FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Java Transformation Library (jTL)

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Bernhard Eischer
Matrikelnummer 0526540

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: O.Univ.Prof. Mag. Dipl.-Ing. Dr.techn. Gerti Kappel
Mitwirkung: Univ.Ass. Privatdoz. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Wien, 19. August 2015

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Java Transformation Library (jTL)

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Bernhard Eischer
Registration Number 0526540

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: O.Univ.Prof. Mag. Dipl.-Ing. Dr.techn. Gerti Kappel
Assistance: Univ.Ass. Privatdoz. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Vienna, 19. August 2015 _____ _____
(Signature of Author) (Signature of Advisor)

Danksagung

Zu allererst möchte ich mich bei meinen Betreuern für ihre tatkräftige Unterstützung bedanken. Einerseits gilt ein großes Dankeschön Frau Prof. Gertrude Kappel, die mich durch ein zufälliges Gespräch maßgeblich motiviert hat meine Diplomarbeit unter ihrer Betreuung endlich ernsthaft und zügig zu schreiben und die mich bei der Ideenfindung erheblich unterstützt hat. Auf der anderen Seite danke ich Mag. Manuel Wimmer für die laufende Betreuung und sein offenes Ohr bei jeglicher Art von Fragen.
Ein ganz besonderes Dankeschön gilt meinem Vater Manfred Eischer, der mich von Kindesbeinen an immer unterstützt und gefördert hat und mir so den Weg für ein erfolgreiches Studium erst geebnet hat. Weiters möchte ich mich bei ihm auch für die großzügige finanzielle Unterstützung bedanken, ohne diese ein so zeitintensives Studium wohl nie möglich gewesen wäre. Einen weiteren großen Dank möchte ich auch meiner Stiefmutter Frau Edith Eischer-Reinthaler, Bakk. aussprechen, die all die Jahre vor allem eine ganz wichtige emotionale Stütze für mich war und mich auch durch manch schwierige Lebensphase begleitet hat.
Weiters bedanke ich mich bei meiner Schwester Frau Mag. (FH) Claudia Duchkowitsch, die immer eine Schwester wie aus dem Bilderbuch für mich war und ist. Nicht zuletzt durch das Korrekturlesen dieser Master Thesis hat sie auch einen wichtigen Beitrag für das Gelingen dieser Arbeit beigetragen.
Natürlich möchte ich mich auch ganz besonders bei meiner Freundin und Lebensgefährtin Frau Lic. Orsolya Lakatos bedanken. Sie war nicht nur ein ganz wesentlicher Motivator mein Studium durch das Schreiben dieser Master Thesis endlich erfolgreich zu beenden, sondern hatte während der Zeit des Schreibens immer ein offenes Ohr für meine Probleme und Sorgen gehabt. Auch wenn sie es manchmal schwer mit mir war, hat sie immer zu mir gehalten und über meine Fehler hinweggesehen. Dafür kann ich ihr gar nicht genug danken.
Auch bei meiner Firma bzw. meinen Chefs möchte ich mich dafür bedanken, dass sie mir durch Genehmigung einer Bildungskarenz bzw. durch flexible Arbeitszeiteinteilung den Freiraum gegeben haben, um diese Arbeit konzentriert und intensiv zu bewältigen.
Zu guter Letzt möchte ich mich bei meinen vielen Freunden bedanken, die mich auf verschiedenste Art und Weise unterstützt haben - sei es nun durch Motivation, Ablenkung oder intensive Gespräche. Es ist schwierige hier einige besonders herauszuheben, da ich allen für die verschiedensten Dinge dankbar bin. Trotzdem möchte ich das an dieser Stelle versuchen. Ganz besonders möchte ich hierbei meinen langjährigen Freund Dipl. Ing. Martin Bachler herausheben, der einerseits für intensive Gespräche, als auch durch ablenkende Abende für mich da war. Weiters danke ich meinem Studienfreund David Schmid, BSc., mit dem ich unzählige Übungen, Projekte und Lernsessions in meiner Studienzeit absolviert habe und wir uns des öfteren gegenseitig

motivieren konnten. Zu guter Letzt danke ich meinen lieben Freundinnen Mag. Karen Matter und Dipl. Ing. Petra Schmid, die vor allem in einer schweren Lebensphase für mich da waren und mit denen ich über alles reden konnte.

Erklärung zur Verfassung der Arbeit

Bernhard Eischer
Fleischmanngasse 5, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Abstract

Although many different model transformation languages (MTLs) and model transformation frameworks (MTFs) exist, each of them is requiring a developer to learn an additional language. Moreover since many of them follow a declarative approach, they are often less intuitive for programmers familiar with imperative language. Therefore a Java based imperative library called jTL has been developed which enables the user to write the whole transformation process in pure Java. The resulting jTL is applicable to a wide range of transformation tasks that include classical graph based source models or especially source code translation, because transformation of legacy code is still a not yet solved challenge in software migration. Since source code transformation is a quite complex and performance intensive task, the jTL represents an easy to use approach, which focuses on goals like maintainability, readability, reusability and performance improvements. To achieve these goals two similar existing approaches were examined to uncover weaknesses and deduce possible improvements for the resulting jTL. Moreover jTL further benefits from new Java 8 features like bulk operation, lambda expression, functional interface and parallel stream processing. Although jTL has been kept as simple as possible, it is still a complete transformation approach, since it also supports further important features like tracing or reuse mechanisms like rule inheritance which should be part of every state of the art transformation language.
As an evaluation, two concrete transformation examples are given that make use of jTL. This should give an idea how a transformation with jTL can be established and what differences and improvements come up with jTL in contrast to other Java based transformation solutions.

Kurzfassung

Seit in den letzten Jahren die modellbasierte Softwareentwicklung erheblich an Bedeutung gewonnen hat, entstanden gleichermaßen auch eine Reihe von Modell-zu-Modell Transformationssprachen (MTL). Heutzutage gibt es eine Vielzahl an populären Vertreter dieser Sprachen, wie zum Beispiel ATL, Kermeta oder Sprachen, die auf QVT basieren. Auch wenn diese speziell für den Zweck der Modelltransformation geschaffen wurden und damit eine benutzerfreundliche und einfache Art der Transformation ermöglichen sollen, verlangen sie dennoch vom Entwickler sich eine komplett neue Sprache anzueignen. Außerdem sind viele dieser MTLs deklarative Sprachen und verfolgen damit oft einen, für viele Entwickler, welche oft vorrangig oder ausschließlich mit imperativen Sprachen arbeiten, komplett neuen Ansatz. Die, in dieser Arbeit entwickelte Bibliothek namens jTL soll Entwicklern die Möglichkeit bieten, eine komplette Modelltransformation in der von ihnen gewohnten Umgebung, nämlich Java zu schreiben. Des Weiteren wurde jTL aber auch mit Hinblick auf komplexe Source Code Übersetzungen entwickelt. Gerade bei diesem Anwendungsfall bietet der imperative Ansatz einen erheblichen Vorteil im Gegensatz zum deklarativen Ansatz. Außerdem bietet sich mit automatisierter Source Code Migration durch direkte Übersetzung eine Möglichkeit an, die es erlaubt Legacy Code in Quelltext moderner Programmiersprachen zu übersetzen. Da in der Industrie noch immer eine große Menge an beispielsweise Cobol oder PL/1 Applikationen verwendet wird und der finanzielle und personelle Aufwand diese zu warten, die Unternehmen vor immer größere Herausforderungen stellen, kann automatisierte Source Code Transformation eine mögliche Strategie bieten, um diese Softwaresysteme in das 21 Jahrhundert überzuführen. jTL erfüllt dabei die Anforderungen um als Fundament einer solchen Übersetzung zu dienen, auch wenn natürlich noch viele weitere Herausforderungen bei dieser Art der Softwaremigration warten.
Um von den Problemen und Schwachstellen vorhandener Java basierten Transformationen zu lernen, wurde in dieser Arbeit eine bereits existierende Bibliothek namens SiTra, sowie ein ebenfalls Java basierter Transformer, der eine konkrete Implementierung einer automatischen Source Code Transformation von Cobol & PL/1 auf Java darstellt, untersucht und analysiert. Des Weiteren wurde ein besonderes Augenmerk auf neue Java 8 Features gelegt, die jTL nicht nur zu einer state-of-the-art Library machen soll, sondern von welchen diese auch im Bezug auf Wartbarkeit, Lesbarkeit und Performance profitieren soll. Auch wenn eines der Hauptziele die komfortable und benutzerfreundliche Nutzung von jTL war, wurden auch wichtige Features wie Tracing und Wiederverwendungsmechanismen wie Rule-Inheritance (Regelvererbung) implementiert. Damit bietet jTL einen kompletten Ansatz um die unterschiedlichsten Transformationen durchführen zu können.

Am Ende dieser Arbeit wird anhand einer Modeltransformation und einer Source Code Transformation die Anwendung von jTL demonstriert und gleichzeitig aufgezeigt welche Vorteile jTL gegenüber den zwei weiteren Java basierten Ansätzen bietet.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Transformation is an important field of software migration and modernization. Although transformation is the purest form of migration, because the conversion takes place in a 1:1 form, there are also some strong arguments why a transformation could face a lot of difficulties and challenges. The superior problem is that a transformation, regardless of whether code, model or data has to be transformed, is only as good as the tool and the underlying transformation library is. So there are many challenges to overcome, like the deep understanding of the source and target technology, the mostly very complex task of the mapping between them, as well as the financial effort. This master thesis focuses on a Java based transformation technique. It addresses the architecture and techniques source model transformation in general as well as source code transformation. Based on a simple transformation library called SiTra and an already existing PL/1/Cobol to Java Transformer, improvement especially by adding new Java 8 features and adaption to the architecture, should lead to a better performance of the transformation and a better readability and maintainability of transformation code. The findings should result in a library called Java Transformation Library (jTL). To test and demonstrate parts of the jTL, a transformation of PL/I source code to Java source code should serve as examples. There is also a special interest in transforming legacy code to state-of-the-art programming language, since the challenge of migrating over decades growing legacy software application is a well-known problem since many years in industry. A well-designed transformation can be the answer to this problem or at least can act as a main element of source code migration.

1.2 Aim of this Work

The aim of this work is to provide a guideline respectively a framework/library for complex transformation issue. The focus in this master thesis lies on the transformation of source code, although the findings and results should also be applicable for transformation of models, data or other issues. Although a number of different model transformation frameworks exist, each of them requires from developers to learn a different language and each of them possessing its own specific language peculiarities, even if they are based on the QVT standard. On the other hand these transformation frameworks were designed for model transformation and not for source code transformation, which would result in a worse maintainability and readability. Based on SiTra (Simple Transformer) and findings from an existing PL/1/Cobol to Java Transformer a Java Transformation Library (jTL) is developed. Adding current Java 8 features like bulk operation, lambda expression, functional interface and parallel stream processing to the transformation library, should contribute to better performance, readability and maintainability as well as a high flexibility of the transformation process. This master thesis should answer the following research questions:

- What are the weaknesses of existing transformation approaches?
- What improvements in contrast to existing approaches can be implemented?
- Which important requirements and goals for a transformation library can be identified?
- Which Java 8 features can be used to improve the transformation library and how can they be assembled?
- What initial steps are necessary to use the resulting jTL also as basis for source code transformation?
- Can the resulting jTL act as basis for source code transformation?
- How can the improvements being measured and what are the results, compared to existing solutions?

1.3 Methodological Approach

1. Literature Review A literature review should serve as the theoretical basis and an introduction to the domain of model transformation, software migration - especially in the area of

language transformations and existing Java-based transformation approaches. Further the literature review should provide a deeper insight into process models for transformations and new Java 8 features, from which a transformation process and jTL can profit.

2. Designing a running Example Two example transformation are introduced in the beginning of this master thesis, which are used throughout this work. It should act as a basis to demonstrate the application of the two analysed transformation issues of SiTra and PC2Java-Converter as well as the resulting jTL.

3. Analysis and Comparison of two existing Transformation Approaches The second chapter of this master thesis covers the analysis of two transformation approaches. The first one covers SiTra (Simple Transformation in Java) - a general library aiming to use Java for implementing model transformation. The second approach is a specific transformer, which was built to automatically generate Java Code out of Cobol or PL/I Code, simply called PLI/Cobol Transformer. A comparison of these two approaches should result in emphasizing possible strengths and weaknesses of both.

4. Process Model for generating a Source Model The resulting jTL should also be useable as a basis for source code transformation. Therefore a general process for source code transformation has to be designed and described to give a guideline which steps are necessary to generate a source model out of the underlying source code. The resulting source model then can be used as an input to the jTL. Since these preparation steps are necessary to transform also source code with the help of jTL, this part act as an excursus and should demonstrate shortly the process of generating a source model.

5. Analysis of how new Java 8 Features could support the Transformation Analysing if and how new Java 8 features, like bulk operation, lambda expression or stream processing can create added value to the transformation. Features that turn out to improve readability or performance of the jTL should further be added to it.

6. Evaluation A qualitative analysis respectively a comparison between the resulting jTL and the existing approaches of SiTra as well as the PL/1/Cobol to Java Transformer should emphasize the changes and improvements of jTL. Furthermore for measuring the execution time should demonstrate the potential improvement to the performance of jTL

1.4 Related Work

This Section is divided into two subsections. The first one addresses a rough overview of model to model transformation approaches, while the second focuses on the aspect of automated translation of legacy code, especially Cobol source code to object oriented code, especially Java.

1.4.1 Model to Model Transformation

Since model transformation is a wide spread and often used approach nowadays, there are many already existing transformation solutions. To get an overview about possible characteristics of model transformation solutions the articles [7] and [24] provide a detailed description of possible classification features. This may help a developer to choose the best suitable transformation approach, by verifying which of these characteristics are important for the transformation task, before one starts the actual work. The following enumeration should give examples for important characteristics of model transformation solutions. For a complete and more detailed description see the article [24].

1. Declarative or operational
2. Unidirectional or bidirectional
3. Scalability
4. Static or dynamic transformation approach (CRUD)
5. Traceability
6. Ability to set up reuse mechanism

Based on these characteristics the resulting output of this work jTL, can be classified as an operational, unidirectional, scalable, dynamic approach which supports tracing and reuse mechanisms by rule inheritance.

Many publications are available which focus on the description and comparison of model transformation languages like [43] [13] [15]. This work should only give a quick overview about the most common ones, that are:

- ATL [17]: developed by ATLAS INRIA & LINA. ATL supports a declarative model transformation approach, but also provides imperative additions, since some transformation problems are not intuitively solvable with a pure declarative approach. Such a mixed

approach is known as hybrid language. Further ATL comes up with a very good IDE integration in Eclipse.

- Kermeta [25]: developed by Triskell/IRISA. It follows an imperative, aspect-oriented and object-oriented approach for the model transformation description. Like ATL, Kermeta has a good IDE integration in Eclipse.

- QVT [30]: is not a concrete implementation of a model transformation approach, but a standard defined by the Object Management Group (OMG). It defines three model transformation languages consisting of QVT-Operational, which follows an imperative approach and QVT-Relations as well as QVT-Core, which both represent a declarative approach for model to model transformation. An example for an implementation of QVT-Operational is SmartQVT [1], while medini QVT [2] is an implementation example for QVT-Relations. In contrast to QVT-Operational and QVT-Relations, no implementation of QVT-core was developed.

Further examples for model transformations are RubyTL [6] which follows a hybrid transformation approach or TGG (Triple Graph Grammar) [18]. It is called Triple Graph Grammar, since the transformation task consists of three graphs, that are the source- and the target graph and the transformation rules that are also represented by a graph called correspondence graph. The TGG is a context-sensitive approach and can be classified as a declarative transformation approach. Altogether have in common that they require from the developer to learn a new language for the transformation task. Further declarative approaches seem to be not the best solution for complex transformation tasks like source code translation. Although source code can be represented by a model like an AST (Abstract Syntax Tree) a fully imperative approach for transformation is the more suitable and comfortable approach. SiTra [38] [1] developed by the University of Birmingham, is an approach that comes up with a light weighted and fully imperative Java-based model transformation language. Since the whole transformation process can be implemented as pure Java, SiTra can be easily used by any Java developer without the need of learning further transformation languages. But there are also some problems that come up with Sitra. Based on this library and a not yet finished Cobol/PLI to Java Transformer from the industry, a general Java 8 transformer library should be developed which also addresses source code transformation and profit from uncovering problems of existing solutions.

[1] QVT - Relations Language Modellierung mit der Query Views Transformation [30]
[2] Official Website: http://projects.ikv.de/qvt [23]

1.4.2 Migration/Transformation of Legacy Code

Legacy code, especially Cobol code is still widely used in industry. Although, in most cases, the performance of Cobol applications is still better than their equivalents in Java, there are a lot of problems when dealing with legacy Cobol applications. The following enumeration should give an overview about the main problems with Cobol applications and other corresponding legacy programming languages [27] [22]:

- **Staffing problem:** Nowadays, it's hard for companies to find developers who have high Cobol skills, since the education in software development concentrates on more recent programming languages like Java or the .NET-Framework.
- **Cost problem:** Higher cost for mainframe infrastructure has to be accepted.
- **Maintenance problem:** Regardless of the staffing problem, the maintenance of monolithic Cobol applications is always a major challenge. Since average Cobol modules consist of about 600 lines of code [10] and further comes up with the bad smell of goto implementations, the maintenance of a Cobol application is in general much harder than it would be for a common Java application, where the average module size is about 30 lines of code. Furthermore the high popularity of Java leads to wider acceptance of unit testing and refactoring techniques that further improves the maintainability.
- **Not state-of-the-art:** Java or .net languages, for example, simple and yet powerful programming languages come up with state of the art technologies like abstraction, modularization, encapsulation and an extensive built-in class library. On the other hand Java refrained from implementing problematic programming concepts like goto-statements. Although Java defines `goto` as an keyword it never have been implemented. Despite the fact that Cobol moved forward too and provides object oriented aspects like classes and polymorphism, these concepts are very rarely used because most Cobol applications exists over decades and where developed at a time where these concepts where not part of the programming language.

So there are many strong reasons for moving from Cobol to state of the art programming languages. But in most cases the migration task is also a very complex and difficult challenge. Problems like poor documentation of existing Cobol applications, less knowledge about the technical implementation as well as the application context and a high financial expense as well as a high time effort for the migration process, will arise in big migration projects.

Nevertheless, if the decision is taken to move from a legacy programming language to a state of the art language there are three different approaches of migration that can be faced [22]:

1. **Reimplementation of source code:** This software migration approach is carried out by developers and can be seen as manual code transformation technique. Since this software migration strategy requires a very deep understanding of the source code and the application context, the time effort as well as the financial effort is enormous for the process of code comprehension and the translation of the source code. In large project this is a nearly insurmountable barrier to overcome. Furthermore a huge amount of tests have to be executed to ensure the same functionality of the reimplemented software in contrast to the previous version. Nevertheless the big advantage of this approach is that a well executed reimplementation project leads to a state of the art software application which ensures readability, maintainability and extensibility.

2. **Source code wrapping:** This approach describes the encapsulation of legacy code, without changing the data and source code of the legacy application. The new system accesses the legacy application over interfaces. The big advantages of the migration strategy 'Wrapping' is that it comes up with a minimal risk and less time effort. Apart from that, the initial problematic of bad written legacy code that is difficult and complex to maintain, comes up with no or less documentation and long for developers who are familiar with the legacy technology, is still present. So this migration strategy is only a short term solution.

3. **Conversion:** The software migration strategy conversion is an automated source code transformation to the target environment. The quality of the output of the transformation depends heavily on the quality of the transformation tool. Since programming language translation is a very complex task the development of such a transformation tool is also a time and cost intensive task. So it also depends on the amount of source code which has to be transformed whether this migration strategy is appropriate or not. Furthermore some refactoring work after the actual transformation process will be necessary in most cases to further improve the quality of the target source code.

Transformation as a possible migration strategy of source code especially from legacy code like COBOL or PL/I to state of the art programming language like Java is still an unsolved problem in science and industry. Sneed [22] illustrates some projects where a transformation of COBOL to object oriented language e.g. Java code were carried out. Most of these migration projects failed or followed a different path. In the early nineties the IBM labour in Toronto made the first attempt of transforming COBOL source code to object oriented code [45], but changed their

strategy soon to encapsulation of COBOL code. Another project which goal was to develop a Cobol-to-Java transformation, named Relativity from the company Triangle Technologies failed, because the refactoring effort after the transformation was too high [44]. But there are also some first achievements like a more or less successful transformation process from Cobol to Java shows, which was developed by the university of Vienna and was carried out by an Austrian social insurance group [41]. The resulting Java code is executable and semantically correct, although the appearance of the resulting Java Code corresponds more to Cobol than to object oriented code. Since there are about 200 billion lines of Cobol code [36] working worldwide, the question of how migrating them is still an important not yet answered question. A well-designed, good maintainable and easy to use Java Transformation Library could act as a basis for complex software migration project, although there are much more challenges to overcome.

1.5 Structure of the Work

This master thesis is divided into the following six chapters that are:

1. Introduction
2. State of the art
3. Java 8 features
4. jTL
5. Evaluation
6. Conclusion and Future Work

After this introduction part, the second chapter 'State of the art' emphasizes two different Java based transformation approaches. One concerns a transformation library called SiTra that is applicable for almost every possible transformation task, while the second is a concrete implementation of a PLI/Cobol to Java transformation. The analysis of them should act as basis for the design and implementation of jTL. The third chapter should conclude the current state analysis by explaining new Java 8 concepts and features that are profitable to use in a transformation library and therefore should further be part of the jTL.
The fourth chapter 'jTL' presents the goals, the architecture and the implementation of jTL in a detailed way. Moreover this Section make references to SiTra and the PC2J-Transformer to give reasons why different implementation approaches and further extensions has been applied

to jTL. The last chapter 'Evaluation' firstly points out which characteristics and implementation approaches of jTL has been established to reach the goals that have been defined in the chapter 'jTL'. Further a simple graph based transformation example is used to measure the performance improvements in execution time of jTL in contrast to SiTra. Finally, at the end of this chapter, a simple PL/1 to Java transformation example demonstrates how jTL can be used to set up a source code translation project.

1.6 Running Examples

For this master thesis two running examples has been chosen. One acts as example for graph based transformation and is used to demonstrate the development of a transformation application with SiTra and jTL. Further it used for the performance comparison between theses two transformation libraries. This example is a well known model transformation example taken out of the literature and transforms a class diagram into an entity relationship diagram. The meta models which act as reference for this example transformation is demonstrated in Figure 1.1. For the sake of clarity these example meta models are always used for this kind of transformation example but it will be slightly adjusted to simplify some demonstrations. Moreover the same applies to the concrete class model, that will also differ slightly from one application example to another.

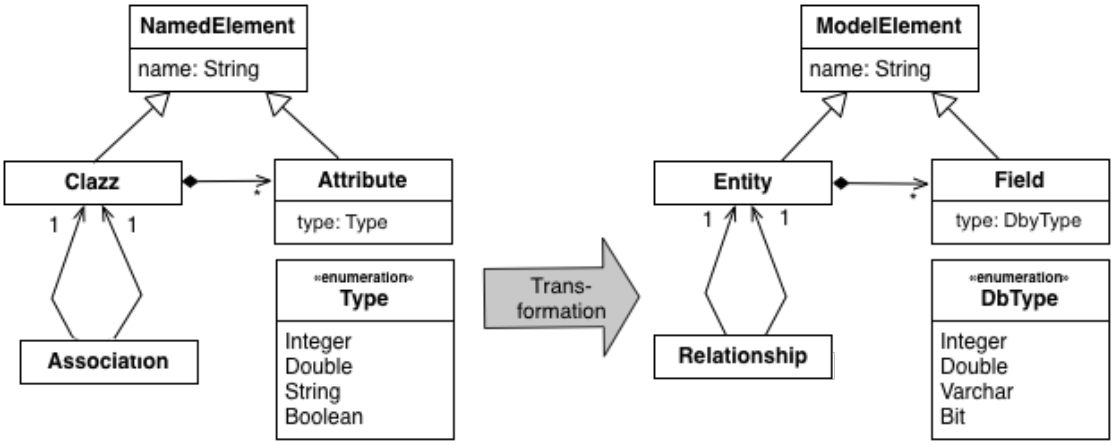

Figure 1.1 Simple metamodels as basis for the transformation of a class diagram to an ER Diagram

The second example should demonstrate a source code translation that can be classified as tree based transformation. Therefore a simple PL/1 code snippet, consisting of variable declarations, assign-statements, if-statements and expressions was designed that further is represented as source object tree (AST) that represents the source code. Therefore Figure 1.2 represents a

small excerpt of an PL/1 meta-model, that is used to construct a compliant AST of the PL/1 code snippet. The concrete PL/1 example code that is used to demonstrate the functioning of the PC2-Transformer and jTL is presented once it is required in this work.

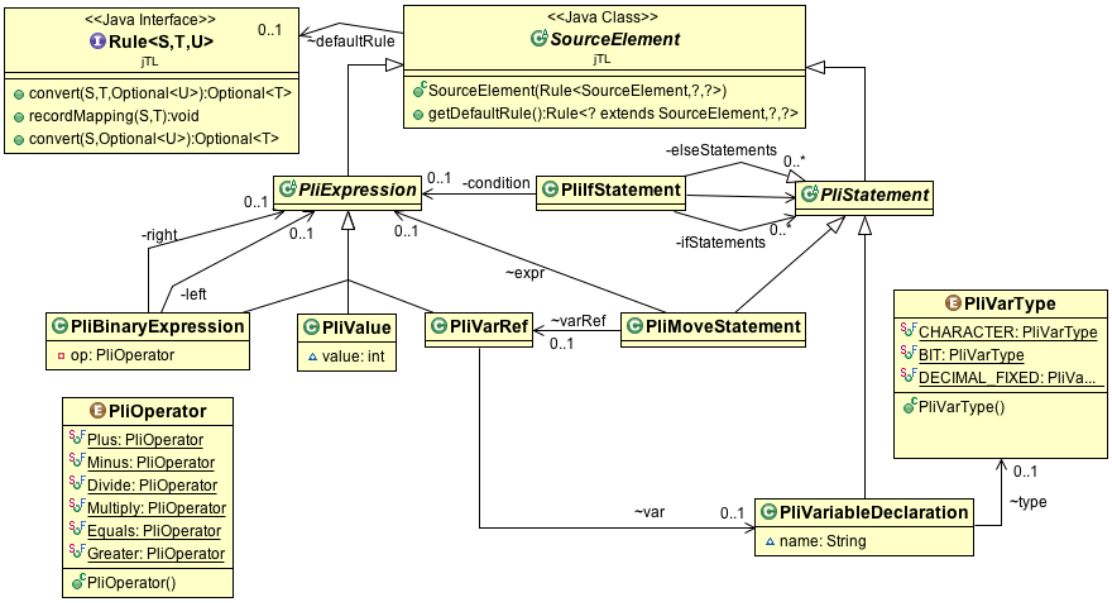

Figure 1.2 Sample Meta Model of the programming language PL/1

CHAPTER 2

State of the Art - Two Approaches

2.1 SiTra

2.1.1 Introduction

SiTra (Simple Transformer) is a small and simple Java library, which defines a framework for model driven transformation, which was designed by Behzad Bordba from the University of Bristol. The goal of this library was to use Java to simplify the challenge of transformation by defining the general architecture and structure as well as the process of transformation.

The task of model transformation is always a complex one, because it forces the understanding of the source domain, respectively the source model and the target domain/model. Even when both domains are well known, the transformation process often requires knowledge in special transformation specification languages. Graph transformation approaches (e.g. TGG) as well as declarative rule based approaches (e.g. ATL) and even imperative approaches (e.g. QVT-O) have in common that novice users, although if they have a lot of experience in software development, have to learn new concepts, new languages or new tools and applications. So a lot of time has to be spent for knowledge acquisition of new environments and applications, configuration and language syntax. To overcome these issues the idea of SiTra was to simply use pure Java for the transformation by using a minimal framework to avoid the overhead of learning a completely new transformation. Although SiTra is not intended to replace a full transformation framework it's enough for the most transformation issues, especially for the transformation of source code from one language to another.

Because the SiTra framework gives very good guiding principles to cover a whole transformation

process and also represents a good basis for language transformation, it should act as a starting point for the Java Transformation Library (jTL) and therefore should be explained in more detail.

2.1.2 Framework - Overview

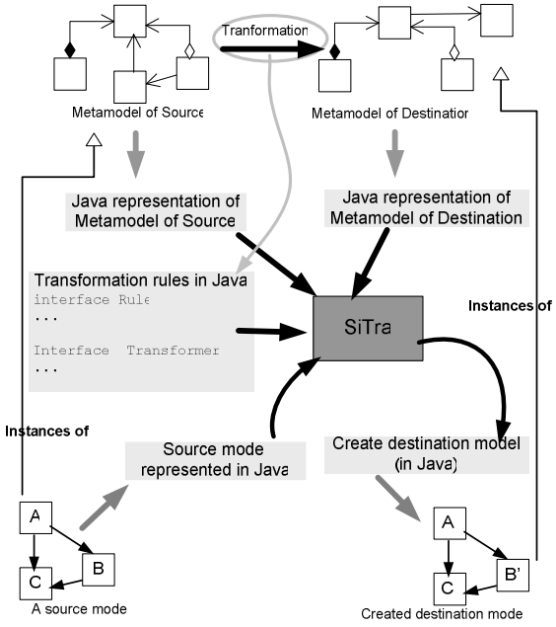

Figure 2.1 An outline of the SiTra framework [5]

A first rough overview over the architecture of SiTra is shown in Figure 2.1. As demonstrated SiTra takes a source model, which is instance of the source meta-model as input and produces a destination model corresponding to the definition of the destination meta-model. The meta-models are depicted with Java classes. So the first step when implementing a model transformation is to define the structure of the source as well as the structure of the target objects which then represent the source meta-model as well as the target-meta-model [5].

