Transformation of Legacy Software into Client/Server Applications through Pattern-based Rearchitecturing

Sascha Hunold*, Matthias Korch*, Björn Krellner‡, Thomas Rauber*, Thomas Reichel‡, Gudula Rünger‡

* *Department of Mathematics and Physics, University of Bayreuth, Germany*
{hunold,matthias.korch,rauber}@uni-bayreuth.de

‡ *Department of Computer Science, Chemnitz University of Technology, Germany*
{bjk,thomr,ruenger}@cs.tu-chemnitz.de

Abstract

In this article, we address the problem of modularizing legacy applications with monolithic structure, primarily focusing on business software written in an object-oriented programming language. We introduce the TransFormr toolkit that guides the developer trough the entire incremental transformation process. It is the goal of the transformation to separate the original software into several independent replaceable components to support the migration of legacy code to new hardware or to integrate legacy components into modern enterprise applications. We show the effectiveness of our approach by demonstrating a pattern-based transformation of classes in a case study.

1. Introduction

The legacy software problem is becoming more relevant than ever since companies are bound to short software release cycles. Code which has been working for years needs to be changed in order to support new environments and to enter new markets. Rewriting huge applications is cost-intensive and error-prone. The main problem however is the monolithic structure of many applications which makes it difficult to extend functionality and to increase portability. Since the software has grown for years, the challenge of entering new markets (like web-enabling the software) or keeping a short release cycle is solved by adding more manpower on the project which in turn increases costs. Another problem is the so-called legacy code of applications in use. This code might be undocumented and the original authors have left the company years ago. To solve these problems, new paradigms (OO, SOA) and new refactoring tools (Eclipse) have been invented to help restructuring the software. However, most solutions are focusing only on a single aspect of the transformation process but the problem of rearchitecturing a legacy application is manifold.

An important class of applications are business software systems which implement custom business processes. Business software systems have many common legacy problems, specific to this type of application, such as tightly-coupled source code and therefore no strict separation of concerns, and no coding conventions. Many of these software systems have been written using an integrated development environment (IDE) which is bound to a particular operating system and programming language. The IDEs used in the 1990s (like Delphi) did not enforce a clean separation of business logic and representation code, leading to the aforementioned tightly coupled source code. Nowadays, these business software systems used in production are far too huge to rewrite the software from scratch. Since different types of software require different strategies, we have proposed an incremental transformation process [13] which addresses the problem of rearchitecturing a monolithic business software. This process defines a chain of transformation steps to transform a monolithic software system into a modular software system which can be executed in a distributed environment. In this article, we outline the incremental transformation process and give an overview of the development status of the transformation toolkit. In the first step of the transformation process, the software is categorized and a language-independent representation (model) of the application is generated by parsing and annotating the original source code. In a second step, this meta information is used to refactor the legacy code, e.g., by removing dependencies between classes or by replacing entire classes. In the code generation step, selected functionality of the legacy software is re-

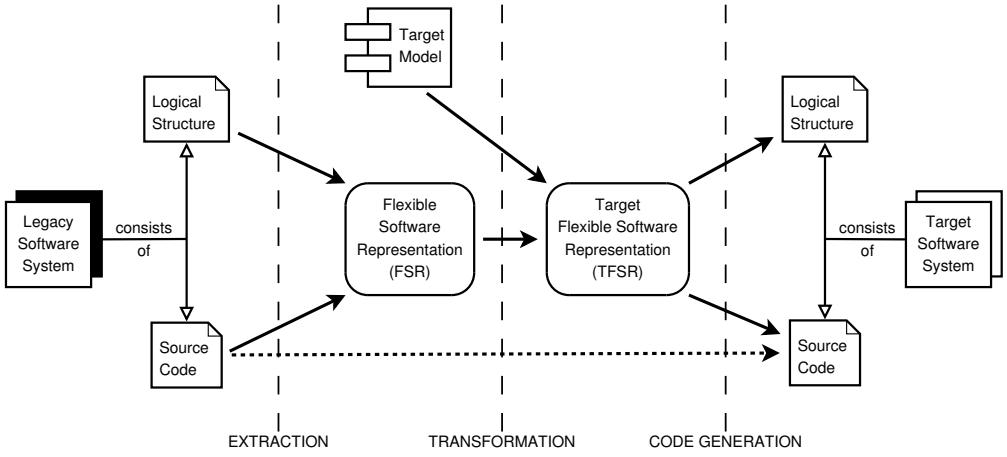


Figure 1. Transformation chain showing the transformation from legacy code to target code.

