# Metaprogramming in the Large

Andreas Ludwig Dirk Heuzeroth
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe

**Abstract.** Software evolution demands continuous adaptation of software systems to continuously changing requirements. Our goal is to cope with software evolution by automating program transformation and system reconfiguration. We show that this can be achieved with a static metaprogramming facility and a library of suitable metaprograms. We show that former approaches of program transformations are not sufficient for large object oriented systems and outline two base transformations that fill the gap.

## 1 Introduction

Many software systems have grown enormously large in the last decades. Requirements for these systems have changed continuously, and still do. Consequently, the software needs to be adapted all the time: software evolves. However, changing software is a complex and error prone task and is likely to produce high costs. One reason for this are lots of dependencies in typical programs, which make them hard to understand. Moreover, the effects of changes to a system are hard to predict. This is especially true for large changes like system restructuring, addition, removal and exchange of components and subsystems as well as composition of components. But even small modifications like changing the signature of a single method may show effects throughout the whole system: for instance, a changed method parameter type can not only influence all references to that method but also all references to overloaded methods which in turn might be defined in any subclass. To deal with those effects, we require global information about the program and must hence operate in the large. We provide further examples in sections 2 and 3.

Software evolution means transforming programs. Our group developed a static metaprogramming facility called COMPOST that provides a library of useful analyses and transformations. We found metaprogramming to be a natural and very promising strategy to carry out changes consistently and thus support both software evolution and software construction.

To cope with software evolution and construction, software must be easy to change. We therefore focus on software changes and start with a concrete scenario, the evolution from JDK 1.0 to JDK 1.3 (section 2). After that, we identify some program transformations that are needed for our purposes, especially to

deal with imperative object oriented programs (section 3). We then outline the infrastructure needed to actually perform those changes (section 4). Finally, we discuss related work and possible uses of metaprogramming in these areas, which we cannot cover in detail here (section 5). We conclude with our experiences and suggestions for future work (section 6).

## 2 Software Evolution Scenario

Assume that a new version of a component offers improved behavior, but also comes with a changed interface. In order to use the revised version, all using contexts of this component have to be updated. Now assume that the programs that have been using the old component are also maintained. Then, binary compatibility is not sufficient. Instead, the programs must be brought to the state of the art at source level.

**Example 1 (JDK update)** There have been about 300 changes in the API of the standard Java libraries up to now — a complete list is available at `java.sun.com/j2se/1.3/docs/api/deprecated-list.html`. These interface changes affect all pieces of Java software that rely on the now "deprecated" features; we classify these changes as follows and will study them in detail later on:

① Local changes of signatures (name or parameters), e. g. in java.awt.Component:

- move(int, int) → setLocation(int, int)
- disable() → setEnabled(false)
- add(String, Component) → add(Component, Object)

② Change of the location of features, e.g. in java.awt:

- Frame.CROSSHAIR_CURSOR → Cursor.CROSSHAIR_CURSOR
- Frame.getCursorType() → Component.getCursor().getType()

③ Removal or replacement of features, e.g.

- java.awt.Component.getPeer() → ∅ (deleted)
- java.io.StringBufferInputStream → java.io.StringReader

④ Change of an entire framework, using multiple design patterns, such as the new event handling framework that replaces a template method based solution by an observer pattern, which concerns some dozen classes in several packages.

How can we automate an update for a large system? The solution is at hands: Let the programmer of the component write a "smart patch" metaprogram that is able to update arbitrary programs.

The scenario illustrates what we consider metaprogramming in the large. In this setting, we have a big advantage, as some of the semantics of the system to change is known. Assuming that the old versions of the components have been used correctly allows to assert many predicates that would have been very hard to find out by analyses, such as object lifetime dependencies. After some basic transformations have been detailed, we illustrate how this knowledge can be used in the second part of the example.

## 3 New Program Transformations

Program transformations have been investigated for years (for a good overview see [CE00]). We discuss how goals and languages have changed since then and now require further transformations. We then sketch an abstract program model to deal with object oriented languages, and finally discuss two fundamental transformations based upon that model.

