Domain Implementation in Software Product Lines Using OSGi

Eduardo S. Almeida, Eduardo C. R. Santos, Alexandre Alvaro, Vinicius C. Garcia, Silvio Lemos Meira
Federal University of Pernambuco, Recife Center for Advanced Studies and Systems (C.E.S.A.R.), Recife, PE, Brazil
{esa2,ecs,aa2,vcg,srlm}@cin.ufpe.br

Daniel Lucrédio, Renata Pontin de Mattos Fortes
University of São Paulo, São Carlos, SP, Brazil
{lucredio,renata}@icmc.usp.br

Abstract

Software product lines (SPL) are a powerful way of ensuring quality, economic efficiency, and manageability of software system families. In SPL, a key aspect is the domain implementation, whose goal is to provide the implementation of reusable assets (components). However, current approaches present some gaps in this direction, such as the lack of definition on how to implement and document software components in a systematic way. In this context, this paper presents a method for domain implementation in software product lines. The method is based on a well defined set of guidelines, inputs, outputs, and roles, and uses OSGi as the main implementation technology. An experimental study evaluates the viability of the use of the method and the impact of applying it to a software development project.

1. Introduction

A software product line is a set of software-intensive systems that share a common, managed feature set, satisfying a particular market segment’s specific needs or mission and that are developed from a common set of core assets in a prescribed way [37]. In general, there are three essential activities: core asset development, product development, and management [37]. The core asset development can be understood as Domain Engineering and product development as Application Engineering. In this paper, we will treat the core asset development activity as Domain Engineering, as previously defined by reuse community [22].

Domain Engineering (DE) is the activity of collecting, organizing, and storing past experience in building systems or parts of systems from a particular domain in the form of reusable assets [18]. The DE life cycle encompasses three steps: Domain Analysis, Domain Design, and Domain Implementation. The literature has several contributions for the Domain Analysis [7, 43, 27, 23, 10, 28, 35, 36, 5, 3] and Domain Design [3, 12, 9, 49] fields, but the Domain Implementation area presents some gaps that need to be closed to allow its use in large-scale.

The key goal of the domain implementation is to provide the implementation of reusable assets (components) [42]. However, even this idea dating back to McIlroy’s work [34], there are still some gaps in reuse processes [3] and specific methods and approaches for component-based development [21, 46, 30, 11, 25, 15, 17, 20, 19, 16]. The main problem is that these works do not define how to implement and document software components in a systematic way. Perhaps the reason for this lack of details is that implementation depends so much on the used technology. Although some of the implementation challenges are the same no matter which technology is used, there are some aspects that must be addressed in a technology-specific way if the method is intended to be practical and systematic. For example, in any product line it should be possible to switch between two or features [27], or to remove an optional feature [27], without too much effort. But how to implement this variation points may drastically differ from one technology to another. Even though design patterns may help in reducing these differences [1], when it comes down to code there are some specific aspects that must be dealt with by the method or process.

In this context, this paper presents a method for domain implementation in software product lines that integrates a specific technology, called OSGi. OSGi was chosen due to its applicability in many different scenarios, and the possibility of being used together with other technologies. Also, the method is based on a set of guidelines, inputs, outputs,
and roles in order to solve the gaps and problems found in previous works.

2. Related Work

2.1. Approaches for CBD

In [46, 30, 11, 25, 17, 20, 19, 16] several approaches for component-based development are presented, involving processes and methodologies based on object-orientation, UML, aspects, and component frameworks. Except by [46, 25], these works do not discuss in details how to implement and document software components. They do not present, for example, the specific details of implementing variability, or documenting the components in a way that facilitates their tracing back to the requirements.

In [46], the authors present a method for directly mapping cooperating suites of classes into encapsulated C++ implementations based on nested classes, parameterization, and inheritance. The method is not easy to understand and apply in large-scale. Additionally, detailed steps are not discussed, input, outputs and component documentation issues. In [19], Jonge proposes a new paradigm: package-based software development, which combines fine-grained software reuse and integrated component development. Jonge defines a life-cycle and the steps for this method. However, according to him, package-based software development is not easy to use because both the paradigm and the tools to support it are still subject of research.

