Electronic Communications of the EASST
Volume 16 (2009)

Proceedings of the
Doctoral Symposium at the
International Conference on Graph Transformation
(ICGT 2008)

Migrating Legacy Systems to Service-Oriented Architectures

Carlos Matos, Reiko Heckel

15 pages

Guest Editors: Andrea Corradini, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Migrating Legacy Systems to Service-Oriented Architectures

Carlos Matos[1,3], **Reiko Heckel**[2]

[1] cmm22@mcs.le.ac.uk
[2] reiko@mcs.le.ac.uk
Department of Computer Science
University of Leicester, United Kingdom

[3] ATX II Tecnologias de Software S.A.
Portugal

Abstract: This paper presents a methodology for migrating legacy systems towards Service-Oriented Architectures. The approach is based on source code analysis for identifying the contribution of code fragments to architectural elements and graph transformation for architectural migration, allowing for a high degree of automation. In order to transform existing application architectures into SOAs, the methodology has to be used in two dimensions, a technological and functional one.

The work presented here is being developed in the context of a collaboration between academia and industry, and is aimed at being applied in real reengineering projects.

Keywords: Reengineering, SOA, Legacy

1 Introduction

Due to the frequency of change in business requirements and evolution in technology, the need to evolve existing software systems is ever increasing. This results in demand for new methods to support this process, in particular where the transition towards modern architectures is concerned. Drivers of this demand are the adoption of object-oriented programming languages, the advent of Web technologies and specifically of Service-Oriented Architectures.

SOAs are steadily becoming mainstream software engineering practice. Reports show that more than 50% of large, newly developed applications and business processes designed during the year 2007 used service-oriented architectures to some extent [AS08]. However, experience also indicates that SOA initiatives rarely start from scratch. The technology market research firm Gartner estimates that by 2011 (with 0.8 probability) more than 80% of existing applications will be at least partly reengineered to participate in service-oriented architectures [NPSI06]. This represents a significant effort for IT departments of organisations.

With this growth in SOA adoption, the need for a systematic approach towards reengineering for SOA becomes ever more pressing. However, several principles of service-orientation pose major challenges for such reengineering efforts:

1. The separation of business from presentation logic

2. The loosely coupled relationship between services

3. The coarse-grained nature of services

As legacy systems were not usually built with these concerns in mind, much effort is needed to accommodate them. Approaches based on wrapping existing applications into web service interfaces do not fully address these principles—a deep restructuring approach is necessary.

This paper presents a methodology to address migration of legacy software to SOA complying with the above principles while allowing for a high degree of automation and providing support for the full reengineering cycle. We see our proposal as an instance of the horseshoe model [KWC98], a conceptual model for reengineering at different levels, with a focus on transformations at the level of architectural models. In this paper, this goal is achieved by using techniques such as code pattern matching and graph transformation. In order to structure the process, we propose an overall methodology, instantiated in two dimensions to address the technological and the functional evolution. The former is concerned with technical purpose of the code and the latter focuses on its implementation of relevant business-level functionalities.

The remainder of this paper is organised as follows: Section 2 describes in more detail the SOA properties mentioned above and Section 3 presents a general methodology for architectural migration. Next, a summary of the technological dimension in Section 4 is presented and the functional dimension is detailed in Section 5. Section 6 presents an initial prototype implementation of the methodology. Section 7 discusses related work and Section 8 concludes the paper and presents future work.

2 Principles of SOA

To support the migration to SOA, it is necessary to address the properties discussed in the previous section. A more detailed description of these is made in the following paragraphs.

2.1 The separation of business from presentation logic

In legacy applications it is common to find, mixed up in a kind of "architectural spaghetti", code fragments concerned with database access, business logic, presentation aspects and exception handling, among others. It is not possible to derive services directly while business logic is tightly coupled with presentation logic. Therefore, an appropriate decomposition of the code is required such that "pure" business functions are isolated as candidate services or service constituents. This technological dimension of reengineering towards SOA amounts to an architectural transformation towards a multi-tiered architecture.

2.2 The loosely coupled relationship between services

It is common to find a complex network of dependencies between different functionalities in existing systems. However, service-orientation principles state that services must interact without tight, cross-service dependencies [Erl05]. Therefore, a decomposition of different functionalities is required to provide a degree of independence.

