A Systematic Process for Domain Engineering

Eduardo Santana de Almeida¹, Alexandre Alvaro¹, Vinicius Cardoso Garcia¹, Daniel Lucredio², Renata Pontin de Mattos Fortes², Silvio Romero de Lemos Meira¹

¹Federal University of Pernambuco, C.E.S.A.R. - Recife Center for Advanced Studies and Systems, Pernambuco, Recife, Brazil
{esa2, aa2, vcg, srlm}@cin.ufpe.br
²University of São Paulo
{lucredio, renata}@icmc.usp.br

Abstract: Software reuse is a key aspect for improving quality and productivity. However, this process is more effective when systematically planned and managed in the context of a specific domain, where application families share some functionality. In this scenario, Domain Engineering (DE) has been seen as a facilitator to obtain the desired benefits. Nevertheless, the existing domain engineering processes present crucial problems, such as lacking of details in the three basic steps of domain engineering and not being systematic. Thus, this paper aims at defining a systematic process to perform domain engineering based on the-state-of-the-art in the area, which includes the steps of domain analysis, domain design, and domain implementation. An experimental study evaluates the viability of the use of the process and the impact of applying it to a domain engineering project.

1 Introduction

In the context of software reuse, important research including company reports [1], [2], [3], [4], informal research [5], [6] and empirical studies [7], [8], [9] have highlighted the relevance of a reuse process, once the most common way of software reuse involves developing applications reusing pre-defined assets.

However, the existing reuse processes present crucial problems [10], such as: they do not cover the three steps of domain engineering: domain analysis, design and implementation; besides, they do not define activities, sub-activities, roles, inputs, outputs of each step in a systematic way.

Thus, this paper presents a systematic software reuse process to perform domain engineering, which includes the steps of domain analysis, domain design, and domain implementation, based on a set of activities, sub-activities, inputs, outputs, principles, guidelines, and roles.

2 The Domain Engineering Process

Domain engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable work products), as well as providing an adequate means for reusing these assets (e.g. retrieval, qualification, dissemination, adaptation, assembly) when building new systems [11].

A domain engineering process should define three important steps: Domain Analysis (DA), Domain Design (DD), and Domain Implementation (DI). In general, the main goal of Domain Analysis is domain scoping and defining a set of reusable, configurable requirements for the systems in the domain. Next, Domain Design develops a common architecture for the system in the domain and devising a product plan. Finally, Domain Implementation implements the reusable assets, for example, reusable components, domain-specific languages, generators, and a production process [11].

The next sections presents each step in details.

2.1 The Domain Analysis Step

The term domain analysis was first introduced by Neighbors [12] as “the activity of identifying the objects and operations of a class of similar systems in a particular problem domain.” However, neither Neighbors nor many other works [13], [14] address the issue of “how to perform” domain analysis. These works focus on the outcome, not on the process, and success stories are more the exception than the rule.

Typically, knowledge of a domain evolves over time until enough experience has been accumulated and several systems have been implemented, so generic abstractions can be isolated and reused. In this context, our goal, in concordance with Prieto-Diaz [14], (pg. 48) is: “to find ways to extract, organize, represent, manipulate and understand reusable information, to formalize the domain analysis process, and to develop technologies and tools to support it.”
The approach for domain analysis has three activities: Plan Domain, Model Domain and Validate Domain.

2.1.1 Plan Domain

The first activity in the approach corresponds to a preparation phase, to determine whether it makes good sense to invest in building a reuse infrastructure. The domain analyst collects the initial information for the subsequent steps, including: identification of the stakeholders, definition of the objectives and constraints, and market analysis. The domain analyst also collects all knowledge regarding the domain, including available documentation and existing applications.

Next, the applications to be supported by the domain are identified, including their exact scope, which features these applications support individually, and the determination of candidates for domain features. The domain analyst, together with the domain experts, develops a list of the applications of the domain, including: existing applications (i.e., applications that have been developed prior to the start of the domain analysis process), future applications (i.e., applications where the requirements are rather clear, but development has not yet started) and potential applications (i.e., applications for which no clear requirements exist yet, but that are seen as relevant). The list of applications also includes a list of features [15] for those applications, identified through the analysis of the applications, their documentation, and the knowledge from the domain expert.

2.1.2 Model Domain

This activity shifts attention from scoping issues to structural issues and conceptual elements within the domain. Thus, a model is developed, describing the commonality and variability within the domain. Rather than building a model for a single application in the domain, or a generic model that may be applicable at a high level to a number of applications, the domain modeling task attempts to formalize the space of variations for individual applications in the domain.

In this approach, the domain model is represented through feature models [15]. The domain analyst groups the features that were identified in the previous step in a features model, using some useful guidelines [16].

