Reengineering Software to Three-tier Applications and Services

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by
Carlos Manuel Pinto de Matos
Department of Computer Science
University of Leicester

October 2011

Reengineering Software to Three-tier Applications and Services

Carlos Manuel Pinto de Matos

Abstract

Driven by the need of a very demanding world, new technology arises as a way to solve problems found in practice. In the context of software, this occurs in the form of new programming paradigms, new application design methodologies, new tool support and new architectural patterns.

Newly developed systems can take advantage of recent advances and choose from a state-of-the-art portfolio of techniques, taking stock of an understanding built across the years, learning from past, good and bad, experiences. However, existing software was built in a completely different context.

Software engineering advances occur at a very fast pace, and applications are quickly seen as legacy due to a number of reasons, including difficulties to adapt to business needs, lack of integration capabilities with other systems, or general maintenance issues.

There are various approaches to address these problems depending on the requirements or major concerns. The solution can either be rewriting the applications from scratch or evolving the existing systems.

This thesis presents a methodology for systematically addressing the evolution of existing application into more modern architectures, including proposing implementations to address several classes of modernisation, with particular emphasis in reengineering towards tiered architectures and service-oriented architectures.

The methodology is based on a combination of source code pattern detection guiding the extraction of structural graph models, rule-based transformations of these models, and the generation and execution of code-level refactoring scripts to affect the actual changes to the software.

This dissertation presents the process, methodology, and tool support. Additionally, the proposed techniques are evaluated in the context of case studies, in order to allow conclusions regarding applicability, scalability, and overall benefits, both in terms of computational and human effort.

Acknowledgments

I would like to express my gratitude to Reiko Heckel. His supervision and guidance enabled me to keep advancing and get to the current milestone. His constant availability and fast responses allowed me to work at unconventional times, and were essential for me to overcome all of the obstacles throughout the years. I thank him for all his suggestions and pragmatism.

I also highly thank Rui Correia, who received me in Leicester and helped me to settle in both the city and the department of Computer Science. Not only that, we also worked together in the early stages of the methodology presented in this dissertation. I thank him both from a human, and technical point of view.

I thank Mohammad El-Ramly for his participation too. His expertise of the software reengineering field contributed to this work.

I thank Artur Boronat for his suggestions regarding my work. I believe that if it was not for me leaving Leicester shortly after he arrived, we could have collaborated even more.

José Luiz Fiadeiro also provided invaluable assistance at many levels. I thank him for his availability and assistance throughout the years.

I also thank Fer-Jan de Vries for his assistance in dealing with administrative

matters, and for his kindness.

Several other people, either through their kindness or friendship, also contributed in some way along the route: Stephan Reiff-Marganiec, Laura Bocchi, Karsten Ehrig, Hong Qing Yu "Harry", Emilio Tuosto, Rick Thomas, Fawad Qayum... I cannot recall a bad time at the department, so I must really thank everyone from staff to students.

During the first and a half year of research, I participated in the Transfer of Knowledge, Industry Academia Partnership Leg2Net (MTK1-CT-2004-003169). This European project was the starting point for my PhD registration.

My participation in another European project, the IST-FET IP SENSORIA (IST-2005-16004), also played an important part.

I also thank several colleagues at ATX Software. Some had direct participation in some aspects of my work, as Georgios Koutsoukos, Luís Andrade, João Gouveia and Bruno Conde. Others contributed in a different way, as Fernando Ramalho (who saw me as a good fit for the Leg2Net project), Miguel Antunes and Nuno Correia who both work directly with me on a day to day basis in the area of Software Reengineering, Alberto Freitas da Silva (for receiving me so well when I joined ATX in Lisbon).

Special thanks to my friend and colleague Stephen Gorton, with whom it has been a pleasure working with, and has helped me immensely at many levels.

Finally, I thank my family for all their patience, love and support and, especially my sons for allowing me to balance work and play.

Thank you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Advances in technology put pressure into software organisations or departments to use new methods and strategies for development. Change also arises in response to business requirements as there is a growing need for functional integration, system flexibility and fast time-to-market aspects. When creating systems, software engineers can take full advantage of new developments in academia and industry, such as the definition of new standards, new programming paradigms or programming language versions, new application design methodologies, improved tool support (e.g. integrated development environments (IDEs)) and new architectural patterns.

Nevertheless, organisations already have large bodies of software systems that, whilst not immediately able to take advantage of all benefits of evolution, are expected to perform at the highest level, especially when competing with newly built systems.

The area of software evolution addresses these concerns, and as new advances

arise, so does the demand for new methods to support this process, in particular where the transition towards modern architectures is concerned.

This demand has been witnessed repeatedly over the past decades, putting pressure on both academia and industry. Large scale examples include the adoption of object-oriented programming languages [80, 31], the wide acceptance and use of Web technologies [103] and specifically Service-Oriented Architectures (SOAs).

Adoption of SOA is steadily growing as a software engineering practice. In [6], the technology market research firm Gartner reports that 50% of large, newly developed applications and business processes designed during the year 2007 used service-oriented architectures to some extent, and in a more recent report estimates that, by 2013, SOA will reach mainstream adoption [101]. Additionally, SOA is the primary model for integrating cloud-based applications into the existing system portfolio, which provides another vector for adoption.

However, experience also indicates that SOA implementation initiatives rarely start from scratch. In the same report [101], Gartner states that legacy modernisation comes together with SOA to integrate legacy applications in support of new deliverables. The implication is that existing applications will be at least partly reengineered to participate in the context of service-oriented architectures. This represents a significant effort on behalf of IT departments of large organisations.

Even though software modernisation is of much interest to companies year over year, lack of action in the past (mostly due to budget or personnel constraints) is creating a greater sense of urgency, as it has hampered the ability of organizations to execute business processes efficiently and effectively [39].

With this growth in SOA adoption, and overall increase of software moderni-

sation urgency, the need for a systematic approach towards reengineering for SOA becomes ever more pressing.

One of the concerns for undergoing a migration to SOA, is that principles of service-orientation pose major challenges for such reengineering efforts. From these, the technical ones are:

1. The separation of business logic from presentation logic;
2. The loosely coupled relationship between services;
3. The coarse-grained nature of services.

The large majority of existing systems were not built with these concerns in mind and consequently considerable effort is necessary to comply to them. Moreover, there is currently no specific end-to-end methodology that defines how to transform existing source code with these properties in consideration.

1.2 Scope of research

Migration to modern architecture can impact several areas in organisations.

In particular, for evolution towards SOA, the issues that must be addressed can have several natures:

- Organisational - where business processes may have to be more well documented or adapted - typically pre-analysis activities ;
- Managerial - the methods for addressing SOA adoption regarding stakeholder involvement and managing expectations;

- Technological - the process of achieving the transition of existing systems to SOA compliance.

This dissertation focuses on the technological aspects.

In order to obtain three tier systems, it is necessary to separate business logic and user interface aspects. Moreover, to achieve SOA compliant applications, all three properties mentioned in the previous section have to be addressed. These are analysed in more detail in the following paragraphs.

1.2.1 Separation of business logic from presentation

Existing systems, as they were developed across the years, some having their lifetime spanning over different concerns for requirements and architectural aspects, frequently have characteristics that do not follow current best-practices or *de facto* standards. An example of this, is that it is common to find, mixed together in a kind of "architectural spaghetti", code fragments concerned with database access, business logic, presentation aspects and exception handling, amongst others.

Interactive programs, a large class of applications, typically exhibit this type of characteristic. They are generally state-machine based, interleaving dialogues with user input and the logic of transactions triggered by their actions. This is found in several patterns including:

- Command language - This style uses commands that must be entered by the user in a syntactically correct form. For example, a text editor following this interaction uses a command language for functions such as "SAVE", "QUIT", and other commands.

- Question / Response - This technique involves a question, usually from the computer to the user, followed by a user response.
- Form filling - This is an extension of the question/response dialogue pattern described above that allows several items of data to be requested and entered in a paper form-filling style.
- Menu driven - The menu-driven technique presents the user with a list of choices from which an action can be selected.

An overview of dialogue styles commonly found in COBOL programs is reported in [95]. Similar coding practices are found in client-server applications like those developed in Oracle Forms, Java-Swing or Visual Basic. Figure 1.1 shows an example of Java-Swing code that mixes presentation and data access aspects.

```
public void operation () {
  try {
    // Data access
    fis = new FileInputStream ("Bank.dat");
    ...
  }
  catch (Exception ex) {
    total = rows;
    if (total == 0) {
      // UI actions
      JOptionPane.showMessageDialog (null, "File is Empty.", "EmptyFile",
        JOptionPane.PLAIN_MESSAGE);
      btnEnable ();
    }
    else {
      try {
        // Data access
        fis.close();
      }
      catch (Exception exp) {
        ...
      }
    }
  }
}
```

Figure 1.1: Java code tangling UI and non-UI concerns

As it is not possible to derive services directly while business logic is tightly coupled with presentation logic, an adequate decomposition of the code is required such that “pure” business functions can then be isolated as candidate

services or service constituents. This technological dimension of reengineering constitutes an architectural transformation towards a multi-tiered architecture, and is a requirement for migrating towards SOA.

The concept of layered architectures is far from new, particularly three-tier architectures [5]. However, it is not the case that even being well established, these guarantee the desired level of separation. Hence, even tiered architecture systems can require significant effort in ensuring such a separation.

1.2.2 The loosely coupled relationship between services

Finding the right modularisation for applications has been a concern since early stages of software engineering [88], but it is a considerably difficult task. 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 [36]. Therefore, it is necessary to ensure a decomposition of different functionalities in order to provide an appropriate degree of independence.

1.2.3 The coarse-grained nature of services

Typical legacy applications consist of fine-grained elements, such as components with operations that represent logical units of work, like reading individual items of data. Object-oriented class methods are an example of such fine-grained operations. There is a fundamental mismatch between typical legacy applications and services in terms of their granularity. The notion of service, is of a different, more coarse-grained, nature. Services represent logical groupings of (possibly fine-grained) operations, work on top of larger data

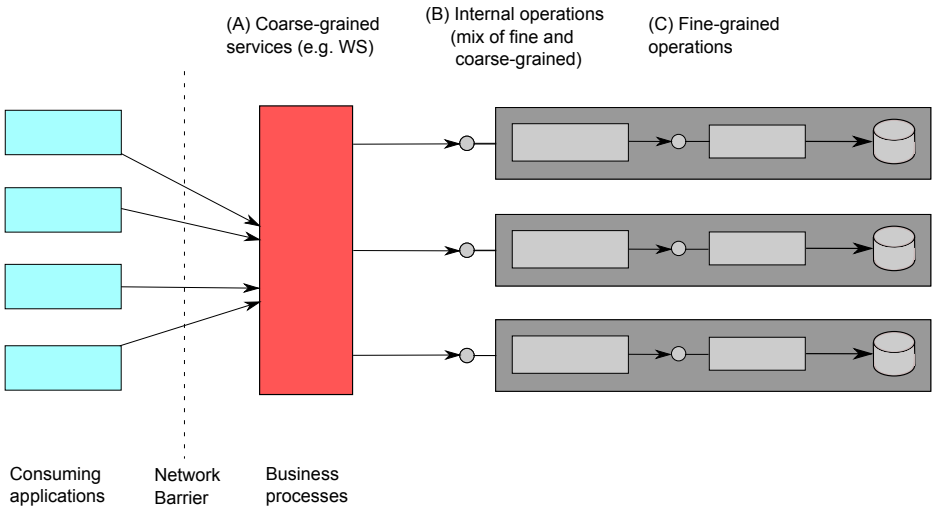

Figure 1.2: Service granularity.

sets, and in general expose a greater range of functionality. In other words, services usually aggregate lower-level features, such as commonly found in the object-oriented view, into business-level functions [87].

Concretely, the typical software service, which is consumed over a network, must comply to this coarse-grained principle in order to limit time spent in transit during requests and responses. The number of remote consumer to provider roundtrips is minimised this way, also reducing the corresponding processing cycles, as described in the context of enterprise application architecture, for instance in the pattern *Remote Facade* [41].

Fig. 1.2 presents a graphical representation of granularity across different application tiers.

In a 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.

1.3 Challenges

The lack of compliance by existing applications to the service-orientation principles stated above presents considerable obstacles to modernisation. This is valid both when the intended target for evolution is a service-oriented architecture, but also when the goal is achieving a fully compliant n-tiered application (especially for n > 2).

For the specific case of migrating towards SOA, there exist strategies based on wrapping applications into web service interfaces, however, the result of these does not bring some of the benefits that are expected when adopting a SOA, such as flexibility in reuse and evolution [96]. It is possible to achieve platform-independent access. However that comes with the cost of layering technology adapters on top of legacy applications that already have issues in terms of structuring, thus resulting in problems in runtime performance and software maintenance. In order to fully achieve SOA compliance, and in particular the properties this document focuses on, a deeper restructuring approach is necessary, where existing applications are separated into components that can then be exposed as services.

This research presents a methodology to address migration of existing software to tiered architectures and SOA, complying with the above principles, whilst allowing for a high degree of automation. Support is provided for the full reengineering cycle, consisting of an instance of the Horseshoe model of reengineering [61]. This is a conceptual model that distinguishes different levels of reengineering while providing a foundation for transformations at each level, with a focus on transformations at the architectural level. In order to structure the process, an overall methodology is proposed, instantiated in two dimensions to address both the technological and the functional evolution. The

former is concerned with the technical purpose of the code while the latter focuses on its implementation of relevant business-level functionalities. The four steps executed in each of these cycles are realised through a combination of code pattern matching, graph transformation, and refactoring-like code transformations.

Technological & Functional Dimensions

In the context of Leg2Net [69] and SENSORIA [99] projects, two dimensions were derived from the properties mentioned above: the technological and the functional dimensions. Architecture migration can involve different steps of decomposition. Depending on the intended target architecture, these are made along either the technological or functional dimensions, or both. The latter is the case of SOAs. Technological decomposition is used in the layering of software systems and may, for example, lead to a 3-tiered architecture, separating logic, data, and user interface (UI). This addresses the first SOA property mentioned in section 1.2. 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 decomposition, that can be designated as *service identification and extraction*, deals with the two last SOA properties mentioned in section 1.2.

1.4 Topic and Problem Statements

The topic of this research focuses on migration towards SOA, with the goal of developing a methodology to address both dimensions of architectural migration by providing a systematic approach and one instantiation for a particular case. The research questions are:

- What should be the general methodology that can address both technological and functional dimensions of architectural migration?
- How can presentation logic and business logic be separated?
- How can existing operations be identified and extracted from source code, while complying to the loose coupling property?
- How can the operations in a given application be combined into meaningful services from a service-oriented perspective?
- How can a methodology addressing the above three items be general enough so it can be used for a variety of projects?
- What are the properties of coverage, reuse and scalability of a specific instance of the above methodology ?

1.5 Research Strategy

The work presented in this dissertation has had several iterations, each following a common pattern. Since the steps of the general reengineering methodology defined can be addressed separately, it was possible to approach each through a process of: study of the state-of-the-art, methodological definition, application in examples, and interpretation of the results. Finally, once all steps were developed into a working prototype tool, it was possible to perform an evaluation of the approach as a whole in regards to relevant aspects that are described throughout this document.

In [100], Shaw describes various strategies for research in software engineering, focusing on combinations of research questions, results and validations that are accepted in the field.

Following the taxonomy used by Shaw, all except the last of the research questions presented in section 1.4 fall in the "method or means of development" category, although in this context it is not development in its typical sense, but reengineering. The last question belongs to the type "design, evaluation, or analysis of a particular instance". In terms of research results, the work presented in this dissertation belongs to several categories, namely: "procedure or technique" - due to the fact it describes a method to perform a task, "descriptive model" - as it presents a structure for a problem area, "notation or tool" - since a tool was implemented to embody the technique, and "answer or judgement" - as it reports on evaluation. Regarding types of validations, this research falls in the "example" type, since it presents example cases for each part of the methodology, the "evaluation" type, as case studies were used to gather data for several factors, and the "analysis" type due to the items studied in the latter.

Shaw [100] focuses on research papers, which typically produce a single type of result. This dissertation reports on PhD work and it has several result types, matching three combinations from the ones generally considered as appropriate by the software engineering community, as presented in table 1.1.

Question	Result	Validation
Development method	Procedure	Example/Evaluation
Development method	Notation or tool	Example/Evaluation
Evaluation of instance	Answer	Analysis

Table 1.1: Research strategies in this work (Shaw terminology)

1.6 Publications

In this subsection some of the relevant co-authored documents are listed. Material from these publications has been used in this dissertation since it was

developed in the context of this PhD research.

Non-refereed:

- R. Correia, C. Matos, M. El-Ramly, R. Heckel, G. Koutsoukos, and L. Andrade. Software reengineering at the architectural level: Transformation of legacy systems. Technical Report CS-06-014, Department of Computer Science, School of Mathematics and Computer Science, University of Leicester, U.K., December 2006.

  This report describes the state of the research as of October 2006. It includes initial work developed jointly by the Leg2Net project team.

- R. Correia, C. Matos, M. El-Ramly, and R. Heckel. Rule-based model extraction from source code. In S. Clarke, L. Moonen, and G. Ramalingam, editors, *Aspects For Legacy Applications*, number 06302 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

  This paper focuses on the code annotation part of the methodology as defined by the Leg2Net project team.

Peer-reviewed:

- R. Correia, C. Matos, R. Heckel, M. El-Ramly. Architecture Migration driven by Code Categorization. In F. Oquendo (ed.), *Proc. of European Conf. Software Architecture (ECSA)*. Lecture Notes in Computer Science 4758, pp. 115-122. Springer-Verlag, 2007.

  This paper presents the general methodology and focuses on the code annotation and redesign steps, representing joint work of the Leg2Net project team.

- R. Heckel, R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos and L. Andrade. *Software Evolution*. Chapter Architectural Transformations: From Legacy to Three-tier and Services, pp. 139-170. Springer-Verlag, 2008.

  This book chapter describes the general methodology and implementation aspects for the technological dimension (status as of 2007).

- C. Matos. Service Extraction from Legacy Systems. In H. Ehrig, R. Heckel, G. Rozenberg and G. Taentzer (eds.), *Proc. International Conference on Graph Transformation (ICGT 2008)*. Section: Doctoral Symposium. Lecture Notes in Computer Science vol. 5214, pp. 505-507. Springer-Verlag, 2008.

  Summary of the work done by the author of this dissertation until September 2008.

- C. Matos, R. Heckel. Migrating Legacy Systems to Service-Oriented Architectures. In A. Corradini, E. Tuosto (eds.), *Post-proceedings of the International Conference on Graph Transformation 2008 (ICGT 08)*. Vol. 16. Electronic Communications of the EASST. 2009.

  More detailed view of the PhD work up until early 2009.

- C. Matos, R. Heckel. *Rigorous Software Engineering for Service-Oriented Systems – Results of the SENSORIA project on Software Engineering for Service-Oriented Computing*. Chapter Legacy Transformations for Extracting Service Components. Springer-Verlag, 2011.

  This book chapter describes in detail the work done by the author of this dissertation up until early 2010.

1.7 Summary and Outline

After this introduction to the theme, including the motivation for the work presented here, the scope of the research, as well as an overview of how the challenges are addressed, and the research strategy followed, this dissertation is structured as follows. Chapter 2 discusses several areas of Computer Science that are central to solutions presented in the remainder of the text. This is not intended as a detailed introduction to all themes, but rather an overview. Chapter 3 presents work previously done in related areas, mostly concentrating in reverse engineering, transformation, forward engineering and migration to SOA. The general reengineering methodology followed throughout the text is described in chapter 4, whilst aspects of two of its instantiations are presented in chapters 5, for the technological dimension, and 6, for the functional dimension. Chapter 7 introduces aspects of implementations for the general methodology, and also details a concrete development of a prototype tool to address reengineering towards three-tiered applications and service-oriented architectures. This is followed by a description of the application of this prototype in the context of case studies, in chapter 8, where evaluation is presented. Finally, chapter 9 concludes on the work presented in this dissertation. Figure 1.3 depicts this structure.

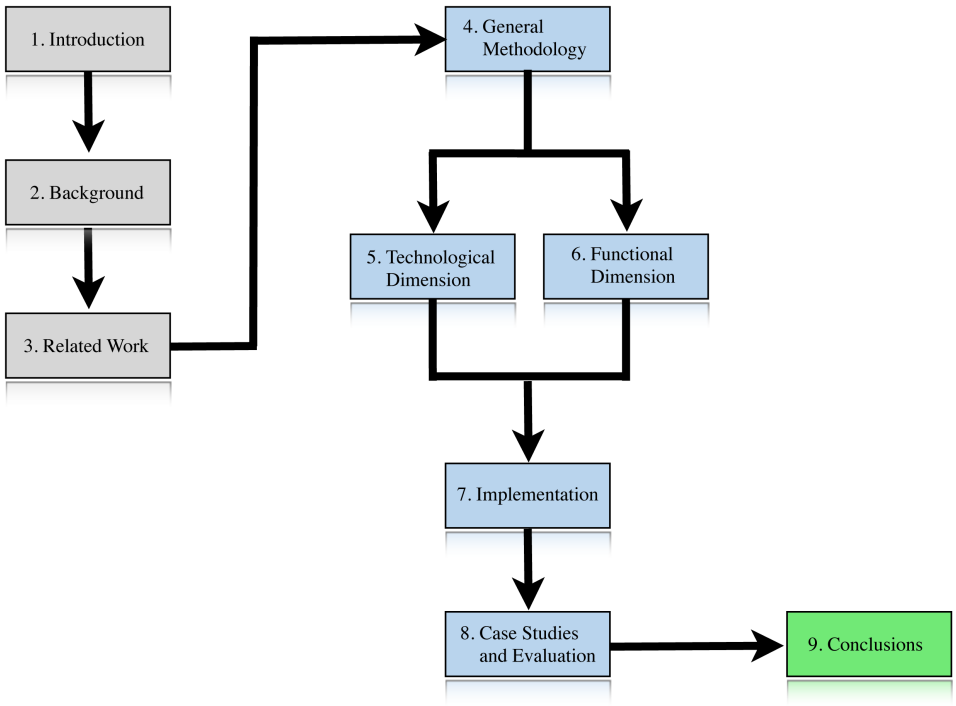

Figure 1.3: Document overview.