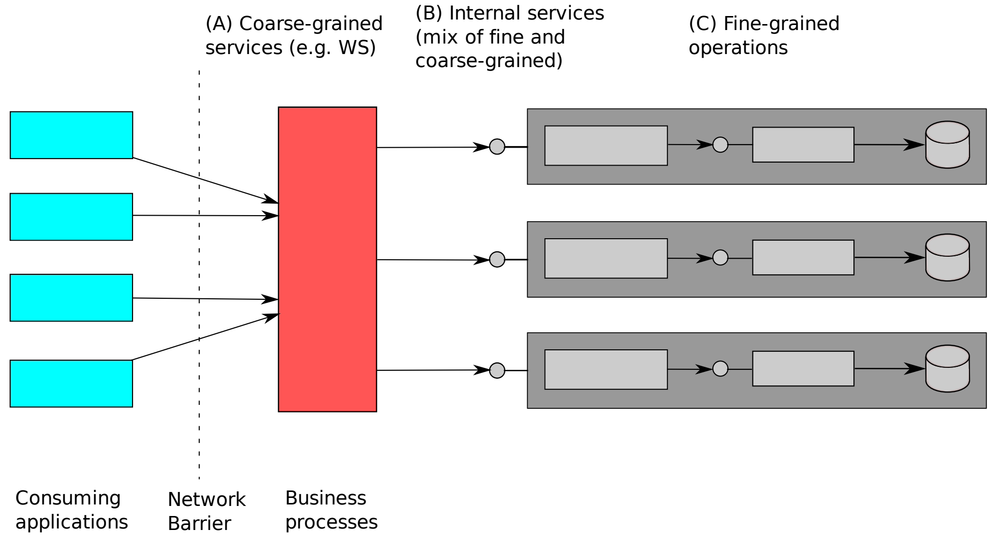

(A) Coarse-grained services (e.g. WS)
(B) Internal services (mix of fine and coarse-grained)
(C) Fine-grained operations
Consuming applications
Network Barrier
Business processes

Figure 1: Service granularity across application tiers

2.3 The coarse-grained nature of services

Legacy applications typically consist of elements that are of a fine-grained nature, for instance components with operations that represent logical units of work, like reading individual items of data. OO class methods are an example of such fine-grained operations. The notion of service, however, is of a different, more coarse-grained nature. Services represent logical groupings of, possibly fine-grained, operations, work on top of larger data sets, and in general expose a greater range of functionality. In particular, services that are deployed and consumed over a network must exhibit such a property in order to limit the number of remote consumer-to-provider roundtrips and the corresponding processing cycles.

Figure 1 presents a graphical representation of granularity across different application tiers. In an SOA context, legacy logical units of work have to be appropriately composed and reengineered in order to form services of desired granularity and of adequate support for multi-party business processes.

3 General Methodology of Architectural Migration

Depending on the intended target architecture, changes are required along either the technological or functional dimensions or, as is the case with SOAs, both. Technological restructuring is used in the layering of software systems and may lead to a 3-tiered architecture, separating logic, data, and user interface (UI). This process addresses the principle of separation between business and presentation logic. Functional restructuring separates components which, after having replaced their UI tier with an appropriate interface and being grouped according to specific parameters, represent services. This dimension addresses the properties of loosely coupled services and their coarse-grained nature.

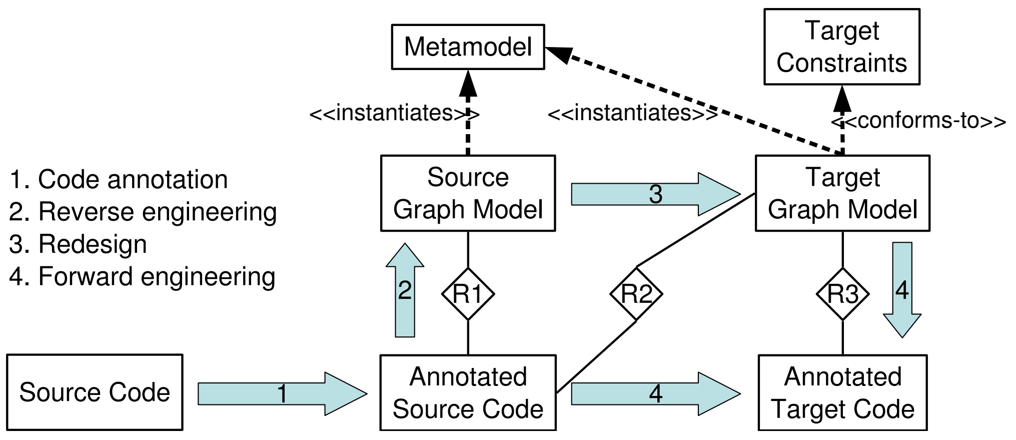


Figure 2: General methodology

The general methodology instanced for both technological and functional dimensions is presented in Figure 2. This consists of the steps described in the next subsections.

3.1 Code annotation

At this stage, the source code is annotated by code categories. Blocks of source code are labelled according to the different elements of the target architecture they will be mapped to. This process is largely automated but, depending on the source system, requires some level of input from the developer that is driving the process. This activity can thus be seen as an interleaving of automated and manual code annotations.

The code categories to be used in this step depend on the target architecture. If migrating to a 3-tier system, the categories to consider are UI, Logic and Data, as is described in this paper for the technological dimension. The categories for the functional dimension are related to the contribution of source code to particular services (e.g. managing accounts, customers, etc).

3.2 Reverse engineering

Based on the information gathered from the annotation process, this step obtains a graph representation of the code. This graph will not have a 1-1 mapping with the source code, and its level of detail depends on the annotation. Structural elements that are annotated as contributing to just one category are represented by a single node. If instead they are fragmented into several categories, each of these fragments has a separate representation in the model. Since, for example, a method completely identified as belonging to the user interface is represented by one node only, this allows the model to be much more succinct than the code, leading to highly scalable solution. Another benefit of a graph representation is that it allows transformations to be described in an intuitive, visual way.

