Definition of a Common Exchange Model

Version 1.0 -- Last Modified: Tuesday, March 31, 1998
Available on the WWW at: http://www.iam.unibe.ch/~famoos/InfoExchFormat/

Abstract

This document defines the exchange model for usage by tool prototypes within the FAMOOS re-engineering project. The model is based upon the CDIF standard so that it can be transferred via flat ASCII streams.

All comments are welcome: famoos@iam.unibe.ch.

1) Introduction

The FAMOOS project (http://www.iam.unibe.ch/~famoos/) aims to develop a re-engineering method for transforming object-oriented legacy code into frameworks. The re-engineering method itself is defined around a life cycle model (see also Figure 1).

1. Model Capture: documenting and understanding the software system
2. Problem Detection: identifying flexibility and quality problems
3. Problem Resolution: selecting new software architectures to correct the problems
4. Reorganisation: transforming the existing software architecture for a new release
5. Change Propagation: ensuring that all client systems benefit from the new release

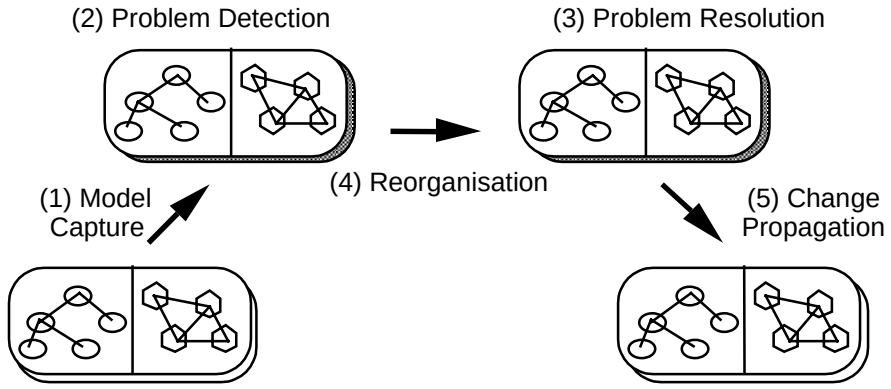

Figure 1: FAMOOS re-engineering life cycle

To realise that life cycle, three research areas –which are likely to furnish solutions– have been selected for further investigation

Metrics & Heuristics [DETECTM]
: Applied in phase (2) to identify problems and phase (3) to measure improvement.

Grouping [DOCUM]
: Applied in phase (1) to form software modules and phase (3) to form target architectures.

Reorganisation Operations [REORGOP]

Applied in phase (4) to perform the actual program transformations and phase (5) to adapt the target software context.

Currently, the FAMOOS partners are building a number of tool prototypes for conducting various experiments within those three research areas. However, the source code available for case studies is written in different implementation languages (C++, Ada and to a lesser extent Java and Smalltalk). To avoid equipping all the tool prototypes with parsing technology for all of the implementation languages, it is necessary to agree on a common information exchange format with language specific extensions (see Figure 2). This document is a specification for such a format.

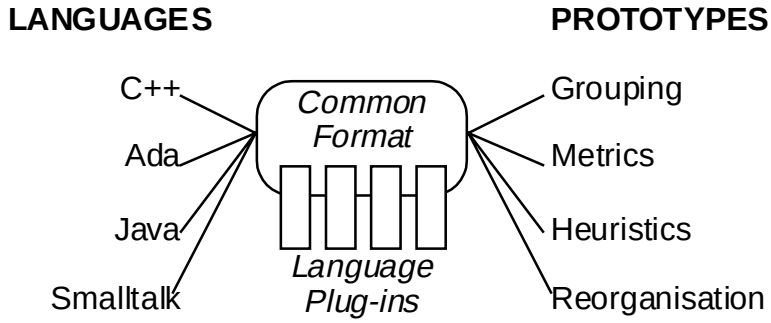

Figure 2: Conception of the Common Exchange Format

2) Requirements Specification

Based on our experiences with the tool prototypes built so far, plus given a survey of the literature on re-engineering repositories and code base management systems we specified the following requirement 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, we set the upper limit for the model as something that can be generated by ordinary code parsers.

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.
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) CDIF Transfer Format

We have chosen CDIF as the basis for the FAMOOS exchange model [EVALCDIF]. CDIF is an industrial standard for transferring models created with different tools. More information concerning the CDIF standard can be found at http://www.cdif.org/. Among others, the CDIF standard provides an encoding that allows the transfer models via flat ASCII streams.

4) The Data Model

4.1. The Core Model

The core model specifies the entities and relations that can and should be extracted immediately from source code. We explicitly opted for longer names here ("`ClassDefinition`" instead of "`Class`", "`MethodDefinition`" instead of "`Method`") because other sources (CASE tools, architecture extractors, etc.) may also provide a "`Method`" and a "`Class`", which is not necessarily the same concept as a method or a class extracted from source code. The long naming scheme should allow us to deal with other information sources.