Chapter 2

Background

This chapter presents some concepts that are central to this dissertation. Its goal is to give a short introduction to the following relevant themes.

2.1 Software Architecture

Software architecture is a level of design concerned with the overall system structure. This is beyond the algorithms and data structures of systems, and includes the assignment of functionality to design elements, physical distribution, scaling and performance matters, amongst others [43]. A software system architecture is the set of principal design decisions about the system, representing the structural and behavioural framework on which all other aspects of the system depend. This is typically structured in terms of components, connections, constraints, and rationale.

Architectural styles define families of architectural instances in terms of structural organisation. In more detail, architectural styles determine the components and connectors that its instances follow, together with a set of constraints

that specify how these can be combined.

These include *client-server*, *pipes and filters*, *event-based*, *siloed applications*, *layered architectures*, and *service-oriented architectures*. The migration approach described in this dissertation is not specific to particular architectural styles. However, two-tier client-server and siloed architectures are two interesting cases. The former is a good candidate for being transformed to a 3-tier architecture, a sub-category of the layered architecture style. A simple illustration of these is presented in Figure 2.1. This allows for a finer separation of concerns, with all the benefits it can bring regarding maintenance and flexibility. Siloed applications consist of systems where integration or consolidation rarely occur, in which a transition to service-oriented architecture can be beneficial to take advantage of reuse and interaction with other applications.

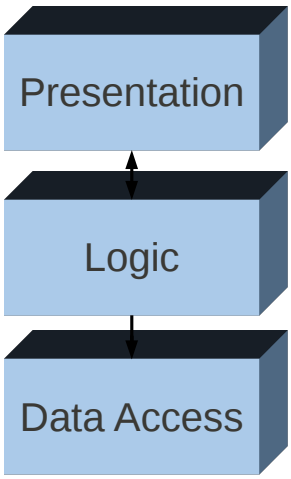

Figure 2.1: Three-tier architecture

This dissertation focuses on three tier architectures, and service-oriented architectures as a target for reengineering. The latter are described in the next section.

2.2 Service-Oriented Architectures

Service-oriented architecture is a software architecture style that encourages individual units of logic to exist autonomously but not completely isolated from each other. As stated in [36], "units of logic are still required to conform to a set of principles that allow them to evolve independently, while still maintaining a sufficient amount of commonality and standardization. Within SOA, these units of logic are known as *services*."

Service-oriented architectures combine ideas from component-based and distributed systems, adding the idea of services as loosely coupled components that may be discovered and linked at runtime.

The OASIS Consortium (Organization for the Advancement of Structured Information Standards) defines SOA as "(...) a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. (...) services are the mechanism by which needs and capabilities are brought together. " [86].

SOA is commonly described by the principles its instances should adhere to [36]. The next section lists a selection of these.

SOA Principles

Reusability: Application logic is divided into services with the intention of promoting reuse.

Composability: Services can be coordinated to form composite services.

Loose coupling: The relationship between services minimises dependencies, allowing them to evolve independently.

Interoperability: Services should be developed following standards in order to foster interoperability, thus enabling potential integration.

Abstraction: Beyond what is described in the service contract, services hide logic from the outside world.

Autonomy: Services have control over the logic they encapsulate.

Discoverability: Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.

2.3 Software Reengineering

Reengineering consists of performing analysis and modifications over existing systems [24], with several possible goals in consideration including: adding functionality, restructuring applications, performing corrections in a systematic way. Reengineering typically involves both reverse engineering and forward engineering techniques. Reverse engineering is often needed for software maintenance tasks, as it is common that the software engineers that conceived and implemented such systems originally are no longer available, and documentation is rarely up-to-date or with an adequate level of detail.

In [61], Kazman *et al* presented a metaphor for reengineering activities based on a horseshoe. Its left-hand side is seen as the process of extracting facts from existing software. The right-hand side represents development activities. The bridge between both sides constitutes transformations from the old to the new system. Figure 2.2 shows a block diagram representation of the horseshoe model.

The work presented in this dissertation is based on the horseshoe model. The methodology includes a reverse engineering step, followed by transformations

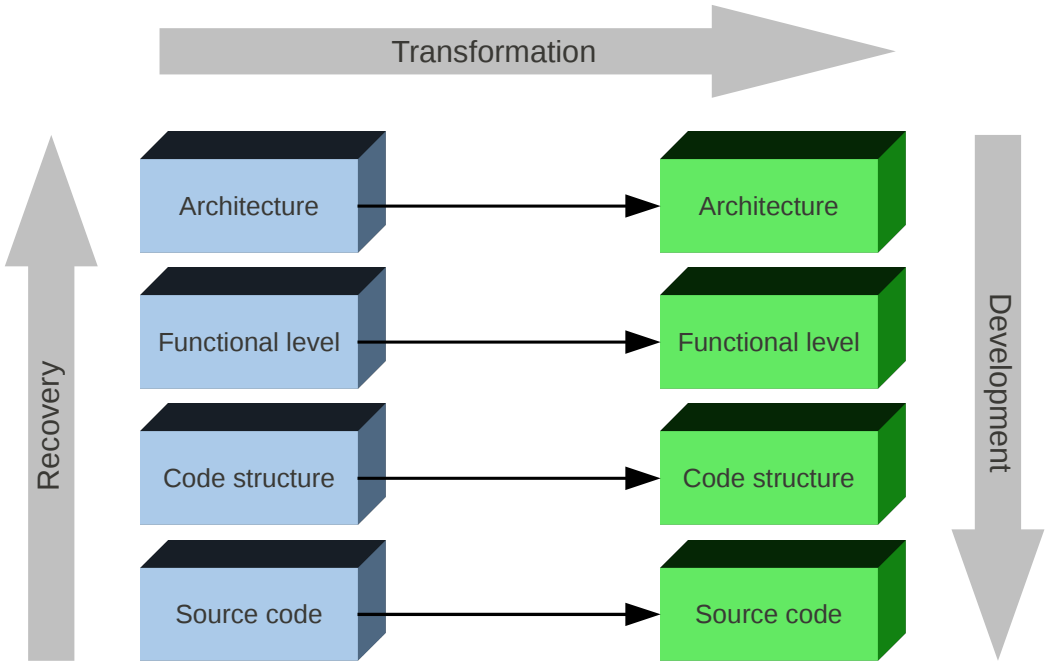

Figure 2.2: Diagrammatic view of the Horseshoe Model.

and forward engineering for obtaining the final code. These steps are preceded by a preparation activity that consists of obtaining information which is represented by code annotations. Details of the approach are given in chapter 4.

2.4 Graphs and Graph Transformation

Visual representations have long been used in the software development process [98, 63], from flowcharts to UML models. These notations produce models that can be represented as graphs, hence graph transformations are present either explicitly or implicitly when defining how models can be built or evolved [50]. Graphs provide a simple mathematical model for representing pairs of objects connected by links. More formally, a graph consists of a set of vertices V and a set of edges E, each edge having a source and a target vertex in V. Graphs can be typed, allowing for the definition of meta-models that describe how instances should be built. Additionally, in order to carry further information,

it is possible to use attributes in graphs, storing values of pre-defined data types.

Redesign by Graph Transformation

The following sections detail the formalism used to specify redesign transformations and discuss potential proof obligations for well-definedness of transformations in terms of their relevance, consequences, and support for verification.

Metamodelling with typed graphs Graphs are often used as abstract representations of models. For example in the UML specification [85] a collection of object graphs is defined by means of a metamodel as abstract syntax of UML models.

Formally, a *graph* consists of a set of vertices V and a set of edges E such that each edge e in E has a source and a target vertex $s(e)$ and $t(e)$ in V, respectively. Advanced graph models use attributed graphs [73] whose vertices and edges are decorated with textual or numerical information, as well as inheritance between node types [33, 77, 79].

In metamodelling, graphs occur at two levels: the type level (representing the metamodel) and the instance level (given by all valid object graphs). This concept can be described more generally by the concept of *typed graphs* [27], where a fixed *type graph* TG serves as abstract representation of the metamodel. Its instances are graphs equipped with a structure-preserving mapping to the type graph, formally expressed as a *graph homomorphism*. For example, the graph on the top right of Figure 2.3 is an instance of the type graph on the top left, with the mapping defined by $type(o) = C$ for each instance node $o : C$.

In order to more precisely define the class of instance graphs, constraints can be added to the type graph expressing, for example, cardinalities for in- or outgoing edges, or acyclicity. Formalizing this in a generic way, it is assumed for each type graph TG a class of constraints $Constr(TG)$ that could be imposed on its instances. A metamodel is thus represented by a type graph TG plus a set $C \subseteq Constr(TG)$ of constraints over TG. The class of instance graphs over TG is denoted by $Inst(TG)$ while writing $Inst(TG, C)$ for the subclass satisfying the constraints C. Thus, if (TG, C) represents a metamodel with constraints, an instance is an element of $Inst(TG, C)$.

The transformations described in this document implement a mapping from a general class of (potentially unstructured) systems into a more specific class of three-tier applications. This restriction is captured by two levels of constraints, global constraints C_g interpreted as requirements for the larger class of all input graphs, also serving as invariants throughout the transformation, and target constraints C_t that are required to hold for the output graphs only. Global constraints express basic well-formedness properties, like that *every code fragment is labelled by exactly one code category and part of exactly one component.* The corresponding target constraint would require that *the component containing the fragment is consistent with the code category.*

Rule-based model transformations After having defined the objects of transformation as instances of type graphs satisfying constraints, model transformations can be specified in terms of graph transformation. A *graph transformation rule* $p : L \to R$ consists of a pair of TG-typed instance graphs L, R such that the union $L \cup R$ is defined. (This means that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name are required to have the same type.) The left-hand side L represents the pre-conditions of the rule while the right-hand

side R describes the post-conditions. Their intersection $L \cap R$ represents the elements that are needed for the transformation to take place, but are not deleted or modified.

A *graph transformation* from a pre-state G to a post-state H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \to G \cup H$, called *occurrence*, such that

- $o(L) \subseteq G$ and $o(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and
- $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e., precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R.

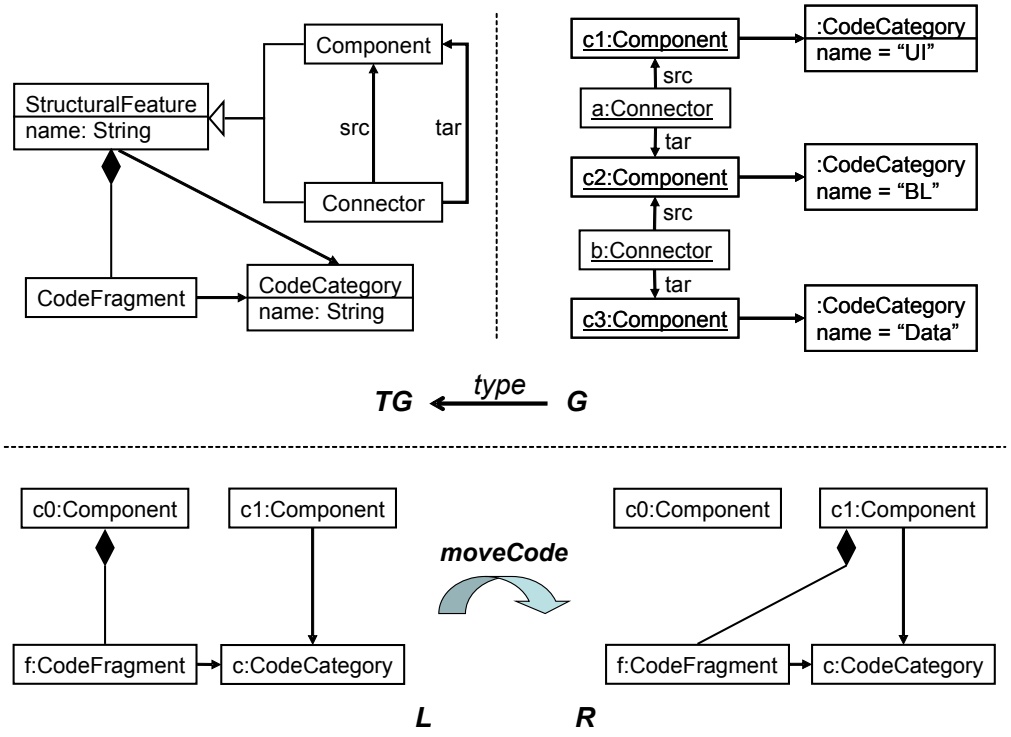

Figure 2.3: Type and instance graph (top) and transformation rule (bottom)

Rule *moveCode* in the lower part of Figure 2.3 specifies the relocation of a code

fragment (e.g. package, class, or method) from one component to another one based on its code category. Operationally, the application of a graph transformation rule is performed in three steps. First, find an occurrence of the left-hand side L in the current object graph. Second, remove all the vertices and edges which are matched by $L \setminus R$. In the present example this applies to the composition edge from *c0:Component* to *f:CodeFragment*. Third, extend the resulting graph with $R \setminus L$ to obtain the derived graph, in this case adding a composition edge from *c1:Component* to *f:CodeFragment*.

Altogether, a transformation system is specified by a four-tuple

$$\mathcal{T} = (TG, C_g, C_t, P)$$

consisting of a type graph with global and target constraints, and a set of rules P.

A sequence like s is *consistent* if all graphs G_i satisfy the global constraints C_g. $G \overset{\surd}{\Longrightarrow} H$ stands for a complete and consistent transformation sequence from G to H in $\mathcal{T}$.

Well-definedness and correctness of transformations Besides offering a high level of abstraction and a visual notation for model transformations, one advantage of graph transformations is their mathematical theory, which can be used to formulate and verify properties of specifications. Given a transformation system $\mathcal{T} = (TG, C_g, C_t, P)$ the following properties provide the ingredients for the familiar notions of partial and total correctness.

Global Consistency. All rule applications preserve the global invariants C_g, i.e., for every graph $G \in Inst(TG, C_g)$ and rule $p \in P$, $G \overset{p(o)}{\Longrightarrow} H$ implies

that $H \in Inst(TG, C_g)$.

Typical examples of global consistency conditions are cardinalities like *each Code Fragment is part of exactly one Structural Feature*. While such basic conditions can be verified statically [52], more complex ones like the (non-)existence of certain paths or cycles may have to be checked at runtime. This is only realistic if, like in the graph transformation language PROGRES [97], database technology can be employed to monitor the validity of constraints in an incremental fashion. Otherwise, runtime monitoring can be used during testing and debugging to identify the causes of failures.

Partial Correctness. Terminating transformation sequences starting out from graphs satisfying the global constraints should end in graphs satisfying the target constraints. A transformation sequence $s = (G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n)$ in $\mathcal{T}$ is terminating if there is no transformation $G_n \overset{p(o)}{\Longrightarrow} X$ extending it any further. The system is *partially correct* if, for all $G_s \in Inst(TG, C_g)$, $G_s \overset{*}{\Longrightarrow} G_t$ terminating implies that $G_t \in Inst(TG, C_t)$.

To verify partial correctness it is necessary to show that the target constraints are satisfied when none of the rules is applicable anymore. In other words, the conjunction of the negated preconditions of all rules in P and the global constraints imply the target constraints C_t. The obvious target constraint with respect to the single rule in Figure 2.3 should state that *every Code Fragment is part of a Component of the same Code Category as the Fragment*, which is obviously true if the rule is no longer applicable.

To verify such a requirement, theorem proving techniques are required which are hard to automate and computationally expensive. On the other hand, since it is only required on the target graphs of transformations, the condition can be checked on a case-by-case basis. Checking these conditions is out of the

scope of this dissertation, but a possibility for performing it would be the use of OCL (Object Constraint Language [84]) tools.

Total Correctness. Assuming partial correctness, it remains to show termination, i.e., that that there are no infinite sequences $G_0 \overset{p_1(o_1)}{\Longrightarrow} G_1 \overset{p_2(o_2)}{\Longrightarrow} G_2) \cdots$ starting out from graphs $G_0 \in Inst(TG, C_g)$ satisfying the global constraints.

Verifying termination typically requires the definition of a mapping of graphs into some well-founded ordered set (like the natural numbers), so that the mapping can be shown to be monotonously decreasing with the application of rules. Such a progress measure is difficult to determine automatically. In this simple example, it could be *the number of Code Fragments in the graph not being part of Components with the same Code Category*. This number is obviously decreasing with the application of rule *moveCode*, so that it would eventually reach a minimum (zero in this case) where the rule is no longer applicable. In [14], the subject of termination of complex transformations is discussed, and an approach to address it was proposed.

Uniqueness. Terminating and globally consistent transformation sequences starting from the same graph produce the same result, that is, for all $G \in Inst(TG, C_g)$, $G \overset{\surd}{\Longrightarrow} H_1$ and $G_s \overset{\surd}{\Longrightarrow} H_2$ implies that H_1 and H_2 are equal up to renaming of elements.

This is a property known by the name of confluence, which has been extensively studied in term rewriting [105]. It is decidable under the condition that the transformation system is terminating. The algorithm has been transferred to graph transformation systems [89] and prototypical tool support is available for part of this verification problem [104].

It is worth noting that, like with all verification problems, a major part of the effort is in the complete formal specification of the desirable properties, in this case the set of global constraints C_g and the target constraints C_t.

Relying on existing editors or parsers it may not always be necessary (for the execution of transformations) to check such conditions on input and output graphs, so the full specification of such constraints may represent an additional burden on the developer. On the other hand they provide an important and more declarative specification of the requirements for model transformations, which need to be understood (if not formalised) in order to implement them correctly and can play a role in testing model transformations.

Chapter 3

Related Work

The area of software reengineering has extensive literature as it provides many interesting problems for research and challenges for industry. Reengineering can be divided, in its fundamental form, in three basic processes: reverse engineering, transformation, and forward engineering. Another dimension of reengineering is the level at which it is performed: source code, code representation, design or architectural representation. For instance, the reverse engineering process can extract a code representation that is transformed at that level according with a certain goal, and then the final source code can be generated. This general view of reengineering was the start of the literature review process for this PhD. A summary of these reengineering concepts is presented in [61]. Next, a selection of the work studied during the literature review phase is resumed. The first three sections correspond to the basic processes of reengineering, as mentioned before. The last presents work concerning the migration to SOA.

3.1 Reverse Engineering

Regarding the area of source code representation and reverse engineering, there are different kinds of developments:

Techniques for source code analysis, in particular feature location and concept assignment, are related to the first step of the methodology described here. There are several techniques for this purpose including the work of Marcus *et al* [75] in applying latent semantic indexing (LSI) to concept location - a technique introduced in [30] as an information retrieval method, the scenario-based (SBP) feature location approach of Antoniol and Guéhéneuc [10] and the work of Eisenbarth *et al* [34] involving both static and dynamic feature location. These techniques are all candidates to be applied in the context of a SOA migration project, considering the first step of the methodology presented in this thesis. The following subsections describe in more detail some of these approaches.

3.1.1 Program Slicing

Program slicing [111] is a technique for abstracting from programs by focusing on selected points of interest. These are specified as a slicing criterion, usually consisting of a set of variables and a program location. Program slicing is the process of computing the set of program statements that affect the slicing criterion. This set of statements is what constitutes the program slice. Slicing can be used in several software engineering activities, including debugging, reengineering, testing and program comprehension.

3.1.2 Software Reconnaissance

Wilde *et al* developed a feature location technique called software reconnaissance [113, 114], which is based on dynamic analysis using carefully designed test cases. This technique consists of: instrumenting and compiling the code, executing a test suite that uses the feature that is intended to be located, executing a test suite that does not use that feature and comparing the traces. This process allows to determine feature-specific code, distinguishing several evels of involvement from code units. The technique's main intention is to provide a starting point for further analysis work.

Ibrahim *et al* report, in [55], on a very small case study of applying the RECON2 code reconnaissance technique and tool [83]. The code is in C and has only 450 lines of code. The main conclusions are that the choice of the test cases for the positive and negative situation is very important and reconnaissance techniques are helpful in reducing the manual effort.

Formal Concept Analysis

The above mentioned work of Eisenbarth *et al* [34] describes a technique to locate features (functionalities) in source code. The main notions used are scenario, feature and computational unit. The first corresponds to a sequence of inputs that lead to an observable result. The second is the implementation of a functional requirement of a system. In the context of this article, non-functional requirements are out of scope. The last is an executable part of a system. The technique is based on a combination of static and dynamic analyses as well as user (expert) input. Thus, this is a semi-automatic process to which several tools contribute. One of the most highlighted (by the authors) characteristics of the technique is the usage of formal concept analysis

(a mathematical technique for analysing binary relations [13]) to aid in the mapping of scenarios to features.

Scenario-based probabilistic (SBP) feature location

In [10], the authors present a feature location scenario-based probabilistic technique. This combines static and dynamic analysis and consists in: 1) building a model of the program; 2) identification of features providing micro-architectures (subsets of programs / similar to slices); 3) comparison between features and micro-architectures to highlight the differences. The paper is concerned with analysing C++ code. The first step is performed using a code parser to generate a model in Abstract Object Language (AOL) and by using the PADL meta-model. The second step is done by using trace collection either by processor emulation or by statistical profiling. This results in a ranked list using a probabilistic model to determine whether an event is related to a feature or not. The third step is accomplished by using model transformation techniques to highlight the differences among micro/architectures. Starting from a "source" micro-architecture, the model transformations necessary to reach a "target" micro-architecture is what is used to differentiate them.

The paper also refers to the limitations of other techniques such as the naive approach (string search "grep style") and formal concept analysis as in Eisenbarth *et al* (for example the inability to distinguish irrelevant but interleaved events). This is also shown in a case study, using a version of Mozilla code base, in which the three techniques are compared. The naive approach is criticised for being difficult to find a search that will filter the results only to the ones relevant and to not provide a relevance index. Formal concept analysis is criticised for being very difficult to inspect the concept lattice when the number of objects is high. However it is more selective and narrows the

search better than the naive approach. The authors technique performs better than the previous two since it filters even more and provides a ranking, with a relevance index, of the elements that belong to a feature.

Considerations about the use of a processor emulator vs. a statistical profiler state that while the former can increase the run time of the scenarios, the latter is subject to the effect of under sampling, thus requiring a great number of executions of the same scenario to avoid making invalid assumptions.