The graph model is based on a metamodel that consists of a type graph that contains: code structure information, its categorisation and association to architectural elements. An example is shown later (cf. Section 6) when discussing a prototype implementation. This metamodel is general enough to accommodate both the source and target system, but also intermediate stages of the redesign transformation. Additionally, it contains the code categories that were available throughout the code annotation step.

3.3 Redesign

This step uses graph transformation rules to produce the target architecture. The rules conceptually extend the graph transformation suggested by Mens *et al* in [MDJ02] to formalise refactoring [Fow99]. The intended result is expressed by an extra set of constraints over the metamodel, which are satisfied when the transformation is complete. For instance, it is possible to specify a constraint that ensures there are no direct edges from a code fragment of a specific category to another of a specific different category. The code categorisation provides the control required to automate the transformation process, limiting the need for user input to the code annotation step. Rule components (left-hand side, right-hand side and negative application conditions), as well as source, target and intermediate graphs are instances of the metamodel. In the technological dimension the rules aim to re-organise the model into a 3-tier architecture, thus complying to the SOA property described in property 2.1. The rules for the functional dimension restructure the model so that services comply to properties 2.2 and 2.3.

3.4 Forward engineering

The final step is the process of obtaining the target code. This can be achieved by keeping a log of the transformations that are applied at model level and use this to drive the code level transformations. The result of this step, the annotated code in relation with a graph model, has the same structure as the input to step 1, thus allowing for several iterations of the whole process.

This is particularly relevant if the reengineering is directed towards service-oriented systems. In this case, the transformation has to address both technological and functional dimensions, i.e., respectively, transformation into a three-tier architecture and decomposition into functional components.

One of the main goals of this methodology is to allow a high degree of automation. Manual intervention, if any, is only required in step 1. A more detailed description of the general methodology can be found in [HCM$^+$08].

4 Technological Dimension

In this section we summarise the instance of the general methodology for the technological dimension. A more detailed description can be found in [CMHE07].

4.1 Code annotation

The code annotation step of the technological dimension is based on code pattern matching rules. These are used to automatically identify what source code fragments belong to User Interface, Logic and Data access concerns. Rules can be specific to a programming language (e.g.

annotating with UI all static method calls to Classes known to deal with presentation aspects), specific to a development paradigm (e.g. assuming OO code) or even technology independent (e.g. rules based in matches from other rules). Although a majority of code patterns can be used in reused for multiple projects, differences in the involved technologies, or unusual coding patterns, make it necessary to have manual intervention.

Some code statements can be considered to fall in more than one category. An example would be the result of a UI method call being used directly in a Logic operation. Since we can perform the annotation at the abstract syntax tree (AST) [KG98] level, it is possible to separate the parts that belong to one category from the others. In terms of transformation this example would lead to the direct UI call being replaced by a Controller method call (in the context of a Model-View-Controller pattern). Similar approaches are used for other kinds of mixed category statements.

4.2 Reverse engineering

The reverse engineering step is fully automated and is achieved by taking the annotated code as input and generating a graph model complying to the specified metamodel. In this representation it is possible to see links between the different architectural concerns and have an overall feel on how the original application is structured.

4.3 Redesign

The redesign step is achieved by executing a set of graph transformation rules over the source graph model. The resulting model, which is achieved automatically, will comply to the specified target constraints and reflect a correct separation between the concerns UI, Logic and Data access. This guarantees, for example, that there are no direct calls from the UI to the Data access layer or calls from Data access to Logic.

4.4 Forward engineering

In order to obtain the final code, the forward engineering step uses the information gathered in the redesign step to drive code transformations. Regarding object-oriented applications, this can be achieved via code refactorings.

5 Functional Dimension

In this section we describe the instance of the general methodology for the functional dimension.

5.1 Code annotation

The code annotation phase in the functional dimension presents more challenges than that of the technological dimension. While in the latter there is a big common ground between different applications (especially if they share the choice of technology), the former depends on application features. This, and the different nature of the two dimensions, also has an effect on the strategy of approach.

The functional code annotation phase consists of two tasks:

1. operation identification

2. grouping of operations into services

In this paper, *operation* stands for a functionality that is likely to be at a too low granularity to be considered as a service in an SOA context. The categories used in this dimension are not known beforehand. It is during the code annotation step that these will be extracted. The names drawn to identify each category are based in the operation identifier thus, depending on the accuracy of these, it may be necessary to intervene manually so adequate names are used.

The identification of operations in source code is performed by first of all locating their entry points. The techniques used for this purpose include:

- Code belonging to the Logic layer that is invoked by the UI - Code that is directly called from user interface components typically represent entry points to relevant functionality;

- External API's (e.g. from IDL files) - APIs that are published for external follow well known structures;

- Code that falls into a typical pattern of control / data flow - There are many code patterns that can help to identify entry points to application functionalities (an example is given in Section 6.1);

- Entry point for code that is mapped to more than one operation - Blocks of code that are used by several different application functionalities have entry points that are likely to lead to relevant operations (given that this is a very general approach, granularity of code blocks identified using it may vary greatly).

