# An Exchange Model for Reengineering Tools

Sander Tichelaar and Serge Demeyer,
Software Composition Group, University of Berne, Switzerland,
{demeyer,tichel}@iam.unibe.ch

## Abstract

Tools support is recognised as a key issue in the reengineering of large scale object-oriented systems. However, due to the heterogeneity in today's object-oriented programming languages, it is hard to reuse reengineering tools across legacy systems. This paper proposes a language independent exchange model, so that tools may perform their tasks independent of the underlying programming language. Beside supporting reusability between tools, we expect that this exchange model will enhance the interoperability between tools for metrics, visualization, reorganisation and other reengineering activities.

The complete model is available at: http://www.iam.unibe.ch/~famoos/InfoExchFormat/
All comments are welcome: famoos@iam.unibe.ch.

## 1) Introduction

Reengineering object-oriented systems is a complex task. Therefore, many tools have been and are being developed to support developers in doing things such as visualization, metrics & heuristics and system reorganisation. However, legacy systems are written in different implementation languages (C++, Ada, Smalltalk and even Java). To avoid equipping all of the tools with parsing technology for all of the implementation languages, we have developed a model for information exchange between reengineering tools. The model is a language independent representation of object-oriented sources and should provide enough information for the reengineering tasks of the tools (see Figure 1).

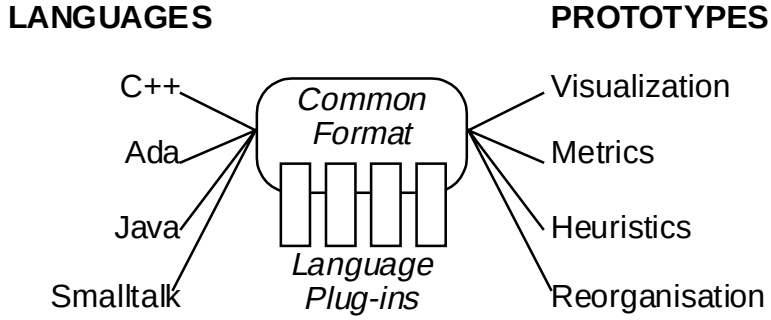


**Figure 1: Conception of the Exchange Format**

However, we cannot know in advance all information that is needed. Therefore, the model is extensible in a couple of ways. First, tools may need to work on the language specific issues of a system. Therefore, we allow *language plug-ins* that extend the model with language-specific items, but don't break the language independent tools. Second, we allow tools

themselves to extend the model, for instance, to store analysis results or layout information for graph tools, again without breaking other tools.

To exchange actual information, i.e. source code expressed in our model, we use the CDIF standard for information exchange[CDIF94a]. CDIF provides us with a standard way of exchanging models and model instances in ASCII representation.

Within the FAMOOS Esprit project we are currently using the exchange model together with its representation in CDIF in our reengineering tool prototypes. Prototypes written in different languages work on sources written in different languages. The first results look promising, but we need to do more experiments to validate the model and the gains it should provide in reusability and interoperability of our tools

In this document we discuss the requirements for the exchange model (section 2). Next, we discuss the core of the model, which contains of the most important information for our reengineering tool prototypes (section 3). After that, we discuss the extensibility aspect of the model and shortly discuss the CDIF standard and why we didn't choose UML[Booc96a] as our exchange model.

# 2) Requirements Specification

Based on our experiences with the tool prototypes built so far, plus given a survey of the literature on reengineering repositories and code base management systems we specified the following requirements list. The list is split up in two, one part defining requirements concerning the data model, the other part specifying issues concerning the representation.

*Data Model*

1. *Extensible.*
   To handle the definition of language plug-ins, the data model must allow extensions with language specific entities and properties. Some tool prototypes may also need to define tool specific properties.
2. *Sufficient basis for metrics, heuristics, grouping and re-engineering operations.*
   To avoid a common denominator that would ineffective for our goals, we set the lower limit for the model to everything that is required to experiment with the tool prototypes.
3. *Readily distillable from source code.*
   Since it is not our aim to define a model that covers all aspects of all languages, the upper limit to the information the model will contain, is what can be generated by basic code parsing (i.e. parsing without any interpretation of the obtained information, for instance, determining if a relation is an aggregation or a composition). The generated information should be usable by any tool , thus also by language independent tools.

*Representation*

1. *Easy to generate by available parsing technology.*
   Since we cannot wait for future developments, we must use parsers available today keeping an eye on short-term evolution. Within the FAMOOS project, parsing technology comes mainly from the FAST library part of the Audit platform. However, there are a number of other viable alternatives: like the SNiFF+ symbol table which is accessible via an API; like Ada compilers which

provide standard API's for accessing internal data structures; like the tables generated by Audit which can be transformed in what is needed; like the Java inspection facilities part of Java.lang.reflect or even the Java byte code itself; like Smalltalk inspection facilities and parsers that are part of every Smalltalk implementation.