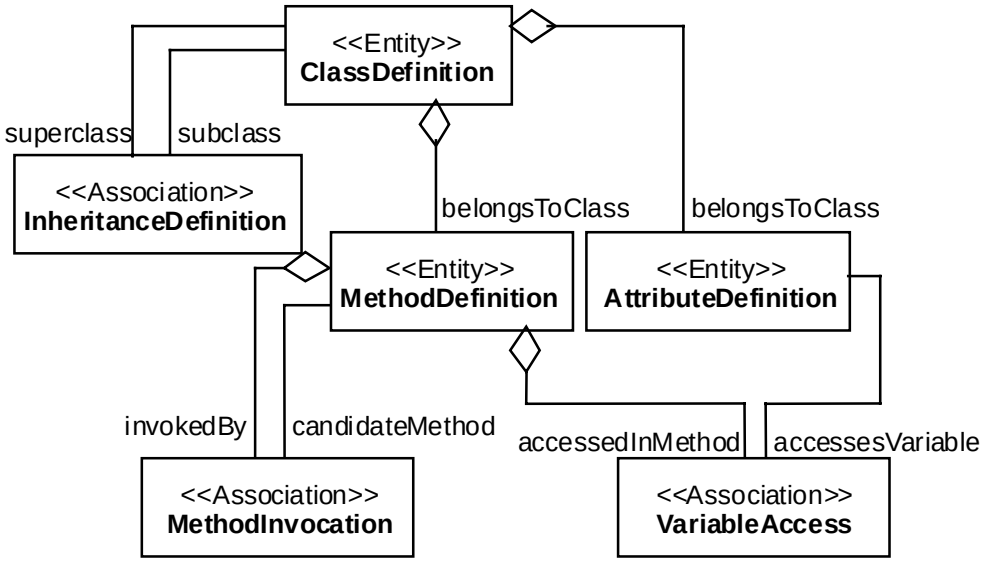

Figure 3: The Core Model

4.2. Level of Reification

The core model contains entities that not all parsers may provide (for instance, the `MethodInvocation` and `VariableAccess` associations are not available from the SNiFF-API). Also, some tools do not always need all of this information (i.e. many metrics can already be gathered from `ClassDefinition`, `MethodDefinition` alone). To allow "incomplete" models, we introduced the *level of reification.*

Basically, the level of reification is an integer, telling how much of the core model is actually available. In principle, the bigger the number, the more information is available. The following table gives an overview of the levels of reification.

Level 1	`ClassDefinition, InheritanceDefinition, MethodDefinition`. Level 1 is the minimum model that parsers should be able to provide and corresponds with what is usually understood as the interface of a class.
Level 2	Level 1 + `AttributeDefinition`
Level 3	Level 2 + `VariableAccess`
Level 4	Level 3 + `MethodInvocation`. The `candidateMethod` role for `MethodInvocation` is empty; i.e. it is not necessary to compute the actual class that defines the invoked method.
Level 5	Level 4 + the `candidateMethod` role. This implies that parsers should provide some details about the types of the arguments that are passed when invoking the method.
Level 6	Level 5 + all information concerning arguments, local variables, global variables, implicit variables.

Table 1: Levels of Reification

4.3. Entity / Association (the Meta-Meta Model)

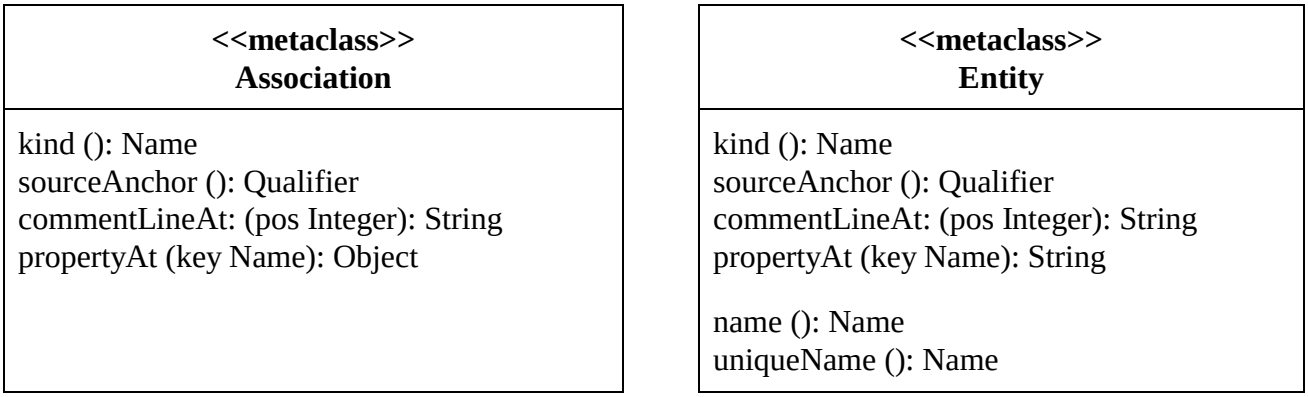

Figure 4: The Meta-Meta Model (Entity, Association)

Entity and Association are explicit parts of the meta-meta model made available to handle the extensibility requirement (see "**2) Requirements** Specification" - p.2). For specifying language plug-ins, it is allowed to define language specific entities and associations plus it is allowed to add language specific properties to existing entities or relations. Tool prototypes are more restricted in extensions to the model: they can define tool specific properties for existing entities or associations, but can not extend the repertoire of entities and associations themselves.

Users who do not need to extend the model may safely assume that the meta-classes `Association` and `Entity` behave like an abstract superclass for the classes in the core model. That is, if a class in the core model is said to be an `Association` or an `Entity` (via a stereotype on the class), it will inherit the protocol from those classes.