### 3.1 Classical Program Transformations Improved

The primary research focus of classical program transformations was program refinement and optimization, as well as the creation of extensible languages. The most important achievements are a set of base transformations such as *add definition*, *fold*, *unfold*, *function generalization* and *specialization*, and techniques to resolve recursions. These transformations were based upon a functional programming model and local contexts only, to avoid the problems introduced by *states* and *scopes*. However, states and scopes are inherent parts of imperative object oriented languages. While the known base transformations reappear in more specific forms to suit the rules of the concrete language, further *base transformations* are required to cover states and scopes.

Transformations come with a set of *constraints* restricting their use, e. g. prohibition of ambiguous names. Single transformations often lack constructive strategies how to fulfill these conditions; usually, only a combination of transformations is applicable. Thus, the goal is to collect sufficient information about the context to derive applicable chains of transformations to achieve the desired effects.

### 3.2 Nested Feature Model For Object Oriented Languages

The notions of component interfaces and scopes are somewhat dependent on the concrete language. Therefore, we introduce an abstract program model consisting of hierarchically nested *features* such as classes, variables, and methods, which may define a scope (see Figure 1). We may regard packages, constructors, attributes or parameters as special cases of classes, methods, or variables. To complete this simplistic model, we distinguish *declarations* of features and *references* to features which are contained within a scope, and assume that each feature has at most one definition, but arbitrarily many references. The exact rules for feature declarations, feature nesting, and feature reference resolution (including feature visibilities, overloading, and inheritance) are part of the language specification. In our experiments, we used Java as a representative object oriented language.

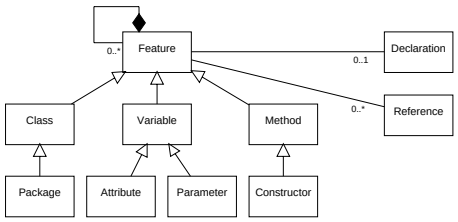



**Fig 1.** Simple Nested Feature Model

### 3.3 A Transformation to Deal with States

The feature model has no visible states to manipulate directly. However, the state transitions defined by variable and method accesses can be used to bring in new states. These accesses may be embedded into arbitrary expression terms. To avoid side effects in the partial expressions that are evaluated prior to the feature access, we need a transformations that shifts these parts and *flattens* the syntax tree of the expression term. The following example illustrates this.

**Example 2 (Expression flattening)** Consider the expression `x = f(g(y), h(k(z))` and assume we want to insert some code before the very moment when `f` is accessed. The transformation should produce: `int i = g(y); boolean b = h(k(z)); /* some new code */ x = f(i, b);` — now the reference to `f` is the first expression that is evaluated in the statement. Note that the types of `g` and `h` cannot be derived from the original expression, and that it was not necessary to flatten `k`.

The transformation takes a reference to a variable or method as argument and flattens the top-level expression term this reference is contained within:

1. Collect all expressions that become evaluated before the relevant feature access and that may have side effects.
2. For each largest expression `e` in the order of evaluation:
   (a) Calculate the type $\tau$ of `e`
   (b) Create a new unambiguous temporary variable `v` of type $\tau$ and insert the declaration of `v` in front of the the top-level expression.
   (c) Insert `e` as initialization code of the variable.
   (d) Substitute the original occurrence of `e` with a new reference to `v`.

This code transformation allows to modify arbitrary accesses to features by simple insertion of new code. It is easy to see how it can be used for instrumentalization or to exchange method calls by other means of communication.

Note that this transformation requires context sensitive information to derive the types of expressions, and to choose an unambiguous variable name.

### 3.4 A Transformation to Deal with Scopes

The most general and basic transformation that is able to reconfigure interfaces is *move feature declarations*.[1] The transformation consists of two subtasks, which are also base transformations: creating new feature declarations in the new scope, and deletions of old declarations. Note that we explicitly allow inheritance, feature hiding, overloading, and combinations thereof.

1. Insert feature declarations in the new scope
   (a) Check if a declaration is forbidden due to ambiguous names in the new scope, or incompatible modifiers or signatures while attempting to overwrite a feature of identical name in a superclass. If so, rename the feature, or optionally change modifiers in the superclass.

[1] Special cases of this transformation are already depicted in [FBB+99] as refactorings. The version presented here is more general and more detailed.

(b) Check if a definition hides less general definitions that are overloaded or overwritten. If so, rename the new features, or strengthen qualification of references to the hidden feature (add type casts or access qualifiers).

