The Story of Moose: an Agile Reengineering Environment

Oscar Nierstrasz
Software Composition Group
University of Berne
Switzerland

Stéphane Ducasse
Software Composition Group
University of Berne
Switzerland

Tudor Gîrba
Software Composition Group
University of Berne
Switzerland

www.iam.unibe.ch/∼scg

ABSTRACT

Moose is a language-independent environment for reverse- and re-engineering complex software systems. Moose provides a set of services including a common meta-model, metrics evaluation and visualization, a model repository, and generic GUI support for querying, browsing and grouping. The development effort invested in Moose has paid off in precisely those research activities that benefit from applying a *combination* of complementary techniques. We describe how Moose has evolved over the years, we draw a number of lessons learned from our experience, and we outline the present and future of Moose.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance—*Restructuring, reverse engineering, and reengineering*

General Terms

Measurement, Design, Experimentation

Keywords

Reverse engineering, Reengineering, Metrics, Visualization

1. INTRODUCTION

Software systems need to evolve continuously if they are to be effective [41]. As systems evolve, their structure decays, unless effort is undertaken to reengineer them [41, 44, 23, 11].

The reengineering process comprises various activities, including model capture and analysis (*i.e., reverse engineering*), assessment of problems to be repaired, and migration from the legacy software towards the reengineered system. Although in practice this is an ongoing and iterative process, we can idealize it (see Figure 1) as a transformation through various abstraction layers from legacy code towards a new system [11, 13, 35].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Proceedings ESEC-FSE'05, pp. 1-10, ISBN 1-59593-014-0. September 5–9, 2005, Lisbon, Portugal.
Copyright 2005 ACM 1-59593-014-0/05/0009 ...$5.00.

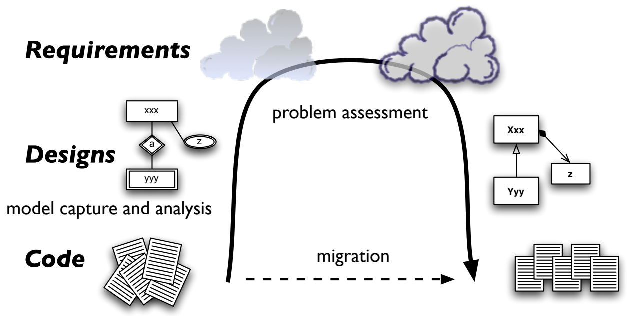


Figure 1: The Reengineering life cycle.

What may not be clear from this very simplified picture is that various *kinds* of documents are available to the software reengineer. In addition to the code base, there may be documentation (though often out of sync with the code), bug reports, tests and test data, database schemas, and especially the version history of the code base. Other important sources of information include the various stakeholders (*i.e.,* users, developers, maintainers, *etc.*), and the running system itself. The reengineer will neither rely on a single source of information, nor on a single technique for extracting and analyzing that information [11].

Reengineering is a complex task, and it usually involves several techniques. The more data we have at hand, the more techniques we require to apply to understand this data. These techniques range from data mining, to data presentation and to data manipulation. Different techniques are implemented in different tools, by different people. An infrastructure is needed for integrating all these tools.

Moose is a reengineering environment that offers a common infrastructure for various reverse- and re-engineering tools [22]. At the core of Moose is a common meta-model for representing software systems in a language-independent way. Around this core are provided various services that are available to the different tools. These services include metrics evaluation and visualization, a repository for storing multiple models, a meta-meta model for tailoring the Moose meta-model, and a generic GUI for browsing, querying and grouping.

Moose has been developed over nearly ten years, and has itself been extensively reengineered during the time that it has evolved. Initially Moose was little more than a common meta-model for integrating various *ad hoc* tools. As it became apparent that these tools would benefit immensely from a common infrastructure, we invested in the evolution and expansion of MOOSE. This expansion faced numerous challenges, particularly that of scalability, since legacy systems tend to be large. Without extensive reengineering, however, MOOSE itself would have quickly become unwieldy and unmanageable. Instead, we have managed to keep the core of MOOSE quite small, and only extended its functionality when there was a clear added value to be obtained. In its latest version MOOSE has 217 implementation classes and 78 test classes.

