A Case Study on Software Evolution towards Service-Oriented Architecture

Félix Cuadrado, Boni García, Juan C. Dueñas, Hugo A. Parada
Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid
{fcuadrado,bgarcia,jcduenas,hparada}@dit.upm.es

Abstract

The evolution of any software product over its lifetime is unavoidable, caused both by bugs to be fixed and by new requirements appearing in the later stages of the product's lifecycle. Traditional development and architecture paradigms have proven to be not suited for these continual changes, resulting in large maintenance costs. This has caused the rise of approaches such as Service Oriented Architectures (SOA), based on loosely coupled, interoperable services, aiming to address these issues. This paper describes a case study of the evolution of an existing legacy system towards a more maintainable SOA. The proposed process includes the recovery of the legacy system architecture, as a first step to define the specific evolution plan to be executed and validated. The case study has been applied to a medical imaging system, evolving it into a service model.

Keywords: Software evolution, architecture recovery, services platform, SOA, OSGi.

1. Introduction

Costs in software development are primarily dominated by software maintenance issues. If in the seventies, estimation studies claimed that maintenance consumed 67 % of total software costs [1], nowadays some authors are already talking about 90% [2]. Other studies have shown that approximately 50% of the time is spent understanding the code [3]. Solving bugs, improving performance, applying security patches or adding new features are part of the everyday jobs of a software developer.

The ISO 9126 [4] standard defines maintainability as the capability of the software product to be modified, including corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. Besides that, ISO 9126 proposed a division of maintainability in the following characteristics: analyzability, changeability, stability, and testability. These features can rely on quality indicators such as coupling, cohesion, and complexity [5]. In this paper we focus on coupling as one the main indicators of maintainability.

Loose coupling is one of the primary benefits of service-oriented systems. SOA (Service-Oriented Architecture) allows breaking the system down into independent modules (i.e. the services) that interact with each other by means of well defined interfaces [6]. All in all, SOA systems exhibit low coupling and thus are more maintainable. Therefore, one way to achieve maintainability of a legacy system is to evolve towards a services platform.

The contribution of this paper is twofold: first, we present a evolution process that includes an architecture recovery step before starting the iterative evolution phase to SOA. Second, we provide practical information on how we have applied this process to a case study. The case study focuses on the evolution of an existing medical imaging system. This system was initially developed by Ibermatica[1] and later made available as open source.

2. Evolution to SOA Process

Our process for evolving legacy systems follows a white-box approach, i.e. it relies on modifying the legacy code. The main characteristics of the process are its flexibility, as it can be applied to different cases; and the integration of architecture recovery techniques into the process. The process is composed of three clearly defined activities: first, architecture recovery of the legacy system is carried out, documenting it properly. Second, an evolution plan is defined, detailing the candidate architecture and the necessary evolution steps. Third, the plan is executed, that is, the system goes forward to SOA.

2.1. Architecture Recovery

The goal of the first activity in the recovery process is to properly document the legacy system. The information thus obtained will be the basis for the evolution plan. Apart from the high-level knowledge acquired over the process (taken from requirements or design documents), the low-level familiarization with the system (source code, configuration options, users' manual) will speed up the execution of the evolution cycles. Although this initial process may require some additional effort at the start, it will pay off in the latter stages of the evolution.

[1] The case study described here has been performed in the context of the European project ITEA-SERIOUS (project number 04032), under grant by Spanish Ministerio de Industria, Turismo y Comercio in the PROFIT program (ref FIT-34000-2007-33).

The process does not rely on any specific tool, as the selection of one or another depends on the specifics of the system (amount of available documentation, size, technologies, etc.).

2.2. Evolution Planning

This activity produces a detailed evolution plan for the system, and the rationale for the decisions and intermediate steps. It is composed of four sequential phases: architecture selection, iterations definition, iterations planning and feasibility check.

