7

Architectural Transformations: From Legacy to Three-Tier and Services

Reiko Heckel[1], Rui Correia[1,2], Carlos Matos[1,2], Mohammad El-Ramly[3], Georgios Koutsoukos[1], and Luís Andrade[2]

[1] Department of Computer Science, University of Leicester, United Kingdom
[2] ATX Software, Lisboa, Portugal
[3] Computer Science Department, Cairo University, Egypt

Summary. With frequent advances in technology, the need to evolve software arises. Given that in most cases it is not desirable to develop everything from scratch, existing software systems end up being reengineered. New software architectures and paradigms are responsible for major changes in the way software is built.

The importance of Service Oriented Architectures (SOAs) has been widely growing over the last years. These present difficult challenges to the reengineering of legacy applications. In this chapter, we present a new methodology to address these challenges. Additionally, we discuss issues of the implementation of the approach based on existing program and model transformation tools and report on an example, the migration of an application from two-tier to three-tier architecture.

7.1 Introduction

As business and technology evolve and software becomes more complex, researchers and tool vendors in reengineering are constantly challenged to come up with new techniques, methods, and solutions to effectively support the transition of legacy systems to modern architectural and technological paradigms. Such pressure has been witnessed repeatedly over the past decades. Examples include the adoption of object-oriented programming languages [380, 149] and more recently the advent of Web technologies [463] and in particular Service-Oriented Architectures (SOAs).

The adoption of SOAs, as well as their enabling technology of Web Services [7], has been steadily growing over the last years. According to Gartner [194], a leading technology market research and analysis firm, "mainstream status for SOA is not far off" and "by 2008, SOA will be a prevailing software engineering practice" [355]. However, practice indicates that Service-Oriented Architecture initiatives rarely start from scratch. Gartner projects that "through 2008, at least 65 percent of custom-developed services for new SOA projects will be implemented via wrapping or re-engineering of established applications (0.8 probability)". In other words, most SOA projects are being implemented on top of existing legacy systems. That being the context, the goal of this work is to present a methodology for reengineering legacy systems towards new architectural styles in general and SOA in particular.

We will argue that, starting from a (monolithic) legacy application, such a transition involves several steps of decomposition, along both technological and functional dimensions. The technological decomposition will lead, for example, to a 3-tiered architecture, separating application logic, data, and user interface (UI). The functional decomposition separates components providing different functions which, when removing their UI tiers, represent candidate services.

The technical contribution of this chapter concentrates on the iterated decomposition, regarding each cycle as an instance of the reengineering Horseshoe Model [271]. 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 to the architectural level. We support it by providing automation through graph-based architectural transformation. This allows us to

- abstract (in large parts of the process) from the specific languages involved, as long as they are based on similar underlying concepts
- describe transformations in a more intuitive and "semantic" way (compared to code level transformations), making them easier to adapt to different architectural styles and technological paradigms

With this in mind the remainder of this chapter is organised as follows: Section 7.2 discusses the impact of service-oriented computing on legacy systems, as well as the issues for reengineering. Section 7.3 presents our methodology for architectural transformation including a formalisation based on typed graph transformation system. The implementation of our approach and an example are discussed in Section 7.4. We review related work in Section 7.5 and discuss conclusions and further work in Section 7.6.

7.2 From Legacy Systems to Three-Tier Applications and Services

The authors' experience with customers from the finance, telecommunications, and public administration sectors as well as IT partners indicates that adoption of SOA in industry is inevitable and that such adoption is typically gradual, evolving through various stages in which different organisational and technical goals and challenges are addressed. A typical first stage, the transition from legacy to web-based systems, consists in the technological separation of GUI from logic and database code, and subsequent replacement of the GUI code by HTML forms. Even if the details are highly dependent on the languages and platforms involved, the perception that organisations have of the overall aim, the transition towards SOA, is largely congruent, and in line with what has also been described by several authors [349, 393, 162] and major technology providers [407, 472, 543, 483].

In Table 7.1 we outline six of the basic SOA principles that constitute important properties of SOA from an industry perspective. It should be noted that our goal is

Table 7.1. Industry View of SOA

SOA Property	Definition
Well-defined interfaces	A well specified description of the service that permits consumers to invoke it in a standard way
Loose coupling	Service consumer is independent of the implementation specifics of service provider
Logical and physical separation of business logic from presentation logic	Service functionality is independent of user interface aspects
Highly reusable services	Services are designed in such a way that they are consumable by multiple applications
Coarse-grained granularity	Services are business-centric, i.e., reflect a meaningful business service not implementation internals
Multi-party & business process orientation	Service orientation involves more than one party (at least one provider and one consumer), each with varying roles, and must provide the capacity to support seamless end-to-end business processes, that may span long periods of time, between such parties

not to provide a comprehensive analysis of how industry views SOA, but, instead, to provide a basis that will help us to explain the impact of service-orientation to legacy systems.

The first two properties in Table 7.1 (Well-defined interfaces, Loose coupling) are typically, at least at the technology level, provided by the underlying SOA implementation infrastructures such as Web Services. The last four properties however, have considerable impact on legacy systems and their reengineering. Such impact is analysed in the next three subsections. The first addresses the "Logical and physical separation of business logic from presentation logic" property, the second analyses the "Highly reusable services" property and the last addresses both "Coarse-grained granularity" and "Multi-party & business process orientation" properties.

7.2.1 Technological Decomposition

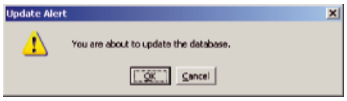
It is a common practice in legacy applications to mix together, in a kind of "architectural spaghetti", code that is concerned with database access, business logic, interaction with the user, presentation aspects, presentation flow, validations and exception handling, among others. For example, consider interactive COBOL programs: Typically these are state-machine programs that interleave the dialog with the user (menus, options, etc.) with the logic of the transactions triggered by their inputs. Similar coding practices are found in client-server applications like Oracle Forms, Java-Swing, VB applications, etc. The code listing in Figure 7.1 presents a simple Java example that partially illustrates this issue. In this code fragment, if data access and data processing code fails or no data is found, a message dialog appears to the user prompting for subsequent actions. The PL/SQL code of Figure 7.2 refers

```
public void Transaction () {
  try {
     //Data access-processing code
    fis = new FileInputStream ("Bank.dat");
    ... //fetch some data from file
  }
  catch (Exception ex) {
    total = rows;
     //Validations and respective UI actions
    if (total == 0) {
      JOptionPane.showMessageDialog (null, "Records File is Empty.\nEnter
        Records First to Display.", "BankSystem - EmptyFile",
        JOptionPane.PLAIN_MESSAGE);
      btnEnable ();
    }
    else {
      try {
         //Data access-processing code
        fis.close();
      }
      catch (Exception exp) {
        ...
      }
    }
  }
}
```

Fig. 7.1. A simple "spaghetti" code example in Java. (We use spaghetti here in the sense of tangling different concerns, not in the sense of having many goto statements)

```
PROCEDURE Confirm()

DECLARE

alert_button NUMBER;

BEGIN

alert_button := SHOW_ALERT('alert_name'); IF alert_button =
ALERT_BUTTON1 THEN
     program statements after clicking first button (OK button);
ELSE
     program statements after clicking second button (Cancel button);
END IF;

END;
```

(a) Alert dialog code mixed with business processing



You are about to update the database.