2.2. Classical Methods for CBD

Component-Based Development methods, such as Catalysis [21] and UML Components [15], although relevant in CBD context, also present gaps when the focus is on component implementation. In Catalysis, for example, the flows among tasks are not concretely defined. Additionally, the method does not deal with component documentation. The same is seen in UML Components.

2.3. Software Reuse Processes

In [3], we surveyed 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), corresponding to the state-of-the-art in the area. This study shows that the processes present some problems and gaps in important activities such as development for and with reuse, besides putting too much emphasis on some specific activities (analysis, design and implementation). In domain implementation, there is a lack of details in component implementation and documentation is not discussed.

3. A Method for Domain Implementation in Software Product Lines Using OSGi

The method is based on a set of principles that help the developer during the development and documentation of software components. The method and its principles are generic, and can be used with different implementation technologies. However, if a method is intended to be practical, it must focus on some technology, because most issues related to implementation are technology-specific.

There are several technologies with features that could be used in domain implementation. For example, the NetBeans Mobility Pack (http://www.netbeans.org/products/mobility/) has a solution for device fragmentation that allows the software engineer to rapidly switch between fragments of code, which could be used to implement variation points. Most component-based frameworks, such as EJB (http://java.sun.com/products/ejb/), can be used to manage the interaction and lifecycle of the components, making the assembly of a product easier to perform. Although these technologies have many features that help in these aspects, they also have other features, such as transaction, security and persistence management, that are not always necessary, in which case the final product becomes heavier than it could - or should - be.

It is also possible to implement the components in the form of Web Services [40], which are platform-independent, thus facilitating the integration of different services into a product. But this technology has some limitations in terms of the degree of interaction between the components (services, in this case). Since most communication must be achieved in terms of method invocations, other types of interaction are more difficult to implement. Also, Web Services are normally used in distributed systems, being less suitable for non-distributed environments.

So, we decided to use the OSGi technology to implement the components and manage their interaction and lifecycle. Next section briefly describes this technology.

3.1. OSGi

Open Service Gateway Interface (OSGi) is a Java-based interface specification that defines a standardized, component-oriented, computing environment for networked services. The scope of the OSGi platform is [38]: a standard, non-proprietary, software component framework for manufacturers, service providers, and developers; a powerful model for co-existence of different components/applications in a single JVM; a flexible deployment API that controls the life cycle of applications; a secure environment that executes applications in a sandbox; and a cooperative model where applications can dynamically dis-
cover and use services provided by other applications running inside the same OSGi platform among others.

The main reason for using OSGi in the proposed method is the fact that it is very suitable for implementing components in software product lines. Firstly, it facilitates the implementation of variation points, by providing an easy way to dynamically change between different implementations, or incorporating new components, without the need to recompile the whole system. It also facilitates the process of upgrading or replacing single components, thus facilitating maintenance within the product line.

Also, although OSGi is specific to Java, it is not platform-specific, requiring only a small runtime container that to manage the component registration. This container has only the minimal features required by the components to execute and interact, thus reducing the amount of unnecessary functionalities present in the product. It can also be used with other technologies as well. For instance, it is possible to combine OSGi capabilities with those from a Web Container. Finally, this is a widely used technology, already established as a standard, being present in several success cases, as can be seen in the OSGi alliance website (www.osgi.org).

A key principle of OSGi is asynchronous management, which means that running applications must be prepared to react immediately to management interactions. For instance, a software component may be upgraded and applications using the component must be able to migrate easily from the old component to the new one. This feature makes OSGi a dynamic environment.

In the current Release (R4), the OSGi software has the following layers [38]: Security Layer, that specifies how to sign OSGi assets. This layer builds heavily on Java JAR signing and the bundle concept [39]; Module Layer, responsible for managing of the unit of modularization, called bundle. A bundle is comprised of java classes and other resources, which together can provide functions to end users; Lifecycle Layer, that provides an API to control the security and lifecycle operations of bundles; and Service Layer, responsible for registering services, searching for them, or receiving notifications when their registration state changes [38].

3.2. The Method

In order to define a practical and effective way to perform the domain implementation, the method is based on a set of Principles (P). These Principles combine the academic experience from our group, gathered from a survey of the state-of-the-art in the area of Component-Based Development [44], and the industrial experience gathered from industry experts.