Combining LSI and SBP

Poshyvanyk *et al*, in [90], present a way to combine two feature location techniques: LSI (static) and SBP (dynamic). This is done by combining both rankings into one. The results of a case study presented by the authors suggest that the combination is significantly more effective that any of the isolated techniques.

Landmark Methods

In [110], Walkinshaw presents a technique for assisting a code inspector to focus on source code elements that are relevant to a particular element of functionality. Slicing can assist in this process, however, slices in object-oriented code tend to be large due to its inherent unpredictability, hence slicing alone is not suitable. In order to provide more focused information, the concept of "landmark methods" was introduced. These consist of code units that must be invoked when the intended feature is executed. This concept allows a slice-based code extraction approach that incorporates information about program execution and expressive criteria.

Other feature location techniques

In [23] the authors present a feature location technique based on dependence graphs. From the source code of a program, an abstract dependence graph (a higher level abstraction than that of the system dependence graph) is created. The user will then interact with a tool that will assist him in the graph navigation. First, a starting node is selected by the user. Then, the tool generates the dependence graph and the user selects the next node to visit. This is added to the search graph. If the user finds that the node is relevant to the feature being searched, the graph will be expanded, else he/she will select a different one for expanding and the current node will be removed from the search graph. This is done until the goal state is reached. There are several strategies for search graph expansion: top-down (from current function to the called ones), bottom-up (from the current function to the ones that call it), backward data flow (expand to the nodes (functions/global variables) that provide specific values) and forward data flow (expand to the nodes (functions/global variables) that use specific values). A case study using NCSA Mosaic is also presented. For the selected scenarios, the user had to visit only 2% of the program's functions to map the features to the code.

In [60], the authors describe a data mining strategy towards program comprehension. In summary, the code is loaded into a model and placed in a database. Then, charts are obtained showing clusters according to specific metrics such as: number of children and coupling between objects. This allows a maintainer to identify the places in the source code that are likely to be more complex to maintain (or less).

The study presented in [66] was primarily about testing the feature location technique of Eisenbarth *et al* [34] by changing the granularity from routine to

basic block level. Another change was that test cases combining more than one feature were also created to see if this does not yield additionally executed units. The conclusions were:

- dynamic feature location based on concept analysis is limited due to the combinatorial explosion of feature compositions - this will greatly increase the number of test cases to be created and executed; the authors suggest that in cases with a large number of feature compositions, a static technique should be used;

- for a smaller set of features, one can apply concept analysis even at the basic block level and this improves the information gathered due to situations where different features are implemented by the same routines.

Hypotheses and Conclusions:

H1: Statement level profiling yields more detailed results at acceptable costs

C1: True

H2: The combination of features does not yield additionally executed units

C2: False

H3: The approach scales for large feature sets

C3: False

Work presented in [74] begins by explaining the types of existing static techniques for concept location. The three identified strategies are: (textual) pattern matching (e.g. grep), dependency graph browsing (e.g. manual, FEAT) and information retrieval (e.g. LSI). The focus of the paper is the applicability of these techniques in OO code both from the necessity and effectiveness points of view. The conclusion is that OO does not dismiss concept location

and, in general, does not make this task easier. From the case study presented in the paper it seems that all three techniques are valid for addressing concept location in OO code.

In [112], Wilde *et al* present work that compares three different techniques of locating features: software reconnaissance (dynamic), dependence graph search (static) and grep-type (static). All techniques were applied to a case study that consisted of locating two features in a legacy Fortran 77 application. The conclusions were that all three methods are valid and should be present in the Software Engineers' toolkit. Software reconnaissance proved to be a fast technique that successfully located both features, however it did not help in the code understanding for the feature that had the least locally comprehensible code. Dependence graph search also successfully located the features but was noticeably the most laborious technique. However, it had the big advantage of providing the best understanding of the code. The grep-type method was the faster but was not successful in locating one of the features. The authors suggest that this method can be the first to be applied since it is very fast and, if it fails, little time is lost. In the conclusion the authors also differentiate between feature location and concept location including an example. Software reconnaissance technique is only applicable to feature location.

A technique to identify use cases in source code is presented in [117]. This technique is based in the concept of Branch-Reserving Call Graph (BRCG). It consists of a call graph that includes branches. One of the assumptions in this work is that branch statements are the only mechanism to distinguish between different use cases in a procedural language. As the BRCG may be too large for the maintainer to browse, a metric-based pruning strategy to get rid of irrelevant branches in the graph is proposed. The graph is then used to get the use cases by generating every possible execution trace from it. This is then

analysed by the maintainer who can also change the result. The modelling of the desired use cases from the set of use cases is also left to the maintainer (actors, relations between use cases). Two case studies are reported, both subjects are small systems written in ANSI C but that are very popular (namely DC and GZIP). For the first example, the technique showed promising results but for the second, some limitations arose. The main was that it was unable to distinguish the compression operation from decompression in GZIP. The authors attribute this to the fact that both functionalities are present in the same procedure and relevant branches are pruned. By changing the pruning configuration this can be fixed but at the cost of having a significantly bigger graph. This can make irrelevant branches appear and thus increasing the manual weight of this approach.

Zhao *et al* [119] put forward a feature location method that intends to solve this problem in a static and non-interactive fashion (named SNIAFL - Static Non-Interactive Approach to Feature Location). The authors claim that this reduces the human effort necessary for dynamic techniques and for traditional static approaches. Dynamic techniques need a careful, and often complex, test case selection in order for it to be effective. Traditional static techniques require, somehow intensively, human interaction to guide the process. In their work, Zhao *et al* propose SNIAFL, that is composed of the following steps: 1) acquire the initial specific connections between features and functions, by using an information retrieval (IR) technique that uses the natural language descriptions of features as queries; 2) Choose the initial specific function for each feature; 3) acquire the relevant functions and possible execution traces using a BRCG; analyse the functions to determine which are the final specific functions.

A study, over a small application (GNU DC), is made to compare the effec-

tiveness of SNIAFL against an IR-only technique and a dynamic approach. From their data, SNIAFL obtains much better results than the others, with the IR-only approach being by far the worse.

In the discussion, the authors emphasize the reduced manual weight of their approach but concede that the maintainer has to describe and retrieve all features before locating a specific one. An important drawback of SNIAFL, which is also identified by its authors, is the lack of flexibility in the granularity. It is not possible to define a lower granularity than that of functions and there are circumstances where this is necessary. Another limitation is that this approach is highly dependent on the quality of the identifier names in the source code and on the distance between the terms used in the natural language description of features and the naming used in the code.

Concept assignment

In [12], Biggerstaff *et al* present the concept assignment problem (also known as concept location) and present several strategies to address it. The problem is identified as a complex one that requires a collaboration between automatic processes and human input. A key idea is that "a parsing-oriented recognition approach based on formal, predominantly structural patterns of programming language features is necessary but not sufficient for solving the general concept assignment problem". This is due to the fact that "the signatures of most human-oriented concepts are not constrained in ways that are convenient to parsing technologies". The authors claim that two general tasks are required when addressing concept assignment: 1. identify the entities and relations that are most relevant in the program; 2. assign them to domain concepts and relations. The former is generic while the latter relies in domain knowledge. The authors then provide an example and explain how it can be tackled using

three different strategies - all using the DESIRE tool suite for automated support to analyse C language programs:

1. Suggestive data names as first clue

The user first identifies the mapping between some parts of the code to concepts by its naming proximity. Then, by analysing the control flow, more code blocks can be also identified. To complete this, slicing techniques can help in identifying the concept's remaining relevant code.

2. Patterns of relationships as first clue

Following this strategy, clusters of related functions and data are identified forming an abstract architecture or framework of the program. This can be achieved using generic rules, for example "functions that are coupled by shared global variables". The clusters can then be assigned by the user to concepts.

3. Intelligent agent provides first clue

A tool can scan the code and provide a list of candidate concepts. This can be based on a domain model database. DESIRE's DM-TAO achieves this by using a semantic/"connectionist" hybrid network in which each concept is represented as a node and the relationships are explicit links. The nodes have different types such as: concept node, feature node, and term node. The links have weights that are used to give clues to the process. The user could then start from a simplified stage in which he would correct the automated result if necessary.

In [19], Carey and Gannod present a technique to filter class models from OO systems to display only the ones relevant to concepts. The authors assume that "Object-oriented design suggests that we ignore the details of the implementation of a class, so we believe that analysing object-oriented software at

the class level is a valid approach to solving the concept assignment problem". This is possibly an over-generalisation since many systems, even the ones that are developed in the beginning following all OO best practices, end up mixing concepts in one class. The methodology is based on the application of statistical machine learning algorithms using OO metrics. A manual classification is made, indicating for a set of classes which represent a concept and which do not. The learning algorithm uses the manual information to propagate the classification to the remaining classes. The results of the two test cases presented in the paper seem promising although several shortcomings are evident. The lower granularity of the process being the class level makes it very limited in situations where there are multiple concepts in the same class or where a concept is scattered among multiple classes. Other limitations are related to the statistical nature of the technique such as: errors in the statistical analysis and overfitting of the data set (i.e. results do not generalise well to other data sets).

Combining program slicing and concept assignment

In [44], Gold *et al* present a higher-level executable source code extraction technique based on unifying program slicing and concept assignment. Its goal is to overcome limitations of slicing, which requires low level criterion, and concept assignment, which does not result in executable code.

In the above article, the authors describe four techniques of "concept slicing" which differ in the way that program slicing and concept assignment are combined and explain the contexts in which each is more beneficial. The presented case studies shows a significant reduction of the amount of source code that needs to be inspected for a code extraction task.

Aspect mining

In [21], the authors present and compare three aspect mining techniques: fan-in, identifier analysis and dynamic analysis. Fan-in is based in method call count, identifier analysis uses the names of methods to perform formal concept analysis and dynamic analysis uses traces of execution also to perform concept analysis. A case study (with JHotDraw) showed that they are somewhat complementary. In a combination study trying to locate four crosscutting concerns, the joint technique worked better to find three but worse in one.

Obtaining a higher-level representation

The Dagstuhl Middle Model (DMM) [70] was developed to solve interoperability issues of reverse engineering tools. Like the approach presented in this dissertation, it keeps traceability to the source code. The DMM is composed by sub-hierarchies that include an abstract view of the program and a source code model. The chosen way to relate these two is via a direct link. The Fujaba (From UML to Java And Back Again) tool suite [59] provides design pattern [42] recognition. The source code representation used for that process is based on an Abstract Syntax Graph (ASG). Another representation is put forward with the Columbus Schema for C++ [38]. Here an AST conforming to the C++ model/schema is built, and a higher level semantic information is derived from types. The work of Ramalingam *et al*, from IBM research, in [91], addresses the reverse engineering of OO data models from programs written in weakly-typed languages like COBOL. In their work, the links between the model and the code are represented in a reference table. This table establishes the link between each model element and the line of code having no intermediate representation. A major difference between the methodology presented in

this dissertation and the above approaches is that it uses a categorisation step that will make possible the automated transformation to a new architectural style. The ARTISAn framework, described by Jakobac, Egyed and Medvidovic in [57], categorises source code. It uses an iterative user-guided method to achieve this. The code categories used are: "processing", "data" and "communication". The approach differs from the one presented in this dissertation 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 categorisation process (called "labeling") is based in clues that result in the categorisation of classes only. For the approach presented in this dissertation there is the need, and support, for the method and code block granularity levels.

3.2 Transformation

Program transformation can occur at different levels of abstraction. The source-to-source level of transformation is the most established one. There are several approaches that led to successful industrial tools. Examples from research include TXL [26] and ASF+SDF [107]. DMS from Semantic Designs [11] and Forms2Net from ATX Software [9] are program transformation tools being successfully applied in the industry. Transformations at the detailed design level, due to its applications as maintenance techniques, have an increasing interest that is following the same path. Practices such as refactoring [40] are driving the implementation of functionalities that automate detailed design level transformations. These are mainly integrated in development environments as is the case of Eclipse [106] and IntelliJ [58]. However,

there is still a lot of ongoing research in this area, for instance, in the determination of dependencies between transformations [78].

Work in the area of architecture transformation is broad and diverse. It includes a few works based on model transformation, automated code transformation, or graph transformation and re-writing, which are closely related to the work in this document. The approaches found in the literature vary in three main aspects: first, the levels of abstraction used for describing the system (architecture models only or interlinked architecture and implementation models), second, the way the architecture models are represented and third, the method and tools used for representing and executing architecture transformation rules. Available case studies are either only concerned with the transformation of high level architecture representations or limited to very specific source and target architectures and programming languages combinations.

The use of graph transformation for reengineering software systems has been previously suggested [28] in a different context (migrating mainframe COBOL to client/server) with similarities to the technological dimension later discussed in this document in chapter 5. One of the main differences is that the latter focuses manual involvement in the initial step, which, amongst others, has the advantages of limiting the type of necessary human activities (with a focus on analysis and not in corrections to the results of various steps), and also of scalability as detailed later in this dissertation. A model-based technique based on graph transformation for *a posteriori* integration of legacy applications into SOA is proposed in [48]. This focuses on generating wrappers and glue code, rather than directly transforming the source code. Although the approach presented in this dissertation makes use of models, other more model-centric, MDE based approaches such as [115], mainly differ from it since it gives much

focus at the code annotation step at a lower level of abstraction before moving to the graph model level, in order to obtain the benefits mentioned in this document.

Kong *et al* [64] developed an approach for software architecture verification and transformation based on graph grammar. First, the approach requires translating UML diagrams describing the system architecture (or acquiring a description for it) to reserved graph grammar formalism (RGG). Then, the properties of the RGG description can be checked automatically. Automatic transformation can also be applied but only at the architecture description level and not at the implementation level.

Ivkovic and Kontogiannis [56] proposed a framework for quality-driven software architecture refactoring using model transformations and semantic annotations. In this method, first, a conceptual architecture view is represented as a UML 2.0 profile with corresponding stereotypes. Second, instantiated architecture models are annotated using elements of the refactoring context, including soft-goals, metrics, and constraints. A generic refactoring context is defined using UML 2.0 profiles that includes "semanticHead" stereotype for denoting the semantic annotations. These semantic annotations are related to system quality improvements. Finally, the actions that are most suitable for the given refactoring context are applied after being selected from a set of possible refactorings. Transformations in this method occur at the conceptual architecture view level using Fowler [40] refactorings.

Fahmy *et al* [37] used graph rewriting to specify architectural transformations. The authors used PROGRES tool [15] to formulate executable graph-rewriting specifications for various architectural transformations. Architecture is represented using directed typed graphs that describe system hierarchy and component interaction. The assumption is that the architecture is extracted using

some extraction tool. Their work is at the architecture description level and no actual transformation is performed on the code, unlike the approach presented in this thesis. In the latter, a graph is built such that it models the software but also maps the code to target architectural elements. It is this information that guides the redesign process, and allows for obtaining the target code.

Unlike the three previous works, the approach of Carrière *et al* [20] implements architectural transformations at the code level using automated code transformation. Their first step is reconstructing the existing software architecture by extracting architecturally important features from the code and aggregating the extracted (low-level) information into an architectural representation. The next step is defining the required transformations. In this work, they were interested in transforming the connectors of a client-server application to separate the client and server sides as much as possible and reduce their mutual dependence. Next, the Reasoning SDK (formerly Refine/C), which provides an environment for language definition, parsing and syntax tree querying and transformation, is used to implement the required connector transformations at code level on the AST of the source system. The major difference from the work described in this dissertation relies on the fact that it uses code categorization to relate the original source code to the intended target architecture. Additionally, transformation is performed at model level allowing for describing transformations in a more intuitive way and adaptable to different targets, while in the approach from [20] it occurs only at code level.

3.3 Forward Engineering

Regarding code generation, there is a significant number of research work and tools available. A comprehensive list is too long to specify here, so only some of

the most commonly used ones are described. The already mentioned Fujaba tool suite supports the generation of Java source code from the design in UML resulting in an executable prototype. The Eclipse Modeling Framework (EMF) [32] can generate Java code from models defined using the Ecore metamodel. This has a number of possible uses such as to help develop an editor for a specific type of model. UModel [8], from Altova, can generate C# and Java source code from UML class or component diagrams. In the Code Generation Network website there is a very extensive list of available tools [81].

3.4 Reengineering to SOA

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 [62]. Work in the area of reengineering to SOA primarily focuses on identifying and extracting services from legacy code bases and then wrapping them for deployment on an SOA. A key assumption in this area is that an evaluation of the legacy system will be conducted to assess if there are valuable reusable and reliable functionalities embedded that are meaningful and useful to be exposed in the service-oriented environment and that are fairly maintainable. Some research focuses on service identification only. For example, Del Grosso *et al* [45] proposed an approach to identify, from database-oriented applications, pieces of functionality to be potentially exported as services. The identification is performed by clustering queries dynamically extracted by observing interactions between the application and the database, through formal concept analysis. Unlike the above, the methodology described in this dissertation consists of an end-to-end reengineering process, and identification uses multiple sources.

The following sections describe several approaches to achieve SOA, including a classification that assists in determining the relevant differences amongst them.

The approach presented in this document is different from other approaches, in that its goal is not just to provide existing functionality as services, but to do so whilst complying to the service-orientation principles described in chapter 1.

General strategy

As in other reengineering strategies, migration to SOA is based on three basic steps:

- Reverse engineering: analysis of existing system, identification of potential services;
- Transformation: restructuring to comply to service-oriented principles;
- Forward engineering: obtaining the original functionality as services.

Classification

Service-oriented architectures were introduced some years ago, but it is an area where there is still intensive ongoing research. Although there are several approaches to migrate to SOA, they differ in several aspects. These methodologies/strategies can be classified according to properties that assist in their understanding and selection for a given purpose. A classification is presented in this section. A systematic review of the field was reported in [94, 93], where existing approaches were organised in families. These consist of groups that are distinguished in terms of scope (coverage) regarding the three steps men-

tioned above, and the levels in which they operate: concept, composite design element, basic design element and code. Whilst this classification is useful to get an understanding of the available approaches, there are other important aspects that are given emphasis in this dissertation. In this context, particular relevance is given to applicability in practice, adequacy to SOA principles and scalability. The following paragraphs describe the aspects in which this classification is applied.

Type

The current approaches of reengineering to SOA can be divided into four types:

- Methodology-only approaches - the output of the strategy is a set of guidelines/procedures or other documentation to support the migration project;
- Replacement - the existing applications are replaced by SOA compliant ones;
- Code wrapping strategies - the existing code is wrapped to provide services without too many internal changes;
- Code restructuring strategies - the existing code is changed to comply to SOA properties.

Methodology-only approaches focus on high level definitions, which include macro-plans, guidelines and documentation to organise the project, but require the use of a concrete wrapping or restructuring approach in order to reach its goal.

Replacement approaches represent the complete replacement of existing applications, or application modules, with SOA enabled ones. This can be achieved

either by redeveloping existing functionality from scratch or by acquiring, and customizing, off-the-shelf packages when available. Both possibilities require a deep knowledge of the current system, and involve significant investment in re-documentation and testing in order to mitigate the risk of losing important business rules.

Wrapping strategies use integration technology in order to act as a services layer over existing applications. Current functionality is offered as services, but the legacy code is typically left untouched. Consequently this leads to a one-to-one mapping between existing operations, and published services. This approach can be more elaborate, and involve writing new coordination code, usually in a more modern programming language, which then interacts with the legacy modules. However, this clearly diminishes benefits in terms of effort and cost. Wrapping methods cannot comply to all the properties mentioned earlier. In particular, a correct separation of presentation logic and business logic is not usually achieved. It is also not possible to guarantee that resulting services will be loosely coupled. Service granularity can be addressed to some extent. These methods are usually seen as temporary solutions.

Restructuring strategies, also known as white-box or invasive reengineering, take advantage of existing code, by restructuring it in order to enable its usage in the context of a SOA. Restructuring can take place at several levels. In the work presented in this dissertation, the requirements are to take into consideration the service-orientation principles presented in chapter 1:

1. The separation of business logic from presentation logic;
2. The loosely coupled relationship between services;
3. The coarse-grained nature of services.

This allows for a high degree of control of the final services infrastructure, as it enables a correct separation of application concerns, as well as the adequate selection of operation granularity. Although it is not the focus of this dissertation, restructuring also allows migration to a more modern technology.

Restructuring methods can address all the properties mentioned above. However, they are almost always more complex and time consuming.

Scope

Another classification can be done based on the scope of each strategy. While some offer full coverage (i.e. from the original source code to the final one, SOA compliant), others only address specific parts of the process.

Automation

The automation of the methodologies also varies. Steps can be fully automated, semi-automated or manual. There is no fully automated method to reengineer systems to SOA.

Technology

Some methodologies are technology independent, whilst others focus on specific programming languages or paradigms.

Existing approaches

This section classifies and summarises a number of existing SOA migration approaches. The goal is not to be exhaustive, but to provide examples of

different types of approaches according to the above classification aspects.

Table 3.1 lists SOA migration approaches as referred above. Note that the last approach on the table, there named "SENSORIA Reengineering Approach" as it was presented in its early stages during the SENSORIA project, is the one presented as the general approach for this thesis. Whilst initially launched by several authors, including this thesis' author, it was then carried forward and developed by him, focussing in its refinement, further progressing the technological dimension, defining the approach for the functional dimension, developing the prototype, and performing the evaluation.

The indications given in the column "Scope & Automation" refer to each reengineering step, and have the following meaning:

(Blank) - No support is given for the step;

M - Manual - The approach identifies how a step can be performed, but provides no automation;

S - Semi-automated - Semi-automated support is used for the step;

A - Automated - Full (or *quasi*-full) automated support exists for the step.

Some of the approaches listed and classified in that table are detailed in this section in order to provide representative examples of the various types. Additionally, the approaches closest to the one presented in this dissertation are also detailed for comparison. This similarity is determined in terms of the satisfaction of the requirements discussed in this dissertation, which include providing support to the full reengineering cycle, and code restructuring.