- Known feature location techniques such as LSI [MSRM04], which is a static approach, and SBP [AG05], a dynamic technique. There are some feature location techniques that have been tested in different environments, typically to aid in software maintenance tasks, and that presented their effectiveness.

The dependencies between each operation entry point and the remaining code can be determined using slicing techniques. A list of candidate operations can then be presented to the developer driving the process allowing human intervention / input either for manual adaptations (supported, for example, by feature location techniques LSI and SBP as mentioned above) or for a new automated round of operation identification.

In the second step of service extraction operations previously obtained are grouped into coherent services. This is an inherently semi-automated task where operations that are related in some manner are grouped together. Automation proposes ranked groupings of operations by using metrics, including: overlapping between operations, actors involved, information about data accessed and similarity measure (e.g. using LSI). User input can then be given to decide which grouping to use, either by selecting one from the proposed automatically or by making manual assignments. The result is the source code annotated according to the operations and services that it will be mapped to afterwards to produce the graph model and drive the redesign process.

5.2 Reverse engineering

The reverse engineering step of the functional dimension has exactly the same requirements as for the technological dimension. This makes it possible to use the same implementation of this step for both.

5.3 Redesign

The graph transformation rules used in this dimension are designed so that operations are grouped into meaningful services (as defined in the annotation step) and so that services have loosely coupled relations, thus complying with the last two SOA properties mentioned in Section 2.

Where in the technological dimension we have mainly rules for decomposing code structures, here there are, additionally, rules that compose/group code structures. The former are used to guarantee loose coupling and the latter to build the adequate granularity of services throughout the system.

5.4 Forward engineering

Despite the differences in the code annotation and redesign steps, the forward engineering step of the functional dimension follows the same principles that were defined for the technological dimension. Due to this, it is possible to use the same implementation of this step for both cases.

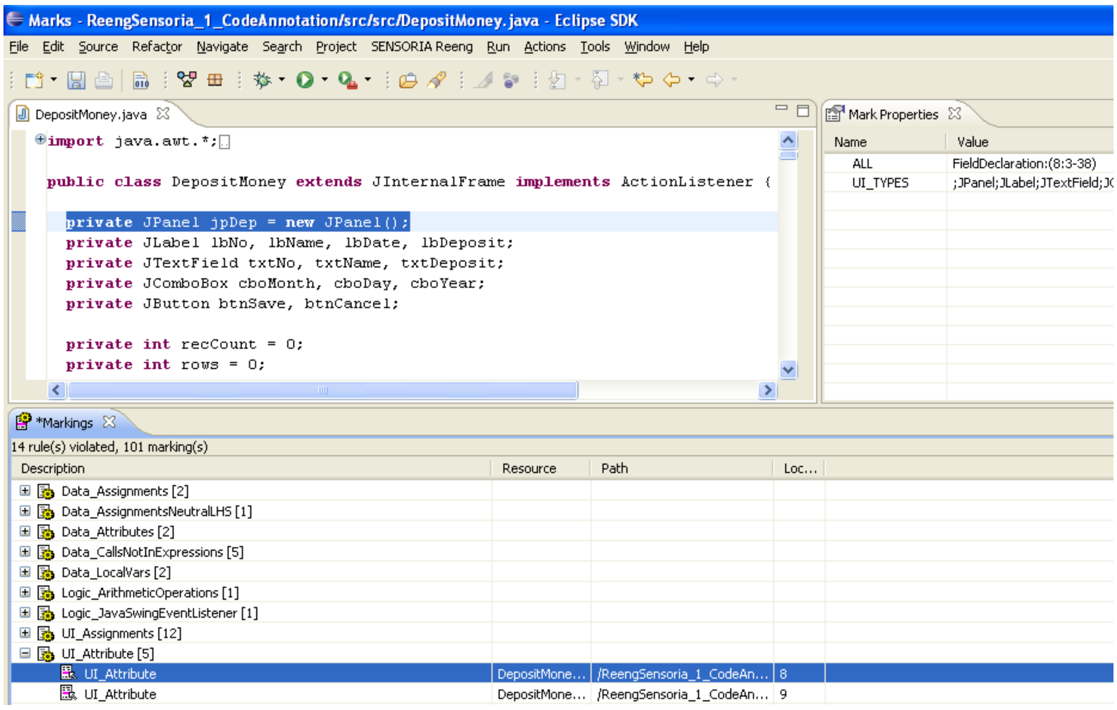
6 Initial Prototype

A prototype is being developed to apply the methodology described here. An initial implementation was demonstrated, using a small banking application in Java as scenario, at a SENSORIA project meeting [SEN]. This version did not yet perform the full code transformation and focused mainly on the technological dimension, but already supported the full reengineering cycle. The following subsections summarise this implementation for each step of the methodology.