- Architecture selection: The objective of the phase is to choose the future architecture of the system. This is one of the key decisions in the evolution process, as architecture plays a fundamental role in the quality of the system. Several candidate architectures and technologies should be evaluated for their suitability to the needs of the system, and compared against the recovered legacy architecture, so advantages and disadvantages of changes can be evaluated.
- Define evolution cycles: The objective of this phase is to define the different evolution cycles the product needs to iterate in order to satisfy the goals driving the evolution process. The results from the architecture recovery process will serve as additional input for defining cycles suited to the system.
- Plan evolution cycles: Once the different evolution cycles have been defined, it is necessary to develop a plan for its implementation. This phase analyses the dependencies between the cycles, scheduling their execution.
- Preliminary feasibility check: Once a plan for the evolution is available, each step should be quickly tested, to verify its feasibility. This allows the early detection of severe problems before the actual refactorings start. If the validations do not succeed it will be necessary to step back to the "define cycles" phase (or in extreme case back to the architecture selection), in order to address these problems. When these tests succeed the process will continue to the execution of the defined cycles.

2.3. Evolution Execution

The definition process has produced a detailed evolution plan, describing every step of the process and providing a schedule for their execution. With this document and the original system as inputs, the system will be refactored to achieve the initial objectives of the process. The actual nature of this process will vary from one case to another. Depending on the requirements, the phases can be executed either sequentially or in parallel. Quality can also be evaluated between each cycle in order to track the evolution of quality over the process.

3. A Case Study in Evolution to SOA

The process has been applied to a case study, where an existing product has been evolved to accomplish several goals. The analysed product is a Java-based medical imaging system, of small size (about 10k lines of code). The product is currently being used in several Spanish hospitals. It allows visualizing several medical images at the same time and applying them image transformations.

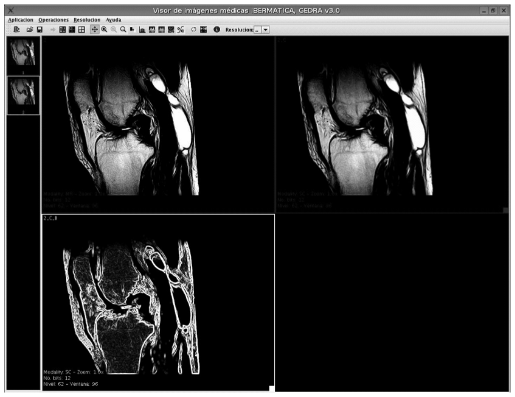

Figure 1. The Medical Imaging System GUI

The main objectives achieved in this case study are:

- Improving the usability and performance of the product. Most complaints from end users refer to the user interface. The overall look & feel of the application should be improved and it should allow a higher degree of customization. Application responsiveness is another important factor affecting the user's experience, and can be addressed by improving system performance.
- Improving the maintainability and interoperability of the system. The original design of the product did not take into account the future needs. It should be refactored to ease future evolution steps and ease its integration with other healthcare systems.

In the following, we describe our experiences in the case study, highlighting the most valuable lessons learnt at each stage.

3.1. Architecture Recovery

The recovery process executed was based on QAR (QUE-es Architecture Recovery) [7]. QAR is a generic architecture recovery workflow that follows the extract-abstract-present paradigm, and divides the extraction process into three activities (documentation analysis, static analysis and dynamic analysis). QAR offers a process framework for architecture recovery that can be tailored to the specifics of the application domain (see Figure 2).

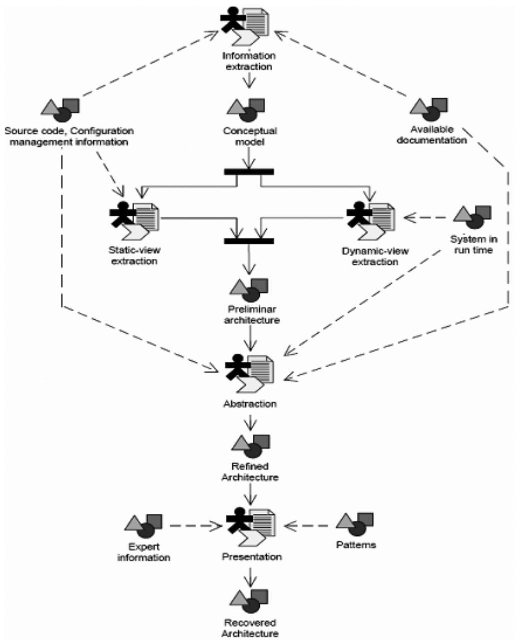

Figure 2. The QAR Workflow

The only available documentation of this system was the user manual. We did not have access to design documents or to developers. These assets are usually very important for traditional recovery approaches.