Approach	Type	Scope & Automation			Technology
		RE	TR	FW	
Service-Oriented Software Reengineering: SoSR [25]	Methodology-only	M		M	Independent
Service-Oriented Migration and Reuse Technique (SMART) [71]	Methodology-only	M			Independent
Composition and Customization of Web Services Using Wrappers: A Formal Approach Based on CSP [49]	Methodology-only			M	Independent
Extracting Reusable Object-Oriented Legacy Code Segments with Combined Formal Concept Analysis and Slicing Techniques for Service Integration [118]	Wrapping	S		M	Object-oriented
Creating Web Services from Legacy Host Programs [103]	Wrapping	M		S	Several
Integrating legacy Software into a Service oriented Architecture [102]	Wrapping	S		S	PL/I, COBOL, and C/C++.
Migrating Interactive Legacy Systems To Web Services [17, 18]	Wrapping	S		S	Interactive systems
Using Grid Technologies for Web-Enabling Legacy Systems [16]	Wrapping			S	
An Architecture Model for Dynamically Converting Components into Web Services [68]	Wrapping			S	Java and C++
A Black-Box Strategy to Migrate GUI-Based Legacy Systems to Web Services [116]	Wrapping			M	Interactive systems
Legacy Systems Interaction Reengineering [35]	Wrapping	S		S	GUI based systems
Wrapping Client-Server Application to Web Services for Internet Computing [47]	Wrapping	S		S	.NET client-server
A Lightweight Approach to Partially Reuse Existing Component-Based System in Service-Oriented Environment [54]	Wrapping	S			EJB-based applications
Web-based specification and integration of legacy services [120]	Wrapping			S	C++
Feature Analysis for Service-Oriented Reengineering [22]	Wrapping	S		S	.NET
Reengineering Legacy Systems with RESTful Web Service [72]	Wrapping	M		A	
A Case Study on Software Evolution towards Service-Oriented Architecture [29]	Restructuring	S	M	M	Java
Towards Applying Model-Transformations and -Queries for SOA-Migration [53]	Restructuring	S	S	S	Java
"SENSORIA Reengineering Approach" [51]	Restructuring	S	A	A	Independent

Table 3.1: SOA migration approaches

Title	Service-Oriented Software Reengineering (SoSR) [25]
Author(s)	Chung, An, Davalos
Method	Methodology-only
Coverage	Full
Automated	-
Semi-automated	-
Manual	Forward Engineering, Reverse Engineering
Technology	Independent

SoSR is a methodological approach that provides a set of guidelines for the reengineering to SOA process. It supports the distribution of tasks to roles in the context of a migration project.

A summary of the process can be seen in the diagram presented in Figure 3.1.

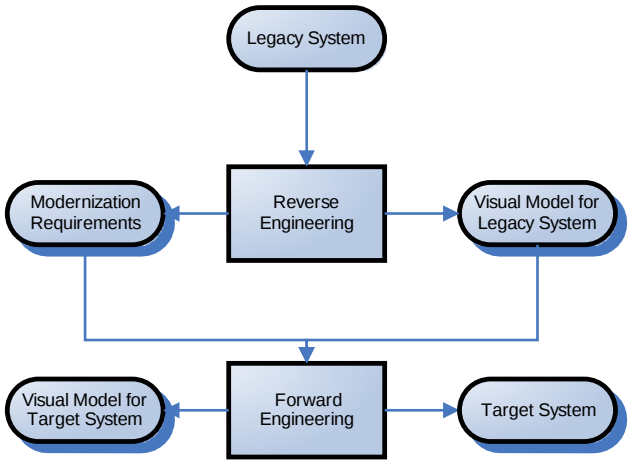

Figure 3.1: Service-Oriented Software Reengineering (SoSR) process.

Title	Service-Oriented Migration and Reuse Technique (SMART) [71]
Author(s)	Lewis, Morris, Smith, OBrien
Method	Methodology-only
Coverage	Reverse Engineering
Automated	-
Semi-automated	-
Manual	Reverse Engineering
Technology	Independent

SMART gathers a wide range of information about legacy components, the target SOA, and potential services to produce a service migration strategy as its primary product.

A summary of the process can be seen in Figure 3.2.

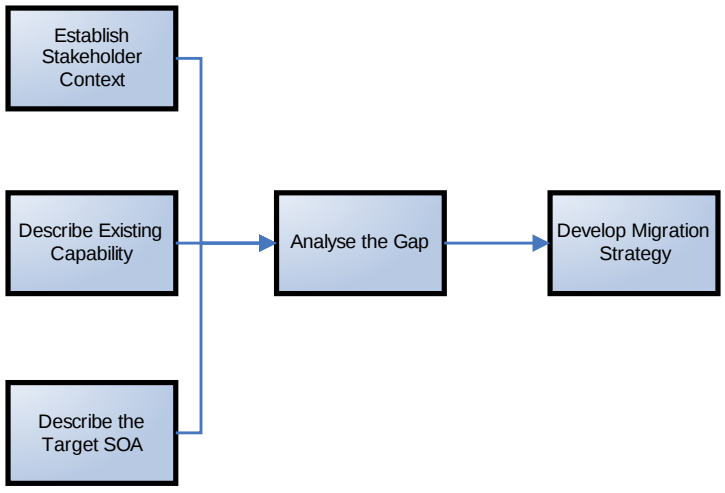

Figure 3.2: Service-Oriented Migration and Reuse Technique (SMART) process.

Title	Extracting Reusable Object-Oriented Legacy Code Segments with Combined Formal Concept Analysis and Slicing Techniques for Service Integration [118]
Author(s)	Zhang, Yang, Chu
Method	Wrapping
Coverage	Full
Automated	-
Semi-automated	Reverse Engineering
Manual	Forward Engineering (wrapping)
Technology	OO

To identify the legacy code to be reused, two reverse engineering/program comprehension techniques are used:

- Formal Concept Analysis (FCA);
- Slicing.

Formal concept analysis is a field of applied mathematics which deals with the study of the relation between elements and element properties to identify conceptual structures among data sets. FCA can be used in software engineering to group code that has similar properties. Program slicing is a well known code analysis technique that can be used to identify parts of the program that are influenced by or influence a given set of program points.

Zhang *et al* [118] proposed an approach for extracting reusable object-oriented legacy code segments with combined formal concept analysis and slicing techniques for service integration. Firstly, an evaluation of legacy systems is performed to confirm the applicability of this approach and to determine other re-engineering activities. Secondly, the legacy system is decomposed into com-

ponent candidates via formal concept analysis. Static program slicing is applied to further understand these component candidates. Finally, component candidates are extracted, refined and encapsulated.

A summary of the process can be seen in Figure 3.3.

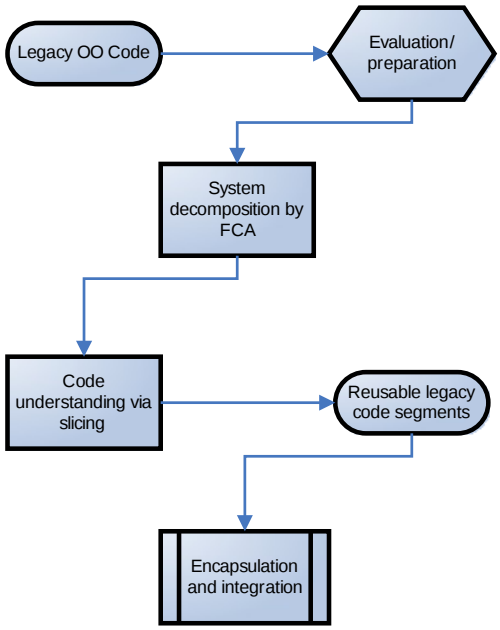

Figure 3.3: Service-Oriented Reengineering (SOR) process.

Title	Creating Web Services from Legacy Host Programs [103]
Author(s)	Sneed (H.), Sneed (S.)
Method	Wrapping
Coverage	Full
Automated	-
Semi-automated	Forward Engineering (extracting+wrapping)
Manual	Reverse Engineering
Technology	Assembly, PL/I, C or COBOL server-side and Java client-side

The process consists of seven steps:

1. Function mining: Functions to be reused are marked and extracted from the existing program together with the data they use;
2. Wrapping: A new interface is created for each of the extracted functions;
3. XML generation: From the interface description in the original language of the program, an XML subschema is generated;
4. Generation of the server stubs for parsing the incoming XML messages and for marshalling the outgoing messages;
5. Client transformation: From the XML subschema the Java classes are generated to send and receive the XML messages;
6. Server linking: The stubs are linked together with their server functions on the host to create DLLs;
7. Client build: The generated Java classes are built into the Java interface component for connecting the website to the web services on the host.

Title	Integrating legacy Software into a Service oriented Architecture [102]
Author(s)	Sneed
Method	Wrapping
Coverage	Full
Automated	-
Semi-automated	Reverse Engineering, Forward Engineering
Manual	-
Technology	

Sneed [102] presents a tool supported method for wrapping legacy PL/I, COBOL, and C/C++ code behind an XML shell which allows individual functions within the programs to be offered as web services to any external user. The

first step consists of identifying candidate functionality for wrapping as a web service. In the second step, the functionality's code is located with the aid of reverse engineering tools. In the third step, that code is extracted and re-assembled it as a separate module with its own interface. This is done by copying the impacted code units into a common framework and by placing all of the data objects they refer to into a common data interface. In the fourth step, extracted components are wrapped with a WSDL interface. The fifth and final step consists of linking the web service to overlying business processes by means of a proxy component.

Title	Migrating Interactive Legacy Systems To Web Services [17, 18]
Author(s)	Canfora, Fasolino, Frattolillo, Tramontana
Method	Wrapping
Coverage	Full
Automated	-
Semi-automated	Reverse Engineering, Forward Engineering
Manual	-
Technology	Interactive systems

A lighter code-independent approach was developed by Canfora *et al* [17], which wraps only the presentation layer of legacy form-based user interfaces (and not the code) as services. In form-based user interfaces, the flow of data between the system and the user is described by a sequence of query/response interactions or forms with fixed screen organization. Their wrapper interacts with the legacy system as though it were a user, with the help of a Finite State Automata (FSA) that describes the interaction between the user and the legacy system. Each use case of the legacy system is described by an FSA and is reengineered to a web service. The FSA states correspond to the legacy screens

and the transitions correspond to the user actions performed on the screen to move to another screen. The wrapper derives the execution of the uses cases on the legacy system by providing it with the needed flow of data and commands using the FSA of the relevant use case. Of particular relevance to the work described in this dissertation is the service identification and extraction task, which is closely related to the functional dimension presented in this document. This task is essential for any code-wrapping approach to reengineering to SOA.

Title	A Case Study on Software Evolution towards Service-Oriented Architecture [29]
Author(s)	Cuadrado, García, Dueñas, Parada
Method	Restructuring
Coverage	Full
Automated	-
Semi-automated	Reverse Engineering
Manual	Transformation, Forward Engineering
Technology	

This approach, also of the restructuring type as the one presented in this dissertation, focuses on architecture recovery. The cited paper presents the overall method in a high level fashion, and describes a case study. In the latter, the migration of a small Java application, architecture recovery was performed via: (1) information extraction (from documentation), (2) static view extraction, (3) dynamic view extraction and (4) abstraction. Tasks (1) and (4) were mostly performed manually, whilst (2) and (3) benefited of some level of tool support. The subsequent steps to obtain the final result were performed manually through iterative refactoring of the original application to match the intended final architecture. The closest aspects with the work presented in this thesis are the focus on gathering information, and the goals

of obtaining a final result that complies with the service-orientation principles. The biggest differences are that the approach from this dissertation offers more detail in terms of methodology, and defines support and automation for the transformation and forward engineering steps.

Title	Towards Applying Model-Transformations and -Queries for SOA-Migration [53]
Author(s)	Horn, Fuhr, Winter
Method	Restructuring
Coverage	Full
Automated	-
Semi-automated	Reverse Engineering, Transformation, Forward Engineering
Manual	-
Technology	

In this approach, the source code is represented by means of a graph, in order to enable graph querying and transformation techniques. This approach is model-driven, and the final code is obtained through generation from the graph. The approach allows selecting the code sets that have been identified as belonging to specific functionalities, and its adaptation in order to fit service specification can be done semi-automatically. This approach has several similarities with the one presented in this dissertation, including usage of graphs, and the goal of restructuring code in order to avoid the downsides of wrapping techniques. However, there are significant differences too. Most notably, in the methodology described in this thesis, the graphs are much more succinct due to a preparatory analysis step, and also the transformations can act at a lower level if necessary, leading to a deeper restructuring. In the case of Horn *et al*'s approach, the extraction uses the granularity of classes, whilst in the approach

from this dissertation the transformations occur also at the method and even code fragment level.

Title	"SENSORIA Reengineering Approach" [51]
Author(s)	Matos, Correia, Heckel, El-Ramly, Koutsoukos, Andrade
Method	Restructuring
Coverage	Full
Automated	Transformation, Forward Engineering
Semi-automated	Reverse Engineering
Manual	-
Technology	Independent

This approach is the one this dissertation is focused in. It performs transformations into four steps. The first step is a preparatory one, where code is annotated. The following steps consist of deriving a graph model representing the annotated code, applying changes to the model through graph transformations, and obtaining the final code via forward engineering.

This approach addresses SOA properties in two dimensions: the technological dimension is concerned with the layering of the system architecture, providing the needed separation between user interface and business logic; the functional dimension addresses the extraction of services according to the loose coupling and coarse-granularity properties.

To change source code to be SOA compliant, the restructuring along both dimensions is applied in sequence. These dimensions have some common points, one of which is the general approach.

Details of this approach are not given in this section since they are the subject of thorough discussion throughout the whole thesis report, particularly from

chapter 4 onwards.

Selecting an approach type

The decision of which approach to adopt for a given migration project deeply depends upon its specific requirements, the nature of the system to migrate, and the organisation's global IT context. Also, it is possible to even select a hybrid approach, using several of the methods presented above. This can be done in parallel, having several subsystems being migrated following different strategies, or a sequential arrangement. A typical example of the latter is followed by organisations that require a quick move to service-orientation, thus initially choosing wrapping as a temporary result, but want the benefits of a deep restructuring, thus subsequently following this approach as a long-term solution. A summary of advantages and disadvantages of each approach is presented in Table 3.2, which is an adaptation from the one presented in [67].

Considerations

One of the goals of the approach described in this dissertation is to provide means for application in practice, going all the way from the original source code to the final one, and being possible to apply systematically. Methodology-only approaches, by definition, do not provide enough details to guarantee this. They are focused in setting best practices, and define procedures, usually away from technical details. In order to fully realise a migration to SOA, they require the use of a concrete wrapping or restructuring approach to support the methodological side.

The previous section provides a differentiation between wrapping and restruc-

Approach	Advantages	Disadvantages
Replacement	- Good granularity and reuse - Adoption of modern technology	- Maintenance, Performance, Scalability problems - Higher risk (e. g. loss of crucial business knowledge or even project failure)
Wrapping	- Fast - Lower cost - Lower risk	- Poor reuse - No long term agility - Poor services granularity - Maintenance, Performance, Scalability problems
Restructuring	- Better maintainability, scalability, performance and long-term agility - Service granularity and reuse better that wrapping - Real multi-party and business process oriented architecture	- More expensive than wrapping - Time consuming

Table 3.2: SOA-enabling approaches: advantages and disadvantages

turing approaches, where it is possible to verify that the former do not satisfy several of the requirements defined in this dissertation. Whilst they enable a transition from existing applications to services, they do so without complying to service-orientation principles. Without this, it is not possible to get all the benefits of SOAs.

To achieve the requirements described in chapter 1, it is necessary to use a restructuring approach. In table 3.1, three of these were identified, including the one described in this dissertation. Whilst similar to the approach presented in this thesis in terms of the importance given to preparatory tasks of information gathering, the work described by Cuadrado *et al* in [29] is very focused on a particular case. Additionally, automation was only used for a few of the tasks. The approach from this thesis includes a more detailed methodology, and defines a higher level of support and automation for the transformation

and forward engineering steps. The approach described by Horn *et al* in [53] bears more similarity with the one presented in this dissertation. This includes usage of graphs, and code restructuring to allow SOA compliance. However, the approach from this thesis, due to its preparatory analysis step, achieves succinct graphs after the reverse engineering step, whilst in Horn *et al*'s case, all the applications language elements are represented in their graphs. In the former, transformations can also act at a lower level if necessary. In the case of Horn *et al*'s approach, the extraction uses the granularity of classes, whilst in the approach from this thesis the transformations occur also at the method and even code fragment level. This allows for deeper restructuring, enabling a finer control on characteristics such as the separation of concerns. In addition, in Horn *et al*'s approach, the transformation's focus is facilitating extraction, rather than closing the gap between the original application's code structure and SOA-compliance. In the approach described in this dissertation, the transformational aspect is more differentiated, thus allowing further possible specifications.

Chapter 4

General Methodology

This chapter presents the general reengineering methodology used in this dissertation. It is based on the Horseshoe model, refining it to support automation and traceability. This is preceded by a discussion about different types of strategies in order to set the context for the methodology.

4.1 Business Context

It is common for enterprises to maintain legacy applications for a large number of years due to their core importance in business operations. However, in order for these to keep up with new requirements, organisations incur high costs for maintenance, and to replace or retain existing staff. The siloed nature of these systems, where integration or consolidation rarely occurred, add to the lack of flexibility which can impair business process efficiency.

In a very aggressive global economy, it is necessary for organisations to have easily adaptable IT solutions. Only then is it possible to adjust to new market needs, often a result of increasing competition. Other requirements that force

companies to quickly change their systems is compliance to new legislation.

Given this context, there is a high pressure on organisations for modernising their existing IT infrastructure, particularly in what can bring cost savings, or a high degree of flexibility [39].

The work presented in this dissertation aims to provide a way of using effective techniques in modernisation projects, both by companies in need of modernising their software infrastructure and by service providers specialized in the reengineering arena. Automation takes a big role in this process in order to enable a scalable approach. As discussed in [7], there are estimates from technology research firm Gartner indicating that a well trained programmer manually migrates 160 lines of code per day. For large scale projects, even with a large team of developers, this translates into a typically undesirable length.

To illustrate how the approach maps to a concrete business context, this chapter draws parallels with the core business model of ATX Software, a company with a focus on software modernisation products and services [1]. Its offer in this area includes migration from COBOL or Oracle Forms to Java or Microsoft .NET. Its clients, from various regions of the world, span the main industry sectors, including banking, public sector, health and insurance. The process is not exclusive to ATX, being followed by multiple companies both in the reengineering area and in general software engineering.

This dissertation's author currently holds the post of senior consultant at ATX Software, hence the use of this particular example. Furthermore, during the author's participation in reengineering projects, he came to understand how the work presented in this document adequately matches current business needs.

Process

A typical reengineering project has three phases.

1. technical analysis and proof-of-concept / pilot
2. transformation
3. testing and transition

Whenever possible, the process starts from a point in which there is already an implementation of the methodology for the general needs of a specific reengineering project. This is the scenario that occurs when a set of tools and methods were developed and accumulated throughout past projects. In cases in which there was no similar prior work, there is the extra preparatory step of creating an instantiation of the overall methodology, which can include combining parts of other previously created, but not directly usable, instantiations. This process is depicted in figure 4.1.

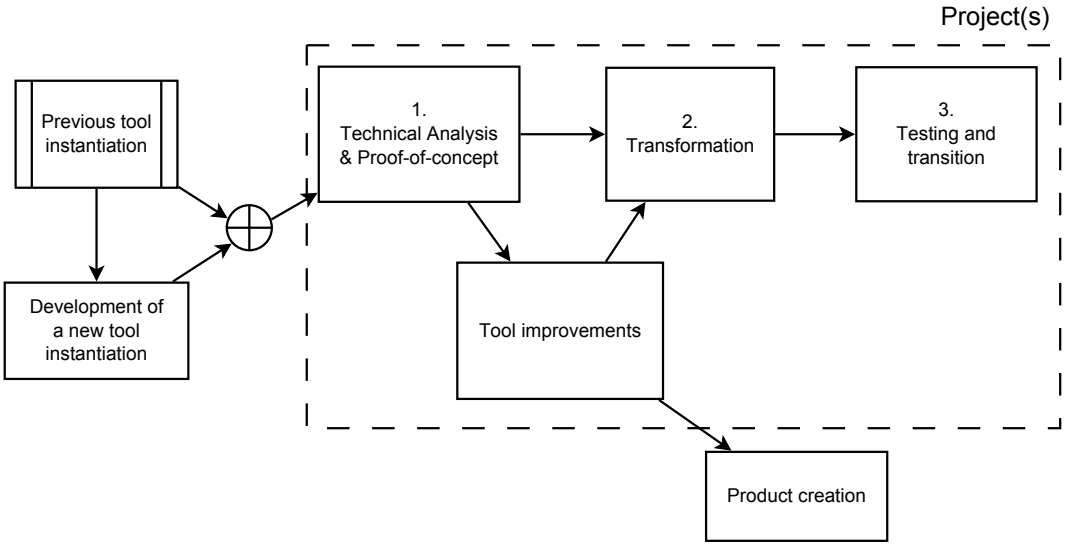

Figure 4.1: Tool development process

Phase 1 includes two main tasks. The first is an analysis of the original application. This is required in order to understand the application's specificities and

eventually identify improvement opportunities over the reengineering solution. Following this task, and any changes driven by it, a subset of the application is selected for a proof of concept. The size of this sample can vary according to each project, and is sometimes subject to a negotiation between the involved parties, but can amount to 10% of the entire system. When a larger subset of the application is chosen, it might more appropriately be called a pilot for the project. It is crucial that the selection of this sample adequately represents the overall system.

For a typical project, it would be expected that using an existing solution 60-80% of the code would be migrated automatically. This result is then improved by a process of customisation, creating and adapting pattern matching and transformation rules until the coverage reaches the desired level of 80-90% of the system sample.

Phase 2 consists of applying the transformation to the full suite of applications. This can occur in several stages, transforming subsets of the system separately. As in phase 1, improvements can be made to the set of rules along the way. Thus, core activities will be the automated application of the transformation tool, followed by the manual confirmation or revision of any code fragments the tool has designated as being less confident about. The latter will possibly lead to the development of new rules.

Depending on external dependencies or the overall IT infrastructure, integration work may need to take place. When the transformed version of the application is complete, it will run in parallel with the legacy version in regression testing mode until there is confirmation that the two are functionally equivalent (phase 3). The application can then be replaced by the new version.