located to a remote server by extracting the interface of individual classes and by automatically generating RPC code to obtain a transparent software extension.

The rest of the paper is organized as follows: Section 2 outlines our approach for rearchitecturing a business software and Section 3 introduces the toolkit which supports the incremental transformation of legacy software. Section 4 presents a case study in which classes of a legacy system are converted into remotely available services. Section 5 gives an overview of related transformation frameworks and Section 6 concludes the article.

2. Incremental transformation process

Business software systems support humans in the management of a business by providing suitable means for storing, editing and processing data and information related to the business and by governing the execution of the business processes. We consider the case of *monolithic* business software systems which consist of a single application program and a database system. The application program fulfills a variety of requirements. It provides a graphical or text-based user interface, formats and displays data, accepts and validates the user's inputs, implements the logic of the business processes, and interacts with the database.

We consider the generation of distributed business software starting from a monolithic legacy software system. The transformation is performed in an incremental process organized in several steps. In this section, we outline the transformation process paying regard to the challenges evoked by the monolithic architecture of the legacy software system, possible solutions, approaches, and technologies involved.

The incremental transformation process is divided into three basic steps: extraction, transformation, and generation, as shown in Figure 1.

Extraction In the extraction step, the legacy code is converted into a software model. We use an intermediate model-oriented software representation during the transformation process which is called *Flexible Software Representation* (FSR). The FSR uses a metalanguage to represent the legacy software which is independent of the actual programming language. To generate the FSR, the source code is analyzed and categorized using predefined categories such as UI related code or database related code. An important challenge in this phase is the extraction of business processes and to make the workflow explicit. An explicit workflow allows the formalization of the business processes of a legacy system using WfMC[1] standards like XPDL. The legacy software can then be extended by integrating workflow engines such as Enhydra Shark, WfMOpen, and jBPM in order to process the standardized workflows.

Transformation It is a major goal of our research to perform most of the required steps during the transformation phase without modifying the code by hand. Thus, the transformation step is mostly performed at higher abstraction levels. All transformation operations are directly applied to the FSR. The FSR contains all the necessary information to provide the developer with several different views of the current transition state of the software model, like dependencies between classes and variables or more abstract UML diagrams. The developer uses this information to perform a stepwise

[1] Workflow Management Coalition

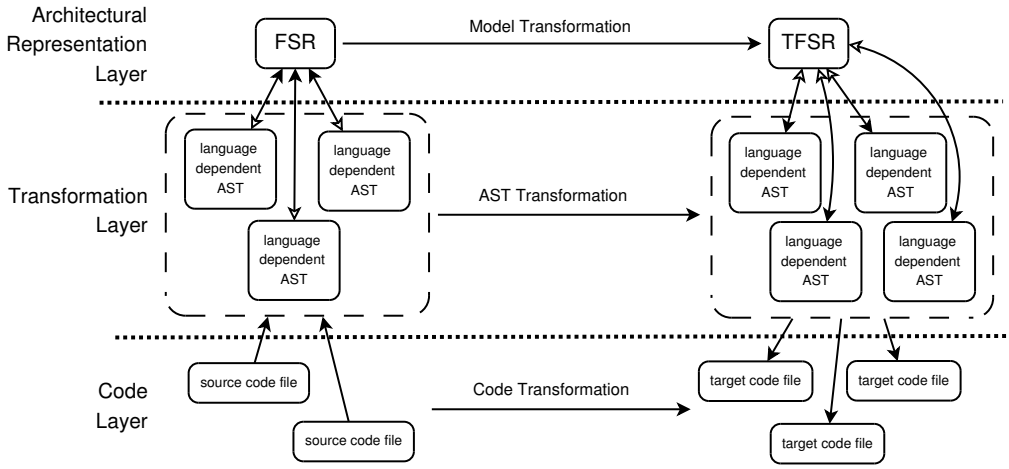


Figure 2. Abstraction layers for each step of the transformation process: model extraction (upwards), transformations (sidewards), and code generation step (downwards).