The two main components of the transformation, that are the interfaces `Rule` and `Transformer` respectively their implementation classes, are also demonstrated in Figure 2.1.

Rules: Rules are specific Java classes which implement the SiTra-interface `Rule`. They describe the guidelines how specific source objects are mapped to target objects. Rules should be split up in a manner, so that they are atomar. This fact leads to a better readability and maintainability and especially to a high reusability, whereas the rule can be reused in different context easily without repeating code in different rules. Further information to the interface `Rule` is given in the Section 2.1.3.

Transformer: The main purpose of the interface `Transformer` respectively its implementation is to execute the defined transformation on a particular source model by choosing the applicable rules (see Figure 2.1). A more detailed description is provided in the Section 2.1.3.

2.1.3 Basic Structure

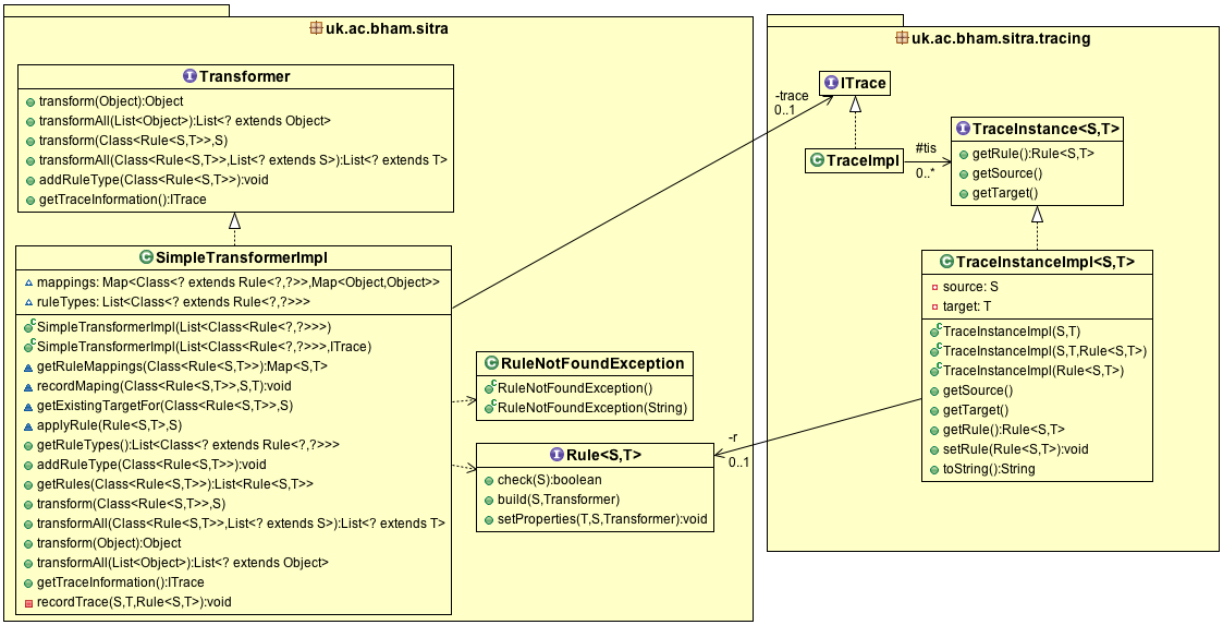

Figure 2.2 Architecture of SiTra

As shown in Figure 2.2, the SiTra library consist of two packages. The first one `uk.ac.bham.sitra` is the core component of the SiTra library, since it consists of interfaces and classes which are needed for the actual transformation, while the second package `uk.ac.bham.sitra.tracing` only tracks the information of the mapping between the source model/source elements and the target model/target elements.

1. **SiTra core package:** `uk.ac.bham.sitra`
   - `Rule`: Interface which represents the structure of a Rule.
   - `Transformer`: Interface which handles the transformation of a specified source object by choosing the corresponding rule.
   - `SimpleTransformerImpl`: An example for an implementation of the `Transformer`-interface.

- `RuleNotFoundException`: Exception which should be raised when the Transformer doesn't find a corresponding rule for the transformation of the specified source object.

2. **SiTra Tracing:** `uk.ac.bham.sitra.tracing` [40]

   - `ITrace.java`: An interface that exposes the methods that can be used for tracing.
   - `TraceInstance`: Interface that represents the mapping between one source element and its corresponding target elements, through a SiTra rule.
   - `TraceImpl`: An already delivered implementation of the interface `ITrace`, which provides several methods to query the collection of `TraceInstance`. More specifically the `resolve`-methods take a source object as input, browse the collection and return the targets elements, which have been generated during the transformation, out of the given source object. Vice versa, the `invreslove`-methods take a target object and return the source object. Unfortunately the inverse resolve method are not yet implemented.
   - `TraceInstanceImpl`: Implementation class of the interface `TraceInstances`, which hold the actual mapping between source an target elements.

Although tracing is not must for source code transformation it is an important feature for model transformation in general. Since the resulting jTL should be applicable for source model transformation too, that do not aim to transform only source code, tracing play also an important role in jTL. So the Section 2.1.3 will give a deeper insight to tracing in general.

Core Package

Rule Interface and Rule Methods The interface `Rule`, which should be implemented by every class which provide mapping information to transform the source elements, consists of three methods:

1. `check`: Before the rule is executed this method checks whether the rule is applicable to the source object or not. So the rule itself checks whether it is able to deal with the type of the source element or not. Another possible special use of this method is to check precondition, before the actual transformation starts.

2. `build`: This method should implement the actual transformation process to generate the target object out of the information of the source object. Although the signature of

this method provides the Transformer-object, it shouldn't be used for calling further rules recursively. This should be outsourced to the setProperties-method (see next point).

3. `setProperties`: After the target construct is committed, this method can be used to set values to the generated object and triggers the transformation for referenced objects or sub-objects. The distinction between the transformation of the source object in the `build`-method and the call of further transformation in the `setProperties`-method is important if the source model is graph-based and consists of cycles. Otherwise a non terminating recursive transformation cycle can occur (see Section 2.1.4).

Transformer The interface `Transformer` of SiTra already comes up with an implementation class called `SimpleTransformerImpl`. It represents the core of SiTra and is responsible for the execution of the defined transformation on a particular source model [5], especially for the selection of the applicable rules.

The core task of the `Transformer` are

1. Holding the rule repository: A collection in the implementation of the `Transformer`-interface should store all known rule-classes. Rules can be set to this collection over the constructor or can be added through the `addRuleType`-method (see next point). The transformer can then pic the applicable rule by iterating over all known rule types.

2. Method `addRuleType`: If the rule repository has not been initialized over the constructor of the `SimpleTransformerImpl`, or additional rules have to set to the repository, this can be done by using this method.

3. Methods `transform/transformAll`: Takes the source element/s as input and delegates them to the rule which are stored in the rule repository (see Section 2.1.3).

Further Transformer Methods All kinds of `transform`-methods of the interface `Transformer` take a source object (or a list of source objects in case of method transformAll) as input and return the target object (respectively a list of target objects). There are two kinds of versions of the transform method, which have to be implemented:

1. For this simple version of the transform-method only the source object has to be passed to the method. The method assumes that all rules, or to be more specific all types of rules (classes which implement the rule-interface), are potential candidates for the transformation of the passed source objects.

```
public Object transform(Object object) throws RuleNotFoundException
```

Listing 2.1 Simple transform-Method without rule specification

The advantage here is that the developer doesn't have to specify the rule for the transformation process. On the other hand the disadvantage is that every registered rule type is a potential candidate. This leads to a situation where the transformer has to call the `check`-method for every single rule (see Section 2.1.3) by passing the type of the source object, to find out whether the visited rule can handle and transform the source object or not. If a complex and extensive large transformation is carried out, this could become a very time and performance consuming job. Nevertheless in most cases this would be the preferred method, since the user do not have to establish type checks on his own.

2. Another signature of the method `transform` is listed in Listing 2.2. This version requires passing also the type of rule for the actual transformation, so that the transformer class receive information which rule should be used to transform the given source object.

```
public <S, T> T transform(Class<? extends Rule<S, T>> ruleClass, S source)  throws
    RuleNotFoundException
```

Listing 2.2 transform-method with rule specification

The advantage of this method is, that the transformer does not have to check all known rules wether they are candidates for transformation or not. Instead the method `getRules()` (see Section 2.1.4) just selects and returns suitable rules to the Transformer, by comparing the given rule type with the rule types from the rule repository.

The disadvantage is that the developer needs knowledge about all the possible rules and has to choose the right rule during compile time. This is of course an uncomfortable and in most cases unwanted overhead.

Tracing/Traceablility

Tracing or Traceability can be understood as the runtime footprint of transformation execution. A common form to hold trace-information is a so called *trace-link*, which holds the association between the source element and the generated target elements, as well as the accomplished transformation rule [7]. Establishing such links, allow defining the reverse transformation automatically [40]. These approach solves the interoperability problem between source and the target model [4]. So one reason is the change propagation in chains of transformation. Another benefit of tracing is the possibility of better debugging opportunities.

Although the jTL should focus on the transformation of programming language, it should also be applicable to a wide range of different transformation task. That's the reason why tracing should also play an important role when designing the jTL. Nevertheless it should be mentioned that the concrete transformation-task of programming language translation does not require tracing.

2.1.4 Process of Transformation

Overview

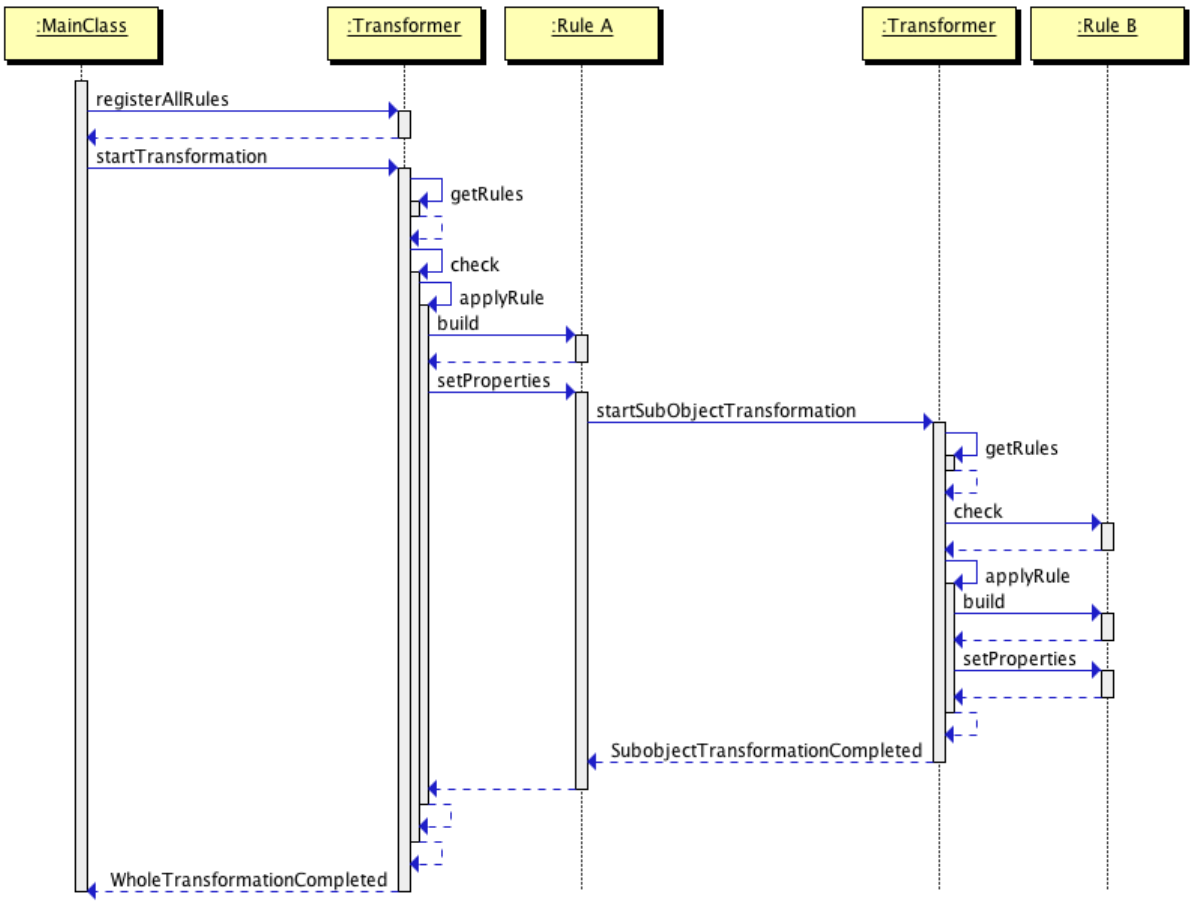

Figure 2.3 Transformation Process with SiTra

A rough overview of the overall transformation process in SiTra is given in Figure 2.3 showing an UML sequence diagram. It assumes that a source object `A` with a referenced object (or subobject) `B` should be transformed. The two lifelines of the transformer objects represent the same instance of the SiTra class `SimpleTransformerImpl` and are only split up for the reason of clarity.

The following enumeration summarizes the individual steps of the progress in more detail.

1. **registerAllRules:** Register all known rules to hold the rule types in a central repository of the Transformer-class (see Section 2.1.3).

2. **transform:** Starting point for the transformation of an individual source object (see Section 2.1.3)

3. **getRules:** This method, which is not part of the `Transformer`-interface but of the sample implementation class `SimpleTransformerImpl`, is used by the `transform`-method. It walks all registered rules and checks if the visited rule is a candidate for the transformation. Therefore the method looks if the passed rule type is a super type of the visited rule type. All rule classes, which fulfill this condition, are returned to the transform method.
   If the `transform`-method was called without specifying the rule type for the transformation of the source object, the transform-method passes a very general rule type `Class<? extends Rule<Object, Object>>` to the method `getRules`. That result in the fact, that the method `getRules`, will return every single rule type which is stored in the rule repository as potential candidate for the transformation of the given source object. Listing 2.3 shows how the potential rule types are selected.

```
<S, T> List<Rule<S, T>> getRules(Class<? extends Rule<S, T>> ruleType) {
        List<Rule<S, T>> rules = new Vector<Rule<S, T>>();
        for (Class<? extends Rule<?, ?>> rt : getRuleTypes()) {
            if (ruleType.isAssignableFrom(rt)) {
                rules.add((Rule<S, T>) rt.newInstance())                    }
        }
        return rules;
    }
```

Listing 2.3 Method getRules of SiTra SimpleTransformerImpl

4. **Check rules:** After all potential rule types are collected, the `transform`-method walks all the items to call the `check`-method of every single rule. A `ClassCastException` is thrown when the passed source object is not supported by the rule. If a rule has been found which fulfills the check-condition the actual transformation starts (see next item - 'Apply Rule'). So the first occurrence of a rule, which is applicable, is executed. If no transformation rule has been found to transform the source object, it is reported by raising the predefined SiTra Exception `RuleNotFoundException`.

5. **Apply rule:** Before the actual transformation starts by calling the applicable `Rule`-class, the `applyRule`-method (see Listing 2.4) checks if the rule has already deployed on the

source object. This ensures that one transformation rule is not executed twice for one and the same source object. That feature is a must if transforming a graph-based structure to avoid non terminating recursive transformation (see Section 2.1.4). If the source object has not been transformed yet the `build`-method of the rule is executed as well as the `setProperties`-method. Afterwards the transformation is finished and the target object is returned.

```
<S, T> T applyRule(Rule<S, T> r, S source) throws RuleNotFoundException {
    Class<? extends Rule<S, T>> ruleType = (Class<? extends Rule<S,T>>)r.getClass
        ();
        T target = getExistingTargetFor(ruleType, source);
        if (target == null) {
            target = r.build(source, this);
            recordMaping(ruleType, source, target);
            r.setProperties(target,source,this);
        }
        return target;
    }
```

Listing 2.4 Method applyRule of SiTra SimpleTransformerImpl

6. **Build:** This method of the rule-class is responsible for the actual transformation and therefore takes the source object and returns the resulting target object.

7. **Set properties:** Executes further configuration on the resulting target object and also looks for further source objects which are linked with this object. If such a linked object or sub-object was found, the `setProperties`-method triggers a further transformation for this object (like it can be seen in Figure 2.3, where `setProperties` calls the `transform`-method for object `B`). So the transformation process starts again from the beginning but certainly without the rule registration step, which is only carried out once.

Transformation Challenges

Transformation Hierarchies Because a transformation process of an object, which doesn't have any sub-objects or referred objects is very rare, a concept is necessary that detects such sub-objects and transform them too.

If a rule detects a sub-object, which has to be transformed, it executes its own transformation process. Therefore it calls the `transform`-method of the passed transformer object, which chooses again the corresponding rule (see the sequence diagram of the transformation process in Figure 2.3). The advantage of this approach is that the first rule does not need to

know which rule has to be used to transform a sub-object. The rule only calls the `transformer`-interface-method `transform` with the sub-object as source object. The `transformer`-implementation then chooses the right rule. This progress results in a calling chain where a transformer calls a rule, the rule calls the transformer and so on. The problem of this approach, is that this chain could also develop to a non-terminating transformation cycle. How this problem could occur and how to overcome this issue is described in the next Section (see Section 2.1.4).

Avoiding non terminating recursive transformations When transforming source code from one programming language to another, the source model is represented by a tree. Because trees don't have cycles the problem of non-terminating recursive transformation can't occur. Nevertheless this problem should be highlighted shortly because it's a well known problem when transforming a graph-based source model and therefore is addressed by SiTra and will be addressed by jTL too.

Figure 2.4 Simple Cycle Problem of bidirectional source objects

Lets assume a bidirectional relationship between two classes where class `A` has a reference to class `B` and class `B` back to class `A` (see Figure2.4). Here the transformer calls the rule to transform class `A`. This rule notices that there is another source object, which has to be transformed that is class `B`. So it calls the transformer, who calls the rule for transformation of class `B`. When transforming class `B` the problem occurs because rule `B` notices that there is another object to transform. So class `B` again calls the transformer, which prompts the transformation of `A` again. There the infinitive loop starts (see Figure 2.5).

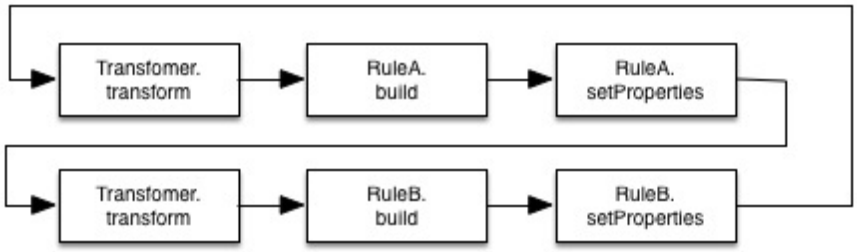

Figure 2.5 S Non-terminating recursive transformation

To overcome this issue the `SimpleTransformerImpl`-class of SiTra holds the following map:

```
Map<Class<? extends Rule<?, ?>>, Map<Object, Object>> mappings;
```

Listing 2.5 Source-Target-Mapping

As shown in Listing 2.4, which represents the `apply`-method of the class `SimpleTransformerImpl`, every time a rule is executed, it has to be checked if the rule was already executed for the specified source object before by calling the method `getExistingTargetFor(sourceObject)`. This method searches in map `mappings` (see Listing 2.5) if the specified source object was already transformed with the specified rule. If a match is found, the already transformed target object is returned. In this case neither the `build`-method nor the `setProperties`-method of the specified rule is executed. The map already returns the target object, without transforming the source object again. So a non-terminating recursive transformation could be avoided with SiTra.

If there is no match in the collection `mapping`, the rule has to be executed for the source object and afterwards the rule, the source-object as well as the target object are stored to the Map `mappings`. This ensures that a source object is not transformed more than once by using the the same rule type. Nevertheless one an the same source objects can be transformed more than once by using different types of rules.

2.2 PC2Java Transformer

2.2.1 Introduction

After discussing SiTra, which introduced a concept and first good guidelines for general transformation issues with Java, the framework should be compared with an implementation of an already existing transformation approach. The so called *PLI/Cobol to Java Transformer* (hereafter in short referred to as *PC2J-Transformer*) is not a framework but a concrete and more specific solution for transformation and focuses on the translation from one programming language to another. Although these two approaches have many things in common, they differ in a few issues, since SiTra is a generic approach and applicable for different kind of model transformations, while the 'PLI/Cobol to Java Transformer' represents a concrete implementation of transformation. So SiTra can be used for transforming a wide range of different kind of model like graph based models with cycles. In contrast the PC2J Transformer can only transform tree based graphs since it was designed to transform source code which is represented by an *Ab-*

stract Syntax Tree (AST). The Abstract Syntax tree is a tree representation of the source code and is called abstract since it not represent every single detail of the source code. Constructs like parentheses, semicolons or dots are not necessary to represent in the AST.

Before the actual transformation process of the PC2J-Transformer, as well as the comparison between SiTra and the PC2J-Transformer is discussed in more detail, the approach to generate the AST out of the underlying source code should be explained briefly. On the one hand the following Section 2.2.2 should give an idea how the AST, which then servers as the source model for the transformation can be generated and on the other hand the following Section should also serves as a demarcation for this work.

2.2.2 Process Model

Since the PC2J-Transformer is a source code transformer we don't get the source model as basis for the transformation for free. There are a few additional steps to overcome until a source model respectively the AST is built. The phase of gaining the source model can be divided into six different tasks, that are meta-model creation, pre-processing, executing a lexer, parsing, optimizing and tree-parsing to generate the final AST. Figure 2.6 shows an overview of the process model which separates the whole process in two different phases. The first phase is the preparation phase which generates the source model represented by an AST, that acts as input for the second phase - the actual transformation phase.

Although the preparation phase is an essential part of the PC2J-Transformer only a short excursus should give an idea of this phase since the core of the work is based on the transformation phase. But to be complete the next Section should give a brief overview about the preparation phase.

2.2.3 Excursus - Generating the Source Model (Preparation Phase)

To generate a source model (that is represented as an AST) as input for the transformer, the underlying source code has to go through certain steps. In PC2J-Transformer the following six steps are obligatory to achieve the source model:

1. Meta-model design
2. Preprocessor
3. Lexer
4. Parser

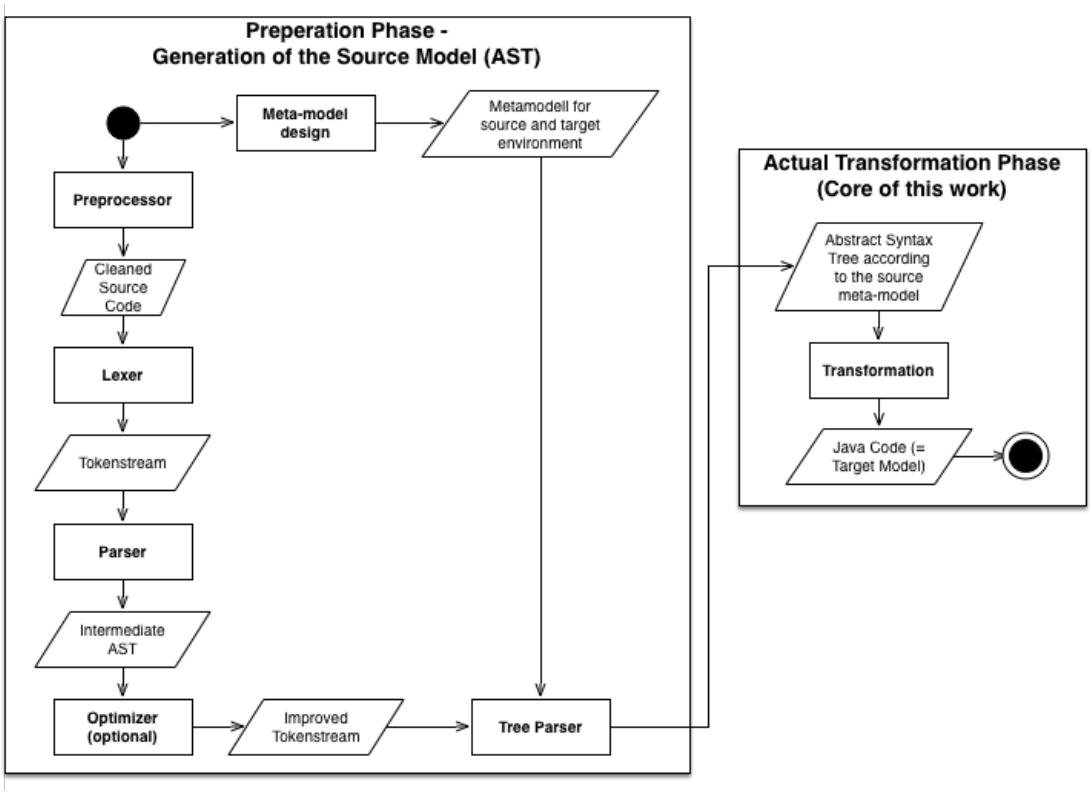

Figure 2.6 Process Model

5. Optimizer

6. Tree parser / Object generator

Meta-model Design

In case of the PC2J-Transformer the source meta-model was designed for PL/I respectively Cobol. The PL/I meta-model alone consists of more than 100 classes. More or less they all were implemented as *POJOs (plain old Java objects)* and act as the framework for the AST Generation of the underlying PL/I source code. The target model is simply the meta-model for the programming language Java. Since the target model of the PC2J-Transformer is more or less Java code there is no need to define a target meta-model based on classes. The use of the library jenesis4Java ensures automatically that the generated source code is valid to the Java meta-model. The first step when working with jenesis4Java is do generate a DOM that represents the Java code. Only when the tree construction phase has finished the DOM is encoded to a Java source code file.

Preprocessor

The preprocessor-task prepares the PL/I or Cobol source code file for the following parsing phase. This simplifies the parsing of the source file. The following list gives an overview which subtask are performed by the preprocessor.

- Remove line numbers: In traditional Cobol and PL/I source code files the first six columns are reserved for the line numbers. This information is not necessary for the transformation to Java code. So the first six characters of each line are removed by the preprocessor.
- Concatenate lines: If literals, expressions or statements are split up into more than one line, this is indicated by a character in the column number seven. So the preprocessor concatenates these lines to one single line because in contrast to the original Cobol or PL/I Compiler the parser of the PC2J-Transformer can naturally deal with lines which are longer than 80 characters.
- Resolve include files: The approach of the PC2J-Transformer is to copy the source code of an include file (in terms of Cobol include files are called copy books) directly into the source file. Although this seems not always to be the most elegant solution, in many cases it is the only possible solution. Since Cobol and PL/I use the include-mechanism also to include source code snippet which can't be executed independently from its surrounding code block. An example would be an include file, which is part of a structure type definition.
- Further tasks: There are a few more special cases where the preprocessor does some important tasks to prepare the source code file to simplify the following parsing phase. Further examples in Cobol are adding 'END-IF'-statements to 'IF'-statements or setting the decimal point to a comma-symbol instead of the dot-symbol, so that the parser could easily differ between the end of a statement and a comma. Also the treatment of some preprocessor directives are necessary before the actual parsing starts.

The result of the preprocessor is a clean source code file which only consists of necessary information for the transformation and unambiguous input for the lexer and parser.

Lexer

PC2J-Transformer uses *ANTLR* [35] as a parser generator. Similar to other technologies like *javacc* or *Coco/R* the basis for the generation of the parser is an *EBNF - Extended Backus-Naur*

context-free grammar. ANTLR is used for the generation of the lexer, the parser (see Section 2.2.3) as well as the tree parser (see Section 2.2.3).
The first phase is called lexical analysis and operates directly on the incoming character stream of the underlying Cobol or PL/I source code file. A lexer grammar written in EBNF is executed to recognize valid words/tokens to generate a stream of tokens which is used as input for the following parser-phase.