6.1 Code annotation

The code annotation step was implemented using CareStudio (Figure 3 provides a screenshot of the application). This is an Eclipse plugin based on a tool by ATX (L-CARE), allowing the specification and execution of code pattern matching rules and storing the resulting markings/annotations in an XML file. The patterns are defined over an XML representation of the AST of the code. Rules for pattern matching are defined as XPath queries that can range from simple expressions to a combination of an arbitrary number of expressions, using the output of one expression as parameters for others. Next we provide a couple of examples of code pattern matching rules that exists in the prototype, the first belonging to the technological dimension and the second one to the functional dimension. Some expressions were simplified for readability.

1. *Attributes that belong to the user interface*. Attributes of types that are known *a priori* to belong to the UI code category can be directly identified as such. The expression used to locate these cases is:


Marks - ReengSensoria_1_CodeAnnotation/src/src/DepositMoney.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project SENSORIA Reeng Run Actions Tools Window Help
DepositMoney.java
Mark Properties
*Markings

```
import java.awt.*;

public class DepositMoney extends JInternalFrame implements ActionListener {

    private JPanel jpDep = new JPanel();
    private JLabel lbNo, lbName, lbDate, lbDeposit;
    private JTextField txtNo, txtName, txtDeposit;
    private JComboBox cboMonth, cboDay, cboYear;
    private JButton btnSave, btnCancel;

    private int recCount = 0;
    private int rows = 0;
```

Name	Value
ALL	FieldDeclaration:(8:3-38)
UI_TYPES	;JPanel;JLabel;JTextField;J(

14 rule(s) violated, 101 marking(s)

Description	Resource	Path	Loc...
Data_Assignments [2]			
Data_AssignmentsNeutralLHS [1]			
Data_Attributes [2]			
Data_CallsNotInExpressions [5]			
Data_LocalVars [2]			
Logic_ArithmeticOperations [1]			
Logic_JavaSwingEventListener [1]			
UI_Assignments [12]			
UI_Attribute [5]			
UI_Attribute	DepositMone...	/ReengSensoria_1_CodeAn...	8
UI_Attribute	DepositMone...	/ReengSensoria_1_CodeAn...	9

Figure 3: CareStudio - an Eclipse plugin for code pattern matching - showing one occurrence of an UI attribute declaration (rule UI_Attribute).

```
parameter equation UI_TYPES{';JPanel;JLabel;JTextField;JComboBox;JButton;'};
main equation ALL{//FieldDeclaration};
condition MAIN_EQ{$ALL[contains($UI_TYPES, concat(";",Type/Name/@value,";"))]};
```

2. *Methods with high Fan-In.* Methods that are called from a variety of locations in source code are likely to have a significant role in an operation (albeit potentially of too low granularity to be alone considered services). A detailed discussion about this technique can be found in [MDM04] where it was used in the context of Aspect Mining. The expression used in CareStudio to locate these situations is (variable N is a parameter for the rule):

```
main equation METHOD{//Method};
condition METHODCALLS{count(//FunctionOp[Name/@value=$METHOD/Name/@value]) > $N};
```

One consideration that can take place when analysing these pattern matching rules is that while some have to be programming language specific, others can be very general as is the case in the second example. Rules of the latter type have a big potential for reuse in multiple projects.

6.2 Reverse engineering

The abstract syntax tree representation, adequate for code annotation, is not scalable for the re-design step. Here, the graph representation allows us to abstract from the specific programming