P1. A component must have interfaces. The functionality of a component is defined by its interfaces. Syntactically, interfaces are similar to object class descriptions: they contain attributes and operations. Semantically, provided interfaces specify those services or capabilities that a component offers to other components in order to carry out its own responsibilities. There is also the concept of required interfaces, which is essential to enabling software plug-and-play [14], and software reuse. A required interface exposes operation invocations that a component makes to other components.

P2. A component should have a transparent life cycle mechanism. The concept of variability in product lines requires that each product has a different arrangement of components in order to implement the features selected for that product. According to the selected features, sometimes it is necessary to add, remove or even discover which components (features) are present in a product, in a dynamic way. In order to make these tasks easier, a component life cycle mechanism has been used with success. Most component-based technologies has a life cycle mechanism, such as EJB, for example. In OSGi, the Life Cycle Layer allows bundles to be managed.

P3. A component should be configurable. Variability in product lines can be also implemented through a configurable component. Sometimes it is possible to use parameters to determine if a feature is present in a product or not. The main advantage of using a configurable component is that it is easier to switch between different implementations than by using different components for different implementations, even with the help of a lifecycle management mechanism (Principle P2). Usually, a configuration file can be used to specify the parameters, so that modifications in the source code are not necessary. Therefore, in order to increase the possibilities of being reused, a component should be as configurable as possible.

P4. A component should have a proper third-party integration mechanism. If a component depends on third-party integration to obtain a service, the third-party API should not be directly used in the component implementation [24]. One interesting strategy can be to develop abstract classes such as adapters, and to isolate third-party calls inside these adapters. This not only facilitates third-party integration, but also makes the incorporation of different third-party implementations easier to perform. Another advantage of this approach is that if changes occur, the adapter needs to be changed, but the component does not. If there are several adapters, the principle P3 (A component should be configurable) can be useful.

P5. Context independence. Software components should state the assumptions made about their context. Thus, components do not need to know which other specific components will they interact with or will be “hard-wired” with them. Components should be freely connected at assembly
time, as long as connections have been verified [47, 14].

P6. Documentation. Component documentation is an important issue because it provides information that assists reusers to assess whether the component fulfills their requirements and architecture and restrictions for using it.

P7. Evolvability. A key premise of plug-and-play is that systems will evolve easily [14]. Thus, components should be added, removed, or replaced after deployment in a transparent way. The use of a life cycle management mechanism, such as the one existent in OSGi, allows these operations to be managed in a coordinated way through command lines or using an administrative visual interface.

P8. Version Compatibility. Versions of components can change during the evolution of applications. Usually, vendors provide compatibility with previous versions. Nevertheless, the management of version numbers for applications becomes more complex because of the availability of multiple component versions. Also, in some cases a new component may change its API (e.g., methods move up or down the inheritance hierarchy, or signatures change). To ensure compatibility, versions can be checked at bind-time if direct interfaces are used. Immutable interfaces could solve this problem. OSGi solves this problem with its built-in version management mechanisms, which allow bundles written for the new release to adapt to the old implementation.

Our method for domain implementation consists of two steps: component implementation and component documentation. Since it is based on OSGi, we will use the term bundle to refer to a software component. However, it is important to stress that this concept of component is the same as known by the reuse community.

3.3. Component Implementation

In this step, the software engineer, based on requirements, implements the software components through a set of well defined activities. Our method is intended to be used in the scope of domain engineering, and therefore it depends on assets developed in domain analysis (feature model, requirements, domain use case model) and domain design (domain-specific software architecture, component specifications).

The method is divided into two sets of activities, each one with a different purpose. Activities 1 to 4 deal with the provided services, i.e., when the developer wants to implement a component to be reused. Activities 5 to 7 deal with required services, i.e., when the developer wants to reuse services from existent components.

Activity 1. The first activity is to describe the component, providing general-purpose information, such as the component vendor, version, package, among others. This information may be used to identify a component, an important issue when components are stored in a repository, for example. It may also be used to dynamically publish the provided interfaces to be used by other components. In OSGi, this description is performed in the manifest file. This file provides information about the component that the OSGi framework needs to correctly install and activate a component. In order to do so, a framework implementation must: process the main section of the manifest, ignore unrecognized manifest headers, and ignore unknown attributes and directives.

