Software Reengineering at the Architectural Level: Transformation of Legacy Systems

R. Correia[1,2], C. Matos[1,2], M. El-Ramly[1], R. Heckel[1], G. Koutsoukos[2], L. Andrade[2]
[1]Department of Computer Science, University of Leicester, U.K.
[2]ATX Software, Lisboa, Portugal
{rmc20,cmm22,mer14,reiko}@le.ac.uk
{georgios.kousoukos,luis.andrade}@atxsoftware.com

Abstract

In this paper, we put forward a methodology for reengineering the architecture of a legacy software system. The proposed approach is not restricted to any specific source and target architectures, or programming language. It consists in (1) achieving a representation of the source code through its categorization and structuring, (2) transforming it into the new intended architecture, and (3) generating the code for the target platform. First, the code is categorized according to its purpose by pre-defined rules and represented as a model that is an instance of a type graph. Then, this representation is transformed into the intended target architectural paradigm using graph transformation techniques. The generation of the target code is not covered in this report but will be studied in the near future. The approach attempts to address problems that are repeatedly encountered in legacy reengineering industry projects.

1 Introduction

In this paper, we define a methodology for architectural transformation that addresses what has become one of the key challenges to any software development method: to define a process through which, given a legacy system with some known architecture, and a requirement to transform it into another known target architecture, we can extract code fragments that correspond to the different source architectural elements and use them to build the elements of the target architecture.

This methodology consists of three separate steps: reverse engineering of the source code to obtain a higher level representation, transformation of this representation to a similar one that considers the intended target architectural paradigm and target code generation from the target representation. The reverse engineering consists of abstracting the source code into a graph based model representation. This is achieved by applying categorization rules to the source code that take its purpose into consideration. The transformation into the intended target architectural paradigm is performed using graph transformation techniques. In this report the third step is not considered but it will be object of research in future work.

With the advent of every new technology, the need arises to reengineer, migrate or adapt the existing legacy applications. Over the last two decades, program transformation slowly emerged as a technology that can be employed to meet this need in many cases. More generally, program transformation established itself as a tool for software reengineering, maintenance and modernization.

A specific case of program transformation is source-to-source transformation which moved from successful research ideas (e.g., TXL [6] and ASF+SDF [4] systems) to powerful industrial tools

(e.g. DMS [3] and Forms2Net [25]). The problems of source-to-source transformation increase in complexity when we move up the abstraction ladder from code to detailed design, then to high-level design and finally to architecture. Transformation at the code level has met big success represented in the massive transformations of billions of lines of code for various purposes in different projects [5]. Transformations at the detailed design level have gained wide popularity as maintenance and reengineering practices since the publishing of Fowler's book on Refactoring [11]. Consequently, some tools arose that automated many refactorings [2], [1].

Design and architecture transformation can be desired to move from one paradigm to another (e.g. procedural to OO [24]), one platform to another (e.g. Oracle Forms to Microsoft .NET) or one architecture to another. At design and architecture level, transformation is much more difficult than lower levels and technology is still at an early stage. This is because transformation needs to happen at different levels of abstraction at the same time, while maintaining backward and forward traceability to the code. Successes at this level were limited to very specific source and target architectures and programming languages.

One example of target software architectures which is attracting many enterprises and researchers are Service-Oriented Architectures (SOAs). SOAs add new levels of complexity to program transformation. Such complexity levels are mainly twofold:

(1) Despaghettization - the "Horizontal" Dimension: The SOA paradigm implies that the services an application provides are "pure" and highly reusable. For instance, DB access statements and business logic are isolated from User Interface (UI) related statements. Typically, this is not the case in legacy applications where spaghetti-style code that mixes together database access, business logic, UI logic, validations and exception handling is a common practice. Fortunately, there already exist reengineering techniques and tools that contribute to such despaghettization. Unfortunately, an organized and systematic way for achieving considerable levels of automation for application transformation and despaghettization in the face of SOA is still missing.

(2) The "Vertical" and Coarse-Grained Dimension: The second level of complexity that SOAs introduce to the transformation of legacy applications is concerned with the "vertical" dimension of applications and the coarse-grained nature of services. The word "vertical" is concerned with the decomposition and allocation of application domain functionalities into the application structure. In other words, whereas typical transformation projects tend to maintain such structures and just address the problem of moving from one language or platform to the other, SOAs imply that such allocation of domain functionalities to software structures should also be recognized and redesigned in a SOA perspective. Moreover, typically such redesign should lead to coarse-grained services instead of the fine-grained code fragments found in most legacy applications. In our view extensions of current automation techniques should also be provided so that those transformations can be performed at a higher level of abstraction.

Our work in Leg2Net project [16] is the formalization of a methodology that addresses the problem of reengineering at the architectural level. The proposed methodology aims to facilitate automating the reengineering process and mitigating its risks. It does not limit itself to transformations between specific source and target architectures or programming languages. Our work on a second project, SENSORIA [23], aims to develop a methodology and tools for transformations from legacy systems to Service-Oriented Architectures.

The work described in this article addresses the horizontal dimension mentioned earlier that is needed for SOA (but is not restricted to that transformation or goal). The vertical dimension will be addressed in the future.

The rest of this paper is organized as follows:

- Section 2 talks about some case studies that motivated this research and introduces one used to validate the methodology;
- Section 3 explains the reengineering methodology and its subsections detail each of the parts;
- Section 4 presents the existing related work;
- Section 5 presents the conclusions and the future work.

2 Case Studies

In this section we present some common pattern of problems previously identified that motivated this research in the first place. Secondly some real world case studies provided by ATX Software are be addressed. Finally a small Java example will is introduced. This application is then be used in the rest of the paper to exemplify different steps of the approach and to validate them.

2.1 Common patterns of problems identified

In the beginning of this research project, some common patterns of problems were identified by experienced ATX members. These general patterns arise two research challenges: how to automate the code categorization and after, how to transform the source code to achieve the intended target architecture. We now present them and when possible include code examples:

1. This is the most common case where the source code has mixed (spaghetti style code) user interface, business logic and database access and we need to separate the different layers. This pattern is illustrated in Figure 1.

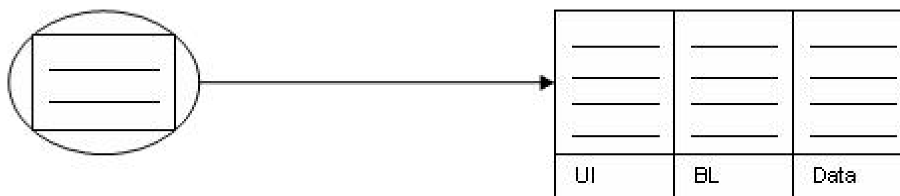

Figure 1: Spaghetti Code separated into different layers

   In the previous case we want to separate the user interface code from the data management one. An example of source code is presented in Figure 2.