Architecture recovery processes can not be efficiently executed without the aid of tools, as these processes involve data gathering and visualization activities, which can be fully automated or at least semi-automated with human supervision. However, there are parts of the process, specially the abstraction phase, which require human reasoning and thus can not be performed by tools. Several recovery-enabling frameworks and tools have been developed to address this need; MOOSE [8] and Rigi [9] are some of the most widely used. Instead of these general recovery framework tools, we have chosen specific Java-based tools for each QAR stage. These tools are Jude [10], Omondo UML Studio [11] and Eclipse TPTP (Test and Performance Tools Platform) [12] [13].

Using these tools puts recovery processes at hand for staff that is not familiar with traditional methods and processes. The main limitation of the process is the scalability. It may need some adjustment for being applicable to larger projects.

3.1.1. Information extraction. In the initial phase we analyzed the system documentation in order to obtain the conceptual model of the system. The main source of information for this process was the user manual, where the main use scenarios of the product were defined. We also analyzed in this process third party libraries used by the product (JAI [14] for advanced image transformations) and obtained basic domain knowledge (about the medical imaging standard DICOM [15] it conforms to) from external sources.

After this analysis, we obtained the following products: a domain model of the analyzed application, a list of use cases covering the functionality provided by the tool and the required software infrastructure of the system.

3.1.2. Static View Extraction. Here we used two freely available reverse-engineering tools to extract the static view of the system: Jude and Omondo UML Studio. These tools automatically analyze Java source code, generating UML class diagrams at class and package level, detecting inheritance and dependency relations between elements. In our initial evaluations both tools provided similar results in the Java code analysis. However, the extraction and diagram manipulation were faster with Jude. On the other hand, the functionality of Omondo was more complete, with the ability to generate package-level diagrams. In the end we used both in order to obtain more complete results.

As a preliminary hypothesis for the system structure we considered each Java package as a module of the system. This partition simplified the analysis although it could be inconsistent with the recovered architecture. Following this approach we used reverse engineering tools to obtain one class diagram for each package neighborhood (constituted by the classes of the package and their immediate dependencies with the rest of the system), plus one global class diagram of the whole system and an inter-package dependencies diagram. This produced 19 diagrams.

3.1.3. Dynamic View Extraction. In this process we used a profiling tool for extracting runtime information. First we defined some representative scenarios to be executed. These scenarios were derived from the use cases obtained in the first step, trying to reflect the typical interactions between the user and the system. Scenarios were also designed with two additional criteria: maximize system coverage, and minimize common execution sequences between scenarios. The main scenarios we defined are:

- Application startup, for identifying the sequence of participating classes in the application launch.
- Image transformation, which represents a typical sequence of operations performed with the application: with the application started we loaded an image, applied a set of transformations (sharpen image, zoom and scroll) for observing the new image better and finally, stored the modified image on the hard drive.

The tool selected for the dynamic analysis was the Java Profiler of TPTP 4.2, a set of Eclipse extensions for systems' monitoring and profiling. TPTP captures three types of run time information: method invocation, execution time and number of instances in memory. The information can be presented in multiple ways, including forms and charts, UML sequence diagrams and graphical execution maps. The main limitation of TPTP is that although the raw data can be exported to XMI format that can be imported by other tools, doing the same with the generated UML diagrams are not possible. This constraint obliges to use the same tool for abstraction tasks over the dynamic view of the system.

3.1.4. Abstraction. The abstraction process was performed by hand, transforming the diagrams in order to obtain a higher level architecture. The process consisted of a series of filtering actions, applied sequentially until a higher level view of the system was obtained.