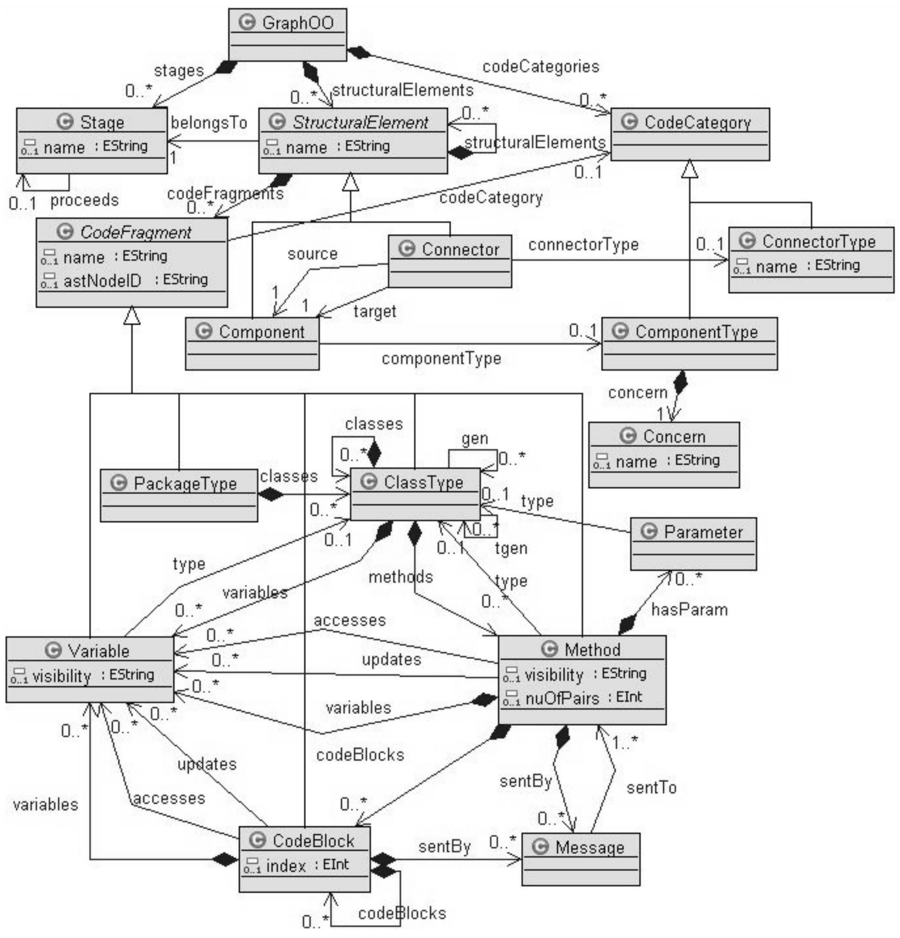


Figure 4: Type graph for the OO paradigm.

languages involved and to describe transformations in a more abstract and intuitive way. Additionally, given that we only represent in these graphs the elements required according to the annotation, the model to be transformed is simpler and the transformation more performant. This is particularly relevant when addressing the migration of large scale systems.

A specialised tool was built based on these requirements to implement the reverse engineering step in the prototype. Because of the choice of tool for the next step, the result of this one will be a graph model represented in EMF [Ecl].

6.3 Redesign

Like in the reverse engineering step, redesign transformations are based on a graph metamodel. The prototype is thus using a type graph that can represent object-oriented applications. Figure 4 presents this type graph that includes both structural information about the code (bottom) but also its categorisation and organisation in architectural components/connectors(top-right and top-left, respectively).

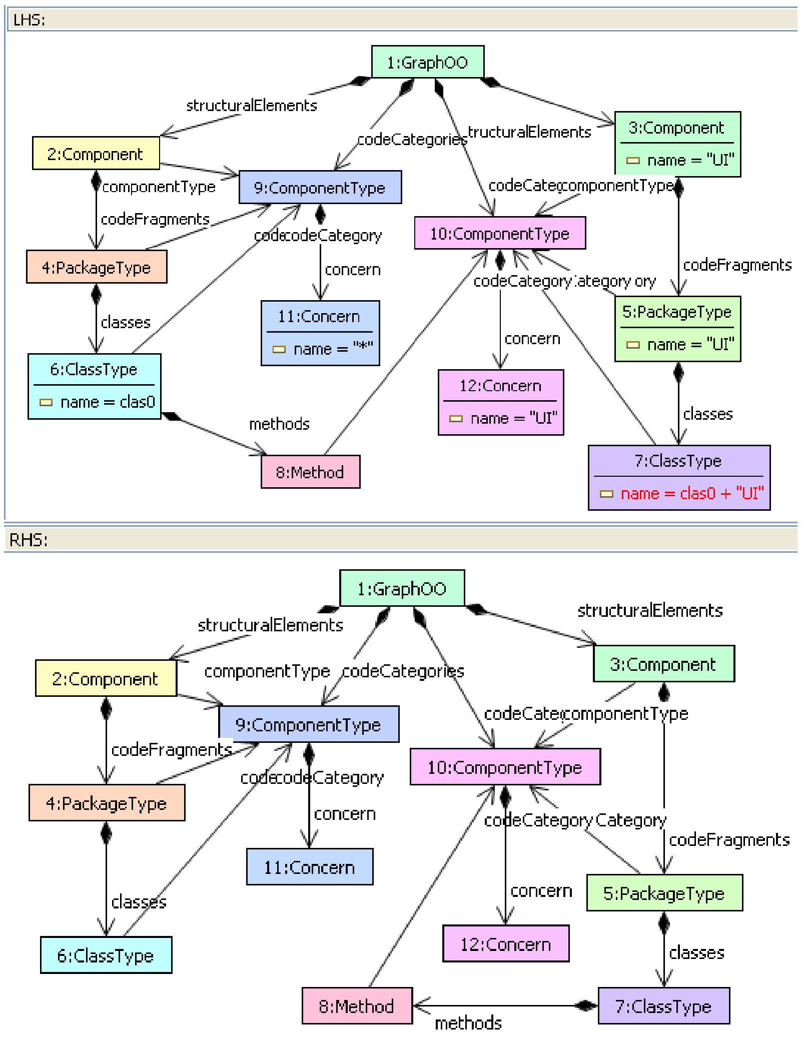


Figure 5: Move Method UI transformation rule.

The graph transformation rules were designed in the Tiger EMF Transformer tool [Tig]. This is an Eclipse plugin application that allows the definition of rules and generates Java code that is capable of executing them over a graph represented in EMF and that complies to the specified type graph. The rule management features of this tool are graphical based, facilitating the rule development.

An example of graph transformation rule can be seen in Figure 5. The top indicates the left-hand side of the rule and the bottom refers to the right-hand side. Negative application conditions are not shown for simplicity. This consists of the *Move Method UI* rule whose purpose is to move methods identified in the code annotation step as belonging to the UI code category from generic classes to UI ones. This rule performs one of the necessary activities of the technological dimension to achieve the SOA property of separating UI code from business logic.