Activity 2. In this second activity, the software engineer should specify the provided services. This is the same as specifying an interface, consisting of operation and attribute definitions. Artifacts developed in domain analysis and design, such as the domain-specific software architecture and component specifications may be reused in this activity, facilitating the software engineer’s work.

Activity 3. In the third activity, the goal is to implement the provided services defined in the previous activity, as well as the code to register these services to be used by other components. In OSGi, components must have an Activator class specified in their manifest file, created in Activity 1. This class implements the BundleActivator interface, which requires the implementation of two methods, start (BundleContext) and stop (BundleContext), used by the programmer to register the component as listener and start any necessary threads. The start method can allocate resources that a component needs, start threads, register services, and more. The stop method must clean up and stop any running threads. The relationship between the framework and its installed components is realized by the use of a BundleContext object, which acts as a proxy to the underlying framework.

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 in the production environment. In OSGi, the software engineer compiles all the files and generates a jar file for the component. Next, the component can be deployed in the framework using command lines or a visual interface, such as the Knopflerfish Open Source OSGi Framework (http://www.knopflerfish.org/), which has an interesting visual administration interface.

Activities 1 to 4 deal with the provided side of a component. In order to implement the required side, three activities should be performed:

Activity 5. First, 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. In OSGi, this is also performed in a manifest file, where the required services and packages are defined.
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, wrappers, or other ways to implement this access. The main goal of this activity is to provide low coupling between the required service and the rest of the code, so that it can be more easily modified or replaced. The OSGi specification offers a ServiceTracker class that can help in achieving this goal, by tracking the registration, modification, and unregistration of services in an efficient way. In practice, the fundamental tasks of a ServiceTracker object are: i. to create an initial list of services as specified by its creator; ii. to listen to ServiceEvent instances so that services of interest to the owner are properly tracked; and iii. to allow the owner to customize the tracking process through programmatic selection of the services to be tracked, as well as to act when a service is added or removed [38].

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

Although these two sets of activities (1-4 and 5-7) are focused on different aspects, in practice they will be performed together, since most components have both provided and required interfaces. However, in the end of the dependency chain between components, there are components that only have provided interfaces, and in this case activities 5 to 7 are not necessary.

Once the component implementation is accomplished, the next step of the method corresponds to the documentation of the software components.

3.4. Component Documentation

Software documentation can reduce the time and effort to develop new software, increase the ease of porting software to different platforms, and help users to understand software more easily [41]. However, there are some problems, such as [50]: There is no universally recognized standard for software documentation, in part because documentation style and content differ among programmers and sometimes differ for the same programmer under different circumstances. Additionally, the choice of programming language and the nature of a program may dictate a particular style of documentation that might not easily apply to another environment.

While there is no universally recognized standard for software documentation, there are some directions that can be useful [41]. The first one is a standard for documenting engineering and scientific software, developed by the American National Standards Institute and the American Nuclear Society in 1995; it is called ANSI/ANS 10.3-1995 Standard for Documentation of Computer Software. The standard is not a rigid set of specifications, but a guide that can apply to most software projects intended for internal or external use. The second one is a consortium of software development and component management tool companies, including IBM Rational, Flashline, LogicLibrary, Borland, and ComponentSource, that defined common conventions for packaging software assets that makes it easier to manage and reuse them in a standard called Reusable Asset Specifications (RAS) [50].

However, we need more data and experience reports in order to use these in large-scale. Additionally, some research works showed that software documentation is, in practice, poor, incomplete, and too difficult to maintain under industry time pressure [13, 31]. They also point out that software engineers typically do not update documentation either timely or completely [31].

In the context of software reuse and component-based development, software documentation is very important, once reusers need to identify, understand, adapt and integrate software components in their applications. Moreover, software components are subject to composition by third parties that sometimes are not known by the developers [47]. Thus, component documentation is a key aspect to development for reuse in general [45, 29, 48]. However the component documentation research area has been neglected [29]. This can be seen in the component-based development methods (Section 2) as well as in the software reuse processes [3]. Not only the existing methods and processes fail to cover the implementation step in a proper way, but they also do not address documentation issues. However, we need to improve it, because without adequate documentation we may become component archeologists, digging through the source code to uncover its functionalities [29].