2. This pattern refers to having mixed code where you can define two different services from it, as illustrated in Figure 3.
   We do not address this problem in the current work since it is part of the Vertical dimension previously discussed in the introduction.
3. The third case refers to code structured having validations separated from business logic. In the target architecture is necessary to combine the validation and the business logic. This situation is often found in source code that have pure business logic validations like the amount of money a client can withdraw per day mixed in the user interface code. If one wants to reengineer such a code to a pure three tier architecture, the combination of these validations and the business logic code is mandatory. This example is showed in Figure 4.
4. There are also concerns about hidden semantics that exist in some applications, based on underlying platform programming API´s, frameworks and idiosyncrasies. We are talking

```
\\concern = Data
 void populateArray() {
        try {
            fis = new FileInputStream("Bank.dat");
            dis = new DataInputStream(fis);
            //Loop to Populate the Array.
            while(true) {
                for(int i = 0;i < 6;i++) {
                    records[rows][i] = dis.readUTF();
                }
                rows++;
            }
        }
\\concern = User Interface
        catch(Exception ex) {
            total = rows;
            if(total == 0) {
                JOptionPane.showMessageDialog(null, "Records File is Empty.
                \nEnter Records First to Display.", "BankSystem - EmptyFile",
                JOptionPane.PLAIN_MESSAGE);
                btnEnable();
            }
\\concern = Data
            else {
                try {
                    dis.close();
                    fis.close();
                }
                catch(Exception exp) { }
            }
        }
    }
```

Figure 2: Java Spaghetti code example

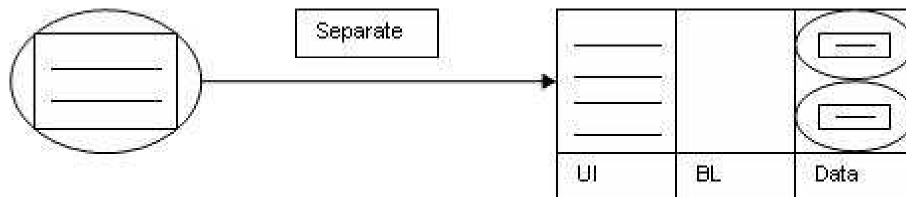

Figure 3: Different services mixed in the code

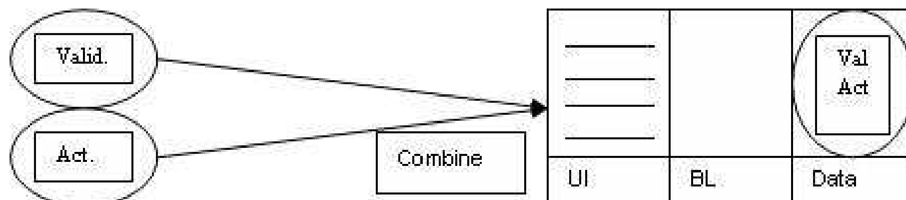

Figure 4: Different services mixed in the code

about, for example, when you save an Oracle Form and without any explicit ”COMMIT” command executed, new data is committed in the database. When transforming this kind of applications the hidden semantics have to be substituted by explicit code in the target architecture, considering that the target programming language does not behave accordingly.

2.2 Industrial tool examples - Forms2Net

While the research was taking place, ATX Software successfully launched a commercial tool [25] that automates the transformation of applications built in Oracle Forms into .Net web based or Windows Forms ones. During the development of this tool, some challenges for our research were encountered and here we resume them. These examples were particularly important because they showed the need for the transformations not being behavior-preserving in certain situations.

1. Alert messages included in routines called by the main program. While in Oracle Forms the routine is called and the main program stops until the user answers, in a web based environment this behavior is naturally undesirable. The connection between the end user and the machine where the main program is running cannot be held for long time, since this would be a costly waste of computer power and bandwidth. A simple example of code before and after conversion is presented in Figures 5 and 6 respectively:

```
PROCEDURE IN_ANUNC.KEY-DELREC
BEGIN
    :global.w_question:='Confirm record delete?';
    answer_yesno('W',:global.w_question,'','');
    if :global.w_conf='YES' then
       delete_record;
       commit;
       bell;
       message('E','QV365','','');
    end if;
END;
```

Figure 5: Original code for calling routine in Oracle Forms

```
public void Inanunc_KeyDelrec() {
    Globals.WQuestion = "Confirm record delete?";
    Model.LIBL45.AnswerYesNo("W", Globals.WQuestion, "", "");
      Toronto.Controllers.Messages.MessageAlWarningRe.ButtonPressed
      += new MessageBoxExEventHandler(Inanunc_KeyDelrecResponse);
}
public void Inanunc_KeyDelrecResponse(object sender,
MessageBoxExEventArgs args)
{
      Model.LIBL45.AnswerYesNoResponse(args);
      if ((Globals.WConf != null) && (Globals.WConf.Equals("YES")))
      {
          this.Model.InAnunc.Delete
          (this.Model.InAnunc.CurrentRowAdapter);
          this.Model.DoCommit();
          Model.LIBL45.Mensagem("E", "QV365", "", "");
      }
}
```

Figure 6: Converted code for calling routine in .Net optimized for a web based environment

2. There are also situations where cycles of updates with user interaction are carried out in Oracle Forms where the the program interactively prompts questions to the user and when the answer is received, the database is updated. One solution is to break the loop into two procedures and create a window for update confirmation: the first call identifies records to

be updated, these records are presented for user confirmation and finally the second call performs the updates. An example of code before conversion is presented in Figure 7:

```
declare
  cursor c1 is
    select (...)
begin
 while c1%found loop
    if some_condition then
      answer_yesno('W',:global.w_question,'','');
      if :global.w_conf != 'YES' then
        update statement;
      end if;
    end if;
    fetch c1 into
      c1_number;
 end loop;
end;
```

Figure 7: Original code for cycle of updates in Oracle Forms

2.3 Java application case study

In this paper a small Java application is used as an running illustrative example (BankSystem). The example was carefully chosen in order to illustrate the kind of spaghetti code that is typically found in legacy applications, which should often be transformed in more adequate horizontal layering. This application is a sample of a bank system with around twenty classes and 3 KLOCs. Its architecture is 2-tier, having the business logic mixed with the UI and Data layers. The UI is implemented on Java Swing and the data store is based in a file system. The application allows the most common banking operations such as new account creation, deposit money, withdraw money and manage customer details. The examples given in this paper address the deposit money operation. The target architecture will be 3-tier. This implies that the transformation will need to carry out code ”despaghettization”.

3 Reengineering methodology

The approach proposed in this paper, as shown in Figure 8, is a reengineering solution composed of three steps derived from the Horseshoe Model [15]:

1. Reverse engineering, in order to achieve a representation of the Source Code through its categorization and structuring;
2. Transformation techniques to achieve a representation of the Target Code from the Source Code representation previously accomplished;
3. Generation of the Target Code based on the combination of the Source Code and the Target Code representation previously achieved.

In the reverse engineering process, the Source Code is divided into blocks that are categorized according to their purpose, for example User Interface Action or Data Definition, using the information contained in the Categorization Rules. The output, called Program Representation, is an abstraction of the Source Code regarding its categories and control/data dependencies.