Common wisdom states that one should only invest enough in a research prototype to achieve the research results that one seeks. Although this tactic generally holds, it fails in the reengineering domain where a common infrastructure is needed to even begin to carry out certain kinds of research. In an nutshell, the added value arises precisely when multiple techniques can be combined to break new research ground.

From another perspective, the research process consists of formulating an hypothesis based on observations, and performing an experiment to evaluate the hypothesis. The shorter the distance between the formulation and the evaluation of an hypothesis, the more hypotheses can be explored. With MOOSE, one can reuse and extend the previous experiences and, in this way, shorten the distance between the original hypothesis and the result.

In this paper we attempt to draw some lessons from our experience developing and using the MOOSE environment. In Section 2 we outline the current architecture of MOOSE, and we briefly recount the history of its evolution. Then, in Section 3 we see how MOOSE has been used to carry out an array of different reverse- and re-engineering research projects. In each case, we can see that the investment in MOOSE paid off precisely at the point where multiple techniques are being combined to achieve a particular research goal. In effect, we could greatly accelerate the research activities because we had an infrastructure upon which we could build, even though this infrastructure was at any point in time rather minimal. In Section 4 we summarize some of the lessons learned. We conclude in Section 5 with some remarks on our ongoing research activities.

2. MOOSE

MOOSE was born in the context of FAMOOS, a European project[1] whose goal was to support the evolution of first-generation object-oriented software towards object-oriented frameworks. FAMOOS focussed on methods and tools to analyse and detect design problems in object-oriented legacy systems, and to migrate these systems towards more flexible architectures. The main results of FAMOOS are summarized in the FAMOOS Handbook [13] and in the "Object-Oriented Reengineering Patterns" book [11].

2.1 What is Moose?

MOOSE is a language-independent environment for reverse- and re-engineering legacy software systems [15]. In essence, MOOSE functions as a repository for software models, providing numerous services for importing, viewing, querying and manipulating these models [22].

[1]ESPRIT Project 21975: "Framework-based Approach for Mastering Object-Oriented Software Evolution". Sept. 1996-Sept. 1999.