2. Remove feature declarations from the old scope
   (a) Check if there are references from the moved features to other features in the old scope. If so, redirect them to the old scope using a proper access path. If no predefined access path is available, create a new one by adding an additional parameter to each feature. If necessary, adapt visibilities of the used features to allow access from the new scope. Alternatively, compute a self-dependent *feature group* and ask the user if this group shall be moved instead.[2]
   (b) Check if there are external references to the features in the old scope. If so, redirect them to the new scope, by using a proper access path.

This transformation allows to reconfigure system interfaces arbitrarily. If one reads design patterns as instructions how to change a system rather than creating one from scratch, then many of them are built upon this transformation.

Finding access paths is a key issue in this transformation when the scopes are classes. In many cases, access paths are predefined, either by the caller of the transformation, or if features are global, or the scopes are nested or inherited (see Figure 2) — then, the underlying objects are either accessible from anywhere, identical, or share the same life time.

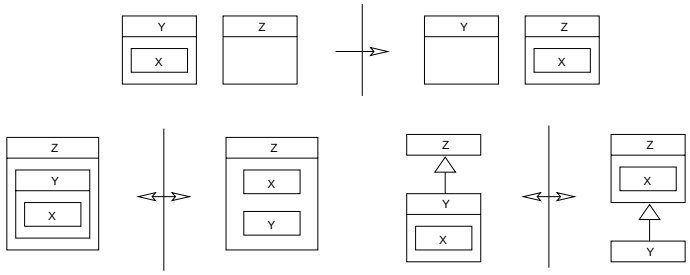



**Fig 2.** Feature Movement Schemes – General Case and Two Specializations (below) — note that the admissible feature types of x, y and z may vary.

In the general case, no predefined access path is available. Sometimes, it is possible to identify a unique path candidate that is free of possible side effects. If there are several solutions, the choice is a genuine design decision and should be left to the user. If there is not even a possible candidate, the transformation can attach a new attribute and insert some initialization code for the target object, if the initialization is not ambiguous.

We learn two lessons from this example: First, user interaction is required when analyses fail and design alternatives must be chosen. Second, a non-trivial transformation requires a lot of subtasks which have to be addressed. Figure 3 illustrates some auxiliary functions.

[2] This is the reason why moving a set of declarations can be easier than a single one.

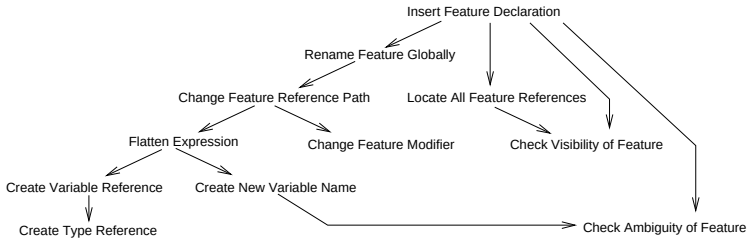



**Fig 3.** Transformation subtasks

**Example 3 (JDK update revisited)** Equipped with the transformations just presented and additional knowledge about concrete components that we have to deal with, we can now try to find a metaprogram that would have saved enormous amounts of programming time...

① Local changes of signatures: Renaming is a task similar to addition of a new feature, while a change of parameters requires a change of arguments on the caller side — flattening the expressions that might have side effects is the most difficult part.

② Change of the location of features: The examples given are easy to handle with feature moves as we have to deal with global constants, or already know a proper replacement access path.

③ Removal or replacement of features: In the case of the deprecated Peer classes, there is no obvious update. However, the only proper use of the peers has been to check for consistent initialization state. We can attempt to pattern match for c.getPeer() != null and replace this expression by c.isDisplayable(). The second example concerning the replacement of an improper Unicode stream, is also hard to resolve, as different data types (byte vs. char) are involved. But again, all meaningful uses should not rely on concrete bit counts, so a simple reassignment of methods to their counterparts is sufficient. Only one method features a different signature: read(char[],int,int) replaces read(byte[],int,int). Typical uses of the old version will have translated the byte array into a string or a character array anyway, but to ensure compatibility, we would have to insert some translation code. We have to attach some glue code to calls of the form in.read(arr, off, len) in order to obtain:

```
char[] tmp = new char[arr.length];
in.read(tmp, off, len);
System.arraycopy(new String(tmp, off, len).getBytes(), 0, arr, off, len);
```

There are some details to get right, such as adding a new statement block when necessary and getting rid of side effect expressions in access paths.