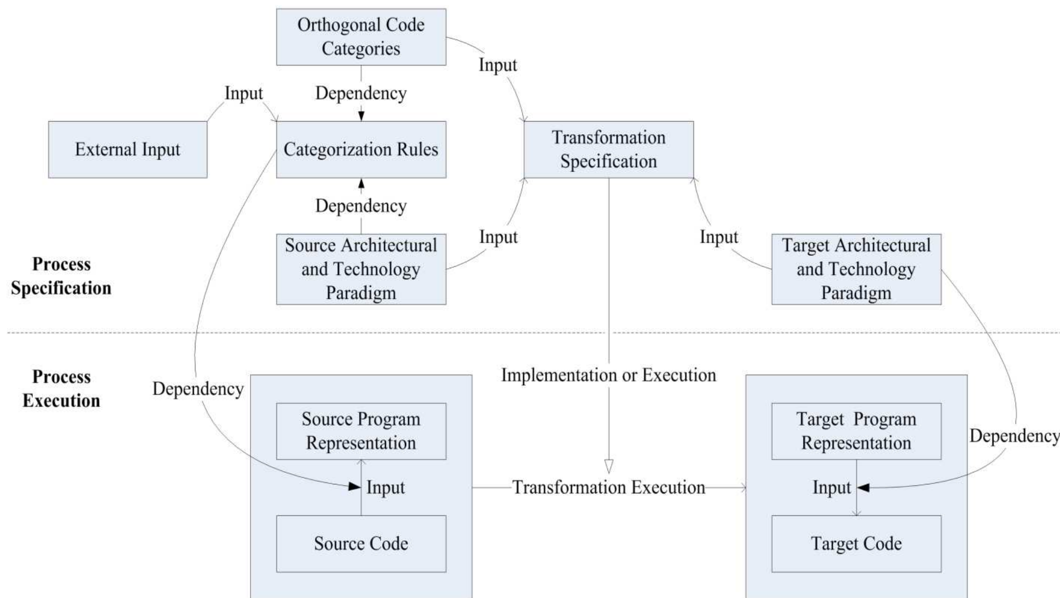

Figure 8: General Approach

The transformation techniques are based in graph transformations. Some of these transformations are Refactoring primitives.

The Target Code can be reached by code generation techniques using the Source Code and the transformation output.

A list of the existing code categories is held in the Orthogonal Code Categories block. The Architectural and Technology Paradigm is composed by a model for the programming language paradigm. The Categorization Rules block is the list of rules for classifying the Source Code according to the pre-existing categories. The Categorization Rules depend not only on the Orthogonal Code Categories but also on the information included in the Architectural and Technology Paradigm. The Source/Target Architectural and Technology Paradigm pair determines the rules that can be used to transform the Program Representation. These rules are defined in the Transformation Specification block.

In Figure 8 there are two different levels: Process Specification and Process Execution. The first corresponds to the methodology level and the later to the implementation level.

Each part of the figure is explained in detail in the following subsections, starting with the Process Specification level.

3.1 Orthogonal Code Categories

This block is a repository of the existing categories used to classify the Source Code. These code categories are used independently of the Architectural and Technology Paradigm. The Orthogonal Code Categories are closely related with the semantics of the Source Code.

They can be divided in two types:

- composed by a concern and a role
- connectors representing links between concerns

Concerns are conceptual classifications of code that are assigned regarding its goal. A non-exclusive list of identified concerns is:

- User Interface
- Business Logic
- Data

Roles are classifications of code according to its execution processing. Some of the identified roles are:

- Definition
- Action
- Validation

Concerns and roles are transversal concepts. A code category of this type can consist of any combination of the two, for example: Business Logic Action or User Interface Validation.

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

- Data storage/retrieval
- Network/communication
- Control: UI to BL
- Control: BL to UI
- Control: BL to Data
- Control: Data to BL

3.2 Architectural and Technology Paradigm

The Architectural and Technology Paradigm contains a model for the programming language paradigm, which is specified by means of a type graph. This model is called the Program Representation Graph (PRG). In Figure 9, it is possible to see a simplified model for OO. This is an extension of the type graph presented by Tom Mens et al in [20]. It was necessary to extend that model in order to introduce classification attributes and the notion of code block. This had to be done because we have the need of lower granularity level than the method for the process of code classification. Additionally, we have included the concepts of *Component* and *Connector* that allow us to represent the mapping between the programming language elements and the system architecture. During a transformation, we may have components and connectors that belong either to the source or the target architectures. For instance, after some transformation rules have been applied there may be still a component of the source architecture and already exist a component of the target. The concept *Stage* was added to cope with those intermediate phases. For the above example, the first component would be connected to the source stage whereas the latter would be connected to the target stage.

Since it is necessary to keep traceability to the code in order to facilitate the transformation / generation process, a method to associate it to the PRG has to be considered. Given that we want the methodology to be as programming language independent as possible we will not link the PRG directly to the source code. Hence, we are studying two solutions to address this issue:

1. A native representation based on an Abstract Syntax Tree (AST)
2. A graphical representation using a metamodel for the programming language

ASTs are very well known representations of source code and allow the referencing that we need. Programming language metamodels are perhaps more difficult to define but are also a solution to take into account. For Java, we can obtain the AST using an Eclipse API from Java Development