Parser

To generate a valid source model in form of an AST the ANTLR-parser validates that input and generates an intermediate AST. In contrast to an *concrete syntax tree*, the abstract syntax tree does not represent every detail of the original syntax. For example, grouping constructs like parentheses or other kind of blocks are expressed implicit. Some other information like dots or semicolons, which emphasize the end of statement are fully ignored by the AST. Furthermore, information which were necessary for the PLI or Cobol compiler but aren't necessary for the Java compiler anymore, are also not depict in the resulting AST. An example for such information would be the length-condition for character strings, since those constructs are transformed into Java string which do not have a length condition. So it is obvious that the PC2J-Transformer already pays attention during the parser-phase which information is necessary for the actual transformation process to Java and what information only leads to an insignificant overhead.
Nevertheless the resulting AST does not only hide unnecessary information, but also adds meaning to sub-nodes by some additional nodes. In ANTLR these additional nodes are called *imaginary tokens*. Especially this concept is unavoidable when the syntax alone is ambiguous and further analysis have to be carried out. For instance in PL/I a Pseudovariablen-assignment has the same syntax than an assignment to an array-field and can't be differentiate by an syntactic analysis alone (see Listing 2.6). So a small semantic analysis, which is also part of the parser, determines the meaning of the statement and adds the gained information as imaginary token to the AST. This is at least the chosen approach of the PC2J-Transformer. A better approach would be a complete separation of syntactic and semantic analysis by performing a pure syntactic analysis in the parser-phase and transfer the semantic analysis to the optimizer-phase, since the optimizer already handle some semantical issues [49].

```
/*Depends on the context whether the following statement is interpreted as a
    pseudovariablen-assignment or an assignment to an array-field.*/
SUBSTR(r,p,l) = v;
```

Listing 2.6 Pseudovariablen-Assignment or Assignment to Arrayfield?

Optimizer

In the optimizer-phase the resulting AST is walked many times in case of the PC2J-Transformer, to prepare the tree for the transformation process. The tasks performed by the optimizer-classes can be multifaceted. Some examples are listed below.

- Refactoring: Removes unused labels, procedures and variables from the AST.
- Resolve 'Gotos': Although Java defines 'goto' as a keyword, it has never been implemented and therefore no goto-concept for Java exists. Furthermore the goto-concept has been considered as harmful programming mechanism since decades [8]. Nevertheless most of the developers had still made use of it and so this concept is unfortunately widespread over old programs which have been written in PL/I or Cobol. The task of resolving 'gotos' automatically is a difficult one. The PC2J-Transformer tries to overcome this issue by building methods instead of using jump-labels. The goto-statement is then replaced by a simple method call. Although that's not always the best solution it is a general concept and applicable for almost every goto-implementation. Other solutions which focus also on the context and the surrounding syntax are discussed in [11].
- Resolve function parameters: In PL/I, parameters which are used in the signature of the function definition have to be declared at the beginning of their body. In Java the declaration is done directly in the signature. According to this principle the optimizer moves up the variable declarations to the header of the function.

The optimizer-phase is probably the most important phase for generating well readable and maintainable Java code and therefore should always considered for further improvements.

Tree Parser / Object Generator

The tree parser is also implemented as an EBNF grammar in ANTLR. The target of this last phase before the actual transformation starts, is, to identify special subtree-structures to generate the appropriate source objects. These source objects then are represented as simple Java instances like if-statement objects or assignment objects. The resulting object structure is then again represented as a tree.

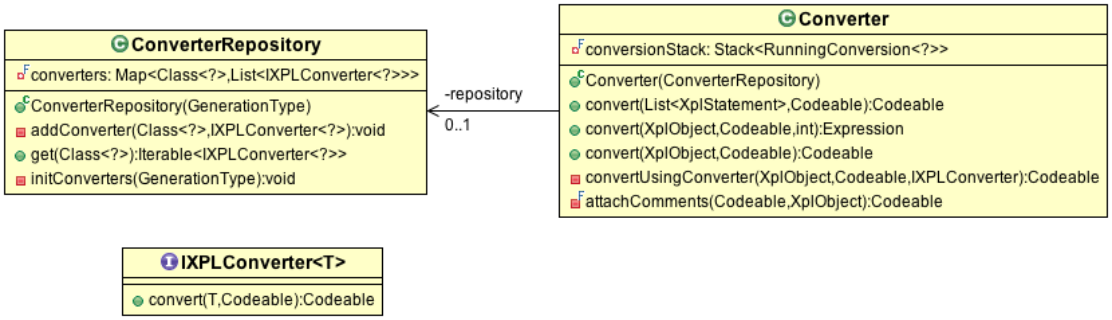

Figure 2.7 Architecture of the PC2J-Transformer core

2.3 SiTra vs. PLI/Cobol Transformer - A Comparison

2.3.1 Overview

Figure 2.7 represents the core of the PC2J-Transformer, which consists of the three items, that are classes `Converter`, `ConverterRepository` and the interface `IXPLConverter`. The following listing briefly explains the domain and tasks of these three items:

1. **Converter:** Looks for the appropriate sub-converter and further executes it to transform the source element to the target element. In SiTra the corresponding interface would be the interface `Transformer` respectively the implementation of it `SimpleTransformerImpl`.

2. **ConverterRepository**: Holds the mapping between the type of the source element and the corresponding Converter, which is responsible for the transformation. In SiTra there is no corresponding construct, since every single rule has to check if it is able to transform the given source object. A concrete mapping between the source element and the rule is not designated in SiTra.

3. **IXPLConverter**: Represents the global interface for all sub-converters. The corresponding SiTra construct would be the interface `Rule`.

For further detailed comparison between SiTra and the PC2J-Transformer, the most important concepts of these two approaches are taken out, to highlight which ideas are equal or similar and what differences can be uncovered. Therefore Table 2.1 gives an first overview.

Conept	SiTra	PLI/Cobol Transformer
Rule	`Rule <S,T>`	`IXPLConverter<T extends XplObject>`
Rule methods	1. `check` 2. `build` 3. `setProperties`	1. `convert`
Transformer interface	`Transformer`	-
Transformer implementation	`SimpleTransformerImpl`	`Converter`
Rule Repository	List of all registered Rules in `SimpleTranformerImpl`	Own class `ConverterRepository` - holds a map which maps known source object types to applicable converters

Table 2.1 Overview - corresponding concepts of SiTra and PLI/Cobol Converter

2.3.2 Sitra's Rule Interface vs. P2J-Transformer's IXPLConverter Interface

As described earlier the `Rule`-interface of SiTra defines the general structure how a source object should be mapped to a target object. This interface has to be inherited by every rule implementation. Further the implementation class of this interface is responsible for the actual transformation of the source element. In case of the P2J-Transformer the corresponding interface to the `Rule`-interface is called `IXPLConverter`.

Listing 2.7 shows how this `Rule`-interface look like in SiTra and Listing 2.8 shows the corresponding Rule-interface, named `IXPLConverter`, in the PLI/Cobol Transformer.

```
public interface Rule<S,T> {
    boolean check(S source);
    T build(S source, Transformer t);
    void setProperties(T target, S source, Transformer t);
}
```

Listing 2.7 Rule Interface (SiTra)

```
public interface IXPLConverter<T extends XplObject> {
    Codeable convert(T xplObject, Codeable parent);
}
```

Listing 2.8 Rule Interface (PLI/Cobol Transformer)

Because SiTra is a general framework, which should be applicable to all possible transformation, the interface defines the source object as well as the target object as generics, so that the target

object type can be different for every single implementation. This can be seen in the signature of the interface definition where `S` represents the generic source object type and `T` represents the generic target object type.

Similar to SiTra the type of the source object is also defined as generic in the corresponding`IXPLConverter` and represented by subtype of the abstract superclass `XplObject`. So every single element of the source model (Abstract Syntax Tree) is a subtype of this class and can be passed to this interface.
In contrast to SiTra the target object isn't specified as generic in the PLI/Cobol Transformer interface `IXPLConverter` because the return type is always from the same type, that is `Codeable`. `Codeable` is an interface of the library *jenesis4java*, and can represent all possible Java source code constructs, such as classes, methods, statements or expressions. Because jenesis4java is a *domain object model (DOM)* of the Java programming language, the target model is again represented by a tree structure.

Excursus - Jenesis4Java [42]

Jenesis4Java is an open source library and was designed for generating Java Code out of a DOM. So a complete Java class including annotations and comments is constructed as a DOM that can be manipulated with many different easy-to-use methods till it is encoded and written out to a Java file. A class called `VirtualMachine` acts as factory for the root of the resulting tree. This root is represented by the object type `CompilationUnit` which is already a subtype of the interface `Codeable`. Outgoing from the root `CompilationUnit`, children like package definition, imports, classes, type definitions, control structures, etc. can be generated which again are subtypes of the interface `Codeable`.

2.3.3 Rule Interface - Methods

According to Listing 2.7 in SiTra the rule interface distinguishes between prerequisite tests, which take place in the `check`-method, the actual transformation in the `build` method and configuration to the target object in the `setProperties`-method as well as further transformation of sub-objects or linked objects, while the PLI/Cobol transformer combines all these activities in only one Method called `convert`.

In SiTra the `check`-method is unavoidable because before the actual transformation takes place by calling the build method, SiTra has to check if the source object is applicable to the specific rule. In case of the PLI/Cobol Transformer this step can be left out because the transformation class `Converter` already ensures that the right rule type is called (see Section 2.3.4).

An important aspect of the interface `Rule` of SiTra is that the actual transformation of a source object is carried out in a separate method (`build`-method), while further transformation of referenced- or sub-object takes place in the `setProperties`-method. For a general framework like SiTra this is a must to avoid the non-terminating recursive transformation problem (see Section 2.1.4). Because the PC2J-Transformer transforms only source code which is represented by an Abstract Syntax Tree (AST), this problem can not occur, since a tree do no contain cycles. So the separation into two methods isn't necessary and simplifies the interface further.

Rule Interface - Implementation

Listing 2.9 shows exemplarily how the framework of an implementation of the `Converter`-interface for a transformation of an if-statement can look like. The Listing emphasizes, how the `convert`-method takes an instance of the class`XplIfStatememt`, which is a sub class of general type `XplObject`, as source object and how it generates an object of the jenesis4Java class `If`, which is a sub-object of `Codeable`. This object represents then the resulting target object.

```
public class XplIfConverter implements IXPLConverter<XplIfStatement> {
    @Override
    public Codeable convert(XplIfStatement ifStatement, Codeable parent) {
        If resultingIfStat = ((Block)parent).newIf(/*<Expression>*/);
        //Actual transformation-code
        return resultingIfStat;
    }
}
```

Listing 2.9 Rule Interface Implementation

2.3.4 Transformer

The core of both transformation approaches is the class which chooses the applicable rule for the transformation of the source object to the target object. In SiTra this class is called `SimpleTransformerImpl` which implements the interface `Transformer`. In the PC2J-Transformer an interface doesn't exist and so this issue is covered only by the class called `Converter`. Since the SiTra `Transformer`-interface was introduced in Section 2.1.3, this Section covers the differences to the `Converter`-class of the PLI/Cobol Transformer.
The major difference is the way the `Transformer` in SiTra and the `Converter` in PC2J-Transformer holds their rule repository. This is described in the next Section (2.3.4).

Rule Repository

In SiTra the rule repository is represented by a simple list in the `SimpleTransformerImpl`, which only holds all registered rule-classes (see Section 2.1.3). The disadvantage of this approach is that there is no mapping between source types and the corresponding rule types. So if the developer doesn't provide any information which rule should be called for transforming the source object, SiTra assumes that every rule is a candidate. As shown in Listing 2.3 the method `getRules` will then return every rule of the repository. So the list of rules must be looped again to call the `check`-method for every possible rule to determine wether the rule can transform the type of the source object or not. The first matching rule is then used for the transformation of the source object.

The PLI/Cobol Transformer chooses a different approach. A separate class called `ConverterRepository` holds not only a list of all registered rules but a collection `Map`, which maps source object types to one or more possible converters (in context of Sitra converters are called rules). The following code snippet in Listing 2.10 shows the declaration of the map and an example of adding a tuple consisting of the source object type `XplIfStatement.class` and the corresponding converter `XplIfStatementConverter` to the map.

```
public class ConverterRepository {

    private final Map<Class<?>, List<IXPLConverter<?>>> converters =
               new HashMap<Class<?>, List<IXPLConverter<?>>>();

    private void addConverter(Class<?> classToConvert, IXPLConverter<?> converterObject
        ) {
        List<IXPLConverter<?>> converterList = converters.get(classToConvert);
        if (converterList == null) {
            converterList = new ArrayList<IXPLConverter<?>>();
            this.converters.put(classToConvert, converterList);
        }
    }

    public Iterable<IXPLConverter<?>> get(Class<?> expressionClass) {
        return this.converters.get(expressionClass);
    }

    private void initConverters(GenerationType generationType) {
        addConverter(XplIfStatement.class, new XplIfConverter());
    //Adding all the other source object types with their corresponding converters to
        the map...
}
```

Listing 2.10 ConverterRepository of the PLI/Cobol Transformer

The method `addConverter` adds a tuple consisting of the source object type and the corresponding converter to the converter-repository. So before the actual transformation starts the whole converter-repository map has to be initialized.

Further the method `get` in the `ConverterRepository` returns the sub-converter according to the passed source object type.

Choosing the right Rule/Converter

In Section 2.1.4 the SiTra's process of choosing the corresponding rule to the specified source object, which was carried out in the class `SimpleTransformImpl`, was already explained. Based on the slightly different rule repository approach of the PC2J-Transformer, the `Converter`-class (in terms of SiTra this class is called `Transformer`) has a slightly different implementation. The source code Listing 2.11 shows the most important parts of the `Converter`-class.

Description of the `Converter`-class

(1) `conversionStack`: stores the call hierarchy of the sub converters (known as rules in SiTra)

(2) `ConverterRepository`: holds the mapping between source object types and applicable converters

(3) `convert`-method: the corresponding method to SiTra's `transform` method. The `xplObject`-parameter represents the current source object while the `parent`-parameter represents the surrounding code block as object.

(4) Gets the type of the source object.

(5) Iteration over the whole `ConverterRepository` to find the corresponding sub-converter to the type of the source object

(6) After finding the right sub-converter the method `convertUsingConverter` (7) is called

(7) `convertUsingConverter`: pushes the Converter to the stack and executes the sub converter.

(8) If no converter was found for the type of the source object, the super class type of the source object is gained. For that super class the `convert` method tries to find a corresponding converter.

To summarize, the significant difference between SiTra and the PLI/Cobol Converter is the rule-repository and the way, how to select the corresponding rule. The rule-repository in SiTra only

```
public class Converter {

    private final Stack<RunningConversion<?>> conversionStack = new Stack<
        RunningConversion<?>>(); //(1)

    private final ConverterRepository repository; //(2)

    public Codeable convert(XplObject xplObject, Codeable parent) { //(3)
        if (xplObject != null) {
            Class<?> expressionClass = xplObject.getClass(); (4)
            while (expressionClass != null) {
                Iterable<IXPLConverter<?>> converterList = this.repository.get(
                    expressionClass); //(5)
                if (converterList != null) {
                    for (IXPLConverter<?> converter : converterList) { //(5)
                        Codeable result = convertUsingConverter(xplObject, parent,
                            converter); //(6)
                        if (result != null) {
                            return result;
                        }
                    }
                }
                expressionClass = expressionClass.getSuperclass();//(8)
            }
        }
        return null;
    }

    private Codeable convertUsingConverter(XplObject xplObject, Codeable parent,
        IXPLConverter converter) { // (7)
        if (converter != null && !converter.equals(Object.class)) {
            this.conversionStack.push(new RunningConversion(xplObject, converter,
                parent));
            try {
                return converter.convert(xplObject, parent);
            } finally {
                this.conversionStack.pop();
            }
        }
        return null;
    }
}
```

Listing 2.11 Converter Class of the PLI/Cobol Transformer

consists of all registered rules and so an additional step, namely the `check`-method for every possible rule, has to be executed, when searching for the corresponding rule to transform the source object. In PLI/Cobol Transformer this extra step can be left out because of the fact that the rule repository already stores the concrete mapping between the source object and the corresponding converter.

Another difference is that the architecture of the PC2J-Converter was built to transform abstract syntax trees. For transforming graphs with cycles this approach is not applicable, since the attempt will result in a non-terminate recursive transformation.

2.3.5 Transformation Example

In this Section a small PL/1 source code snippet acts as example (see Figure 2.8) and should demonstrate a basic transformation process for a simple if-statement in PLI using the PC2J-Transformer. Therefore the excerpt of the PL/1 meta model is used, that was presented in the introduction of this work (see Figure 1.2 in Section 1.6). The PC2J-Transformer uses simple Java classes (POJOs) as representation technique of this meta model, while instances of these classes act as the actual source elements which have to be transformed and together build the source model as AST.

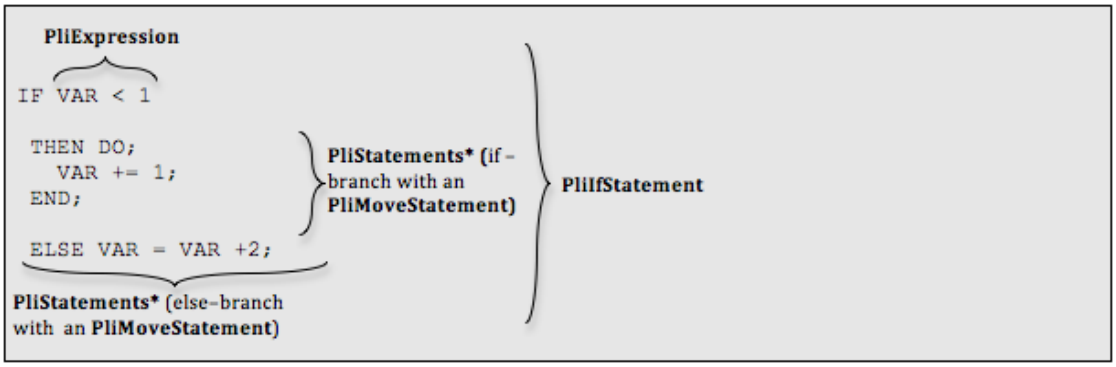

Figure 2.8 Source Code of a simple PLI if-statement

After the meta model has been designed the next step covers the extraction of the source object out of the plain PLI source code. The easiest way to do this is by writing a grammar to construct an abstract syntax tree out of the source code. Figure 2.8 shows an if-statement written in PL/I. The braces emphasizes roughly the source objects - corresponding to the meta model - resulting from an extraction process. The simplified corresponding AST and thus the tree which is the actual input for the transformer is shown in Figure 2.9.

Converter - Actual Transformation

Listing 2.12 shows the Listing of the specific converter PliIfConverter, which is responsible to generate a Java if-statement. To focus on the key point of this class some non-significant issues were shorten. For example the instance of the general `Converter`-class (see Listing 2.11) is used directly in the following Listing, although there is no corresponding declaration respectively initialization for it. In the original code every sub converter would have an instance of a so-called context-object, which holds an object of the main converter class. By calling for the global `Converter`-instance the sub converter (in this example the PliIfConverter) is able to launch further sub converters. Holding the global instance of the class `Converter` in a public

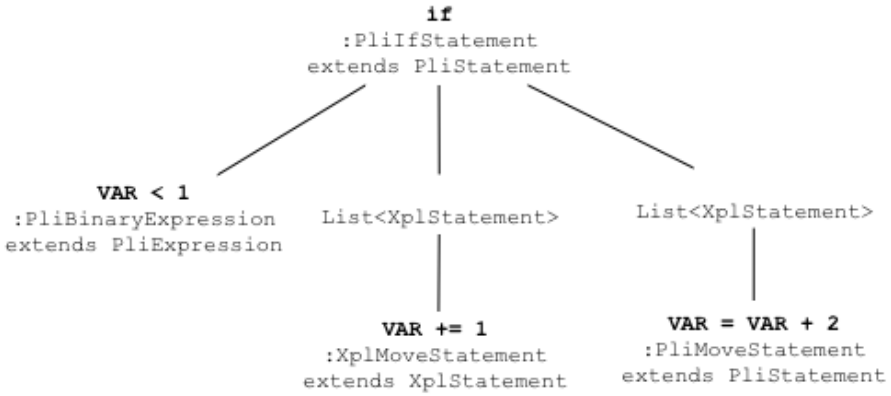

Figure 2.9 AST of a simple PLI if-statement

context, saves the developer from passing the instance to every single sub-converter like it is necessary in SiTra. The following enumeration emphasizes the most important statements of Listing 2.12

```
public class PliIfConverter implements IXPLConverter<PliIfStatement> { //(1)

    @Override
    public Codeable convert(PliIfStatement ifStatement, Codeable parent) { //2)
        If ifStmt; //(3)
        final Block block = (Block) parent;

        Expression expression = (Expression) converter.convert(ifStatement.getCondition
            (), parent); //(4)
        ifStmt = block.newIf(expression); //(5)

        converter.convert(ifStatement.getIfStatements(), ifStmt);//(6)
        converter.convert(ifStatement.getElseStatements(), ifStmt.getElse());//(6)
        return ifStmt; (7)
    }
}
```

Listing 2.12 Simple Converter Example for an if-statement

(1) Every sub-converter has to implement the general `IXPLConverter`-interface, which consists only of the method `convert`. This is similar to SiTra where every rule has to implement the interface `Rule`.

(2) The `convert`-method holds the actual transformation process. As arguments a subtype of PliObject has to be passed, which represents the source element, as well as the surrounding Java code block of already transformed source code.

(3) Instance which will hold the resulting Java if-statement.

(4) Transformation of the expression which further will represent the if-condition.

(5) Generating the if-statement with the previous transformed if-condition in the surrounding code-block.

(6) Calling the additional sub-converters to transform further statement of the two branches of the if-statement. If all sub-statements have been generated, the transformation of the if statement is finished.

(7) According to the library jenesis4Java the instance of class `If` is a `Codeable` and can be returned to the surrounding converter.

CHAPTER 3

Java 8 Features for Transformations

3.1 Overview

Java 8 was recently released on March 18, 2014 with some very useful new features. The most important ones are listed below

1. *Functional programming*
2. *Lambda expression*
3. *Bulk operation and streams*
4. *Default methods*
5. *Method references*
6. *Optional<T>*
7. *Parallel operation on arrays, etc.*
8. Date And Time API
9. Java FX

Further the focus lies on the first seven points, since these concepts could be used for transformation issues and result in different advantages which also have a high relevance when transforming issues have to carry out.

In [48], Richard Warburton states the needs as well as the advantage of Java 8. In the last few years the rise of multicore CPUs was an important development. With this trend there is also a need for programming language to support this behaviour. Java 1.0 already supports threads and locks as a way to deal with concurrency, which was the best practice at the time. Nevertheless this concept forced the developer to write a lot of boilerplate code, which often leads to an avoiding of this concept. Java 5 then invented the concept of thread pools and concurrent collection (package `java.util.concurrent`), while Java 7 added the fork/join mechanism, making parallelism more practical but still difficult [46]. Although the fork/join mechanism helps to write code that performs well on multicore CPUs with an much easier approach than by implementing the same tasks with threads, it still has not been used frequently. This results in the fact, that time consuming iteration issues over large collection were carried out in a sequential way. So there was a strong need for a concept to easily implement bulk operation in a parallel way on collection [48] .

Java 8 supports parallel bulk operation for collections in a compact, user-friendly and easy to read way. Although there are some new concepts to learn like lambda expressions, streams, bulk operation and method references this kind of parallel programming is much easier than to handwrite a large quantity of complex thread-safe source code.

Another advantage is that the resulting code of an iteration over a collection is shorter and easier to read. There is no need for defining iterators (`java.util.Iterator`) and implementing loops by hand. So the focus lies on the business logic without defining much overhead code. This all results into source code which is easier to read and maintain and also code which is more reliable and less error-prone. To support these new features of Java 8 appropriately, the concept of *functional programming* was improved, to a more user friendly way.

Transformation projects are often very complex and also performance consuming jobs. Therefore transformation issues can benefit from the new Java 8 features. Before Section 3.3 discusses in which manner lambda expressions, bulk operations and method references can support the transformation process and how the transformation process benefits from these new concepts, a short introduction to these concepts should be given.

3.2 Functional Programming

With Java 8 a new programming paradigm to Java was invented, which already was in use in other object-oriented like C#, Groovy or Scala. With this new feature it is possible to pass code (behaviour) to an API, return it or even store it to a data structure. In programming language there is a distinction between first-class values and second-class values. A first-class value (also

called citizen, object or entity) is an entity which supports all the operations generally available to other entities. These operations typically include being passed as a parameter, returned from a function, and assigned to a variable. [39]. Second-class values like functions, methods or classes which describe the behaviour of the application or express the structure of first-class values, can't be passed to around during the program execution. [47] According to this definition in traditional Java, first-class values are primitive values as well as concrete objects, while second-class values are methods and classes.

Since Java 8 there is an easy way to pass around methods at runtime, and hence making them to first-class values (see Lamda Expression in Section 3.3). Experiments in other languages like Groovy or Scala determine the high relevance of this concept and how it can support the daily programming routines. [46] Although previous Java versions support behaviour parameterization by making use of anonymous inner class, this approach showed increased code conciseness. To demonstrate the benefit of lambdas, as implementation technique for functional programming, a simple example is given, which only sorts a list of Integer. Listing 3.1 shows the implementation with the old approach of an anonymous inner class, while Listing 3.2 shows a solution by using functional programming with lambda expressions

```
        Collections.sort(numberList, new Comparator<String>(){
              public int compare(String a, String b){
                return a.compareTo(b);
              }
            });
```

Listing 3.1 Sort by using anonymous inner class

```
Collections.sort(numberList, (a,b)-> {return a.compareTo(b);});
```

Listing 3.2 Sort by using functional programming with Lambda Expression

The benefits of the new functional programming approach of Java 8 are obvious, namely less and better readable source code. The next Section gives a brief overview about the Java specific implementation of functional programming with the help of lambda expressions.

3.3 Lamda Expressions

> 'A lambda expression can be understood as a concise representation of an anonymous function that can be passed around: it doesn't have a name, but it has a list of parameters and a body.' [48]

According to this definition the general structure of a lambda expression is:

(parameter-list) -> {single expression or statement block}

The following Listing (3.3) shows three different implementation techniques of lambda expressions which further are explained in more detail [33].

```
//Parameter with type definition and a statement block
(int x, int y) -> {return x + y;}       //(1)

//Parameter without type definition and an expression
(x, y) ->  x+y;                 //(2)
a -> a+1;                   //(3)
```

Listing 3.3 Lamda Expression Examples

(1) The use of more than one parameter requires to enclose the parameter elements with parenthesis. If the function consists of more than one expression or of one or more statements the braces are obligatory.

(2) The specification of the parameter types is optional and could be omitted since the type could be derived from the functional interface (see Section 3.3.1). If the body of the lambda expression consists only of one expression, the braces can also be omitted. In this case the expression is evaluated and the result is returned automatically without the need of using the return statement explicitly.

(3) If the parameter list consists of only one parameter the parentheses are optional.

After clarifying what benefits the use of lambda expressions offers and how the syntax looks like the question of 'Where lambda expression could be used' rises which is answered in the next Section 3.3.1.

3.3.1 Functional Interfaces

Lambda expression can be used in the context of a functional interface. The term of a functional interface is new in Java since version 8. It represents an interface which consists of exactly one abstract method. These types of interfaces are also called *SAM type - Single Abstract Method type* and aren't really new to Java, although there was no well known term for it. Examples for SAMs (resp. functional interfaces) in former Java versions are `Runnable`, `EventHandler` or

`ActionListener`. They all have in common that the implementation of these interfaces were often carried out with the help of anonymous inner classes [16]. The reason for that approach was, that an implementation of these interfaces in an own class file, which only consist of one single method, would be an overwhelming approach. As mentioned earlier the use of anonymous inner class was the only way to pass functionality to such a functional interface but since Java 8 lambda expressions provide a much more comfortable way to do this. So the direct connection between functional interface and lambda expression is that lambda expression provides an easy way to pass an implementation of method of a functional interface.

Java 8 provides a list of about 40 predefined functional interfaces in the package `java.util.function`, which should protect the developer from defining trivial interface over and over again. The Table 3.1 shows some examples of useful functional interfaces of the package `java.util.function`.

Interface	Method	Usage
`Consumer <T>`	`void accept(T)`	Represents an operation that accepts a single input argument and returns no result.
`Supplier <T>`	`void get(T)`	Represents a supplier of results (for example to generate objects)
`Predicate <T>`	`boolean test(T)`	Represents a predicate (boolean-valued function) of one argument
`Function<T,R>`	`R apply(T)`	Represents a function that accepts one argument and produces a result. This further represents the transformation concept.

Table 3.1 Sample of Functional Interfaces - taken out of the Java API [34]

3.3.2 Type Inference

The concept of type inference isn't new to Java, since it was already used in Java 7 in context of generics and the diamond operator. Listing 3.4 shows an example for type inference in Java 7 where the diamond operator asks the Java compiler (javac) to infer the generic arguments from the `List`-declaration. [48]

```
List<Integer>numberList =  new ArrayList<>();
```

Listing 3.4 Type inference in Java 7

Java 8 goes one step further and allows to leave the types for all parameters of a lambda expression. Listing 3.5 demonstrates how lambda expressions are used without specifying the optional type. There is no need to specify the type for the parameter `number`, because it can derived from the generic type of the functional interface `Predicate <Integer> predicate`.

```
public static void condition(List<Integer> list, Predicate<Integer> predicate) {
    //testSomething
}

public void typeinference(List<Integer> numberList) {
    //Without type definition
    condition(numberList, number -> number%2==0);
    //With type definition
    condition(numberList, (Integer number) -> number%2==0);
}
```

Listing 3.5 Type Inference with Lambda Expression

3.3.3 Default Methods

Default methods are a brand new concept of Java 8, which allow to add a method with implementation to an interface by declaring it with the keyword *default*. That feature offers the opportunity to make changes to already existing interfaces without loosing backward-compatibility. Before inventing default methods in Java 8, an adding of a method to an interface results in the need, of reimplementing all classes which inherit from this interface. Because the introduction of lambda expressions needs changes to already existing interfaces in the JDK, the introducing of the concept 'default methods' was more or less a must.

