# Construct to Reconstruct—Reverse Engineering Java Code with JaMoPP

Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende
*Lehrstuhl Softwaretechnologie*
*Fakultät Informatik*
*Technische Universität Dresden*
*Dresden, Germany*
{*florian.heidenreich,jendrik.johannes,mirko.seifert,c.wende*}*@tu-dresden.de*

***Abstract*—Reconstructing models from software artefacts is of utter importance to migrate existing systems to the age of Model-Driven Software Development. To do so, the software artefacts must be analysed and corresponding models need to be created. Many approaches perform this task in a rather direct way. Behavioural or structural models (e.g., UML state machines or class diagrams) are reconstructed from source code. However, this direct model reconstruction implies some major drawbacks. First, parsers and code analysis are reimplemented for different reconstruction tasks. Second, tools developed for forward engineering can not be reused.**

**To resolve these drawbacks, we propose a multi-step procedure, that analyses the structure of artefacts and transforms them to the technological target spaces. The resulting model of the software artefact contains all information present in the artefact. It is therefore independent of specific reconstruction tasks. Then, the model reconstruction itself is performed. We exemplify this approach with our experiences from reverse engineering Java code with JaMoPP—a tool that creates models from Java source code. As JaMoPP was originally developed for forward engineering, we will also discuss how JaMoPP (and other tools) can automatically contribute to reverse engineering.**

***Keywords*-reverse engineering; modelling;**

## I. Introduction

Reverse engineering software has been quite a topic both for researchers and industry within the last decades. With the advent of Model-Driven Software Development (MDSD) the topic became even more important, since companies want to continue to develop existing applications using the new methods and tools. Thus, they must analyse existing source code and other artefacts to obtain models that represent the existing systems.

While developing the Java Model Parser and Printer (JaMoPP) [2]—a tool dedicated to the model-based representation of Java programs—mainly for forward engineering purposes, we discovered that it is also useful for reverse engineering. Moreover, feedback from the research community indicated, that existing reverse engineering tools for Java often take a different approach. JaMoPP and its ability to create exact and complete models from Java source code allows to perform arbitrary reverse engineering tasks. This is in contrast to tools that directly create target models (e.g., UML [7] class diagrams).

The position we want to state in this paper is twofold. First, we claim that specific generic steps need to be taken to reverse engineer artefacts. We present these steps, explain the order in which they should be taken and try to argue about the benefits of this particular procedure. Second, we show how reverse engineering can benefit from activities that take place during forward engineering as observed in the case of JaMoPP.

## II. General Reverse Engineering Procedure

Software artefacts in the most general sense are sequences of bits and bytes. This is the way they are stored and usually the starting point for a reverse engineering process. To obtain a useful model from the raw representation of artefacts, several steps must be taken. From our point of view, three major steps need to be considered. After explaining each of these steps in detail, we discuss the order in which they must be executed.

### A. Reverse Engineering Steps

*Structure Derivation* Every software artefact represents some kind of structure. In the most basic case, the structure of an artefact can be simply a sequence of information units. More complex artefacts (e.g., XML files) represent a tree structure. Programs are usually represented as graphs. The concrete kind of structure depends on the type of artefact that is subject to reverse engineering. To run analysis on artefacts or to reconstruct valuable information establishing the structural relations is indispensable. Technically the structure is often established using parsers and static semantic analysis.

*Bridging Technological Spaces* Representations of software artefacts can be quite different. Some may be present as XML data, others may be stored in a database. Depending on the goal of the reverse engineering process, the target *technological space* [6] can also be different. If statistical data is required, spreadsheets may be the most appropriate target space. If structural or behavioural models are needed, models complying to some metamodel are needed. To reverse engineer, these different technological spaces need to be bridged. For example, to create a UML model from a Java program, the information that is found in the source code needs to be translated to the modelling space.

*Model Reconstruction* Artefacts are always analysed with a certain goal in mind. For example, one can extract information about the interaction between components of a software system. One may also reconstruct the behaviour within parts of a system. In any case, information that is important with regard to a certain goal is extracted. The information originally present in the artefact(s) is reduced or transformed to obtain the required pieces of data.