2. *Simple to process.*
   As the exchange format will be fed into a wide variety of tool prototypes, the format itself should be quite easy to convert into the internal data structures of those prototypes. On top of that, processing by "standard" file utilities (i.e., grep, sed) and scripting languages (i.e., perl, python) must be easy since they may be necessary to cope with format mismatches.
3. *Convenient for querying.*
   A large portion of re-engineering is devoted to the search for information. The representation should be chosen so that it may easily be transformed into an input-stream for querying tools (i.e., spreadsheets and databases).
4. *Human readable.*
   The exchange format will be employed by (buggy) prototypes. To ease debugging, the format itself should be readable by humans. Especially, references between entities should be by name rather than by identifiers bearing no semantics.
5. *Allows combination with information from other sources.*
   Although most of the data model will be extracted from source code, we expect that other origins can provide input as well. Especially CASE tools with design diagrams (e.g., TDE or Rational/Rose) are likely candidates. Thus, the representation should allow merging information from other origins. Note that — just like with the "human readable" requirement— this implies that references between entities should be by name rather than by identifiers bearing no semantics.
6. *Supports industry standards.*
   Since the tool prototypes must be utilised within an industry context, they must integrate with whatever tools already in use. Ad hoc exchange formats (even when they can be translated with scripts) hinder such integration, and --when available-- the representation should favour an industry standard.

# 3) The Data Model

## 3.1. The Core Model

The core model specifies the entities and relations that can and should be extracted immediately from source code (see Figure 2). The core model consists of the main OO entities, namely `Class`, `Method`, `Attribute` and `InheritanceDefinition`. For reengineering we need the other two, the associations `Invocation` and `Access`. An `Invocation` represents the definition of a `Method` calling another `Method` and an `Access` represents a `Method` accessing an `Attribute`[1]. These abstractions are needed for reengineering tasks such as dependency analysis, metrics computation and reengineering operation. Typical questions we

[1] Actually, the complete model is more general: an Invocation is about behavioural entities (such as methods and functions) calling other behavioural entities and an Access is about a behavioural entity accessing a structural entity (such as attributes and global variables).

need answers for are: “are entities strongly coupled?”, “which methods are never invoked?”, “I change this method. Where do I need to change the invocations on this method?”.

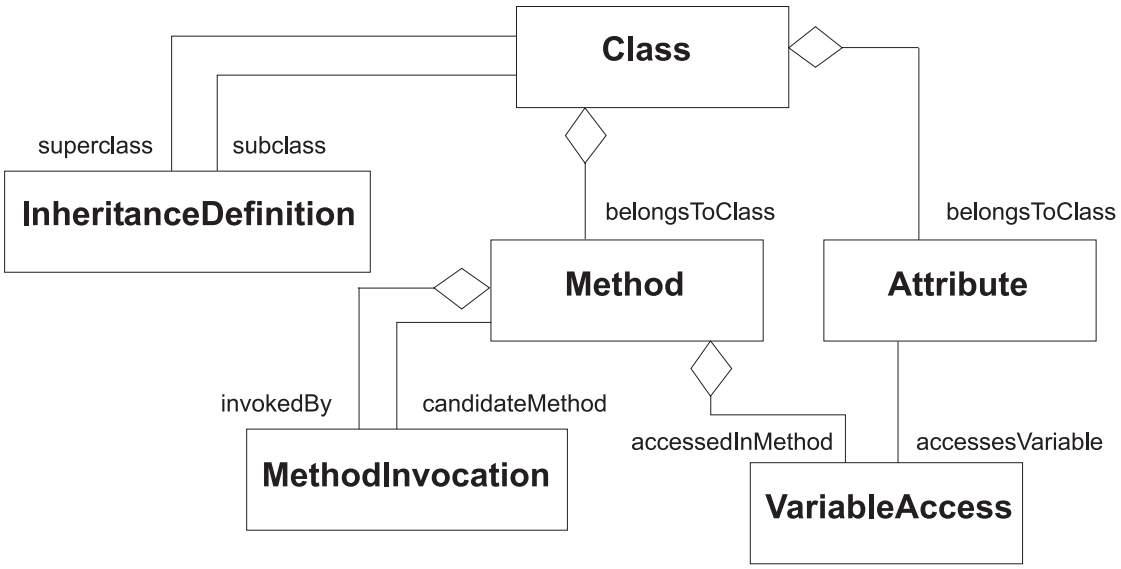


**Figure 2: The Core Model**

## 3.2. The complete model

The structure of the complete model is shown in Figure 3. `Object`, `Property`, `Entity` and `Association` are made available to handle the extensibility requirement (see "Requirements Specification" - p.2). For specifying language plug-ins, it is allowed to define language specific classes and to add language specific attributes to existing `Objects`. Tool prototypes are more restricted in extending the model: they can define tool specific `Properties` for and can add attributes to existing `Objects`. They are, however, not allowed to extend the repertoire of entities and associations.