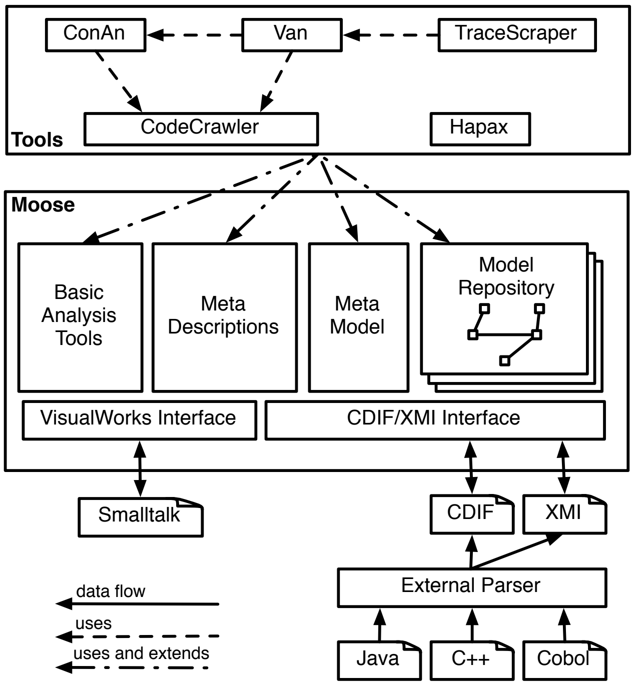


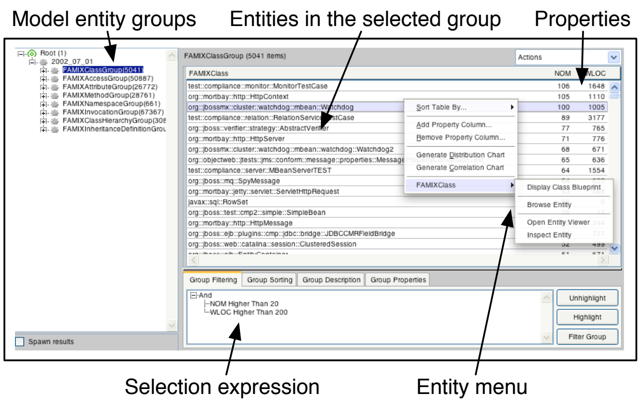
Figure 2: The architecture of MOOSE.

At the core of MOOSE is the language-independent meta-model. The meta-model determines how software systems are modeled within MOOSE. MOOSE focuses on object-oriented software, so its meta-model provides object-oriented modeling features. At the same time, MOOSE is designed to support reverse- and re-engineering activities, so the meta-model includes features for modeling, for example, method invocations and attribute accesses [57, 56].

Software source code, written in different programming languages, can be parsed using various third-party tools, and imported into MOOSE by means of the CDIF or XMI exchange formats.

Although MOOSE mainly focuses on object-oriented systems, one important requirement is that it not impose a fixed meta-model, but rather provide the infrastructure to *extend the meta-model* according to the needs of the analysis. We therefore take an active stand towards supporting the various needs of tools, rather than confining them to a fixed view of the world. We achieve this by providing for a MOF-like meta-meta-model. Each entity in the meta-model is described according to the meta-meta-model. Based on this description we provide services such as a generic GUI for querying and manipulating models.

The user interface offers services for basic manipulation of entities: selection, introspection and navigation. These services too make use of the meta-descriptions. As MOOSE is designed to be extended by different reengineering tools, we provide a registration mechanism for these tools. For example, each entity in the GUI offers a menu, to which relevant tools can register themselves. Thus, a visualization tool that provides for a class view, would register itself to the class entity, and the user can apply the visualization to a given class from anywhere in the environment.



FAMIXClass	NOM	WLOC
test::compliance::monitor::MonitorTestCase	106	1648
org::mortbay::http::HttpContext	105	1110
org::jbossmx::cluster::watchdog::mbean::Watchdog	100	1005
test::compliance::relation::RelationServiceTestCase	89	3177
org::jboss::verifier::strategy::AbstractVerifier	77	765
org::mortbay::http::HttpServer	71	776
org::jbossmx::cluster::watchdog::mbean::watchdog::Watchdog2	68	671
org::objectweb::jtests::jms::conform::message::properties::MessagePropertyConversionTest	65	636
test::compliance::server::MBeanServerTEST	64	1554
org::jboss::mq::SpyMessage		
org::mortbay::jetty::servlet::ServletHttpRequest		
javax::sql::RowSet		
org::jboss::test::cmp2::simple::SimpleBean		
org::mortbay::http::HttpMessage		
org::jboss::ejb::plugins::cmp::jdbc::bridge::JDBCCMRFieldBridge		
org::jboss::web::catalina::session::ClusteredSession	52	499

```
And
├NOM Higher Than 20
└WLOC Higher Than 200
```

Figure 3: Moose Browser.

Figure 3 shows the Moose Browser. In the left pane we have the model and the groups of entities. The selected group spawns on the right side a view of its entities. In our example, we have selected the group of classes of JBOSS. Everywhere in the environment, one can interact with an entity (be it a class, method or group) by spawning the entity's associated menu. In the lower part of the browser we have a query engine that can combine metrics, Smalltalk queries and logic queries. The result of a query is another group that can be manipulated with the same user interface. In the same browser, one can manipulate the entities in a spreadsheet manner, by adding columns with different properties (*e.g.,* metrics) or sorting the table. The browser is general enough to work with any type of entity, and therefore, whenever a tool extends the meta-model with a particular entity, it is provided with the default browsing capabilities.