For large scale migrations, this may involve an incremental replacement. The

reengineering pattern *Make a Bridge to the New Town* can be used to address data dependencies between the legacy and new system modules until the migration process is complete [31]. This is achieved by creating a data bridge that will be used to incrementally transfer data from the old system to the new, as more components are ready to take over from their legacy counterparts.

Sometimes parts of existing systems are only available in executable state, because the source code was lost, or never owned by the company. This scenario, which is out of the scope of this work, typically involves replacing these modules by newly built ones, already using the intended target architecture for the migration of the overall system. Alternatively, a service wrapping technique can be used but this is not always viable [96], and does not allow the same degree of flexibility.

As shown in figure 4.1, a tool chain evolves whilst being used in the context of a project, or a set of projects. For service providers like ATX, this process allows for such a concrete approach to mature from an internally used solution to a commercially viable product that can be used by external parties.

Having developed a concrete instantiation of the methodology, this is added to the portfolio of approaches that can be reused. This instantiation can be the result of the fulfilment of a perceived need in the market and the subsequent investment in the development of an individual type of transformation, or the product of a specific projects requirement.

To summarise, the requirements that derive from the above process for a reengineering tool (or set of tools) consist of:

- Reuse - In developing a new toolset, even to address a particular reengineering scenario, it is important to ensure this is as general as possible in order to allow use in multiple contexts.

- Flexibility - Each project has its own specificities, hence it is important for a tool set to be capable of being incrementally developed, with each usage contributing to its improvement.
- Scalability - For an approach to be applicable in real world scenarios, it has to cater for several system sizes, particularly large-scale applications.
- Automation - In order to achieve consistent results whilst also reduce the amount of effort for the complete process, it is necessary to provide a high level of automation.

4.2 Reengineering Strategies

In reengineering, like in other areas of software engineering, it is possible to follow a top-down, bottom-up or meet-in-the-middle (hybrid) strategy to accomplish the intended goals. These strategies are discussed next.

Top-down This approach begins with an analysis of the business domain and can lead to business processes or enterprise architecture models using frameworks such as [46]. From these it is possible to determine the necessary software systems to implement the requirements. It is then necessary to map these to existing applications, identifying where reengineering will be applied.

Bottom-up This is where reengineering opportunities are identified by analysing the existing code and low-level documentation (e.g. identifying candidate services). A number of techniques can be used for this purpose, including an analysis of data consumption.

Meet-in-the-middle This is a combination of the approaches, in which high level models are merged with what is found in the existing system, leading to changes at both ends: corrections in the business models with what is found in reality and changes to what is needed to reengineer at the code level.

All of the above methods require code analysis and transformation. Whilst these techniques could seem necessary only for the bottom-up and meet-in-the-middle approaches, they are needed for the top-down approach to map the high level requirements and processes to software modules. Given the typical volume of applications considered for reengineering (e.g. migration to SOA), both require automation. Analysis can be performed using a number of available techniques, the main one being code pattern detection. Transformations need to be done in a consistent way, being uniformly applied, so to ensure both the compliance to the requirements, but also to guarantee code that can be adequately maintained.

4.3 Methodology

The general methodology for architectural migration developed in this research began in the context of European projects Leg2Net and SENSORIA. In this work, the methodology is instantiated to support both technological and functional transformations, as described in chapters 5 and 6, but it is general enough to be used in other circumstances, such as identifying and extracting application code that can be moved to the database.

This methodology separates transformations into four steps. The starting point is the original source code of the application. The first step is a preparatory one, where code is annotated in order to provide critical information for the remaining steps. Its result are the annotations, not necessarily in the form

of actual changes to the code, but possibly in an external document that refers to source code fragments. The following steps consist of deriving a model representing the annotated code via a reverse engineering step, transforming it through redesign techniques, and obtaining the final code via forward engineering.

One of the goals of defining this methodology is to obtain a process to systematically address reengineering projects. Additionally, a high degree of automation is required. Supporting transformations for a wide range of applications includes addressing large scale systems, which due to both temporal and financial aspects, are extremely difficult to undertake with manual approaches. This is one of the main reasons for the separation between the activities of code annotation and reverse engineering. Such separation makes it possible to concentrate manual work in the former, where due to work being done early in the process, allows for most of the human effort to consist of analytical work, rather than the repetitive tasks that are more adequately left for machines. It is possible to achieve this since the manual effort focuses on finding patterns that once generalised can be turned into code pattern matching rules. These can then be detected in the complete system in an automated fashion. Hence, this activity is a semi-automated process, since the pattern detection is performed via a tool and there is the possibility of reusing a set of rules from an existing portfolio built over previous projects (see section 4.1). As such, manual effort can be further reduced. The remaining three steps of the methodology then take advantage of the product of this step to maximise automation.

The methodology is illustrated in Figure 4.2 and its four steps are detailed next:

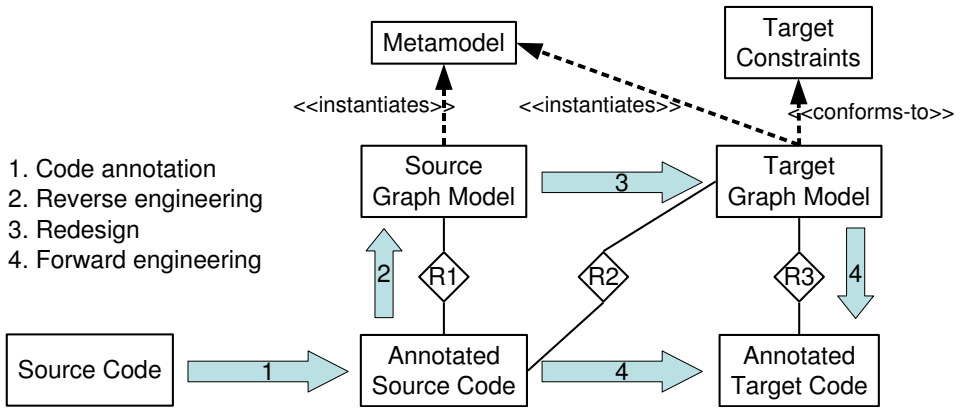

Figure 4.2: Methodology for transformation-based reengineering

1. Code Annotation. In this step, the source code is annotated by code categories. These distinguish its constituents (packages, classes, methods, or fragments thereof) with respect to their desired association to architectural elements of the target system.

The annotation process is a combination of automated pattern matching rules and input from a developer. Depending on how complete is the rule set, this process can lead to an interleaving of automated and manual code annotations or rule revisions. The amount of manual effort depends on the patterns found in the original software, of which older systems, due to usually having had different developers involved during many years, tend to have a broader variety.

The target architecture defines the code categories to be used for the annotation process. These represent the architectural elements that code fragments will be mapped into. When migrating to a 3-tier system, the categories to consider are User Interface (UI), Logic and Data, as is described in this thesis 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 and employees).

Reengineering tasks sometimes involve annotating code, typically in the form of comments following a convention such as prefixing them with the initials

of the developer that made the comments and the date. As described in the pattern *Tie Code and Questions* this can assist in source code understanding during reengineering activities [31]. In the case of this methodology, this is translated into the goal of facilitating automated transformation through the subsequent steps.

Annotation does not necessarily have to be applied, or even represented, at the source code level. The pattern matching rules can be applied at a higher level representation, and the resulting annotations can be defined by references to the code.

2. Reverse Engineering. This step obtains a graph representation of the code using the information gathered during the annotation process. The resulting graph does not have a one-to-one mapping with the source code. Its level of detail heavily depends on the result of the annotation. This is achieved by controlling granularity such that structural elements that are annotated as contributing to just one code category are represented by a single node in the graph. If, instead, elements have fragments of several categories, each of these has a separate representation in the model. Since, for example in object-oriented systems, 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 a highly scalable solution. The ratio between the number of nodes of the abstract syntax tree vs. nodes of the graph is discussed in chapter 8. Another benefit of a graph representation is that it allows transformations to be described in an intuitive, visual way. The relation *R1* between the original (annotated) source code and the graph model is kept to support traceability.

The resulting graph model is based on a metamodel (formally a type graph)

that contains a representation of the code structure as well as of its categorisation and association to architectural elements. The 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 are derived by the code annotation step. An example of metamodel (type graph) is presented in the next chapter.

3. Redesign. In this step, the source graph model is restructured to reflect the association between code fragments and target architectural elements. This transformation is driven by the code categories by which each node has been annotated with, and is specified by graph transformation rules. The execution of these rules enables producing the target architecture. This allows:

- abstracting (in large parts of the process) from the programming languages involved, as long as they belong to the same paradigm, being based on similar concepts;
- describing transformations in a more intuitive and "semantic" way (compared to code level transformations), making them easier to adapt to different targets.

The rules conceptually extend the graph transformation suggested by Mens *et al* in [77] to formalise refactoring [40]. 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. During the transformation, the relation with the original source code is kept as *R2* in order to support the code generation in the next step.

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. An example of graph transformation rule can be seen in Figure 2.3. In the technological dimension the rules aim to re-organise the model into a three-tier architecture, thus complying to the SOA principle of separation of business logic from presentation (cf. Introduction). The rules for the functional dimension restructure the model so that services comply to the principles of loose coupling and course-grained nature.

4. Forward Engineering. The goal of the forward engineering step is to obtain the target code. This is achieved by code level transformations that are determined by the graph transformation rules which occurred at model level. During the redesign stage, a log of all transformations is kept, identifying the rules applied, as well as the graph element instances involved. This is used to drive the forward engineering, which maps each model level transformation to one or more code restructuring rules.

The result of this step, the annotated code keeping a relation with the graph model, allows for several iterations of the four step cycle in this methodology, enabling the creation of a chain of different types of transformations. This is particularly relevant if the reengineering is directed towards service-oriented systems. In this case the transformation has to address both the technological and functional dimensions, e.g., transformation into a three-tier architecture should be followed up by a decomposition into functional components.

Chapter 5

Technological Dimension

This chapter presents details of the instance of the general methodology for the technological dimension.

5.1 Overview

The goal of this dimension is transforming applications into layered architectures. The source of this process can range from monolithic applications, where modularity was not a concern when initially developed, to applications which were already created whilst following a layering principle, but it was not accurately followed. The reasons for the latter to happen include software erosion [108]. In the context of this dissertation, the technological dimension is focused on transformation towards three-tier architectures. These architectures foster the development of systems that separate presentation aspects, application logic and data access.

5.2 Type graph

The class of applications that is addressed in more detail throughout this dissertation is object-oriented (OO). With this context, a type graph was defined in order to accommodate the source and target models, as well as the graph transformation rules.

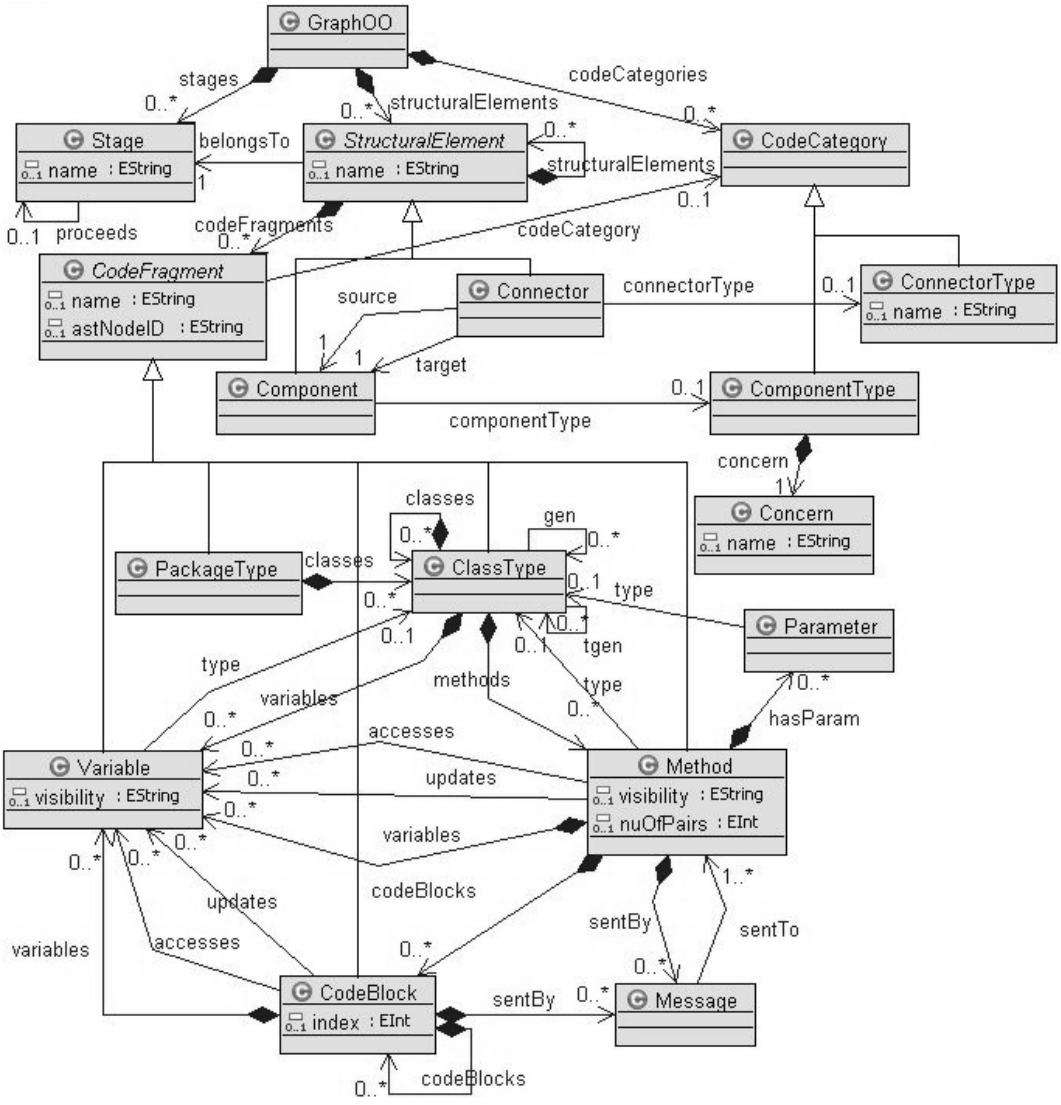

Figure 5.1: Type graph for the OO paradigm.

The model shown in Figure 5.1 has the goal of being flexible enough so it can be instantiated by any OO application regardless of the specific technology. This way there is a better chance that it can be reused for different instan-

tiations of the methodology described in this thesis. The type graph is an extension of the one presented by Mens *et al* in [78] in order to introduce classification attributes and the notion of code blocks, needed because the code categorisation requires a granularity lower than that of methods. *CodeFragment* elements are physical pieces of code which implies that they belong only to one *StructuralElement* (component or connector). Additionally, the concepts of *Component* and *Connector* were included in order to allow representing the mapping between the programming language elements and the architecture level. Note that the names for nodes *ClassType* and *PackageType* were defined as such, instead of *Class* and *Package*, to avoid collisions with Java reserved keywords, since Java code is generated from the model in this implementation.

During a transformation, there may exist components and connectors that belong either to the source or the target architecture. For instance, after some transformation rules have been applied, components of the source and target architectures may coexist in the model. The concept of *Stage* was added to cope with those intermediate phases. Depending on which stage node an *StructuralElement* is associated to, it is possible to determine whether it belongs to the source or target architecture. This also allows the design of transformation rules that make use of this information, for instance looking for elements contained in source architecture components to be moved to target ones.

Since it is necessary to keep traceability to the code in order to facilitate the transformation / generation process, a way to associate it to the type graph had to be considered. Given that it is desirable to be as language-independent as possible, the type graph was not directly linked to the source code. Instead, an attribute (ASTNodeID) was used to associate some of its

elements to the Abstract Syntax Tree (AST) of the program. ASTs are very common representations of source code and, in this case, allow for a loose integration between the model and the programming language.

Note that this type graph is used also in the functional dimension.

5.3 Code annotation

Given the architectural style, the goal of code annotation for the technological dimension is to identify in the source code the code categories of user interface (UI), logic and data access. This step of the technological dimension is based on code pattern matching rules. These 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 object-oriented code) or even technology independent (e.g. rules based in matches from other rules).

These rule sets can be reused in multiple projects. Depending on the technologies involved, or the use of unusual coding patterns, there may be the need for manual intervention or the creation of new detection rules.

Some code fragments can be considered to fall into more than one category, such as the result of a UI method call being used directly in a Logic operation. Since the annotation can be performed at the abstract syntax tree (AST) [65] 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.

Next, some of the code pattern matching rules defined for the technological dimension are presented. These were designed in an ATX reengineering environment called CareStudio. This is an Eclipse [106] plugin based on a tool by ATX (L-CARE) that, amongst other code analysis and reengineering functionalities, allows for the specification and execution of code pattern matching rules and storing of the resulting markings/annotations in XML format. The code patterns are defined over an XML representation of the abstract syntax tree of the code. Rules for code pattern matching are defined as XPath queries (XML query language [109]) as this is the querying language made available in the CareStudio tool. These can range from simple expressions to a combination of an arbitrary number of expressions, using the output of expressions ("parameter equation <identifier>") as parameters for others. It is also possible to specify conditions ("condition <identifier>") to the main expression ("main <identifier>").

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

Additionally, there are many rules of a higher abstraction level, in the sense that they make use of the result of applying others first (e.g. obtaining nodes annotated by other rules by using a call to function "getMarkedNodes"). From the following list, the third and fourth examples belong to this type of code pattern matching rules.

With this type of sequential dependencies, it is necessary to ensure that rules will be attempted to execute in the correct order. For this purpose, it is possible to make dependencies explicit in CareStudio.

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:

```
parameter equation UI_TYPES{getConst("uitypes")};
main equation ALL{//FieldDeclaration};
condition MAIN_EQ{$ALL[contains($UI_TYPES,
                  concat(";",Type/Name/@value,";"))]};
```

   The first equation loads known UI types into variable *UI_TYPES*, whilst the second keeps a list of all field declarations in variable *ALL*. The third line consists of a condition that will only be satisfied for field declarations whose type is in the list of UI types previously obtained.

2. *Comments in user interface context.* Even comments can be identified as belonging to a specific concern based on the context in which they appear. The AST generated by CareStudio facilitates achieving this since the comments are already associated to (non-comment) statements to which they refer too. The following is an example of a UI comment (the "(...)" were used to simplify the rule expression for this thesis):

```
main equation ALL{//Comment};
parameter equation UI_NODES{
  getMarkedNodes("UI_Assignments")
  | getMarkedNodes("UI_Blocks")
  | getMarkedNodes("UI_LocalVars")
  | getMarkedNodes("UI_StatementExpressions")
```

```
  (...)};
condition COND{count($UI_NODES[$ALL/parent::node()/@ID
                                                = @ID])};
```

The first equation keeps a list of all comments in variable *MAIN*, and the second one gathers all AST nodes previously annotated by UI rules (e.g. UI assignments). The third line consists of a condition that verifies if comments are associated with UI nodes by testing if their parent identifier (*ID* attribute in the rule) matches one of an UI node.

3. *Data access methods.* Methods in which all contents have already been identified as belonging to the Data code category can be categorised as belonging to it:

```
main equation METHODS{
  getMarkedNodes("Data_Blocks")/parent::*
                          /parent::Method};
```

4. *Calls to methods previously categorised as belonging to data access.* Calls to methods that were identified as belonging to the Data code category by the previous rule can be themselves categorised also as Data:

```
main equation CALLS{//FunctionOp[Name/@value=
  getMarkedNodes("Data_Methods")/Name/@value]
                                      /parent::*};
```

5.4 Reverse engineering

Taking advantage of the information produced by the code annotation, the reverse engineering step is fully automatic. Source code fragments were previ-

ously identified as belonging to specific code categories, so what is needed at this stage is to create elements in the graph model with the right granularity. This graph is generated whilst complying to the specified metamodel.

There is no need to represent every code fragment in the graph. Considering the code as represented by an AST, from a containment relation perspective, only the elements that contain children of different categories, or those that belong to a different category than their siblings, need to be represented as nodes in the graph. This reduces the overall size of the graph to the absolute necessary, and the redesign step can take advantage of this succinctness.

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.

5.5 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.

An example of graph transformation rule can be seen in Figure 5.2, as designed in the Tiger EMF Transformer tool [4]. Note that, for simplicity, node attributes are not presented in the right-hand side as they were not modified. The numbers prefixing each node name are indexes used to distinguish between nodes, especially useful to differentiate elements of the same type. Negative

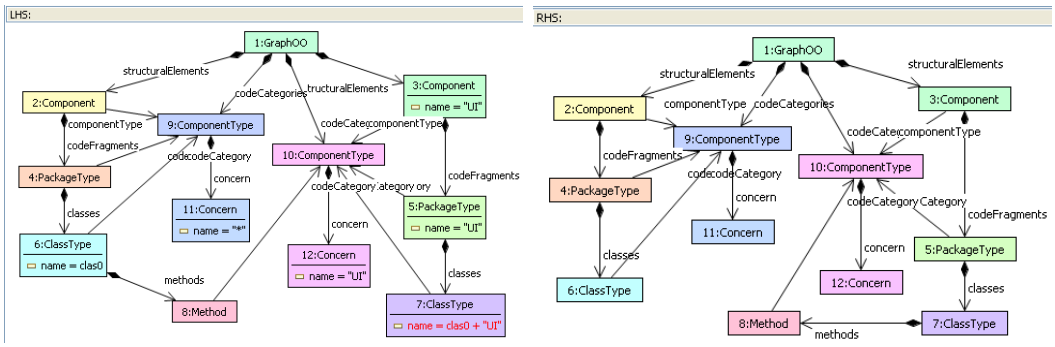

Figure 5.2: Move Method UI transformation rule.