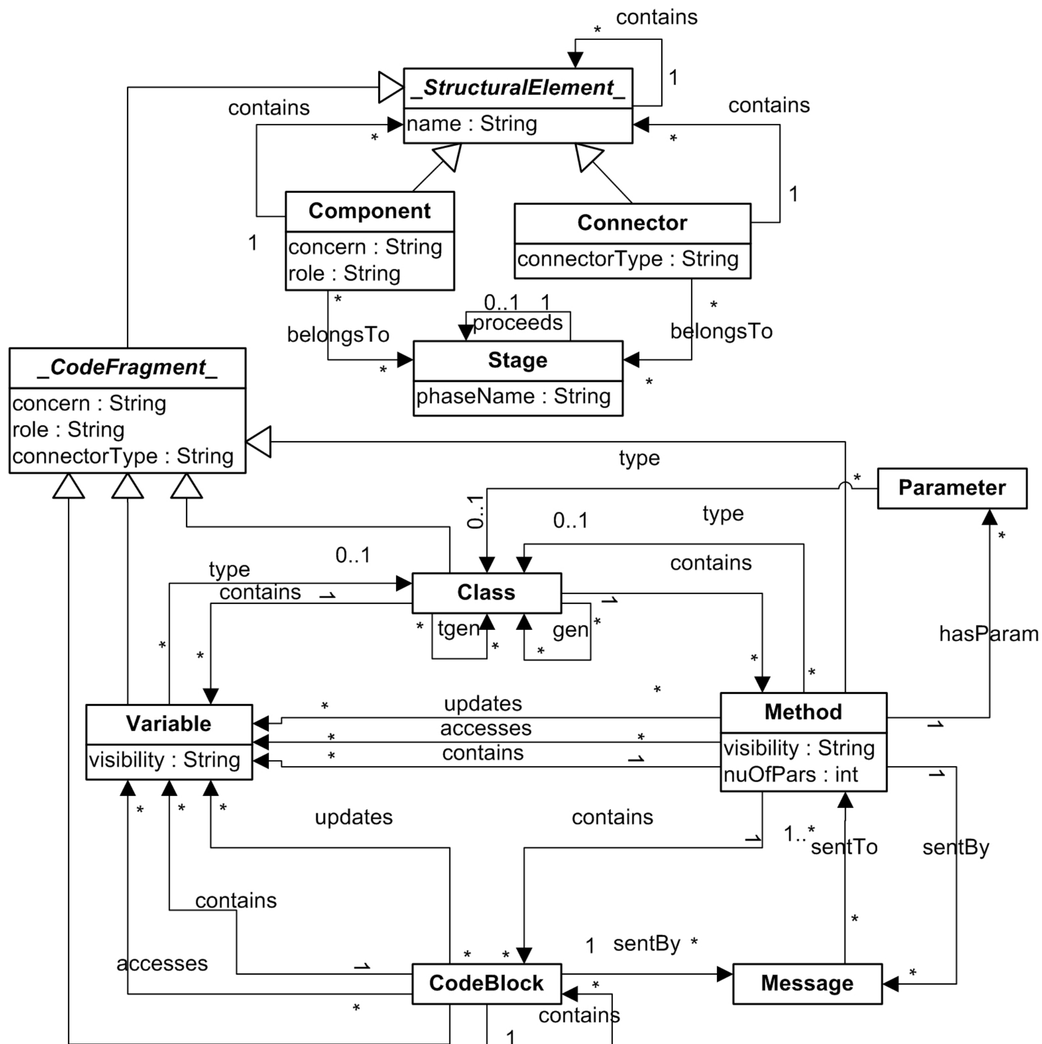

Figure 9: Program Representation Graph (type graph) for the OO paradigm (simplified). Note that, in order to be more clear visually, the abstract elements have extra '_' in their name as prefix and postfix in addition to the usual italic

Tools (JDT) [8]. As for the metamodel, an Eclipse Modeling Framework (EMF [7]) based metamodel for Java called Java EMF Model (JEM) is part of the Eclipse's Visual Editor project, but is still in its early development stages [9].

A different issue is the type of relation between the PRG and the model used to reference the source code. We are considering the following alternatives:

1. An extension to the type graph (PRG) by adding an attribute to the abstract class *CodeFragment* that uniquely identifies the equivalent element in the source code reference model
2. The use of an association model that defines the mapping between elements of the PRG and corresponding elements of the source code reference model

The first alternative is a straight-forward solution that has a tight relation between both models. This is a drawback because with this option the PRG will depend on the chosen source code reference model. Its main advantage is the simplicity of implementation where no extra models are

necessary. To use this solution the only extension necessary is an extra attribute in the abstract class *CodeFragment* of the type graph. The second alternative has greater flexibility because it uses an association model to link the source code reference model and the PRG. This is more flexible than the first alternative because it allows the PRG to be independent of the chosen source code reference model. However, this flexibility adds complexity since a model for the association is required. By

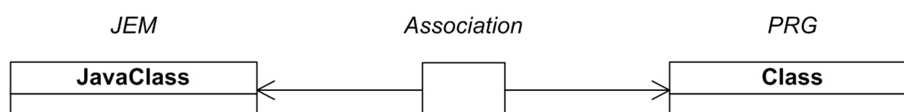

Figure 10: Example using JEM as the source code reference model and an association model to the PRG

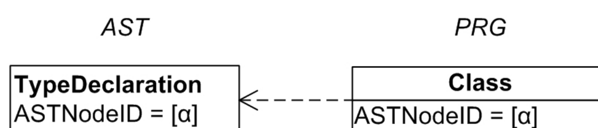

Figure 11: Example using the AST from Eclipse JDT and an extra attribute in the PRG to act as link (the dashed arrow is only for illustrative purposes)

considering alternatives for the source code reference model and the relation model used to link it to with the PRG, we get four possible options. We present two of the alternatives in Figures 10 and 11. Figure 10 shows an example using JEM as the source code reference model and an association model to the PRG for the element *Class*. Figure 11 presents an example using the AST from Eclipse JDT and the PRG extended with an attribute to reference it.

For simplicity reasons, in the rest of the paper only one of the possible four solutions will be used: the AST as source code reference and the extended PRG to provide the link between them (as in Figure 11).

3.3 External Input

In order to achieve greater correctness in the categorization of source code, the input of a programmer that has deep knowledge of the application or even a higher level business person that works as an end user or is familiarized with the application is of great importance. Taking this issue into consideration, the methodology has been thought to have means of human input be provided into the categorization rules. This block has not been thoroughly thought yet but we believe that will play an important role in the future, namely in the vertical categorization of the applications.

3.4 Categorization Rules

The rules that are used in the categorization process are based on the source code characteristics and are applied over the AST. The definition of these rules is not closed but some of them were already obtained. The following examples can be given in an informal way:

1. Statements that consist of variable/attribute declarations for a type that is known to belong to a certain concern, will be categorized as belonging to the same concern and having the role

Definition.
Example: the Java statement `'private JLabel lbNo;'` is categorized as UI Definition because it is known that JLabel belongs to the UI concern;

2. Attributions to variables/attributes that are known to belong to a certain concern and whose right hand side only includes the use of elements (e.g. variables or method invocations) that belong to the same concern, will have that concern and the role Definition.
   Example: the Java statement `'lbNo = new JLabel ("Account No:");'` is categorized as UI Definition because it is known that the attribute lbNo and the JLabel method/constructor invocation belongs to the UI concern;
3. Variables/attributes/parameters definition/attribution that are used to store values directly from Data Action methods/functions belong to the Data Action category.
   Example: the Java statement `'records[rows][i] = dis.readUTF();'` belongs to the Data Action because the readUTF operation is know to belong to that category.

The rules will have to be applied in multiple-pass. The reason for this is that the application of a rule can enable the application of another. An example for this need can be given using rule number 2: if a method invocation that exists in the right hand side of the attribution is not yet categorized, the rule will not be applied. However, after some other rule categorizes the method, rule number 2 can be applied. The implementation of an engine that supports the categorization process will have to contain stop conditions.

The categorization rules are defined formally with left hand side (LHS) and right hand side expressions (RHS). The LHS consists in the prerequisites that must be satisfied in order to apply the rule. The RHS is the result of the rule application. The expressions are based on the architectural and technology paradigm models. Both in the LHS and RHS there can be elements from the AST and the type graph. An example of rule definition and the result of applying it is presented in the next subsection.