transition of the software, i.e., making modifications based on the FSR. These modifications include

- common refactorings (rename, move, extract),
- filter operations to generate different versions of the legacy software (number of features), and
- extract and externalize operations which extract selected functions from the code and make them transparently accessible as remote services.

Each modification can be applied to the FSR, leading to a chain of FSRs. The final FSR is used to generate the actual source code for a target system and is called *Target FSR* (TFSR).

Generation In this last step, the code for the target system is generated from the TFSR. The code generation phase requires more custom code for a specific legacy software than the other steps. Depending on the target hardware and software architecture, the TFSR is transformed into compilable source code. It is important to indicate that most of the required code cannot be provided in advance, e.g., if the developer decides to migrate a software system from standard Java and Unix to Enterprise Java and Windows, glue code has to be written to embed the legacy classes into this environment. Therefore, it is crucial for a successful transformation approach to generate as much code as possible, e.g., using code templates. Before the generation process can be executed, the developer must select the configuration of the target system such as the software architecture, the operating system, and the libraries.

3. Transformation toolkit: TransFormr

To support the reengineering process we propose a transformation toolkit for an interactive reorganization of software. The toolkit guides the developer through all phases of the transformation process. The transformation process works on three different abstraction layers as illustrated in Figure 2. The source code can be found in the bottom layer. The intermediate layer contains language-dependent abstract syntax trees (ASTs), and the architectural model of the legacy code is located at the top of the abstraction model. The language-dependent ASTs are temporary entities of the transformation process. In the extraction phase, the ASTs provide information about the structure of the source code used to categorize the legacy software. In the transformation and code generation phase, ASTs help to translate architectural changes to modifications of the source code, i.e., model-driven architectural transformations are compiled to AST transformation rules.

The rest of this section introduces the working method of the transformation toolkit TransFormr for each transformation step.

3.1. Extraction

In the initial step, the logical structure of the legacy software is extracted from the source code. The logical structure defines the architectural model described by the FSR. The initial input consists of many files in different subdirectories. The TransFormr toolkit uses a language transformation processor (LTP) to extract the FSR from the source code. The LTP builds language-dependent ASTs using the grammar of the program-

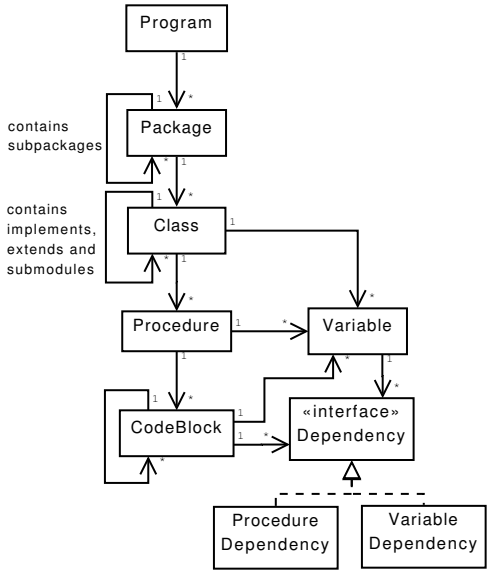


Figure 3. Structure of the FSR.

ming language of the legacy code. Currently, the toolkit supports Java and Delphi as input languages. After building the ASTs, the FSR is generated by traversing each AST and applying predefined categorization rules. In the extraction phase, TransFormr tries to exploit all available information to obtain the software model, e.g., the directory structure can be used to obtain a first abstract view. Currently, the LTP is implemented using the source transformation program TXL [4]. TXL is a functional programming language created to support code analysis and source transformation tasks. From each source file an intermediate representation file is generated. The FSR is assembled from these intermediate files. Since TXL can only parse one file at a time, resolving the type of variables requires multiple parser passes. In a first pass, all types of variables are replaced by their fully qualified name and in a second pass, TXL constructs the FSR.