Legacy software systems tend to be large, so scalability is a key requirement for MOOSE. One key technique that allows MOOSE to deal with large volumes of data is lazy evaluation combined with caching. For example, although MOOSE provides for an extensive set of metrics, not all of them are needed for all analyses. Even the relevant metrics are typically not needed for all entities in the system. Thus, instead of computing all metrics for all entities at once, our infrastructure allows for lazy computation of the metrics.

In MOOSE we can manipulate multiple models at the same time due to the multi-model repository. The main benefit of this feature is the ability to store the models of different versions of the system and thus to be able to analyze its evolution.

2.2 History of Moose

In the beginning of the FAMOOS project MOOSE was merely the implementation of a language independent meta-model known as FAMIX [12].

The parsing of C/C++ code was done through Sniff+ [7], and the produced models were imported via the CDIF standard [8]. Initially, MOOSE provided for a hard-coded importer and served as basis for simple visualization and program fact extractor (1997). Then it started to be used to compute metrics [9, 10]. Later on, as the meta-model evolved, it became apparent that the import/export service should be orthogonal to the meta-model and most important that the environment should support meta-model extension. As a consequence, a first, extremely simple meta-meta-model was implemented, which, at the time, could represent entities and relationships (1998).

With the introduction of the XMI standard, a first MOF meta-meta-model was implemented and CDIF meta-models were transformed into MOF meta-models for the XMI model generation. However, MOF was not used as the underlying MOOSE meta-meta-model [54].

In parallel, the visualization development led to the extension of the set of metrics computed. At the time, CODECRAWLER was the flagship application of MOOSE, and for a significant period CODECRAWLER influenced the architecture of MOOSE (1999) [40]. For example, the metrics had to be computed for all entities before the views could be generated.

The interest in researching the evolution of systems led to the implementation of the meta-model repository. As such, the first application was the Evolution Matrix (2001) [38]. Later on, more research was invested in understanding the evolution of systems, resulting in the development of VAN (2002). Because the evolution analysis requires large amounts of data to be manipulated, it was not feasible anymore to manipulate all the model information all the time. Also, the computation of the metrics beforehand for all entities in the model was another bottleneck. As a consequence, several services were implemented: partial loading of the models, lazy computation of the properties, and caching of results.

It became apparent that the meta-descriptions are a powerful way of separating the data representation (*i.e.,* the meta-model) from the different techniques to manipulate this data. We consequently started to implement a MOF-like meta-meta-model (2002) and replaced the original one. It offers an architecture similar to that of the Eclipse Modeling Framwork (EMF).

As an application of the meta-description, the development of a generic GUI was started to provide basic services such as navigation, querying, and introspection (2003). An important role in the caching mechanism and in the querying is played by the notion of a *group* as a first-class entity: every query or selection in MOOSE yields a group, and any group can be manipulated in the Browser (2003).

Because more and more tools started to be implemented on top of MOOSE, a plug-in mechanism was needed to allow these tools to complement each other without imposing a hard-coded dependency between them. Each tool can register itself to the menu attached to each entity in the meta-model.

The combination of menus and groups meant that complex analyses could be broken down into multiple steps, each of which may make use of a different tool. Combining and composing tools thereby becomes natural and transparent.

3. APPLICATIONS

MOOSE has served as the basis for a large number of research projects in the areas of reverse- and re-engineering. In this section we provide an overview of some of these applications. Most of our attention has been focused on reverse engineering and problem analysis rather than actual transformation of legacy systems, so the applications we present here reflect this focus.

3.1 Metrics and Visualization

Moose provides the basic bricks needed for several high level analyses [9, 10]. For example, Moose allows for a set of metrics to be attached to each entity. In a reverse engineering context software metrics are interesting because they can be used to assess the quality and complexity of a system and because they are known to scale up. However, metric measurements are typically presented in huge tables that can be hard to interpret. This situation is worse when metric values are combined.