2.1.3 Validate Domain

Domain validation is achieved in five sub-activities:

- **A1. Document features.** Our approach uses the template defined by Czarnecki & Eisenecker [11]. In this template each feature consists of: Semantic description; Rationale; Stakeholders and client programs; Example applications; Constraints; Priorities; and Open/closed points in the domain;

- **A2. Check for homonyms.** As a complementary sub-activity to the search for synonyms, it is necessary to search for homonyms, i.e., the same literal term used with different meanings in different contexts;

- **A3. Model Validation.** This sub-activity corresponds to the matching of the requirements that were expressed by stakeholders and the domain model, in order to validate its completeness and accuracy; and

- **A4. Document the domain.** In order to document the domain, the meta-model defined in [17] is used, consisting of the following information: Domain description; Domain defining rules; Exemplar system selection; Documentation; Domain Context Relationship; Domain genealogy; and Feature.

2.2 The Domain Design Step

The design step consists of seven activities, presented in the next sections. The approach is influenced by several works from the literature, such as ADD [18], UML Components [19], and the weak and strong points from reuse processes [10].

2.2.1 Module decomposition

The first step in the approach corresponds to an abstraction and decomposition phase. Initially, the domain architect chooses which domain architecture modules to decompose, usually starting with complete domain applications, which are further decomposed into subsystems and submodules.

In our approach, we still do not have a set of criteria to be used in module decomposition, as in Parnas' work [20], nor a set of rule of thumbs. However, we consider that the following issues should be balanced: availability, coupling, extensibility, flexibility, functionality, information hiding, maintainability, modifiability, performance, separation of concerns, scalability, security, and usability.

2.2.2 Module refinement

The module refinement is an iterative process that can be divided into three activities:

- **Choose the architectural drivers.** According to Bass et al. [18], architectural drivers are the combination of functional and quality requirements that “shape” the architecture or the particular module under consideration. The drivers are found among the top-priority requirements for the module. We base the module decomposition on the architectural drivers, to reduce the problem of satisfying the most important ones. In our approach, the drivers are the requirements expressed by feature model, the quality attributes and the scenarios.

- **Choose the architectural patterns.** Here, the domain architect selects the architectural patterns that can be applied. The patterns satisfy the architectural drivers and are constructed by composing selected tactics. Two factors guide tactic selection. The first are the drivers themselves, and the second are the side effects that a pattern
implementing a tactic has on others. Our vision of a tactic agrees with Bass et al.’s, who define it as a design decision that influences the control of a quality attribute response.

Allocate functionality using views. In this activity, the goal is to define how modules can be instantiated. The criteria are similar to those used in functionality-based design methods, such as most object-oriented design methods, but with a variation to treat features. Two approaches are proposed: i) to allocate functionality based on use cases, and ii) to allocate functionality based on features.

2.2.3 Variability representation

Variability is the ability to change or customize a system [21]. However, even with several approaches available in the literature, software architects do not have effective ways to do it [22]. In our approach, we propose the use of Design Patterns [23] to solve this problem, as already used by other works [24], [25]. Differently from these works, however, we provide guidelines for how and why each pattern should be used in each situation.

In order to design the variability of each module, we consider that it should be traceable from domain analysis assets (features) to the architecture, according to alternative, or and optional features [25]. Depending on the kind of association between features, different design patterns can be used. Each design pattern provides a different option for the designer, which makes the decision on which patterns to use according to some guidelines. For example, for alternative features, when a feature can be directly mapped into a single class, we suggest the Prototype [23], because it is simpler and allows to instantiate a specific object, depending on the feature that is used in the product, through simple inheritance. Another suggestion is to use the Singleton [24] pattern to keep and manage a unique instance of this class. More guidelines, for other possible situations, may be seen in [26].

2.2.4 Component grouping

This step is composed of four activities:

Measure functional dependency. The domain analyst determines the relationships between the use cases, using different metrics [26];

Cluster use cases. The domain architect defines candidate components by clustering related use cases. The clustering algorithm used for this task uses a row and column shifting method.

Allocate classes to components. Here, the domain architect locates sequence diagrams for use cases included in each component identified in the previous activity. Next, the classes participating in these sequence diagrams are assigned to the corresponding component.

Select candidate components. The domain architect, in conjunction with the project manager, identifies candidate components. The value of $t$ used in the process defines the number of components and their granularity. Thus, it is recommended to apply different values of $t$ to generate different clustering results and to let architects and project managers choose an optimal clustering result using the criteria. Additionally, costs and complexity can be used to select the candidate components.

2.2.5 Component identification and specification

In this step, the goal is to refine the components, including their system and business interfaces, and the core classes. For each use case, the domain analyst considers whether or not there are system responsibilities that must be modeled. If so, they are represented as one or more operations of the interfaces (just signatures). This gives an initial set of interfaces and operations.