## B. Order

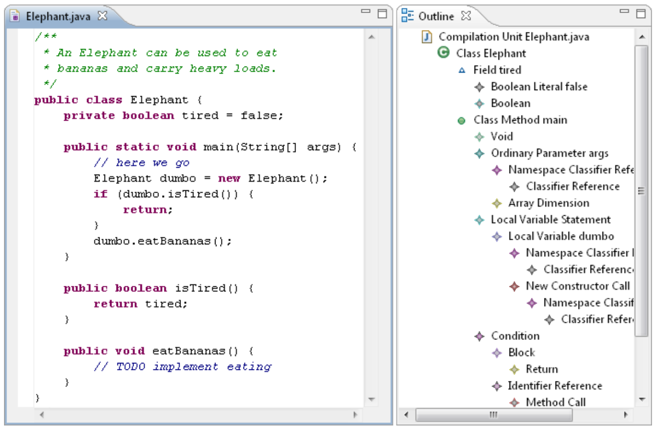
To apply the three steps mentioned in the previous section in a meaningful order, their dependencies need to be analysed. Obviously, the reconstruction of models must be performed after the structural analysis, because it depends on the structure of the artefacts. For example, directly reconstructing the data-flow from a program in its binary representation is simply not feasible. The same applies for the transformation between technological spaces. Most technological spaces imply some kind of structure. XML Schemata enforce a tree structure, while Meta Object Facility (MOF) models can handle graphs. To transform data present in an artefact into one of these technological spaces, the information needs to be converted accordingly. Thus, the structural analysis is most conveniently performed first. This leaves the question whether the conversion between technological spaces should be performed before or after the model reconstruction.

In principle both orders are possible. One can either analyse or abstract from the structured data found in artefacts to obtain a usable model and then convert the data to the technological target space or the other way around. However, one can observe that the second option (i.e., conversion of data to the target space first) is more convenient. This is simply, because engineers are more familiar with the tools of the target space. For example, developers that use XML a lot, are usually familiar with XPath or XSLT to query or transform XML documents. Thus, it is easier for them to perform reconstruction tasks in the XML domain rather than on a concrete syntax tree produced by a parser even though both representations may share the same content.

From our perspective, the most natural order for the three steps is to (1) rebuild the structural information of an artefact, (2) transform the structured data to the technological target space, and (3) abstract or transform to obtain the required information.

## III. Application

JaMoPP covers the first two steps—rebuilding the structural information of an artefact and transforming the structured data to the technological target space—for the Java language, where the modelling space is the target space. Rather than abstracting the information, JaMoPP preserves all information from the Java source code but converts it into a graph structured representation in the modelling space. To



```
/**
 * An Elephant can be used to eat
 * bananas and carry heavy loads.
 */
public class Elephant {
    private boolean tired = false;

    public static void main(String[] args) {
        // here we go
        Elephant dumbo = new Elephant();
        if (dumbo.isTired()) {
            return;
        }
        dumbo.eatBananas();
    }

    public boolean isTired() {
        return tired;
    }

    public void eatBananas() {
        // TODO implement eating
    }
}
```

```
Compilation Unit Elephant.java
  Class Elephant
    Field tired
      Boolean Literal false
      Boolean
    Class Method main
      Void
      Ordinary Parameter args
        Namespace Classifier Refe
          Classifier Reference
        Array Dimension
      Local Variable Statement
        Local Variable dumbo
          Namespace Classifier I
            Classifier Referenc
          New Constructor Call
            Namespace Classif
              Classifier Refer
      Condition
        Block
          Return
        Identifier Reference
          Method Call
```

Figure 1. JaMoPP transparently transforms Java code into a model and back without information loss

demonstrate this, Figure 1 shows a Java class as source code (left) and the corresponding model representation created by JaMoPP (right). We can observe that, for example, all statements and arguments are still present in the model.

Consequently, arbitrary model analysis and transformations (step 3) can be performed without worrying about the first two steps. Figure 2 demonstrates this showing different possible applications in reverse engineering.

Any legacy Java code can be transferred into a complete model of the Java part of an application with JaMoPP out of the box (upper left of Fig. 2). In the modelling space, the model of the Java code can be treated similarly to other models that are extracted from other technology spaces (e.g., an XML model produced by an XML-to-Ecore binding[1]) or that already existed in the modelling space (e.g., class diagrams of the original system design). This could be a scenario of reverse engineering an Enterprise Java Beans (EJB) application that consists of Java code and XML configuration files.

Thanks to JaMoPP, all information contained in the Java code of the application is now available in the modelling space. Thus, all kinds of reverse engineering operations can be performed in the modelling space only using model analysis and transformation approaches. This allows for example to extract data or data flow models from the models of the applications using model transformations that work on all information—i.e., all models—needed (middle of Fig. 2), including all information contained in the Java code.