An example for a standard interface of the JDK is `java.util.Collection`. This interface was already invented with Java version 1.2 [31]. Until Java 7 the signature of this interface didn't change and so it was no problem to upgrade the Java version for applications which implement this interface. However in Java 8 there was a strong need to upgrade the interface `java.util.Collection`, to supply new methods for using streams like `stream()` and `parallelstream()` or to support the use of lambda expression like the method `removeIf(Predicate<? super E> filter)` does. A change to the interface in a traditional way, would have led to backward-compatibility problem, since all existing Java applications which implement this interface would have to be refactored to implement these new methods. The answer to overcome this problem was to invent the concept of default methods. On the one hand backward-compatibility was ensured and on the other new methods including their implementation could have been added to the `Collection`-interface.

3.3.4 Method References

Method References are also a new concept in Java 8 to achieve an even better readability of the source code. For lambda expressions which only consist of one method call, a method reference can be used as shorter and more compact equivalent. Listing 3.6 contrasts these two approaches. The syntax of a method reference in general is:

<class>::<methodname>

```
List<String>numberList =  Arrays.asList(new String[] {"6","2","7"});
//Lamda Expression
numberList.stream().forEach(number -> System.out.println(number));

//Equivalent Method Referencee
numberList.stream().forEach(System.out::println);
```

Listing 3.6 Method Reference Example

As demonstrated in Listing 3.6 the parameter specification can be omitted, since the compiler determines the information and passes them automatically when the method is called. This automatically derivation of the parameter works only if the signature of the called method is compatible with the used functional interface. [19]

Another concept, which can be seen in the previous Listing 3.6, is the use of the `Collection.stream()`-method which is another big improvement of Java 8 and therefore is described in the next Section 3.4.

3.4 Streams and Bulk Operations

Although `Collections` is the most heavily used API in Java, the way how to deal with collections was neglected and often results in writing nearly the same Code again and again. The focus when implementing manipulation on collection lies rather on the way how to manipulate them, than on the actual task (see Section 3.4.1). Before Java 8 was released, the poor support of manipulating collections was a well known problem and so there have been a few attempts to provide Java programmers with better libraries to overcome this problem. Guava, Apache Commons Collection library and lambda are examples for such libraries. But finally Java 8 fulfills the need for a better manipulation of collections, by inventing the concept of streams (`java.util.Stream`) and comes up with its own official library to deal with collections in a more declarative way [46]. Instead of operating on the collection directly, the collection is converted to a stream. This enables many new opportunities and advantages to the developer:

- A declarative approach to work with collections
- An easier and shorter source code
- An easy way for data parallelism (see Section 3.4.3)

3.4.1 External vs. Internal Iteration

The traditional approach in Java when working with collections was to iterate over a whole collection to execute the desired operation on every single element. This approach is also known as external iteration. But there are several problems which come with this approach. Boilerplate code was needed to be written to iterate over all elements of the collection.

Another problem with this approach is, that it was hard to implement parallelism, so that operations on the elements of the collection could exploit the power of multi-core processors and executes them in a parallel way. Even more boilerplate would have been necessary, to fulfill the ability of making use of parallel execution on collections. So, in nearly every Java application, collections were used in a sequential way to overcome the problem of high implementing effort and worse readable source code.

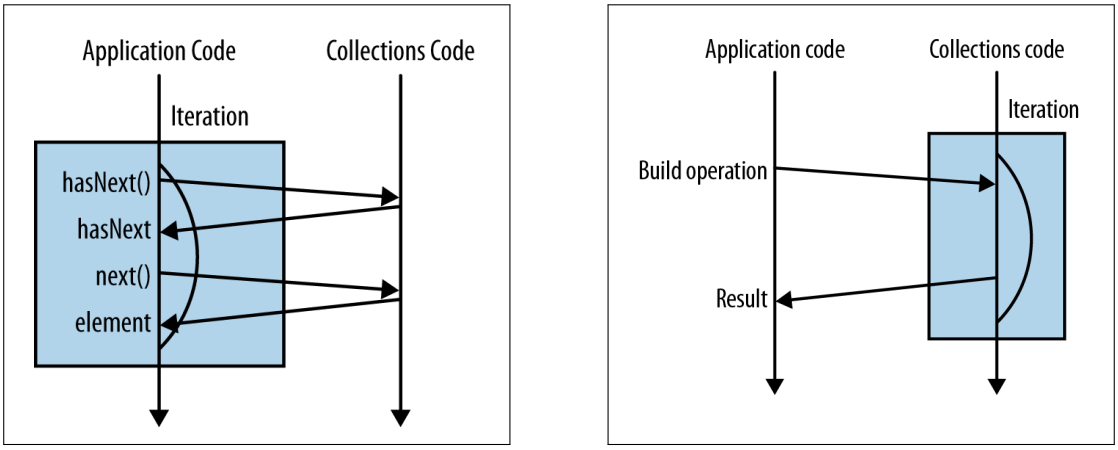

Figure 3.1 External vs. Internal Iteration [48]

With Java 8 an alternative approach, called internal Iteration, comes up and makes life much easier when dealing with collection, even in a parallel way. The basic idea of this concept is to use streams and pass functionality to internal iteration. Figures 3.1 shows the difference between external and internal iteration. While external iteration has to deal with operation like creating an iterator and boilerplate methods like `hasNext()` or `next()`, the internal iteration only focuses on the actual operations which has do be performed on the underlying collection.

These two approaches should be demonstrated by using an example from the context of transformation (see Listing 3.7 and 3.8). The source code searches for tokens which are from the type 'VARREF' (Variable Reference) and transfers the matches to a resulting list.

Although Listing 3.7 could have used the for-each loop instead, which only acts as syntactical sugar, the differences of these two listings and the advantages of the internal iteration are

```
for (Iterator<E> i= tokenList.iterator(); i.hasNext();) {
    Token nextToken = i.next();
    if (nextToken.isTypeOf(TokenTypes.VARREF))
        resultingList.add(nextToken);
}
```

Listing 3.7 External Iteration

```
resultingList = numberList.stream().filter(token -> token.isTypeOf(TokenTypes.VARREF)).
    collect(toList());
```

Listing 3.8 Internal Iteration

obvious. The internal iteration is much shorter, better readable and focuses only on the actual behaviour (see Section 3.4.2). Further it opens the opportunity to parallelism by only replacing the method `stream()` with the method `parallelstream()` (see Section 3.4.3).

3.4.2 Using Streams

A data stream transports data from a source, for example a collection, through a pipeline. A pipeline exists of a sequence of stream-methods. In general these methods use lambda expressions as parameter to pass the actual functionality to the internal iteration. The elements are not stored in the pipeline, but be passed from one method to another [2]. This results in the fact, that a stream can only be traversed/consumed once (similar to iterators). So if the stream has to be traversed more than once the first operation on the stream has to return a stream again, to support further operation. As shown in Figure 3.2, working with streams consists of three steps, while the second step is optional.

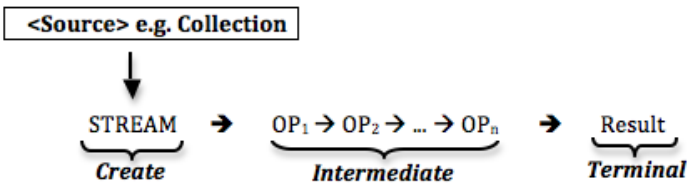

Figure 3.2 Pipeline for Stream Processing

The new interface `java.util.Stream` acts as implementation of this concept, and supports that chaining mechanism of methods to execute function on collection in data streams. In the following the implementation of these three steps in Java should briefly be discussed.

1. Create Stream

This operation supports the creation of a stream out of a collection or array. The methods `stream()` for collections and arrays and `parallelStream()` for collections only act as such creation methods. The result of this methods is a stream from the source type of the collection, for example `Stream<String>`, if the source collection was from the type String.

2. Intermediate Operation

In contrast to terminal operation intermediate operations have in common that they can be connected in series. As mentioned earlier streams only can be processed once, so in case of intermediate operation, after the operation was carried out, a new stream is returned. This enables a chaining mechanism to execute several methods in a row.

Many of the intermediate methods in Java are so called *higher-order functions*. Higher-order functions are methods which take another function as an argument [48]. In case of Java such functions are recognizable by a functional interface defined as parameter. So higher-order functions take behaviour as input.

Intermediate Operation can be categorized into three different fields [16]:

1. *Stateless*: Stateless operation executes the action on each elements of the stream independently on each element. So the result of an operation on an element does not depend on the result of another one. Therefore the stateless operation can be parallelized excellently.

2. *Stateful*: In contrast to stateless operation, stateful operation on single elements of the streams requires the knowledge of the other elements or at least a subset of elements of the same stream. Examples for such operations are the `distinct` or `sorted`.

3. *Short-circuiting*: Short-circuiting operation often doesn't have to be carried out on all elements of the stream. They stop as soon as they fulfill their operation like in case of the method `findFirst()`. That's the reason why short-circuiting operations have advantages in performance, especially when they are carried out in a parallel way. Short-circuiting operation can be stateless and stateful intermediate operation as well as terminal operation.

3. Terminal Operation

Terminal operation returns a result from a stream pipeline. This result is a non stream value such as any kind of collection, primitive types or even void. So a terminal operation concludes

the stream operations and no further operation on the stream is possible. On the other hand, a terminal operations is obligatory, since intermediate operations don't perform any processing until a terminal operation is invoked on the stream pipeline. So the intermediate operation can be seen as lazy methods. [32].

3.4.3 Parallel Data Processing with Streams

In contrast to collection or arrays, streams do not cache data (or in case of stateful operations - only few data) and so they consume significantly less memory space. Therefore the construction of streams commonly results in a lower memory demand and execution time than using collections. In Java 8, the possibility to achieve even better performance by working on stream in a parallel way, can be implanted in an easy user-friendly way. The only thing to do is to use the creation method `parallelStream()` instead of `stream()`. For arrays the combination of the `stream()`-method and the `parallel()`-method is necessary since the method `parallelStream()` doesn't exist (see Listing 3.9).

```
//for Collections
Stream <String> = myStringCollection.parallelStream();
//for Arrays
Stream <String> = Array.stream(myStringArray).parallel();
```

Listing 3.9 Use of parallel Streams

Before Java 8 comes up, the way to go to achieve parallel data processing, was a hard one. Roughly it consists of the following steps [9]

1. Splitting the data into subparts
2. Assigning each subpart to different threads
3. Synchronize them to avoid unwanted race condition. Further wait for the completion of all threads and combine them again.

This complex procedure leads to a situation, that collections were processed nearly solely in a sequential way. Although the *Fork/Join Framework* of Java 7 leads to a better usability of parallel data processing, it also comes up with a lot of boilerplate code. So there was a strong need for an easy way of parallelism in the context of collection. Java 8 has given the answer by inventing the concept of parallel stream processing. In fact the methods `parallelStream()` and `parallel()` internally make use of the Java Fork/Join mechanism but

hiding it from the developer, so that the user do not have to deal with writing complex boiler-plate code. Behind the scenes the parallel-methods determine the number of processor by calling `Runtime.getRuntime().availableProcessors()`. According to the result of this method the specific number of threads is created. [46]

Ordering Problem when using parallel Stream

When using the `forEach()`-method in combination with a parallel stream, it's important to notice that the operation of the `forEach()`-method is not executed in an ordered way. Since stream operations use internal iteration when processing elements of a stream, the Java compiler and runtime determine the order in which to process the stream's elements to maximize the benefits of parallel computing unless otherwise specified by the stream operation. Listing 3.10 shows the behaviour of sequential stream and parallel stream in combination with `forEach()`, while the following enumeration serves as explanation to this Listing.

```
public void testOrderOfParallelProcessing (String []args) {

    Integer[] intArray = {1, 2, 3, 4, 5, 6, 7, 8 };

/*(1)*/ Arrays.stream(intArray).forEach(System.out::print);
    // Output--> 12345678

/*(2)*/ Arrays.stream(intArray).parallel().forEach(System.out::print);
    // Output --> 68572314

/*(3)*/ Arrays.stream(intArray).parallel().forEachOrdered(System.out::print);
    // Output --> 12345678

/*(4)*/ List<Integer> list= Arrays.stream(intArray).parallel().map(s -> s+1).collect(
    Collectors.toList());
    list.forEach(System.out::print);
    // Output --> 1020304050607080

/*(5)*/ Stream<Integer> stream = Arrays.stream(intArray).parallel().map(s -> s+1);
            stream.sequential().forEach(System.out::print);
    // Output --> 1020304050607080
    }
```

Listing 3.10 Ordering Problem in parallel mode

(1) Sequential stream processing with `forEach()` –> the result remains in the same order

(2) Parallel stream processing with `forEach()` –> the result appears in a random order

(3) Parallel stream processing with `forEachOrdered()` –> parallel processing but the <result appears in the same order like the starting stream

(4) Parallel stream processing with `map()`, `collect()` and `forEach()` –> after executing the intermediate operation `map()` and the terminal operation `collect()`, the elements in the resulting collection are again in the right order.

(5) Combination of parallel and sequential stream processing –> the method `map()` is executed on a parallel stream, but for the output with the method`forEach` the stream is switched to a sequential one. This again leads to an ordered output result. Nevertheless the use of switching a parallel stream to a sequential one should be avoided if possible, since the performance benefit of the parallel execution will be shot down again.

Parallel Data Processing - Field of Application

Although the implementation of parallel data processing isn't a difficult task anymore, the use of it is not always profitable. The main reason is that the processing time isn't always lower when implementing data processing parallelly. According to small examples in [48], [46] and experiments in the context of this work (see Section 3.4.4), some situation can lead even to a much higher execution time when using parallel streams, than using them in a sequential way (see Section 3.4.4). It has to be noted that parallel streams have a much higher overhead compared to sequential ones. Coordinating, splitting and combining are mandatory and time consuming jobs when dealing with parallel streams. But these are not the only reasons why parallel stream processing could behave worse than sequential ones. So the question is: when to use parallel data processing, and when to use streams with sequential access or even use external iteration? The answer to that question is not as simple as it seems at first sight, because there are many factors which influence the execution time and further the decision if it's worth to use parallel streams. Nevertheless the following list should give some indicators and a basis for decision-making [46] [21].

1. **CPUs**: The fundamental requirement to benefit from parallel stream processing is a CPU with several cores or even a computer with multiple CPUs. If this requirement is not fulfilled, the use of parallel streams would be nonsense since the execution time would even rise.

2. **Amount of data (N - Number of elements)**: The amount of data, which has to be processed, is an important indicator for determining whether the use of parallel streams is profitable or not. For small amount of data the use of parallel stream is almost never a winning decision. The advantages of parallel processing of only a few elements aren't enough to compensate the additional cost introduced by the parallelization process.

3. **Complexity of the task (Q - Cost)** : Let's assume that `N` is the number of elements to be processed (see previous list-item) and `Q` represents the costs of processing the elements through the stream pipeline. Then `N*Q` gives an approximate estimation of this cost. So not only a higher value of `N` but also a higher value of `Q` leads to a higher chance of a better performance when using a parallel stream.

4. **Kind of operation**: The benefit of using parallel streams also depends on the operation used on the stream. Some methods will naturally perform worse in parallel mode than on a sequential stream. Such operations like `findFirst()` or `limit` often rely on the order of the stream and so they are expensive in parallel mode. Others like `findAny()` or `map()` will perform much better with parallel streams because it isn't constrained to operate in the encountered order.

   Another problem is the use of functions which do I/O or synchronization. Those functions are extreme examples as they don't pass the independence criterion. Parallelizing them would not make much sense. If every subtask (parallel operation) may be blocked for a significant time waiting for I/O or access, CPU resources may sit idle without any possibility for the JVM to use them [21].

5. **Underlying data structure**: Another important point to consider is the underlying data structure (Collection or Array) of the stream. An `ArrayList` can be split with much less effort than a `LinkedList` because the `ArrayList` can be divided without traversing the whole collection, as it is necessary for linked data structures (see performance experiment in Section 3.4.4).

6. **Boxing and unboxing**: *Boxing* (or *Autoboxing*) is the automatic conversion that the Java compiler accomplished between a primitive type and the corresponding object wrapper class, for example between the primitive type int and its corresponding class Integer. This operation can influence the costs of processing of streams in a very bad way. The solution for this problem when working with primitive values is to use also primitive streams which comes with Java 8. These primitive streams are represented in Java 8 by the classes `IntStream`, `DoubleStream` and `LongStream`. By using these classes instead of using for example `Stream<Integer>`, the additional and expensive step of boxing can be omitted.

The above enumeration outlines how many aspects can influence the performance of parallel stream processing. They all have to be considered when making decisions between sequential and parallel processing. Although they give good indicators only more experience in dealing

Test	CPU cores	Amount of Data (N)	Complexity of task (Q)	Kind of operation	Underlying Data Structure	Boxing
Test A	2	high - 1.Mill	low	non-blocking	multiple Collection types	yes
Test B	2	very high - 20.Mill	low	non-blocking	multiple Collection types	yes & no
Test C	2	low - 10	high	non-blocking	multiple Collection types	yes
Test D	2	medium - 100.000	low	blocking	ArrayList	-

Table 3.2 Overview of the following test scenarios

with streams and parallel streams can prevent the programmer of picking the wrong one. So in case of having doubt about the best solution it is advisable to measure the time-effort of the two approaches and compare them. Since the change between the sequential and the parallel version of stream processing is an easy task by only substitute a method name like `stream()` to `parallelStream()` the effort of testing which processing method behaves faster, is low.

3.4.4 Quantitative Comparison between external Iteration, sequential and parallel Stream Processing

Because the topic of parallel stream processing in Java is brand new and innovative, the necessary experience when dealing with them is very low. This is also discernible in literature. Until now, only very few literature sources exist, which address this topic and intensive studies and comparisons from which new insights can be derived, are completely missing. Another problem is that some findings of the few sources are contradictory.

So this Section should give an idea how the difference factors (compare with the enumeration of the previous Section 3.4.3) like amount of data, complexity of task/operation, kind of operation, Boxing and the the underlying data structure influence the execution time. Therefore four different test scenarios should measure how the execution time changes by influencing these factors. So the execution time in milliseconds acts as the benchmark. Table 3.2 summarizes the different test scenarios and emphasizes the test settings roughly.

Test Description

Test Settings: According to Table 3.2, the first test scenario Test A demonstrates an example from the article 'Iterating over collections in Java 8' [26] executed on a 64 Bit Windows system with a dual-core Pentium processor and 4 GB RAM, while the following three test scenarios

(Test B - Test D) are own designed tests executed on a 64 Bit Mac OSX system with a Intel i5 - 2,5 GHz processor (two cores) and 8 GB RAM.

Measuring Approach: To measure the execution time, one and the same test scenario were executed 100 times. The duration of each iteration was measured in nanoseconds, converted to milliseconds and averaged over all 100 iterations (see Listing 3.11). This is how the values in the result tables were calculated. Since for every single test scenario (like an external iteration of an ArrayList or a parallel stream processing of an ArrayList), a new application was launched, the JVM warm up phase was not taken into account, because the prerequisites were the same for all of them. This fact ensures the comparability of the results.

To demonstrate also the impact of the underlying data structure when traversing collections, the tests 'A', 'B' and 'C' were executed for the most common ones, that are `LinkedList`, `ArrayList` ,`HashSet`, `LinkedHashSet` and `TreeSet`.

```
public static void main (String []args) {
    final int TEST_RUNS=100;
    double duration;
    double executionTime = 0;

    for (int i=0; i<TEST_RUNS; i++) {
        duration = System.nanoTime();
        //EXECUTION OF THE SPECIFIC TEST SCENARIO HERE
        executionTime += duration = (duration - System.nanoTime());
        System.out.println(duration);
    }
     System.out.println((System.nanoTime() - duration)/1000000/TEST_RUNS);
}
```

Listing 3.11 General Measurement Approach

Result Representation: The results of the test scenario are represented by a Table with the following columns:

- **Collection type**: Underlying type of data structure on which the test scenario is executed.
- **External iteration (A)**: Execution time in milliseconds for processing the collection with an external iteration approach.
- **Sequential stream (B)**: Execution time in ms for processing the collection with a sequential stream approach.
- **Parallel stream (C)**: Execution time in ms for processing the collection with a parallel stream approach.

- **Relative change (C/A)**: Shows the relative change of execution time between external iteration and parallel stream processing. A positive percentage value indicates a higher execution time for parallel processing in contrast to external iteration, while a negative one indicates a faster execution time.
- **Relative change (C/B)**: Similar to the previous point, but this column demonstrates the relative change between sequential stream and parallel stream processing
- **Trend**: Possible values are ↗, indicating a slower execution time of parallel execution, and ↘ to indicate a faster execution time parallel execution.

Test 1

This test scenario is taken from the article [26]. The benchmark code counts the numbers of names in a collection with 1 million elements (N=1.000.000) that begin with letter 'A'. The costs for each element (Q) are very low, since there is only one method call to check if the element starts with the letter 'A'.
For the test three different methods were used which all do the same, but by using different approaches. The first method `testExternalIterator` uses an external iteration approach, while the second `testSequential` and the third `testParallel` method use an internal iteration approach by using streams. In contrast to the second method, the third method is based on parallel stream processing (see Listing A.1 in appendix). All these test scenarios where run with five different collection types (see Table 3.3).

The average results of several runs are shown in Table 3.3.

Collection Type	External Iteration (A)	Sequ. Stream (B)	Parallel Stream (C)	Trend	Relative Change (C/A)	Relative Change (C/B)
`LinkedList`	5,21	5,17	7,99	↗	+ 53,36%	+ 54,55%
`ArrayList`	3,21	3,67	2,62	↘	- 18,38%	- 28,61%
`HashSet`	11,43	14,98	8,72	↘	- 23,71%	- 41,79%
`LinkedHashSet`	5,21	5,34	8,26	↗	+ 58,41%	+ 54,68%
`TreeSet`	11,80	12,42	8,51	↘	- 27,88%	- 31,48%

Table 3.3 Test 1 - Iteration benchmark results [26]

Test 2

To verify the data taken out of the literature in the last Section, a similar test should be performed where the number of elements of the collection (N = 20 Mill.) is much higher. To examine also

the influence of boxing (autoboxing), the test scenario has been changed slightly. The test now deals with integers and checks a modulo-condition (see Listing A.2 in appendix). This fulfills the requirements needed to emphasize the impact of autoboxing. The last method in Listing A.2 shows the same test scenario, but by using an `IntStream` in parallel mode to avoid auto boxing, to emphasize also the impact of boxing.

Table 3.4 shows the measurement results. The trends of these results are the same like in the previous test (compare with Table 3.3), but in contrast to them, the influence on the execution time is more distinct, since the number of elements (`N`) is much higher.
The additional last row in Table 3.3, shows the influence on execution time when omitting autoboxing, by using an `IntStream` instead of `Stream<Integer>`. As a result the execution time decreased dramatically, no matter which iteration approach was used (external, internal or parallel internal approach).

Collection Type	External Iteration (A)	Sequ. Stream (B)	Parallel Stream (C)	Trend	Relative Change (C/A)	Relative Change (C/B)
`LinkedList`	120,45	106,31	176,17	↗	+ 46,26%	+ 65,71%
`ArrayList`	62,97	59,76	35,65	↘	- 43,39%	- 40,34%
`HashSet`	151,68	189,43	105,64	↘	- 30,35%	- 44,23%
`LinkedHashSet`	142,22	149,17	228,58	↗	+ 60,72%	+ 53,23%
`TreeSet`	166,07	158,79	89,69	↘	- 45,99%	- 43,52%
`IntStream.Range`	24,19	48,24	23,83	↘	- 1,49%	- 50,60%

Table 3.4 Test 2 - Iteration benchmark results

Test 3

In contrast to the last test scenario, the number of elements is now very small (N=10) while the cost (Q = 2 Mill.) of processing each element is much higher by iterating over a loop 2 million times (see Listing A.3 in appendix).

The measurement results are represented in the Table 3.5. The finding clearly shows that the influence of the underlying data structure is negligible in this case. That is because the number of elements which has to be processed is with only 10 element very low, and so the influence of a worse decomposability of some data structure is marginal. That leads to an improvement of execution time when using parallel stream processing, no matter which type of data structure the underlying collection has.

Collection Type	External Iteration (A)	Sequ. Stream (B)	Parallel Stream (C)	Trend	Relative Change (C/A)	Relative Change (C/B)
`LinkedList`	80,8	85,5	66,4	↘	- 17,82%	- 22,34%
`ArrayList`	83,6	88,2	67,5	↘	- 19,26%	- 23,47%
`HashSet`	85,5	86,6	68	↘	- 20,47%	- 21,48%
`LinkedHashSet`	85,8	84,6	65,9	↘	- 23,19%	- 22,10%
`TreeSet`	86,4	84,2	66,8	↘	- 22,69%	- 20,67%

Table 3.5 Test 3 - Iteration benchmark results

Test 4

The last test should demonstrate the influence of blocking or synchronized functions. For that test scenario, only the collection type `ArrayList` is used (see Listing A.4 in appendix), which is a perfect decomposable data structure and consists of 100.000 elements (N). The cost for processing each element (Q) is low since every element is only outputted to console without doing further operations. As the values in Table 3.6 show, this task is not suitable for parallelism, because the output function is a blocking function. Because of the additional overhead, which has to be performed when using streams and parallel streams, the execution time is even higher when parallel stream processing is used.

Collection Type	External Iteration (A)	Sequ. Stream (B)	Parallel Stream (C)	Trend	Relative Change (C/A)	Relative Change (C/B)
`LinkedList`	686,4	824,2	988,4	↗	+ 43,99%	+ 19,92%

Table 3.6 Test 4 - Iteration benchmark results

CHAPTER 4

jTL - Java Transformation Library

4.1 Introduction

This chapter describes in a detailed way a general transformation approach by introducing an Java-based library called *Java Transformation Library*, in short *jTL*. Based on the findings of the Chapters 2.1.1 and 2.2.1 an architecture is designed that combines the advantages of these two approaches and add some further improvements. These improvements also concerns the concrete implementation of the jTL, as it takes also the benefit of using novel Java 8 features into account.
The basic idea of this work was to design a transformation library which can be used in different fields of application. This addresses graph based transformation as well as translation of programms. Thus the architecture of jTL should at least fulfill a general approach, to use it for a wide range of transformation problems. This means that the resulting jTL should not only deal with tree based source models like an AST, but also with all kind of graph based source models. Furthermore the jTL should be able to deal with cycles in the source model and therefore should address the problem of non-terminating recursive transformation properly. Last but not least a tracing approach and rule inheritance as reuse mechanism should complement the jTL.

The most important principles respectively goals of the jTL, which should be achieved when designing and implementing the library are:

1. Performance improvements: especially in contrast to SiTra, where Section 2.1.1 reveals some shortcomings.

2. Flexibility (external as well as internal flexibility): External flexibility means, that jTL should address a wide range of application scopes, while internal flexibility addresses issues like the flexibility of rule selection.

3. Simplicity and clarity

Simplicity/Clarity on the one hand and Flexibility on the other hand are somehow conflicting issues. Therefore a satisfying and good tradeoff of these two characteristics has to be found.

4.2 Architecture

4.2.1 Naming Conventions

For the design of jTL, the naming convention should follow the well known notations which are widely used in the area of model transformation and therefore contribute to a better readability and maintainability. In contrast to the PC2J-Transformer which has slightly different naming convention which seems to be a bit confusing, SiTra already follows these conventions. The jTL has therefore the following named components

- Transformer - interface
- Rule - functional interface
- RuleTransformationException - exception class
- SourceElement - abstract class
- TransformationContext - class
- Tracing - class

4.2.2 Overview

Figure 4.1 shows the overview of the jTL consisting of the seven elements which were already mentioned in previous enumeration 4.2.1. They all together are part of the package `jTL` and build the basis for transformation process.
In the following the components of the jTL are discussed in more detail, without forgetting to make cross-comparison between jTL and SiTra as well as PC2J-Transformer. These cross-comparisons should especially outline which motivation and reasons cause differences in the architecture and implementation compared to SiTra (2.1.1) and PC2J-Transformer (2.2.1) and what advantages occur from them.

