Revista de Sistemas de Informação da FSMA
n. 12 (2013) pp. 8-20

SISTEMAS DE INFORMAÇÃO
http://www.fsma.edu.br/si/sistemas.html

TRILHA PRINCIPAL

Adaptive Software Development supported by an Automated Process: a Reference Model

Frank J. Affonso, *Assistant professor at UNESP (Univ Estadual Paulista)*,
Maria C. V. S. Carneiro, *Assistant professor at UNESP (Univ Estadual Paulista)*
Evandro L. L. Rodrigues, *Associate professor at USP (University of São Paulo)*, and
Elisa Y. Nakagawa, *Associate professor at USP (University of São Paulo)*

***Abstract*—This article presents a reference model as an automated process to assist the adaptive software development at runtime, also known as Self-adaptive Systems (SaS) at runtime. This type of software has specific characteristics in comparison to traditional one, since it allows that changes (structural or behavioral) to be incorporated at runtime. Automated processes have been used as a feasible solution to conduct software adaptation at runtime by minimizing human involvement (developers) and speeding up the execution of tasks. In parallel, reference models have been used to aggregate knowledge and architectural artifacts, since they capture the systems essence in specific domains. However, presently no there is reference model based on reflection for the automation of software adaptation at runtime. In this scenario, this article presents a reference model based on reflection, as an automated process, for the development of software systems that require adaptation at runtime. To show the applicability of the model, a case study was conducted and a good perspective to efficiently contribute to the area of SaS has been obtained.**

***Index Terms*—Self-adaptive software, Automated process, Reference model, Development of software systems.**

I. Introduction

SOFTWARE systems have played an important role in the modern society, since they are present in many segments and ensure the services provision by government or private institutions. Actually, increasing needs for systems adaptation have been noticed, justified by the constant changes in the requirements of users, which correspond to the emergence of new needs of stakeholders or adaptation of current technologies [1] [2] [3] [4]. Moreover, such systems must be prepared to operate under adverse conditions maintaining their integrity and characteristics, such as robustness, reliability, scalability, customization, self-organization, and self-adaptation have been increasingly required. The latter three characteristics fit a specific context of software engineering, which represents the self-adaptive systems[1] at runtime. These systems are considered specific for enabling new requirements (structural and/or behavioral) to be incorporated with no interruption in the execution [2] [4] [5] [6] [7]. They also require methodologies, such as Reconfigurable Software Development Methodology (RSDM), Reconfigurable Execution Environment (REE), and automatic mechanisms (automated processes) able to monitor and modify the software at runtime [8] [9] [10].

The RSDM offers a set of guidelines to assist software engineers in the software development in layers. For example, the logical layer has only software entities[2] with attributes and access methods (getters and setters). Another important characteristic of the RSDM is that it uses a flexible architectural pattern, which enables software entities to receive modifications naturally, facilitating the actuation of automated processes [5] [6] [11]. When a software entity is developed, the software engineer must define the information to be modified (structural and/or behavioral). In the next phase, this information is used in the adaptation (automated processes) of the software entity at runtime without the developers' participation, reducing the generation of uncertainties in the adaptation process [8] [9] [10].

The automatic mechanisms represent a comprehensive and complex solution that acts in the software adaptation at runtime. These mechanisms are organized into a reference model composed of seven modules (Section IV). Such modules automate the guidelines prescribed in the RSDM and supervise the software entities in the REE, analyzing requests for modifications and performing them whenever possible [8] [9] [10] [12].

This article presents a reference model for SaS development, which contains a set of modules that monitor and adapt software systems (entities) at runtime. The model uses computational reflection, which is an important resource for the inspection and modification of software. In

Corresponding author: Frank J. Affonso, *frank@rc.unesp.br*

[1] The term self-adaptive systems is used in different areas/domains. In this article, it will focus only in the software domain, so it will be referenced as Self-adaptive Software (SaS).
[2] From this point onwards, SaS will be also referred to as software entities or simply entities.

order to show the viability of the model, a case study was conducted. As a main result, this reference model can be considered an important contribution to the area of SaS.

The article is organized as follows: Section II presents the background and related work; Section III describes the process for the SaS development; Section IV presents the reference model; Section V describes a study case; Section VI provides a brief discussion on the results and limitations of the model; finally, Section VII concludes the article and presents some future works.

II. Background and Related Work

THIS section presents the background (concepts and definitions) and related work that contributed to the development of this reference model. Initially concepts of reflection and its adaptation techniques are described. Next the related work on software engineering tools and software architecture is addressed.

A. Background

Computational reflection, or simply reflection, can be defined as any activity performed by a system on itself, and is very similar to human reflection. The main goal is to obtain information about its own activities, aiming to improve its performance, introduce new capabilities (structural and/or behavioral), or even solve its problems by choosing the best procedure. Furthermore, the use of reflection enables the software to be more flexible and susceptible to changes [1] [2] [3] [4].

Tanter et al. [13] and Janik & Zielinski [14] developed two techniques for software adaptation based on Aspect-Oriented Programming (AOP). The first acts on structural and/or behavioral adaptations through weaving functional and non-functional requirements by a tool named Reflex. The second adapts the non-functional requirements, leaving the functional level intact. The authors proposed an AOP extension, named AAOP (Adaptable AOP), which represents a set of adaptable non-functional aspects integrated into the software system at runtime.

In Borde et al. [15], Hussein & Gomaa [16], Kasten & McKinley [17], and Bastide [18], reflection is used as a technique for the software components adaptation into structure and behavior. Basically, software components are wrapped due to original interface incompatibilities. The existing functionalities are preserved and others, related to new requirements are added, constituting a new software component. Kim et al. [19] approached the same technique (components wrapping), but they commented on their preoccupation to control the size of software components when an adaptation is performed, since these components can increase in size quickly and this could make unfeasible future adaptations.

B. Related Work

According to Whitehead [20], Nakagawa & Maldonado [21], Nakagawa et al. [22], and Dong et al. [23], software engineering has been facing problems concerning the existence of methods and tools that assist the software engineers in all development stages of a software life cycle. Although a large number of tools has been directed to solving these problems, few of tools have attended the needs of developers, due to changes in the systems development tendencies. However, when some tools attend, there is a problem related to the high cost, which prevents their acquisition by small and medium companies.

Henttonen & Matinlassi [24] evaluated and classified Software Engineering Tools (SET) directed to the reuse and sharing of information in a distributed environment. The authors commented on the use of repositories as a mechanism for storage and information reuse and highlighted the criteria of communication among the developers as a success factor in the elaboration of reuse-oriented software projects. The necessity of an architectural pattern is mentioned by these authors as an important factor to increase the software reuse.

According to Gray [25], SET reduces the time and cost of development in a software project. These author commented on the existence of many tools for specific purposes and their cooperation in the execution of a software project. A brief classification of SET is also provided by author. These tools are directed to specific problems and the most extensive, which acts in various phases of a software life cycle.

For Williams & Carver [26] and Hongzhen & Guosun [7], the changes that occur in the life cycle software are inevitable. The new needs of stakeholders and issues related to technological adaptations are the principal reasons for the changes. The authors point out a study on the flexibility of the software architecture to attend the required changes without burdening the maintenance activity and some indicators that measure the impact changes that might occur. These indicators are organized in a taxonomy, which can help the software engineer in the design and maintenance of the software.

A framework for the analysis and design of Reference Software Architectures (RSA) can be found in Angelov et al. [27]. According to the authors, the RSA directly influences the quality and design of concrete architectures and systems generated based on these architectures. The framework is composed of a set of 24 RSA, organized and classified in a multi-dimensional space contemplating five categories of reference architectures and facilitating the analysis and design of new systems.

According to Shi et al. [5] and Peng et al. [6], the reflection has been successfully used in the reuse of software components and it has been implemented on a large-scale in the reuse of software architecture. Both authors divided the software architecture into two levels: (i) meta, which contains the architectural components, information on the base-level as architecture topology, components, and connectors; and (ii) base, which can be considered as the traditional software architecture.

Considering the studies presented in this section, three considerations can be highlighted. First, important initiatives of using reflection in the development of SaS have been taken. Second, the interest in the development of SaS for different domains, since the diversity of related work shows the relevance of reference model and reference architecture for SaS. Third, the importance of STE and automated processes in the execution of task in a software projects. Specifically in the SaS context, this consideration represents an excellent alternative to maximize the speed of the development of SaS and minimize the involvement of developers in the adaptation process.