The business interfaces are abstractions of the information that must be managed by components. Our process for identifying them is the following: to analyze the feature model to identify classes (for each module and component); to represent the classes based on features with attributes and multiplicity; and to refine the business rules using formal language.

After identifying the interfaces, the domain architect decides which classes from each module are in the core [19]. A core class is a business type that has independent existence within the business. The purpose of identifying core classes is to start defining which information is dependent on others, and which information can stand alone. The general rule is that we create one business interface for each core class, to manage the information represented by the core class.

2.2.6 Domain architecture representation

Once the component specification is performed, the domain architect represents the initial domain architecture based on components. Architectural views and component diagrams are used to show the components, their interconnection, and the provided and required interfaces. During this step, the domain architect can discover and refine other components, using, for example, collaboration diagrams.

2.3 The Domain Implementation Step

In our approach, we decided to use OSGi [27] to implement the components and manage their interaction and lifecycle, due to its applicability in many different scenarios, and the possibility of being used together with other technologies. This step consists of two activities: component implementation and component documentation.

2.3.1 Component implementation

In this step, the software engineer, based on requirements, implements the software components through two sets of activities, each one with a different purpose. Activities 1 to 4 deal with the provided services, and activities 5 to 7 deal with required services.
Activity 1. The first activity is to describe the component, providing general-purpose information, such as the vendor, version, package, among others.

Activity 2. In this second activity, the software engineer should specify the provided services. Artifacts developed in domain analysis and design may be reused in this activity.

Activity 3. In the third activity, the goal is to implement the provided services, as well as the code to register these services to be used by other components.

Activity 4. In this activity, which concludes the provided side of the component, the goal is to build and install the component. This involves compiling and packaging the component in a form that is suitable to be deployed.

Activity 5. In order to reuse some component, the software engineer needs to describe the component that will reuse other services. This is similar to Activity 1, but with the focus on the services that are required.

Activity 6. In this activity, the software engineer should implement the connection between the required services with the rest of the code. Here, different techniques can be employed, such as the use of adapters and wrappers, for example.

Activity 7. The last activity corresponds to building and installing the component that reuses the services, which is similar to Activity 4.

2.3.2 Component documentation

Most work related to component documentation [28, 29, 30, 31] are pattern-based approaches. Our step for component documentation has two important differences: i. it is based on previous works (pattern-based approaches including weak and strong points) and real world experience; ii. it follows some Principles for component documentation: use of hypertext; embed content in source code; automation; leverage programming languages semantics; use of diagrams and figures.

Thus, in order to document the components, a template composed of five sections is used: Basic Information, Detailed Information, Quality Information, Deploy Information, and Support Information. More information about the domain implementation phase can be seen in a previous work [32].

3 The Experimental Study

In order to determine whether the process meets its proposed goals, an experimental study was performed. The plan of the experiment to be presented follows the model proposed in [33], and uses the future tense, symbolizing the precedence of the plan in relation to its execution.

3.1 The Definition

Goal. To analyze the domain engineering process for the purpose of evaluating it with respect to the efficiency and difficulties of its use from the point of view of researcher in the context of domain engineering projects.

3.2 The Planning

Context. The domain engineering project will be conducted in a university laboratory with the requirements defined by the experimental staff based on real-world projects. The study will be conducted as single object study which is characterized as being a study which examines an object on a single team and a single project. All the subjects will be trained to use the process and will receive two questionnaires to provide their information (QT1) and their impression on the process (QT2).

Criteria. The benefits obtained will be evaluated quantitatively through the domain architecture and components, using the instability (I) [34], maintainability (MI) [35], and complexity (CC) [36] metrics. We decided to use classic Object Orientation metrics to evaluate the process because they are more well-established after years of experience with case studies and experiments. Reuse-specific metrics, although more suited to this context, still need more experimentation and use [37]. Besides, issues such maintainability, stability and complexity have a large influence on the architecture. For example, if a component has low maintainability, it will be probably harder to reuse. So, by measuring these aspects, we are, to some extent, measuring reuse. Moreover, the difficulties will also be evaluated using qualitative data from the questionnaires.

Null Hypothesis. This is the hypothesis that the experimenter wants to reject. In this study, it determines that the use of the process does not produce benefits that justify its use and that the subjects have difficulties to apply it:

\[ H_0: \mu \text{the process generates the architecture with } I < 0.5, \text{MI} < 85, \text{CC} > 21. \]

The values for I, MI and CC were obtained from the literature [34, 35, 36].

Alternative Hypothesis. This is the hypothesis in favor of which the null hypothesis is rejected. In this study, the alternative hypothesis determines that the use of the process produces benefits that justify its use. Thus, the following hypothesis can be defined:

\[ H_1: \mu \text{the process generates the architecture with } I < 0.5, \text{MI} > 85, \text{CC} < 21 \]

Quantitative analysis. In this study, descriptive statistics will be used to analyze the data set [33].