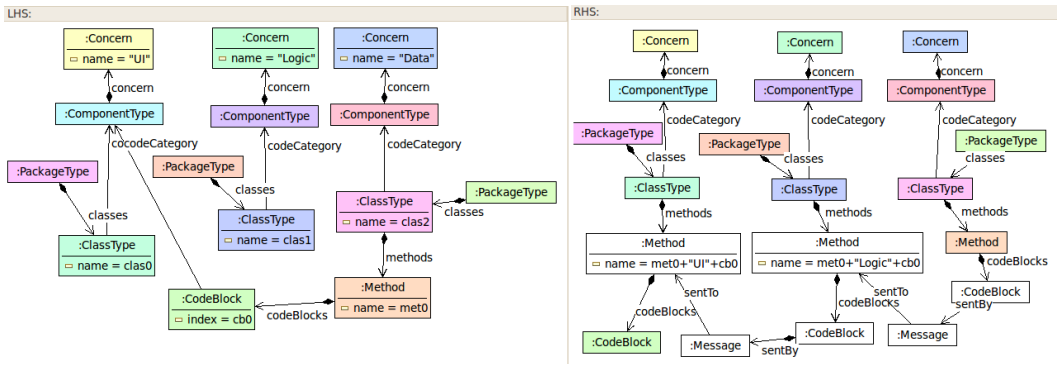

Figure 5.3: Extract Method Data UI transformation rule.

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.

Another example of graph transformation rule is depicted in Figure 5.3. This consists of the *Extract Method Data UI* rule whose purpose is to extract code blocks that are present in the Data layer but belong to the UI. Each code block found in these conditions becomes a method, and this is moved to the UI layer. Additionally, a delegate method is created in the Logic layer.

5.6 Forward engineering

The input for the final step of the process, the forward engineering, is the log of transformation rules that were applied in the redesign step, the source code, and the relationship between these (kept in the graph model). This information is used to select and guide the code level transformations that are necessary to execute. In the case of object-oriented applications, these consist of refactorings.

Each model-level transformation is mapped to one or more refactorings. An example of the former is transformation rule *Move Method UI* (cf. figure 5.2), which is mapped directly to a *Move Method* code refactoring. More complex graph transformations, as *Extract Method Data UI* rule (cf. figure 5.3), may require several code level refactorings. For this example, three refactorings are necessary:

1. Extract Method - Extracts the code block, resulting in a new method still in the Data layer class;
2. Move Method - Moves the new method to the Logic layer;
3. Move Method - Moves the method from the Logic layer to the UI layer, leaving a delegate in the former.

Chapter 6

Functional Dimension

This chapter describes the instance of the general methodology for the functional dimension.

6.1 Overview

The goal of this dimension is the restructuring of existing applications such that the resulting components, after having replaced their presentation tier, represent services. As with the technological dimension, this process can have different kinds of applications as input. These can be systems where modularity was not considered from a functional standpoint, or where this separation was broken by maintenance activities. The decomposition addresses the two last service-oriented properties mentioned in section 1.2.

6.2 Code annotation

Compared to the technological dimension, the functional one presents more challenges for the code annotation step. Whilst in the former it is possible to find many similarities between different applications (especially if they are based on the same technology or framework), the latter depends on application functionality. 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:

- operation identification
- grouping of operations into services

In this thesis, *operation* stands for a functionality that is likely to be at a too low granularity to be considered as a service in a 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, so depending on the accuracy of the identifiers used it may be necessary to define the names of code categories manually.

The identification of operations in source code is performed by first locating their entry points. The techniques used for this purpose include the ones described in the next paragraphs.

Code belonging to the Logic layer that is invoked by the UI

Even in systems where there is no actual separation between application layers such as UI and Logic, it is possible to find blocks of code that perform a

UI operation that triggers application logic. This kind of transition between application concerns is a natural candidate to be inspected as there is a high probability of the corresponding Logic layer code being the start of an operation that will be a service constituent (see Figure 6.1). To find this pattern in source code it is necessary, first of all, to identify which blocks of code belong to the UI or Logic. The process to reach such a goal, regardless of how well the architectural layers are separated in the input code, is part of the role of the technological dimension of the approach presented in this dissertation. The output of that process can then be used to design this code pattern (following the same notation as in chapter 5):

```
main equation CALLS{getMarkedNodes("UI_Methods")
    //FunctionOp[Name/@value=getMarkedNodes("Logic_Methods")
    /Name/@value]/parent::*};
```

In cases in which there are runtime frameworks that handle presentation related requests, these connections between UI and Logic may not always be explicit. To address these scenarios, it is possible to define specific queries that take that into consideration. For example, in a Swing-based Java application, events trigger methods called “actionPerformed” in classes that implement the “ActionListener” interface. A rule that takes that information in order to identify UI to Logic calls can be created as the following:

```
main equation CALLS{//Class[Interfaces/Type
    /Name/@value='ActionListener']
    /Method[Name/@value='actionPerformed']};
```

External APIs

APIs that are published for external use refer to relevant features. A list of operations that are available can be obtained using several techniques, e.g., by parsing IDL (Interface Definition Language) files and identifying remote calls.

Typical patterns of control / data flow

There are many code patterns that can help to identify entry points to application functionalities. The number of distinct calls that a method is subject to (FAN-IN) gives an indication of how reused that method can be. In the case of high FAN-IN, the method can be classified as an operation but this is subject to a granularity analysis. It is necessary to distinguish between methods that are called from several sources because: they provide common low level operations such as reading data from a file or converting between different data types; or they are of a higher granularity thus being more meaningful. Pattern matching rules of this type are presented next:

- *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 [76] 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};
```

- *Methods with high Fan-Out.* Methods that call many other methods are typically of a level that is meaningful enough to be considered as candidates to be part of services (variable N is a parameter for the rule):

```
main equation METHOD{//Method};
condition METHODCALLS{$Method[count(
 descendant::FunctionOp/DotOp) > $N]};
```

Known operations typically follow common patterns of control and data flow, such as authentication, validation and exception handling, and can be identified using this strategy.

Analysing the Data Access Operations

Operations that involve access to a data source typically have relevant importance and thus are candidates to be part of a service. The different types of access (Create, Read, Update, Delete) and the specific data that is involved also gives information on how to group operations into services. To identify the code that originates such data access calls, it is possible to use backward slicing.

Code that is shared by several operations

Blocks of code that are used by several different application functions 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). This is represented in the graphical example of Figure 6.1 as grey triangles.

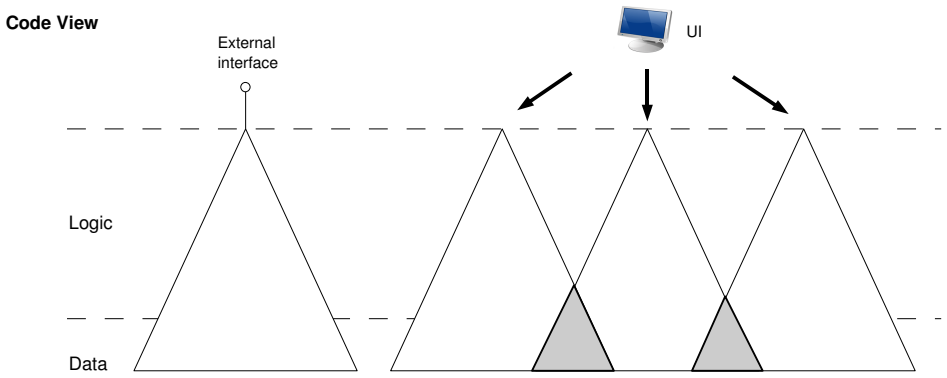

Figure 6.1: Identification of operations

Feature location techniques

Known feature location techniques such as LSI [75], a static approach, SBP [10], a dynamic technique or others as discussed in chapter 3. There are several 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 related in some manner are grouped together. Automation proposes ranked groupings of operations by using metrics based on several aspects, including:

- overlapping between operations

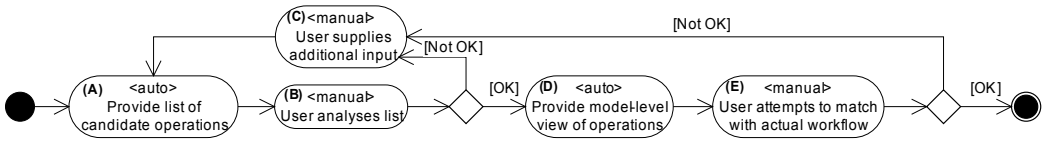

Figure 6.2: Meet-in-the-middle approach for identification of operations

- actors involved
- information about data accessed
- 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, to produce the graph model and drive the redesign process.

Meet-in-the-middle Approach

Whilst it is not explored in depth in this dissertation, it is possible to address the first task of the code annotation step of functional dimension with a meet-in-the-middle strategy as discussed in section 4.2. A possible approach is depicted in the activity diagram in figure 6.2 and each activity is described in the following paragraphs.

(A) The first data that will be presented to the user is a list of operations that includes: names and parameters, order, and data access involved (obtained via a bottom-up approach).

(B) The user analyses the list provided in the previous activity to check whether it is correct or not.

(C) If necessary, the user supplies additional information such as operations not to consider or more operations to include. This can be assisted with feature location techniques previously mentioned (e.g. LSI and SBP).

(D) Once it is possible to proceed, a model-level view of the operations and corresponding interactions/relations that resulted from the previous tasks is produced and presented to the user.

(E) At this stage, the user can compare the outcome of the previous task to the actual workflow(s) in which the software is used, possibly obtained from a top-down approach. The user can either confirm what is presented or go back and provide further feedback.

6.3 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.

6.4 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 principles mentioned in the Introduction.

An example of graph transformation rule for the functional dimension is de-

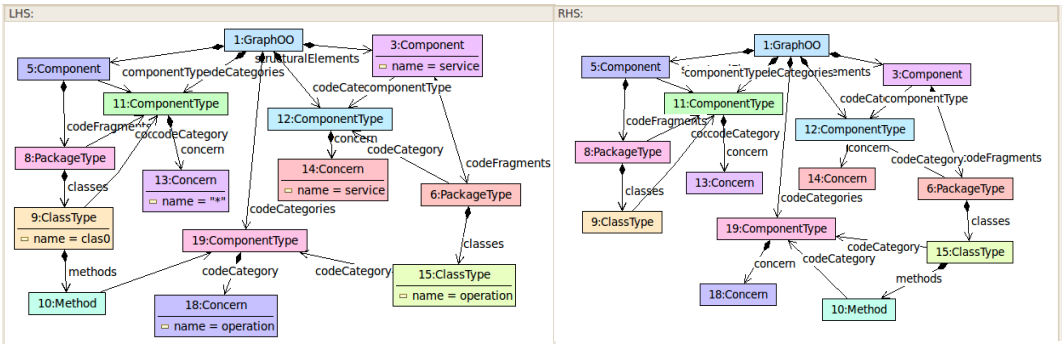

Figure 6.3: Move Method Operation transformation rule.

picted in Figure 6.3. This consists of the *Move Method Operation* rule whose purpose is to move methods that are constituents of a given operation into the class that was created to hold them. The latter is already present in a package that belongs to the appropriate service.

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

6.5 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 fact, it is possible to use the same implementation of this step for both cases.

Chapter 7

Implementation

This chapter starts by describing the process of creating an instantiation of the general methodology presented in this document, and then proceeds further by giving a concrete example that was mainly developed for evaluation purposes but that can also be used as a reference implementation.

7.1 Process

The implementation of a toolset to give support for a particular use of the general methodology presented in this document should start by taking into consideration a set of aspects.

For the first-time definition of an instance of the general methodology presented in this document, it is necessary to implement all of its building blocks (Figure 4.2). This has the prerequisite of both source and target architectural and technology paradigms having been defined. This is essential early on, since that information is necessary for specifying the reengineering process, in particular the metamodel that will be used.

7.1.1 Metamodel definition

The metamodel definition includes two tasks: determining the code categories and creating the type graph.

These categories are used in all steps of the methodology. The set of code categories are derived from the intended target architecture. Code categories can differ in their types, from simple labels to combinations of these.

The other component of the metamodel is the type graph used for the source model, target model and transformation rules. This needs to capture elements from both source and target programming language paradigms, as well as incorporate their relations to code categories.

7.1.2 Code annotation strategy, tools and artefacts

The code annotation strategy can differ according to the kind of transformation that is required, as can be seen in the technological and functional dimensions' discussions (in chapters 5 and 6). This definition task consists of detailing the strategy that will be followed, and creating the artefacts needed. After selecting the strategy, which can range from the usage of code pattern matching to feature location techniques, it is necessary to define the tool(s) to support it. With this, it is possible then to create the necessary artefacts, as code pattern matching rules or input to feature location tools (e.g. search expressions).

7.1.3 Transformation rules

Another activity that depends on a good knowledge about the architectural and technology paradigms for both source and target, is the definition of the

graph transformation rules.

Rules have to be designed to move from a source graph representing the original source code, and take advantage of the code annotation to reach a model which complies to the intended target architecture. These rules can be defined directly in the tool that will be chosen to execute them, so a choice of tool should precede this activity.

7.1.4 Target constraints

Whilst not being absolutely necessary, having specified target constraints allows for verifying whether the transformation occurred correctly, or even for termination purposes.

7.1.5 Tool support

In order to guarantee scalability, it is necessary to use automation as much as possible throughout the process. Selecting the right tools, or developing them when no adequate ones exist, for the methodology steps of code annotation, reverse engineering, redesign and forward engineering is crucial for the success of an implementation.

7.2 Prototype

This section describes the definition and implementation of a prototype, that can be both seen as a reference implementation of the methodology, but also a means to evaluate the approach described in this document.

An initial version was developed whilst using a small banking application in Java as scenario. The prototype was then extended and also used in the context of evaluating the results based on the above scenario and a larger real world application.

This prototype has support for the full reengineering cycle but not 100% coverage as this depends on each project requirements, and its main goal is to evaluate the approach under several scenarios.

As mentioned above, the prototype supports the steps of code annotation, reverse engineering, redesign and forward engineering. This implementation has as target the migration of Java applications but, as described in the previous sections, following this methodology provides a great degree of independence that is further explored throughout this section.

The following subsections summarise this implementation.

7.2.1 Metamodel definition

Code categories As stated in Section 7.1.1, code categories are derived from the target architectural and technology paradigm. Different models can be used for the categories. The chosen one is presented in Figure 7.1 and explained next, together with the instantiation.

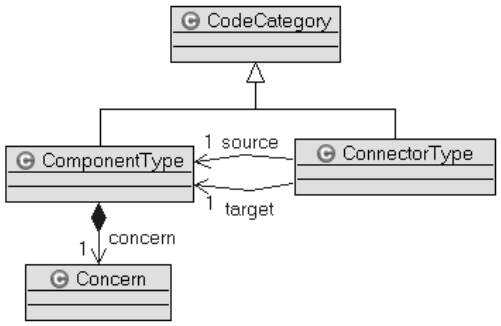

Figure 7.1: Code categories model for 3-tier architecture

In the chosen model, code categories can be divided into two types:

- components consisting of a concern
- connectors representing links between components

Concerns are conceptual classifications of code fragments that derive from their purpose, i.e., the tiers found in 3-tier architectures:

- User Interface (UI)
- Business Logic (BL)
- Data

The connectors are one-way (non-commutative) links between different concerns and include:

- Control: UI to BL
- Control: BL to UI
- Control: BL to Data
- Control: Data to BL

This model is detailed enough to capture the distinctions required by the target architecture for the technological dimension; other architectural paradigms might require different categories and more complex ways to represent them (as the introduction of a "Role" node as described in the paragraph below). In the case of the functional dimension, the categories are specific to each application, and consist of the names of functionalities. These can be represented using only the concern part of the above model.

It may be desirable in some situations to allow multiple categories for the same element. In Figure 7.1, components and connectors are represented by "ComponentType" and "ConnectorType", respectively, in order not to use the names attributed to architectural concepts. Both "ComponentType" and "Concern" exist for reusability issues given that the first is likely to be extended for certain types of target architectures. For instance, if the goal was to achieve a rich-client 2-tier architecture, then "ComponentType" would contain also a "Role" concept whose values would be "Definition", "Action" and "Validation".

Type graph The next metalevel activity consists of the definition of a model for program representation which, like the categories, may be shared with other instantiations of the methodology, either as source or target. In order to take advantage of graph transformation rules in the transformation specification, the model was developed in the form of a type graph. This consists of an EMF representation of the type graph presented in chapter 5, in figure 5.1.

7.2.2 Code annotation

The code annotation step was implemented using CareStudio (Figure 7.2 provides a screenshot of the application). The code pattern matching rules were developed in this tool as XPath queries, as already discussed in chapters 5 and 6.

Currently there exist around 40 code annotation rules in this prototype, most of which are programming language independent (approximately two thirds). Additionally, even rules that need to take specific language aspects into consideration can be configured in such a way that they can be easily ported to

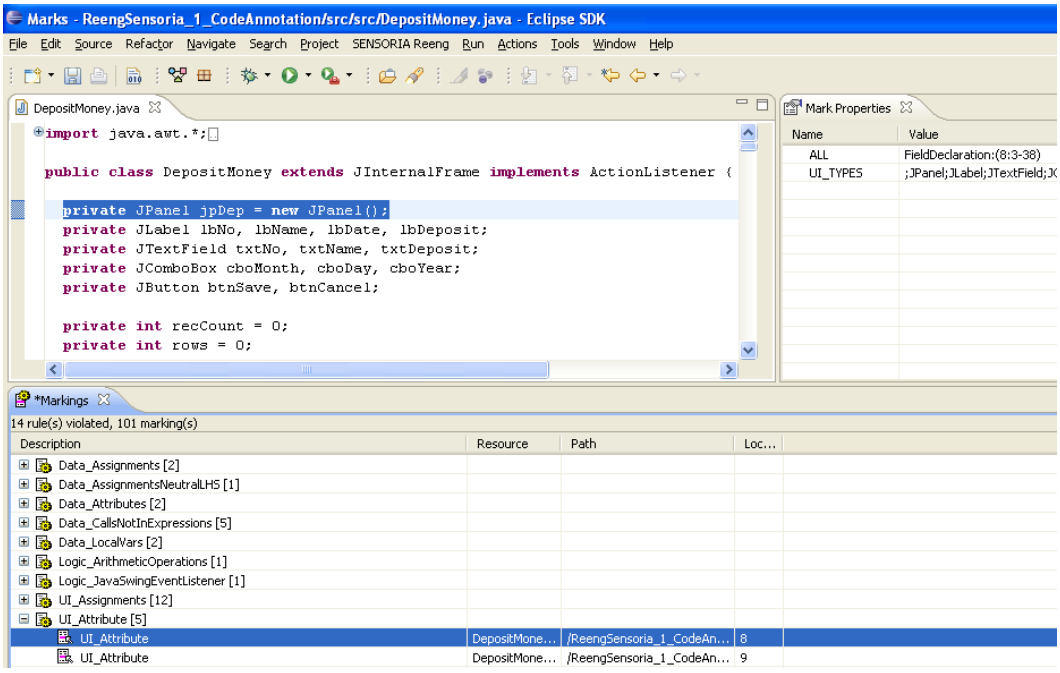

Figure 7.2: CareStudio - an Eclipse plugin for code pattern matching - showing one occurrence of an UI attribute declaration (rule UI_Attribute).

another context. This is the case of the first rule presented in chapter 5, where a number of programming language classes that are seen as belonging to the UI are stored in a global variable called "uitypes". This way, all rules that depend on this aspect, can work in other contexts just by having a different configuration file.

7.2.3 Reverse engineering

The main goals for this step are to achieve a more abstract representation than the abstract syntax tree, allowing the description of transformations in a more intuitive way and for these to be programming language independent. Additionally, given that, in these graphs, only the elements required according to the annotation are represented, the model to be transformed is simpler and the transformation process has better performance. This is particularly relevant when addressing the migration of large scale systems.