④ Change of an entire framework: Beginning with JDK 1.1, event handling changed to an observer pattern replacing redefined methods of the corresponding components. The following code snippets show a simple solution obtained by some feature moves. The former action method in class C

```
public boolean action(Event e, Object what) {
    // action code here
    return super.action(e, what); // or other expression
}
```

is transformed into a new attribute declaration in C

```
ActionListener actions = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // user code from super.action goes here
        // action code here; 'what' redirected to e.getActionCommand()
        // no return value here - no need to consume events
    }
};
```

plus an additional call addActionListener(actions); in each constructor of C.

Example 3 shows that an automatic update is actually possible while human interaction is necessary only in few cases.

In the following section, we discuss the specific requirements and challenges for a technical infrastructure supporting the construction of metaprograms.

## 4 Object Oriented Metaprogramming Realized

The general procedure of source to source transformations is illustrated in Figure 4: A compiler frontend produces a metamodel from the sources, which includes all syntactic data necessary for the pretty printer to reproduce the sources later on. Transformations can use any derived information to change the syntactic parts.

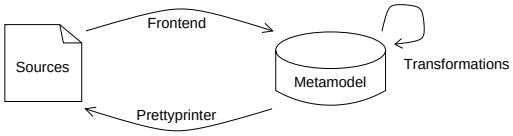



**Fig 4.** Source Level Transformations

### 4.1 Requirements for Metaprogramming Systems

Metaprograms must cover some important requirements, and these must be considered when supplying tools to facilitate the construction.

**Readability:** A metaprogramming system changes source codes. The result should obey coding conventions and should not destroy formatting. In this setting, blank spaces become important and must be protected. In general, "code bloat" should be kept to a minimum.

**Scalability:** In contrast to compilers, metaprogramming systems change the syntactic base of the metamodel during runtime. Hence, when used in an interactive and incremental environment, information derived from the syntactic model must also be updated incrementally. To retain scalability, it is not feasible to invalidate the whole model after a change. Instead, it is necessary to carefully analyze *change impacts* to prevent transitive propagation throughout the model. For instance, a change of a feature declaration may change the assignment of all references to the feature and all hidden ones.

Note that the smart patch scenario is not necessarily incremental — usually it is possible to do all necessary analyses in advance and delay the transformations.

**Extensibility:** There are lots of possible transformations motivated by high level design problems. A library of metaprograms is therefore unlikely to be complete, but it should be extensible. A proper system design should facilitate the creation of metaprograms. For instance, the metamodel interface should hide the process of information retrieval, offer navigational links, and provide meaningful abstractions over the concrete representations. Auxiliary transformations, analyses and code snippet generators should become part of the library.

### 4.2 Limitations of Metaprogramming Systems

Source level metaprogramming suffers from some general problems.

*Source code hygiene* and especially comment assignment are an issue. Comments will not always be assigned properly and may move with the wrong element. Also, it will not always be possible to avoid some redundant code.

In general, *comments and documentation* do not obey a formal semantics and hence cannot be maintained automatically. However, it is possible to automate changes of designated, formal parts. For instance, the Java documentation conventions require to tag feature references with `@see` or `@link` making them accessible for analysis and transformations.

*Analyses are conservative* and may fail to derive enough information to do certain decisions automatically. Then, interaction with the human programmer is needed. Sometimes an alternative approach can be found, which might be far from perfect, but functional — e. g. to insert a new attribute and ignore an uncertain candidate. Possible improvements could then be offered in an interactive session after the main transformations have taken place.

### 4.3 COMPOST - An Implementation

We have been constructing a metaprogramming system for the Java programming language called COMPOST (COMPOsition SysTem). The system is available from `i44www.info.uni-karlsruhe.de/~compost` and consists of nearly 600 classes. It contains a Java frontend providing facilities to resolve feature references and for cross reference information. The pretty printer is configurable to fit specific conventions and is able to reproduce (or improve upon) the original input and to insert new code seamlessly. We have built a library of auxiliary functions and transformations also containing the auxiliaries from Figure 3.