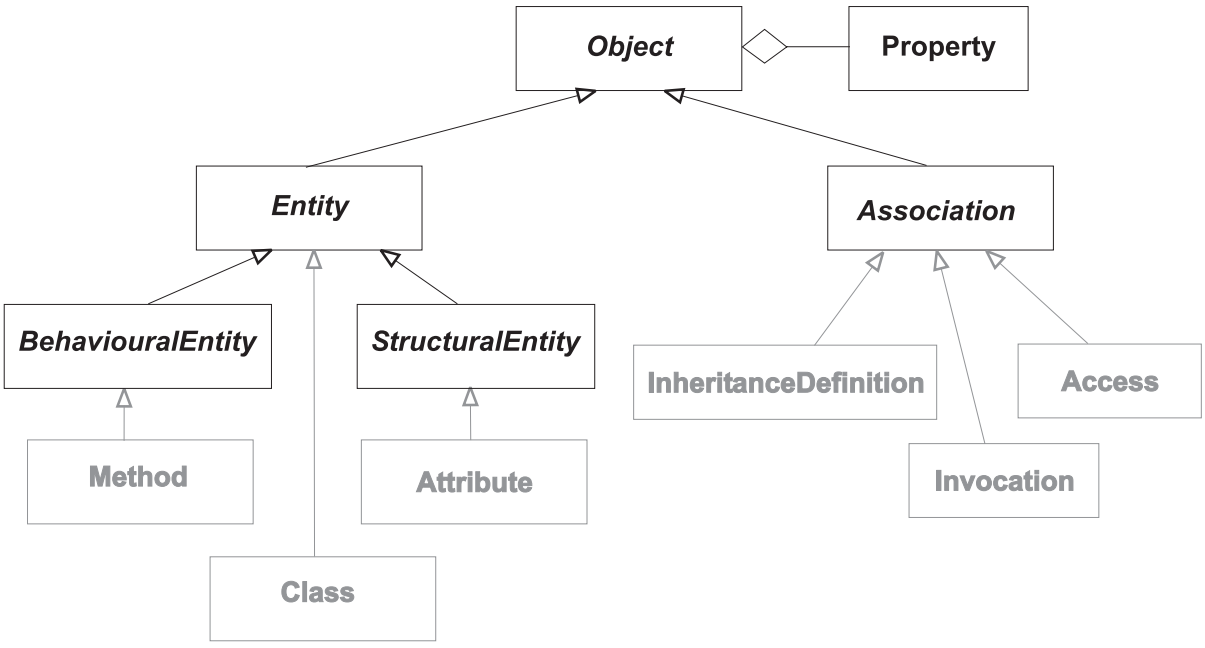


**Figure 3: Basic structure of the complete model**

# 4) CDIF Transfer Format

We have adopted CDIF[CDIF94a] as the basis for the information exchange of information in the FAMOOS exchange model. CDIF is an industrial standard for transferring models created with different tools. The main reasons for adopting CDIF are, that firstly it is an industry standard, and secondly it has a standard plain text encoding which tackles the requirements of convenient querying and human readability. Next to that the CDIF framework supports the extensibility we need to define our model and language plug-ins.

# 5) Why not UML?

The Unified Modelling Language (UML) [Booc96a] is rapidly becoming the standard modelling language for object-oriented software, even in industry. So, UML is a viable candidate for serving as the data model behind our exchange format. Nevertheless, UML is geared towards an analysis / design language and there exists no accurate and straightforward mapping from source-code to UML. For instance, inheritance like applied in an implementation does not necessarily correspond to generalisation like specified in UML (e.g., in an implementation a Rectangle might be a subclass of Square while a correct generalisation is the other way around). Likewise, attribute definitions do not always correspond with aggregation (e.g., is a Rectangle an aggregation of two instances of Point or is it an aggregation of four integers). Thus choosing UML would violate the requirement that the data model should be readily distillable from source code (see p.2) and that's the first motivation to rule out UML.

Moreover, extracting an accurate UML model from source code is considered quite important for model capture. We will definitely investigate that topic in further depth, and we do not want to hamper such investigations by choosing a straightforward but inaccurate mapping. That is the second motivation to rule out UML.

Finallly, UML does not include internal dependencies such as invocations and accesses. Those dependencies are necessary for problem detection and reorganisation operations. Thus, choosing UML would violate the requirement of being a sufficient basis for re-engineering operations (see p.2).

However, we relied heavily on UML in the terminology and naming conventions applied in our model to become independent of the implementation language. For example, we talk about attributes instead of members (C++) or instance variables (Smalltalk) and we talk about classes instead of types (Ada).


### Acknowledgements

This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT IV programme Project no. 21975 (FAMOOS, see http://www.iam.unibe.ch/~famoos/).


### References


[Booc96a] Booch, G., Jacobson, I. and Rumbaugh, J, The Unified Modelling Language for Object-Oriented Development. See http://www.rational.com/.

[CDIF94a] "CDIF Framework for Modeling and Extensibility", Electronic Industries Assocaiation, EIA/IS-107, January 1994, online available at http://www.cdif.org/.