The analysis was executed combining both the static and dynamic system views. On the positive side, working with both views under the same environment greatly simplified the task; however, it was still necessary to manually synchronize abstractions performed at both levels.

Regarding the dynamic view, the sequence diagrams were simplified along this process by applying two different techniques [16]: horizontal abstraction (fusion of several lifelines into one) and vertical abstraction (collapsing method calls in order to hide its inner sequence of interactions). Both techniques are natively supported by TPTP.

The next step aimed to detect the fundamental classes of the system. The objective was to identify the most important classes of the system, according to several criteria. The analysis was simultaneously performed at two levels: analyzing each package neighborhood and analyzing the whole system with the inter-package dependencies. The results from this step discarded 119 classes from the system architecture.

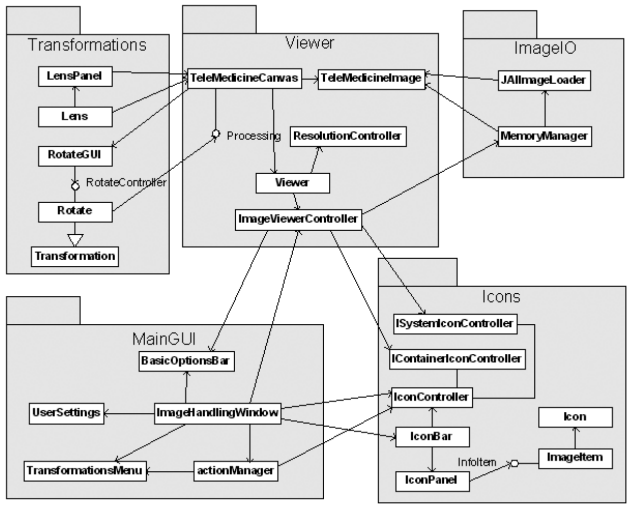

Figure 3. Recovered Architecture Diagram

From the beginning phases of the abstraction process it became clear that the initial hypothesis, one module per Java package, was not a good representation of the underlying architecture. Coupling between packages is high and there are several dependency cycles between them. Because of that, the final step restructured classes and packages into coarser-grained high level modules. The main criterion for that separation was twofold: clearly separating the main areas of functionality of the system, and isolating dependencies on the software infrastructure. After this reasoning we defined three modules: the user interface, the image transformations functionality, and the image i/o access. User interface is composed by three sub-modules (Icons, MainGUI, Viewer) due to its complexity. Figure 3 shows the resulting diagram after the abstraction process.

3.2. Evolution Definition

The four phases of this sub process produced a detailed evolution plan for the system which accomplishes the goals of the evolution process. The plan was also adapted to the singularities of the system, reflected in the recovered architecture.

3.2.1. Architecture selection. We were looking for a new architecture which supports an increased degree of maintainability and interoperability for the system. These are general good design principles, which are explicitly supported by Service-Oriented Architectures [17]. In the Java context, the OSGi [18] specification provides a functional framework based on these principles.

The OSGi service platform is a specification developed by the OSGi Alliance, which defines a framework for service execution, plus some basic services and facilities for service lifecycle management, including a registry of services and locally available service implementations.

These characteristics have lead us to select OSGi as the enabling technology for our future architecture, which will be composed of a set of loosely coupled dynamic components integrating seamlessly via services. Thus, moving the system architecture to the OSGi service platform is a requisite to evolve the system into a full-fledged SOA.

The selection of this service execution framework and the evolution of the medical imaging system towards its usage would provide the following advantages:

- Improved maintainability of the system and the architecture, as being loosely coupled services, the interrelations are governed by the services registry. Relations can be changed at runtime and services can be changed to interact remotely in a transparent manner.
- Improved configurability of the system, as services can be combined in different ways and lead to different runtime configurations.
- Improved sustituibility of parts of the system, as the interactions between services allow for the replacement of service implementations. In some cases, complete

subsystems can be replaced by new libraries or third party provided services.