6.4 Forward engineering

The forward engineering step of this prototype is achieved by creating a log of the redesign step, containing information on the graph transformation rules that were applied as well as their order and the nodes of the graph that were affected. This log is mapped to a sequence of code refactorings which, when applied to the source code, yield the transformed target code. The tool uses Eclipse's built-in refactorings for this purpose so the work focused mainly in mapping graph transformation rules to the right refactorings.

All the tools used for the prototype were implemented in Eclipse and can be used as plugins to this IDE. The prototype is under development and the main objective is to provide means to fully address reengineering projects. Details about future work are given in Section 8.

7 Related work

Given its wide scope, the work presented here falls into several research areas. Next we describe related work in: source code analysis, architectural transformation and reengineering to SOA.

Source code analysis, in particular feature/concept location, is related to the first step of the methodology described here. There are several techniques for this purpose including the work of Marcus *et al* [MSRM04] in applying LSI to concept location, the scenario-based (SBP) feature identification approach of Antoniol and Gueheneuc [AG05] and the work of Eisenbarth *et al* [EKS03] involving both static and dynamic feature location. These techniques are all candidates to be applied in the context of an SOA migration project, considering the first step of the methodology presented in this paper. The ARTISAn framework, described by Jakobac, Egyed and Medvidovic in [JEM05], categorises code using an iterative user-guided method. The categories used are: “processing”, “data” and “communication”. The approach differs from ours in several aspects. Firstly, the goal of the framework is program understanding and not the creation of a representation that is aimed to be used as input for the transformation part of a reengineering methodology. Another important difference is that in ARTISAn the annotation/categorisation process (called “labeling”) is based on clues that result in the categorisation of classes only. In our approach we need, and support, the method and code block granularity levels.

Work in the area of architecture transformation is diverse and includes examples closely related to our work. Use of graph transformation for reengineering has been suggested previously [CMW02] in a different context (migrating mainframe COBOL to client/server) with similarities to our technological dimension. A model-based technique based on graph transformation for a-posteriori integration of legacy applications into SOA is proposed in [Haa07]. This focuses on generating wrappers and glue code, rather than transforming directly the source code. Ivkovic and Kontogiannis [IK06] proposed a framework for quality-driven software architecture refactoring using model transformations and semantic annotations. Fahmy *et al* [FHC01] use graph rewriting to specify architectural transformations at the description level. In our case, we build a graph that models the software but also maps the code to target architectural elements. It is this information that guides the redesign process described in this paper.

Even before the advent of SOAs, approaches for reengineering business applications were proposed, based on the integration of legacy components after separating application logic from presentation [KKNS94]. Work in reengineering to SOA is new. It primarily focuses on identifying and extracting services from legacy code and wrapping them for deployment. We name just two examples in this section. Sneed [Sne06] presents a method for wrapping PL/I, COBOL, and C/C++ code behind an XML shell which allows functions within the programs to be offered as web services. A lighter code-independent approach was developed by Canfora *et al* [CFFT06], which wraps only the presentation layer of legacy form-based UI as services. Our work is different from other approaches, in that our goal is not just to provide existing functionality as services but to do so while complying to the service-orientation principles described in Section 2.

8 Conclusion

The main contribution of this work is the definition of a methodology that can be used for multiple types of reengineering projects, including migration towards service-oriented architectures, while having a high level of automation. Code pattern matching and graph transformation are central to these aims. By applying those in the technological and functional dimensions, the result is a concrete process of addressing SOA migration projects in a systematic way. This work is ongoing, but originated from problems in real projects and is developed together with industry.

A prototype is currently being developed to evaluate this approach in a larger scale and an initial version is already being applied to an example application. The early tests show that this approach can be put into practice with good results.

Next in this work is the consolidation of the code pattern matching rules for both the technological and functional dimension to achieve a more complete coverage of situations that can be found in legacy applications. This will be done in parallel with overall improvements in the techniques that will be found when testing more situations. After these activities and the prototype is benefiting from them, further validation will consist in its application to a large sized system.

Acknowledgements: This work is partially supported by the European project IST-FET IP SENSORIA (IST-2005-16004). Thanks to R. Correia (formerly ATX II Tecnologias de Software S. A.) and Dr. M. El-Ramly (Cairo Univ., previously at Univ. of Leicester) for their participation.

Bibliography

[AG05] G. Antoniol, Y.-G. Gueheneuc. Feature Identification: A Novel Approach and a Case Study. In *Proc. of Int'l Conf. Software Maintenance (ICSM)*. Pp. 357–366. IEEE Computer Society, Washington, DC, USA, 2005.
doi:10.1109/ICSM.2005.48

[AS08] C. Abrams, R. W. Schulte. Service-Oriented Architecture Overview and Guide to SOA Research. Technical report G00154463, Gartner Research, January 2008.