Visualization is a well-known technique to support the understanding of large applications [55]. CodeCrawler is a visualization engine [40] built on top of Moose that presents metrics visually as *polymetric views* [37]. A polymetric view, is a two-dimensional visualization of nodes (as entities) and edges (as relationships) that maps various metric values to attributes of the nodes and edges. For example, different metrics can be mapped to the size, position and color of a node, or to the thickness and color of the edge.

Polymetric views can be generated for different purposes: coarse-grained views to assess global system properties, fine-grained views to assess properties of individual software artifacts, and evolutionary views to assess properties over time.

Figure 4 shows a System Complexity View which is coarse grained view [39]. The figure shows the hierarchies of CodeCrawler itself. Each node represents a class, and each edge represents an inheritance relationship. The height of a node represents the number of methods, the width represents the number of attributes and the color represents the number of lines of code. Such views can help to direct a reverse engineers' attention to *Exceptional Entities* [11]. For example, tall, isolated, dark nodes have many methods, many lines of code, and few attributes, and they may be signs of procedural classes with long, algorithmic methods.

To understand a system, one needs to go beyond the overall structure and into the fine-grained details of its parts. An example of such a fine-grained polymetric view is the Class Blueprint [16]. The goal is to obtain an understanding of the inner structure of one class or several classes at once. Furthermore, it is useful for detecting patterns in the implementation of classes and class hierarchies. These patterns help one to answer questions regarding the internal structure of classes and class hierarchies and are also useful for the detection of design patterns [34].

Another high-level visualization is the Evolution Matrix [38]. Whereas the coarse-grained and fine-grained views focus on the structure of the system, the Evolution Matrix presents a view of the evolution of the system. It displays several versions of the classes of the system by displaying each class as a node in a matrix. Each row in the matrix represents the overall history (*i.e.,* the versions) of the class while each column represents a version of the system. This view reveals class evolution patterns like: growing classes, stable classes, *etc.*

Multiple metrics can also be effectively combined into *butterfly views* to provide high-level characterizations of software entities such as packages [18].

Though Figure 2 shows CodeCrawler as being one of the tools built on top of Moose, in fact CodeCrawler provides visualization services for numerous other tools that use Moose.

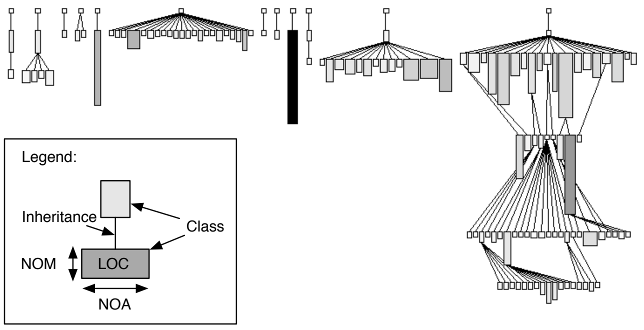


Figure 4: CodeCrawler displaying itself in a System-Complexity View.

3.2 Concept Analysis

Visualization of direct metrics can be a very effective tool for exploring a software model at various levels of granularity, but it does not, by itself, help one to identify implicit recurring patterns in the code. Coding idioms, design patterns, architectural constraints and contracts are all examples of such recurring patterns. They are *implicit* in the sense that no language feature or direct metric can be used to identify them. Furthermore, if one is not actively *looking* for a specific pattern, one might never discover it.

One way to uncover such recurring patterns is to apply *clustering* techniques. Formal Concept Analysis (FCA) is a well-known clustering technique that is especially well suited to this problem. FCA takes as input a collection of *entities*, each of which exhibits a certain set of *properties*, and produces as output a set of *concepts*, which are clusters of entities exhibiting the same properties. Furthermore, the concepts are organized into a lattice, such that concepts that are higher in the lattice exhibit more properties but fewer entities. The top and bottom of the lattice represent, respectively, the concept consisting of all entities but no common properties, and the concept with all properties, but no entities sharing all these properties.