- Improved service management capabilities, as the definition of isolated services on top of the OSGi service platform allows for the (remote) management of each of them. This opens a wide range of possibilities, where the company providing the software could face a change of business model and provide (parts of) the service instead.

3.2.2. Definition of the steps. With our objectives in mind we defined the required evolution cycles to achieve the desired targets. This phase produced these four cycles, based on the goals and the future architecture:

- Migrate the product to the selected architecture, the OSGi Service Platform. In this case study we have chosen Eclipse Equinox (OSGi R4 implementation).
- Refactor the product as a set of components interacting through services.
- Develop a substitute user interface for the system, based on RCP (Rich Client Platform). RCP is a framework built over Eclipse which allows rapid development of client applications, allowing for high customizability and integration with different tools [19]. The successful Eclipse IDE itself has been built using this model. It also should improve the reliability because of the use of SWT, a low-level graphic widget toolkit substituting the standard Java Swing, which is slower as it is emulated over the virtual machine instead of invoking system calls.
- Open up the system to interoperability with PACS (Picture Archiving and Communication Systems). The current version of the product works with locally-stored images, while the new SOA migrated system would be able to interact with remote image servers, and also be able to be included into a medical workflow by means of the publication of the service interface using Web Services.

3.2.3. Planning of the steps. In this step we planned a workflow for the execution of the cycles. The first step has to be the migration of the architecture to the OSGi service model, which is mandatory before any of the other steps are executed. The second step must also be the refactoring of the legacy components into OSGi components and services, which will greatly simplify the other two cycles. The final two cycles can be executed in any order, or even be executed in parallel, thanks to the refactoring done in the second step.

3.2.4. Feasibility check of the steps. Some quick tests of the interoperability of the technologies involved were made in order to make a preliminary validation of the defined process. To do that, we have performed unit-testing on the OSGi components (bundles) using JUnit. These tests did not expose any problems, and the experience obtained with this effort helped in the evolution execution stage.

3.3. Evolution Execution

This step consists of refactoring the legacy product into a set of loosely coupled services, deployed as bundles. The documentation obtained from the architecture recovery process is really helpful here in order to separate the functionality of the legacy code, easing the refactoring. The identified modules in the recovery are: graphical user interface (the most part of the application), the image transformation component, tied to the JAI library, and the image access/storage functionality. For each functional module we defined a generic service, with a specific implementation in the existing code base. This allows the task of extending the application by substituting one service implementation for another (i.e. local folder access for remote server access).

For decoupling the system components we have chosen the whiteboard pattern [20]. This pattern is an example of inversion of control (IoC), that is: “do not call us, we will call you”. The service providers register listeners in the OSGi Service Registry. When a consumer needs that service, looks for it in the Service Registry and binds it (see Figure 4). This way presentation is completely decoupled from underlying logic.

The last step performed has been the duplication of GUI by offering two possibilities for the underlying platform: Swing and RCP. The functionality of the system has not been changed, but the usage characteristics do, as each of the libraries provides different look and feel and performance figures. By using the whiteboard pattern in runtime –which is allowed by the OSGi platform- the selection of GUI service implementation can be changed during execution, so the goal of adaptation to the user has also been met.

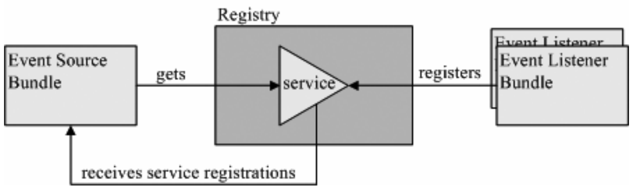

Figure 4. Whiteboard Actors in the OSGi Framework

4. Conclusions

This article has presented a case study for the evolution of software systems with the particularity of being poorly or scarcely documented. It has been carried out to a medium-sized legacy system with satisfactory results, and has focused on the evolution of an existing medical imaging system to adopt SOA principles. The case study description provides valuable contributions for practitioners, who will find useful guidelines and recommendations on how to use tools such as Eclipse TPTP or Omondo UML to enact evolution processes that require architecture recovery tasks. In general, the results obtained with these tools have been acceptable, but they have some limitations that should be addressed to improve the productivity of software evolution processes, e.g. it is not possible to export gathered data to a common format for data exchange between tools and they lack facilities for the programmatic manipulation of data, which could be needed when working with large systems.