III. Process for the Development of Adaptive Software

THE development of adaptive software has specific characteristics compared to traditional one, since this type of software allows changes to be incorporated at runtime. It also requires different approaches, such as methodologies, tools (automated processes) and a flexible architectural model. This section presents an overview of the development of adaptive software focused on a methodology (RSDM), an execution environment (REE) and automated processes [8] [9] [10]. Figure 1 shows the main phases of the development process of the adaptive software.

The methodology (RSDM) assists software engineers in the development of SaS. Basically, the software entities developed with RSDM guidelines (part A of Figure 1) are composed of attributes and access methods (getters and setters). In this step, software engineers can use SET to model the systems diagrams (Part A of Figure 1). Then, aided by the annotation module (`annotationManager`), they define the adaptation levels of each software entity. The entities developed in this step are transferred to the REE by an automated process (Transition arrow between Parts A and B of Figure 1) [8] [9] [10].

When the entities are imported into REE (Part B of Figure 1), they undergo a verification process to ensure the annotations (adaptation level) have been implemented. Such entities are then transferred to information repositories and installed in the REE, occupying two states: (1) **running**, when at least a client application (stakeholder) uses one or more implemented functionalities, or (2) **storage**, when the entity is inactive, "sleeping" in information repositories and waiting to be invoked [8] [9] [10].

To finish this section, it is noteworthy that the software entities are organized in an architectural pattern into four layers: presentation, middleware, application, and persistence. The presentation layer represents the user's interface (clients of system). The middleware layer is related to the architectural styles of the system, such as (i) client-server on the web, (ii) web services, and (iii) invocation of remote objects. The application layer represents the system's logics and the system's business logics. Finally, the persistence layer represents the database connectivity. This architectural pattern offers no salutary innovation, however the system organization into layers favors the action of the modules of reference model that it will be presented in the next section [7] [8] [9] [10] [26].

IV. Reference Model as an Automated Process for the Development of Adaptive Software

THIS section presents a reference model based on reflection for automating the SaS development. This model represents a real solution (abstraction), based on a particular domain (self-adaptive systems) and experience (patterns) to treat the software adaptation at runtime [7] [8] [9] [10] [26]. The solution adopted for this model is directed to systems developed in programming language that have the following features: reflection, dynamic compilation and dynamic loading. Reflection is associated with the architectural flexibility of the software entity, since the information on its structure and execution state can be retrieved and reused when the entity is modified. The dynamic compilation and dynamic loading of software entities are related to how they can be obtained, compiled and reinserted in the execution environment [2] [3] [4] [5] [6] [8] [9] [10]. Figure 2 shows this reference model organized into seven modules: adaptation (`adapterURLManager`), annotation (`annotationManager`), configuration (`configurationManager`), query and rules (`queryManager`), reflection (`reflectManager`), source code generator (`sourceCodeManager`), and state management (`stateManager`). The following sections briefly describe each module, emphasizing the modules for adaptation of a software entity at runtime (annotation, reflection, source code generator, adaptation).

A. Annotation Module

The annotation module (Figure 3) was developed to assist the software engineer in defining the adaptation level of software entities. It is composed of two packages: (i) `model`, which represents the logics of the module, containing annotations for classes (`ClassAnnotation`), attributes (`AtributteAnnotation`), and methods (`MethodAnnotation`), and (ii) `annotations`, which contains a set of classes that act in the recovery of annotations (`model` package) inserted in software entities in the development stage.

In the `model` package, the `ClassAnnotation` annotation is the "main point" of the module. Basically, it defines two important functionalities: (1) adaptation levels (`enum Artefact`), which can be applied to software entities, and (2) artifacts types (`enum ArtefactType`), which receive the adaptation information.

The adaptation level is defined in the `enum Artefact` (`CLASS`, `FIELDS`, `METHODS`, and `UNADAPTABLE`), which determines how the modifications can be applied. When a software entity is noted as `CLASS`, there must be annotations determining the attributes and/or methods that can be modified. When it is noted as `FIELDS`, only the attributes can be modified (added or removed). In this case, getters and setters methods also suffer the same changes, but they do not need to be noted. When it is noted as `METHODS`, changes occur only in the methods. Finally, when it is noted as `UNADAPTABLE`, no modification in the software entity can be accomplished.

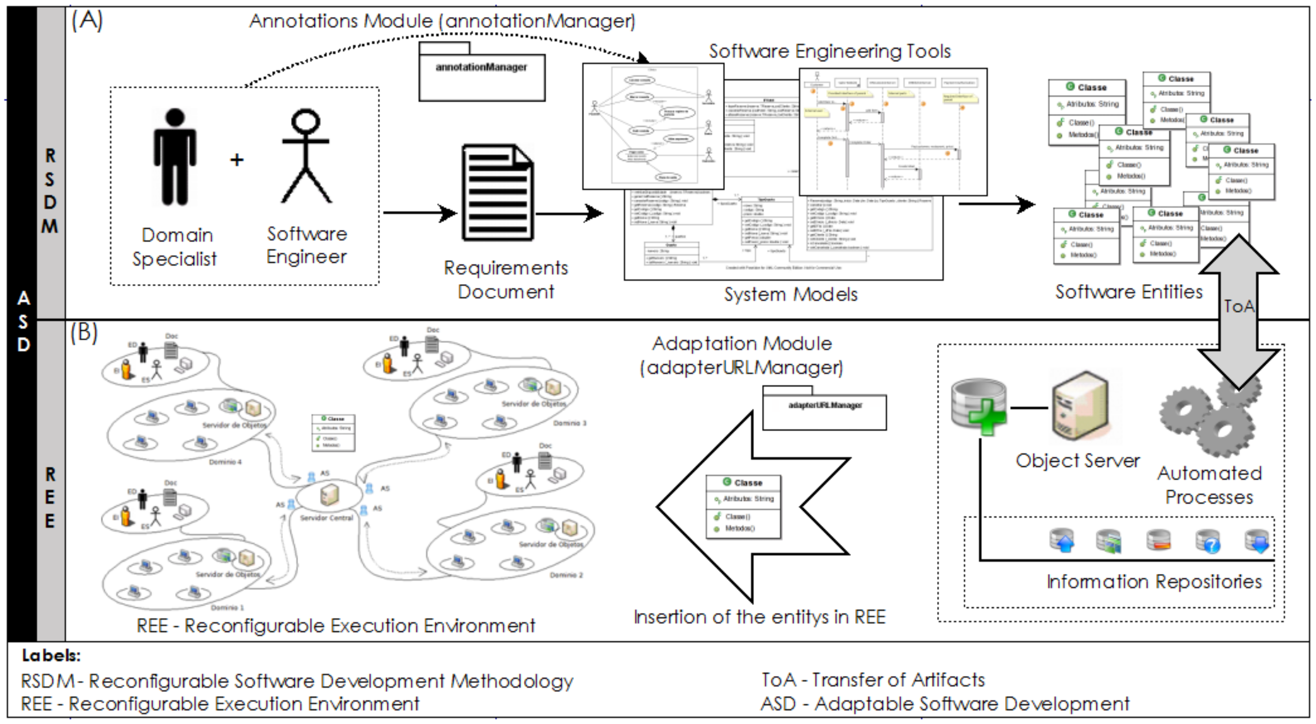

ASD
RSDM
REE
(A)
Annotations Module (annotationManager)
annotationManager
Domain Specialist
+
Software Engineer
Requirements Document
Software Engineering Tools
System Models
Software Entities
ToA
(B)
REE - Reconfigurable Execution Environment
Adaptation Module (adapterURLManager)
adapterURLManager
Insertion of the entitys in REE
Object Server
Automated Processes
Information Repositories
Labels:
RSDM - Reconfigurable Software Development Methodology
REE - Reconfigurable Execution Environment
ToA - Transfer of Artifacts
ASD - Adaptable Software Development

Fig. 1. The Reconfigurable Software Development Process (main phases) [8] [9] [10]