[CFFT06] G. Canfora, A. R. Fasolino, G. Frattolillo, P. Tramontana. Migrating Interactive Legacy Systems To Web Services. In *Proc. of European Conf. Software Maintenance and Reengineering (CSMR)*. Pp. 24–36. IEEE Computer Society, Washing-

ton, DC, USA, 2006.
doi:10.1109/CSMR.2006.34

[CMHE07] R. Correia, C. Matos, R. Heckel, M. El-Ramly. Architecture Migration driven by Code Categorization. In Oquendo (ed.), *Proc. of European Conf. Software Architecture (ECSA)*. LNCS 4758, pp. 115–122. Springer-Verlag, 2007.
doi:10.1007/978-3-540-75132-8_10

[CMW02] K. Cremer, A. Marburger, B. Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance* 14(4):257–292, 2002.
doi:10.1002/smr.254

[Ecl] Eclipse. Eclipse Modeling Framework.
http://www.eclipse.org/emf/

[EKS03] T. Eisenbarth, R. Koschke, D. Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering* 29(3):210–224, 2003.
doi:10.1109/TSE.2003.1183929

[Erl05] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[FHC01] H. Fahmy, R. C. Holt, J. R. Cordy. Wins and Losses of Algebraic Transformations of Software Architectures. In *Proc. of Int'l Conf. Automated Software Engineering (ASE)*. Pp. 51–60. IEEE Computer Society, Washington, DC, USA, 2001.
doi:10.1109/ASE.2001.989790

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[Haa07] T. Haase. Model-Driven Service Development for A-posteriori Application Integration. In *Proc. of International Conference on e-Business Engineering (ICEBE)*. Pp. 649–656. IEEE Computer Society, Washington, DC, USA, 2007.
doi:10.1109/ICEBE.2007.81

[HCM$^+$08] R. Heckel, R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos, L. Andrade. *Software Evolution*. Chapter Architectural Transformations: From Legacy to Three-tier and Services, pp. 139–170. Springer, 2008.
doi:10.1007/978-3-540-76440-3_7

[IK06] I. Ivkovic, K. Kontogiannis. A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations. In *Proc. of European Conf. Software Maintenance and Reengineering (CSMR)*. Pp. 135–144. IEEE Computer Society, Washington, DC, USA, 2006.
doi:10.1109/CSMR.2006.3

[JEM05] V. Jakobac, A. Egyed, N. Medvidovic. Improving System Understanding via Interactive, Tailorable, Source Code Analysis. In Cerioli (ed.), *Proc. of Fundamental*

Approaches to Software Engineering (FASE). LNCS 3442, pp. 253–268. Springer-Verlag, 2005.
doi:10.1007/b107062

[KG98] R. Koschke, J.-F. Girard. An Intermediate Representation for Reverse Engineering Analyses. In *Proc. of Work. Conf. Reverse Engineering (WCRE)*. Pp. 241–250. 1998.
doi:10.1109/WCRE.1998.723194

[KKNS94] N. Kiesel, P. Klein, M. Nagl, V. Schmidt. Verteilung in betriebswirtschaftlichen Anwendungen: Einige Bemerkungen von Seiten der Softwarearchitektur. In Jhnichen (ed.), *Online '94 Congress VI*. Pp. C.620.01–C.620.29. 1994.

[KWC98] R. Kazman, S. Woods, J. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Proc. of Work. Conf. Reverse Engineering (WCRE)*. Pp. 154–163. IEEE Computer Society, Washington, DC, USA, 1998.
doi:10.1109/WCRE.1998.723185

[MDJ02] T. Mens, S. Demeyer, D. Janssens. Formalising Behaviour Preserving Program Transformations. In Corradini et al. (eds.), *Proc. of Int'l Conf. Graph Transformation (ICGT)*. LNCS 2505, pp. 286–301. Springer-Verlag, 2002.
doi:10.1007/3-540-45832-8_22

[MDM04] M. Marin, A. van Deursen, L. Moonen. Identifying Aspects Using Fan-In Analysis. In *Proc. of Work. Conf. Reverse Engineering (WCRE)*. Pp. 132–141. IEEE Computer Society, Washington, DC, USA, 2004.
doi:10.1109/WCRE.2004.23

[MSRM04] A. Marcus, A. Sergeyev, V. Rajlich, J. I. Maletic. An Information Retrieval Approach to Concept Location in Source Code. In *Proc. of Work. Conf. Reverse Engineering (WCRE)*. Pp. 214–223. IEEE Computer Society, Washington, DC, USA, 2004.
doi:10.1109/WCRE.2004.10

[NPSI06] Y. V. Natis, M. Pezzini, R. W. Schulte, K. Iijima. Predicts 2007: SOA Advances. Technical report G00144445, Gartner Research, November 2006.

[SEN] SENSORIA. Software Engineering for Service-Oriented Overlay Computers.
http://sensoria.fast.de/

[Sne06] H. Sneed. Integrating legacy Software into a Service oriented Architecture. In *Proc. of European Conf. Software Maintenance and Reengineering (CSMR)*. Pp. 3–14. IEEE Computer Society, Los Alamitos, CA, USA, 2006.
doi:10.1109/CSMR.2006.28

[Tig] Tiger EMF Transformer.
http://tfs.cs.tu-berlin.de/emftrans/