```
<?xml version="1.0" encoding="ASCII"?>
<OO:GraphOO xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:OO=
  <structuralElements xsi:type="OO:Component" name="General" componentType="//@codeCategories.0">
    <codeFragments xsi:type="OO:PackageType" name="General" codeCategory="//@codeCategories.0">
      <classes name="DepositMoney" astNodeID="DepositMoney.java.ast.xml#1010" codeCategory="//@codeCategories.0" tgen="//@struct
        <variables name="jpDep" astNodeID="DepositMoney.java.ast.xml#1017" codeCategory="//@codeCategories.1"/>
        <variables name="lbNo" astNodeID="DepositMoney.java.ast.xml#1027" codeCategory="//@codeCategories.1"/>
        (...)
        <methods name="actionPerformed" astNodeID="DepositMoney.java.ast.xml#1925" codeCategory="//@codeCategories.1"/>
        <methods name="populateArray" astNodeID="DepositMoney.java.ast.xml#2050" codeCategory="//@codeCategories.0">
          <codeBlocks name="CodeBlock" astNodeID="DepositMoney.java.ast.xml#2058" codeCategory="//@codeCategories.3" index="1"/>
          <codeBlocks name="CodeBlock" astNodeID="DepositMoney.java.ast.xml#2132" codeCategory="//@codeCategories.1" index="2"/>
          <codeBlocks name="CodeBlock" astNodeID="DepositMoney.java.ast.xml#2153" codeCategory="//@codeCategories.3" index="3"/>
        </methods>
    </codeFragments>
    <codeFragments xsi:type="OO:PackageType" name="External" codeCategory="//@codeCategories.0">
      <classes name="JInternalFrame" astNodeID="DepositMoney.java.ast.xml#1012" codeCategory="//@codeCategories.1" gen="//@struc
    </codeFragments>
  </structuralElements>
  (...)
  <codeCategories xsi:type="OO:ComponentType">
    <concern name="*"/>
  </codeCategories>
  <codeCategories xsi:type="OO:ComponentType">
    <concern name="UI"/>
  </codeCategories>
  (...)
```

Figure 7.3: XML representation of graph obtained through the reverse engineering step.

Given that no tool existed that could take the output from CareStudio and produce a result with the format needed for the next step, a new tool was built specifically for this purpose. This was developed based in the above requirements and, due to the choice of tool for the redesign step, the result is a graph model represented in EMF [32].

The tool receives the code annotation (as produced by CareStudio) and the original source code files and maps the information to graph elements as defined in the graph metamodel, generating an XML file per Java class.

An example of graph generated by the reverse engineering tool can be seen in Figure 7.3.

7.2.4 Redesign

The graph transformation rules were designed in the Tiger EMF Transformer tool [4]. 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. This plugin's code has since been committed to the Henshin Eclipse project [2]. The fact that this tool is Eclipse based, like CareStudio, contributed to its selection.

This tool does not have support for fuzzy reasoning, but fulfils the need for the transformations used for the work presented in this paper as the human contribution is focused on the code annotation step. One of the main goals is, to the extent possible, the reuse of the rules for multiple projects, so there is an interest in investing in more precise rule design even if that means more effort is spent in the first iteration, rather than using a smaller set of rules, fuzzy based, that could have a faster turnaround for a particular use, but that would require more effort to be spent considering multiple projects. However, for particular instances of the methodology presented in this paper it may make sense to use fuzzy reasoning to a certain level. Given the heuristic nature of the pattern matching, an approach such as in [82] combining graphical pattern matching with fuzzy reasoning could be used to define levels of confidence below which user interaction may be required. The choice of matching rules in XPath was motivated in part by the availability of the Eclipse-based pattern matching engine of L-CARE used in the majority of ATX's reengineering projects. Also, according to the methodology, the pattern matching occurs before the construction of the graph model, which is not a complete representation of the source code but only captures its structure with links to the actual code fragments.

The execution of the Java code generated by the tool produces a new graph, after application of the previously specified rules. In order to facilitate the next step, a logging aspect was also added to the generated code (with AspectJ), reporting every transformation made in the model, in order to guide the final

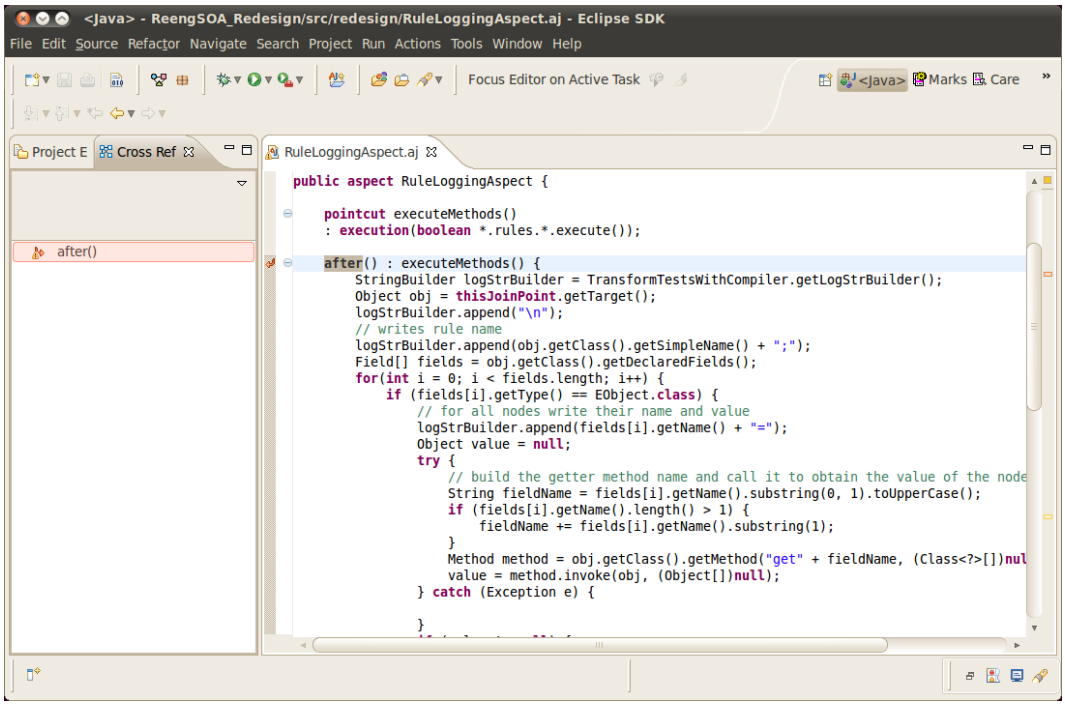

Figure 7.4: Logging aspect for retrieving information during graph transformation execution.

step. This is shown in figure 7.4.

7.2.5 Forward engineering

The log created by the redesign step includes information on the graph transformation rules that were applied as well as their order and the nodes of the graph that were affected. The way the forward engineering step works is to map this log to a sequence of code refactorings that, after applied to the source code, produce the final, transformed, target code.

In some simple cases, one graph transformation rule could be mapped to a single refactoring, while in other cases, like the ones involving the extraction of a code block into a new method (and placed in the correct class), it involved more than one refactoring.

The Java application that was built for this purpose uses Eclipse's built-in

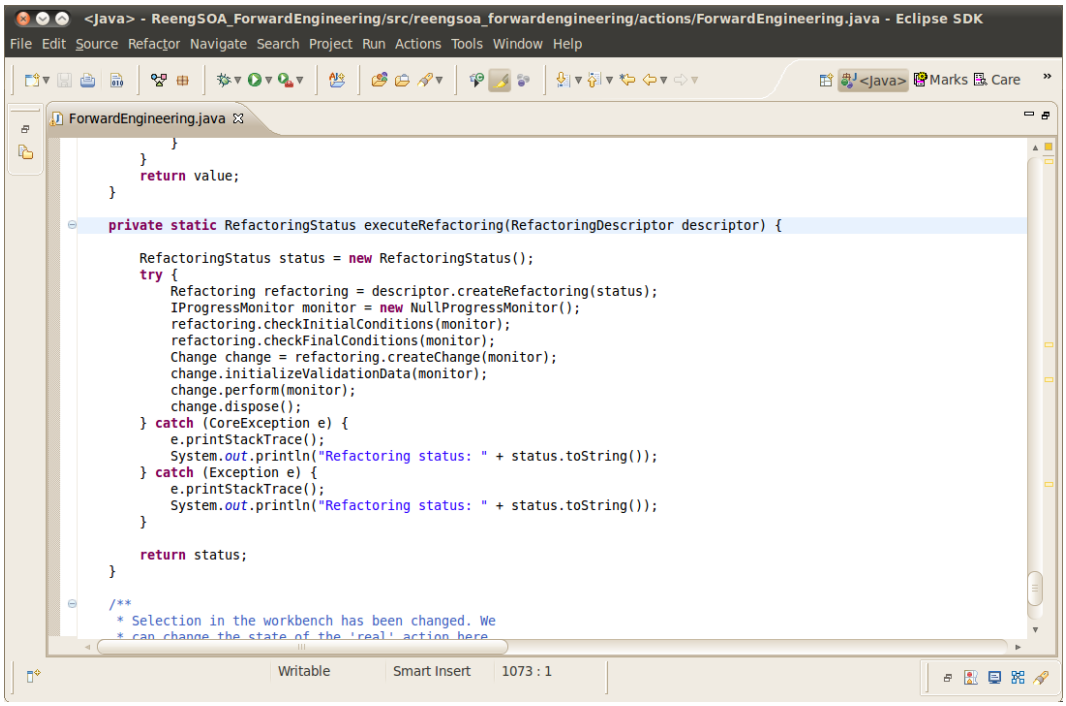

Figure 7.5: Eclipse refactoring execution (general).

refactorings so the work is mainly focused in mapping graph transformation rules to the right refactorings and dealing with their parametrisation. In short, the tool parses the log from the redesign step, for each executed graph transformation rule identifies the refactoring, or group of refactorings, to be applied, and executes the transformation (cf. figure 7.5).

The optimal result for this step is code that complies to the service orientation principles discussed in the beginning of this report. However, in situations in which not all necessary code annotation rules or graph transformation rules were specified the result may be incomplete. By analysing this and then reviewing the process it is possible to extend support by, for example, adding new rules.

All the tools used for the prototype were implemented in Eclipse and are used as plugins to this IDE. The tools involved, as well as the artefacts of each step of the process, are shown in figure 7.6.

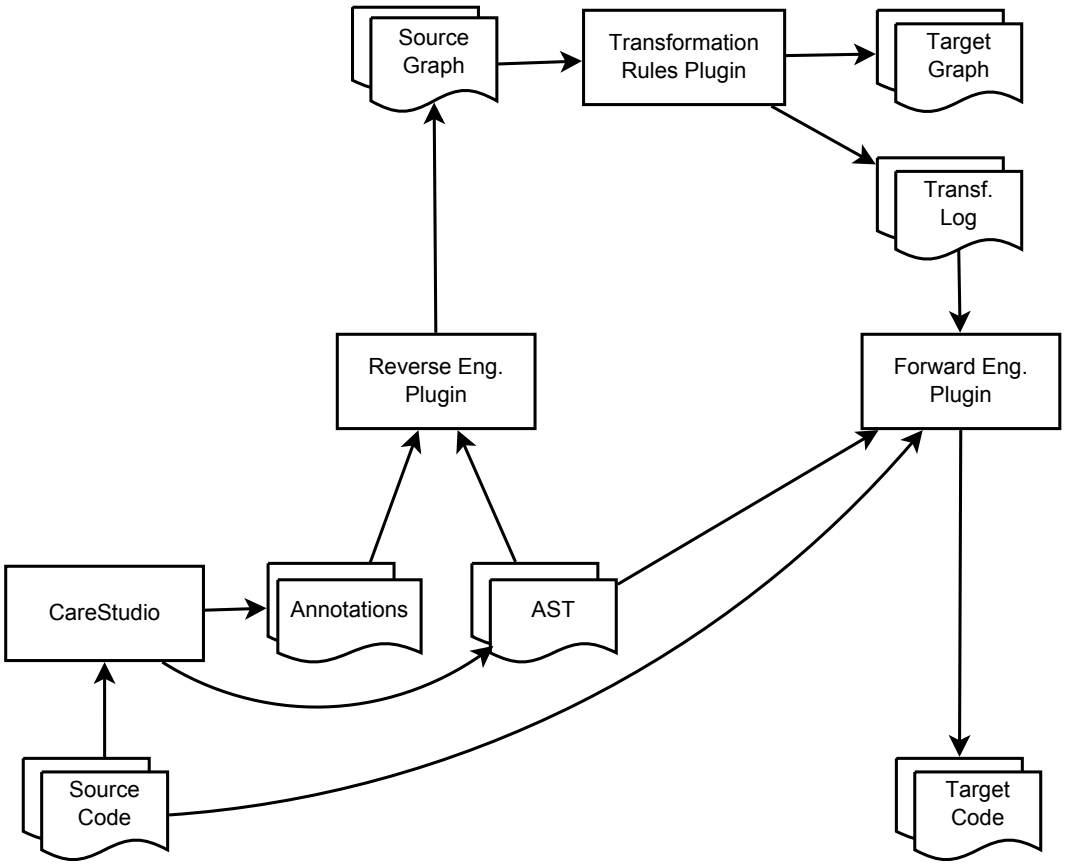

Figure 7.6: Prototype architecture.

This prototype architecture supports the several steps of the methodology described in chapter 4. Support for target constraints checking was not included in the prototype, and can be part of future work. Summarising the defined process, the starting point is the source code files. The code annotation step is performed by CareStudio, producing both the ASTs and annotations. These are the input for the reverse engineering plugin, which generates the source graphs. The redesign step is performed via a plugin that executes the rules, and generates both the target graphs and logs of the transformations. These, together with the original source code, and the ASTs, are the input for the forward engineering plugin, which through code level refactorings, produces the target code.

Chapter 8

Case Studies and Evaluation

According to the process described in section 4.1, the evaluation consists of several stages. As a preliminary step (1), a basic set of rules has been created and evaluated on a small case study to test the technical validity of the approach. For the actual validation according to the process, a large case study has been selected. Of this large case study, a subset of components was selected as a proof-of-concept sample on which the basic set of rules was executed first (2), followed by a period of extension of the rule set based on the analysis of the sample. The extended set of rules was executed first on the sample (3) and then on the entire case study (4).

Both case studies were selected as having Java as source, but with different domains and being developed by different teams, as it allows the evaluation of possible reuse between similar projects. Although Java is not one of the oldest technologies available, and not the one most would think of when discussing legacy applications, the first version of the language was released in January 1996, in which the context was different than nowadays. Java had a very fast adoption, and as a consequence, there is a large body of applications that were built in monolithic fashion, following patterns that, today, are considered

legacy.

The following sections provide details on each of the case studies and the steps (1-4) undertaken as well as the respective efforts involved.

8.1 Basic Evaluation

To validate the implementation of the prototype during its development, a small example banking application (*BankSystem*) in Java with 21 classes was used. Its GUI is written in Swing and GUI code is mixed up with both application logic and data access code. Its data persistence is file based. Functionally, this application allows common banking operations such as deposit money, withdraw money and view customer details.

In order to determine basic performance and scalability, the prototype was applied to this application and the time spent for each of its steps was recorded. The results are recorded in Table 8.1. The table lists the applications addressed and gives for each of them the number of lines of codes. It also states which of the two sets of rules has been used in the particular experiment and details the run time of the individual phases as well as the total time. Finally, the coverage is given in percentage of the total lines of code, as measured by a small Eclipse plugin counting the lines covered by code annotations. This is possible since each annotation indicates the start and end position of the pattern match. It is worth noting that the total time for this process was still under two minutes. The redesign step was particularly fast which means that the contribution of the reverse engineering step in producing the smallest graph model possible may prove very valuable in larger scenarios.

It is worth noting that the time for the code annotation (90% of coverage

Application (LoC)	Rule Set (size)	Annotate	Abstract	Redesign	Transform	Total	LoC Covered
BankSystem (3,259)	basic (50)	23.50 s	53.02 s	0.92 s	35.05 s	112.49 s	90%
RapidMiner PoC (42,200)	basic (50)	5 min	12 min	1 min	10 min	28 min	70%
RapidMiner PoC (42,200)	extended (60)	5 min	12 min	1 min	10 min	28 min	80%
RapidMiner Full (370,000)	extended (60)	52 min	140 min	12 min	105 min	309 min	75%

Table 8.1: Run times obtained while using the prototype on the example banking application. Times in the centre columns are in seconds (s) and minutes (min)

```
package ui;

import java.awt.Color;
(...)
import logic.DepositMoneyLogic;
public class DepositMoneyUI extends JInternalFrame implements ActionListener {

    private DepositMoneyLogic __logic = new DepositMoneyLogic(this);
    private JLabel lbNo;
    (...)
    DepositMoneyUI () {

        // super(Title, Resizable, Closable, Maximizable, Iconifiable)
        super ("Deposit Money", false, true, false, true);
        setSize (335, 235);

        jpDep.setLayout (null);