Most work related to component documentation [45, 29, 48, 8] 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), a component quality model [6], and real world experience; ii. it follows some Principles (P) for component documentation:

P1. HyperText. HyperText has proven to be effective to documentation. For instance, it helps to increase the knowledge that software engineers can acquire by browsing through component repositories [26]. It also facilitates the process of navigating through information, since most users are already familiar with hypertext. Therefore, whenever possible, the documentation should be presented as hypertext;

P2. Embed content in source code. The component documentation should be always synchronized and updated with the source code. By keeping the documentation embedded in the source code, it is easier to update the documentation whenever changes are made in the source code,
since both are located at the same place;

**P3 Automation.** In order to relieve the workload of the component developers, documentation production and maintenance should be as automated as possible:

**P4. Leverage programming languages semantics.** Programming language constructions are very useful to software reuse. For example, signature matching [52] takes advantage of interface definition in order to determine if a component can be reused in some context. It is also possible, for example, to determine some characteristics of the component by examining the source code, even in an automatic way. Therefore, the documentation should get advantage of all the information about the component that is inherent in the source code;

**P5. Diagrams and Figures.** The documentation should include diagrams and figures for clarification purposes. To optimize this principle, the Automation (P3) principle should be used whenever possible.

Thus, in order to document software components, the software engineer uses a template composed of five sections and its related fields: **Basic information,** containing general descriptions about the component, such as name, purpose and keywords; **Detailed description,** containing definition of the interfaces, and information about implementation language, versioning, etc; **Quality information,** describing information necessary to assure the component’s quality, such as test package; **Deploy information,** including all that the dependencies that necessary to deploy the component; and **Support information,** with an install guide and contacts that help in identifying the people responsible for providing technical support to the component.

It is important to highlight that as the method for Domain Implementation uses the assets developed in Domain Analysis and Domain Design as input, some parts of the component documentation were already documented in the component specification and can be just refined. However, there some parts that still need to be defined, mainly those related to implementation, which were not covered by the previous Domain Engineering phases. It is also important to stress that the documentation process should be systematic, and not ad hoc, as it is normally conducted.

### 4. Experimental Study

In order to validate our approach, we performed an experimental study, to analyze the applicability of the method, both quantitatively and qualitatively. A pilot project was conducted in order to help in the definition and planning of the study, and in the elaboration of the training material. We followed the GQM (Goal/Question/Metric) paradigm to define the study. Thus, we defined the Goal (G), Questions (Q) and Metrics (M) as follows:

**G.** To analyze the domain implementation method with 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.

**Q1.** Does the method generate the domain architecture with low stability?

**Q2.** Does the method generate the domain architecture with low maintainability?

**Q3.** Does the method generate components with low complexity?

**M1. Module Stability:** A design is rigid if it cannot be easily changed, i.e., a single change begins a cascade of changes of independent modules. Moreover, when the extent of change cannot be predicted by the designer, the impact of the change cannot be estimated. It makes difficult the cost of the changes and this way, the design becomes rigid. Thus, the Instability (I) metric will measure the module stability in order to assess it. This metric is defined as:

\[
I: \text{Instability, where } I = \frac{Ce}{Ce + Ca} + 1
\]

*Ce:* Efferent Coupling, the number of classes outside this category that depend upon classes within this category

*Ca:* Affercnt Coupling, the number of classes outside this category that depend upon classes outside this category

The instability metric has range [0, 1], where I = 0 indicates a maximally stable category and I = 1 indicates a maximally unstable category.

**M2. Module Maintainability:** In domain engineering projects, maintainability is very important since a domain is continuously in evolution with new features or products being developed. Thus, the Maintainability Index (MI) metric will measure a module’s maintainability in order to assess it. This metric is defined as:

\[
MI = 171 - 5.2 * \ln(\text{aveV}) - 0.23 * \text{aveV} + 16.2 * \ln(\text{aveLOC}) + 50 * \frac{\sin(\sqrt{2.4 * \text{perCM}})}{}
\]