- `kind: Name; mandatory`
  Is a name that identifies the class (type) of an object.

- `sourceAnchor: Qualifier; optional, default = 'null'`
  Is a qualifier that identifies the location in the source where the information is extracted. The exact format of the qualifier is dependent on the source of the information. Usually, it will be an anchor in a source file, in which case the following format should be used
  `(file <filespec> start <start_index> stop <stop_index>).`
  Where `<filespec>` is a string holding the name of the source-file in an operating system dependent format (preferably a filename relative to some project directory). Where `<start_index>` and `<stop_index>` are indices starting at 1 and holding the beginning / ending character position in the source file.
  Extra position indices may be added to handle anchors in files that have been edited after parsing. For instance, the line & column of the character (`startline`, `startcol`, `stopline`, `stopcol`). Or the negative offset counting from the end of the file instead of from the beginning (`negstart`, `negstop`).

- `commentLines: 0..N String`
  Entities and associations may own a number of comment lines, where tools can store textual information about the object. One particular comment line is accessed via its position.

- `properties: 0..N Object`
  Entities and associations may own a number of properties where extensions of the core model may be stored. One particular property is accessed via its name.

To enable a global referencing scheme based on names, the key classes in the model should respect the minimal interface of `Entity`.

- `name: String; mandatory`
  Is a string that provides some human readable reference to an entity.

- `uniqueName: String; mandatory`
  Is a string that is computed based on the name of the entity. Each class must define its specific formula. The uniqueName serves as an external reference to that entity and must be unique for all entities in the model.

4.4. Basic Data Types

Besides the usual primitive data types (String, Integer, Boolean…) there are a number of extra data types that are considered "basic". Note that all data types have the possibility of representing a null-value.

Name
name: String
isNull (): Boolean notNull (): Boolean

Qualifier
qualifier: String
isNull (): Boolean notNull (): Boolean

Index
index: Integer
isNull (): Boolean notNull (): Boolean

Figure 5: Basic Data Types (Name, Qualifier and Index)

- `Name` vs. `Qualifier`
  A `Name` is a string that bears semantics inside the model, while a `Qualifier` is a string that gets its semantics from outside the model. A `String` does not bear any semantics. For instance, a `uniqueName` may be used to refer to another object, hence bears semantics inside the model. However, a `sourceAnchor` will store some information that must be interpreted by applications outside the model, hence is a qualifier. Finally, a comment line is a string, since it does not bear any semantics understandable by a computer.

- `Index`
  An Index represents a position in some sequence. Indices always have a base of 1, leaving 0 as the null value.

4.5. Core Entity: ClassDefinition

<<Entity>> **ClassDefinition**
isAbstract (): BooleanOrNull scopeQualifier (): Qualifier

Figure 6: ClassDefinition

A `ClassDefinition` represents the definition of a class in source code. What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Entity, it has the following attributes:

- `isAbstract: Boolean; optional, default = 'null'`
  Is a predicate telling whether the class is declared abstract. Abstract classes are important in OO modelling, but how they are recognised in source code is a language dependent issue.

- `scopeQualifier: Qualifier; optional, default = 'null'`
  Is a string with a language dependent interpretation, that defines the scope of a class. A null scopeQualifier is allowed, it means that the class has global scope. The scopeQualifier concatenated with the name of the class must provide a unique name for that class within the model.

- formula for `uniqueName`

```
if isNull (scopeQualifier(class)) then
  uniqueName (class) = name (class)
else
  uniqueName (class) = scopeQualifier (class)
    + "::" + name (class)
```

4.6. Core Entity: MethodDefinition

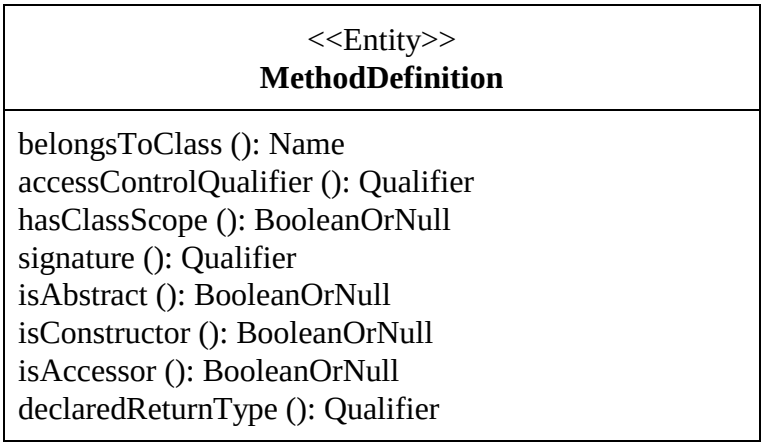

Figure 7: MethodDefinition

A `MethodDefinition` represents the definition in source code of an aspect of the behaviour of a class. What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Entity, it has the following attributes:

- `belongsToClass: Name; mandatory`
  Is a name referring to the class owning the method. It uses the uniqueName of the class as a reference.

- `accessControlQualifier: Qualifier; optional, default = 'null'`
  Is a string with a language dependent interpretation, that defines who is allowed to invoke it (for instance, 'public', 'private'…).

