
private DataSource ds;
```

Following code fragment illustrates how *ProjectBasicFLR* uses the *DataSource* to retrieve all projects from database:

```
1. public List<ProjectTO> getAllProjects() throws Exception {
2.    List<ProjectTO> projects = new ArrayList<>();
3.    Connection con = null;
4.    Statement stmt = null;
5.    ResultSet resultSet = null;
6.    try {
7.          con = ds.getConnection();
8.          stmt = con.createStatement();
9.          resultSet = stmt.executeQuery("SELECT p.* FROM Project p");
10.         ProjectTO p;
11.         while (resultSet.next()) {
12.               p = new ProjectTO();
13.               int tId = resultSet.getInt(1);
14.               String tDescription = resultSet.getString(2);
15.               String tName = resultSet.getString(3);
16.               Date tStartDate = resultSet.getDate(4);
17.               p.setId(tId);
18.               p.setDescription(tDescription);
19.               p.setName(tName);
20.               p.setStartDate(tStartDate);
21.               projects.add(p);
22.         }
23.         return projects;
24.   } catch (SQLException ex) {
25.         throw new Exception(ex.getMessage());
26.   }
27. }
```

Line 7 uses the *DataSource* to get the database connection. Lines 8 and 9 prepare and execute the retrieval query. Lines 11 to 22 deal with the encapsulation of the primitives contained in each record of the *resultSet* to a corresponding *TO*. Once the *TO* is created and initialized, it is added to the *TOs* list in line 21. Line 23 returns the resulting *TOs* list to the *Boundary*.

ANNEX VI. HIGH SYNC TIME EXECUTION IMPLEMENTATION

The following code shows the implementation of data retrieval in *ProjectOptimizedFLR* (see figure 20).

```
1. public List<ProjectTO> getAllProjects(int start, int maxResults)
2. throws Exception {
3.    List<ProjectTO> projects = new ArrayList<>();
4.    Connection con = null;
5.    Statement stmt = null;
6.    ResultSet resultSet = null;
7.    try {
8.          con = ds.getConnection();
9.          stmt = con.createStatement();
10.         resultSet = stmt.executeQuery("SELECT p.* FROM Project p");
11.         if (start != 0)
12.               resultSet.absolute(start);
13.         int i = 0;
14.         ProjectTO p;
15.         while (resultSet.next()  && i < maxResults) {
16.               p = new ProjectTO();
17.               int tId = resultSet.getInt(1);
18.               String tDescription = resultSet.getString(2);
19.               String tName = resultSet.getString(3);
20.               Date tStartDate = resultSet.getDate(4);
21.               p.setId(tId);
22.               p.setDescription(tDescription);
23.               p.setName(tName);
24.               p.setStartDate(tStartDate);
25.               projects.add(p);
26.               i++;
27.         }
28.         return projects;
29.   } catch (SQLException ex) {
30.         throw new Exception(ex.getMessage());
31.   }
32. }
```

Line 8 uses the *DataSource* to get the database connection. Lines 9 and 10 prepare and execute the retrieval query. Lines 11 and 12 set the current page based on *start* value. Lines 15 to 27 deal with the encapsulation of the primitives contained in each record of the *resultSet* to a corresponding *TO*. It is important to highlight the condition in line 15, which stops retrieving data when `i` counter reaches the value contained in *maxResults*. Line 28 returns the resulting *TOs* list to the *Boundary*.

ANNEX V. HIGH ASYNC TIME EXECUTION IMPLEMENTATION

The following code fragment shows the implementation of `listAllProjects(count)` service of *ProjectParallelizer* (see figure 23). Keep in mind that it parts from the previous injection of *ProjectAsyncWorker* bean.

```
// ProjectAsyncWorker injection
@EJB
ProjectAsyncWorker worker;

1. public List<ProjectTO> getAllProjects(long pCount) {
2.    List<Future<List<ProjectTO>>> futures = new LinkedList<>();
3.    List<ProjectTO> projects = new LinkedList<>();
4.    int start = 0;
5.    int maxResults = 1;
6.    int iterations = (int) (pCount / maxResults);
7.    for (int i = 0; i < iterations; i++) {
8.          futures.add(worker.getAllProjects(start, maxResults));
9.          start += maxResults;
10.    }
11.    for (Future<List<ProjectTO>> future : futures) {
12.          try {
13.                projects.addAll(future.get());
14.          } catch (InterruptedException e) {
15.                // TODO Auto-generated catch block
16.                e.printStackTrace();
17.          } catch (ExecutionException e) {
18.                // TODO Auto-generated catch block
19.                e.printStackTrace();
20.          }
21.    }
22.    return projects;
23. }
```

Line 2 creates the Future list that will store results from each thread. A Future in java represents the result of an asynchronous computation [116]. Line 4 and 5 initialize the page and chunk parameters. Line 6 calculates how many Threads will be needed to retrieve the project lists. Lines 7 to 10 launch each Thread to start retrieving data. Lines 11 to 20 iterate over the instantiated futures, and get the resulting list from each one (line 13). It is important to have two main loops because the use of the `get()` method of a Future freezes the current execution Thread, thus, if it is invoked within the same loop, each thread must be fully executed to launch the next one, causing a sequential retrieval of data. Implementation of `listAllProjects(start, maxResults)` from *ProjectAsyncWorker* is the same as the one shown in "high sync level" section. The only difference is that the resulting list is wrapped into a Future to enable asynchronous operations.