(b) Oracle Forms Alert dialog for confirmation

Fig. 7.2. PL/SQL example

to a similar scenario in Oracle Forms: The whole business processing code (in bold in Figure 7.2a), which may concern complex calculations and updates of database tables, is placed together with the code that manages the interaction with the user via a simple alert dialog (Figure 7.2b) prompting for confirmation for performing such a transaction. In such cases, since the business logic is tightly coupled with the presentation logic, it is impossible to derive services directly. Therefore, what is required is an appropriate decoupling of the code, such that "pure" business processes are isolated as candidate services or service constituents. This technological dimension of reengineering towards SOA amounts to an architectural transformation towards a multi-tiered architecture.

7.2.2 Reusable Services

For an SOA initiative to realise its full potential a significant number of implemented and deployed services should be actually invoked by more than one application. Service repositories facilitate such reuse, but only *a posteriori*, i.e., after the reusable services have been identified. From a reengineering perspective what is needed is support for the *a priori* identification of reusable services across multiple functional domains. Unfortunately, many legacy systems can be characterised as "silos", i.e., consisting of independent applications where lots of functionality is redundant or duplicated (cf. Chapter 2). Even worse, in many cases such redundancy and duplication also exists within the same application. Take, for instance, the example of financial systems, where the interest calculation functionality is very often implemented multiple times in different applications only to accommodate the needs of the various departments that those applications are designed to serve. But even within single applications such redundancy is a common practice, for instance between interactive and batch parts. The ability to identify such redundant functionality and its appropriate refactoring to reusable services is vital for the success of service-oriented computing initiatives.

7.2.3 Functional Decomposition

Most legacy applications were developed with different architectural paradigms in mind and typically consist of elements that are of a fine-grained nature, for instance components with operations that represent logical units of work, like reading individual items of data. OO class methods are an example of such fine-grained operations. The notion of service, however, is of a different, more coarse-grained nature. Services represent logical groupings of, possibly fine-grained, operations, work on top of larger data sets, and in general expose a greater range of functionality. In particular, services that are deployed and consumed over a network must exhibit such a property in order to limit the number of remote consumer-to-provider roundtrips and the corresponding processing cycles. In general, finding the right balance of service granularity is a challenging design task that is also related with the service reusability issue above. A good discussion on the granularity of services in systems that follow the SOA paradigm can be found in [543], from where Figure 7.3 has been

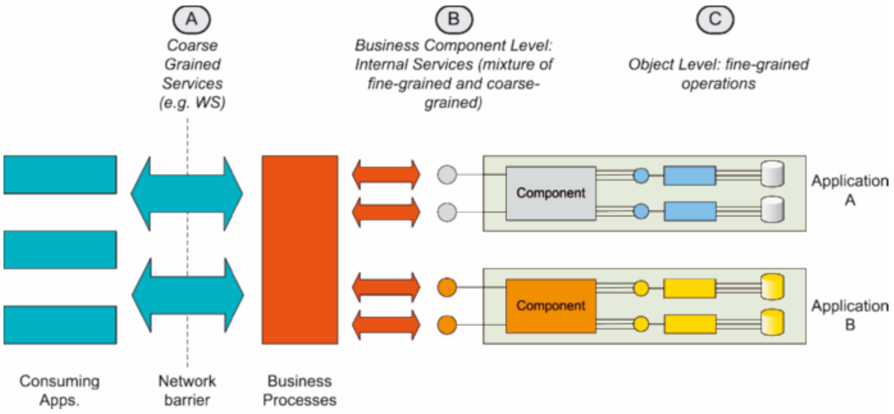

A
Coarse Grained Services (e.g. WS)
B
Business Component Level: Internal Services (mixture of fine-grained and coarse-grained)
C
Object Level: fine-grained operations
Component
Application A
Component
Application B
Consuming Apps.
Network barrier
Business Processes

Fig. 7.3. Service granularity across application tiers

adapted to show the various levels of service granularity. In general, it is clear that the granularity of services has major implications for what concerns legacy reengineering. As already mentioned at the beginning of the paragraph, most of the systems currently in use were not built with service orientation in mind. Hence, existing services at levels B and C of Figure 7.3 are not at the level of granularity required for SOA.

The granularity problem is also associated with the fact that service-orientation involves more than one party (at least one provider and one consumer), each with varying roles and, if designed properly, must provide the capacity to support seamless end-to-end business processes (spanning long periods of time) between such parties. This is a fundamental shift from previous architectural paradigms in which the business processes workflow and rules are typically defined by one party only and executed entirely on the IT system of this same party (e.g., a customer self-service system). Legacy systems are not prepared for such a shift in paradigm: For example, a legacy function that returns information from a single transaction was not intended to be called several times in succession in order to obtain the larger set of data that a service consumer may require. Even more recent systems, built on top of web services technologies that expose services at level A in Figure 7.3 suffer from poor granularity decisions and are unable to support the desired end-to-end multi-party business processes. Hence, software reengineering solutions with respect to service orientation are concerned with all 3 levels (A,B,C) of services depicted in Figure 7.3. In particular, we are convinced that methods and tools are needed that allow service designers to discover the allocation of domain functionalities into the code structure so that legacy logical units of work can be appropriately composed and reengineered in order to form services of desired granularity and of adequate support for multi-party business processes.

In the following sections we are going to concentrate on the technical aspect of the decomposition, rather than on questions of granularity and reusability. While the techniques described below are applicable to both technological and functional decomposition we will use an example of technological decomposition to illustrate them.

7.3 The Approach to Architectural Transformation

In this section we discuss methodological as well as formal aspects of the approach to architectural redesign. Methodologically we are following the Horseshoe Model, refining it to support automation and traceability. Formally our models are represented as graphs conforming to a metamodel with constraints while transformations are specified by graph transformation rules.

7.3.1 Methodology

Our methodology consists of the three steps of *reverse engineering, redesign*, and *forward engineering*, preceded by a preparatory step of *code annotation*. The separation between code annotation and reverse engineering is made in order to distinguish the three fully automated steps of the methodology from the first one, which involves input from the developer, making it semi-automatic. The steps are illustrated in Figure 7.4.

1. Code Annotation

The source code is annotated by *code categories*, distinguishing its constituents (packages, classes, methods, or fragments thereof) with respect to their foreseen association to architectural elements of the target system, e.g., as GUI, Application Logic, or Data.