3.4.1 Example

The rule number 1 (introduced in the previous subsection) can be represented as shown in Figure 12 for the declaration of attributes of type JLabel. The LHS expresses the type of nodes that are matched by the rule: all AST nodes of type *FieldDeclaration* that have the base type *SimpleType* and type name *JLabel*. In the AST, a field declaration statement has a list of children; each element corresponds to an attribute being declared in the statement. These are called declaration fragments. The RHS shows that the rule application results in the creation of a node of type *Variable* in the instance graph for each of the declaration fragments. This node will have the attributes *concern* and *role* with the values "UI" and "Definition", respectively. The attribute *name* of the new graph nodes is the same of the declaration fragments in the AST. The relation between the graph node and the AST is made via the *ASTNodeID* attribute.

An example of the application of this rule can be seen by relating Figures 13 and 14b. The LHS of rule number 1 is matched in the AST (Figure 14b) with the variables instantiated as follows: [α] = "ASTNode0010", [β] = "ASTNode0002" and [X] = "lbNo". The result of applying the rule, as stated by the RHS, is the creation of the *Variable* element in the PRG with *name* = "lbNo", *concern* = "UI", *role* = "Definition" and *ASTNodeID* = "ASTNode0002" (Figure 13). This last attribute of the new graph element is the one that specifies its relation to the AST.

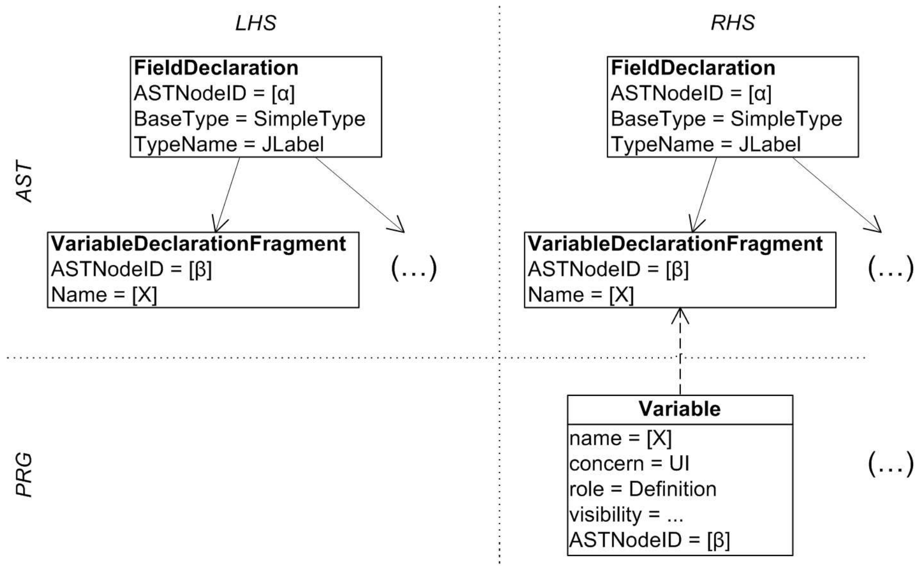

Figure 12: Categorization rule 1 (Variable/attribute declaration of known type) example definition for the JLabel type. (The dashed arrow is used in this figure only to make the relation between the graph and the AST more visually explicit.) Note that the text between square brackets represents variables

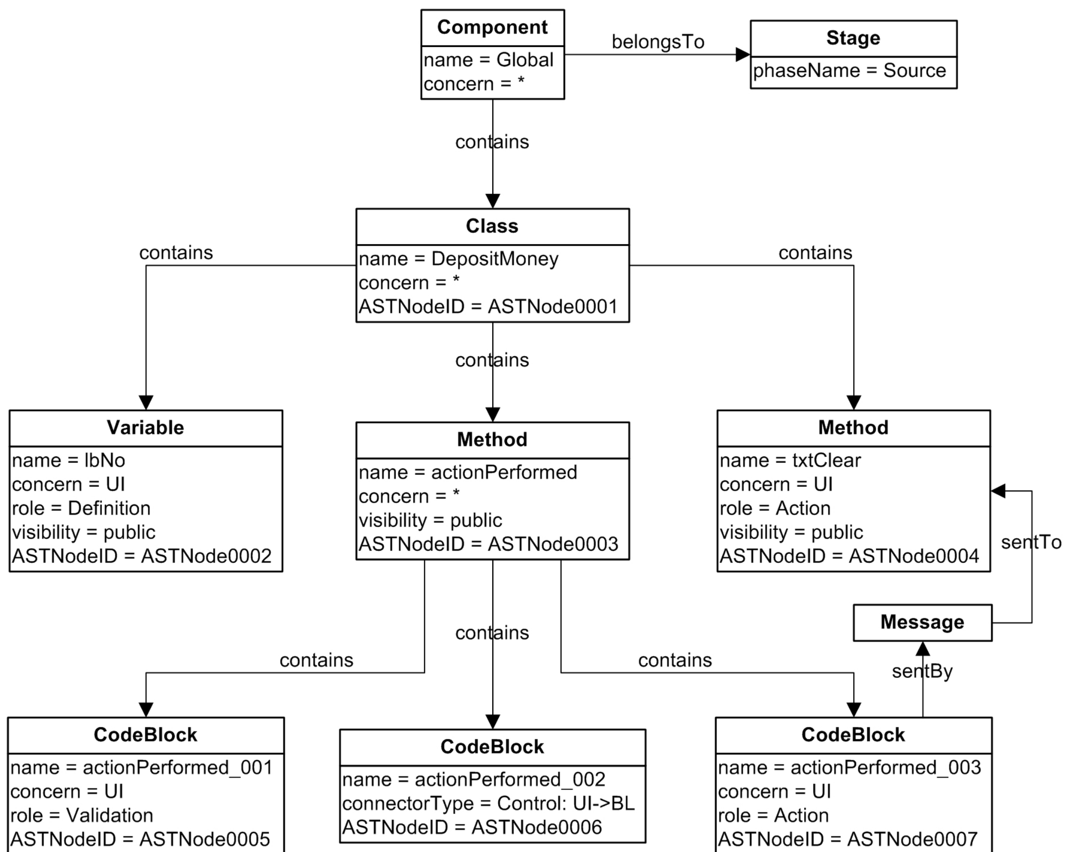

Figure 13: Graph representing a subset of a Java sample application

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DepositMoney extends JInternalFrame
       implements ActionListener { // ASTNode0001
  private JLabel lbNo /* ASTNode0002 */, lbName,
                lbDate, lbDeposit; // ASTNode0010
  private JTextField txtNo, txtName, txtDeposit;
  private JButton btnSave, btnCancel;
  // (...)
  public void actionPerformed(ActionEvent ae) {
                                      // ASTNode0003
    Object obj = ae.getSource();
    if(obj == btnSave) { // ASTNode0005
      if(txtNo.getText().equals("")) {
        JOptionPane.showMessageDialog(this,
                "Customer id", "Empty",
                JOptionPane.PLAIN_MESSAGE);
        txtNo.requestFocus();
      }
      else {
        if(txtDeposit.getText().equals("")) {
          JOptionPane.showMessageDialog(this,
                  "Amount", "Empty",
                  JOptionPane.PLAIN_MESSAGE);
          txtDeposit.requestFocus();
        }
        else { // ASTNode0006
          editRec();
        }
      }
    }
    if(obj == btnCancel) { // ASTNode0007
      txtClear();
      setVisible(false);
      dispose();
    }
  }
  //(...)
  void txtClear() { // ASTNode0004
    txtNo.setText("");
    txtName.setText("");
    txtDeposit.setText("");
    txtNo.requestFocus();
  }
  //(...)
  public void editRec () {
    //(...)
  }
  //(...)
}
```

(a) Example source code