FCA can be applied to the analysis of software systems by specifying which are the entities and properties of interest [1]. For example by choosing methods and instance variables of classes as the entities, and calling and accessing relationships as properties, FCA can identify various canonical types of classes as concepts, such as those whose methods systematically access all the state of the object, and those in which groups of methods access only part of the state [3]. FCA can also be applied at much coarser levels of granularity, for example, to identify implicit contracts within class hierarchies [4], and to detect recurring collaboration patterns amongst groups of classes [2].

Although, it was not the focus of the research, during the experimentation CodeCrawler was heavily used as a visualization back-end of the concepts. For example, Figure 5 shows a visualization of State Usage inside a Class Blueprint. The rightmost nodes represent the attributes in the class. The rest of the nodes represent the methods in the class. The methods marked with green (grey in the print version) are the methods that are related to both attributes.

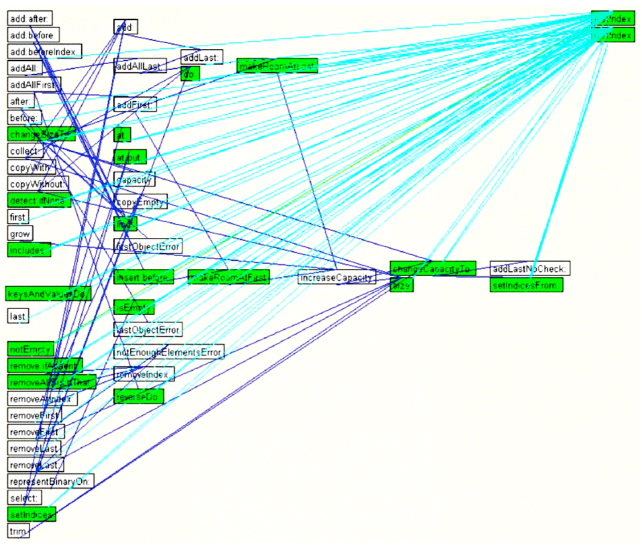


Figure 5: ConAn **using** CodeCrawler **to visualize how state is used in the methods of a class.**

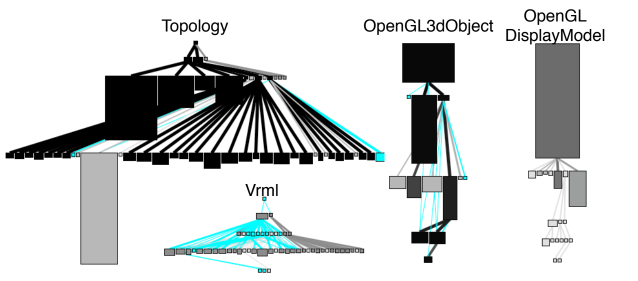


Figure 6: Van **using** CodeCrawler **to show how class hierarchies evolve.**

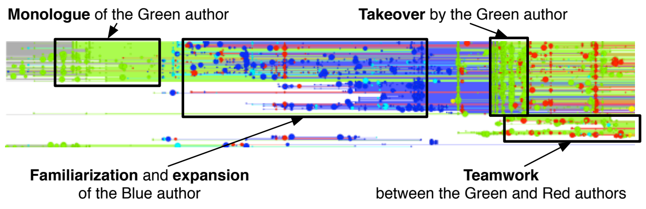


Figure 7: Chronia **visualizing how developers change the files.**

3.3 History Analysis

Reengineering efforts classically focus on the most recent version of a software system, since that is the snapshot that reflects most of the requirements and contains most of the features. On the other hand, it is the *history* of the system that reveals how it has evolved, and where chronic problems may lie. For example, analyzing the past can reveal dependency patterns based on co-change history [25] or can be used to predict change propagation [33].