**M3. Component Complexity:** A critical issue in software engineering is related to separation of concerns and how to modularize a software system in order to result in modules and components that are well defined, testable, maintainable and more reusable. During the 70s, McCabe developed a mathematical technique [33] which provides a quantitative basis for modularization and identifying soft-
ware modules that will be difficult to test or maintain. In this technique, the complexity measure was presented and used to measure and control the number of paths through a program. This measure is also useful to keep the size of the modules manageable and allow the testing of all the independent paths. Thus, the cyclomatic complexity metric will be used to analyze the complexity of each component developed. The cyclomatic complexity of a module is calculated from a connected graph of the module:

\[ CC(G) = E - N + p \]

where:

- \( E \) = the number of edges of the graph
- \( N \) = the number of nodes of the graph
- \( P \) = the number of connected components

This metric indicates that a unit with cyclometric complexity between 1 and 10 is a simple program, without much risk. A value between 11 and 20 represents a more complex program, with moderate risk. Values ranging between 21 and 50 represent a complex program, with high risk. Finally, a complexity greater than 50 is an untestable program and presents a high risk.

We decided to use classic Object Orientation metrics to evaluate the method 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 have some problems, such as the ones raised by another study from our group [32]. 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.

4.1. Null Hypothesis

This is the hypothesis that the experimenter wants to reject with as high significance as possible. In this study, the null hypothesis determines that the use of the process in domain engineering projects does not produce benefits that justify its use. Thus, according to the selected criteria, the following hypothesis can be defined:

\[ H_0: \mu \text{the process generates the architecture with } I \geq 0.5, \ MI \leq 85 \text{ and components with } CC \geq 21 \]

4.2. Alternative Hypothesis

This 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:

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

4.3. The Context

Our experimental study using the method consisted of performing the domain implementation in the starship game domain. This project is part of the RiSE project [2]. RiSE’s goal is to develop a robust framework for software reuse, in conjunction with the industry, involving processes, methods, environment and tools.

In this case, our focus is on software reuse processes. Thus, initially, we performed the domain analysis using the approach defined in [5]. The result of this effort was about 43 features in this domain. Next, the domain design phase was performed based on the approach presented in [1]. At the end, the domain implementation method described in this paper was applied to develop and document components.

4.4. Subjects

The subjects were 7 M.Sc. students from the Federal University of Pernambuco. 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, such as components, frameworks development, or web services. All the subjects are familiar with at least one domain analysis process (FODA); three subjects had training in conferences on some issues related to reuse, such as design patterns and component-based development; and finally, two subjects had co-authored papers involving some aspects of software reuse. They were selected by convenience sampling [51] representing a non-random subset from the universe of Software Engineering students.

4.5. Training

The training of the subjects was conducted in a classroom at the university. The training was divided in two steps: in the first one, concepts related to software reuse, variability, component-based development, domain engineering, software product lines, asset repository, software reuse metrics, and software reuse processes were explained during eleven lectures with two hours each. Next, the domain implementation method was discussed during three lectures. A set of slides and recommended readings was also made available to the subjects.

4.6. Threats to Validity

The internal validity of the study is dependent on the number of subjects. This study is supposed to have at least

---

1SIMPLE Software Factory’s website - https://simple.dev.java.net/
seven or eight subjects. Also, the material provided to the students was considered sufficient, since it covers most of the related literature, and the training was thorough and complete. This helped to reduce the need for further studies during the execution of the study, what could interfere in the results.

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. Also, the environment where the study was conducted is particularly rich in students that are familiar with reuse concepts. Other research groups interested in replicating the study should be aware of this, and perhaps extend the basic training in order to increase the subjects’ knowledge on software reuse.

4.7. Analysis of the Results

Two different kinds of data were obtained from the study. Quantitative data was obtained using the metrics defined earlier (Maintainability, Stability and Complexity), and by measuring other aspects of the produced assets, such as lines of code, development duration, number of features and number of classes. Qualitative data was also collected, through a questionnaire where the subjects could express their impression on the method.

The project was developed in 355 hours, 23 minutes and 57 seconds (total men/hour). In 44 features were specified, and the implementation produced 101 packages, and 72 classes. Additionally, 5 components and 1 example application were also developed, totaling 3732 lines of code.