```
PACKAGE: null
IMPORTS(3)
TYPES(1)
  TypeDeclaration
    ASTNode0001
    type binding: DepositMoney
    BODY_DECLARATIONS(6)
      FieldDeclaration
        ASTNode0010
        TYPE
          SimpleType
            type binding:
            javax.swing.JLabel
        FRAGMENTS(4)
          VariableDeclarationFragment
            ASTNode0002
            variable binding:
            DepositMoney.lbNo
      (...)
      MethodDeclaration
        ASTNode0003
        method binding:
        DepositMoney.actionPerformed()
        BODY
          IfStatement
            ASTNode0005
            IfStatement
              EXPRESSION
              THEN_STATEMENT
              ELSE_STATEMENT
                IfStatement
                  EXPRESSION
                  THEN_STATEMENT
                  ELSE_STATEMENT
                    ASTNode0006
          IfStatement
            ASTNode0007
      MethodDeclaration
        ASTNode0004
        method binding:
        DepositMoney.txtClear()
      (...)
```

(b) Example AST

Figure 14: Source code and AST extracts from a Java sample application.

3.5 Transformation Specification

The Transformation Specification provides rules for transforming a system that complies with a certain Architectural and Technology Paradigm into a different one.

The rules are specified as Graph Transformation Rules (GTR). Each GTR is composed by a Left-Hand Side (LHS) and a Right-Hand Side (RHS) instance graphs over the same type graph. The LHS

of the rule specifies the pre-conditions that must be satisfied so that the rule can be applied. The RHS corresponds to the post-conditions of applying the rule. To enrich the rules, it is also possible to specify Negative Application Conditions (NACs). A NAC is an instance graph (also over the same type graph as both the LHS and RHS) that denotes a pre-condition that must not be satisfied. Thus, the transformation rule is only applied when the following conditions are true:

- There is an occurrence O of the LHS in the given graph G;
- The occurrence O does not satisfy any of the NACs.

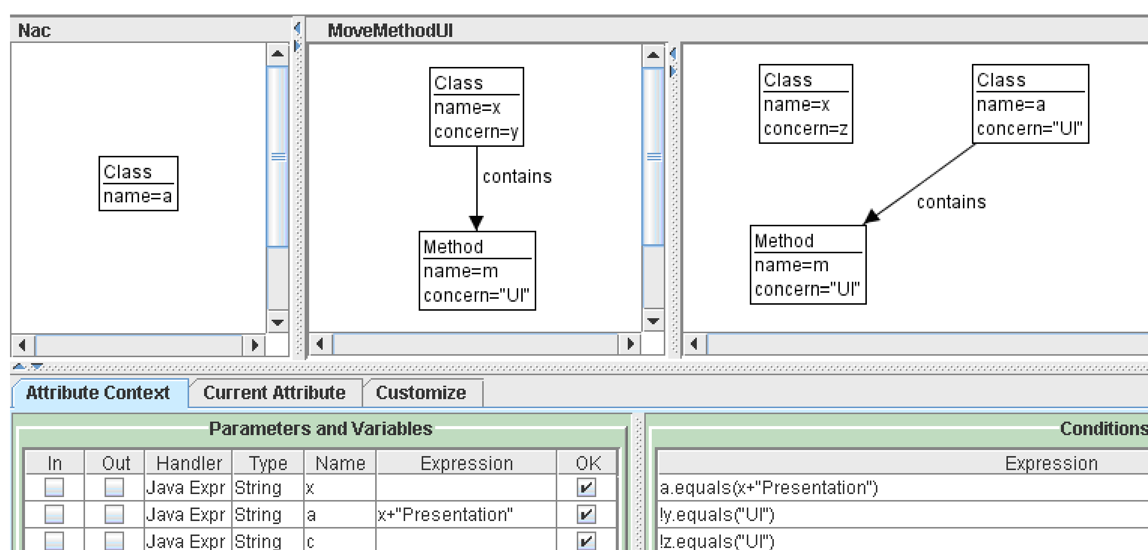

Figure 15: Move Method UI transformation rule: The leftmost graph corresponds to the NAC, in the middle is the rule LHS and the rightmost graph is the rule RHS. The expressions present in the graphs are defined below them. Note that the components/connectors are not shown here to simplify the rule presentation.

These graph related concepts are described in [13].

An example of transformation rule specification is given in Figure 15. Note that not all the NAC's for this rule are specified here to keep the example simple.

This is a similar approach to what is being used in refactoring research [19]. However, as it was presented in example 2 of subsection 2.2, refactoring rules are not enough for all reengineering purposes because sometimes it is necessary to perform transformations that are not completely behavior-preserving. An example of this is when we want to transform a legacy client-server system into a web-based application. The UI has to be changed because of the differences in the user communication paradigm between these different Architectural Styles. For instance, in the legacy application we may have a feature that performs a database query and then asks a question to the user, waits synchronously for the answer and then, based on the user input, updates a row in the database. To transform this code to a web-based system, it is not enough to separate the UI from the Data access layer. Due to the way that requests are processed on the web, we have to transform the UI in such a way that the communication will be asynchronous.

3.6 Program Representation

The Program Representation is an abstraction of the code. It is achieved by mapping the code into the categories defined in the Orthogonal Code Categories, according to the Categorization Rules, forming a structure that stores that information together with control/data dependencies. This representation keeps traceability to the code in order to facilitate the transformation / generation process.

The Program Representation uses an abstract notation and a concrete one. The abstract notation is an instance of the PRG while the concrete notation is based on an extended class diagram. The first will be the one used for the transformation purposes while the latter is more suitable for visualization. The abstract notation, as it includes many details in a graph form, can be very complex especially if the size of the system being reengineered is considerable, and thus very difficult to read for a human. The concrete notation also includes good level of detail but can be seen in two different levels. For instance, it is possible to see the class diagram first and only read the relevant extended parts. It is also more intuitive for people in general to understand a class diagram than a graph.

For our methodology, it is necessary to have traceability to the code in order to allow the transformation. The use of an existing Architecture Description Language (ADL) was considered for the concrete notation, however, due to the distance that ADLs have to the source code, that possibility was discarded [18]. In the following subsections both notations are shown in detail.

3.6.1 Abstract notation

For the abstract notation we are using a graph that is an instance of the type graph previously defined and shown in Figure 9, the PRG, where the code is categorized and its dependencies are defined. For now, the mapping between code elements and components and connectors is performed manually. and An example can be seen in Figure 13. The value ”*” for the attribute ”concern” means that the element contains more than one concern. For example, the ”actionPerformed” method contains three code blocks that include the concern ”UI” and the connector ”Control: UI -> BL”. In these situations we are omitting the attribute ”role” for simplicity.

Figure 13 is an instance of the extended type graph with the *ASTNodeID* attribute used to reference the corresponding AST node. This graph is obtained from the AST presented in Figure 14b. The corresponding source code can be seen in Figure 14a.

3.6.2 Concrete notation