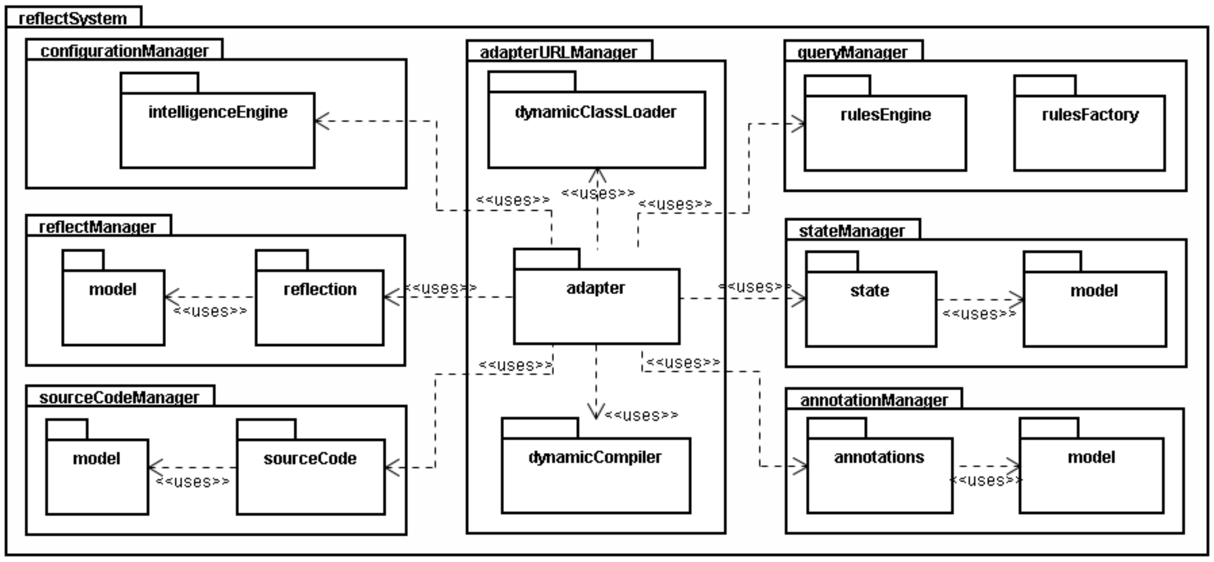


Fig. 2. UML model of reference model [10]

The `enum ArtefactType` (`FUNCIONALITY`, `PERSISTENT`) determines the type of software entity that can be adapted. A `FUNCIONALITY` entity represents a class composed of one or more methods that can be reused and transformed into either a remote method or a web service [8] [9]. A `PERSISTENT` entity represents a system logical class endowed with persistence mechanisms (manual or automated frameworks) into a database.

Apart from the information considered technical (adaptation level and type of software artifact), it is possible to insert semantic information into the software entity. This type of information describes the functionality of the software entity and is used in the information retrieval module (`queryManager`) for a selection of software entities in the REE. Basically, the semantic query uses the information contained in the `semanticInformation` method - implemented in all annotations of `model` package (`annotationManager` module) - to find a specific software entity.

Finally, after inserting the annotations into the software entities, the software engineer must start the validation process, which ensures that the software entities have been noted correctly and can be inserted in the execution environment (REE) waiting for a solicitation by adaptation.

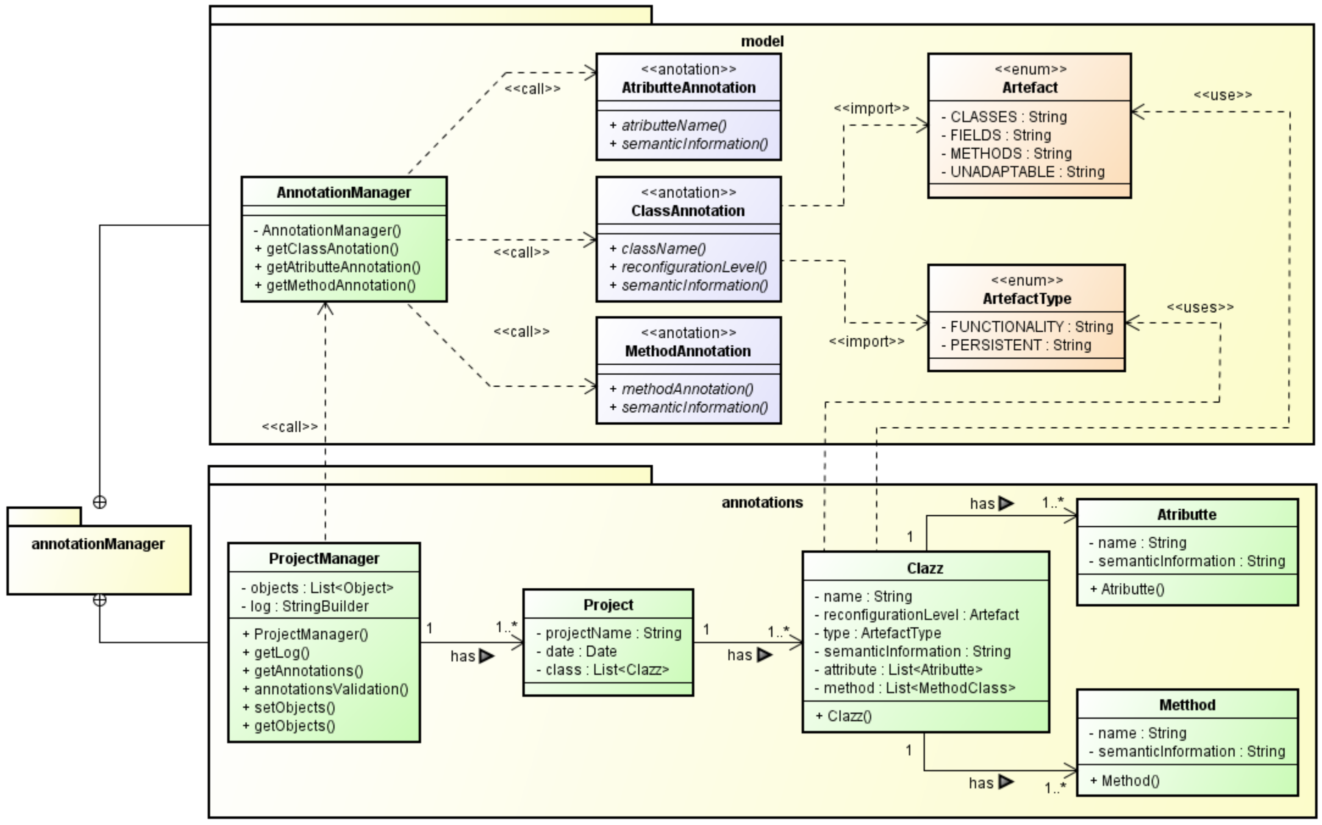


Fig. 3. UML model of annotation module

B. Reflection and Source Code Generator Modules

As mentioned in the previous section (Section IV-A), the annotations are used to identify the adaptation level of software entities, i.e., the modifications that can be applied to these entities at runtime. For this, the following steps are required: (1) entities “disassembly”, (2) inclusion of changes (adding or removing information), (3) generation of a new entity, and (4) reinsertion into the execution environment (REE). Figure 4 shows the reflection module (`reflectManager` package) and the source code generator module (`sourceCodeManager` package) responsible for steps 1 to 3, respectively.

The reflection module (`reflectManager` package) in Figure 4 is composed of `reflection` and `model` packages. The `model` package contains a set of classes that represents a metamodel for a software entity. The `reflection` package contains a set of classes that acts in the handling of this metamodel through the following operations: (i) information retrieval from a software entity at runtime, (ii) instance of the metamodel with the information obtained in step (i); and (iii) transference of this metamodel (step ii) to the source code generator module so that the source code of each new software entity can be generated.

The adaptation process is initiated by the `Project` class in the `model` package. This class has a list of objects (`List <Object>`), which receives all software entities that must be adapted. These entities are the variables of software systems (objects, software components and/or web services) existing in a software project. After instantiating this list, the `Project` class makes a call to `ClassReflect` class (`reflection` package), specifically in the `disassemblingClazz` method, to start the disassembly process of software entities.

The disassembly process retrieves structural and/or behavioral information on software entities and inserts it into a metamodel (`reflectManager.model` package). For this, the Reflection API[3] is used as a computational resource to find information at runtime [2] [3]. Although this metamodel has reflective characteristics, it represents a generic solution. Thus, other computational resources (programming languages) can also be used to instantiate it, provided they meet the requirements mentioned in the Section IV-C.