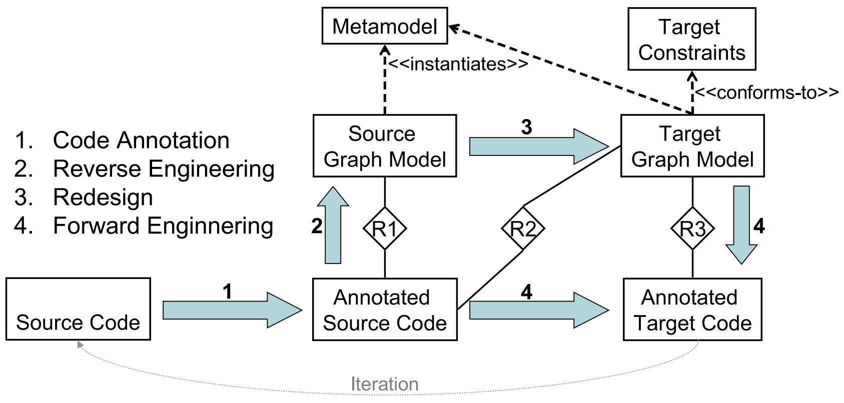


Fig. 7.4. Methodology for transformation-based reengineering

The annotation is based on input by the developer, propagated through the code by *categorisation rules* defined at the level of abstract syntax trees, and taking into account information obtained through control and data flow analysis. The results may have to be revised and the propagation repeated in several iterations, leading to an interleaving of automatic and manual annotations.

The code categories to be used depend on the target architecture, which depends on the technology paradigm but also on the intended functional decomposition of the target system. Thus, depending on the type of decomposition to be performed, we consider either technological categories, like user interface, application logic, and data management, or functional ones, like the contribution to particular services for managing accounts, customers, employees, etc.

2. Reverse Engineering

From the annotated source code, a *graph model* is created, whose level of detail depends on the annotation. For example, a method wholly annotated with the same code category is represented as a single node, but if the method is fragmented into several categories, each of these fragments has to have a separate representation in the model. The relation *R1* between the original (annotated) source code and the graph model is kept to support traceability. This step is a straightforward translation of the relevant part of the abstract syntax tree representation of the code into its graph-based representation.

The AST representation is more adequate, both from a performance point of view and because of the amount of information present, to the annotation process. However, for the redesign step, the graph representation allows us to abstract from the specific programming languages involved and to describe transformations in a more intuitive way. Additionally, given that we only represent in graphs the elements that we need according to the annotation, as explained in the previous paragraph, the model to be transformed is simpler and the performance needs are not so demanding.

The graph model is based on a *metamodel* which is general enough to accommodate both the source and the target system, but also all intermediate stages of the redesign transformation. Additionally, this metamodel contains the code categories that were available in the code annotation step. An example of graph model and metamodel (type graph) is presented in Figure 7.5.

3. Redesign

The source graph model is restructured to reflect the association between code fragments and target architectural elements. The intended result is expressed by an extra set of constraints over the metamodel, which are satisfied when the transformation is complete. 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.

This *code category-driven transformation* is specified by graph transformation rules, conceptually extending those suggested by Mens et al. [365] to formalise refactoring by graph transformation. Indeed, in our approach, code categories provide the

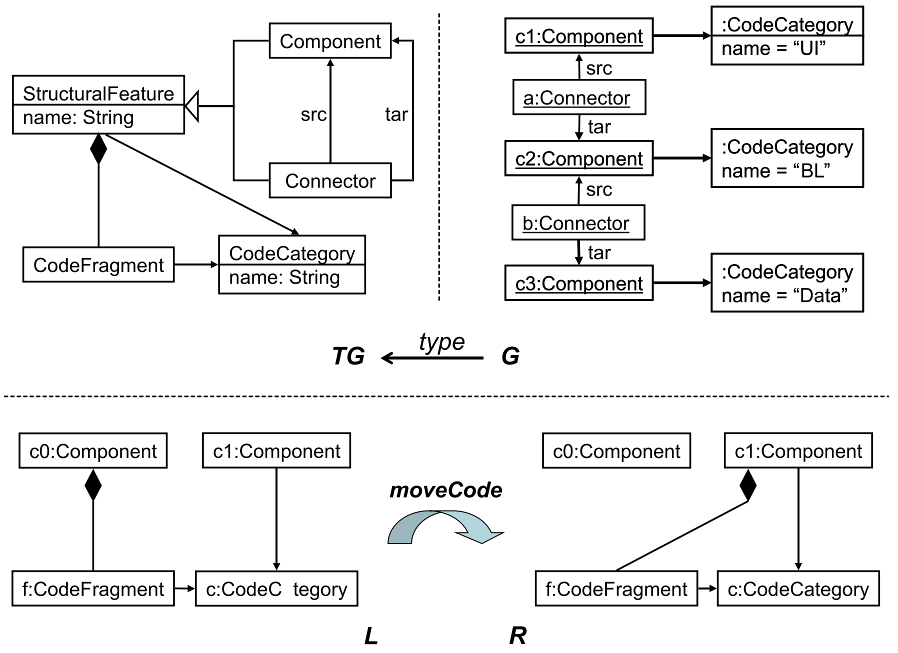


Fig. 7.5. Type and instance graph (top) and transformation rule (*bottom*)

control required to automate the transformation process, focussing user input on the annotation phase. An example of graph transformation rule can be seen in Figure 7.5.

Rules as well as source, target and intermediate graphs are instances of the meta-model. Additional *target constraints* are given to specify the success criteria of the transformation.

4. Forward Engineering

The target code is either generated from the target graph model and the original source code or obtained through the use of refactorings at code level. The result of this step, the annotated code in relation with a graph model, has the same structure as the input to Step 1. Hence the process can be iterated.

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 (cf. Section 7.2). For example, if the first iteration separates application logic and data from user interface code, the latter can be removed (and substituted with the appropriate service infrastructure) in a second round of transformation, thus exposing basic functionality as a service.

7.3.2 Redesign by Graph Transformation

Next we 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 [398] 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 [332] whose vertices and edges are decorated with textual or numerical information, as well as inheritance between node types [157, 365, 370].

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* [129], 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 in the top right of Figure 7.5 is an instance of the type graph in the top left, with the mapping defined by $type(o) = C$ for each instance node $o : C$.

In order to define more precisely the class of instance graphs, constraints can be added to the type graph expressing, for example, cardinalities for in- or outgoing edges, acyclicity, etc. Formalising this in a generic way, we assume 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 we write $Inst(TG, C)$ 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 paper implement a mapping from a general class of (potentially unstructured) systems into a more specific one 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 our 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 have to have the same type, etc.) 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.

Rule *moveCode* in the lower part of Figure 7.5 specifies the relocation of a code fragment (package, class, method, etc.) 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 our 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 our 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. We write $G \overset{\surd}{\Longrightarrow} H$ 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 [227], 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 [456], 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 we have 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 our single rule in Figure 7.5 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.

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 to define 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 our 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 our case) where the rule is not applicable anymore.

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 [60]. It is decidable under the condition that the transformation systems is terminating. The algorithm has been transferred to graph transformation systems [419] and prototypical tool support is available for part of this verification problem [487].

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 our 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.

7.4 Implementation and Example

In this section we describe an implementation of our methodology, demonstrating it on an example. This implementation addresses the technological decomposition (cf. Section 7.2) that is one of the steps to achieve SOA. The four-step methodology presented and formalised in Section 7.3 is instantiated for transforming Java 2-tier applications to comply with a 3-tier architecture.

We present the metamodel definition and the four steps as applied to a simple example, a Java client-server application composed of twenty one classes and over three thousand lines of code (LOC). The example was chosen to illustrate the kind of entanglement between different concerns that is typically found in the source code of legacy applications. For presentation purposes, in this chapter we will focus on a couple of methods of one of the classes only. Both categorisation and transformation are based on a metamodel describing the source and target architectural paradigms.