The structure of the FSR is outlined in Figure 3 and contains the following entities: The *program* represents the complete software system, its parts (packages), classes, and the relationships between them. It contains packages and is the entry point for each transformation operation. A *package* corresponds to a part of the software system and can contain further packages or classes. A *class* contains procedures and variables. The class description additionally includes information about the relations between classes, e.g., relations defined by generalization, realization, or aggregation. Some programming languages also allow nested classes, for example, anonymous classes and private inner classes of Java. These nested classes can also be captured in the FSR. Classes are only included in the FSR if their source code is part of the legacy software. System libraries or external libraries are only referenced by name. A *procedure* corresponds to a method of a class or a procedure in a procedural language. It consists of a name, a list of variables, a modifier, and a return value. *Variables* occur as parameters to procedures or to class constructors, as class member variables, as local variables inside procedures, or as global variables. The *codeblock* entity represents basic blocks, i.e., a group of statements delimited by curly brackets or begin/end statements. It has been incorporated into TransFormr to take account of variables with the same name but different scope within a procedure. The *dependency* entity denotes either a dependency on a variable or on a function call. *Variable dependencies* are abstractions of (read or write) accesses to local or other public member variables. A *procedure dependency* is added to the FSR for each procedure call statement.

As seen in Figure 3, the FSR does not capture single statements, conditions, or loops. Therefore a migration between programming languages within code blocks is not possible with the current model. If such migration is required, the FSR has to be extended by a code model which covers every single statement, e.g., using JDT of Eclipse.

3.2. Transformation

TransFormr supports a number of basic and composite transformation operations. Examples of basic transformation operations are *create*, *rename*, and *move*. The create operation is used to create new entities, e.g., if the software expert decides to split a legacy module into new submodules. The rename operation can be applied to each FSR component (package, class, procedure, or variable) and the move operation moves entities to another position in the model, e.g., it can be used to move procedures from one class to another.

The basic transformations differ in terms of complexity, and the complexity of applying an operation also depends on the hierarchy level of the entity which is the target of the transformation. Let us consider the complexity of the rename and move operation. Renaming a variable or an entire class is fairly simple compared to moving this entity. The rename operation only ensures that the new name follows the coding conventions, avoids a name clash, and substitutes all occurrences of the name in the source tree. Moving an entity to another scope implies a deeper analysis of the source code. The move operation has to ensure correctness of the scopes which are target and source of the move operation as well as all entities which are dependent on

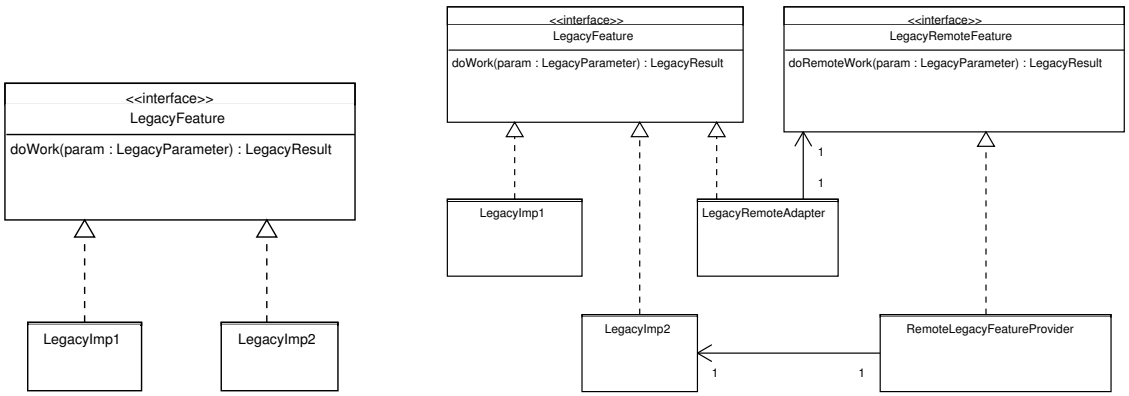


Figure 4. UML transformation pattern: Remote Adapter Pattern.