After the disassembly process has been completed, the responsible developers (software engineer and domain specialist) define the information (structural and/or behavioral) to be inserted to or removed from the software entities so that the source code of a new software entity can be generated. This activity is performed at a high level of abstraction by the manipulation of the metamodel, since the developers define whether attributes (`FieldClass`) and/or methods (`MethodClass`), both class in `reflectManager.model` package, can be added or removed from the software entity.

Figure 4 (bottom part) shows the source code generator

[3] http://docs.oracle.com/javase/tutorial/reflect/

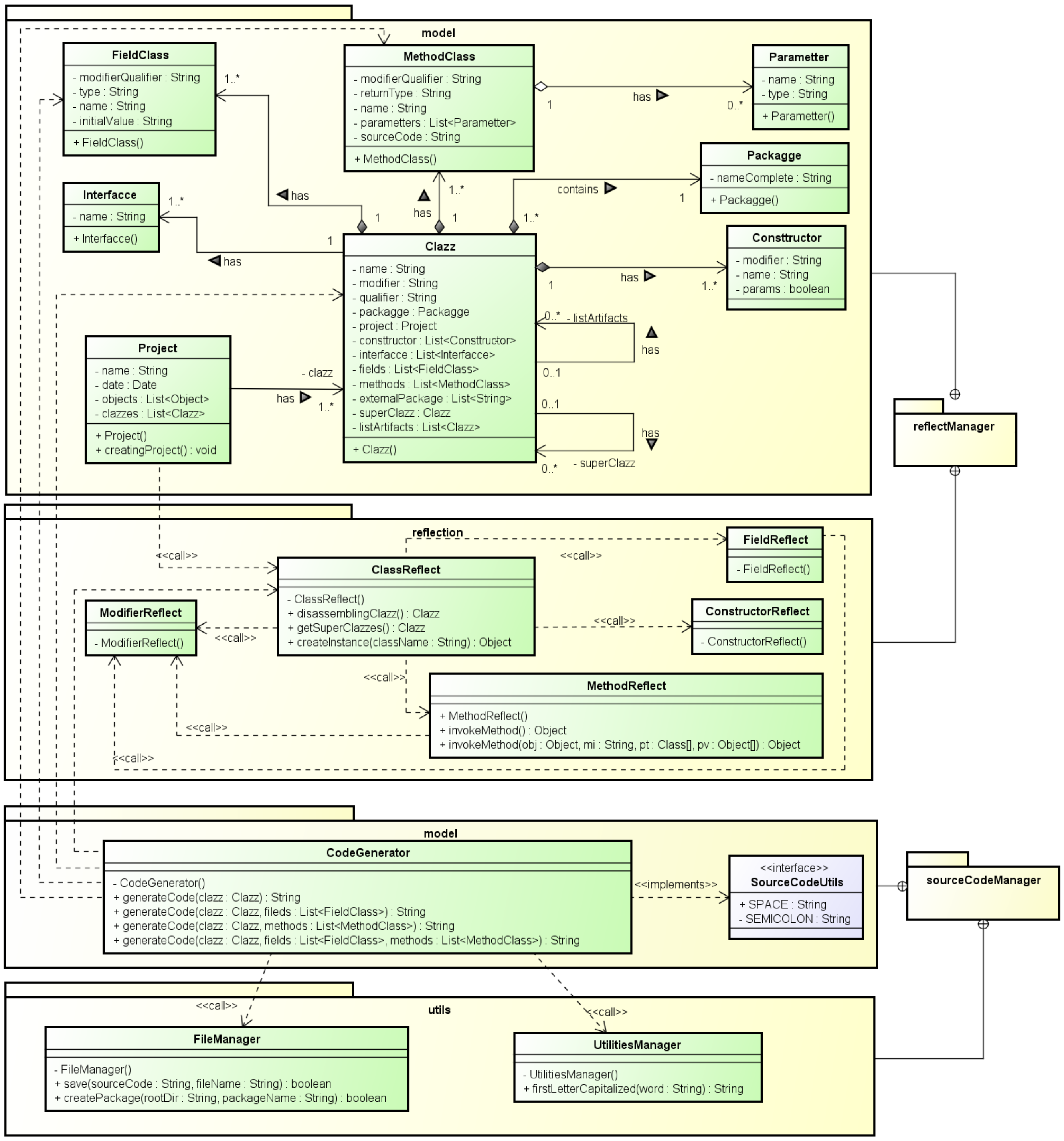


Fig. 4. UML model of reflection and source code generator modules

module (`sourceCodeManager` package), composed of the `model` and `utils` packages. The former contains a set of classes for manipulating files (read and write operations) and the latter contains the `CodeGenerator` class, responsible for generating the source code of software entities. As a preliminary activity, the software engineer must elaborate one or more source code templates to generate the software entities, as previously established by the architectural pattern (Section III). These templates are "injected" directly into the `CodeGenerator` class to generate the source code of each software entity.

The `CodeGenerator` class implements the method, named `generateCode`, which must be overloaded to attend the possible adaptation interests: (i) **structural**, when only one or a list of attributes should be added to or removed from the entity. In this case, the getters and setters methods that manipulate these attributes are modified; (ii) **behavioral**, when only a list of methods should be added to or removed from the entity; and (iii) **structural and behavioral**, when one or a list of attributes/methods should be added to or removed from the entity. In both interests, a `String` containing the new source code of the software entity to be saved by the `FileManager` class (`utils` package) must be returned. Finally, this module has a mechanism for the version control of the entities source code that prevents them from being overlapped and keeps versions of all stakeholders.

C. Adaptation Module

This section presents the adaptation module (`adapterURLManager`) shown in the Figure 5. The module can be considered the "orchestrator" of the reference model presented in this article (Figure 2), since it makes calls and coordinates all activities of the other modules. A "connection point", as a system supervisor (`adapter` package), must be implemented between the dynamic compiler (`dynamicCompiler` package) and the dynamic class loader (`dynamicClassLoader` package). This system aims to compile/recompile the software entities and upload (replace) their binary code at runtime. In addition, a desirable characteristic for this module, specifically in the dynamic compiler, is the ability to diagnose errors in the source code, since the error messages are useful information for the interpretation and corrections in the source code.

The adaptation module (Figure 5) is composed of five packages: `classloaderLanguage`, `compilerLanguage`, `utils`, `compiler` and `adapter`. The former two represent the loaders and compilers of the software entities, respectively. These packages are specific in each programming language, therefore they are not presented in details in this article. The latter three, which are presented in greater details in this section, are responsible for adapting the software entities.

The `utils` package is composed only of the `ClassURL` class, which contains the location (`url`), entity name (`className`), parameters (`paramsType`), and `paramsValue`) of the software entities that must be adapted. The compiler of the software entities is represented by the `CompilerManager` class, which must implement a full compiler for the system adaptation with the following operations: (i) generation of compiled code, and (ii) errors messages (diagnostics) in source code. Finally, it is worth emphasizing the relationship among the classes in the `compilerLanguage` package (compiler of programming languages) with the `CompilerManager` class (compiler of reference model), since it represents the dependency between the model and the computational resource (programming language).

The `adapter` package has only the `RunReload` class, which receives the software entities from the `ClassURL` class, forwards them to be compiled into the `CompileManager` class and loads them into memory with the help of classes in the `classloaderLanguage` package. In operational terms, the `RunReload` class implements the `compileExecute` and `compileInstance` methods. The overload in each of the methods attends the number of software entities (one or a list of `ClassURL` class) to be compiled and instantiated into the environment execution.

The `compileExecute` method compiles and executes the software entity by the method name passed as parameter (`methodName`). In this case, no result is returned to the client program (stakeholders) that has called it, taking the execution by exclusive responsibility of software entity that is being invoked.

The `compileInstance` method compiles the software entity and returns an object (`Object`) to the client program (stakeholders) that has called it. Thus, this client program can make calls of its interest to method of the entity. For this, the software entities that have been invoked must implement an interface to be returned as an instance of this program. Therefore, it is noteworthy that this strategy of access, the client program will have access only the functionalities of software entities and not their content (source code).