Transformations are responsible for the maintainance of the abstract syntax model, while the pretty printer handles the concrete syntax, and lookup services update the derived semantic information. Transformations report the syntactic effects of their changes, and this information is interpreted by a responsible service as soon as new semantic information is required.

### 4.4 Experiences

Memory consumption of a metaprogramming system is high, but does not exceed that of an optimizing compiler, and seems to scale with the size of the program. We were able to analyze 3000 classes on a state of the art workstation.

Naive incremental model updates are sufficient for a handful of classes only, but more sophisticated algorithms should be able to handle a big number of classes, as some sparse experiments in literature have indicated [BS86] [Wag98].

We have chosen Java as our primary target language, but metaprogramming should be applicable for all programming languages. In general, languages providing many static predicates allow to do more decisions automatically based on the additional knowledge represented by these predicates.

## 5 Related Work

We already compared metaprogramming with program transformation systems in section 3. Now we discuss related work in the area of software evolution.

Extreme programming (XP) [Bec99] is a development process to support evolutionary programming. XP relies on testing to assure the correctness of changes. Verified metaprograms would no longer require additional tests.

The base mechanics of XP are refactorings [FBB+99]. Refactorings are program transformations that optimize for readability. To create sophisticated tooling, the strategies given in the catalogues are still too vague and heavily rely on the programmer. Compiler construction research can provide solutions for many problems, but we need a more formal background to do so. The feature movement transformation is an attempt to generalize many special cases.

While refactorings are too local to have a meaningful motivation by themselves, design patterns [GHJV95] are complete development processes leading from a single design problem down to implementations. In many cases, the pattern can be regarded as an instruction to change software, using a series of refactorings and similar transformations [Zim97]. If this is possible, then we can derive a metaprogram instantiating the pattern. For instance, moving features is fundamental to create new levels of indirection widely used in patterns.

Architectural systems [SDZ95] allow to exchange *connectors* representing a communication association. Similarly, composition languages [NM95] focus on the wiring of software components. Metaprograms can easily exchange connectors or compose programs avoiding special languages.

Generic programming allows to bind parameters embedded in the code with concrete syntactic content. Metaprogramming can simulate the instantiation of a parameter and even attempt to identify the replacement locations even if they were not tagged as parametric.

Aspect oriented programming [KIL+97] or multidimensional separation of concerns [OT99] intend to offer multiple views on the (abstract) program that might not be identical to the primary functional modularization. Each aspect, or concern, can be changed separately, and must then be merged, or woven, with the other aspects to derive the final model. This weaving process can be implemented by a metaprogram.

In this work we concentrated on the support of transformations for refactoring software systems rather than the configuration of system families. The latter is subject of the generative programming paradigm [CE00].

## 6 Conclusion

We showed that program transformations realized by metaprograms significantly facilitate consistent and correct changes and that this approach is a well suited technique for software evolution. Furthermore, we demonstrated that metaprogramming fits seamlessly into emerging design techniques such as design patterns, separation of concerns, and refactorings.

We identified two important basic transformations that are necessary for reconfiguring large imperative object oriented systems. We also showed that metaprogramming is feasible for real world languages with the right tool support. We described the infrastructure necessary for source to source metaprogramming. COMPOST is an implementation of such an infrastructure.

Future work comprises the identification of further base transformations for object oriented programs as well as fast incremental model updates since there is still little experience in this area.

## References


[Bec99] Kent Beck. *extreme Programming explained.* Addison Wesley, Reading, MA, 1999.

[BS86] Rolf Bahlke and Gregor Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM TOPLAS*, 8(4):547–576, October 1986.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming.* Addison Wesley, 2000.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, MA, 1995.

[KIL+97] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented Programming. In *ECOOP'97*, pp 220–242. Springer-Verlag, 1997.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In O. Nierstrasz P. Ciancarini and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer-Verlag, 1995.

[OT99] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical report, IBM T.J. Watson Research Center, 1999.

[SDZ95] Mary Shaw, Robert DeLine, and Gregory Zelinski. Abstraction and Implementation for Architectural Connections. Technical report, CMU, 1995.

[Wag98] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments.* PhD thesis, Computer Science Division, EECS Department, University of California, March 1998.

[Zim97] Walter Zimmer. *Frameworks und Entwurfsmuster.* PhD thesis, Universität Karlsruhe, February 1997.