them. As mentioned before, the complexity also increases by the scope level (granularity) of the entity to be transformed, e.g., it is harder to move a (member) method between classes than to move a class between packages.

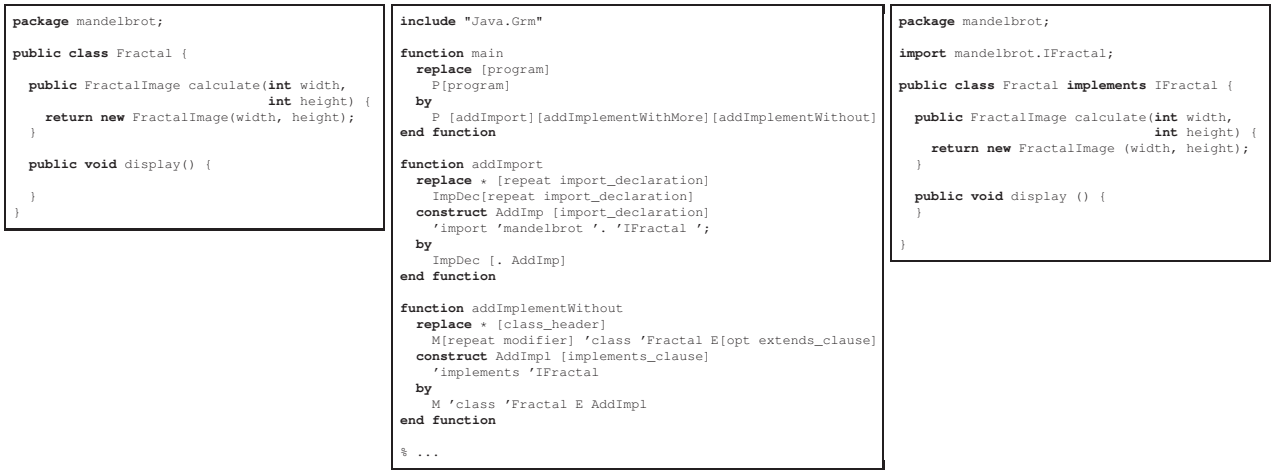
These transformation operations are based on reengineering patterns which are introduced in [6]. More complex operations can be assembled using the basic operations, e.g., functions to apply a certain design pattern. An example of such a composite operation is the UML pattern substitution. The TransFormr toolkit enables the developer to inspect the logical structure of the system by generating UML diagrams. When applying a UML pattern substitution, parts (classes, dependencies, function names) of the current FSR are replaced by a defined target model if the FSR matches the specified UML pattern. Such a substitution within a software model can be used to simplify the transformation process (refactoring) or can help to extend the functionality of the software. In case that the FSR of the legacy software or any of its transition states does not provide the required source pattern, the developer must first utilize the refactoring tools of TransFormr to match the conditions. Figure 4 shows the *Remote Adapter Pattern* as an example of a UML substitution pattern. This pattern is used to substitute classes which are executed on the client side with classes that implement the same functionality but are executed on the server side. A more detailed discussion of the *Remote Adapter Pattern* is given in Section 4.

Although the transformations of the actual source code are only carried out when the target code is generated, the transformation rules have to be created during the transformation step. In order to apply these basic transformations, TransFormr generates TXL rule files from a set of pre-defined templates. A transformation operation for a specific programming language is associated with a number of TXL rule templates. Figure 5 presents such a rule-based transformation in which a Java class is translated into a new Java class to implement a defined interface. The TXL rule file (in the center) is generated from a template to meet the specific needs of this transformation, i.e., naming the source class and the interface.