D. Others Modules

The configuration module (`configurationManager`) controls the size of software entities when the adaptation is performed. Initially, a software entity is developed to meet specific requirements and act in a specific domain (goals). In short, some changes may occur in the environment and these entities have to adapt to operate in the same domain or different domains. Therefore, a configuration manager enables the control of the number of adaptations (size) and goals of the software entities, maintaining its integrity (goals) over the original domain and not invalidating future adaptations.

The query and rules module (`queryManager`) is responsible for locating software entities in the repositories. When an entity is developed and inserted into the environment execution (repositories), an automatic mechanism (`rulesFactory`) disassembles the entity and creates a set of rules that describes it (structure and behavior). These rules are stored in the repositories and reused when a search (`rulesEngine`) is performed. The rules model (`rulesFactory` package) must be compatible with the software entity metamodel in the `reflectManager.model` package.

The state management module (`stateManager`) preserves the execution state of the software entity. When an entity is selected for adaptation, the information contained in its current state is preserved. The entity is modified and the information is reinserted so that the execution state (current information) is not interrupted. Basically, this module should have two functionalities: (i) the first conversion of an entity into a file (`.xml`) with structural information and its content, and (ii) representation of the reverse process (XML file to entity). The choice of XML (eXtensible Markup Language) to perform these operations is related to the following facilities: files handling (reading and writing), integration with different programming languages, and implementation.

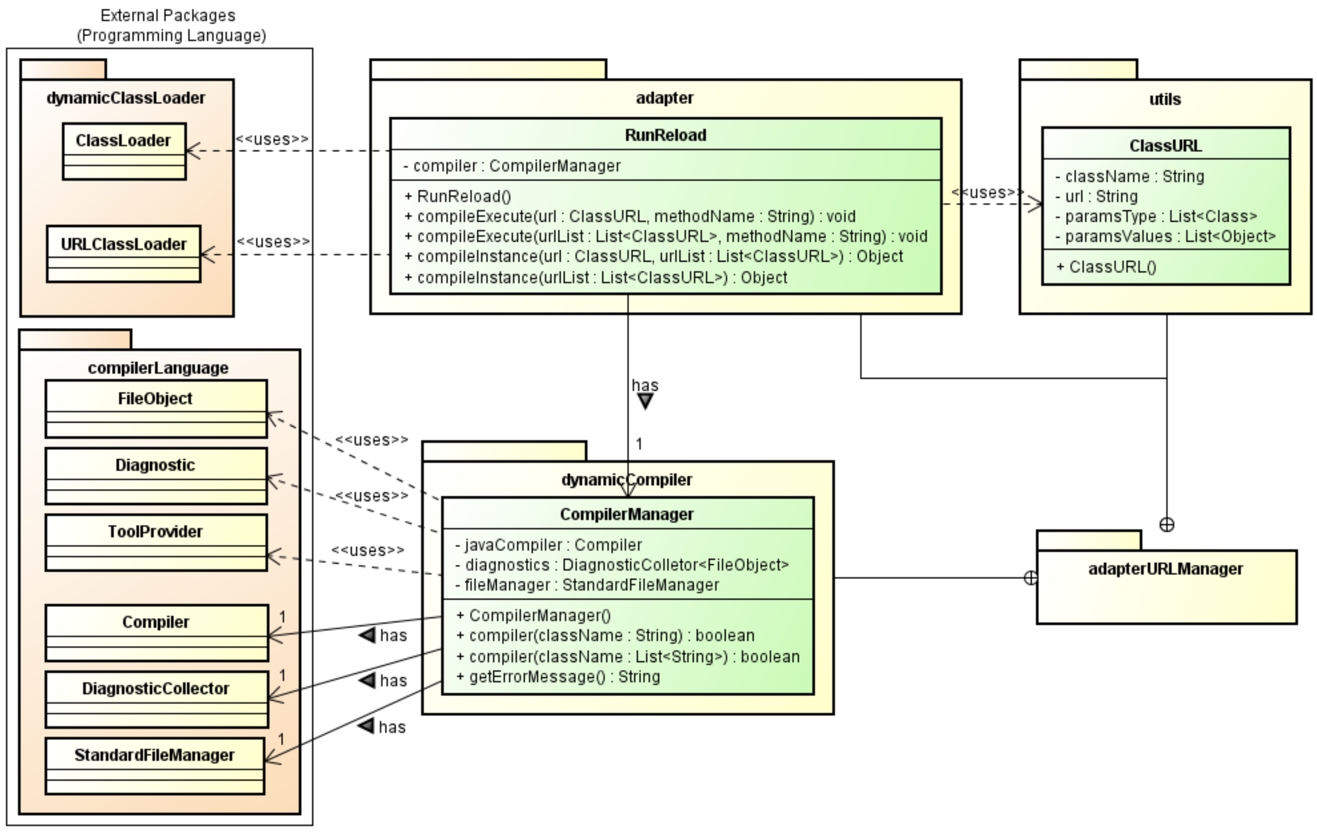

External Packages
(Programming Language)
dynamicClassLoader
ClassLoader
URLClassLoader
compilerLanguage
FileObject
Diagnostic
ToolProvider
Compiler
DiagnosticCollector
StandardFileManager
<<uses>>
adapter
RunReload
- compiler : CompilerManager
+ RunReload()
+ compileExecute(url : ClassURL, methodName : String) : void
+ compileExecute(urlList : List<ClassURL>, methodName : String) : void
+ compileInstance(url : ClassURL, urlList : List<ClassURL>) : Object
+ compileInstance(urlList : List<ClassURL>) : Object
utils
ClassURL
- className : String
- url : String
- paramsType : List<Class>
- paramsValues : List<Object>
+ ClassURL()
has
1
dynamicCompiler
CompilerManager
- javaCompiler : Compiler
- diagnostics : DiagnosticColletor<FileObject>
- fileManager : StandardFileManager
+ CompilerManager()
+ compiler(className : String) : boolean
+ compiler(className : List<String>) : boolean
+ getErrorMessage() : String
adapterURLManager

Fig. 5. UML model of adaptation module

V. Case Study

THIS case study concerns the software adaptation supported by automated process of the reference model (Section IV). To show its applicability, two types of modification will be considered:

1) **Association of new functionalities**, which corresponds to the addition of new information (classes) to the selected software entity by aggregation, composition, or association;
2) **Extension of new functionalities**, which corresponds to the addition of new information (classes) to the selected software entity by the a inheritance relationship.

Before describing the case study, three considerations must be emphasized. The **first** is related to the implementation of the reference model (Section IV), instantiated in Java[4] programming language. For space reasons, the implementation details are not presented in this article. The **second** refers to the size and logic organization of the chosen system, which can be characterized as an information system for the bookstore management. This system has been selected as it can show the use of this reference model (Section IV). The **third** refers to entity adaptation, since the adaptation of software entities can occur in both structural and behavioral changes. In this article, for space reasons, a structural adaptation was chosen, since it can show the applicability of the reference model to the adaptation of a software entity.