7.4.1 Metamodel

The metamodel is composed of two parts, detailing code categories and the architectural and technology paradigms used. Its definition is a metalevel activity, preceding the actual reengineering process. The same metamodel (or after slight changes) can be used in different projects where the source and target architectural and technology paradigms are similar.

Code Categories

As stated in Section 7.3, code categories are derived from the target architectural and technology paradigm. Different models can be used for the categories. We have opted for the one presented in Figure 7.6 and explained next, together with our instantiation.

In the chosen model, code categories can be divided in 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)

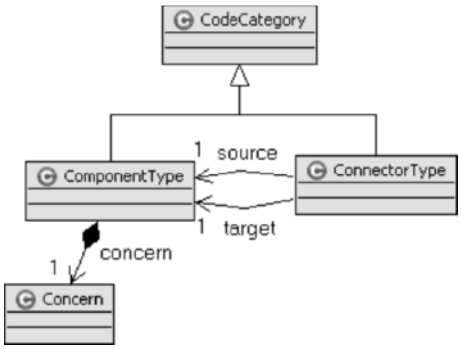


Fig. 7.6. Code categories model for 3-tier

- 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; other architectural paradigms might require different categories and more complex ways to represent them. It may be even desirable in some situations to allow multiple categories for the same element. In Figure 7.6, components and connectors are represented by "ComponentType" and "ConnectorType", respectively, in order not to use the names attributed to architectural concepts. We have both "ComponentType" and "Concern" for reusability issues given that the first is likely to be extended for certain types of target architectures. For instance, if our 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".

Architectural and Technology Paradigm

The next metalevel activity consists in 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. As we are going to take advantage of graph transformation rules in the transformation specification, we developed the model in the form of a type graph.

The model shown in Figure 7.7 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 instantiations of our methodology. The type graph is an extension of the one presented by Mens et al. in [368] 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, we have included the concepts of *Component* and *Connector* that allow us to represent 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 we generate Java code from the model in this implementation.

During a transformation, we may have 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.

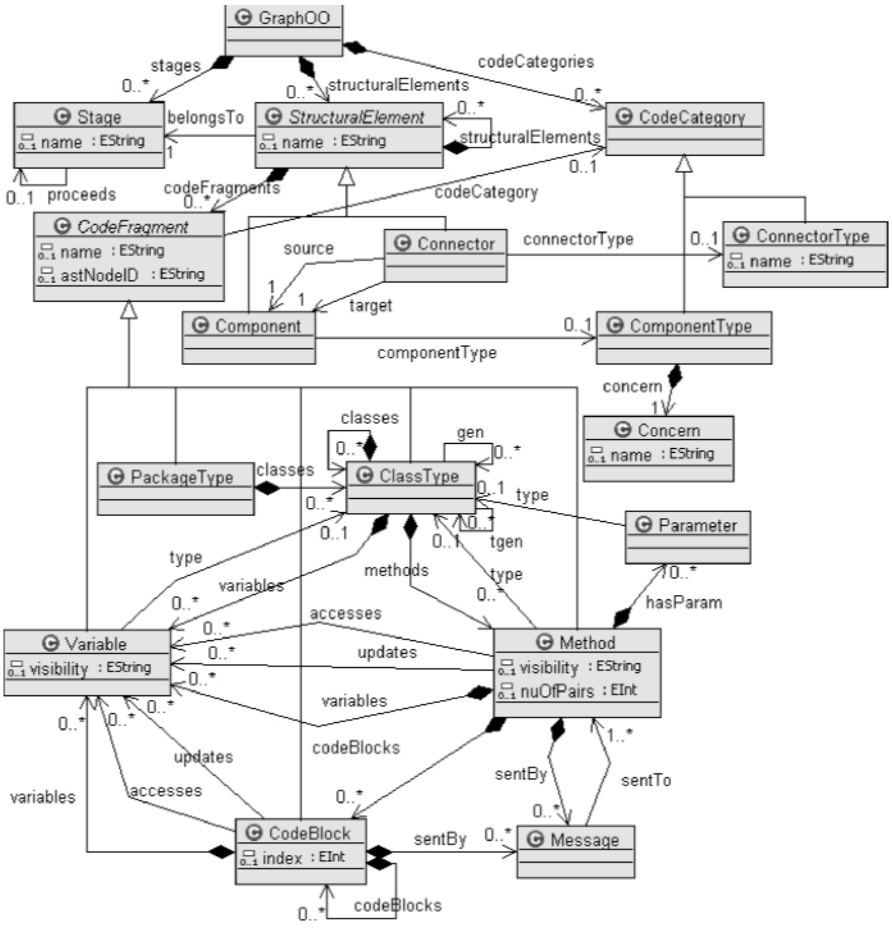


Fig. 7.7. Type graph for the OO paradigm

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 we want to be as language-independent as possible we did not link the type graph directly to the source code. Instead, we used an attribute (ASTNodeID) 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 our case, allow for a loose integration between the model and the programming language.

7.4.2 Code Annotation

The annotated source code is obtained through an iteration of manual input and the application of categorisation rules, based on the categories defined in the metamodel.

Categorisation Rules

The rules used in the categorisation process are applied over the AST rather than based on the graph-based presentation. The following examples are presented informally.

1. Statements that consist of variable/attribute declarations for a type that is known to belong to a certain concern, will be categorised as belonging to the same concern.
   Example: the Java statement '`private JLabel lbNo;`' is categorised as UI Definition because it is known that *JLabel* belongs to the UI concern;
2. Assignments 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.
   Example: the Java statement '`lbNo = new JLabel ("Account No:");`' is categorised as UI Definition because it is known that the attribute *lbNo* and the *JLabel* method/constructor invocation belong to the UI concern;
3. Variables/attributes/parameters definition/assignment 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 known to belong to that category.

The same rule might have to be applied multiple times. The reason for this is that the application of a rule can enable the application of another. An example for this can be given using rule number 2: if a method invocation that exists in the right-hand side of the assignment is not yet categorised, the rule will not be applied. However, after some other rule categorises the method, rule number 2 can be applied. The transformation stops if no more rules are applicable or the code is completely categorised. Given that our rules do not delete previous categorisations nor change them, the transformation is guaranteed to terminate. For pattern matching and rule application over ASTs we can use the L-CARE tool, which provides a scalable solution to program transformation problems.

Example

As mentioned above, the original source code is categorised considering the intended target architecture. In Figure 7.8 we present the code that has been previously used to explain the implementation of our methodology. The code is annotated using simple comments.

In this paragraph we explain how the categorisation was achieved for some of the statements of this example. The first three attribute declarations are categorised as UI Definition based on the first categorisation rule previously presented, since it is known that *JLabel*, *JTextField* and *JButton* belong to the UI concern in Java. The assignment to array *records* is categorised as Data Action based on rule number 3 which states that variables assignments used to store values from Data Action functions belong to the same category, and the *readUTF()* operation belongs to this category. This enables the categorisation of the assignment of *dis* as Data Definition through rule 2 since this variable is used next in Data Action code.

The need of several iterations of the categorisation process is now clear. In the first iteration the assignment of *dis* could not be categorised, but after categorising some of the following statements as Data Action, a second iteration is able to identify this statement as Data Definition.