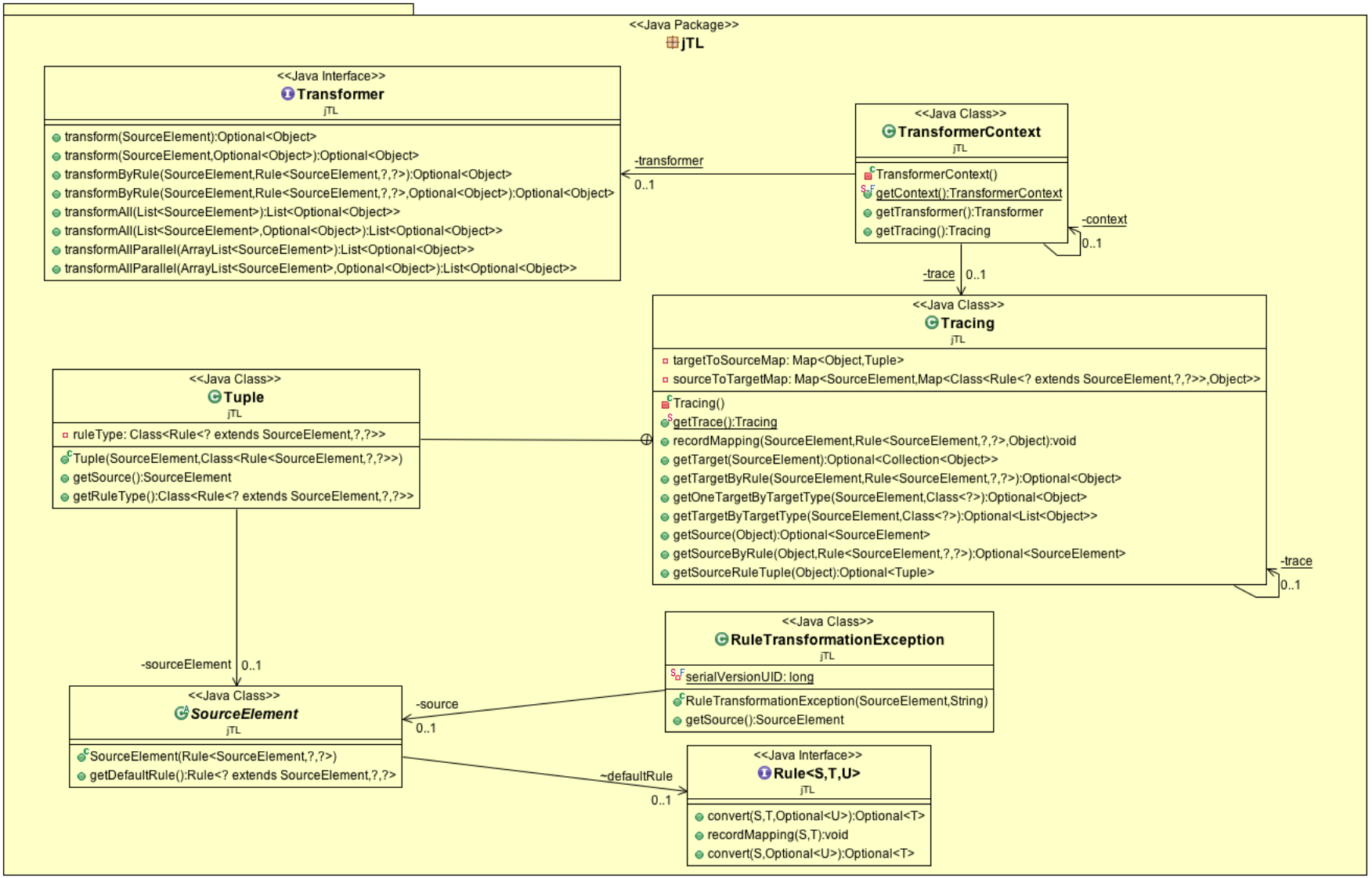

Figure 4.1 Architecture of the jTL

4.3 Source Element

The starting point of every transformation is the source model consisting of the different source elements according to the source meta-model. The abstract class `SourceElement` should serve as the general superclass for all kinds of source elements. Since there is no reason to create an instance of this class it has been implemented as an abstract class. Although the source elements in general have different kind of structures there are two reasons, which are introduced in the next two subsections, why a general superclass `SourceElement` was designed to be part of the jTL.

4.3.1 Default Rule

The main reason for the abstract class `SourceElement` being part of the jTL is the idea of binding a default rule to every type of source element. This concept is new since SiTra and PC2J-Transformer go completely different ways. Especially SiTra was the reason to rethink the concept of how a corresponding rule is chosen to transform a source element.
SiTra's concept is to call every single rule of the rule-repository to check if the visited rule is able to transform the source element by checking the type and comparing it with the type which the rule can handle. This concept is neither intuitive or practicable nor performant and leads to an unnecessary overhead.
The PC2J-Transformer already follows a better and more performant approach by holding a rule-repository (name `ConverterRepository`), which already links every source element with the corresponding rule implemented by a Hashmap. In contrast to SiTra it is not necessary anymore to iterate over all rules, since one single request to the rule-repository is enough to retrieve the applicable rule. Although this approach is mature and practicable there is one aspect which influences the decision against using a rule repository in jTL. In general every source element type is always transformed by one and the same rule. In fact there is a fix linkage between the source element type and its rule. So the question when designing the jTL was: 'Is there a real need for a repository to map this linkage?'. In fact no, because this linkage can already be expressed in the source element. So the decision in jTL was to hold the mapping to the applicable rule by referencing the rule in the world `SourceElement`. To fulfill also the design principle of 'Separation of Concerns' [37], it was important that the `SourceElement` does not already hold the actual transformation code, but only holds the reference to the applicable rule. So every concrete source element should inherit from the abstract class `SourceElement` which already makes available an instance of the type `Rule` which represents the link to the default rule, as well as a constructor to initialize the rule-object and an according getter for the `Rule`-instance (see Listing 4.1).

```
public abstract class SourceElement {

    Rule<? extends SourceElement,?,?> defaultRule;

    public SourceElement(Rule<? extends SourceElement,?,?> defaultRule) {
        this.defaultRule=defaultRule;
    }

    public Rule<? extends SourceElement,?,?> getDefaultRule() {
        return defaultRule;
    }
}
```

Listing 4.1 Source Element (Abstract Class)

SiTra's as well as PC2J-Transformer architecture enables the possibility that theoretically one source element can have more than one applicable rule that transform it. In SiTra the `check`-method of a rule could return true for more than one rule when passing one and the same source object, while the PC2J-Transformer enables this possibility by mapping a list of rules to one and the same source element. But in practice the implementation showed that SiTra only would execute the rule which is found first and ignoring further possible matches, while the PC2J-Transformer never made use of assigning more than one rule to a source element. Although it would make use of this feature, the concrete implementation shows that it would only pics the last rule of the list for the transformation of the source element and ignore the previous once. Based on these findings the jTL avoids the possibility to link more than one rule to a source element, since there is no use for it and would only worsen the readability. Nevertheless jTL provides a possibility to transform an source object by an alternative rule instead of using the default one. This approach is explained in the next paragraph.

In nearly every transformation case the use of the default rule would be the matter of choice when transforming a source element. The use of a default rule would result in simplicity and clarity, which was defined as one important goal when designing the jTL. But the problem with this is that it would violate the goal of flexibility, since there would be no possibility to run a different rule for a source element type in exceptional cases. To overcome this issue the interface `Transformer` supports a possibility to pass an alternative rule for the transformation. This possibility is described in more detail in the Section 4.6.2. The third goal 'Performance Improvements' defined earlier is satisfied, because the applicable rule is already known, when a source element is passed to transform, since the source element holds the reference to this rule. In contrast to SiTra, where the transformer has to search the applicable rule by iterating over all rules and checking the compliance of the types, this is a big advantage of the jTL in performance.

4.4 Rule

4.4.1 Changes compared to SiTra

The interface `Rule` acts as super type for all concrete implementation of transformation rule. In contrast to the corresponding interface in SiTra, it only consists of one abstract method named `convert` and a default method `recordMapping`. SiTra's equivalent of the interface `Rule` consists of two more methods that are `check` and `setProperties`. The reason why the SiTra library forces the developer to implement these methods were already explained earlier (see Section 2.1.3). Hence, the following description is limited to the argumentation why the jTL can waive the use of these methods.

- **Waiver of the the `check`-method:** Since SiTra does not hold any information to emphasize the linkage between a source element type and its corresponding rule-class it has to make use of this method to check every single rule whether it is applicable for the specific source type or not. In contrast the jTL has already knowledge about the corresponding rule class. Regardless of whether the default rule is used, which is referenced directly in the source element (see Section 4.3.1), or any other specific alternative rule is chosen in the Transformer (see 4.6.3), the called rule class can be sure that it can handle and transform the passed source element. So this time consuming step has been omitted in the jTL.

- **Waiver of `setProperties`-method:** SiTra uses this method to split up the actual transformation of the source element from the transformation of further referenced source elements to avoid the non-terminating recursive transformation problem. But as mentioned above there are also many cases where this kind of problem can't happen, for example when transforming a tree based source model. Nevertheless SiTra forces the developer to implement this method and split up the actual transformation process. In my opinion this is an overwhelming and in addition not necessary approach. Therefore the jTL relies on a little more light weighted implementation by using the default method `recordMapping` to avoid the non-terminating recursive transformation problem. As soon as the target element is created the method `recordMapping` has to be called to register it to the class `Tracing` (see 4.8.2). That has to happen before any further linked source objects are transformed. So it's up to the developer to handle the registration properly and in the right order. But in case of SiTra it's also up to the developer to implement the methods `convert` and `setProperties` as it is supposed to be and call them in right order. So the jTL approach does not lead to any usability disadvantage. Moreover jTL enables the user to avoid the call of the registration method `recordMapping`, when

in a transformation case the non-terminating recursive can't occur and the tracing ability is not needed. In contrast to jTL, SiTra forces the user to implement the methods `convert` and `setProperties`, although the split up wouldn't be necessary.

4.4.2 Overview

```
import java.util.Optional;
import static jTL.TransformerContext.getContext;

@FunctionalInterface
public interface Rule<S extends SourceElement,T, U extends Object> {

    public Optional<T> convert(S source, T target, Optional<U> parent);

    public default void recordMapping(S source, T target) {
        getContext().getTracing().recordMapping(source, this, target);
    }

    public default Optional<T> convert(S source, Optional<U> parent) {
        return convert(source, null, parent);
    }
}
```

Listing 4.2 Rule (Functional Interface)

When looking on Listing 4.2 the annotation `@FunctionalInterface` is noticeable which indicates that the following interface definition has to be a functional one (also called SAM Type - Single Abstract Method Type). The concept of functional interface was already invented in the Section 3.3.1 but without noticing the annotation. The annotation `@FunctionalInterface` can be used for denoting an interface as a functional interface. Although this decoration is not necessary, it can support the development process by catching an error in the interface definition at compile time. As soon as more than one abstract method is defined, the compiler will raise an exception. Addition default method can be used anyway (see method `recordMapping` in interface `Rule` in Listing 4.2).

The reason for the decision to use a functional interface for the interface `Rule` was to enable the possibility to pass functionality by using lambda expressions. This is the prerequisite to define the implementation of the interface method and passing it to the `Transformer` without creating an extra class for the implementation of the interface. The approach of using lambda expression and gaining flexibility in the transformation process without loosing readability is explained in more detail in the Section 4.6.2. Nevertheless the classical approach, by implementing the interface `Rule` in an extra class-file, is the preferred solution if the `convert`-method, which contains the conversion code, is more complex or longer.

4.4.3 Method `convert`

The interface `Rule` comes up with two different signatures of the convert-methods. The one which only consists of one parameter called `source` and the placeholder for an optional parent object, has a default implementation and shouldn't be overwritten by the user. Unfortunately this can't be forced by using the keyword `final`, since a default method of a functional interface does not support the use of it. This method only takes the source object and delegates it to the second abstract version of the `convert`-method, by adding null as parameter for the target object. The reason for this detour lies in the reuse mechanism rule inheritance and was necessary to provide this important mechanism to the user. Section 4.9 describes this approach in more detail and gives also a detailed answer why the `Rule`-interface has to hold two versions of the method `convert`. For now it is enough to know that, whenever rule inheritance as reuse mechanism is not used, the call of the simpler version of `convert` by passing only the source object satisfies the needs for transforming a source object. This `convert`-method takes a source element from the class-type `SourceElement` 4.1 and returns an `Optional<Object>`. The parameter parent from the type `Optional<Object>` represents the parent object of the return element respectively the current transformed target element. Since the parent target object is often necessary for code generation when transforming source code, this parameter was necessary to be part of `convert` method. For example if a move statement has to be part of a loop statement, the rule which converts the move statement has to have knowledge about the parent loop statement in which the move statement has to be generated. Thats the reason for the additional parameter `parent`.

A possible implementation of this method should follow the steps emphasized in the Figure 4.2.

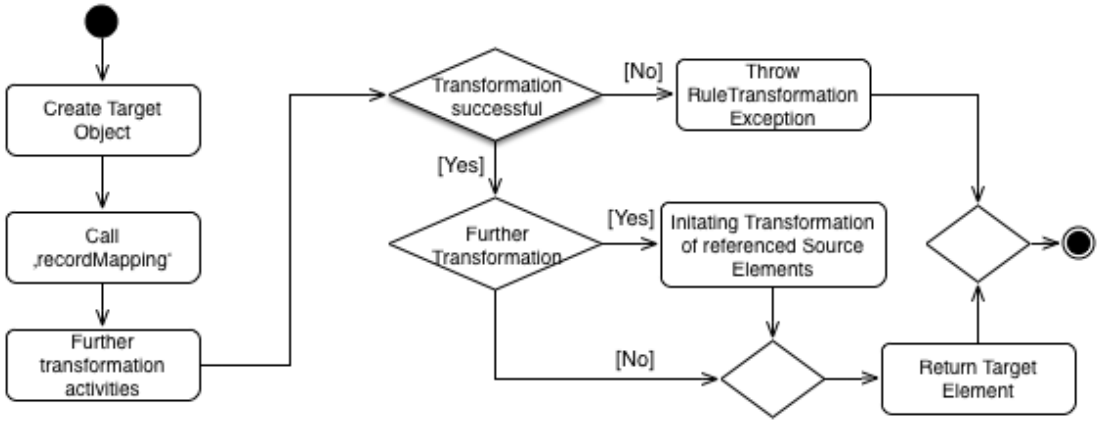

Figure 4.2 Guideline for the process of the `convert`-method

1. The first step is the creation of the target element. At that moment the target element does not have to be fully transformed respectively configured. Only an instance as placeholder has to be generated, to register this target element in the next step.

2. Call of the default method `recordMapping`: Optional step - only necessary if the source model contains cycle or tracing is required. Registering the previously generated target element by calling the `recordMapping`, further indicates that the transformation is in progress or already done. This is the prerequisite to avoid the non-terminating recursive transformation. Furthermore the check if a specific source element was already transformed, has to be done in the `Transformer` (see Section 4.6.2).

3. Further transformation activities: This contains all the necessary steps to configuring the target element according to the source element. Optional these activities can also be executed after further linked source element has been transformed. So the order of the procedure is not fixed. If the actual rule-transformation code detects further linked source element, which has to be transformed by a separate rule, the rule delegates these source elements to the `Transformer`.

4. Return the target element: For a transformation library like jTL which should cover a wide range of possible transformation tasks, the target type has to be declared as general as possible. This is the reason why the class `Object` has to be used as generic type. In some cases one source element is transformed to more than one target element, so the `convert`-method has to be able to return even a collection of target elements. This requirement can only be fulfilled by using the general type `Object`. Although this is not a very practicable approach, since every transformed target object has to be cast to the expected type, it is the only possible approach to fulfill the goal of flexibility. Compared to the library SiTra this approach is very similar. But in contrast to that, the PC2J-Transformer is limited to transform source elements only to Java code, which is depicted by the jenesis4Java class `Codeable`. Since the PC2J-Transformer is already a concrete implementation of a transformation process, it can use the predefined return type `Codeable` and does not have to execute additional casting operation but it does not fulfill the goal of flexibility. In fact it is a rigid concept.
   Instead of returning the target element directly as a type of `Object`, the new Java 8 concept of `Optional<T>` is used. The reason and benefit of this approach is explained in the next Section 4.4.4.

5. Raise exception: In exceptional cases the transformation by a rule can't be executed successfully. Examples would be the misconfiguration of a source element or missing in-

formation. Then the rule implementation should raise the predefined exception `RuleTransformationException` to indicate the fail of the transformation process (see Section 4.5).

4.4.4 Excursus: Return Type Optional

In jTL, the decision was made to use a further new concept of Java 8 that is the class `Optional<T>`. The idea of the class is to avoid `NullPointerException`. Tony Hoare said at an important speech: 'I call it my billion-dollar mistake. It was the invention of the null reference in 1965. ... But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years' [14]. Furthermore he pointed out that more recent programming languages like *Spec#* have introduced concepts to overcome these issues, like declaration for null values. Since version 8, Java invented also a way to face this problem by introducing the class `Optional`. Although most developers have learned to live with NullPointerException over the years, it has always been a questionable concept. Consequently the advantages of using `Optional<T>` should be emphasized, why jTL relies on this concept. The following enumeration describes three advantages of using `Optional<T>`:

1. A method which returns an object of `Optional<T>` states explicitly that the return value is optional. For a developer who uses an API, for example jTL, it is obvious that she/he has to deal with the optional value by implementing also a strategy to face an empty value.

2. With the consequent use of `Optional<T>` the occurrence of `NullPointerException` is a thing of the past. Often `NullPointerException` firstly occurred during the operational phase, which leads to an unexpected error and a rise in costs.

3. Last but not least the class `Optional` provides many useful methods to easily deal which alternatives. Such methods are `isPresent`, `ifPresent`, `orElse`, `orElseGet` or `orElseThrow`. Some of them expect functionality as an argument, which can be passed simply by a lambda expression.

From my own experience, when working on the PC2J-Transformer, `NullPointerExceptions` occurred very often during the conversion phase, since a rule which can't transform a source element successfully, returns a null-value. In the implementation phase it was often forgotten to handle the case of a null-value, which leads to an unexpected error in a later phase of development. So the developer was busy by searching the reason for that error and designing a

solution for the exceptional case. With the use of the class `Optional<T>` the risk of forgetting to handle the case of an empty return value is much lower.

4.5 RuleTransformationException

```
public class RuleTransformationException extends RuntimeException{
    private SourceElement source;

    public RuleTransformationException (SourceElement source, String msg) {
        super(msg);
        this.source = source;
    }
    public SourceElement getSource() {
        return source;
    }
}
```

Listing 4.3 Rule (Functional Interface)

The exception `RuleTransformationException` should be thrown in the method `convert()` of any implementation class of the `Rule`-interface, when the transformation of the source element can't be processed successfully. This can happen when the source element is configured wrong or if for example no parent target object was passed in case of source code transformation. When throwing this exception the constructor of `RuleTransformationException` forces the developer to pass also the source element where the transformation error has occurred.
If the conversion is successfully but no target element has to be created, then this should be obvious through an empty `Optional`-value by returning `Optional.empty()`.

4.6 Transformer

4.6.1 Comparison to SiTra

The interface `Transformer` represents the core of the jTL and fulfills managing and distribution issues. The name of the interface is the same like in SiTra and also the tasks are similar. Nevertheless the way of realization is significantly different compared to its counterpart in SiTra.

On the one hand defining an interface which specifies the structure of the transformation task and giving the developer the chance to set up his own implementation of the `Transformer`-interface represents a good best practice approach and further fulfills the goal of flexibility which was defined as one of the three important goals when designing the jTL (see Section 4.1). But on the other hand the two other defined goals that are 'Simplicity and Clarity' and 'Performance

improvements' are only addressed insufficient or even not at all. Although the goal 'Simplicity and Clarity' is satisfied through an interface with less and clearly distinguished method signatures, as in case of SiTra's interface `Transformer`, the whole implementation of the interface has to be addressed by the developer. In practice this seems to be an unnecessary approach, since the concrete implementation of the interface `Transformer` would differ only in the rarest of cases. That is because the transformer is going to address always the same tasks, that are managing incoming source elements and delegating them to the applicable rules. So the question is: 'Why has the developer to be forced to implement a task that probably stays the same over different transformation contexts and fields?'.
Furthermore an example of a concrete implementation of SiTra's interface `Transformer` shows that further methods like for example `getRule`, `applyRule` or `recordMapping` are more or less necessary to implement a complete transformer which however is not part of the interface definition. So in my opinion SiTra's interface is an insufficient, incomplete one. Although it maybe fulfills the goal of 'flexibility' perfectly, the definition seems to be too vague.

4.6.2 `Transformer`-interface of jTL

To overcome the above mentioned flaws, a goal when designing core components of jTL was to provide not only a vague structure to the developer but also a well designed implementation, that is applicable for most of the possible transformation tasks. Nevertheless it was important not to lose sight of the goal 'flexibly'. So the aim was to design a framework component, which already comes up with a ready-to-use implementation but also providing the possibility for easily extending it by the developer. This of course would fulfill the goal of 'simplicity and clarity', since the developer is not forced to write his own implementation but without violating the goal of 'flexibility', since he would be able to override and reimplement the already provided implementation.

With interface default methods of Java 8 a new option arises to fulfill the above mentioned requirements. Before Java 8 was released, only abstract classes or concrete classes were possible methods of choice. But since the core component of the transformer only consists of public methods without declarations of any fields also an interface could be use to provide all the advantages to the developer. So the resulting component is an interface called `Transformer` as it is also provided by SiTra, but this time in jTL it already comes up with a default implementation for each method. So the user of this interface easily notices that he has the possibility to make his own implementation of the interface when needed in special cases, but on the other hand has an already-to-use default implementation which is adequate in most situations. So the goals of 'flexibility' and 'simplicity and Clarity' are fulfilled with this approach.

4.6.3 Interface Methods

```
public interface Transformer{
    /******************* TRANSFORM ONE BY DEFAULT RULE ************************/
    public default Optional<Object> transform(SourceElement source) {
        return this.transform(source, Optional.empty());
    }
    public default Optional<Object> transform(SourceElement source, Optional<? extends
        Object> parent) {
        return transformByRule(source, source.getDefaultRule(), parent);
    }
    /******************* TRANSFORM ONE BY SPECIFIC RULE ************************/
    public default Optional<Object> transformByRule(SourceElement source, Rule<?
        extends SourceElement, ?, ?> rule) {
        return this.transformByRule(source, rule, Optional.empty());
    }
    public default Optional<Object> transformByRule(SourceElement source, Rule<?
        extends SourceElement, ?, ?> rule, Optional<? extends Object> parent) {
        Optional<Object> target = getContext().getTracing().getTargetByRule(source,
            rule);
        return target.isPresent() ? target : ((Rule<SourceElement, Object, Object>)rule
            ).convert(source, (Optional<Object>) parent);
    }
    /*********************** TRANSFORM ALL SEQUENTIAL ***************************/
    public default List<Optional<Object>> transformAll(List<? extends SourceElement>
        sourceList) {
        return this.transformAll(sourceList, Optional.empty());
    }
    public default List<Optional<Object>> transformAll(List<? extends SourceElement>
        sourceList, Optional<? extends Object> parent) {
        return sourceList.stream().map(s -> transform(s,parent)).collect(Collectors.
            toList());
    }
    /************************ TRANSFORM ALL PARALLEL ****************************/
    public default List<Optional<Object>> transformAllParallel(ArrayList<? extends
        SourceElement> sourceList) {
        return this.transformAllParallel(sourceList, Optional.empty());
    }
    public default List<Optional<Object>> transformAllParallel(ArrayList<? extends
        SourceElement> sourceList, Optional<? extends Object> parent) {
        return sourceList.parallelStream().map(s -> transform(s,parent)).collect(
            Collectors.toList());
    }
}
```

Listing 4.4 Transformer Interface with default Method

Listing 4.4 shows the interface `Transformer` with its default methods which consists of eight default methods, where four of these methods are only overloaded delegator methods which takes the additonal parameter called `parent`. As already mentioned the parent object of the current target element is often necessary when transforming for example source code. Nevertheless, since four methods are only overloaded delegators, the focus will lie on the four different named method in general. The following classification shown in Figure 4.3 should give a first sight when to use which method. Further the method are described in more detail.

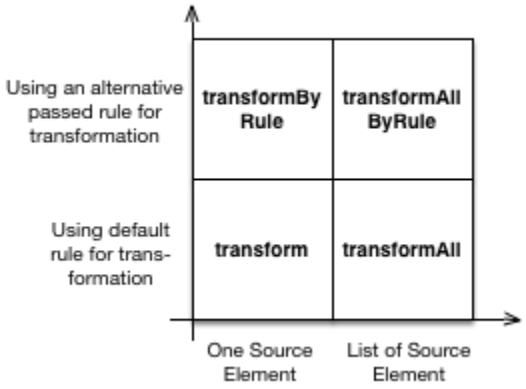

Figure 4.3 Classification of the `Transformer`-methods

Interface Method - `transformByRule`

This method takes not only the source element as input parameter, but also the rule which is applicable for the source element and should be used for the transformation. This rule can be the default rule which is referenced by the source element (see 4.6.3), or it can be a different one specified by the developer. This ensures the achievement of the goal 'flexibility' which was defined above. One obvious possibility to pass a different rule than the default one, is to design a class which implements the interface `Rule` and defines its behaviour and transform instructions in the method `convert`. But since the `Rule`-interface is a functional interface with only one abstract method, this enables also the possibility of using anonymous inner classes or lambda expression as implementation definition. While the implementation of anonymous inner classes is a well known practice in Java, the use of lambda expression is new since Java 8. Therefore the example 4.5 should demonstrate a possibility to call the `transformByRule`-method by passing the source element as well as a specified rule by using a lambda expression.

```
//Long Version
target = transformer.transformByRule(source,
  (Source Element srcObj) -> {return Optional.of(new TargetClass(srcObj.getName()));});

//Short Version
target = getContext().getTransformer().transformByRule(man,
  (srcObj) -> Optional.of(new TargetClass(srcObj.getName())));
```

Listing 4.5 Call of `transformByRule` by passing a Rule as lambda expression

Listing 4.5 shows how compact but still readable the transformation code of a rule can be passed to the `Transformer` by using lambda expressions. The first and little more longer version of the lambda expression enables the developer to also specifies more than one statement to

describe the rule-transformation-code. Nevertheless it should be mentioned that the more statements a lambda expression comprises, the more the readability and maintainability will suffer. It's up to the developer to decide whether a lambda expression is the better and easier solution or not. If there are too many statements needed to describe a transformation process of a rule, the better maintainable solution would be the implementation of a concrete rule-class derived from the interface `Rule` in an extra Java-file.

Implementation technique to avoid the non-terminating recursive transformation problem: Before any source object is passed to it's applicable rule the main task of the `Transformer` is to check if the source element was already transformed. This is established by using the class `Tracing`, which is described in more detail in Section 4.8. To examine if the source object was already transformed, the method `getTarget` is called by passing the source element as well as the rule, which further should be used to transform the element. If the source element was already transformed by the specified rule, the previously transformed target element is returned. If not, the method `getTarget` returns an empty `Optional`-object. In this case the transformation for the passed source element has to be established by calling the `transform`-method of the class `Transformer`.

```
    return Optional.of(
        getContext().getTracing().getTargetByRule(source, rule).
        orElseGet(
        () -> ((Rule<SourceElement, Object, Object>)rule).convert(source, (Optional<
            Object>)parent)));
```

Listing 4.6 Alternative implementation of `transformByRule` with `orElseGet`

Another elegant solution to check whether the passed source-element was already transformed or not, would be the use of the method `orElseGet` (see Listing 4.6). Again the compiler checks if the source element was already transformed. If an `Optional.empty` is returned the specified lambda expression in the method `orElseGet()` is executed, which then converts the source element to the target element. Using the method `orElse` of the class `Optional` instead of `orElseGet` could worsen the performance and lead to an unnecessary overhead. Because, in contrast to the method `orElseGet`, the method `orElse` is always executed, although the else-branch is not needed and does not influence the result. According to the API of Java 8, the method `orElseGet`, should be used when the computing of the else-branch is slow. Since in this case the transformation of a source-element and all its possible linked source-elements can be a very time consuming job, it's strongly recommended to use the `orElseGet`-method.

Interface Method - `transform`

The simple `transform`-method is the method of choice if the default rule should be used for the transformation. Therefore the developer only has to pass the source element, but does not need to specify the rule for the transformation rule. The `transform`-method then delegates the calls to the method `transformByRule` (see Section 4.6.3) by passing the source element as well as the default rule which is referenced by the source object. So in most cases this method would fulfill the needs to transform a source object.

Interface Method - `transformAll`

This method take a collection from the type `List`, which represents the source elements, as input and return a List of `Optional<Object>`, which represents the list of the transformed target object. These method should be used if multiple elements has to be transformed by using their default rules.

Interface Method - `transformAllParallel`

This method also transforms the passed collection of source elements. But in contrast to the previous mentioned method `transformAll`, the implementation of the method `transformAllParallel` executes the actual transformation in parallel mode by using.

Input Parameter - `ArrayList` The method `transformAllParallel`, forces the developer to pass an `ArrayList`. At first sign this is a very unusual definition since usually a more generic type like the interface `List` should be used. But according to findings in Chapter 3.4.4 the type of the underlying data structure is very important and significant for a possible performance increase. While the use of linked data structures like `LinkedList` can worsen the performance up to 60%, a data structure based on the type `ArrayList` can improve the performance up to 43% (see the benchmark in Section 3.4.4). Choosing the generic type `List` as parameter otherwise would enable the developer to choose also a `LinkedList`, what should be avoided.

Performance As mentioned in the beginning of this Chapter, the goal of 'Performance Improvements' is also one of the three important ones to achieve. One approach to fulfill this goal was the easy and fast selection of the applicable rule (see Section 4.3.1) in contrast to SiTra. While SiTra has to iterate over all known rules the source elements in jTL already know the according rule. Although if the developer wants do define another rule than the default one for the transformation, the performance will not deteriorate.