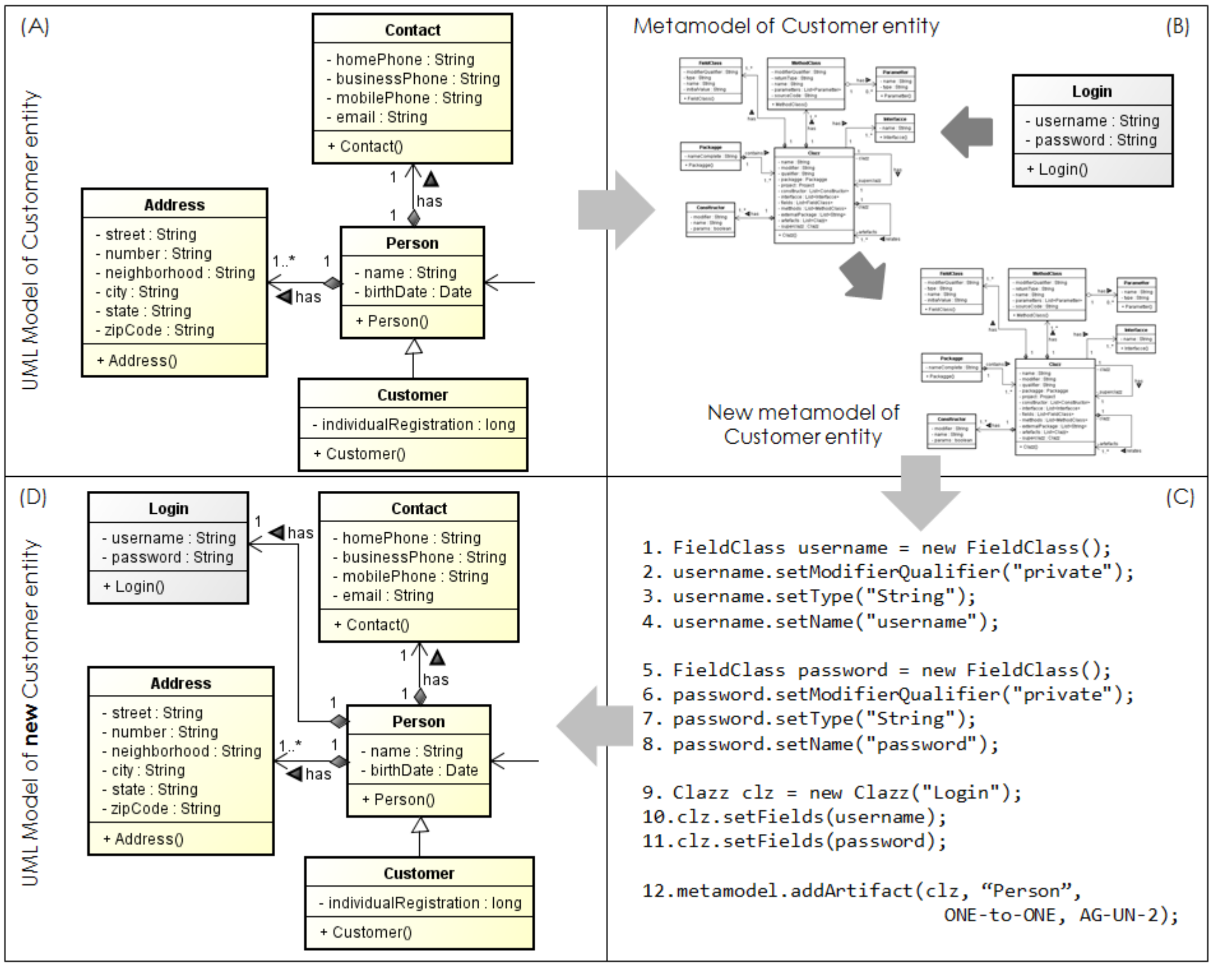
To present the types of modification (1 and 2), only the client software entity of the bookstore system, represented by `Customer` class (Figure 6) is considered. This entity is the base level of the inheritance relationship and having no relationship with others system class (`Address` and `Contact`). Thus, the `Person` class receives the annotations as an adaptive software entity.

Line 1 of Figure 6 shows the `Person` class, i.e. parent class of `Customer`, has an `Artifact.CLASSES` adaptation level. This level requires annotations for the adaptation of attributes (`@AtributteAnnotation`) and methods (`@MethodAnnotation`) - lines 3 and 6, respectively. Still in line 1, the entity was noted as `ArtifactType.PERSISTENT` type, indicating that its information is stored in the database.

To show the first type of adaptation, the `Customer` entity, initially developed to act from a local system for a virtual bookstore system, will be adapted to act in a web system with authentication (same domain). An entity, named `Login`, with `username` and `password` attributes will be created and associated (composition) to the `Customer` entity. Figure 7 shows the adaptation of `Customer` entity.

Square A of Figure 7 shows the UML model of `Customer` entity being "disassembled" and a metamodel in

[4] http://www.oracle.com/technetwork/pt/java/javase/overview/index.html



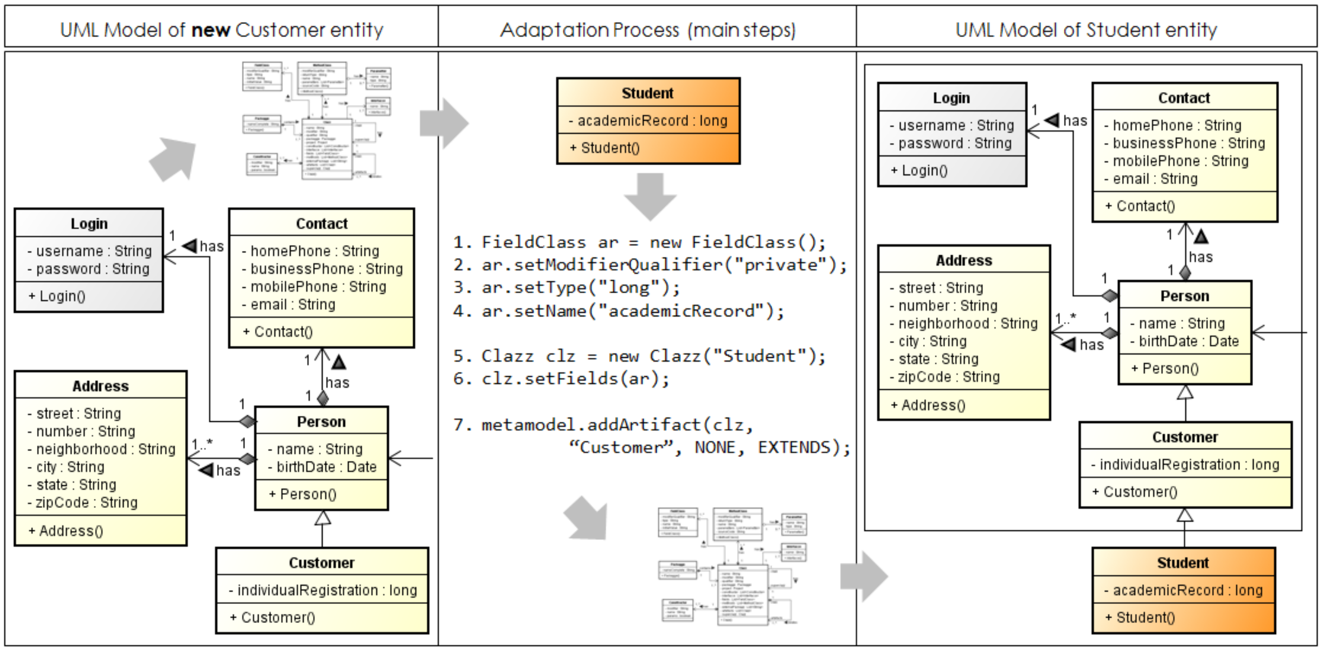
```
1. FieldClass username = new FieldClass();
2. username.setModifierQualifier("private");
3. username.setType("String");
4. username.setName("username");

5. FieldClass password = new FieldClass();
6. password.setModifierQualifier("private");
7. password.setType("String");
8. password.setName("password");

9. Clazz clz = new Clazz("Login");
10.clz.setFields(username);
11.clz.setFields(password);

12.metamodel.addArtifact(clz, “Person”,
                    ONE-to-ONE, AG-UN-2);
```

Fig. 7. Adaptation of `Customer` entity for web system with authentication.

`reflectManager.model` package (Figure 4) is instantiated (Square B) with structural and behavioral information on this entity. For this entity to act in a web system with authentication, two attributes must be added: `username` and `password`. The solution adopted to attend the first type of adaptation was to create a `Login` software entity with these attributes, which will be associated (composition) with the `Customer` entity (lines 1 to 12 - Square C of Figure 7). Initially, the `username` attribute (lines 1 to 4) is created and access modifier, type, and attribute name are defined. The `password` attribute is created in a similar way (lines 5 to 8). The `"Login"` entity is created between lines 9 and 11 (note that the class constructor (`Clazz ("Login")`) defines the model entity name in the UML model). Lines 10 and 11 show the attributes (`username` and `password`) added to the entity (`Login`). Finally, line 12 shows the formation of the composition relationship between the entities (`Person` and `Login`). The first parameter (`clz`) represents the entity that is being created (`Login`). The second (`"Person"`) and third (`ONE-to-ONE`) parameters indicate the entity name (metamodel) that will be associate with the new entity (first parameter) and cardinality between these entities, respectively. The fourth parameter (`AG-UN-2`) indicates the type of relationship (composition) and navigability (unidirectional) of this relationship (`Person` to `Login`). The UML model (Square D) shows the new software entity (`Customer`).