- `hasClassScope: Boolean; optional, default = 'false'`
  Is a predicate telling whether the method has class scope (i.e., invoked on the class) or instance scope (i.e., invoked on an instance of that class).

- `signature (): Qualifier; mandatory`
  Is a string that allows to uniquely distinguish a method within a class. This is necessary because there exist OO languages (i.e., C++, Java) that allow to overload methods, so that the same method name may be associated with different parameter lists, each with its own method body. Also, sometimes it is possible to define the same method name once on the instance level and once on the class level.
  The way the signature string is composed is language dependent, but it should at least include the name of the method.

- `isAbstract: Boolean; optional, default = 'null'`
  Is a predicate telling whether the method is declared abstract. Abstract methods are important in OO modelling, but how they are recognised in source code is a language dependent issue.

- `isConstructor: Boolean; optional, default = 'null'`
  Is a predicate telling whether the method is a constructor. A constructor is a method that creates an (initialised) instance of the class it is defined on. Thus a method that creates an instance of another class is not considered a constructor. How constructor methods are recognised in source code is a language dependent issue.

- `isAccessor: Boolean; optional, default = 'null'`
  Is a predicate telling whether the method is an accessor. There are two kinds of accessors, a reader accessor and a writer accessor. A reader-accessor is a method with one receiver parameter, returning the value of an attribute of the class the method is defined on. A writer method is a method with one receiver parameter and one value parameter, storing the value inside the attribute of a class. How accessor methods are recognised in source code is a language dependent issue.

- `declaredReturnType: Qualifier; optional, default = 'null'`
  Is a qualifier that via interpretation outside the model refers to the class of the returned object.
  Note that we need a language dependent interpretation to link a typename to a class name, because in most OO languages, types are not always equivalent to a class. How the declaredReturnType may be recognised in source code and how the return type matches to a class are language dependent issue

- formula for `uniqueName`

```
uniqueName (method) = belongsToClass (method) +
  "::" + signature (method)
```

4.7. Core Entity: AttributeDefinition

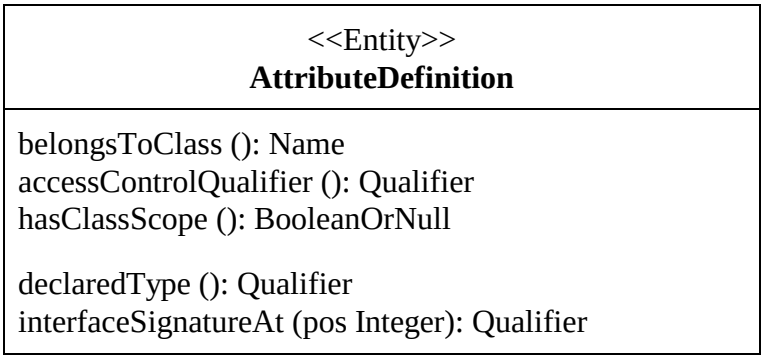

Figure 8: AttributeDefinition

An `AttributeDefinition` represents the definition in source code of an aspect of the state of a class. What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Entity, it has the following attributes:

- `belongsToClass: Name; mandatory`
  Is a name referring to the class owning the attribute. It uses the uniqueName of the class as a reference.

- `accessControlQualifier: Qualifier; optional, default = 'null'`
  Is a string with a language dependent interpretation, that defines who is allowed to access it (for instance, 'public', 'private'…).

- `hasClassScope: Boolean; optional, default = 'false'`
  Is a predicate telling whether the attribute has class scope (i.e., shared memory location for all instances of the class) or instance scope (i.e., separate memory location for each instance of the class).

- `declaredType: Qualifier; optional, default = 'null'`
  See definition of VariableDefinition (see p.13).

- `interfaceSignatures: 0 .. N Qualifier`
  See definition of VariableDefinition

- formula for `uniqueName`

```
uniqueName (attribute) = belongsToClass (attribute) +
  "." + name (attribute)
```

4.8. Core Association: InheritanceDefinition

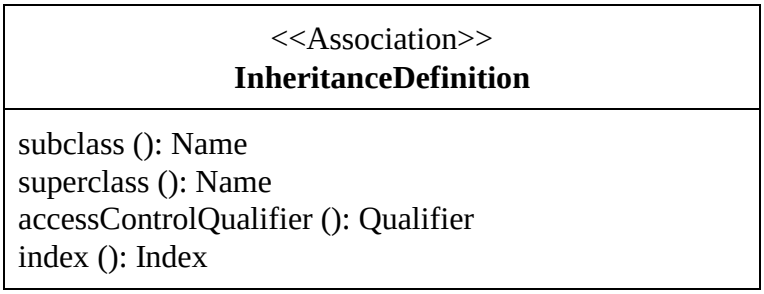

Figure 9: InheritanceDefinition

An `InheritanceDefinition` represents the definition in source code of an inheritance association between two classes. One class then plays the role of the superclass, the other plays the role of the subclass. What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Association, it has the following attributes:

- `subclass: Name; mandatory`
  Is a name referring to the class that inherits. It uses the uniqueName of the class as a reference.

- `superclass: Name; mandatory`
  Is a name referring to the class that is inherited from. It uses the uniqueName of the class as a reference.

- `accessControlQualifier: Qualifier; optional, default = 'null'`
  Is a string with a language dependent interpretation, that defines how subclasses access their superclasses (for instance, 'public', 'private'…).