As mentioned before, the concrete notation is based on an extended class diagram. This extension to the class diagram allows us to reach a lower level of granularity in order to be possible to classify pieces of code smaller than a Method. This extension is only necessary if the Class or the Class Member is composed by Code Blocks with different categorizations. It also implements some hierarchical structure necessary to know to which Class or Class Member a Code Block belongs to. The need for this information has to do with the fact that when the Transformation takes place, one will have to deal with the implications of breaking up code structures (Classes, Methods) and cannot just think in terms of simple pieces of code. From now on in this paper, for simplicity reasons, whenever a piece of code that can be a Code Block, a Class or a Class Member is referred, it will be called a Code Fragment which is described as follows.

Class / Class Member: <Code Block N>;
<Code Block N+1>;

<Code Block N+2>.

Code Fragment:

<AST Node ID>;
<Method / function invocations and external data usage>;
<Category>.

Here the code is uniquely identified, categorized and its dependencies are defined. Regardless of being a Class Member or a Code Block, the starting and ending line and column identify precisely which part of the code is being addressed and its Category. All its dependencies are also detailed here, whether they are control or data dependencies, as method / function invocations and external data usage. In this section only some of the elements of the Class "DepositMoney" are being presented, namely: the attribute "lbNo" and the methods "txtClear" and "actionPerformed". The attribute "lbNo" corresponds to a label that exists in the UI - it is the label that states 'Account No:' before the text box that prompts for the customer account number to which the deposit money operation is being done. The method "txtClear" has the goal of clearing all the input fields for the deposit money window. The "actionPerformed" method is called each time a user interface event is triggered, for example, when the customer account number field loses the focus.

Part of the source code shown in Figure 14a is represented in Figure 16 in the concrete notation.

3.7 Transformation Execution

The Transformation Execution applies the rules defined in the Transformation Specification to the Source Program Representation to obtain the target one.

If the programming language is also being changed in a concrete reengineering, it will be necessary to include a translation/generation step.

The example graph for the BankSystem sample application seen previously (Figure 13) is a candidate for the application of the Move Method UI transformation. This transformation is an example of a rule that contributes to the Horizontal layering of the application.

As we can see by the transformation rule specification, this graph has an occurrence of the LHS and is not matched by any NAC. As a result, we can apply the rule, obtaining the graph show in Figure 17.

A new class "DepositMoneyPresentation" was added and the method "txtClear" was moved to there from the class "DepositMoney". The Target Code for this particular example is not presented in this paper since the code changes are minimal: the code from method "txtClear" is not changed but only moved to the new class. The other modification that results from this transformation is that, now, the Code Block "actionPerformed_003" has to access "txtClear" in a different class.

4 Related work

As stated in the Introduction, program transformation can occur in different levels of abstraction. The source-to-source level of transformation is the most established both in research and in industrial implementations. There are several research ideas, e.g. TXL [6] and ASF+SDF [4], which led to successful industrial tools, e.g., DMS [3] and Forms2Net [25]. 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 [11] are driving the implementation of functionalities that automate detailed design level transformations. These are mainly integrated

DepositMoney
lbNo : JLabel
txtClear() actionPerformed()

(a) Class model

lbNo:
ASTNode0002;
Null;
UI, Definition.

txtClear:
ASTNode0004;
txtNo.setText(String),
txtName.setText(String), txtDeposit.setText(String), txtNo.requestFocus();
UI, Action.

actionPerformed:
<actionPerformed_001>;
<actionPerformed_002>;
<actionPerformed_003>.

actionPerformed_001:
ASTNode0005;
btnSave, txtNo.getText(),
txtNo.requestFocus(),
txtDeposit.requestFocus(),
JOptionPane.showMessageDialog(String);
UI, Validation.

actionPerformed_002:
ASTNode0006;
editRec();
Control: UI to Business Logic.

actionPerformed_003:
ASTNode0007;
btnCancel, txtClear(), dispose();
UI, Action.

(b) Extension of class model

Figure 16: Concrete notation example - class diagram with extension

in development environments as is the case of Eclipse [1] and IntelliJ [2]. However, there is still a lot of ongoing research in this area, for instance, in the determination of dependencies between transformations [20]. At the architectural level of program transformation there is some important research, e.g. the work in the Software Engineering Institute of CMU [15], but the industrial cases have been limited to specific source and target architectures and programming languages.

Regarding the area of source code representation, there is a lot of research. Here we brief a few

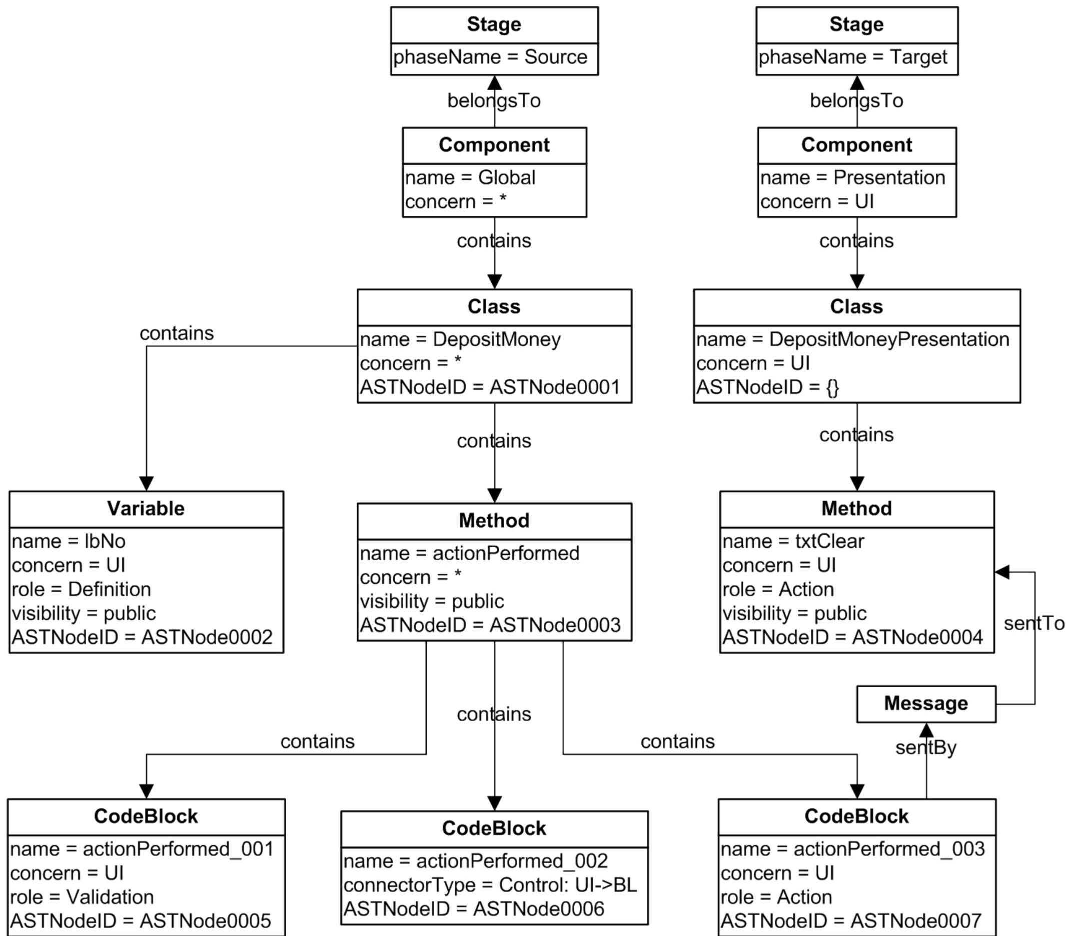

Figure 17: Graph representing a subset of the BankSystem sample application after the execution of rule Move Method UI

example works to show how this issue is dealt within different contexts.

The Dagstuhl Middle Model (DMM) [17] was developed to solve interoperability issues of reverse engineering tools. Like our approach, 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 [21] uses design pattern [12] 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++ [10]. 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, 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 [22].

The ARTISAn framework, described by Jakobac, Egyed and Medvidovic in [14], like our approach, categorizes source code. The code categories used are: ”processing”, ”data” and ”communication”. The approach differs from ours in several aspects. Firstly, the goal of the framework is program understanding and not the creation of a representation that is aimed to be used as in-

put for the transformation part of a reengineering methodology. Another important difference is that in ARTISAn the categorization process (called "labeling") is based in clues that result in the categorization of classes only. In our approach we need, and support, the method and code block granularity levels.

5 Conclusions and Future work