The second and further important approach to improve the performance in contrast to SiTra and also PC2J-Transformer, is the implementation of parallelism. Since Java 8 this can be easily done by using parallel streams instead of implementing a lot of boilerplate code to use threads or the fork-join mechanism (see Section 3.4.3).

```
//Long Version
sourceList.parallelStream().map((SourceElement s) -> {return transform(s,parent)}).
    collect(Collectors.toList());

//Short Version
sourceList.parallelStream().map(s -> transform(s,parent)).collect(Collectors.toList());
```

Listing 4.7 Using parallel Streams to start for parallel rule invocation

Parallel Streams Listing 4.7 shows two possibilities to start parallel transformation by using parallel streams. Both have the following issues in common

- Based on the `ArrayList` which represents the passed source elements, a parallel stream is created via the standard API-method `parallelStream`.
- The method `map` is executed on the resulting parallel stream, which means on every single element of it. As input parameter this method takes a function which describes how the elements of the streams should be mapped to target elements. Therefore a lambda expression is used to call the `convert`-method of the applicable rule.
- The previous described method `map` is an intermediate operation which returns a stream again. Therefore an additional terminal operation has to be executed which is represented by the `collect`-method. This method makes use of the standard API class `Collectors` that provides various useful reduction operations. The `Collectors.toList`-method which is used in the interface `Transformer` converts the stream back to a list.

4.7 TransformerContext

4.7.1 Purpose of `TransformerContext`

The class `TransformerContext` is an approach which already proves very useful in the PC2J-Transformer. It is designated to hold all necessary environment variables and offering useful utility and helper methods for the transformation process. Initial in jTL this class already comes up with two environment variables, that are from the class-types `Transformer` and `Tracing` and its related getters. Since the use of these instances are wide spread over the

whole transformation process and many different rule-classes, a central unit where these objects are stored is very expedient. In contrast to jTL, SiTra do not make use of this concept which results in passing the `Transformer`-instance to every single rule and referenced sub-rules.

```
public class TransformerContext {

  private static TransformerContext context;
  private static Transformer transformer;
  private static Tracing trace;

  private TransformerContext () {}

  public static final TransformerContext getContext() {
    if (context == null) {
        context = new TransformerContext ();
        transformer = new TransformerImpl();
        trace = Tracing.getTrace();
    }
    return TransformerContext.context;
  }
  // getter for Transfomer and Tracing object
}
```

Listing 4.8 Singleton - TransformerContext

4.7.2 TransformerContext as Singleton

The `TransformerContext` was designed as singleton pattern, which is a well known creational pattern of the GoF [12]. It ensures that one class holds exactly one instance of itself and provides one global access point to it. This approach perfectly fulfill the needs of the TransformerContext. Further this pattern is also used for the class `Tracing` (see Section 4.8) which is referenced within the `TransformerContext`.
Since the global access point for the instance of `TransformerContext` is declared as 'static final' (Listing 4.8), the instance of this central class can be accessed from any other class through a static import of `TransformerContext.getContext`. So every rule-class which needs access to the `Transformer`-class to transform further referenced source elements, can directly get access to the `TransformerContext` and further to the `Transformer`-object (see Listing 4.9).

```
    import static jTL.TransformerContext.getContext;
        //Convert-method by using the TransformerContext
        getContext().getTransformer().transform(a);
```

Listing 4.9 Global Access to TransformerContext

4.8 Tracing

4.8.1 Purpose of Tracing in jTL

The needs and benefits of tracing, were already discussed in detail in Section 2.1.3. To sum up the most important reasons for tracing are:

- Detection of cycles in the underlying source model
- Solving the interoperability problem by providing the possibility for reverse engineering
- Provides good debugging opportunities

Especially the first reason 'Detection of cycles in the underlying source model' plays a very important and indispensable role when implementing a general approach of a transformation library. As soon as a source model is graph based and not tree based and consists of cycles the feature of tracing is a must. Since jTL should not only be applicable to source code transformation, where the source model is based on an AST, which obviously does not contain cycles, it also relies on Tracing. To put it in a nutshell, when an object of the source model is transformed to an object in the target model, and later on the same source object appears as a reference, it should not be transformed again. Instead the previously generated object of the target model should be returned [3]. This kind of implementation was already demonstrated in the default method `transformByRule` of the interface `Transform` (see Listing 4.4), where before a concrete source object is transformed, it's checked whether this object was already transformed or not. If the concrete source object was already transformed, tracing is used to query the corresponding target object/s, without repeating the actual transformation. Otherwise, if the transformation would be repeated, this would lead to the non-terminating recursive transformation problem (see Section 2.1.4).

4.8.2 Tracing implementation in jTL

In jTL the utility class `Tracing` holds the mapping between the source and the target objects as trace links and provides different kinds of resolve methods. The class `Tracing`, which is referenced by the previously mentioned class `TransformerContext`, is implemented as a singleton class, since is it necessary that only one instance of this class exists, which represents the mapping between the source and the target model for the whole transformation process.

Trace Links

To receive all necessary information from tracing, the following attributes have to be represented by a trace link:

- Source object
- Rule type which was used for the transformation
- Target object/s

So a trace link can be represented as a triple consisting of the above listed information. Storing these triples/trace links in a simple list, for example a `java.util.List`, would violate the previously defined goal of 'performance improving' of jTL, since every request would result in a time consuming iteration process to find the corresponding target object to a specified source object. If number of n trace links are hold in the list, a worst case scenario would need n iteration steps to find the designated trace link . To overcome this issue, it was important to choose an applicable data structure that has low access time. Therefore two instances of `HashMap` were chosen - one to hold the mapping from the source object to the target object/s and a second one which holds the mapping from the target object back to the source object. Although this approach needs two HashMap to represent the two unidirectional links from source to target object and vice versa, the benefit of this approach is the low access time, since the cost for several iteration steps are omitted [28].

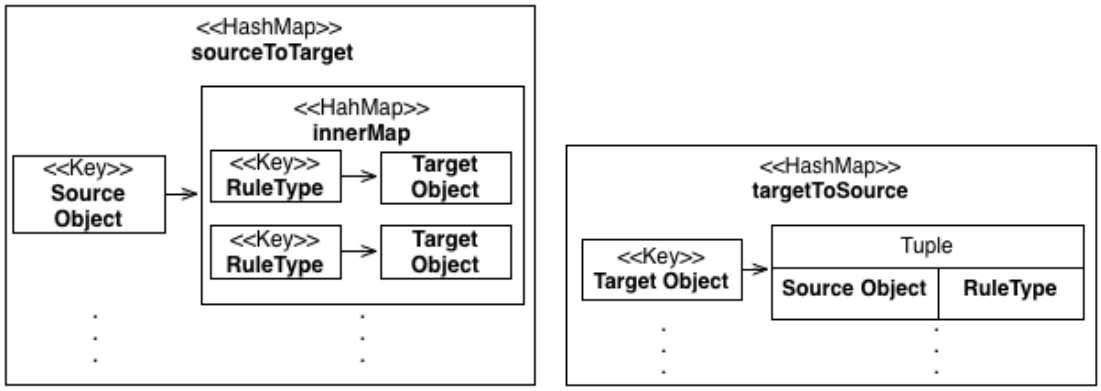

Figure 4.4 Hash Maps holding the trace links: source → target & target → source

1. **sourceToTargetMap**: Holds the unidirectional link between the source object and it's corresponding target object/s, where the source object represents the key of the `sourceToTargetMap`. Since one and the same source object could be mapped to different target object/s by choosing different rule types for the transformation process, this results in an one-to-many relationship between the source object and its corresponding target objects. To represent this structure, every source object key, references another inner hash map. This inner hash map consists of the rule type as key and the target object as value (see Figure 4.4). These data structures ensures not only a fast access to the target object but also the possibility to hold several trace links from one and the same source object to different target objects, transformed by different rule types.

2. **targetToSourceMap**: To enable also a fast resolution from a target object back to the corresponding source object, a second instance from the type `HashMap` is available in the class `Tracing` called `targetToSourceMap`. This map stores the target object as key and a tuple of source object and rule type as value. So the developer can gain the information from which source object the specified target object was generated and which rule type was used to transform it. The tuple, which consists of the source objects and the rule type, is implemented as inner class, called `Tuple` which is part of the surrounding class `Tracing`.

Although at first glance the use of two data structures with the same data leads to redundancy and involves the danger of inconsistent data, it is still the best solution to minimize the time complexity. When looking at SiTra, every resolve request leads to an iteration of all trace links until the right one is found, which results in a high time complexity. The average iteration steps which are needed to find the right trace link is n/2, for n trace links. Since every single source object transformation leads to a recording of a trace link, the number soon rises dramatically.
Therefore in contrast to SiTra, jTL's time complexity was decreased significantly to find the right trace link, by using hash maps which omit the time consuming iteration task. This significant improvement in time complexity is the reason for accepting the data redundancy. To avoid the danger of inconsistency between the two hash maps, the manipulation of the two maps is encapsulated in only one method called `recordMapping` (see Section 4.8.2) in the class `Tracing`. Since the two hash maps are declared as private attributes, and the `recordMapping` is the only method which manipulates them, the risk of running into inconsistency is marginal.

Record Method - `recordMapping`

The method `recordMapping` of the class `Tracing` (see Listing 4.10) is responsible to store the trace link into the two hash maps `sourceToTargetMap` and `targetToSourceMap`. This method has to be called as soon as the target object is generated by a rule class. Therefore the interface `Rule` of jTL comes up with a default method also called `recordMapping`, to forward the call to the method `Tracing.recordMapping`. It takes the source-, the target object as well as the transformation rule as input to transfer this information into the hash maps.

Resolve Methods

Resolve Function in QVT	Resolve Function in jTL
resolve	getTarget
resolveIn	getTargetByRule
-	getTargetByTargetType
resolveOne	getOneTargetByTargetType
invresolve	getSource
invresolveIn	getSourceByRule
-	getSourceRuleTuple

Table 4.1 Resolve functions in QVT [29] and jTL

In QVT there are several proposed resolve-methods [3] [29]. Table 4.1 lists the most important resolve-methods of QVT that have corresponding methods in jTL, plus two additional methods that are specific to jTL. Also SiTra consists of similar methods to resolve the results of tracing, except the issue that all inverse-resolve methods are not implemented yet in SiTra. The following enumeration, describes the resolve methods of jTL in more detail. The implementation of them can be found in Listing 4.10.

1. `resolve` vs. `getTarget`:
   The counterpart of QVT's function `resolve` in jTL, has exact the same functionality. They both return a collection of target objects. These objects are the result of an recent transformation [3]. The rule type which was responsible for the transformation is not taken into account. This means, if one source object was transformed to different target objects by different rule types, then this method return all target objects which were generated from the source object.

2. `resolveIn` vs. `getTargetByRule`:
   Again these two methods have the exact same behaviour. In contrast to the previous mentioned methods, these methods take also the rule type into account. In QVT as well

as in jTL the user passes not only the source object but also the rule type. These methods then return only the target object which were transformed out of the source object by the specified rule type.

3. `getTargetByTargetType`:
   This method is specific to jTL. The user passes the source element and the desired type of the target type. In the end this method returns all target objects that have the requested target type and were transformed out of the source object. Since one source element can be transformed by different rule and nevertheless results in the same target type, this method can of course return several target objects.

4. `resolveOne` vs. `getTargetByTargetType`:
   These two functions have a similar behaviour. Like the above mentioned method `getTargetByType`, they take the source element and the target type as input parameter. Instead of returning all found target object, these methods return only the first found target object that fulfill the requirements.

5. `invresolve` vs. `getSource`:
   QVT's `invresolve`, as well as its corresponding method `getSource` in jTL, take a target object as input and return the source object.

6. `invresolveIn` vs. `getSourceByRule`:
   Similar as above mentioned methods, but the user also has to specify the mapping/rule-type which was used to for the transformation to the target object. The source object is only returned if the target object was transformed by the specified rule type.

7. `getSourceRuleTuple` This specific jTL resolve-method takes a target object as input and returns not only the source element, but also the rule type which was used for the transformation, to provide the user additional information about the transformation process.

```
public class Tracing {
    private static Tracing trace;
    private Map<SourceElement, Map<Class<Rule<? extends SourceElement, ?, ?>>, Object>>
         sourceToTargetMap;
    private Map<Object, Tuple> targetToSourceMap;


    public synchronized Optional<Collection<Object>> getTarget (SourceElement source) {
        if (sourceToTargetMap.containsKey(source))
            return Optional.ofNullable(sourceToTargetMap.get(source).values());
```

```
        else return Optional.empty();
    }
    public synchronized Optional<Object> getTargetByRule (SourceElement source, Rule<?
        extends SourceElement, ?, ?> rule) {
        if (sourceToTargetMap.containsKey(source))
            return Optional.ofNullable(sourceToTargetMap.get(source).get(rule.getClass
                ()));
        else return Optional.empty();
    }

    public synchronized Optional<Object> getOneTargetByTargetType (SourceElement source
        , Class<?> type) {
        if (sourceToTargetMap.containsKey(source)) {
            ArrayList<Object> list = new ArrayList<Object>(sourceToTargetMap.get(source
                ).values());
            return list.stream().filter(o -> o.getClass()==type).findFirst();
        }
        else return Optional.empty();
    }
    public synchronized Optional<List<Object>> getTargetByTargetType (SourceElement
        source, Class<?> type) {
        if (sourceToTargetMap.containsKey(source)) {
            ArrayList<Object> list = new ArrayList<Object>(sourceToTargetMap.get(source
                ).values());
            return Optional.of(list.stream().filter(o -> o.getClass()==type).collect(
                Collectors.toList()));
        }
        else return Optional.empty();
    }
    public Optional<SourceElement> getSourceByRule(Object target,  Rule<? extends
        SourceElement, ?, ?> rule) {
        if (targetToSourceMap.get(target).getRuleType()==rule.getClass())
            targetToSourceMap.get(target).getSource();
        return Optional.empty();
    }
    public Optional<SourceElement> getSource(Object target) {
        return Optional.ofNullable(targetToSourceMap.get(target).getSource());
    }
    public Optional<Tuple> getSourceRuleTuple(Object target) {
        return Optional.ofNullable(targetToSourceMap.get(target));
    }

    private Tracing () {
        sourceToTargetMap = new HashMap<SourceElement, Map<Class<Rule<? extends
            SourceElement, ?, ?>>, Object>>();
        targetToSourceMap = new HashMap<Object, Tuple>();
    }

    public static Tracing getTrace() {
        return trace==null? trace = new Tracing ():trace;
    }
```

```
    public synchronized void recordMapping (SourceElement source,Rule<? extends
        SourceElement, ?, ?> rule,Object target)  {
        Class<Rule<? extends SourceElement, ?, ?>> ruleType = (Class<Rule<? extends
            SourceElement, ?, ?>>) rule.getClass();
        if (!(sourceToTargetMap.containsKey(source) && sourceToTargetMap.get(source).
            containsKey(ruleType))) {
            Map<Class<Rule<? extends SourceElement, ?, ?>>,Object> innerMap;
            if (sourceToTargetMap.containsKey(source)) {
                innerMap = sourceToTargetMap.get(source);
            }
            else innerMap = new HashMap<Class<Rule<? extends SourceElement, ?, ?>>,
                Object>();
            innerMap.put(ruleType, target);
            sourceToTargetMap.put(source, innerMap);
            targetToSourceMap.put(target, new Tuple(source, ruleType));
        }
    }

    public class Tuple {
      private SourceElement source;
      private Class<Rule<? extends SourceElement, ?, ?>> ruleType;
      //..... Constructor and Getter
    }
}
```

Listing 4.10 First part of Tracing Class - get-methods

4.9 Rule Inheritance as reuse mechanism

In the area of model transformation reuse mechanisms are indispensable to support development productivity, maintainability, readability as well as quality of transformation [20]. Rule inheritance is one of the most important reuse mechanism and therefore should also be applicable in jTL. Unfortunately this is not as straight forward and easy to implement as for example in ATL, since of the down casting problem in Java (see Section 4.9.1). Nevertheless a good trade off has been found in jTL to enable this reuse mechanism for the user

To demonstrate rule inheritance in jTL a simplified version of the clazz-to-entity transformation example is used, which was already presented in the introduction (see Figure 1.1) and is also based on the example taken from the article [20] (see Figure 4.5).

To simplify the example even a bit more only the components NamedElement and Class were taken out to transform it to ModelElement and Entity. Figure 4.6 shows the implementation in jTL, consisting of the two source elements, the super type `NamedElement` and the subtype `Clazz` and the two corresponding target elements, that are the super type `ModelElement` and

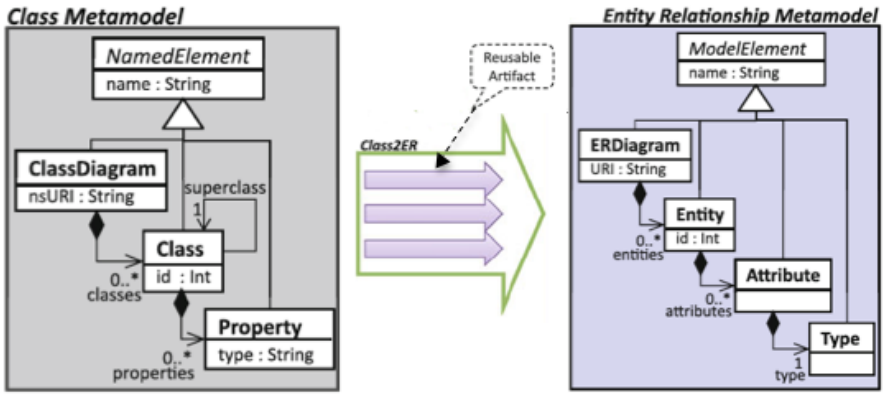

Figure 4.5 Transformation example for rule inheritance reuse mechanism [20]

the subtype `Entity`. According to the transformation link between the `NamedElement` and `ModelElement` as well as `Clazz` and `Entity` two rule-types have been implemented that are also in an inheritance hierarchy. The super rule type `NamedElem2ModelElem` transforms the name of `NamedElement` to the name of the `ModelElement`, while the sub rule type `Clazz2Entity` transforms the id of a `Clazz` to an id of `Entity`. As soon as a source object from the type `Clazz` should be transformed the rule inheritance reuse mechanism is used. This means that not only the rule `Clazz2Enity` is used for actual transformation, but also the super type `NamedElem2ModelElem` is called during the transformation process (see next Section 4.9.1).

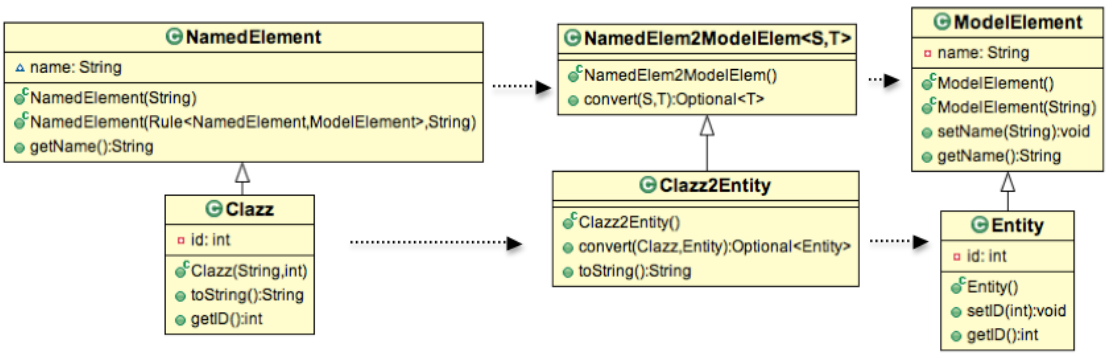

Figure 4.6 Example for rule inheritance in jTL

4.9.1 Possible Approaches fo Rule Inheritance

The principle of rule inheritance is to reuse transformation code from super type rules without the need of reimplementation. So a sub-rule like `Clazz2Entity` should only execute transforma-

tion work that is specific to a `Clazz` and should delegate all general transformation work that is applicable to the general type `NamedElement` to the super-rule `NamedElem2ModelElem`.

Downcast Problem

The problem with rule inheritance in jTL was the challenge to overcome the downcast problem in Java. In jTL every rule takes a specific source object and transform it to the final version of the target object. According to the example of Figure 4.6, this results in a generation of a target object `ModelElement` when using the super-rule type`NamedElem2ModelElem` and a generation of `Entity` when using the sub-rule type `Clazz2Enity`. This is of course the right and desired solution when rule inheritance is not used at all.
But as soon as rule inheritance comes into play this approach has some limits. Back to the example of Figure 4.6, a call of `NamedElem2ModelElem.convert` from the sub-rule type `Clazz2Entity` will result in returning a target object of type `ModelElement` to the sub-rule type `Clazz2Entity`. Since a downcast of a target type `ModelElement` to an `Entity`, does not work in Java, but further transformation activities have to be carried out on an object of `Entity` and not on an object of type `ModelElement`, a solution had to be found to overcome this problem. Three conceivable solutions were identified, weighed and compared:

1. Approach - 'Conditional instantiation': For this solution approach the super-rule type could check the source object type and already generate the final type of the target object. According to the example of Figure 4.6, the `NamedElem2ModelElem` would have to check if the incoming source object is from the subtype `Clazz` and consequently would have to generate a target object from the subtype `Entity`, that further is returned to the sub-rule `Clazz2Entity`, which then could executed further transformation actives on the `Entity`-object. Otherwise, if the source object is from the super type object `NamedElement`, the rule `NamedElem2ModelElem` would generate a target object from the general type `ModelElement`.
   Although this solution would work somehow, there are some weaknesses respectively imperfections. On the one hand the developer has to check every possible source object sub type in the super rule type to generate the right target object type, which can result in a considerable effort. On the other hand this solution would violate the separation of concern principle, because as soon as a new sub type for the source object as well as the target object is added, also the super rule type has to be adapted by adding a new if-statement to it. Because of this weakness this solution was discarded.

2. Approach - 'Copy constructor': This solution can be seen as workaround for the down casting problem by adding something similar like a copy constructor to the target sub type. According to example of Figure 4.6, the sub rule type `Clazz2Entity` would call the super rule type `NamedElem2ModelElem` and would get back a target object from the super type `ModelElement`. The target type `Entity` should further provide a constructor that takes a `ModelElement`-object. The returned target object could then be 'downcasted' from the super type `ModelElement` to the desired sub type `Entity` by calling this constructor.
   Although this approach is a progress in contrast to the 'Conditional Instantiation' approach, the developer is still faced with a considerable increase of implementation work, since every target sub rule type would have to hold an applicable copy constructor for downcasting.

3. Approach 'Passing the target type': The concept of this solution is based on the idea, that the sub rule type already creates an object of the final sub target type and passes it to the super type. The automatic upcast ensures that the super type rule can handle the already created target object and can execute further configuration on this target object. Afterwards the super rule returns the further configured target object back to the sub-rule, which is able to downcast the object again and executes its own transformation code. Although this solution is not as straight forward than the rule inheritance mechanism of ATL, it is still the best solution with the lowest development effort and an approach which fulfill the concept of 'Separation of concern'.

After weighing all pros and cons of the different rule inheritance approaches, the rule inheritance mechanism of jTL was implemented by using the approach 'Passing the target type'. The decisive factors to choose this approach were that this approach fulfills the requirement of object oriented programming best and that it comes with low development effort.

4.9.2 Rule inheritance by example ('Passing the Target Type' approach)

To enable rule inheritance in jTL, by using the the approach 'Passing the Target Type' some precondition has to be fulfilled.

1. **Interface `Rule`:** First of all the interface `Rule` has to provide a convert method which enables the user not only to pass the source element but also an already created target object. This is the reason why the `Rule` interface of jTL comes up with two different signatures of the method `convert`. The first one which only takes the source element

is implemented as a default method, and delegates the call to the second version of the `convert`-method, by passing `null` as reference for the target object. This indicates that a target object has not yet been created and that the convert method is responsible for the creation of the object. So the simpler version of the `convert`-method is the means of choice if rule inheritance is not used, and prevents the `Transformer` or the user of passing `null` every time he calls a rule directly.
The second version of the method `convert`, which takes also the target instance as input has to be called when using rule inheritance. Therefore the sub-rule additionally passes the already created target object to the super-rule.

2. **Super rule:** A rule which is intended to be a super rule, has to check the additional parameter 'target'. As shown in Listing 4.11, the super-rule `NamedElem2ModelElem` checks first whether a target object was already created or not. If the target object is null, then the super rule wasn't called by an according sub-rule and therefore creates the super type `ModelElement`. Otherwise, if an already created target object has been passed from the sub-rule `Clazz2Entity` to the super-rule`NamedElem2ModelElem`, further configurations are applied directly on the passed target object, without instantiating a new one.

```
public class NamedElem2ModelElem<S extends NamedElement, T extends ModelElement>
    implements Rule<S, T, Object>{
    @Override
    public Optional<T> convert(S source, T target, Optional<Object> parent) {
        if (target==null)
            target = (T) new ModelElement();
        target.setName(source.getName());
        return Optional.of(target);
    }
}
```

Listing 4.11 NamedElem2ModelElem - Rule Supertype

3. **Sub-rule:** To enable rule inheritance in the sub-rule, only one additional line has to be inserted. This line consists of the explicit call of the super rule and the downcast of the returned object. As shown in Listing 4.12, after the instantiation of the sub target object from the type `Entity` and the logging of the trace link, the `convert`-method of the super rule is called by passing the already created target object. Since the super `convert`-method executes an upcast to the supertype `ModelElement`, the return value has to be downcasted again to the subtype `Entity`. After that further configuration to the resulting `Entity`-object could be executed.

```
public class Clazz2Entity extends NamedElem2ModelElem<Clazz,Entity> {
    @Override
    public Optional<Entity> convert(Clazz source, Entity target, Optional<Object>
        parent) {
        target = new Entity();
        recordMapping(source, target);
        target = (Entity) super.convert(source, target, parent).get();
        target.setID(source.getID());

        return Optional.of(target);
    }
}
```

Listing 4.12 Clazz2Entity - Rule Subtype

4.10 Summary

This chapter described in a detailed way the general architecture of jTL as well as each component in an abstract manner. Further the concepts of tracing and rule inheritance and how they have been implemented in jTL were explained. The next Chapter 'Evaluation' 5, emphasizes which constructs, methods and approaches was implemented in jTL to fulfill the earlier defined main goals. A comparison between jTL, SiTra and PC2J-Transformer according to their characteristics should further demonstrates the differences and especially the improvements of jTL. Further the changes in execution time and performance of jTL in contrast to SiTra should be tested. Moreover two sample transformations should exemplary demonstrate the use of jTL. One is based on the well known graph based 'Class2ER' transformation and the second is based on a simple source code translation, starting from the abstract syntax tree.

CHAPTER 5

Evaluation

5.1 Goal Achievements of jTL

As described in the previous Chapter 4.1, three superior goals were defined as requirements for the jTL. These goals where 'Simplicity and Clarity', 'Flexibility' and 'Performance improvements'. According to this definition the following list emphasizes which requirements were implemented to fulfill these three basic goals.

1. Simplicity and Clarity
    - **Basic concepts already implemented:** All methods that come up with the core component `Transformer` are realized as default methods. This ensures that the developer does not have to think about the implementation, but can use the default methods of the interface.
    - **Rule selection:** With the concept of default rules, which are already referenced by the source element itself, the selection of the applicable rule is an easy task.
    - **Rule inheritance as reuse mechanism:** The architecture of jTL comes up with an approach of rule inheritance. In contrast to SiTra and PC2J-Transformer where this reuse mechanism is not possible this is an enormous improvement, especially for complex transformations.
    - **Simple Tracing:** In contrast to PC2J-Transformer where traceability is not part of the Transformer, and SiTra where traceability is not completely implemented yet, jTL already comes up with most of the resolve-methods that are proposed by the QVT Specification. The developer does not have to think about the implementation

of tracing, since all necessary methods for recording and resolving are part of the jTL.

2. Flexibility
    - **Specification of alternative rule:** Although in most cases the default rule implementation would be the method of choice for the transformation task, the developer always has the possibility to define a different one to transform a source element to satisfy also special cases. Since the interface `Rule` is a functional interface this can be done with the help of lambda expression, anonymous inner class or even an extra implementation class. While the PC2J-Transformer does not enable a possibility to define an alternative rule for the transformation of one and the same source object type, SiTra does provide this possibility. But in contrast to jTL, SiTra only enables this possibility by implementing an additional rule-class file. Since SiTra was developed before Java 8 was published and does not provide the interface `Rule` as functional interface, lambda expression can't be used for the definition of alternative rules.
    - **Generic return types:** In contrast to the PC2J-Transformer, that was defined only for source code transformation and therefore only allows the type `Codeable` from the framework jenesis4java as return type, jTL as well as SiTra support any kind of return types for the target object. There is no constraint on it. This ensures that any kind of transformation can be established by the jTL.
    - **Transformation of source models with cycles:** The default methods of the interface `Transformer` of jTL check already if a source element was already transformed. This ensures that the non-terminating recursive transformation problem can't occur, although the source model consists of cycles. SiTra also fulfills this requirement, while PC2J-Transformer is only accurate for tree transformations and therefore does not support cycle based graph transformations.
3. Performance improvements
    - **No time-consuming search for applicable rule:** Since every source element of type `SourceElement` provides already a link to the default rule there is no need to iterate over all known rules to find the right one for transformation of a source element. Especially when there is a huge amount of rules and source elements, such an iteration process for every single source element can be a very time consuming task (see test scenarios in Section 5.3.2 and Section 5.3.2). Nevertheless jTL provides additionally the possibility of specifying a different rule for the transformation (see previous goal 'Flexibility'). While PC2J-Transformer also provides a short rule

selection time by using a rule-repository based on a `HashMap`, SiTra relies on a time consuming iteration process for every single rule selection and source object.