        lbNo = new JLabel ("Account No:");
        lbNo.setForeground (Color.black);
        lbNo.setBounds (15, 20, 80, 25);
            lbName = new JLabel ("Person Name:");
        (...)
```

Figure 8.1: Sample of target code. Several members were moved from original class DepositMoney to DepositMoneyUI.

for this application) depends on the number of annotation rules, i.e., as more annotation rules are added to complete the rule base, these times may increase.

Figure 8.1 shows sample code from the target of the prototype. Several members, including fields, methods and inheritance of the original DepositMoney class are now in class DepositMoneyUI, which only contains presentation aspects. The rule base resulting from this step includes 50 annotation rules which are the input to the next step.

8.2 Proof-of-concept Sample

A larger, real world application was required as input to this stage. The chosen one was *RapidMiner*, a leading open source tool for data mining offering a range of functions for data integration, analytical ETL, data analysis and reporting. RapidMiner is available from *sourceforge.net*, the open source project

repository, at [92]. The application has a large user base, having been downloaded more than 520,000 times at the time of writing and version 5.0 consists of more than 370,000 lines of code. Like the case study described in the previous subsection, this is a Java application and its GUI is written in Swing. The choice was an application that is reasonably close in terms of the technology used so that it would be possible to evaluate the reusability of the tools and definitions in similar projects and estimate the extra effort involved. Having said this, the application is not typical for the kind of application usually the target of ATX's reengineering efforts. However, an interesting application scenario (outside the scope of this thesis) could be the offering of data mining facilities as web services.

It was observed that the application has a better separation between user interface and other concerns than the BankSystem, but is still not compliant with the service-orientation principle. As a proof-of-concept sample, a subset of RapidMiner was used, consisting of 337 out of the total 2505 Java files contained in version 5.0 of the project.

This was done solely to allow a more in depth look at the achieved results, but also goes in line with what is an industry practice of addressing the reengineering of subsystems individually.

The code annotation step, which is where the strategy from this dissertation is intended to concentrate the manual part of the process, achieved a lower coverage (approximate average of 70%) than with the BankSystem application. This was expected as different coding styles can have an impact on the performance of the code annotation rules.

The steps of reverse engineering, redesign and forward engineering for RapidMiner, executed in the same fashion as in BankSystem, give a good indication

that this reengineering methodology and its implementation have good support for reuse in different projects. In order to determine the impact of a larger code base (and also different programming practices) in the performance of the prototype, the time spent for this application in each of its steps was recorded. The details are again recorded in Table 8.1. Time is measured in minutes, rather than seconds, but the breakdown for individual steps follow a similar distribution as for the BankSystem application transformation.

The total time is good considering the number of files, and also that this process, once validated, does not need to be repeated for the full source code. Still, it is important to have a run time as low as possible, as it allows any revision of the process, particularly in its early stages, to be tested as quickly as possible.

8.3 Extended Rule Base

After the initial run of the rules on the proof-of-concept sample, it was concluded that there was a need to extend and customise the existing rule base. An effort limit of three working days (24 hours) was set for one person, with knowledge on the reengineering process, to improve the rule base. Given that the initial version of the rules was tested on the small BankSystem application only, the effort for the first large case study was expected to be higher than in subsequent projects, in which a more thoroughly tested base is available from the start. In the three days allocated it was possible to add ten more rules as well as to modify (i.e., generalise) four existing rules. As a result, the coverage was increased from 70% to about 80%. Any further improvements will usually take more effort because the cases left to be considered for creating new rules will be more and more specific, i.e., occurring less frequently and having less

impact in the overall coverage. The timing does not change perceptibly from the previous version.

8.4 Full Case Study

The following task was running the improved set of rules on the entire case study. As seen from Table 8.1, the coverage achieved is 75%, down 5% from the proof-of-concept. It is believed that this is due to the fact that, while technologically homogeneous, as an open source project the RapidMiner displays a variety of different individual coding styles. Nevertheless, an increase of 5% in coverage would result in an additional 18,500 lines to be transformed automatically, which for three person-days corresponds to 6,167 lines transformed per day, by far outperforming the manual migration estimate of 160 lines of code per day discussed in section 4.1.

For what concerns scalability, the total run time of about five hours is considerable, but follows an almost linear relationship (about 0.04 seconds per line of code in the proof-of-concept vs. 0.05 seconds in the full RapidMiner). The distribution repeats the established pattern. In particular, the redesign phase is relatively fast compared to the others, benefiting from the high level of abstraction of the graph model, while the phases working at code level have to deal with the full detail of the abstract syntax tree.

8.5 Code Quality

One of the main concerns with automated reengineering solutions is the resulting code quality, specifically in terms of readability and complexity. In order

to analyse this important aspect, Q-CARE [3] was used, a code certification tool from ATX, to check for code quality issues in source code vs. target code. With this, for the set of code quality rules existing in the tool, it is possible to attest if undesirable effects have occurred due to the whole approach. The tool results for the BankSystem application showed similar counts for quality rules violations for source vs. target code, but not an exact match. For instance, since the target code has more classes than the source one, as code is split in more files (classes), the rule "EachFileMustHaveHeader", which validates if files have a header comment, had a higher count in the target code. This happened due to the fact that the original files did not have a header comment, so one was not propagated to the target ones, but this could be avoided if the class creation templates always specified one if none existed in the source, for example retrieving information from the class name, and setting a creation date. Another rule that showed a significant different count between source and target was "UseFullyQualifiedImport", which checks if generic imports are being used without being necessary, but in this case it was the target that had a lower number of occurrences, since a reorganisation of import statements occurs in the transformation process. A screenshot of Q-CARE is presented in figure 8.2.

8.6 Threats to Validity

Since the work presented in this thesis constitutes original work done in the context of a PhD, the evaluation work was performed solely by an author of the approach. Due to his familiarity with all the process, including the experience in improving it with practical experiments, he is likely to have achieved better results in terms of rules design effectiveness (code coverage) than an external

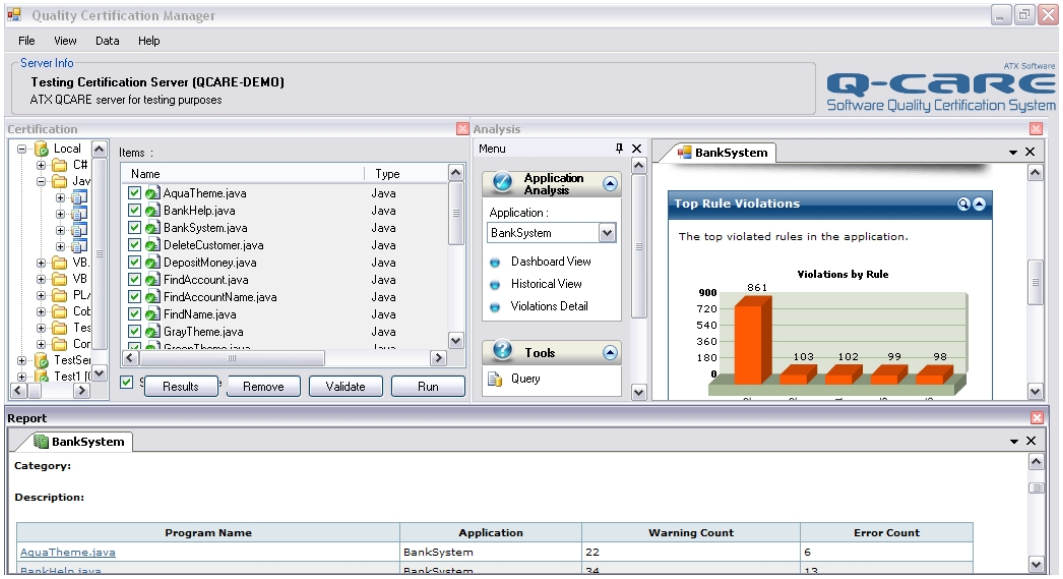

Figure 8.2: Q-CARE code certification tool

individual who is not versed in the process. Of course, this kind of activity, like many others, has its learning curve and, with experience, other software engineers can gain the same performance. In terms of the run-time performance or scalability aspects the above setting has a lower potential impact, as these are a consequence of the approach but could, in extreme situations (for instance very poor code pattern matching rules design) be affected to some extent. The code quality evaluation was performed using a tool that, whilst belonging to ATX, this dissertation's author has not participated in its development, nor rules design, so it is not susceptible of being affected by the above conditions.

8.7 Further Evaluation

Besides the various aspects that were given more focus in this thesis, there are others of relevance for evaluating the result of applying its approach, some of which can only be fully evaluated in the context of a real project. These include specific management, financial and infrastructural constraints, and aspects such as process flexibility (e.g. dividing the several steps between team

members with different expertise) could be analysed.

Addressing all of Rapid Miner code base could provide additional useful metrics, as full effort required, including for functional testing, which in the context of this work, was only done for the Banking System application, and would be a good setting to also attest project management specifics (e.g. team member specialisation and parallel tasks).

Evaluating the approach in other scenarios, such as having a source programming language from a paradigm other than OO, for example COBOL or PL-I, is a possible vector of future work, as it would allow, amongst other things, to verify the reuse of pattern matching, and graph transformation rules between different technologies.

Another important aspect to evaluate is source code quality as this is a major factor since it impacts software maintenance, which is known to be one of the most relevant costs in IT departments. The final quality will depend upon the original one, but there are measures that can be taken to get the best result. This is a concern that has been present during the approach and the prototype development, particularly regarding the nomenclature used in the generated code, which is based in the original one, and the code patterns used, which were based in Java's known best practices. A more formal evaluation of this aspect is planned as future work. One possible way to evaluate it is through the use of quality assurance tools, that can statically validate if the code complies to a set of rules. An initial evaluation of code quality, based in a quality assurance tool (Q-CARE), was already described in this dissertation. This can be complemented by a dynamic approach, in order to evaluate non-functional aspects such as performance, which be done by comparison between the original and transformed applications.

Chapter 9

Conclusions

The main contribution of this work is in the definition, application and evaluation of a generic methodology of architectural transformation to address the problem of migration towards a more modern architecture, with a particular emphasis on layered and service-oriented architectures. The method supports a process where migration is offered as a service including the customisation of a generic rule set to the application(s) at hand. With this preparatory step, the transformation achieves a high level of automation and allows for a cost-effective deep restructuring of the code towards the intended target whilst complying to important SOA principles, as opposed to a wrapping approach.

Code pattern matching and graph transformations are central to the four-step method which is applied in two iterations to achieve both technological and functional decomposition of the application. Due to its high level of abstraction, the redesign transformation at the core of the method is programming-language independent as well as highly scalable. The abstraction is enabled by a reverse engineering step which keeps only such details as are relevant to the transformation and could at times represent an entire class by a single node in the graph model. By working at this level of abstraction, it is also possible

to take advantage of graph/model analysis techniques.

The approach presented in this dissertation uses graph transformation for the central redesign step, where a graph model of the relevant source architecture is transformed by means of refactoring rules into the target architecture. Despite the relative complexity of graph transformation when compared to, for example, tree rewriting, it turns out that this step is highly scalable due to the benefits of a concise graph-based representation. The decision not to represent the entire detail of the source code in the graph, but only to the level required to support the redesign, is key to achieve this property.

A significant part of the effort in developing the tool support was required for the reverse and forward engineering steps. These are mostly generic and can be reused across different types of transformations. The more flexible rule-based approaches of XPath based pattern matching and graph transformation are reserved for the code annotation and redesign steps, which can be adapted for every new instantiation of the methodology. This is especially the case for the former, which is more dependent upon each application's specifics.

A prototype has been developed to apply this approach to a sequence of case studies. This allowed for the evaluation of the approach in regards to the requirements it was set up to address. The methodology is general enough to enable both technological and functional dimensions of migration, including the restructuring of applications for compliance with service-orientation principles. The evaluation also included the analysis of the properties of coverage, reuse and scalability, which produced very positive results.

The contribution of this work is a concrete process of addressing architecture transformation projects in a systematic way, with an emphasis on transformation to layered architectures and SOA migration. The study and comparison

described in chapter 3 showed that there are not many approaches that were developed allowing for SOA principles-compliance (restructuring approaches). Amongst these, the approach described in this dissertation stands out due to the deep restructuring it supports, its scalability, and support for automation.

Future Work

Plans for future work include the evaluation of the method and tool in a real project as well as its adaptation to other languages and architectural styles. Only in the context of a real project it is possible to evaluate management, financial and infrastructural constraints.

Particularly if of a large scale, the experience of such a real project would allow further work over the code annotation rule set, but could also contribute with improvements to the technique itself.

Developing a prototype of the approach for other scenarios, such as addressing different programming language paradigms besides OO, constitute possible future work. From an evaluation standpoint, this would allow verifying the reuse of pattern matching, and graph transformation rules between different technologies. From a business perspective, this would broaden the spectrum of projects that could be addressed.

As mentioned earlier in this thesis, important aspects such as the resulting source code quality are amongst those in which it will be relevant to evaluate and, if necessary, improve upon. This has an impact on maintenance so it is also a considerable factor when considering commercialisation. There were already several measures built into the prototype to achieve this, particularly regarding the nomenclature used in the generated code and the code patterns applied.

A different possible future work would be on enhancing the approach to take into concern matters of service discoverability. Devising techniques to work together with the migration process described in this dissertation, gathering meta data for publishing in a service registry, would add to its value.

Whilst the methodology was defined having layered and service-oriented architectures as target, it can be applied for different architectural transformations. Exploring its usage for migration to other intended architectural styles is a possible vector of future work.

Additionally, whilst the methodology is focused in architectural transformations, it can adequately be used for expressive code transformations that do not change the architecture level, but improve the code structure. Advanced refactoring could be specified as graph transformation rules, making use of information gathered by purposefully designed code annotation rules.

Bibliography

[1] ATX Software. http://www.atxtechnologies.co.uk.

[2] Henshin project. http://www.eclipse.org/modeling/emft/henshin/.

[3] Q-care. http://www.atxtechnologies.co.uk/other/development/qcare/q-care/.

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

[5] Amund Aarsten, Davide Brugali, and Giuseppe Menga. Patterns for three-tier client/server applications. In *Proceedings of Pattern Languages of Programs (PLoP)*, 1996.

[6] Charles Abrams and Roy W. Schulte. Service-oriented architecture overview and guide to SOA research. Technical Report G00154463, Gartner Research, January 2008.

[7] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Case study: Re-engineering c++ component models via automatic program transformation. *Information and Software Technology*, 49:275–291, March 2007.

[8] Altova. UModel UML software development tool. http://www.altova.com/products/umodel/uml_tool.html.

[9] Luís Andrade, João Gouveia, Miguel Antunes, Mohammad El-Ramly, and Georgios Koutsoukos. Forms2Net - Migrating Oracle Forms to Microsoft .NET. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 261–277. Springer-Verlag, 2006.

[10] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: A novel approach and a case study. In *Proceedings of International Conference Software Maintenance (ICSM)*, pages 357–366, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Ira Baxter, Christopher Pidgeon, and Michael Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[12] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[13] Garret Birkhoff. *Lattice theory*, volume 25 of *Colloquium publications*. American Mathematical Society, 1940.

[14] Dénes Bisztray and Reiko Heckel. Combining termination criteria by isolating deletion. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proceedings of International Conference on Graph Transformation (ICGT)*, Lecture Notes in Computer Science, pages 203–217, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] Dorothea Blostein and Andy Schürr. Computing with Graphs and Graph Rewriting. 29(3):1–21, 1999.

[16] Thierry Bodhuin and Maria Tortorella. Using grid technologies for web-enabling legacy systems. In *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice*, pages 186–195, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. Migrating interactive legacy systems to web services. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 24–36, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81:463–480, April 2008.

[19] Maurice M. Carey and Gerald C. Gannod. Recovering concepts from source code with automated concept identification. In *International Conference on Program Comprehension*, pages 27–36, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[20] S. Jeromy Carrière, Steven G. Woods, and Rick Kazman. Software architectural transformation. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pages 13–23, Washington, DC, USA, 1999. IEEE Computer Society.

[21] Mariano Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Control*, 14(3):209–231, 2006.

[22] Feng Chen, Shaoyun Li, and William Cheng-Chung Chu. Feature analysis for service-oriented reengineering. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 201–208, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Kunrong Chen and Vaclav Rajlich. Case study of feature location using dependence graph. In *International Workshop on Program Comprehension*, page 241, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[24] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7:13–17, January 1990.

[25] Sam Chung, Joseph Byung Chul An, and Sergio Davalos. Service-oriented software reengineering: Sosr. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, pages 172c–, Washington, DC, USA, 2007. IEEE Computer Society.

[26] James Cordy, Thomas Dean, Andrew Malton, and Kevin Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, 2002.

[27] Andrea Corradini, Ugo Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3 and 4):241–265, 1996.

[28] Katja Cremer, André Marburger, and Bernhard Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance*, 14(4):257–292, 2002.

[29] Félix Cuadrado, Boni García, Juan C. Dueñas, and Hugo A. Parada. A case study on software evolution towards service-oriented architecture. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications - Workshops*, AINAW '08, pages 1399–1404, Washington, DC, USA, 2008. IEEE Computer Society.

[30] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[31] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, San Francisco, CA, USA, 2002.

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

[33] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006.

[34] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[35] Mohammad El-Ramly, Eleni Stroulia, and Hani Samir. Legacy systems interaction reengineering. In Ahmed Seffah, Jean Vanderdonckt, and Michel C. Desmarais, editors, *Human-Centered Software Engineering*, Human-Computer Interaction Series, pages 316–333. Springer London, 2009. 10.1007/978-1-84800-907-3_15.

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

[37] Hoda Fahmy, Richard C. Holt, and James R. Cordy. Wins and losses of algebraic transformations of software architectures. In *Proceedings of International Conference on Automated Software Engineering (ASE)*, pages 51–60, Washington, DC, USA, 2001. IEEE Computer Society.

[38] Rudolf Ferenc and Árpád Beszédes. Data exchange with the columbus schema for c++. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 59–66, Washington, DC, USA, 2002. IEEE Computer Society.

[39] Ian Finley and Mike Blechar. Hype cycle for application development, 2011. Technical Report G00214153, Gartner Research, July 2011.

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

[41] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[42] Erich Gamma, Richard Helm, Ralf Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addison-Wesley, 1994.

[43] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering (SEKE)*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[44] Nicolas E. Gold, Mark Harman, David Binkley, and Robert M. Hierons. Unifying program slicing and concept assignment for higher-level executable source code extraction. *Software-Practice and Experience*, 35:977–1006, August 2005.

[45] Concettina Del Grosso, Massimiliano Di Penta, and Ignacio Garcia-Rodriguez de Guzman. An approach for mining services in database oriented applications. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 287–296, Washington, DC, USA, 2007. IEEE Computer Society.

[46] The Open Group. TOGAF - The Open Group Architecture Framework. http://www.opengroup.org/togaf/.

[47] He Guo, Chunyan Guo, Feng Chen, and Hongji Yang. Wrapping client-server application to web services for internet computing. In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing Applications and Technologies*, PDCAT '05, pages 366–370, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Thomas Haase. Model-driven service development for a-posteriori application integration. In *Proc. of International Conference on e-Business Engineering (ICEBE)*, pages 649–656, Washington, DC, USA, 2007. IEEE Computer Society.

[49] Ali Nasrat Haidar and Ali E. Abdallah. Composition and customization of web services using wrappers: A formal approach based on csp. In *Proceedings of the 2008 32nd Annual IEEE Software Engineering Workshop*, SEW '08, pages 187–194, Washington, DC, USA, 2008. IEEE Computer Society.

[50] Reiko Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 148(1):187–198, 2006.

[51] Reiko Heckel, Rui Correia, Carlos Matos, Mohammad El-Ramly, Georgios Koutsoukos, and Luis Andrade. *Software Evolution*, chapter Architectural Transformations: From Legacy to Three-tier and Services, pages 139–170. Springer-Verlag, 2008.

[52] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars: A constructive approach. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2:118–126, 1995.

[53] Tassilo Horn, Andreas Fuhr, and Andreas Winter. Towards applying model-transformations and -queries for soa-migration. In *Proceedings of MDD, SOA und IT-Management (MSI 2009)*, 2009.

[54] He Yuan Huang, Hua Fang Tan, Jun Zhu, and Wei Zhao. A lightweight approach to partially reuse existing component-based system in service-oriented environment. In *Proceedings of the 10th international conference on Software Reuse: High Confidence Software Reuse in Large Systems*, ICSR '08, pages 245–256, Berlin, Heidelberg, 2008. Springer-Verlag.

[55] Suhaimi Ibrahim, Norbik Bashah Idris, and Aziz Deraman. Case study: Reconnaissance techniques to support feature location using recon2. In *Asia-Pacific Software Engineering Conference*, page 371, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[56] Igor Ivkovic and Kostas Kontogiannis. A framework for software architecture refactoring using model transformations and semantic annotations. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 135–144, Washington, DC, USA, 2006. IEEE Computer Society.

[57] Vladimir Jakobac, Alexander Egyed, and Nenad Medvidovic. Improving system understanding via interactive, tailorable, source code analysis. In Maura Cerioli, editor, *Proceedings of Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *Lecture Notes in Computer Science*, pages 253–268. Springer-Verlag, 2005.

[58] JetBrains. IntelliJ IDEA. http://www.jetbrains.com/idea/.

[59] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 338–348. IEEE Computer Society Press, May 2002.

[60] Yiannis Kanellopoulos, Thimios Dimopulos, Christos Tjortjis, and Christos Makris. Mining source code elements for comprehending object-oriented systems and evaluating their maintainability. *SIGKDD Explor. Newsl.*, 8(1):33–40, 2006.

[61] Rick Kazman, Steven Woods, and Jeromy Carrière. Requirements for integrating software architecture and reengineering models: CORUM II. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pages 154–163, Washington, DC, USA, 1998. IEEE Computer Society.

[62] N. Kiesel, P. Klein, M. Nagl, and V. Schmidt. Verteilung in betriebswirtschaftlichen anwendungen: Einige bemerkungen von seiten der softwarearchitektur. In S. Jhnichen, editor, *Online '94 Congress VI*, pages C.620.01–C.620.29, 1994.

[63] Donald E. Knuth. Computer-drawn flowcharts. *Communications of the ACM*, 6:555–563, September 1963.

[64] Jun Kong, Kang Zhang, Jing Dong, and Guanglei Song. A graph grammar approach to software architecture verification and transformation. pages 492–497, Washington, DC, USA, 2003. IEEE Computer Society.

[65] Rainer Koschke and Jean-Francois Girard. An intermediate representation for reverse engineering analyses. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pages 241–250, 1998.

[66] Rainer Koschke and Jochen Quante. On dynamic feature location. In *Automated Software Engineering*, pages 86–95, New York, NY, USA, 2005. ACM Press.

[67] Georgios Koutsoukos, Luis Andrade, João Gouveia, and Mohammad El-Ramly. Service extraction. Technical Report D6.2a, SENSORIA Project, August 2006.

[68] Roger Lee, Haeng-Kon Kim, and Hae Sool Yang. An architecture model for dynamically converting components into web services. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 648–654, Washington, DC, USA, 2004. IEEE Computer Society.

[69] Leg2Net. From legacy systems to services in the net. http://www.cs.le.ac.uk/SoftSD/Leg2Net/.

[70] Timothy Lethbridge, Erhard Plödereder, Sander Tichelaar, Claudio Riva, Panos Linos, and Sergei Marchenko. The Dagstuhl Middle Model (DMM). http://www.ece.queensu.ca/hpages/courses /elec875/pdf/DMMDescriptionV0006.pdf.

[71] Grace Lewis, Edwin Morris, Dennis Smith, and Liam O'Brien. Service-oriented migration and reuse technique (smart). In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 222–229, Washington, DC, USA, 2005. IEEE Computer Society.

[72] Yan Liu, Qingling Wang, Mingguang Zhuang, and Yunyun Zhu. Reengineering legacy systems with restful web service. In *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, COMPSAC '08, pages 785–790, Washington, DC, USA, 2008. IEEE Computer Society.

[73] Michael Löwe, Martin Korff, and Annika Wagner. An algebraic framework for the transformation of attributed graphs. In Ronan Sleep, Marinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[74] Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *International Workshop on Program Comprehension*, pages 33–42, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[75] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[76] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society.

[77] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proceedings of International Conference on Graph Transformation (ICGT)*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002.

[78] Tom Mens, Gabriele Taentzer, and Olga Runge. Analyzing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

[79] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July/August 2005.

[80] Maxim Mossienko. Automated Cobol to Java recycling. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 40–50, Washington, DC, USA, 2003. IEEE Computer Society.

[81] Code Generation Network. List of code generators. http://www.codegeneration.net/generators.php.

[82] Jörg Niere and Albert Zündorf. Reverse engineering with fuzzy layered graph grammars. Technical report, University of Paderborn, 2003.

[83] SERC University of West Florida. Recon2. http://www.cs.uwf.edu/ recon/recon2/index.html.

[84] Object Management Group (OMG). Object Constraint Language specification. http://www.omg.org/spec/OCL.

[85] Object Management Group (OMG). Unified Modeling Language: Superstructure version 2.0. http://www.omg.org/spec/UML/2.0, August 2005.

[86] Organization for the Advancement of Structured Information Standards (OASIS). Reference model for service oriented architecture 1.0. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html.

[87] Michael P Papazoglou and Willem-Jan Van Den Heuvel. Service-oriented design and development methodology. *International Journal of Web Engineering and Technology*, 2:412–442, July 2006.

[88] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[89] Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In M. R. Sleep, M. J. Plasmeijer, and M. C. van Eekelen,

editors, *Term Graph Rewriting: Theory and Practice*, chapter 15, pages 201–213. Wiley, 1993.

[90] Denys Poshyvanyk, Andrian Marcus, Vaclav Rajlich, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *International Conference on Program Comprehension*, pages 137–148, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[91] Ganesan Ramalingam, Raghavan Komondoor, John Field, and Saurabh Sinha. Semantics-based reverse engineering of object-oriented data models. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 192–201, New York, NY, USA, 2006. ACM Press.

[92] Rapid-I. RapidMiner. http://sourceforge.net/projects/yale/.

[93] Maryam Razavian and Patricia Lago. A frame of reference for soa migration - appendix.
http://www.few.vu.nl/ mrazavi/SOAMigrationAppendix.pdf.

[94] Maryam Razavian and Patricia Lago. A frame of reference for soa migration. In *ServiceWave*, volume 6481 of *Lecture Notes in Computer Science*, pages 150–162. Springer-Verlag, 2010.

[95] Roy Martin Richards. Implementing user/computer dialogue in COBOL. *ACM SIGCSE Bulletin*, 19:15–19, June 1987.

[96] Bradley Schmerl, David Garlan, Vishal Dwivedi, Michael W. Bigrigg, and Kathleen M. Carley. Sorascs: a case study in soa-based platform design for socio-cultural analysis. In *Proceedings of International Conference on Software Engineering (ICSE)*, ICSE '11, pages 643–652, New York, NY, USA, 2011. ACM.

[97] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 3, pages 487–550. World Scientific, 1999.

[98] A. E. Scott. Automatic preparation of flow chart listings. *Journal of the ACM (JACM)*, 5:57–66, January 1958.

[99] SENSORIA. Software engineering for service-oriented overlay computers. http://www.sensoria-ist.eu/.

[100] Mary Shaw. What makes good research in software engineering. *International Journal of Software Tools for Technology Transfer (STTT)*, 4:1–7, 2002.

[101] Daniel Sholler. Hype cycle for application architecture, 2011. Technical Report G00213388, Gartner Research, July 2011.

[102] Harry Sneed. Integrating legacy software into a service oriented architecture. In *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 3–14, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[103] Harry Sneed and Stephan Sneed. Creating web services from legacy host programs. In *Proceedings of International Symposium on Website Evolution (WSE)*, pages 59–65, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[104] Gabriele Taentzer. AGG. http://tfs.cs.tu-berlin.de/agg/index.html, 2007.

[105] TeReSe. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, March 2003.

[106] The Eclipse Foundation. Eclipse. http://www.eclipse.org/.

[107] Mark van den Brand, Jan Heering, Paul Klint, and Pieter Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Programming Languages and Systems*, 24(4):334–368, July 2002.

[108] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *The Journal of Systems and Software*, 61:105–119, March 2002.

[109] W3C. XPath specification. http://www.w3.org/TR/xpath.

[110] Neil Walkinshaw. *Partitioning Object-Oriented Source Code for Inspections*. PhD thesis, University of Strathclyde, 2006.

[111] Mark Weiser. Program slicing. In *Proceedings of International Conference on Software Engineering (ICSE)*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[112] Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. A comparison of methods for locating features in legacy software. *The Journal of Systems and Software*, 65(2):105–114, February 2003.

[113] Norman Wilde, Juan A. Gomez, Thomas Gust, and Douglas Strasburg. Locating user functionality in old code. In *Proceedings of International Conference Software Maintenance (ICSM)*, pages 200 –205, nov 1992.

[114] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7:49–62, January 1995.

[115] Andreas Winter and Jörg Ziemann. Model-based migration to service-oriented architectures - a project outline. In Harry Sneed, editor, *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, pages 107–110. Vrije Universiteit Amsterdam, 2007.

[116] Bo Zhang, Liang Bao, Rumin Zhou, Shengming Hu, and Ping Chen. A black-box strategy to migrate gui-based legacy systems to web services. In *Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 25–31, Washington, DC, USA, 2008. IEEE Computer Society.

[117] Lu Zhang, Tao Qin, Zhiying Zhou, Dan Hao, and Jiasu Sun. Identifying use cases in source code. *The Journal of Systems and Software*, 79(11):1588–1598, November 2006.

[118] Zhuopeng Zhang, Hongji Yang, and William C. Chu. Extracting reusable object-oriented legacy code segments with combined formal concept analysis and slicing techniques for service integration. In *Proceedings of International Conference on Software Quality (QSIC)*, pages 385–392, Washington, DC, USA, 2006. IEEE Computer Society.

[119] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *International Conference on Software Engineering*, pages 293–303, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[120] Ying Zou and Kostas Kontogiannis. Web-based specification and integration of legacy services. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '00, pages 17–. IBM Press, 2000.