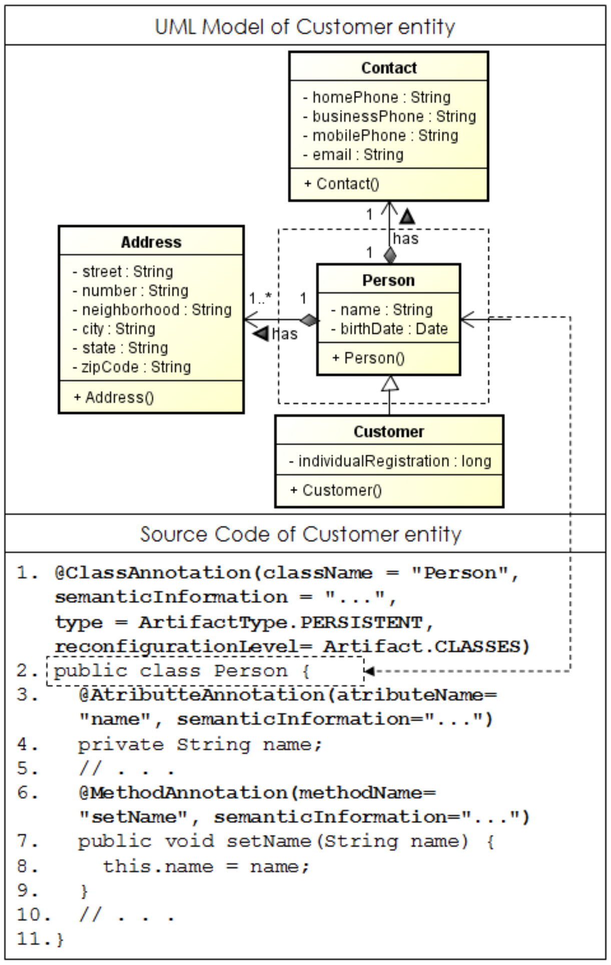
To attend the second type of adaptation, the need to adapt the new `Customer` entity to act in a school management system is considered. Initially this entity was designed to act in a local bookstore system, and subsequently adapted to act in a virtual bookstore system, but maintaining the development context. This type of adaptation requires that the entity operate in a different context from which it was developed. Figure 8 shows the summarized steps of adaptation from `Customer` entity to `Student`.

The adaptation of entities (`Customer` to `Student`) will not be presented at the same detailed level as the previous one (Figure 7), once the functioning and performance of modules are similar. Based on the original entity (First lane of Figure 8), a metamodel was instantiated with structural and behavioral information on the `Customer` entity. For this entity to act in a school management system, a `Student` entity with `academicRecord` attribute must be created (Second lane). The creation of `Student` entity is represented between lines 1 and 7 (Second lane).



```
1. FieldClass ar = new FieldClass();
2. ar.setModifierQualifier("private");
3. ar.setType("long");
4. ar.setName("academicRecord");

5. Clazz clz = new Clazz("Student");
6. clz.setFields(ar);

7. metamodel.addArtifact(clz,
           "Customer", NONE, EXTENDS);
```

Fig. 8. Adaptation of `Customer` entity.



```
@ClassAnnotation(className = "Person",
semanticInformation = "...",
type = ArtifactType.PERSISTENT,
reconfigurationLevel= Artifact.CLASSES)
public class Person {
  @AtributteAnnotation(atributeName=
  "name", semanticInformation="...")
  private String name;
  // . . .
  @MethodAnnotation(methodName=
  "setName", semanticInformation="...")
  public void setName(String name) {
    this.name = name;
  }
  // . . .
}
```

Fig. 6. UML model of adaptation module

Initially, the `academicRecord` attribute is created (lines 1 to 4) and access modifier, type, and attribute name are defined. In lines 5 to 6, this attribute is added to the class `Clazz`. Finally, line 7 shows the formation of the relationship between the entities (`Person` and Student). The first parameter (`clz`) represents the entity that is being created. The second parameter (`"Customer"`) indicates the name of the entity (metamodel) that will be associated with the new entity (first parameter). The third (`NONE`) and fourth (`EXTENDS`) parameters show that there are no cardinality and the type of relationship (inheritance) between these entities, respectively. The third lane shows the UML model of the `Student` entity.

VI. Brief Discussion on the Results and Limitations

THE main results and limitations of this research are addressed in the next sections.

A. *Automated Processes in Software Engineering*

According to these authors [28], [29], [30], software adaptation is an error-prone activity and very onerous when performed manually. Thus, automated processes are important for the development of software systems, since they increase the execution speed of tasks, minimizing time and cost and reducing or eliminating the involvement of developers. Based on this scenario, our reference model represents a comprehensive solution and a good perspective to efficiently contribute to the area of SaS. Software entities are adapted by a set of modules in a well-defined sequence of steps ("assembly line"). Thus, these modules enable the software engineers to act in a higher level of abstraction, avoiding them from having to use specific knowledge so that a software entity to be modified.

B. Reference Model for SaS

Several studies that approach the software adaptation at runtime in different abstraction levels can be found in the literature. As the lowest abstraction level, one can cite the ASM framework[5], which requires from the developers a specific level of knowledge (manipulation in machine code). In our reference model, software entities are adapted in a comprehensive step sequence as assembly line. Basically, the software entity is "disassembled" (retrieved via reflection) and its information is inserted into a metamodel, which is modified and receives new information (structural or behavioral) for the new entity to be generated. Therefore, it can be said that software engineers work in a higher level of abstraction, performing tasks closer to those that are accustomed to do.

As our reference model is based on reflection, it can be adapted to attend other implementation approaches, such as Aspect-Oriented Programming (AOP) [6], [7], [29]. Thus, changes would not affect the structure of the proposed modules of this model.

In this article, the reference model was instantiated in the Java programming language, since it meets the requirements of the model. However, the results have made us believe that this model can be implemented in other programming languages.

C. Limitations

The reference model presented in this article does not cover all cases and does not solve all problems related to software adaptation at runtime. Software adaptation is a broad and complex issue, since there are many involved details so that a software entity is adapted at runtime (specific characteristic of adaptation, techniques of implementation, and others). Although the proposed model can be considered a generic solution, it has not been investigated the its applicability in the development of critical embedded systems [31] [32] [33]. However, the results make us believe that it can be extended or applied in this type of systems, since its guidelines can be adjusted (reused or adapted) to automatically conduct the development of SaS.

In this article, the reference model was instantiated in the Java programming language. It was not still instantiated to other programming languages, intending to evaluate its applicability and behavior in the adaptation of software entities.

Finally, the reference model has not been used in the industry and totally validated. Case studies have been planned in that direction.

VII. Conclusions and Future Works

THIS article presented a reference model to support the development and adaptation of software systems at runtime. Using this model, software entities are transparently monitored and adapted at runtime, without the perception of their users. To perform these operations, the model uses modules that work in an assembly line, i.e., a software entity is automatically disassembled, adapted, and reassembled by the modules in an automated process.

The main contributions of this article are:

1) To the area of SaS with a means that facilitates the development of systems with runtime adaptations;
2) To the area of reference architecture and reference model by proposing a first reference model based on reflection that considers adaptations of software entities at runtime;
3) To the area of software automation, since our reference model presents means of assembly line. The module of this model performs adaptation of software entities without intervention of developers;
4) Finally, it is believed that this reference model may be adequately used together with software development processes that have been used by companies, since both reference model and these processes seem to be complementary.

As future works, some activities are being planned: (i) the first is related to case studies that will be conducted for the evaluation of the reference model presented in this article, since other type of adaptation of software entities must be investigated; (ii) the second is related to the instance of the reference model, other programming languages to evaluate its applicability and behavior in the adaptation of software entity must be tested; (iii) the third is related to the use of the model in the industry, intending to evaluate the behavior of this model when it is applied in larger environments of development and execution; and (iv) the fourth is related to execution performance of the reference model when a software entity is adapted. In the adaptation process of the model, a set of tasks is sequentially executed. The results combined with our experience enable us to identify that some of adaptation task can be executed in parallel. The parallelization of these tasks can considerably improve the adaptation time of the software entities, besides broadening the applicability of this model to the domains of software systems whose time is considered as critical factor. Therefore, we have a positive scenario of research, intending to become this model an effective contribution to the community of software development.

Acknowledgment

The authors would like to acknowledge PROPe/UNESP (Pro-rector of Research / Univ Estadual Paulista) for the financial support provided to this research.