Ultimately, the extracted information would be used to create a modernised system. At this stage, we can generate the system from the models we created before (bottom of Fig. 2). Since the original Java model is still available, we can also incorporate unchanged parts of the system. For example, if the scenario is to migrate an EJB2 to an EJB3

[1]http://www.eclipse.org/modeling/emf/docs/2.x/tutorials/xlibmod/xlibmod_emf2.0.html

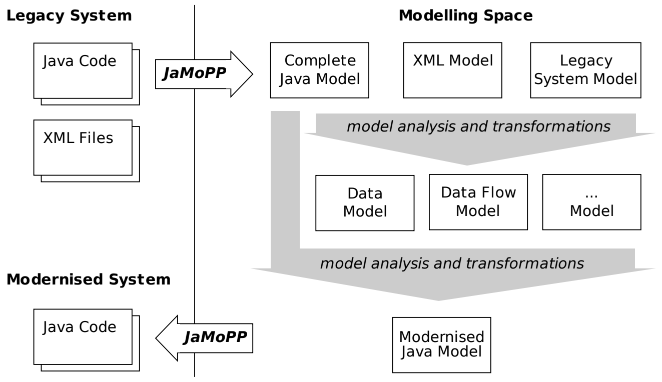


Figure 2. JaMoPP applied in reverse engineering

application, we might do modifications to the interfaces of bean classes (e.g., add new annotations, split or merge interfaces) but leave the business logic untouched. The business logic (i.e., lists of Java statements) can be extracted directly from the original Java model. Small changes (e.g., renaming interfaces) that concerns these statements could still be reflected by the transformation by taking other models into account.

The fact that JaMoPP bidirectionally maps the complete information present in the source code to a model allows us to use it both for forward and reverse engineering. We believe that this approach in beneficial for any kind of artefact, because once the specification of the mapping is available, many software engineering tasks can instantly take advantage of it.

## IV. Related Work

Most modern IDEs support reverse engineering of Java Source Code in one form or another. Especially creating UML diagrams from existing Java applications is quite common and covered by many tools. However, reverse engineering activities other than UML model reconstruction are supported less frequently. For example, obtaining models in other modelling languages (e.g., Domain-Specific Languages (DSLs)) is not possible. In addition, existing tools and IDEs often make implicit assumptions about the way the code was derived from models. If developers need to make different assumptions (e.g., because they have used different code templates) tools often fail.

Because of the huge amount of tools and approaches that exist to reverse engineer Java source code, we will only name the ones that are most relevant here. Namely the various approaches that describe how to visualise and analyse software are not included here, because JaMoPP was built for a different purpose. We consider analysis, transformation and visualisation as tasks that can be performed upon the Java models created by JaMoPP, which relieves developers from building parsers or static semantic analysers over and over again. Merely, the idea is to wrap all information that is present in Java source code in a model and let developers decide what to do with this data. Thus, we will compare JaMoPP mainly to tools that follow this line by providing models for source code possibly independent of concrete reverse engineering scenarios.

Netbeans[2] can read existing Java source code and create a model for each Java class. This model is independent from the specific reverse engineering task, similar to the models created by JaMoPP. However, the granularity of the models captures only information that is relevant for creating UML models later on. Statements or other elements below the level of types and members are not captured by Netbeans' intermediate models. This is fine for the purpose of generating UML models, but restricts Netbeans ability to mine other kinds of models from Java applications.

The Fujaba[3] tool suite also provides functionality to read and transform Java source code to construct models. Fujaba creates a full-fledged Java AST from the code which can be analysed and transformed using declarative patterns called Story Diagrams. This approach is quite powerful and has been applied to solve problems such as design pattern recovery [4]. Especially the declarative and graphical specification of the patterns to search for contributes to Fujaba's popularity. The main drawback of Fujaba is the fact that it does not have an explicit metamodel for the Java AST. Implemented using a set of plain Java classes instead of using a standardised metamodelling language, the application of other transformation or analysis tools is hard. There has been work carried out to solve this issue [5], but this is only part of a solution.

MoDisco[4] is an Eclipse[5] Plug-in that aims at discovering models from legacy applications. It does handle Java source code as well as other artefacts (e.g., configuration files). Based on the Eclipse Java Development Tools (JDT) parser and the resulting AST, MoDisco creates Java models from source code files. The metamodel for this models is defined in Ecore—an implementation of the OMG standard Essential MOF (EMOF). Thus, all tools that are capable of handling Ecore models can be used to explore the models created by MoDisco. Transformations can be easily specified using existing languages (e.g., Atlas Transformation Language (ATL)). Visualisations can be obtained with frameworks such as the Graphical Modeling Framework (GMF). MoDisco therefore enables arbitrary reverse engineering tasks. It shares the idea of a designated metamodel for Java and the grounding on a standardised meta language.