Table 1 summarizes the results for the quantitative analysis. It presents the resulting mean, maximum and minimum values for the three metrics, considering all the developed components. 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. The high instability value (0.8) was observed in only one of the components, responsible for screen management, while the other four had values below 0.5. This is expected, since in the game domain, screen management is the most intensive task that is performed, and thus this component had to be highly coupled with the others, resulting in high instability. However, if we consider the mean value, the null hypothesis is reject, which means that the method can help to increase stability for most components.

In addition to these results, the subjects’ feedback was analyzed through questionnaires, also indicating that the method offers considerable guidance through implementation and documentation. The following benefits were pointed out:

- **Guidance:** the method presents a systematic sequence of steps that helps in developing the components. Moreover, by being tailored to a technology (OSGi), the method offered a pre-defined way to implement the components;
- **Documentation:** the subjects observed that the generated documentation is useful and simple to develop, due to the template. Moreover, in the future, specific tools can be used to automate this task using, for example, documentation generators;
- **Integration and Flexibility in a reuse process:** several approaches and methods for CBD are used in isolated ways, such as UML Components and Catalysis. The method described in this paper can also be used in an isolated way, just to perform the domain implementation. But its main benefits arise when used in conjunction with a reuse process. This context is being explored in our group [4, 1] and we believe that the method is well integrated with domain analysis and design steps.

Even with these benefits, the following drawbacks could be observed:

- **Technology:** the subjects who had previous experience in developing components reported that lack of experience with OSGi made the implementation more complex. However, they considered that with more training the difficulties could be avoided;
- **Integration:** as the development was distributed and always one software engineer was responsible for the component implementation, the subjects reported that the integration task for the application development was complex. In this aspect, they pointed out the need for guidelines to help in component integration.

Besides these drawbacks, the subjects also reported other problems, such as difficulties in the communication among the distributed staff, the lack of experience with components, and the lack of knowledge on the domain. These

<table>
<thead>
<tr>
<th>Metric</th>
<th>I</th>
<th>CC</th>
<th>MI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>0.4442</td>
<td>1.499</td>
<td>126.4058377</td>
</tr>
<tr>
<td>Maximum</td>
<td>0.8</td>
<td>1.625</td>
<td>150.2795238</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.233</td>
<td>1.205</td>
<td>101.9545455</td>
</tr>
<tr>
<td>$H_0$</td>
<td>$\geq 0.5$</td>
<td>$\geq 21$</td>
<td>$\leq 85$</td>
</tr>
<tr>
<td>$H_1$</td>
<td>$&lt; 0.5$</td>
<td>$&lt; 21$</td>
<td>$&gt; 85$</td>
</tr>
</tbody>
</table>

Table 1. Results for the quantitative analysis
should be further analyzed in order to determine how the method could be improved to minimize these problems.

5. Concluding Remarks and Future Work

Domain implementation methods are not given the proper attention by the research community. A possible reason could be that there are already a high number of implementation techniques available today, each one with its own benefits in terms of the different quality attributes that a component must have. However, we believe that these benefits, especially reusability, could be significantly enhanced when the implementation technology is properly tailored to a domain engineering process, culminating with activities that focus on the main concepts and principles behind software reuse.

In this paper we present a method that attempts to put this in practice, by providing a set of principles that software engineers can keep in mind while performing activities to implement and document software components using the OSGi technology. Although many of the activities are very close to the usual way of using OSGi, the focus here is on providing a systematic way of implementing components for reuse within a software product line. Issues such as variability and reusability are covered, increasing the benefits that can be obtained by a software organization.

The experimental study results indicated, both quantitatively and qualitatively that the method’s activities provide useful guidance in the development of reusable software components. Even with some difficulties, the subjects of the study managed to produce good components, which are stable, maintainable and with low complexity.

As future work, we are planning to use the component quality model defined in [6] to assure the component’s quality. We also intend to extend the experimental study to consider other domain engineering phases, in order to evaluate the overall benefits of using a systematic reuse process, from analysis to implementation.

6. Acknowledgments

We would like to thank the members of Reuse in Software Engineering (RISE) group for their participation in the method evaluation. This work is sponsored by Brazilian Innovation Agency (FINEP (MCT/FINEP/COMPGOV project)) and CAPES (Reuse-Oriented Automated Reasoning Software (ROARS) project).

References


[38] OSGi. OSGi Service Platform Core Specification, Release 4, 2005.