[5] http://asm.ow2.org/

References

[1] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *Computer*, vol. 42, no. 10, pp. 44–51, oct. 2009.

[2] P. Maes, "Concepts and experiments in computational reflection," in *Conference proceedings on Object-oriented programming systems, languages and applications*, ser. OOPSLA '87. New York, NY, USA: ACM, 1987, pp. 147–155.

[3] I. R. Forman and N. Forman, *Java Reflection in Action (In Action series)*. Greenwich, CT, USA: Manning Publications Co., 2004.

[4] G. Coulson, G. Blair, and P. Grace, "On the performance of reflective systems software," in *Performance, Computing, and Communications, 2004 IEEE International Conference on*, 2004, pp. 763 – 769.

[5] Y. Shi, L. ZaoQing, W. JunLi, and W. FuDi, "A reflection mechanism for reusing software architecture," in *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, oct. 2006, pp. 235 –243.

[6] Y. Peng, Y. Shi, J. Xiang-Yang, Y. Jun-Feng, L. Ju-Bo, and Y. Wen-Jie, "A reflective information model for reusing software architecture," in *Computing, Communication, Control, and Management, 2008. CCCM '08. ISECS International Colloquium on*, vol. 1, 2008, pp. 270 –275.

[7] X. Hongzhen and Z. Guosun, "Retracted: Specification and verification of dynamic evolution of software architectures," *Journal of Systems Architecture*, vol. 56, no. 10, pp. 523 – 533, 2010.

[8] F. J. Affonso and E. L. L. Rodrigues, "Estudo comparativo da adaptação de software utilizando chamada de métodos remotos e serviços web," *Revista de Sistemas de Informação da FSMA*, vol. 7, no. 1, pp. 22–31, june 2011.

[9] ——, "Reflecttools: Uma ferramenta de apoio ao desenvolvimento de software reconfigurável," *Revista Brasileira de Computação Aplicada*, vol. 3, no. 2, pp. 73–90, 2011.

[10] ——, "A proposal of reference architecture for the reconfigurable software development," in *The 24th International Conference on Software Engineering and Knowledge Engineering*, 2012.

[11] N. Bencomo and G. Blair, "Using architecture models to support the generation and operation of component-based adaptive systems," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems, pp. 183–200.

[12] J. Estublier, G. Vega, P. Lalanda, and T. Leveque, "Domain specific engineering environments," in *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, dec. 2008, pp. 553 –560.

[13] E. Tanter, R. Toledo, G. Pothier, and J. Noyé, "Flexible metaprogramming and aop in java," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 22–30, Jun. 2008.

[14] A. Janik and K. Zielinski, "Aaop-based dynamically reconfigurable monitoring system," *Inf. Softw. Technol.*, vol. 52, no. 4, pp. 380–396, Apr. 2010.

[15] E. Borde, G. Haïk, and L. Pautet, "Mode-based reconfiguration of critical software component architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 1160–1165.

[16] H. Gomaa and M. Hussein, "Software reconfiguration patterns for dynamic evolution of software architectures," in *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, june 2004, pp. 79 – 88.

[17] E. Kasten, P. McKinley, S. Sadjadi, and R. Stirewalt, "Separating introspection and intercession to support metamorphic distributed systems," in *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, 2002, pp. 465 – 472.

[18] G. Bastide, A. Seriai, and M. Oussalah, "Dynamic adaptation of software component structures," in *IEEE International Conference on Information Reuse and Integration, 2006*, sept. 2006, pp. 404 –409.

[19] J. A. Kim, O.-C. Kwon, J. Lee, and G.-S. Shin, "Component adaptation using adaptation pattern components," in *IEEE International Conference on Systems, Man, and Cybernetics, 2001*, vol. 2, 2001, pp. 1025 –1029 vol.2.

[20] J. Whitehead, "Collaboration in software engineering: A roadmap," in *2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 214–225.

[21] E. Y. Nakagawa and J. C. Maldonado, "Requisitos arquiteturais como base para a qualidade de ambientes de engenharia de software," *IEEE Latin America Transactions*, vol. 6, no. 3, July 2008.

[22] E. Y. Nakagawa, F. C. Ferrari, M. M. Sasaki, and J. C. Maldonado, "An aspect-oriented reference architecture for software engineering environments," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1670 – 1684, 2011.

[23] J. Dong, J. Wang, D. Sun, and H. Lu, "The research of software product line engineering process and its integrated development environment model," in *Computer Science and Computational Technology, 2008. ISCSCT '08. International Symposium on*, vol. 1, dec. 2008, pp. 66 –71.

[24] K. Henttonen and M. Matinlassi, "Open source based tools for sharing and reuse of software architectural knowledge," in *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, sept. 2009, pp. 41 –50.

[25] J. Grundy, "Software engineering tools," in *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, 2001, pp. 3914–3914.

[26] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information and Software Technology*, vol. 52, no. 1, pp. 31 – 51, 2010.

[27] S. Angelov, P. Grefen, and D. Greefhorst, "A framework for analysis and design of software reference architectures," *Information and Software Technology*, vol. 54, no. 4, pp. 417 – 431, 2012.

[28] S. Vinoski, "A time for reflection [software reflection]," *Internet Computing, IEEE*, vol. 9, no. 1, pp. 86 – 89, jan.-feb. 2005.

[29] J. Andersson, R. de Lemos, S. Malek, and D. Weyns, "Reflecting on self-adaptive software systems," in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*, may 2009, pp. 38 –47.

[30] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, May 2009.

[31] H. Gill, "Challenges for critical embedded systems," in *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, 2005, pp. 7–9.

[32] L. Dobrica and E. Niemelä, "An approach to reference architecture design for different domains of embedded systems." in *Software Engineering Research and Practice*, H. R. Arabnia and H. Reza, Eds. CSREA Press, 2008, pp. 287–293.

[33] V. Januzaj, S. Kugele, B. Langer, C. Schallhart, and H. Veith, "New challenges in the development of critical embedded systems - an "aeromotive" perspective," in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2010, vol. 6415, pp. 1–2.

Frank José Affonso has a bachelor's degree in Computer Science from UNICEP (Central Paulista University), Master's degree in Computer Science from DC/UFSCAR (Department of Computation-Federal University of São Carlos) and Ph.D. in Electrical Engineering from DEEC/ESSC/USP (Department of Electrical Engineering and Computation - Engineering School of São Carlos - University of São Paulo). He is an assistant professor at UNESP (Univ Estadual Paulista). He has experience in computer science, with emphasis on Software Engineering and JAVA Programming Language. He has worked on the following research lines: Software Architecture, Dynamic Software Architecture, and Self-adaptive Systems.

Maria Cecília Vecchiato Saenz Carneiro has a bachelor's degree in Mathematics from UNESP (Univ Estadual Paulista), Master's degree in Computer Science and Computational Mathematics from USP (University of São Paulo) and Ph.D. in Environmental Engineering Science from UNESP (Univ Estadual Paulista). She is an assistant professor at UNESP (Univ Estadual Paulista). She has experience in computer science, with emphasis on Software Engineering, performing on the following topics: methodologies of systems development, UML modelling language, architecture and architectural software design, support systems of decision aimed to public politics.

Evandro Luís Linhari Rodrigues has a degree in Electrical Engineering from ESL (Engineering School of Lins), Master's degree in Electrical Engineering from DEEC/ESSC/USP (Department of Electrical Engineering and Computation - Engineering School of São Carlos - University of São Paulo) and PhD in physics from PI/USP (Physics Institute - University of São Paulo). He is a professor at the University of São Paulo. He has experience in Electrical Engineering with emphasis on Electronic Process Automation and Industrial Electrical. He has worked on the following research topic: image processing, microprocessors/microcontrollers, computer vision, carpal analysis and automation.

Elisa Yumi Nakagawa is Associate Professor in the Department of Computer Systems at University of São Paulo (USP), Brazil. She received her BS degree in Computer Science in 1995 from the University of São Paulo, her M.S. degree in 1998 and her Ph.D. degree in 2006 in Computer Science and Computational Mathematics from the University of São Paulo. She conducted her Post-Doctoral in 2011-2012 in the Fraunhofer Institute for Experimental Software Engineering (IESE), Germany. Her main research interests include software architecture, reference architecture, systems of systems, software product line, and open source.