Our case study includes an architecture recovery stage, prior to the planning and execution of the evolution cycles. Architecture recovery saves work for subsequent tasks. Furthermore, in some cases it is simply a must, because the system is too complex to be understood just by visual code inspection. Once the architecture of the system is recovered, we recommend applying a series of refactoring iterations to evolve the system. This process has proven to be suitable for medium-sized systems, but it might not be the best option for very large systems. Instead, creating black-box wrappings around the product might be the only real alternative. In addition to the process description it is interesting to point out that the selected architecture for the case study, the OSGi Service Platform, has proven to be a natural candidate for evolving legacy Java applications to service-oriented systems. In addition, the OSGi platform leverages extensibility mechanisms over a lightweight core and provides seamless interoperability with other SOA technologies such as Web Services.

5. References

[1] Zelkowitz, M., Shaw, A. & Gannon, J. "Principles of Software Engineering and Design". Prentice-Hall, 1979.

[2] Erlikh, L. "Leveraging legacy system dollars for E-business". IEEE IT Pro, May/June 2000.

[3] Standish, T. "An essay on software reuse". IEEE Transactions on Software Engineering SE-10, 1984.

[4] ISO/IEC, "ISO/IEC 9126. Information technology - Software product evaluation - Quality characteristics and guidelines for their use" International Standards Organization, Geneva 1991.

[5] Zou, Y., Kontogiannis, K. "Migration to Object Oriented Platforms: A State Transformation Approach" ICSM 2002.

[6] Pulier, E., Taylor, H. "Understanding Enterprise SOA". Manning, 2006.

[7] Arciniegas, J.L, "Contribution to Quality-driven Evolutionary Software Development Process for Service-Oriented Architecture", Ph. D. Thesis, Polytechnic University of Madrid, 2006.

[8] Ducasse, S., Lanza, M. & Tichelaar, S. "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems" Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools), June 2000

[9] Kazman, R. O'Brien, L.and Verhoef, C, Architecture Reconstruction Guidelines, 2nd Edition (CMU/SEI-2002-TR-034). 2002.

[10] Jude (Java and UML Developer Environment), a Java UML modeling tool. http://jude.change-vision.com

[11] Omondo Eclipse UML Studio, an Eclipse plug-in for UML modelling. http://www.omondo.com

[12] Mehregani, A. & Mehregani D., "Gnireenigne Esrever Fo Tra Enif Eft, or the Fine Art of Reverse Engineering", EclipseReview Magazine, Spring 2006 issue. Available at http://www.eclipsereview.org

[13] Eclipse TPTP (Test and Performance Tools Project), an Eclipse Top-level project. http://www.eclipse.org/tptp

[14] JAI, Java Advanced Imaging API, http://java.sun.com/javase/technologies/desktop/media/jai

[15] DICOM, Digital Imaging and Communication in Medicine, a medical standard for patient image handling, http://medical.nema.org/dicom/2007

[16] Krikhaar, R., Pennings, M. & Zonneveld, J. "Employing use/cases and domain knowledge for comprehending resource usage" In Proc. 3rd European Conference on Software Maintenance and Reengineering (CSMR), pages 14-21, March 1999.

[17] Erl, T. "Service-Oriented Architecture: Concepts, Technology, and Design". Upper Saddle River: Prentice Hall, 2005.

[18] The OSGi Alliance, "About the OSGi platform", Technical Whitepaper, 2005.

[19] McAffer, Jeff., Lemieux, J.M. "Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications". Addison Wesley Professional, 2005.

[20] The OSGi Alliance, "Listeners Considered Harmful: The Whiteboard Pattern". Technical Whitepaper, 2004.