Qualitative Analysis. The qualitative analysis aims to evaluate the difficulty of the application of the proposed process and the quality of the material used in the study. This analysis will be performed through questionnaire QT2.

Internal Validity. The internal validity of the study is dependent of the number of subjects. This study is supposed to have at least between seven and eight subjects to guarantee a good internal validity.

External Validity. A possible problem related to the external validity is the subjects’ motivation, since some subjects can perform the study without responsibility or without a real interest in performing the project with a good quality as it could happen in an industrial project. This will be assessed through questionnaire QT2.
Construct Validity. In this study, a relatively well known and easily understandable problem domain was chosen to prevent the experienced users in a certain domain to make use of it.

Conclusion Validity. This validity is concerned with the relationship between the treatment and the outcome, and determines the capability of the study to generate conclusions [33]. This conclusion will be drawn by the use of descriptive statistic.

3.3 The Project used in the Study
The project used in the experimental study was the domain engineering of the starship game domain. Three games in this domain were presented to the subjects, who had just the executables without any documentation. After performing the domain engineering, the subjects were asked to implement one application reusing the developed assets.

3.4 The Operation
Experimental Environment. The experimental study was conducted during part of a M.Sc. and Ph.D. Course in Software Reuse, during April-September 2006, at Federal University of Pernambuco. The experiment was composed of seven subjects and all the project was developed in 355 hours, 23 minutes and 57 seconds. In this project, 44 features, 33 packages, and 79 classes were created. Additionally, 5 components and 1 example application were also developed, totaling 3638 lines of code.

Training. The subjects who used the proposed process were trained before the study began. The training took 28 hours, divided into 14 lectures with two hours each, during the course.

Subjects. The subjects were 7 MS.c. students selected by convenience sampling [33]. All the subjects had industrial experience in software development (more than one year). Three subjects had participated in industrial projects involving some kind of reuse activity, for instance, component development, framework development, or web services development. All the subjects known at least one domain analysis process (FODA); three subjects had training in conferences on some issues related to software reuse, such as design patterns and component-based development; and finally, two subjects had co-authored papers involving some aspects of software reuse.

3.5 The Analysis
Quantitative Analysis. The quantitative analysis was divided in four analyses: instability and maintainability for the architecture, complexity for the components, and the difficulties found in the analysis, design, and implementation steps. The analyses were performed using descriptive statistics. Table 1 shows the summary of the analysis.

As it can be seen, the mean values for all metrics reject the null hypothesis. Also, except for the instability metric, the maximum and minimum values still reject the null hypothesis. Thus, the results indicate that the alternative hypothesis may be true, i.e. the method helps in producing components with low complexity and high maintainability.

Qualitative Analysis. After concluding the quantitative analysis, the qualitative analysis was performed. This analysis is based on the answers defined for the QT2.

Usefulness of the Process. All the subjects reported that the process was useful to perform the domain engineering project. However, four subjects indicated some improvements in domain analysis; for five subjects, some aspects in design should be reviewed; and, finally, six subjects discussed some improvements in domain implementation.

More details about the experimental evaluation may be seen in a previous work [38].

4 Related Works
In a previous work [10], eleven software reuse processes based on domain engineering (DRACO, ROSE, ODM, RSEB, FeatuRSEB, FORM) and software product lines (PuLSE, KobrA, CoPAM, PECOS, FORM’s extension) are discussed, corresponding to the state-of-the-art in the area. This study shows that the processes present crucial problems such as: they do not cover the steps of domain engineering: domain analysis, design and implementation; besides, they do not define activities, sub-activities, roles, inputs, outputs of each step in a systematic way.

5 Conclusion and Future Works
Domain Engineering is a key requirement in a reuse process. However, the available reuse processes do not cover the three basic steps of domain engineering - domain analysis, domain design, and domain implementation - and neither define activities, sub-activities, roles, inputs, outputs of each step in a systematic way.
In this sense, in order to solve the problems identified in the available reuse process, this paper presented a systematic process for domain engineering, which defines a systematic way to perform it based on a set of principles, guidelines, inputs, outputs, and roles. The process is based on an extensive review of the software reuse processes, involving their weak and strong points. Additionally, an experimental study evaluated the viability of the use of the process and the impact of applying it to a domain engineering project. As future work, we are planning to improve the process with the results obtained in the experimental study and replicate it in different contexts. More information about the domain analysis, design and implementations steps can be found in [39],[26],[40][41].

Acknowledgments

This work was developed with the financial support from CAPES and CNPq (process number: 475743/2007-5). The authors would also like to thank the people who contributed with insights to this work with discussions in worldofreuse.blogspot.com.

References

[23] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1995.