3.3. Code generation

In the code generation phase the transformation operations are compiled into code modification rules in order to produce platform-dependent executable code. Similar to the code extraction phase, the source code, the grammar of the source language, and code modification rules are used by the LTP to generate the target code. TXL is also used as LTP in the code generation phase. The code modification rules are generated from information stored in the TFSR which contains the necessary references to the source code of the legacy software. In other words, the TFSR describes the target software architecture and stores all the necessary information to reuse software parts. TransFormr uses language-dependent rule patterns to generate transformation rules.

As mentioned above, TransFormr supports complex transformation operations such as the substitution of UML patterns. Substituting parts of the code is especially effective if most of the target code can be generated automatically. Therefore, the front-end of TransFormr also contains a template engine to generate source code from a defined set of templates. Code templates are provided for each transformation pattern, e.g., for the *Remote Adapter Pattern*. In order to generate the classes from these templates automatically, the developer specifies the required information which is specific to the pattern. In case of the remote adapter, the



```
package mandelbrot;

public class Fractal {

  public FractalImage calculate(int width,
                                int height) {
    return new FractalImage(width, height);
  }

  public void display() {

  }
}
```

```
include "Java.Grm"

function main
  replace [program]
    P[program]
  by
    P [addImport][addImplementWithMore][addImplementWithout]
end function

function addImport
  replace * [repeat import_declaration]
    ImpDec[repeat import_declaration]
  construct AddImp [import_declaration]
    'import 'mandelbrot '. 'IFractal ';
  by
    ImpDec [. AddImp]
end function

function addImplementWithout
  replace * [class_header]
    M[repeat modifier] 'class 'Fractal E[opt extends_clause]
  construct AddImpl [implements_clause]
    'implements 'IFractal
  by
    M 'class 'Fractal E AddImpl
end function

% ...
```

```
package mandelbrot;

import mandelbrot.IFractal;

public class Fractal implements IFractal {

  public FractalImage calculate(int width,
                                int height) {
    return new FractalImage (width, height);
  }

  public void display () {
  }

}
```

Figure 5. Example of a TXL-based transformation of a Java class. From left to right: original source code, TXL transformation rules (fragment), target code.

developer names the functions to be relocated and the names of the adapter classes.

4. Pattern-based transformation of legacy classes into remote services

In this section, we show how TransFormr can be used to relocate functions of legacy classes to remote servers. The relocation or externalization of functions of a software system is driven by different needs. Often legacy systems simply lack functionality which is provided by some external services (web service, database server) and which should be integrated. Other reasons for providing external services are increasing data safety (external database servers), enhancing data integrity (transaction protocol of a database, centralized data checking), or improving performance (using dedicated servers for computationally intensive tasks).

In this article, we focus on improving the performance of a legacy software system by integrating remote services. Thus, we present a case study in which a legacy class is extracted and relocated to a remote server by transparently binding the new code via RMI calls. This transformation can be done using the UML pattern substitution and the *Remote Adapter Pattern* which is depicted in Figure 4. The source pattern consists of a legacy class called LegacyFeature which is realized by two implementations, LegacyImp1 and LegacyImp2. The transformation pattern assumes that LegacyImp2 is the implementation to be made available remotely.

In most of the cases, the legacy code does not provide a clean separation into LegacyFeature interface and its implementations. In these cases, the developer must extract the interface from classes and all subclasses. The 'extract interface' refactoring is implemented for Java by most IDEs such as Eclipse and Netbeans, and thus, TransFormr can utilize this API in case of Java. For Delphi and other object oriented languages, TransFormr has to be extended to support the 'extract interface' refactoring. Figure 6 shows an example for extracting an interface from a class where the developer wants to expose the functions calculate() and display() via the IFractal interface. The modification of the source code to implement the new interface is done with TXL as shown in Figure 5.