7.4.3 Reverse Engineering

After having annotated the code, the process of transforming it into a graph model is straightforward. Its level of granularity is controlled by the results of the categorisation and the needs of the transformation process.

Program Representation

The graph model together with its traceability relation to the original code constitutes the *program representation*. This is an instance of the type graph previously defined and shown in Figure 7.7, where the code is categorised and its dependencies are defined. An example can be seen in Figure 7.10. The value "*" for the attribute "name" of the "concern" means that the element contains more than one concern. For example, the "populateArray" method contains three code blocks that include the concerns "UI" and "Data". This graph is obtained from the AST presented in Figure 7.9b. The corresponding source code can be seen in Figure 7.9a.

In this section only some of the elements of the Class "DepositMoney" are being presented, namely the attribute "lbNo" and the methods "txtClear" and "populateArray". 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 "populateArray" method is called each time it is necessary to refresh the data in the window.

```
\\concern = *
package General;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Deposit Money extends JInternalFrame implements ActionListener {
\\concern = UI
private JLabel lbNo, lbName, lbDate, lbDeposit;
private JTextField txtNo, txtName, txtDeposit;
private JButton btnSave, btnCancel;
\\concern = *
void populateArray () {
\\connectorType = Control UI -> BL
    try {
\\concern = Data
        fis = new FileInputStream ("Bank.dat");
        dis = new DataInputStream (fis);
\\concern = Data
        while (true) {
            for (int i = 0; i < 6; i++) {
                records[rows][i] = dis.readUTF();
            }
            rows++;
        }
\\connectorType = Control UI -> BL
    }
    catch (Exception ex) {
        total = rows;
        if (total == 0) {
\\concern = UI
            JOptionPane.showMessageDialog (null, "Records File is Empty.
                \nEnter Records First to Display.", "BankSystem
                    - EmptyFile", JOptionPane.PLAIN_MESSAGE);
            btnEnable ();
\\connectorType = Control UI -> BL
        }
        else {
\\concern = Data
            try {
                dis.close();
                fis.close();
            }
            catch (Exception exp) { }
\\connectorType = Control UI -> BL
        }
    }
\\concern = *
}
\\concern = UI
void txtClear() {
    txtNo.setText("");
    txtName.setText("");
    txtDeposit.setText("");
    txtNo.requestFocus();
}
```