- **Parallelism as standard implementation:** Especially since multi core CPUs are more or less a standard and model transformations soon get complex and time consuming, parallel transformation can easily lead to a significant improvement of the performance. Therefore the jTL already comes up with a standard implementation of parallel stream processing to transform several source elements at the same time. Since parallel stream processing is already implemented in the default methods of the interface `Transformer`, the developer does not have to take further steps and gets the parallelism nearly for free. Neither SiTra nor PC2J-Transformer support any kind of parallelism.
- **Fast Tracing:** Instead of using an implementation of the interface `List` to hold the trace links and force the application to iterate over all of them to find the designated one, like SiTra does, jTL comes up with two collections from the type `HashMap`. One maps source objects to target objects and another vice versa. This ensures a fast access, although a huge amount of trace links are already present in the hashmaps. The PC2J-Transformer does not support Tracing at all.

5.2 Comparison: jTL/Sitra/PC2J-Transformer

Table 5.1 summarizes the differences respectively the improvements of jTL in contrast to the existing solutions of SiTra and PC2J-Transformer. Furthermore this Table emphasizes that jTL combines the advantages of SiTra and PC2-Transformer and further adds some extra benefit like rule inheritance and parallelism to it that has whether been part of SiTra nor PC2J-Transformer.

5.3 jTL vs. SiTra - Comparison by Example

To compare jTL with SiTra, especially the usage of both libraries as well as difference in performance between them, a simple transformation example should be given. This example transforms a simple UML class diagram into an ER diagram. The meta models for this example was already presented in the introduction of this work (see Figure 1.1). These meta models shows an excerpt respectively a significantly simplified version of the UML meta model as well as the corresponding meta model for an entity relationship diagram.

Listing 5.1 demonstrates a concrete rule implementation called `Clazz2Table`, which transforms a source object of the type `Clazz` to a target object of the corresponding type `Table`. The equally named corresponding rule implementation, using SiTra as transformation library,

	jTL	SiTra	PC2J-Transformer
Definition of alternative Rule	yes	yes	no
Generic Return Type	yes (`Object`)	yes (`Object`)	no (`jenesis4java.Codeable`)
Rule Inheritance	yes	no	no
Transformation of Graphs with Cycles	yes	yes	no
Tracing	yes - partially QVT compliant	yes - fully QVT compliant but not completely implemented	no
Parallelism	yes - parallel Streams	no	no
Runtime - Rule Selection - Iteration steps needed (Worst Case with n Rules)	1	n	1
Runtime - Tracing - Iteration steps needed (Worst Case with n Tracelinks)	1	n	-

Table 5.1 jTL vs. SiTra vs. PC2J-Transformer

is shown in Listing 5.2. Two differences can be identified, when looking at the two different implementation classes. Firstly SiTra requires the implementation of three methods. While the method `check` does not need an equivalent in jTL, because of the different rule selection mechanisms (see Section 4.4), the further two methods `build` and `setProperties`, were combined to only one method called `convert` in jTL. On the one hand that was the precondition to design the interface `Rule` of jTL as a functional interface to further enable the use of lambda expressions for rule specification, on the other hand this approach requires the user to manually call the method `recordMapping` after creating the target object and before executing further transformation of referenced object. This is of course the second significant difference between jTL's and SiTra's rule implementation approach.

```
public class Clazz2Table implements Rule<Clazz, Table,
    Object> {

    @Override
    public Optional<Table> convert(Clazz source, Table
        target, Optional<Object> parent) {
        Table table = new Table(source.getName());
        recordMapping(source, table);

        List<Field> fieldList = new ArrayList<Field>();
        for (Attribute a : source.getAttributeList()) {
            fieldList.add((Field)getContext().getTransformer
                ().transform(a).get());
        }
        table.setFieldList(fieldList);
        return Optional.of(table);
    }
}
```

Listing 5.1 Rule Clazz2Entity - jTL

```
public class Clazz2Table implements Rule<Clazz, Table> {she

    @Override
    public boolean check(Clazz source) {
        return true;
    }

    @Override
    public Table build(Clazz source, Transformer t) {
        Table table = new Table(source.getName());
        return table;
    }

    @Override
    public void setProperties(Table target, Clazz source,
        Transformer t) {
        List<Field> fieldList = new ArrayList<Field>();
        for (Attribute a : source.getAttributeList()) {
            try {
                fieldList.add((Field)t.transform(a));
            } catch (RuleNotFoundException e) {
                e.printStackTrace();
            }
        }
        target.setFieldList(fieldList);
    }
}
```

Listing 5.2 Corresponding Rule Clazz2Entity - SiTra

5.3.1 Test Settings

Figure 5.1 demonstrates an example according to the meta model shown in Figure 1.1. It consists of all possible source objects, that are 'Clazz', 'Association', 'Attribute' and 'Type' and further of a cycle which should demonstrate the ability of SiTra and jTL to handle such constructs without running into the non-terminating recursive transformation problem. This example acts further as a basis for performance comparisons between SiTra and jTL to test the processing time of the transformation. Therefore the number of source objects as well as the number of rules are going to be increased to demonstrate the changes in transformation time. To get representative results, every test scenario was carried out 100 times in a row and further has been averaged. Since the warm up phase of the Java virtual machine (jvm) takes a while and would result in a significant longer transformation time and no meaningful time measurements, a initial warm up phase without measuring of the execution time is taken into account (see description of the test class in Section 5.3.4).

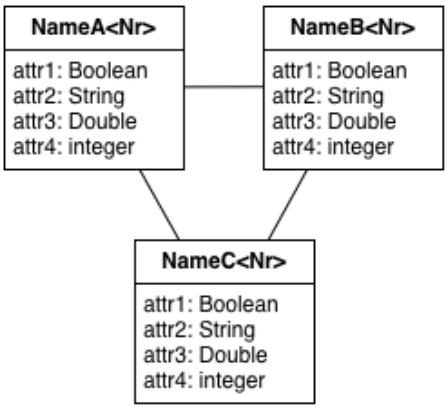

Figure 5.1 Simplified Class Diagram which is going to be transformed to a simple ER-Diagram

5.3.2 Sequential Test Scenarios

Table 5.2 emphasizes the execution time of the transformation of one and the same test scenario for SiTra and jTL. The used transformation method for jTL as well as SiTra was `transform` which executes every source object individually in a sequential mode. Therefore a loop passes one single source object after another to the `Transformer`-class which then calls the applicable rule. Further the following two variables were invented:

- **`m` - Number of source objects constructs:** The chosen source object construct was already demonstrated in Figure 5.1, which altogether consists of three class-objects, three associations and twelve attributes as well as twelve datatypes. Since transformation tasks

are mostly much more complex and consist of significantly more source object the number of these constructs (`m`) were increased stepwise, to achieve more representative results.

- **`n` - Number of rules:** Another important factor is the number of rules (`n`) that are implemented by the developer. This factor does not influence the execution time of jTL, since every source object in jTL knows already its default rule and does not rely on a time consuming iteration process to find the applicable rule (see Section 5.1), but it influences the transformation time in SiTra dramatically. Therefore a mock-rule called `DummyRule` was invented that does nothing but returning false by invoking the `check`-method. This tells SiTra that this rule is not the right one for the transformation of a source object. These mock-rules are used in the second test scenario (see) to investigate the influence on execution time by the presence of much more rules.

First Test Scenario - n=3

According to the example shown in Figure 5.1 this first test scenario consists of a fixed amount of three rules (n=3) without adding mock rules to the transformation task, whereas the amount of source objects is increased stepwise by the factor 10. Since the used constructs consist of altogether 21 source objects, the first test case sets `m` to 21.

	Transformation Time in microseconds (μs)		
m - Number of Source Objects	**SiTra**	**jTL**	**Relative Share** jTL / SiTra
21	105	11	10,77%
210	556	63	11,33%
2100	1063	353	33,21%
21000	2913	1202	41,26%
210000	29913	13137	43,92%

Table 5.2 Average execution time (in micro-seconds) of jTL compared to Sitra with small n (after 10 runs)

Discussion: The results presented in Table 5.2, clearly state the performance increase of jTL in contrast to SiTra, nevertheless if a small or a higher amount of source objects has to be transformed. With a low number of source objects the performance increase is enormous, since the transformation task in jTL lasts only a tenth of the transformation time of SiTra. If the number of source objects is increased, the gap in execution time for the transformation decreases but still the transformation with jTL is twice as fast as it is by using SiTra. The reason for the decreasing difference in execution time between SiTra and jTL lies in the optimization which is carried out by the jvm. Since, in constrast to jTL, SiTra longs for additional method calls like

the methods `check()` and `setProperties` the optimization of the jvm affects the execution time of SiTra in a more significant way.
For the explanation of the significant speed up by using jTL two reasons can be identified. First the slow rule-selection process by iterating over the whole rule repository, that affects the transformation time even if there are only few rule types in the rule repository and second, the additional method calls that are needed for transforming a source object. While jTL only calls one method named `convert`, SiTra longs for three method-calls, that are `check`, `build` and `setProperties`.

Second Test Scenario - n=103

This test scenario focuses on cases where the number of rules is much higher. Since for example a transformation of a Cobol file needs slightly more than 100 hundred rules, this test scenario sets n to 103. To achieve SiTra's average search time to find the applicable rule, the first 50 rules in SiTra's rule repository are mock-rules from the type `DummyRule`, followed by the three rule types `Clazz2Table`, `Association2Relation` and `Attribute2Field`, that are necessary for the transformation. To be complete further 50 mock-rules are added at the end of the rule repository. The number of source objects (`m`) is again increased stepwise by the factor 10. The results of these test scenarios are represented in Table 5.3

	Transformation Time in microseconds (μs)		
m - Number of Source Objects	**SiTra**	**jTL**	**Relative Share jTL/SiTra**
21	604	18	2,98%
210	1469	30	2,04%
2100	3781	161	4,26%
21000	33095	1272	3,84%

Table 5.3 Average execution time (in micro-seconds) of jTL compared to Sitra with high n (after 10 runs)

Discussion: The results in Table 5.3 demonstrate the enormous improvement of execution time by using jTL when a higher number of rules is necessary. In contrast to SiTra, jTL only takes 2%-4% of the execution time that would be needed for the same transformation task by using SiTra. At the same time this test scenario emphasizes the greatest weakness of SiTra, that is the rule selection process, which is only suitable when a very small number of rules is needed for the transformation task. But for more complex transformations, for instance for source code transformation, this approach is quite inappropriate.

5.3.3 Parallel Test Scenarios

Third Test Scenario - Parallel Execution

The following test scenario should emphasize the changes in execution time when using parallel stream processing instead of using external iteration. Since SiTra does not support parallel transformation the execution time of jTL's parallel transformation can only be compared to the sequential execution of jTL. As the results in Table 5.2 and Table 5.3 of the previous two sections already pointed out, the sequential execution of the transformation with jTL is much faster than the transformation with SiTra. So the following test scenarios should demonstrate whether there could be a further improvement in execution time when using jTL's parallel processing and how this changes the time benchmark. To test the parallel transformation, the method `transformAll` of jTL's `Transformer` is used, which takes a list of source objects and uses parallel stream processing to transform each of the passed object in the list parallel. To clearly state the difference in performance between sequential and parallel processing the test setting is changed a little bit, to achieve the transformation of totally independent objects. On the one hand parallel stream processing will perform better if the source objects do not depend on each other, but on the other hand the source code transformation will always end up in transforming independent source object since the AST does not contain cycles. For these test scenarios 10 independent class-object are generated, without any association between them and each of these class-objects have 4 attributes. The number of class-objects is then again increased stepwise by the factor 10. Although the number of rules are not significantly important for the execution time of jTL, it should be mentioned that again three rule types are used for the transformation.

	Transformation Time in microseconds (μs)		
m - Number of Clazz objects	**jTL - sequential**	**jTL - parallel**	**Relative Share sequ./par.**
10	40	235	587,5%
100	183	229	125,14%
1000	885	871	98,42%
10000	6486	2355	36,31%
100000	77612	28930	37,28%

Table 5.4 Average execution time (in micro-seconds) - Sequential use of jTL vs. Parallel use of jTL

Discussion: The result Table 5.4 shows, that the effort of parallel transformation is only profitable if a high number of source object exists. Solving a problem in a parallel way always involves performing more actual work than doing it in a sequential way. There are some overhead tasks to do, that involves splitting the work between several threads and joining and merging

the results. So the number of source objects has to be quite high or the actual transformation of the source object has to be a complex and time intensive job. Only if at least one of these requirements is fulfilled the execution time benefits from using parallel execution. According to this test scenario, the breakeven point, with a quite simple transformation task, is reached when passing about 1000 source object to the transformation method `transformAll` of jTL. But as soon this limit is exceeded the time effort for transformation decreases dramatically and end up in a time effort that is only one third of the sequential processing time.

5.3.4 Test Class

The test class which is designated to execute the above presented test scenarios can be found in the appendix (see Listing A.5). It consists of following four phases:

1. Configuration phase (before the test class is executed): Before the test class is used to start the test scenarios some configurations are necessary. The user can set the following parameters:
   - WARM_UP_RUNS: In order to get comparable and meaningful execution time results, the test class has to perform a JVM (Java virtual machine) warm up (see warm up phase below). With this parameter the user can define the number of iterations for this phase.
   - RUNS: Another important factor to achieve meaningful and comparable results for the execution time is to measure the transformation time of several runs of the same transformation and average them. This ensures that outliers do not distort the measurement results. The higher this parameter is set, the more meaningful the results are, but also the more time is required to perform the different test scenarios.
   - SOURCE_OBJECT_CONSTRUCTS: For generating the source objects the two external classes `JTLGenerator` and `SitraGenerator` are used. By setting this parameter, the number of constructs which has to be created by these classes can be determined by the users.
   - DUMMY_RULE: This parameter is only used by SiTra test scenarios. Since the execution time of the transformation with SiTra strongly depends on the amount of rule types, this parameter sets the number of dummy rules which should be added to the rule repository of SiTra.
2. Setup phase

- Setup of SiTra test cases - `setupSitraTest()`: This method consists of adding the rule types as well as the dummy rule types to the rule repository of SiTra. Further the source objects, which are going to be transformed, are generated here, by using a further external generation class called `SitraGenerator`.
- Setup of jTL test cases - `setupJTLTest`: This method is responsible for generating the source objects only, by using the generation class `JTLGenerator`.

3. Warm up phase - `warmUp()`: As mentioned above this phase is very important to get comparable and meaningful execution time results. If a collection of one and the same source object is transformed several time the execution time is getting lower until a certain threshold is reached. So the user sets the parameter WARM_UP_RUNS to determine how often a transformation should be carried out, until the execution time is going to be measured. Several trials showed that about four warm-up-runs are enough in most cases.

4. Test phase: The last phase is the actual execution of the different test scenarios. This is done by calling the method `startTest(int kind,...)` (see Listing 5.3), which serves as distributing method. The parameter `int kind` is used to select the desired transformation method, like jTL's sequential or parallel transformation or SiTra's sequential transformation. Moreover this method carries out the actual measurement of the execution time. The measurement-results of several runs per test scenario are averaged and returned.

```
public long startTest(int kind, ArrayList<SourceElement> jTLSourceObjs, List<
    Object> sitraSourceObjs)
```

Listing 5.3 Method for test scenario selection

5.3.5 jTL vs. SiTra Conclusion

When looking at the performance benchmarks of jTL and SiTra the results demonstrate the enormous improvements when using jTL. No matter how many source objects have to be transformed or how many rules have to be implemented to fulfill the actual transformation task - the improvement in processing time remains on a high level. In every test scenario this results in an improvement of more than 50%. The more complex a transformation task becomes and the more rule types are necessary, the greater the gap between the execution time of jTL and SiTra gets. Since source code transformation are such complex transformation tasks and will certainly end up in implementing more than 100 rules, the improvement in execution time for the trans-

formation in jTL is even more obvious. The results of test scenario int Table 5.3 show, that jTL only needs 2-4% of the execution that SiTra would need for the same transformation task.

Further Table 5.4 emphasizes, that also the parallel execution that comes up as one possibility in jTL, could lead to a further improvement in execution time. This strongly depends on two factors that are the number of source objects that has to be transformed and the complexity of each rule. If the amount of source objects exceeds about 1000 and simple rules are implemented, the execution time of parallel processing reaches the break even point and can drop to about 35%. Since Cobol files often consist of over more than 10000 statements, which further represent the source objects, the use parallel execution could be profitable in many situations. Moreover the complexity of rules normally is much higher in source code transformation than it has been established in this test scenario. So the break-even point would be reached earlier in source code transformation applications.

5.4 Source Code Transformation with jTL

The last Section of this chapter should demonstrate how jTL can be used to set up a source code transformation. Therefore the simple source code snippet in Listing 5.4 written in Pl/1, which consists of simple variable declaration statements, assignment statements and if-statements, should act as basis for a transformation example.
Further the framework jenesis4Java is used by jTL to generate the resulting Java source code in a very handy way, whereas jenesis4Java is a DOM (domain object model) of the Java programming language. In jenesis4Java the complete Java class including annotations and comments is constructed as a DOM until it is encoded to a Java file [42]. Since jenesis4java uses a DOM the parent target object has to be passed to a rule, so that the current target object can be appended to its parent object.

```
DCL varA DEC FIXED(15,0);
DCL varB DEC FIXED(15,0);

varA = 540;
varB = 0 ;

IF varA > 0 THEN
BEGIN;
    varA = varA / 100 * 90;
    IF varA > 100 THEN
        varB = varA;
END;
ELSE varA = varB;
```

Listing 5.4 Simple Pl/1 source code

5.4.1 Preparation Steps

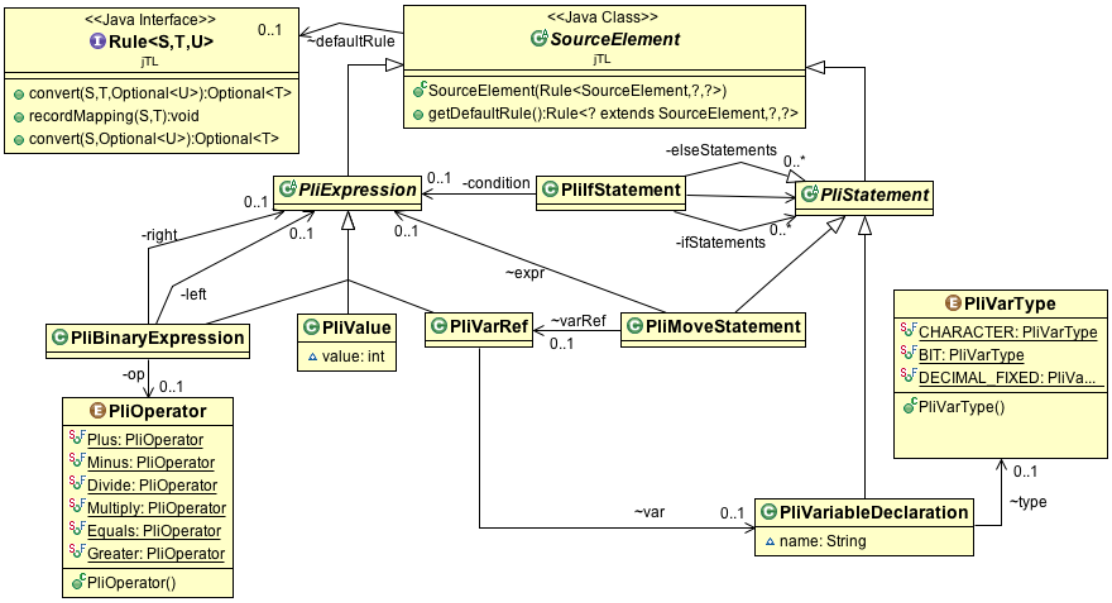

Figure 5.2 Small excerpt of a PL/1 Meta-Model incl. link to jTL

A process model which is designated to generate a source model respectively an AST from source code was already explained in Section 2.2.3. According to this process model, the first step is to define a meta model. The meta model which satisfies the need for the small Pl/1 example code from Listing 5.4 was already invented in the introduction of this work (see Figure 1.2). Figure 5.2 extends the meta model by the jTL specific classes `SourceElement` and `Rule`. Since `PliExpression` and `PliStatement` are only super types and can't be instantiated directly they are both defined as abstract classes.
Figure 5.2 further demonstrates the association between jTL and the meta model, which is recognizable through the inheritance between the jTL class `SourceElement`, which acts as super type, and all the statement and expression objects of PL/1. Moreover the relationship between the class `SourceElement` and the jTL interface `Rule` shows, that every source object has to have a link to its applicable rule.

All the other tasks of the process model 2.2.3, like pre-processing, parsing, optimizing or object generation, aiming at generating the actual source model. Since the implementation of this process model is not the scope of this work, the resulting source model was generated manually (see 5.4.4). Nevertheless the output would be pretty the same if it is generated automatically by implementing the process model. The resulting source model, which is an abstract depiction of the PL/1 source code Listing 5.4 and consists of all necessary information for the transformation,

is represented by an AST. That AST has to be compliant to the previously defined meta model (see Figure 5.2). For the sake of clarity the last if statement of Listing 5.4 is only merely hinted in Figure 5.3. Apart from the root-node, which acts as imaginary starting point, all the source objects are compliant to the meta model in Figure 5.2 and together constitutes the source object model which is the input for the actual transformation with jTL.

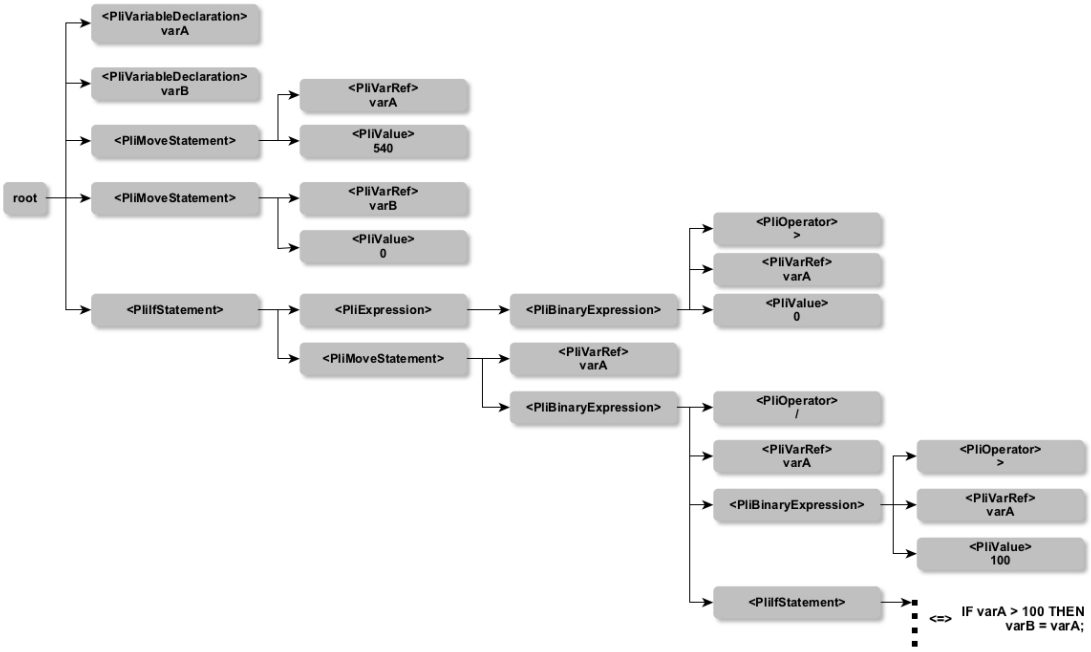

Figure 5.3 Abstract Syntax Tree of the PL/1 source code of Listing 5.4

5.4.2 Actual Transformation with jTL

After all preparation work is done, the implementation of the transformation can be established. Figure 5.4 shows the most important parts of the transformation. The rules which hold the actual transformation instructions and represents the core of the transformation are all derived from the jTL interface `Rule`. As it can be seen in Figure 5.4, every source element type, defined in the meta model (see Figure 5.2), has its corresponding rule type except the two abstract classes `PliStatement` and `PliExpression` as well as the enumeration types `PliVarType` and `PliOperator`, which do not long for an own rule implementation. This results in defining six rules, where for example the rule `PliBinaryExpressionRule` is responsible for the transformation of source object types `PliBinaryExpression`. To get an idea how such a rule can be implemented in jTL to generate Java source code, Section 5.4.3 explains the `PliMoveStatementRule` in more detail.

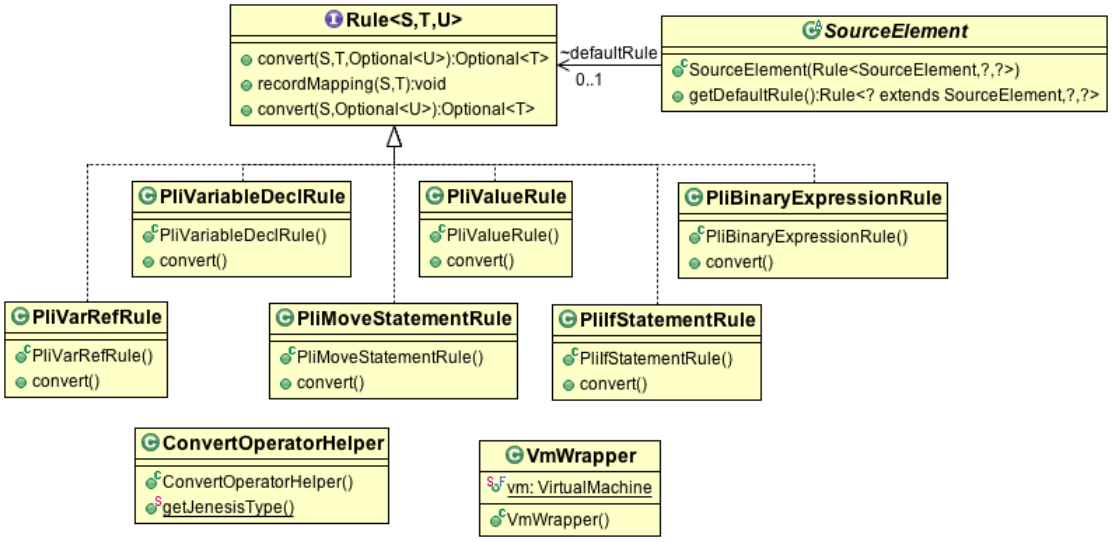

Figure 5.4 Rules for the transformation of Pl/1 source objects

For the reason of clarity, only the two components `Rule` and `SourceElement` of jTL, which have a direct linkage to the implemented rules respectively the source objects, are represented in Figure 5.4. Nevertheless there are other components of jTL like `Transformer` or `TransformerContext` that have to be part of the transformation (see jTL Architecture in Figure 4.1), except the `Tracing`. Since an AST does not have cycles and does not contain one and the same source object more than once, there is no need for Tracing. Nevertheless it is up to the developer to decide whether to make use of tracing or not.

Figure 5.4 further shows the two classes `ConvertOperatorHelper` and `VmWrapper` which both intended to be helper classes. The `ConvertOperatorHelper` holds a mapping between Pl/1 operators and the corresponding Java operators, respectively the jenesis4Java representations of them, and is needed by rules which transform expressions such as the `PliBinaryExpressionRule`. The class `VmWrapper` holds information, which is needed by the framework jenesis4Java, such as an instance of the type `VirtualMachine` to generate source code, or the current source code block where the generated source code has to be included.

5.4.3 Rule Implementation

This Section briefly explains the concrete implementation of a rule, with the aid of an example, that is `PliMoveStatementRule`. As shown in Listing 5.5 this rule, which is derived from jTL's `Rule`-interface, takes a `PliMoveStatement`-object and transforms it to an object of jenesis4java's container interface `Codeable`.

```
import jTL.*;
import static jTL.TransformerContext.getContext;
import net.sourceforge.jenesis4java.*;
import sourceObjects.PliMoveStatement;
import static utils.VmWrapper.vm;
import java.util.Optional;

public class PliMoveStatementRule implements Rule<PliMoveStatement, Codeable, Block>{
    @Override
    public Optional<Codeable> convert(PliMoveStatement source, Codeable target,
        Optional<Block> parent) {
        Variable var = (Variable)getContext().getTransformer().transform(source.
            getVarRef(), parent).get();
        Expression expr = (Expression)getContext().getTransformer().transform(source.
            getExpr(), parent).get();
        Assign assign = vm.newAssign(var, expr);
        parent.orElseThrow(()->new RuleTransformationException(source,"No parent object
             was defined."));
        parent.ifPresent(x -> x.newStmt(assign));
        return Optional.of(assign);
    }
}
```

Listing 5.5 Simple Pl/1 source code

The actual transformation takes place in the method `convert` and consists of the following steps:

1. Generation of the left hand side of the move-statement: To transform the left hand side, which has to be a variable reference, the jTL class `Transformer` is called, which further calls the rule `PliVarRefRule` and returns the already transformed variable reference as Java/jenesis4java variable reference.