- `index: Index; optional, default = 'null'`
  In languages with multiple inheritance, this is the position of the superclass in the list of superclasses of one subclass. Usually this will have a null value, but it may be necessary for OO languages with multiple inheritance that resolve name collisions via the order of the superclasses (i.e., CLOS).

4.9. Core Association: VariableAccess

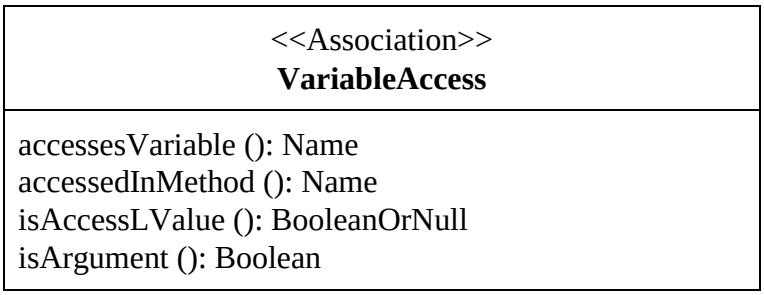

Figure 10: VariableAccess

A `VariableAccess` represents the definition in source code of a method accessing a variable. Depending on the level of reification (see Table 1, p. 4), that variable may be an attribute, a local variable, an argument, a global variable…. What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Association, it has the following attributes:

- `accessesVariable: Name; mandatory`
  Is a name referring to the variable being accessed. It uses the uniqueName of the variable as a reference.

- `accessedInMethod: Name; mandatory`
  Is a name referring to the method doing the access. It uses the uniqueName of the variable as a reference.

- `isAccessLValue: Boolean; optional, default = 'null'`
  Is a predicate telling whether the value was accessed as Lvalue, i.e. a location value or a value on the left side of an assignment. When the predicate is true, the memory location denoted by the variable might change it value; false means that the contents of the memory location is read; null means that it is unknown.
  Note that LValue is the inverse of RValue.

- `isArgument: Boolean; mandatory, default = 'false'`
  Is a predicate telling whether the variable access is contained within an ArgumentDescriptor (see p.12). The default value is false and users need to concern about this predicate only when they reify argument passing on method invocations (see Table 1, p. 4).

4.10. Core Association: MethodInvocation

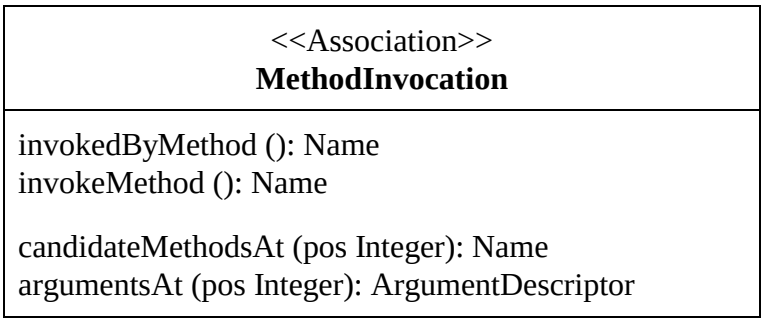

Figure 11: MethodInvocation

A `MethodInvocation` represents the definition in source code of a method invoking another method. What exactly constitutes such a definition is a language dependent issue. However, it is important to note that due to late binding polymorphism, a method invocation is quite different from a procedure call. Especially, it means that at parse time there exist a one-to-many relationship between the method invocation and the actual method invoked. This explains the presence of the `candidateMethods` aggregation.

Besides the attributes inherited from the meta-class Association, it has the following attributes:

- `invokedByMethod: Name; mandatory`
  Is a name referring to the method doing the method invocation. It uses the uniqueName of the method as a reference.

- `invokeMethod: Name; mandatory`
  Is a name holding the name of the method invoked. Due to late binding polymorphism, the name of the invoked method is not enough to assess which method is invoked. Further analysis based on the arguments is necessary.

- `candidateMethods: 0 .. N Name`
  Is a multi-valued attribute holding a number of method names. Each method name refers to a method that may be the actual method invoked at run-time. When the collection is empty, it does not mean that no candidate method exists —since we assume to start from a working system, all methods invoked should exist. Rather an empty collection denotes that a parser did not perform the extra analysis to compute the candidate methods.

- `arguments: 0 .. N ArgumentDescriptor`
  Is a multi-valued attribute holding a number of argument descriptors. Each argument descriptor gives more information about an argument being passed. When the collection is empty it does not mean that no arguments are passed —since each method invocation involves at least has one receiver there is always one argument, otherwise it is just a procedure call. Rather, an empty collection denotes that a parser did not extract information about the arguments.