Fig. 7.8. Source code categorised

Example

The annotated source code previously presented is translated, in a straight-forward way, into the source graph model (cf. Figure 7.10). Naturally this graph has to conform to the type graph in Figure 7.7 in order for the transformation to be possible.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
// ASTNode0001
public class DepositMoney
      extends JInternalFrame
      implements ActionListener {
 private JLabel lbNo /* ASTNode0002 */,
                lbName, lbDate,
                lbDeposit; // ASTNode0010
 private JTextField txtNo, txtName,
                    txtDeposit;
 private JButton btnSave, btnCancel;
 // (...)
 // ASTNode0003
 void populateArray () {
   // ASTNode0005
   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++;
    }
   }
   catch (Exception ex) {
   // ASTNode0006
    total = rows;
    if (total == 0) {
        JOptionPane.showMessageDialog (null,
         "Records File is Empty.\nEnter
            Records First to Display.",
              "BankSystem - EmptyFile",
                 JOptionPane.PLAIN_MESSAGE);
        btnEnable ();
    }
   // ASTNode0007
    else {
        try {
            dis.close();
            fis.close();
        }
        catch (Exception exp) { }
    }
   }
}
 //(...)
 // ASTNode0004
 void txtClear() {
   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.populateArray()
     BODY
      TryCatchStatement
       ASTNode0005
       TryStatement
        EXPRESSION
        EXPRESSION
        WhileStatement
       CatchStatement
        ASTNode0006
        EXPRESSION
        IfStatement
         THEN_STATEMENT
         ASTNode0007
         ELSE_STATEMENT
          TryCatchStatement
          (...)
   MethodDeclaration
    ASTNode0004
    method binding:
    DepositMoney.txtClear()
    (...)
```

(b) Example AST

Fig. 7.9. Source code and AST extracts from a Java sample application

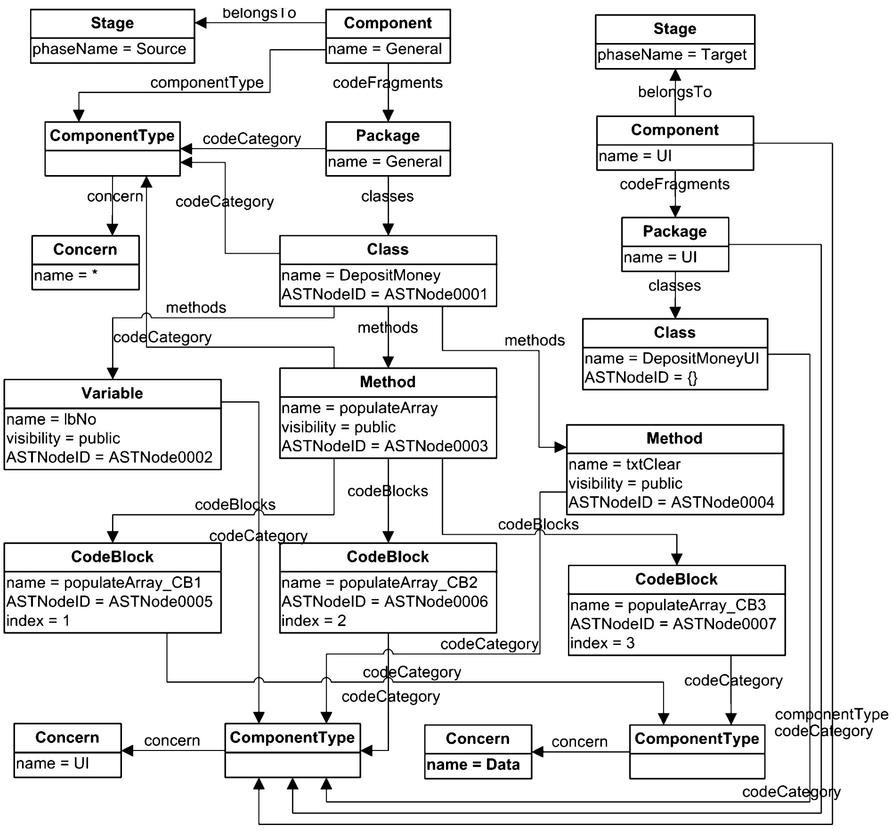


Fig. 7.10. Graph representing a subset of a Java sample application

For this translation one can parse the annotated AST and create the corresponding instance of the type graph, which will next be used as the start graph for the architectural redesign.

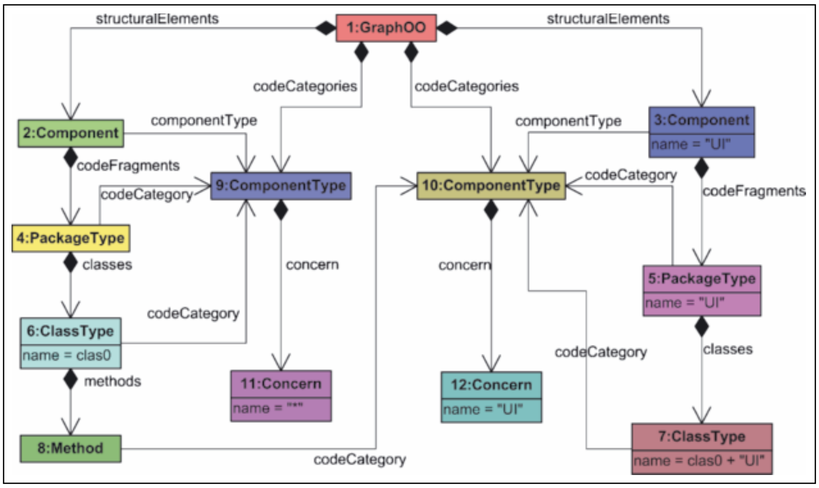
7.4.4 Redesign

For transforming the graph model we create transformation rules that, applied to the source model, yield a model complying to the target constraints.

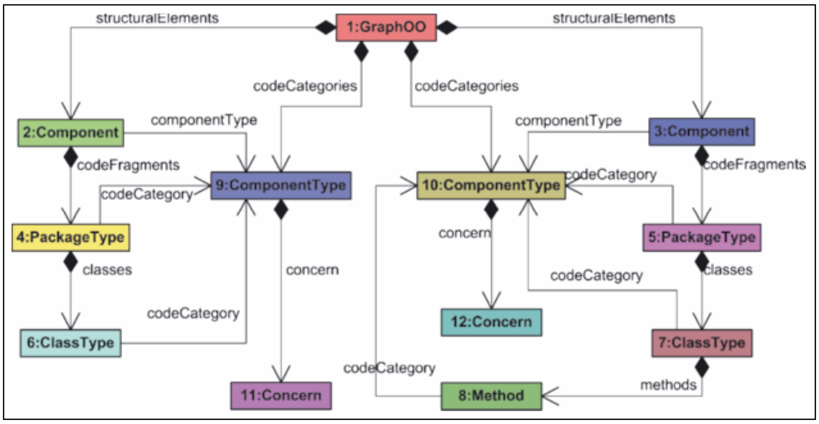
Transformation Specification

A sample transformation rule is given in Figure 7.11.

Its specification, according to the graph transformation rules fundaments previously formalised and explained, is defined visually in Tiger EMF Transformation [488]. This is an Eclipse plugin for model transformation that allows to design rules and apply them to an instance graph.



(a) Left-Hand Side



(b) Right-Hand Side

Fig. 7.11. Move Method UI transformation rule

The approach is similar to refactoring by graph transformation [370], except for the use of code categories for controlling the application. More generally, refactoring rules may not be enough for all redesign transformations because sometimes it is necessary to apply changes that are not behaviour preserving. An example is the transformation of 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 into 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.

In order to ensure that the target model complies with the desired architecture, it is possible to define constraints over the metamodel that correctly reflect the architectural paradigm. For instance, in 3-tier applications:

- there should be no UI and Business Logic methods in the same class
- no direct links from UI to Data allowed
- ...

Transformation Execution

The example graph for the BankSystem application seen previously (Figure 7.10) is a candidate for the application of the Move Method UI transformation. This transformation is an example of a rule that contributes to the technological layering of the application.

As we can see from the transformation rule, this graph has an occurrence of the LHS. As a result, we can apply the rule, obtaining the graph shown in Figure 7.12.

The method "txtClear" was moved from the class "DepositMoney" to "DepositMoneyUI", a class belonging to the UI concern.

The execution of transformation rules can either be based on a tool that interprets the transformation specification, or a manually developed transformation program using the set of rules as requirement specification. We are presently using the code generation facility of the Tiger tool.

Example