2. Generation of the right hand side of the move statement: Therefore a further transformation of the expression which is represented by the source object `PliExpression` has to take place. So this source object is passed to the `Transformer` interface of jTL, which then calls the applicable rule for the transformation of the right hand side expression, that is the rule type `PliBinaryExpression`, and returns the target expression back to the `PliMoveStatementRule`.

3. Generation of an assign-statement which consists of the left hand side variable and the right hand side expression.

4. Checking if the parent target object was passed to the Rule. If not a `RuleTransformationException` will be thrown.

5. Attaching the assign-statement to the parent object respectively to the jenesis4java DOM.

6. Returning the assign-statement

All the other rule types have a very similar implementation-concept to fulfill the transformation requirements. Therefore the listings of only two further rules, that are `PliIfStatement` and `PliBinaryExpression` (see Listing A.7 and Listing A.8), are demonstrated in the appendix.

5.4.4 Test Class

This Section should shortly introduce the class which tests the transformation. The source code Listing, including a few comments for the reason of explanation, of this class can be found in the appendix A.6. As previously mentioned the source model (AST) of the PL/1 source code Listing is generated manually in this work. Therefore the test class consists of the following three methods:

1. **`generateSourceObjects()`**: This method is responsible for generating the complete AST to the corresponding PL/1 source Listing 5.4, as it is demonstrated in Figure 5.3, .Therefore it generates all corresponding statements, like variable declarations, variable initialisations, if-statements, etc. as object tree. This object tree further serves as input to the method `startTransformation`.

2. **`generateSkeleton()`**: For sake of clarity and simplicity the transformation example consists only of a small PL/1 source code snippet. So there is no surrounding function for the PL/1 code, which has to be transformed. Nevertheless, in order to get a runnable Java class as output, a skeleton class file is generated as jenesis4java-DOM before the actual transformation starts. Therefore this method creates a public class with a `main`-method, to act as container for the transformed Java source code.

3. **`startTransformation(List<PliStatements>)`**: This method starts the actual transformation and adds the transformed jenesis4Java objects to the DOM. After the transformation is done the DOM is encoded to a Java class file.

5.4.5 Transformation Output - Java Class

Finally this last Section demonstrates the output of the small transformation example. Listing 5.6 should give an idea how the transformed PL/1 Code of Listing 5.4 could look like.

```
public class GeneratedJavaClass {
    public static void main(String[] args) {
        int varA = 0;
        int varB = 0;
        varA = 540;
        varB = 0;
        if (varA > 0) {
            varA = varA / 100 * 90;
            if (varA > 100) {
                varB = varA;
            }
            else {
                varB = varA;
            }
        }
        varA = varA / 100 * 90;
    }
}
```

Listing 5.6 Generated Java Source Code

CHAPTER 6

Conclusion and Future Work

The last Chapter of this work is devided into two sub sections. The first one summarizes this thesis and its results. The second Section 'Future work' should emphasize additional possible improvements of jTL that are not already part of it right now. Furthermore it should give an overview about further challenges that a developer has to face when setting up a source code transformation project in general and in particular with jTL. These challenges like for example goto eliminations or clustering are scientific efforts, that are still not completely answered and could therefore be topics for further scientific works, which may be based on jTL.

6.1 Conclusion

Since model-driven software engineering has become an important field of software development, also model transformation technologies have developed to a very important task. This leads to a wide range of different model transformation languages (MTLs) and tools that are available to support this process. ATL, Kermeta or QVT based languages are only a few examples for model transformation languages. Although most of them perfectly fit to traditional model transformation tasks each of them requires to learn a completely new language. Further the declarative approach, which is used by ATL, or the graph based approach, which is used by TTG, are approaches that have some merits, but require for many developers a new way of thinking, since they are mostly confronted with imperative languages in daily business. Last but not least, the implementation of complex transformation, like source code transformation, are much more intuitive to write with an imperative language. So the idea of this work was to provide a Java based transformation library that provides a simple framework to the developer

to set up wide range of possible transformation tasks. Such a Java based imperative approach enables the user to set up a transformation in a well known language without the need of learning a new one. Moreover jTL provides also a very good basis for much more complex source code transformation projects.

Although pure Java based transformation approaches are very rare, two of them, that are SiTra and PC2J-Transformer, have been found and evaluated. The strengths and weaknesses of these approaches have been determined, just to profit from them when designing and implementing the jTL. Moreover these insights were used as a basis to build upon goals that should be focused on, when building the jTL. Further, to let jTL represent a state of the art library that profit from the new features of Java 8 the most important features like parallel stream processing, functional interfaces, lambda expression or the new interface `Optional` were determined and integrated. According to these facts the following enumeration summarizes the most important advantages of the resulting jTL:

- External Flexibility: jTL is applicable for a wide range of possible transformation tasks. No matter whether a small graph based transformation with cycles, or a complex abstract syntax tree transformation, like source code transformation, should be carried out.
- Internal Flexibility: Generic target types and the possibility of defining another than the default rule for a specific source object, support a flexibel transformation, even special cases should be addressed.
- Improved Performance: A significantly improved rule selection mechanism, in contrast to SiTra, a fast tracing implementation and the possibility of parallel stream processing, ensures a fast execution time of jTL, which is a very important factor if complex source code transformation should be carried out.
- Easy to use: A very simple rule selection mechanism supported by the definition of default rule, a simple `Transformer` that already comes up with a default implementation, an already implemented transformer method that allows the developer to switch to parallel execution and a ready use tracing approach ensures that the jTL is a complete and easy to use library for transformation tasks.

Finally the previous chapter 'Evaluation' has demonstrated two different transformation scenarios to test the jTL. An excerpt of a graph based 'Class-to-ER' transformation as well as a tree based 'PL/1-to-Java' transformation were established and served as examples to demonstrate the functionality and usage of the library and finally leads to the right and expected result output.

6.2 Future Work

6.2.1 jTL Extensions

Two features that have not been addressed in this work, since they are not a must for a complete transformation framework, but would further improve jTL are listed below.

- **Ecore integration:** Currently the developer who uses jTL, has to write the meta model as pure Java classes. Since ecore diagrams of the eclipse modeling framework (EMF) are a widely used approach to define meta models, a direct integration of an ecore model into jTL would be a convenient solution. This would enable the developer to design the meta model with EMF in a very handy way.
- **Change propagation by using tracing information:** Bonde [4] applies the traceability to change propagation, thus if the source changes slightly, the change can be reflected in the destination without re-running the whole transformation. Since Tracing is part of the jTL the basis for change propagation is present. Nevertheless, due the complexity of change propagation, no algorithm has been implemented that enables the use of change propagation.

6.2.2 Source Code Transformation

As pointed out in this work, automated software migration by source code transformation is a very challenging task. Until now there is now really satisfying product or solution that transforms legacy code like Cobol or PL/1 source code to state of the art programming languages. Although there have been many attempts, only few projects end up by replacing legacy code with the automated transformed new source code. The reason for that was the mostly bad quality of the resulting source code, that does not fulfill the requirements of good readability and maintainability. For example, characteristics like object orientation, encapsulation or separation of concern were not addressed properly. So the choice was to do a lot of refactoring work to improve and adjust the source code or to use the resulting source code as it is with all the disadvantages that come up with it.

So the challenge in the future will be the improvement of the automatic transformation process to increase the quality of the resulting source code. Therefore many different fields of research have to be addressed. The following enumeration should give a few example for that.

- **Goto Elimination:** The use of goto statements in legacy code is unfortunately widely used. Since state of the art programming language blessedly does not support this out-

dated and very problematic concept any longer, goto-constructs can't be replaced in an one-to-one manner. Since in many cases the resolve of goto-statements longs for different kinds of treatment, there has to be a bundle of different strategies to resolve such constructs.

- **Clustering and separation of concerns:** Legacy code is often represented by a large monolithic system architecture. The clustering into different functions or the separation into different source code files were often completely missing or replaced by confusing goto-constructions. Further the separation of concern concept wasn't addressed at all since it was not a well known paradigm at the time when the legacy code was written. So a one-to-one transformation of such code would not lead to satisfying output, since the result would be again a monolithic application, that do not fulfill the requirements of modern and for example object oriented software.

- **Automated testing:** Nevertheless how well the transformation process is designed, testing is a must when an automated software migration is performed. For sure manual testing has to be part of the test phase, but can't replace the need for an automated approach, since otherwise the test of every single test scenario would become an overwhelming task. So one further important scientific research area is the development of a process of automated testing. Such a process has to test the transformed source code as well as the legacy code to compare the results of them, to emphasize deviations.

As soon as these problems can be addressed in a satisfying way, software migration by automated transformation will become a very important concept to get rid of legacy applications.

APPENDIX A

Source Code Listings

A.1 Quantitive Comparison between external Iteration, sequential and parallel Stream Processing

A.1.1 Test Scenario 1

```
private static long testExternalIterator(Collection<String> names) {
    long count = 0;
    for (String name : names) {
                if (name.startsWith("A"))
                    ++count;
        }
        return count;
}

private static long testSequential(Collection<String> names) {
    return names.stream().filter(name -> name.startsWith("A")).count();
}

private static long testParallel(Collection<String> names) {
    return names.parallelStream().filter(name -> name.startsWith("A")).count();
}
```

Listing A.1 Test1 - Test Scenarios

A.1.2 Test Scenario 2

```
private static long testExternalIterator(Collection<Integer> numbers) {
    long count = 0;
    for (Integer number : numbers) {
            if (number%4==0)
                ++count;
    }
}

private static long testSequential(Collection<Integer> numbers {
    return numbers.stream().filter(i -> i%4==0).count();
}

private static long testParallel(Collection<Integer> numbers) {
    return names.parallelStream().filter(i -> i%4==0).count();
}

//Method to test the execution time of IntStream, to avoid auto boxing in parallel mode
private static long testIntStreamParallel(int[] numbers) {
    IntStream intStream = Arrays.stream(numbers);
    return intStream.parallel().filter(i -> i%4==0).count();
}
```

Listing A.2 Test2 - Test Scenarios

A.1.3 Test Scenario 3

```
private static long testExternalIterator(Collection<Integer> numbers) {
    for (Integer number : numbers) {
        for (int i=0; i<2000000; i++){
                    number++;
           }
    }
}
private static void testSequential(Collection<Integer> numbers) {
    numbers.stream().forEach(Benchmarking2::incr);
}

private static void testParallel(Collection<Integer> numbers) {
    numbers.parallelStream().forEach(Benchmarking2::incr);
}

private static void incr (Integer number) {
    for (int i=0; i<2000000; i++){
                number++;
        }
}
```

Listing A.3 Test3 - Test Scenarios

A.1.4 Test Scenario 4

```
private static void testExternalIterator(Collection<Integer> numbers) {
    for (Integer number : numbers)
        System.out.println(number);
}

private static void testSequential(Collection<Integer> numbers) {
    numbers.stream().forEach(System.out::println);
}

private static void testParallel(Collection<Integer> numbers) {
    numbers.parallelStream().forEach(System.out::println);
}
```

Listing A.4 Test1 - Test Scenarios - Blocking function

A.2 Test Class to compare the Transformation Time of SiTra & jTL

```
public class ComparisonTest {
    private jTL.Transformer jTLTransformer = getContext().getTransformer();
    private uk.ac.bham.sitra.Transformer sitraTransformer;

    private final static int WARM_UP_RUNS = 4;
    private final static int RUNS = 100;
    private final static int SOURCE_OBJECT_CONSTRUCTS = 1000;
    private final static int DUMMY_RULES = 100;

    public static void main (String args[]) {
        ComparisonTest test = new ComparisonTest();
        ArrayList<SourceElement> jTLSourceObjs = test.setupJTLTest();
        List<Object> sitraSourceObjs = test.setupSitraTest();
        long duration;

        test.warmUp(jTLSourceObjs, sitraSourceObjs);

        duration = test.startTest(0, jTLSourceObjs, sitraSourceObjs);
        System.out.println("Sitra-Indiviual:" + duration/RUNS);

        duration = test.startTest(1, jTLSourceObjs, sitraSourceObjs);
        System.out.println("Sitra-ALL:" + duration/RUNS);

        duration = test.startTest(2, jTLSourceObjs, sitraSourceObjs);
        System.out.println("jTL-Individual:" + duration/RUNS);

        duration = test.startTest(3, jTLSourceObjs, sitraSourceObjs);
        System.out.println("jTL-parallel:" + duration/RUNS);
    }
```

```
/** SETUP METHODS **/
public ArrayList<SourceElement> setupJTLTest() {
    //return JTLGenerator.generateConstructs(CONSTRUCTS);
    return JTLGenerator.generateClazzes(SOURCE_OBJECT_CONSTRUCTS);
}

public List<Object> setupSitraTest() {
    List<Class<? extends Rule<?,?>>> rules = new ArrayList<Class<? extends Rule
        <?,?>>>();

    //Adding half (DUMMY_RULES/2) of the mock rules before the actual rules
    //and adding the further half mock rules to the end of the list
    for (int i=0; i<DUMMY_RULES/2; i++)
        rules.add((Class<? extends Rule<?, ?>>) DummyRule.class);
    rules.add((Class<? extends Rule<?, ?>>) Clazz2Table.class);
    rules.add((Class<? extends Rule<?, ?>>) Association2Relation.class);
    rules.add((Class<? extends Rule<?, ?>>) Attribute2Field.class);
    for (int i=0; i<DUMMY_RULES/2; i++)
        rules.add((Class<? extends Rule<?, ?>>) DummyRule.class);

    sitraTransformer = new SimpleTransformerImpl(rules);
    //return SitraGenerator.generateConstructs(CONSTRUCTS);
    return SitraGenerator.generateClazzes(SOURCE_OBJECT_CONSTRUCTS);
}

/** TEST METHODS **/
public long startTest(int kind, ArrayList<SourceElement> jTLSourceObjs, List<Object
    > sitraSourceObjs) {
    long duration=0;
    for (int i=0; i<RUNS; i++) {
        long start = System.nanoTime();
        switch (kind) {
            case 0: testSitraIndividual(sitraSourceObjs);
                break;
            case 1: testSitraAll(sitraSourceObjs);
                break;
            case 2: testJTLIndividual(jTLSourceObjs);
                break;
            case 3: testJTLAll(jTLSourceObjs);
                break;
        }
        duration += (System.nanoTime()-start)/1000;
        //System.out.println((System.nanoTime()-start)/1000);
    }
    return duration;
}

public void testJTLAll (ArrayList<SourceElement> jTLSourceObjs) {
    jTLTransformer.transformAll(jTLSourceObjs);
}
```

```
    public void testSitraAll (List<Object> sitraSourceObjs) {
        try {
            sitraTransformer.transformAll(sitraSourceObjs);
        } catch (RuleNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void testJTLIndividual (ArrayList<SourceElement> jTLSourceObjs) {
        for (SourceElement temp : jTLSourceObjs)
            jTLTransformer.transform(temp);
    }

    public void testSitraIndividual (List<Object> sitraSourceObjs) {
        try {
            for (Object temp : sitraSourceObjs)
                sitraTransformer.transform(temp);
        } catch (RuleNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void warmUp(ArrayList<SourceElement> jTLSourceObjs, List<Object>
        sitraSourceObjs) {
        for (int i=0; i<WARM_UP_RUNS;i++) {
            startTest(0, jTLSourceObjs, sitraSourceObjs);
            startTest(2, jTLSourceObjs, sitraSourceObjs);
        }
    }
}
```

Listing A.5 Test class to measure the transformation time of SiTra and jTL

A.3 Test Class to transform Pl/1 Source Code Model to Java Source Code

```
public class P2JTransformer {

    private CompilationUnit unit;
    private String filePath = new File("").getAbsolutePath();

    public static void main (String []args) throws IOException {
        P2JTransformer p2jTransformer = new P2JTransformer();
        List<PliStatement> statements = p2jTransformer.generateSourceObjects();
        ClassMethod skeleton = p2jTransformer.generateSkeleton();
        p2jTransformer.startTransformation(statements, skeleton);
    }
```

```
public ClassMethod generateSkeleton() throws IOException {
    unit = vm.newCompilationUnit(filePath + "/PLI2Java/generated");
    unit.setNamespace("generatedJava");
    //Generate Class
    PackageClass cls = unit.newClass("GeneratedJavaClass");
    cls.setAccess(Access.PUBLIC);
    unit.setComment(Comment.D, "Generated by jTL and jenesis4Java");
    //Generate main method
    ClassMethod method = cls.newMethod(vm.newType(Type.VOID), "main");
    method.setAccess(Access.PUBLIC);
    method.isStatic(true);
    method.addParameter(vm.newArray("String", 1), "args");
    return method;
}

public List<PliStatement> generateSourceObjects() {
    // The source code for which the source model is generated in this method
    /*
     * DCL varA DEC FIXED(15,0);
     * DCL varB DEC FIXED(15,0);
     *
     * varA = 540;
     * varB = 0 ;
     *
     * IF varA > 0 THEN DO;
     *      varA = varA / 100 * 90;
     *      IF varA > 100 THEN
     *          varB = varA;
     *      ELSE varB = varA;
     * END;
     */
    List<PliStatement> statements = new ArrayList<PliStatement>();
    ArrayList<PliStatement> ifstmnts1 = new ArrayList<PliStatement>();
    ArrayList<PliStatement> ifstmnts2 = new ArrayList<PliStatement>();

    //Variable declaration:
    //  int varA = null;
    //  int varB = null;
    PliVariableDeclaration varA = new PliVariableDeclaration("varA", PliVarType.
        DECIMAL_FIXED);
    PliVariableDeclaration varB = new PliVariableDeclaration("varB", PliVarType.
        DECIMAL_FIXED);
    statements.add(varA);
    statements.add(varB);


    //Variable initialisations
    //  varA = 540;
    //  varA = 0;
    PliVarRef refA = new PliVarRef(varA);
```

```
        PliVarRef refB = new PliVarRef(varB);
        statements.add(new PliMoveStatement(refA, new PliValue(540)));
        statements.add(new PliMoveStatement(refB, new PliValue(0)));

        // Creating the INNER-IF
        //  if (varA > 100) {
        //      varB = varA;
        //  }
        PliExpression condition2 = new PliBinaryExpression(PliOperator.Greater, refA,
            new PliValue(100));
        PliMoveStatement move2 = new PliMoveStatement(refB, refA);
        ifstmnts2.add(move2);
        PliIfStatement ifStmnt2 = new PliIfStatement(condition2, ifstmnts2,ifstmnts2);
        // Creating the OUTER-IF
        //  if (varA > 0) {
        //      varA = varA / 100 * 90;
        //      <inner-if>
        //  }
        PliExpression condition1 = new PliBinaryExpression(PliOperator.Greater, refA,
            new PliValue(0));
        PliBinaryExpression expr = new PliBinaryExpression(PliOperator.Divide, refA,
                new PliBinaryExpression(PliOperator.Multiply, new PliValue(100), new
                    PliValue(90)));
        PliMoveStatement move1 = new PliMoveStatement(refA, expr);
        ifstmnts1.add(move1);
        ifstmnts1.add(ifStmnt2);
        PliIfStatement ifStmnt1 = new PliIfStatement(condition1, ifstmnts1);
        //Adding the OUTER-IF to the statments --> ready for transformation
        statements.add(ifStmnt1);
        statements.add(move1);
        return statements;
    }

    public void startTransformation(List<PliStatement> statements, Block skeleton)
        throws IOException {
        //Alle Statements werden iterativ transformiert
        for (PliStatement stmnt : statements)
            getContext().getTransformer().transform(stmnt, Optional.of(skeleton));

//Alternative calls (pass a collection) - transform it with ser. or parall. execution
        /*************************************************************************/
        //getContext().getTransformer().transformAll((ArrayList<PliStatement>)
            statements, Optional.of(skeleton));

        //getContext().getTransformer().transformAllParallel((ArrayList<PliStatement>)
            statements, Optional.of(skeleton));
        /*************************************************************************/
        unit.encode();
        String content = new String(Files.readAllBytes(Paths.get(filePath + "/PLI2Java/
            generated/generatedJava/GeneratedJavaClass.java")));
        System.out.println(content);
```

```
    }
}
```

Listing A.6 Test Class as starting point to transform a PI/1 source model to Java source code

A.4 jTL-Rule to transform an if-Statement

```
public class PliIfStatementRule implements Rule<PliIfStatement, Codeable, Block>{
    public Optional<Codeable> convert(PliIfStatement ifStatement, Codeable target,
        Optional<Block> parent) {
            recordMapping(ifStatement,null);
        Expression expression = (Expression) getContext().getTransformer().transform(
            ifStatement.getCondition(), parent).get();
        //IF Generation with transformation of the condition-expression
        parent.orElseThrow(()->new RuleTransformationException(ifStatement,"No parent
            object was defined."));
        If ifStmt = parent.map(p -> p.newIf(expression)).get();
        //IF-Block Transformation
        for (PliStatement stmnt : ifStatement.getIfStatements())
            getContext().getTransformer().transform(stmnt, Optional.of(ifStmt));
        Else elseBlock = ifStmt.getElse();
        //ELSE-Block Transformation
        if (ifStatement.getElseStatements()!=null) {
            for (PliStatement stmnt : ifStatement.getElseStatements())
                getContext().getTransformer().transform(stmnt, Optional.of(elseBlock));
        }
        recordMapping(ifStatement,ifStmt);
        return Optional.of(ifStmt);
    }
}
```

Listing A.7 PliIfStatementRule - Rule to transform an if-statement

A.5 jTL-Rule to transform a binary Expression

```
public class PliBinaryExpressionRule implements Rule<PliBinaryExpression, Codeable,
    Block> {
    @Override
    public Optional<Codeable> convert(PliBinaryExpression source, Codeable target,
        Optional<Block> parent) {
        Expression expr1 = (Expression)getContext().getTransformer().transform(source.
            getLeft(), parent).get();
        Expression expr2 = (Expression)getContext().getTransformer().transform(source.
            getRight(), parent).get();
        return Optional.of(vm.newBinary(ConvertOperatorHelper.getJenesisType(source.
            getOp()), expr1, expr2));
    }
}
```

Listing A.8 PliBinaryExpressionRule - Rule to transform a binary expression

Bibliography

[1] D. H. Akehurst, B. Bordbar, and M. J. Evans. SiTra: Simple Transformations in Java. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems, LNCS*, volume 4199, pages 351–364. Springer, 2006.

[2] H. Balzert. *Java: Objektorientiert Programmieren*. W3L-Verlag, 2014.

[3] P.J. Barendrecht. Modeling transformations using QVT Operational Mappings. Technical report, Eindhoven University of Technology Department of Mechanical Engineering Systems Engineering Group, 2010.

[4] L. Bondé, P. Boulet, and J.-L. Dekeyser. Traceability and interoperability at different levels of abstraction in model transformations. In *Proceedings of the Forum on Specification and Design Languages*. Springer, 2005.

[5] B. Bordbar, G. Howells, M. Evans, and A. Staikopoulos. Model Transformation from OWL-S to BPEL Via SiTra. In *Proceedings of the European Conf. on Model Driven Architecture- Foundations and Applications, LNCS*, volume 4530, pages 43–58. Springer, 2007.

[6] J.S. Cuadrado, J.G. Molina, and M. Menarguez. RubyTL: A Practical, Extensible Transformation Language. In *Proceedings of Model Driven Architecture – Foundations and Applications, LNCS*, volume 4066, pages 158–172. Springer, 2006.

[7] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[8] Edsger W. Dijkstra. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM*, 11(3):147–148, 1968.

[9] L. Doug. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, 2000.

[10] Derek du Preez. Banks will stick with COBOL because Java has performance issues, claims quality guru Bill Curtis. http://www.computerworlduk.com/news/applications/3452537/banks-will-stick-with-cobol-because-java-has-performance-issues-claims-quality-guru-bill-curtis/, June 2013. Accessed: November 2014.

[11] A. Erosa and L.J. Hendren. Taming Control Flow: A Structured Approach to Eliminating Goto Statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994.

[12] E. Gamma, C. Helm, R. Johnson, and J Vlissides. *Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 6. edition, 2011.

[13] R. Gronmo, B. Moller-Pedersen, and G.K. Olsen. Comparison of Three Model Transformation Languages. In *Proceedings of European Conference on Model Driven Architecture (ECMDA-FA)*, volume 5562, pages 2–17. Springer LNCS, 2009.

[14] T. Hoare. Null References: The Billion Dollar Mistake. In *QCon London Software Development Conference*, 2009.

[15] P. Huber. The Model Transformation Language Jungle - An evaluation and Extension of existing Approaches. Master's thesis, University of Technology Vienna, 2008.

[16] M. Inden. *Java 8 Die Neuerungen , Lamdas, Streams, Date and Time API und JavaFX8 im Überblick*. dpunkt.verlag, 1 edition, 2014.

[17] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[18] Alexander Königs and Andy Schürr. Tool integration with triple graph grammars - a survey. *Electron. Notes Theor. Comput. Sci.*, 148(1):113–150, 2006.

[19] G. Krüger and H. Hansen. *Java Programmierung - Das Handbuch zu Java 8: Pragmatic Functional Programming*. O'Reilly Media, Inc., 2014.

[20] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *Software & Systems Modeling*, pages 1–36, 2013.

[21] D. Lea, B. Goetz, P. Sandoz, A. Shipilev, H. Kabutz, and J. Bowbeer. When to use parallel Streams. `Online-Draft:http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html`, Sept 2014.

[22] Sneed H. M., E Wolf, and Heilmann H. *Software-migration in der Praxis Übertragung alter Softwaresysteme in eine moderne Umgebung*. dpunkt.verlag, 1 edition, 2010.

[23] Mauersberger. Medini QVT. `http://projects.ikv.de/qvt`, 04 2012. Accessed: April 2015.

[24] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006.

[25] Naouel Moha, Sagar Sen, Cyril Faucher, and Jean-Marc Barais, Olivier. Evaluation of kermeta for solving graph-based problems. *Int. J. Softw. Tools Technol. Transf.*, 12(3-4):273–285, 2010.

[26] J. I. Moore. Iterating over Collections in Java 8. *Java World*, pages 1–36, Aug. 2014.

[27] Maxim Mossienko. Automated Cobol to Java Recycling. In *Proceedings European Conf. Software Maintenance and Reengineering (CSMR)*, pages 40–50. IEEE, 2003.

[28] M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., 2006.

[29] S. Nolte. *QVT - Operational Mappings*. Xpert.press Springer Verlag, 2010.

[30] Siegfried Nolte. *QVT - Relations Language Modellierung mit der Query Views Transformation*. Springer-Verlag Berlin Heidelberg, 2009.

[31] Oracle. Oracle: Java 1.2 API - Collection. `http://pages.cs.wisc.edu/~hasti/cs368/JavaTutorial/jdk1.2/api/java/util/Collection.html`, 1998. Accessed: August 2014.

[32] Oracle. Oracle: Java 8 - Aggregate Operations - Parallelism. `http://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html`, 2014. Accessed: August 2014.

[33] Oracle. Oracle: Java 8 Tutorial - Lamda Expression. `http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`, 2014. Accessed: August 2014.

[34] Oracle. Oracle: Java API - Package java.util.function. `http://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`, 2014. Accessed: August 2014.

[35] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[36] A.. Rodger. COBOL – continuing to drive value in the 21st Century - The landscape for COBOL asset ownership. `http://www.microfocus.com/000/COBOL_continuing_to_drive_value_in_the_21st_Century_tcm21-23652.pdf`, 2008. Accessed: August 2014.

[37] S. R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill, Inc., 2011.

[38] School of Computer Science, University of Birmingham. SiTra: Official Website. `http://www.cs.bham.ac.uk/~bxb/Sitra/index.html`, 2008. Accessed: August 2014.

[39] M. L. Scott. *Programming Language Pragmatics*. Academic Press, 2006.

[40] Seyyed Shah, Kyriakos Anastasakis, and Behzad Bordbar. Using Traceability for Reverse Instance Transformations with SiTra. In *Proceedings of the Conf. on Design and Architectures for Signal and Image Processing*, 2008.

[41] Harry M. Sneed. Messung und Nachdokumentation eines uralten COBOL-Systems zwecks der Migration zu Java. *Softwaretechnik-Trends*, 29(2):26–28, 2009.

[42] Sourceforge. Jenesis 4 Java Code Generator. `http://jenesis4java.sourceforge.net/`, 2014. Accessed: August 2014.

[43] M. Stephan and Andrew Stevenson. A comparative Look at Model Transformation Languages. Technical report, Software Technology Laboratory at Queens University, 2009.

[44] Triangle Technologies. Relativity – A Tool for the automated Translation of COBOL to Java, 2001.

[45] Marijana Tomic. A possible Approach to Object-oriented Reengineering of Cobol Programs. *SIGSOFT Softw. Eng. Notes*, 19(2):29–34, 1994.

[46] R.G. Urma, M. Fuso, and A. Mycroft. *Java 8 in Action*. Manning, 2014.

[47] D. Wampler. *Functional Programming for Java Developers*. O'Reilly, 2011.

[48] Richard Warburton. *Java 8 Lambdas : Pragmatic Functional Programming*. O'Reilly Media, Inc., 2014.

[49] R. Wilhelm, H. Seidl, and S. Hack. *Compiler Design Syntactic and Semanctic Analysis*. Springer-Verlag, 2013.