4.11. ArgumentDescriptor, ComplexExpression & SimpleVariableAccess

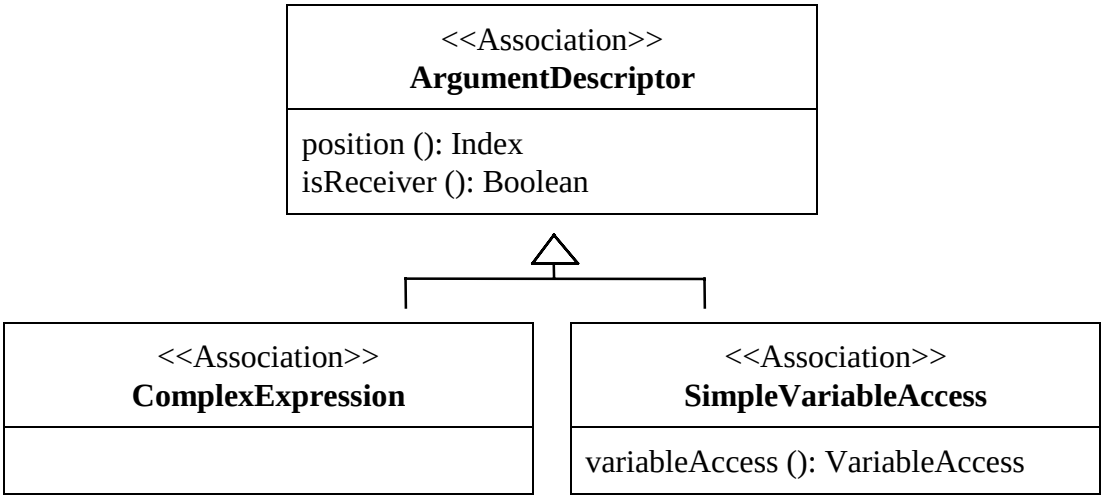

Figure 12: ArgumentDescriptor, ComplexExpression & SimpleVariableAccess

An `ArgumentDescriptor` represents the passing of an argument when invoking a method. What exactly constitutes such a definition is a language dependent issue. The model distinguishes between two kind of arguments, a complex expression or a simple variable access. The former means that some expression is passed, in that case the contents of the expression is not further specified. The latter means that some variable is passed, in which case a variable access is maintained.

Besides the attributes inherited from the meta-class Association, it has the following attributes:

- `position: Index; mandatory`
  The position of the argument in the list of arguments.

- `isReceiver: Boolean; mandatory`
  Is a predicate telling wether this argument plays the role of the receiever in the containing method invocation.

- `variableAccess: VariableAccess; mandatory`
  Holds an instance of variable access.

4.12. VariableDefinition Hierarchy

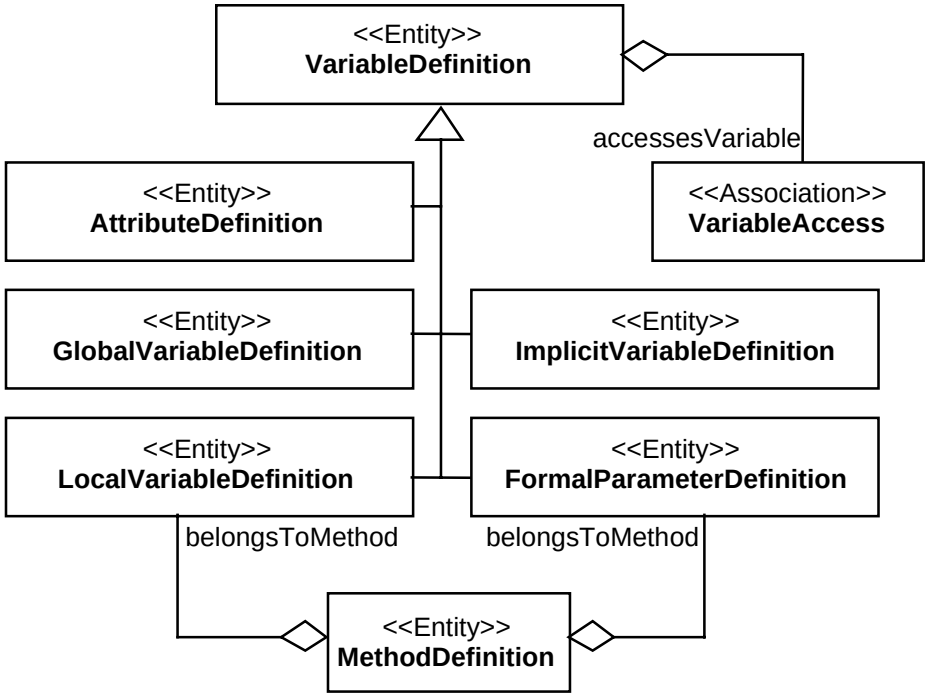

Figure 13: VariableDefinition Hierarchy

All possible variable definitions are subclasses of the class VariableDefinition. VariableDefinition itself participates in the VariableAccess association.

4.13. VariableDefinition

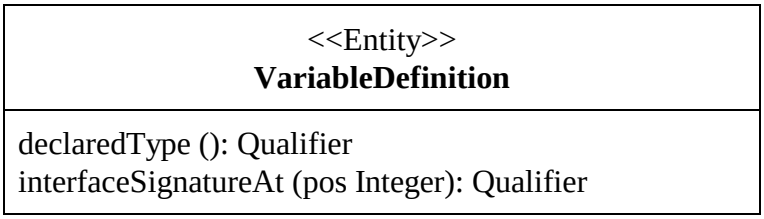

Figure 14: VariableDefinition

A `VariableDefinition` represents the definition in source code of a variable, i.e. a named memory location. Subclasses of this class represent different mechanisms for defining such a variable. Besides the attributes inherited from the meta-class Entity, it has the following attributes:

- `declaredType: Qualifier; optional, default = 'null'`
  Is a qualifier that via interpretation outside the model refers to the class of the returned object.
  Note that we need a language dependent interpretation to link a typename to a class name, because in most OO languages, types are not always equivalent to a class. How the declaredType may be recognised in source code and how the type matches to a class are language dependent issue.