At this stage we have a graphical representation of the categorised source code conforming to the type graph. Having designed the transformation rules in Tiger EMF we can generate the transformation code automatically.

As an example, when we apply the transformation rule Move Method UI illustrated in Figure 7.11 to the start graph of Figure 7.10, we obtain the representation shown in Figure 7.12.

If we keep applying appropriate transformation rules, this graph will be transformed until the representation achieved complies to the intended target model. The rule Move Variable UI can transfer variable *lbNo* to the class *DepositMoneyUI*. Applying the rules Move Code Block Data, followed by Move Code Block UI and finally Move Code Block Data again, we can completely transform the source graph into one that conforms to the constraints defined for the target model. This graph is presented in Figures 7.13 and 7.14. The size of this graph made us divide it into two figures to render it readable. The ellipses show the connections between the two Figures. The first shows the architecture level and includes packages. The second includes all the information from the code block level until class level. For transforming the whole application from 2-Tier to 3-Tier, 26 transformation rules are needed.

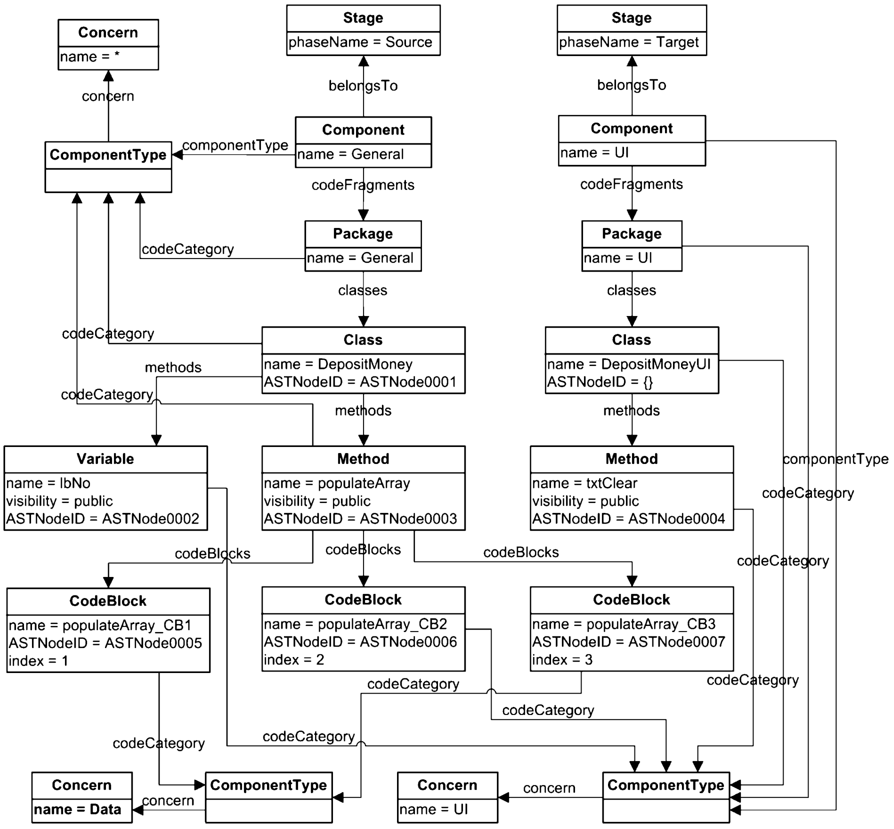


Fig. 7.12. Graph representing a subset of the BankSystem sample application after the execution of rule Move Method UI. { } represents new AST nodes identifiers created during the transformation process

Some of these rules, namely the ones that move code blocks, are quite more complex than Move Method UI, which was selected to be presented for readability purposes.

7.4.5 Forward Engineering

The target code can be achieved using two alternative strategies discussed below. However, we are still exploring both of them to see their practicality, challenges and limitations.

1. During transformation execution, a log of the applied rules is kept, to be replicated at the code level using a standard refactoring tool. This requires to associate each graph transformation rule to one or more standard code refactorings. Depending on the complexity of the rules or the specificities of the situation, traditional refactorings may not be enough, in which case it is necessary to develop more complex code level transformations.

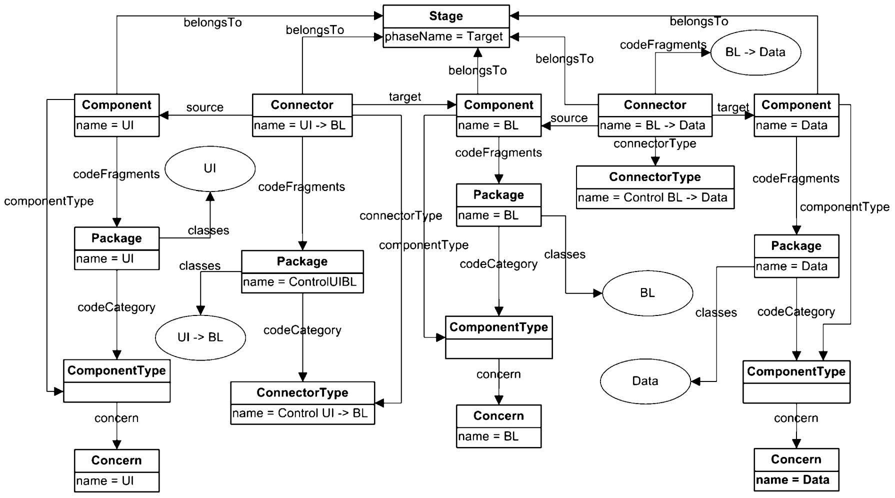


Fig. 7.13. Graph representing a subset of the BankSystem sample application after being transformed (Architecture and packages). The ellipses show the connections with Figure 7.14

2. Alternatively we can generate the code directly, using the target program representation and the links to the original AST. The code can be generated top-down, copying most of it from the original source code to the new structure and generating the necessary "extra code". This "extra code" can be, for example, the code that changes method invocations when the called method has moved to a new class.

Example

Finally the code complying to the desired architecture can be generated. For this purpose, we use the target model previously generated as well as the traceability relation with the original source code. In this process the code is refactored to be coherent with the model and *glue* code vital for the preservation of the application's functionality is created.

A sample of the code transformed is presented in Figures 7.15, 7.16, 7.17 and 7.18. The code is not integrally presented for simplicity reasons.

7.5 Related Work

Three areas of research constitute relevant work related to the approach presented in this chapter: program representation/reverse engineering, program transformation and code generation. However, the work in these areas is evolving mostly independently, i.e., not as part of an integrated reengineering methodology as in our case. We also present recent work that addresses specifically reengineering to SOA.

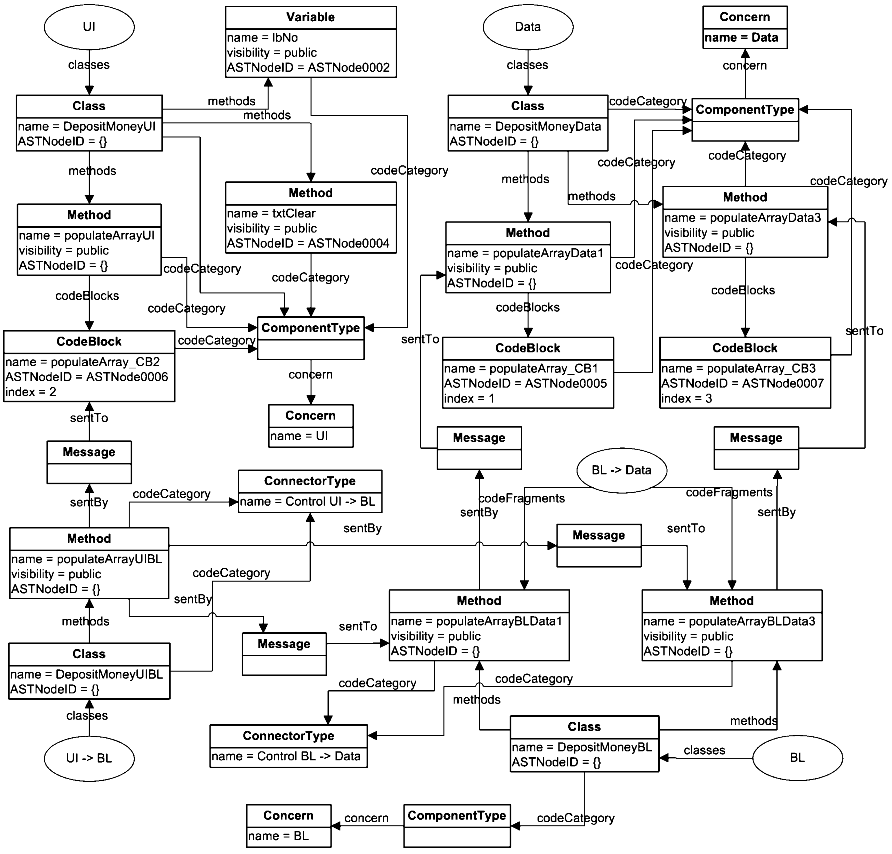


Fig. 7.14. Graph representing a subset of the BankSystem sample application after being transformed (from code block level until class level). The ellipses show the connections with Figure 7.13

Regarding the area of source code representation and reverse engineering, we briefly describe a few examples that show how this issue is dealt with in different contexts.

The Dagstuhl Middle Model (DMM) [325] 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 [395] provides design pattern [190] 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++ [173]. 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 [426], addresses