When all conditions to apply the transformation pattern are met, TransFormr replaces the old legacy class LegacyImp2 with the LegacyRemoteAdapter that encapsulates the remote communication and hides the remote features from the legacy code. The LegacyRemoteAdapter contains the client-side code necessary for integrating the remote service. The remote service itself is realized by the RemoteLegacyFeatureProvider which can be implemented in different ways, e.g., as RMI server or as Java bean. All additional classes are generated using the template engine of TransFormr. Figure 7 shows a sample template for an adapter class using Java and RMI. The variables marked by '<< >>' are substituted by TransFormr. Most of these updates of template variables are replacements of identifiers. However, TransFormr also generates complete methods, e.g., `<<AdapterFunction>>` is replaced by the call to `doWork` which transparently redirects to `doRemoteWork`.

After the adapter code and the server code have been generated, TransFormr also replaces the code ref-

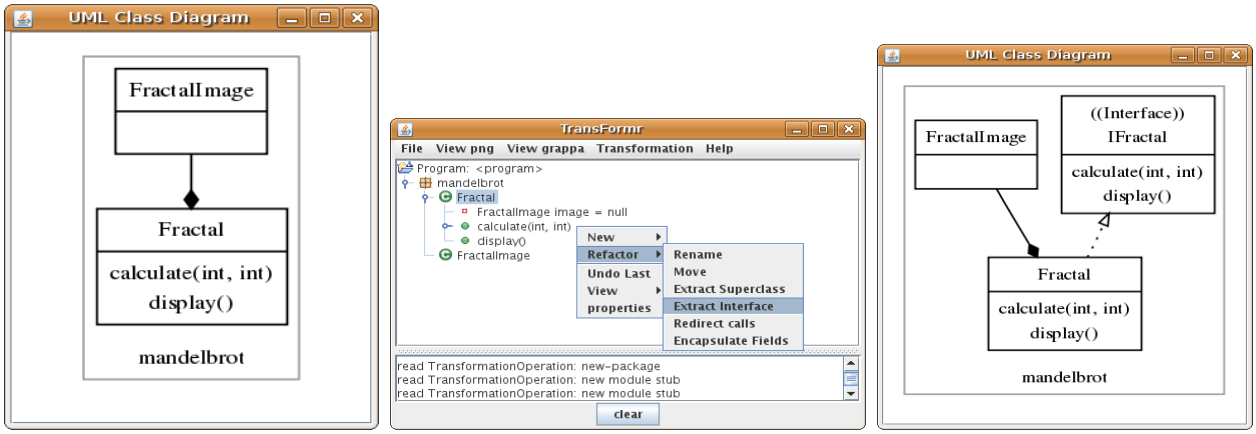


Figure 6. Extracting an interface in order to apply an UML pattern (Remote Adapter).

erences to class LegacyImp2 with the adapter class. If now the local client of the transformed legacy software calls method `doWork` on an instance of LegacyRemoteAdapter, the call is redirected to the server which performs the task and sends the result back to the client.

5. Related work

The problem of legacy software has always been relevant to software companies as well as to researchers. Reengineering or rearchitecturing of legacy code is a challenging task and may often fail [2] if not planned in detail. Several approaches to do reengineering have been proposed which aim at different abstraction levels and different software layers. Tools have been developed to help extracting or understanding the software architecture by visualizing its structure [11, 14]. The Object Management Group is working on interoperability standards for modernization tools (Architecture-Driven Modernization) [1]. A two-step transformation process to change the software architecture has been presented in which the old architecture is extracted using special algebra which can be validated, visualized, and later transformed into modification rules [10]. Kazman et al. proposed the horseshoe model [9] as generic framework for program transformation. The framework consists of three processes: (1) architecture recovery, (2) architecture transformation, (3) code generation. An example for code-based reengineering as well as rearchitecturing using concepts of the horseshoe model is presented in [3]. A similar work categorizes and transforms a legacy software using TXL to convert the user interface [5]. A classification of schemas is given in [8] aiming to enable the interoperability of several integration technologies like CORUM II [9]. Reverse engineering plays an important role in understanding the system architecture, and choosing an architecture is a crucial task [12]. Successful software transformation approaches often follow the same pattern of restructuring the application. An introduction to common reengineering patterns which occur in transformation processes is given in [6]. The problem of validating the architectural design and its implementation can be handled by systems like DiscoTect [15] which monitors events and creates an Acme [7] description of the discovered architecture.