- `interfaceSignatures: 0 .. N Qualifier`
  Is a multi-valued attribute holding a number of method signatures. Each method signature tells that this method is actually invoked with that variable playing the role of a receiver. This should allow to compare the public interface of the declared type with the interface that is really used from the outside.
  A parser may decide not to extract information about the method signatures, in that case the collection is empty. However, an empty collection may also imply that no methods are invoked on that variable.

4.14. GlobalVariableDefinition

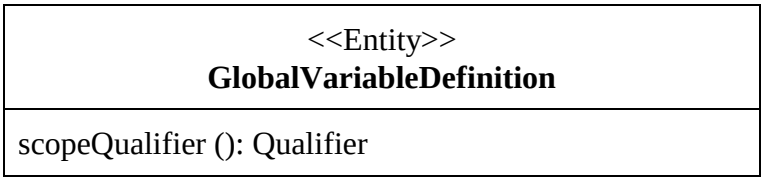

Figure 15: GlobalVariableDefinition

A `GlobalVariableDefinition` represents the definition in source code of a variable with a lifetime equal to the lifetime of a running system, and which is globally accessible. What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Entity and the class Variable, it has the following attributes:

- `scopeQualifier: Qualifier; optional, default = 'null'`
  Is a string with a language dependent interpretation, that defines a possible scope of the variable. A null scopeQualifier is allowed, it means that the variable must not be explicitly imported before using it. The scopeQualifier concatenated with the name of the variable must provide a unique name for that variable within the model.

- formula for `uniqueName`

```
if isNull (scopeQualifier(globalVariable)) then
  uniqueName (globalVariable) = name (globalVariable)
else
  uniqueName (globalVariable) = scopeQualifier (globalVariable)
    + "." + name (globalVariable)
```

4.15. ImplicitVariableDefinition

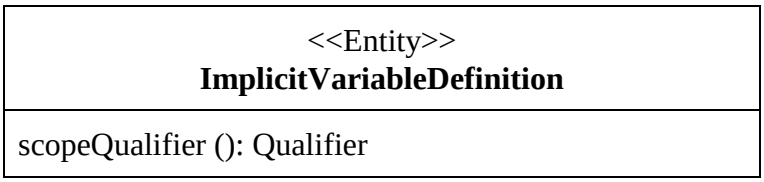

Figure 16: ImplicitVariableDefinition

An `ImplicitVariableDefinition` represents the definition in source code of context dependent reference to a memory location (i.e., 'this' in C++ and Java, 'self' and 'super' in Smalltalk). What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Entity and the class Variable, it has the following attributes:

- `scopeQualifier: Qualifier; optional, default = 'null'`
  Is a string with a language dependent interpretation, that defines a possible scope of the variable. A null scopeQualifier is allowed, it means that the variable has universal scope. The scopeQualifier concatenated with the name of the variable must provide a unique name for that variable within the model.

- formula for `uniqueName`

```
if isNull (scopeQualifier(implicitVariable)) then
  uniqueName (implicitVariable) = name (implicitVariable)
else
  uniqueName (implicitVariable) =
    scopeQualifier (implicitVariable)
      + "." + name (implicitVariable)
```

4.16. LocalVariableDefinition

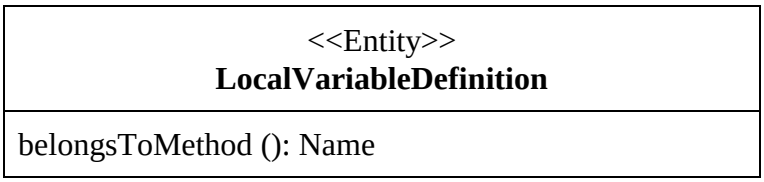

Figure 17: LocalVariableDefinition

A `LocalVariableDefinition` represents the definition in source code of a variable defined locally to a method. What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Entity and the class Variable, it has the following attributes:

- `belongsToMethod: Name; mandatory`
  Is a name referring to the method owning the variable. It uses the uniqueName of the method as a reference.

- formula for `uniqueName`

```
uniqueName (localVar) = belongsToMethod (localVar) +
  "." + name (localVar)
```

4.17. FormalParameterDefinition

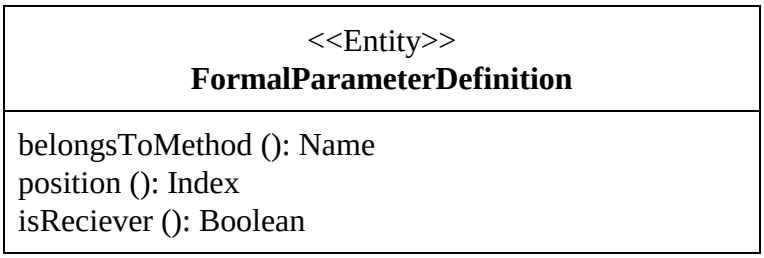

Figure 18: FormalParameterDefinition

A `FormalParameterDefinition` represents the definition in source code of a formal parameter. What exactly constitutes such a definition is a language dependent issue. Besides the attributes inherited from the meta-class Entity and the class Variable, it has the following attributes:

- `belongsToMethod: Name`
  Is a name referring to the method owning the variable. It uses the uniqueName of the method as a reference.