Most of the ongoing research in the context of automated software transformation as well as existing industrial tools mainly focus on textual and structural transformation techniques that intend to solve very specific problems within well defined domains (e.g. program restructuring, program renovation, language-platform migration). Our experience indicates that such techniques fall short to address as a systematic way the complexity of the architecture-based transformation problem. In practice, when such a problem arises the previous approaches are typically put together in a trial and error way the success of which often depends on the experience of the reengineering team and on the specific problem at hand. On the other hand, there exist techniques and tools that work well at an architectural level, but with the main goal of documenting and visualizing the architecture of applications rather than supporting increased levels of automation in achieving architecture-based transformations. Although such tools can provide a very good starting point and facilitate the subsequent effort, in industry projects a reengineering approach that starts with re-documenting architectures is often too limited given the time and budget constraints. In this work we have presented a systematic approach in order to explicitly address this issue. We take stock of our experience in dealing with such problems and have presented a method that attempts to bridge in a pragmatic way the gap outlined above. By pragmatic we mean (a) the ability to work well in real life legacy applications of large size, heterogeneity and complexity (b) be "implementable" in industry tools (c) be extendable in order to support the industry trend for reengineering legacy systems to a SOA paradigm. This paper has reported in detail the intended approach: the notations used to represent the source code, the code categorization and how to use them to change the source architecture allowing more adequate horizontal layering, and the graph transformation techniques to obtain the target architecture. The categorization plays a very important role in the methodology because it allows the identification of code fragments that can be used in the transformation rules specification. However, this research is still in its early stages and a lot of work is still to be done in order to achieve the final goal. One of the next steps to be performed is the automation of the categorization process.

Finally, as previously mentioned, one of the main interests of this approach is being able to respond to SOA needs and for this reason, an orthogonal (Vertical) process related with the functional logic of the system must be addressed. This is one of the main goals of Leg2Net and SENSORIA, and a great commitment to achieve it will be pursued in the near future.

Acknowledgments

R. Correia and C. Matos are Marie-Curie Fellows seconded to the University of Leicester as part of the Transfer of Knowledge, Industry Academia Partnership Leg2Net (MTK1-CT-2004-003169). This work has also been supported by the IST-FET IP SENSORIA (IST-2005-16004).

We would like to thank José Luiz Fiadeiro and Cristóvão Oliveira (University of Leicester) for their feedback and suggestions.

We would also like to thank Miguel Antunes and João Gouveia(ATX Software) for presenting real-world case studies and insights on industry reengineering projects.

References

[1] Eclipse. http://www.eclipse.org/.

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

[3] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the Twenty Sixth International Conference on Software Engineering*, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[4] M. G. J. V. D. Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, July 2002.

[5] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Software engineering by source transformation-experience with TXL. *Proceedings IEEE First International Workshop on Source Code Analysis and Manipulation (SCAM'01)*, pages 168–178, 2001.

[6] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, 2002.

[7] Eclipse. Eclipse modeling framework. http://www.eclipse.org/emf/.

[8] Eclipse. JDT - AST. http://help.eclipse.org/help30/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/AST.html.

[9] Eclipse. Visual editor project. http://www.eclipse.org/vep.

[10] R. Ferenc and Árpád Beszédes. Data exchange with the columbus schema for c++. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 59–66, Washington, DC, USA, 2002. IEEE Computer Society.

[11] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

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

[14] V. Jakobac, A. Egyed, and N. Medvidovic. Improving system understanding via interactive, tailorable, source code analysis. In *Fundamental Approaches to Software Engineering (FASE'05)*, pages 253–268. Springer Berlin / Heidelberg, 2005.

[15] R. Kazman, S. G. Woods, and S. J. Carrière. Requirements for integrating software architecture and reengineering models: CORUM II. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98)*, pages 154–163, Washington, DC, USA, 1998. IEEE Computer Society.

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

[17] T. C. Lethbridge, E. Plödereder, S. Tichelaar, C. Riva, P. Linos, and S. Marchenko. The Dagstuhl Middle Model (DMM). http://www.ece.queensu.ca/hpages/courses /elec875/pdf/DMMDescriptionV0006.pdf.

[18] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[19] T. Mens, N. V. Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[20] T. Mens, G. Taentzer, and O. Runge. Analyzing refactoring dependencies using graph transformation. *Software Systems Modeling (SoSyM), to appear*, 2006.

[21] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proceedings of the Twenty Fourth International Conference on Software Engineering (ICSE'02), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[22] G. Ramalingam, R. Komondoor, J. Field, and S. Sinha. Semantics-based reverse engineering of object-oriented data models. In *Proceeding of the Twenty Eighth International Conference on Software Engineering (ICSE'06)*, pages 192–201, New York, NY, USA, 2006. ACM Press.

[23] SENSORIA. Software engineering for service-oriented overlay computers. http://sensoria.fast.de/.

[24] H. M. Sneed. Object-oriented COBOL recycling. In *Proceedings of the Third Working Conference on Reverse Engineering (WCRE '96)*, pages 169–178, Washington, DC, USA, 1996. IEEE Computer Society.

[25] ATX Software. Forms2Net - from Oracle Forms to Microsoft .NET. http://forms2net.atxsoftware.com.