Similar to MoDisco, Spoon [9] is based on the Eclipse JDT. It does also provide an Ecore metamodel for Java, which is more detailed than the one from MoDisco. The metamodel covers all elements of Java down to the statement

[2]http://www.netbeans.org
[3]http://www.fujaba.de
[4]http://www.eclipse.org/gmt/modisco/
[5]http://www.eclipse.org

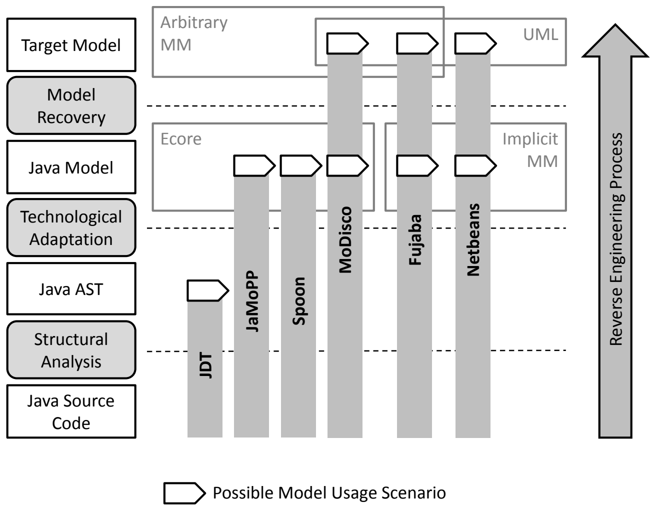


Figure 3. Tool Classification for Reverse Engineering Java Source Code

and expression level. Spoon also performs static semantics analysis using the JDT, which leads to a close coupling with the latter.

Figure 3 summarises the features of the tools mentioned above. To obtain models that are based on a standardised metamodel MoDisco, Spoon and JaMoPP are feasible. MoDisco does already provide some transformations to translate Java models (e.g., to obtain instances of the Knowledge Discovery Metamodel (KDM) [8]). Fujaba and Netbeans are different because they use an internal metamodel.

From a reverse engineering point of view, MoDisco and Spoon are as generic and powerful as JaMoPP. But, being based on the concrete syntax mapper EMFText [1], JaMoPP can be also used for forward engineering. It can easily be extended to realise new language features for Java and was successfully applied to create a safe template language for Java [3].

## V. Conclusion

In this paper we presented our view on the generic steps thats need to be taken to reverse engineer artefacts (derive structure, bridge technological spaces, reconstruct) and outlined the possibilities in which JaMoPP can be used in this context. We emphasised on the idea of transforming artefacts to a technological target space while preserving all information available in the original artefacts. Representing Java code as a model allows for using arbitrary model transformation and analysis tools available in this technological space by decoupling the steps of bridging technological spaces and reconstructing the actual artefacts. Furthermore, we discussed how a tool that has been originally designed for forward engineering can contribute to the task of reverse engineering.

To this end, we believe that tools which serve both forward and reverse engineering are superior, because every artefact that is generated by forward engineering today is most probably a candidate for reverse engineering tomorrow.

## Acknowledgment

This research has been co-funded by the European Commission within the FP6 project MODELPLEX #034081, the FP7 project MOST #216691 and by the German Ministry of Education and Research within the projects feasiPLe and SuReal.

## References

[1] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, "Derivation and Refinement of Textual Syntax for Models," in *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)*, ser. LNCS, vol. 5562. Springer, 2009, pp. 114–129.

[2] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the Gap between Modelling and Java," 2009, accepted for tool demonstration at SLE'09.

[3] F. Heidenreich, J. Johannes, M. Seifert, C. Wende, and M. Böhme, "Generating Safe Template Languages," 2009, accepted for publication at GPCE'09.

[4] J. H. Jahnke and A. Zündorf, "Rewriting poor design patterns by good design patterns," in *Proc. of the ESEC/FSE Workshop on Object-Oriented Re-engineering*, S. Demeyer and H. Gall, Eds. Technical Report TUV-1841-97-10, Technical University of Vienna, Information Systems Institute, Distributed Systems Group, September 1997.

[5] J. Johannes, "Letting EMF Tools Talk to Fujaba through Adapters," in *Fujaba Days 2008*, U. Aßmann, J. Johannes, and A. Zündorf, Eds., 2008.

[6] I. Kurtev, J. Bézivin, and M. Akşit, "Technological spaces: An initial appraisal," in *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.

[7] OMG, "Unified Modeling Language: Superstructure Version 2.2," Object Management Group (OMG), Tech. Rep., 2007. [Online]. Available: http://www.omg.org/docs/formal/09-02-02.pdf

[8] ——, "Knowledge Discovery Metamodel (KDM) Version 1.1," Object Management Group (OMG), Tech. Rep., 2009. [Online]. Available: http://www.omg.org/spec/KDM/1.1/PDF/

[9] R. Pawlak, "Spoon: Compile-time Annotation Processing for Middleware," *IEEE Distributed Systems Online*, vol. 7, no. 11, 2006.