- `position: Index; mandatory`
  The position of the parameter in the list of parameters.

- `isReciever: Boolean; mandatory; default = 'false'`
  Is a predicate telling whether the parameter plays the role of the receiver. This is required in those cases where the receiver is not passed via an implicit variable, which is quite unusual hence the default null value.

- formula for `uniqueName`

```
uniqueName (formalPar) = belongsToMethod (formalPar) +
  "." + name (formalPar)
```

5) Open Questions

5.1. Why not UML?

The unified Modelling Language (UML) [BoochEtAl'96] 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 during the model capture phase of the re-engineering life cycle (see Figure 1). The FAMOOS project 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.

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 variable (Smalltalk) and we talk about classes instead of types (Ada).

5.2. Why not CORBA/IDL?

Corba is receiving widespread attention as interoperability standard between different object-oriented implementation languages. The IDL (interface description language) is used to specify the external interface of a software component and there are tools that extract IDL from source code. As such, Corba/IDL is a viable candidate to serve as our exchange format.

However, Corba/IDL only describes the interface of a software component, not the internal dependencies like method invocations and variable accesses. Those dependencies are necessary in the problem detection and reorganisation phases of the re-engineering life cycle (Figure 1). Thus, choosing Corba/IDL would violate the requirement of being a sufficient basis for re-engineering operations (see p.2).

5.3. What about Dynamic Information?

Because of late binding polymorphism, not all method invocations can be resolved at compile time. Also, a model based on source code is not ideal for identifying sequences of interactions between objects. Thus, basing the model solely on static information eliminates some interesting facts about a software system and one might consider including run-time information as well.

For the moment we consider the issue too premature to include in an information exchange standard. The technology is available (i.e., Look for C++, method wrappers for Smalltalk) but is certainly not part of the standard tool repertoire. And extracting run-time information generates such a wealth of data that we cannot asses what is important enough to maintain.

5.4. How do you handle hybrid languages (C++, Ada...)?

Some OO languages are extensions of older procedural languages, and as such allow a hybrid programming style. Part of the object-oriented re-engineering problem is precisely that programmers did not use object-oriented constructs where it would have been advantageous. For problem detection, it might be worthwhile to include procedural constructs in the model.

For the moment we decided to ignore the issue. We have some ideas on expressing procedural programming constructs as degenerated object-oriented constructs (e.g., define a procedure as a method defined on a dummy class) but no concrete proposal in that direction.

6) References

6.1. FAMOOS Internal References

[DETECTM] FAMOOS Achievement Report DETECTM-A.2.3.2. " Specification of Techniques and Strategies for Problem Detection". Benedikt Schulz, Forschungszentrum Informatik.

[DOCUM] FAMOOS Achievement Report DOCUM-A.2.3.1. " Documentation and Model Capture Method(Grouping)". Oliver Ciupke, Forschungszentrum Informatik.

[EVALCDIF] FAMOOS Achievement Report EVALCDIF "Evaluation of the CDIF Transfer-Format". Thomas Kohler, Daimler-Benz AG.

[REORGOP] FAMOOS Achievement Report REORGOP-A.2.3.3./A.2.3.4. " Specification of Complex Re-engineering Operations and Target Structures ". Joachim Weisbrod, Forschungszentrum Informatik.

6.2. External References

[BoochEtAl'96] Booch, G., Jacobson, I. and Rumbaugh, J, The Unified Modelling Language for Object-Oriented Development. See http://www.rational.com/.

Cover Pages

Achievement A2.4.1

Definition of a Common Exchange Model

1) Identification

Project Id:	Esprit IV #21975 “FAMOOS”
Deliverable Id:	D 2.2 – FINALFHB Final FAMOOS Methodology Handbook
Date for delivery:	31.03.98
Planned date for delivery:	31.03.98
WP(s) contributing to:	2
Author(s):	S. Demeyer, S. Ducasse, T. Richner, M. Rieger, P. Steyaert, S. Tichelaar

2) Abstract

This document defines the exchange model for usage by tool prototypes within the FAMOOS re-engineering project. The model is based upon the CDIF standard so that it can be transferred via flat ASCII streams.

3) Keywords

Object-oriented, reengineering, reverse engineering, code repository, FAMOOS.

4) Version History

Ver	Date	Editor(s)	Status & Notes
0.4	17.11.97	S. Demeyer; P. Steyaert	First draft version. Released to all the participants of the Ulm-workshop (21.11.97).
0.5	24.11.97	S. Demeyer	Quick tour of revised model; incorporates feedback generated during workshops at FZI (20.11.97) and Daimler-Benz (21.11.97).
0.6	09.01.98	S. Demeyer	Expanded quick tour into a full specification. Changed original document template for convenient generation of HTML.

			Document is now ready for reviewing and defining language plug-ins.
1.0	30.03.98	S. Demeyer	Final release: • Incorporated feedback given on prior release. • Adapted meta-model to be streamlined with CDIF; removed examples, we first need some tool experience with CDIF. • Introduced the notion of “level of reification”.

7) Table of Contents

8) List of Figures

9) List of Tables