6. Conclusions

In this article, we have presented a framework for an iterative transformation process of business software systems. We have introduced the toolkit TransFormr which supports the developer in recovering the system's architecture, creating an architectural model, making modifications to the model, and generating code to create an enhanced and modular version of the old software. The work presented in this article is targeted but not limited to general business software, i.e., our approach could also help to transform other legacy applications. We have also shown how the toolkit can be used to externalize functionality of legacy software by using UML patterns to replace parts of the legacy code. Future work aims at new techniques to modify the architectural model by evaluating new refactorings and by adding new pattern-based substitution rules.

7. Acknowledgment

The transformation approach described herein as well as the associated toolkit are part of the results of the joint research project called TransBS funded by the German Federal Ministry of Education and Research.

```
package <<TargetPackage>>;

import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class <<AdapterName>> implements <<LegacyInterface>> {

  private <<LegacyRemoteInterface>> <<RemoteHandleVar>>;

  public <<AdapterName>>() throws LegacyAdapterException {
    try {
      RemoteProperties props = new RemoteProperties();
      Registry registry = LocateRegistry.getRegistry(props.getServerName());
      <<RemoteHandleVar>> = (<<LegacyRemoteInterface>>) registry.lookup("<<RmiObjectId>>");
    } catch(RemoteException e) {
      throw new LegacyAdapterException(e);
    } catch(NotBoundException e) {
      throw new LegacyAdapterException(e);
    } catch(IOException e) {
      throw new LegacyAdapterException(e);
    }
  }

  /* adapter is generated automatically */
  <<AdapterFunction>>
}
```

Figure 7. Template of a legacy feature adapter in Java.

References

[1] ADM task force. Architecture-Driven Modernization: Transforming the Enterprise. http://adm.omg.org, 2007.

[2] J. Bergey, D. Smith, S. Tilley, N. Weiderman, and S. Woods. Why Reengineering Projects Fail. Technical report, Carnegie Mellon Software Engineering Institute, Pittsburgh, PA 15213-3890, 1999.

[3] S. J. Carrière, S. G. Woods, and R. Kazman. Software Architecture Transformation. In *Proceedings of WCRE 99, (Atlanta, GA)*, Oct. 1999.

[4] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[5] 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, Department of Computer Science, University of Leicester, U.K., 2006.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[7] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[8] D. Jin, J. R. Cordy, and T. R. Dean. Where's the Schema? A Taxonomy of Patterns for Software Exchange. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, page 65, Washington, DC, USA, 2002. IEEE Computer Society.

[9] R. Kazman, S. G. Woods, and S. J. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Proceedings of WCRE 98, (Honolulu, HI)*, Oct. 1998.

[10] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A Two-Phase Process for Software Architecture Improvement. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, volume 00, page 371. IEEE Computer Society, 1999.

[11] W. Löwe, M. Ericsson, J. Lundberg, T. Panas, and N. Pettersson. VizzAnalyzer - A Software Comprehension Framework. In *Software Engineering Research and Practice - SERPS 03*, October 2003.

[12] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse Engineering: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA, 2000. ACM Press.

[13] T. Rauber and G. Rünger. Transformation of Legacy Business Software into Client-Server Acrchitectures. In *Proc. of the 9th International Conference on Enterprise Information Systems*, Funchal, Madeira - Portugal, 2007.

[14] J. Rilling and S. P. Mudur. 3D visualization techniques to support slicing-based program comprehension. *Computers & Graphics*, 29(3):311–329, 2005.

[15] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *IEEE Trans. Softw. Eng.*, 32(7):454–466, 2006.