```
package SoftwareEvolutionBookChapter.ControlUIBL;

import SoftwareEvolutionBookChapter.UI.DepositMoneyUI;
import SoftwareEvolutionBookChapter.BL.DepositMoneyBL;

public class DepositMoneyUIBL {

    private int rows = 0;
    private int total = 0;

    DepositMoneyUI depositMoneyUI = new DepositMoneyUI();
    DepositMoneyBL depositMoneyBL = new DepositMoneyBL();

    void populateArrayUIBL() {
        try {
            depositMoneyBL.populateArrayBL1(rows);
        } catch (Exception e) {
            rows = depositMoneyBL.getRows();
            total = rows;
            if (total == 0) {
                depositMoneyUI.populateArrayUI();
            } else {
                depositMoneyBL.populateArrayBL3();
            }
        }
    }

}
```

Fig. 7.15. Code Transformed (Package ControlUIBL)

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. One major difference between our methodology and the above approaches is that ours 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 [250], like our approach, 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 ours in several aspects. Firstly, the goal of the framework is program understanding and not the creation of a representation that is aimed to be used as input for the transformation part of a reengineering methodology. Another important difference is that in ARTISAn the categorisation process (called "labeling") is based in clues that result in the categorisation of classes only. In our approach we need, and support, the method and code block granularity levels.

The next related area is program transformation, which can occur in different levels of abstraction. The source-to-source level of transformation is the most established one, both in research and in industrial implementations. There are several research ideas that led to successful industrial tools. Examples from research include TXL [126] and ASF+SDF [513]. DMS from Semantic Designs [46] and Forms2Net from ATX Software [12] are program transformation tools being successfully applied in the industry. Transformations at the detailed design level, due to its applications as

```
package SoftwareEvolutionBookChapter.UI;

import javax.swing.JOptionPane;
import javax.swing.JTextField;

public class DepositMoneyUI {

    private JTextField txtNo, txtName, txtDeposit;
    private int rows = 0;

    public int getRows() { return rows; }

    public void populateArrayUI() {
        JOptionPane.showMessageDialog (null, "Records File is Empty.\n
                                       Enter Records First to Display.",
                "BankSystem - EmptyFile", JOptionPane.PLAIN_MESSAGE);
        btnEnable ();

    }

    //Function use to Clear all TextFields of Window.
    void txtClear () {
        txtNo.setText ("");
        txtName.setText ("");
        txtDeposit.setText ("");
        txtNo.requestFocus ();
    }

    void btnEnable() {
        // ...
    }

}
```

Fig. 7.16. Code Transformed (Package UI)

maintenance techniques, have an increasing interest that is following the same path. Practices such as refactoring [183] 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 [494] and IntelliJ [255]. However, there is still a lot of ongoing research in this area, for instance, in the determination of dependencies between transformations [368].

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 chapter. The approaches found in the literature vary in three main things: 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.

Kong et al. [288] developed an approach for software architecture verification and transformation based on graph grammar. First, the approach requires translating

```
package SoftwareEvolutionBookChapter.BL;

import SoftwareEvolutionBookChapter.Data.DepositMoneyData;

public class DepositMoneyBL {

    DepositMoneyData depositMoneyData = new DepositMoneyData();

    public int getRows() { return depositMoneyData.getRows(); }

    public void populateArrayBL1(int rows) throws Exception {
        depositMoneyData.populateArrayData1(rows);
    }

    public void populateArrayBL3() {
        depositMoneyData.populateArrayData3();
    }

}
```

Fig. 7.17. Code Transformed (Package ControlBLData)

```
package SoftwareEvolutionBookChapter.Data;

import java.io.DataInputStream;
import java.io.FileInputStream;

public class DepositMoneyData {

    private FileInputStream fis;
    private DataInputStream dis;

    private String records[][] = new String [500][6];
    private int rows = 0;

    public int getRows() { return rows; }

    public void populateArrayData1(int p_rows) throws Exception {
        rows = p_rows;
        fis = new FileInputStream ("Bank.dat");
        dis = new DataInputStream (fis);
        while (true) {
            for (int i = 0; i < 6; i++) {
                records[rows][i] = dis.readUTF();
            }
            rows++;
        }
    }

    public void populateArrayData3() {
        try {
            dis.close();
            fis.close();
        }
        catch (Exception exp) { }
    }

}
```

Fig. 7.18. Code Transformed (Package ControlData)

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. Also, automatic transformation can also be applied but only at the architecture description level and not at the implementation level.

Ivkovic and Kontogiannis [244] proposed a framework for quality-driven software architecture refactoring using model transformations and semantic annotations. In this method, first, 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 [183] refactorings.

Fahmy et al. [165] used graph rewriting to specify architectural transformations. They used PROGRES tool [70] to formulate executable graph-rewriting specifications for various architectural transformations. They represent architecture using directed typed graphs that represent 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 three previous works, the approach of Carrière et al. [106] 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 our work relies on the fact that we use code categorisation to relate the original source code with the intended target architecture. We also transform at model level while this approach does it at code level.

Regarding code generation, there is a significant number of research work and tools available. A comprehensive list is already too long to specify in the context of this chapter so we only name a few. The already mentioned Fujaba tool suite supports the generation of Java sourcecode from the design in UML resulting in an executable prototype. The Eclipse Modeling Framework (EMF) [495] can generate Java code from models defined using the Ecore meta-model. This has a number of possible uses such as to help develop an editor for a specific type of models. 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 [120].

Work in the area of reengineering to SOA is new. It primarily focuses on identifying and extracting services from legacy code bases and then wrapping them for deployment on a 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. Sneed [465] 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 is identifying candidate functionality for wrapping as a web service. The second step is locating the code of the functionality, with the aid of reverse engineering tools. The third step is extracting that code and reassembling 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. The fourth step is wrapping the component extracted with a WSDL interface. The last step is linking the web service to overlying business processes by means of a proxy component.

A lighter code-independent approach was developed by Canfora et al. [100], 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 organisation. There wrappers 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 a 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 our work is the service identification and extraction task, which is closely related to the vertical dimension mentioned earlier in this chapter, but not reported here. This task is essential for any code-wrapping approach to reengineering to SOA. Some works focus primarily on this aspect. For example, Del Grosso et al. [145] proposed an approach to identify, from database-oriented applications, pieces of functionality to be potentially exported as services. The identification is performed by clustering, through formal concept analysis, queries dynamically extracted by observing interactions between the application and the database. Zhang et al. [565] 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 component candidates via formal concept analysis. Static program slicing is applied to further understand these component candidates. Then, component candidates are extracted, refined and encapsulated.

7.6 Conclusion

Most of the ongoing research in the context of automated software transformation, as well as existing industrial tools, 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 of addressing, in a systematic way, the complexity of the architecture-based transformation problem. In practice, when such a problem arises, these approaches have to be combined in a trial and error fashion, 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 the architectural level, but with the main goal of documenting and visualising the architecture of applications rather than supporting increased levels of automation in 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 redocumenting architectures is often limited given the time and budget constraints.

SOA is becoming a prevailing software engineering practice and presents challenges that add to the difficulty of the architectural transformation process. In this work we have presented a systematic approach in order to explicitly address these issues. This chapter has reported in detail our approach: the code annotation process, code representation, architectural transformation using graph transformation techniques.

While in this chapter we presented an instantiation of the technique to transform Java 2-tier applications to 3-tier to address the technological dimension of SOA, the general technique can be used in a variety of contexts, tailored to the specific requirements of a particular redesign problem by adapting the code annotation and transformation rules. Possible instantiations include, for example, the migration of monolithic applications into thin-client 2-tier architectures or of 3-tier applications into SOA. Our current implementation serves as a demonstration of the methodology and is incomplete in the sense that more categorisation rules may have to be added to allow a more complete automation of the step 1 of the methodology, and that code generation is not yet automated. However, from our experiments, we can see the potential of using this methodology in industry.

Presently we are in the process of completing the tools in order to apply them to a large real-world scenario. Another branch of work is to develop an instantiation of the methodology to address the functional dimension of SOA. By applying it in sequence in both dimensions it will be possible to transform legacy systems to Service-Oriented Architectures.

Acknowledgement. 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).

G. Koutsoukos participated in the development of the presented reengineering methodology while at ATX Software. M. El-Ramly contributed to this work while lecturer at the University of Leicester.