A key principle behind the development of Moose is to make the meta-model explicit. Although a great deal of research interest for software evolution has been awakened in recent years, most of the research is entrenched in the software models imposed by current versioning systems. For example, CVS is a widely used versioning system and it offers the infrastructure to manipulate files and text. However, no semantics of the structure of the code can be directly accessed or manipulated through CVS.

To compensate for this shortcoming, we developed Hismo, a meta-model that models history as a first class entity [14]. A history is a collection of versions. Thus, in Hismo, a class history is modeled as a collection of class versions, a method history as a collection of method versions, and so on. Hismo is used by Van, a tool for analyzing version histories built on top of Moose.

The first applications of Hismo are historical measurements. One example is “Yesterday’s Weather” [26], a measure intended to indicate the need to reengineer certain parts of the system based on which parts have been undergoing the most change in the recent past. This reflects the retrospective empirical observation that the parts of the system that have changed in the near past are most likely to change in the near future.

In another application, we used the historical information to define a so-called time-based detection strategy [45]. A detection strategy denotes a measurement-based rule for detecting design flaws [42]. For example, God Class [51] is one “bad smell” that can be detected using such a detection strategy. The resulted classes that result from the detection strategies are candidates for refactoring. Yet, not all God Classes are equally bad. We used the historical information to define a time-based detection strategy to detect God Classes which were stable, or God Classes which were persistent. In our example, we used this information to classify the “badness” of the God Classes.

We also used CodeCrawler to characterize how class hierarchies evolve [30]. Figure 6 shows a polymetric view of Jun, an Open-Source 3D Multimedia Library. This polymetric view is similar to the System Complexity View, except that it displays evolution properties. The darker a node is, the older the class is, and the bigger a node is, the more the class was changed. Along the same line, the darker and thicker an edge is, the older the inheritance relationship is. Cyan (light grey) denotes nodes and edges that are no longer present in the last version of the system.

In another application we used ConAn to combine historical analysis with Formal Concept Analysis to detect concepts related to co-changing entities [27]. We used this technique to detect parallel inheritance or “shotgun surgery” [24] based on which classes consistently change in tandem during the same iteration.

Although Hismo was only used in the context of software reverse engineering, we showed that the concept of history does not depend on the type of data, and it can be applied to any structural meta-model [28].

Recently, we have developed Chronia, a new tool to analyze directly CVS. An application of this tool is the Ownership Map, a visualization describing the way developers change code [29]. Figure 7 shows the Ownership Map of a medium Java system: each line in the visualization represents the history of a file and the color of a pixel in the line is given by the author that owns the most lines in the file at the respective moment. From the visualization, we can detect several patterns of the behavior of the developers: teamwork, takeover, familiarization *etc.*

3.4 Dynamic Analysis

Most of the analyses performed with the help of MOOSE focus on modeling and manipulating the static information extracted from the source code of the systems under study. Source code alone, however, cannot tell you what run-time structures will be created, how frequently certain messages will be sent, or, in the presence of polymorphism, to instances of which classes variables will be dynamically bound.

A reverse engineer is not restricted to analyzing source code. The system under analysis can be instrumented, and execution traces can be generated for selected scenarios. Although such traces can easily generate massive amounts of data (*i.e.,* typically many megabytes for just short execution runs), it is possible to effectively manage this data by exploiting the static knowledge as well.

Perspectives are high-level views of a software system expressed, for example, in terms of components and connectors, or in terms of roles and collaborations. GAUDI represents our first effort to recover high-level views by correlating static and dynamic information [46, 47, 48]. In this approach, MOOSE is used to build up a static model of the software system under study. This model is then transformed into a set of Prolog facts. Next, the source code is instrumented and various scenarios are executed to generate execution traces. Once again, these traces are transformed into sets of Prolog facts that encode which events took place at run-time. Next, a series of Prolog-queries are iteratively posed to build the high-level view of the system. This information can then be used to graphically render, for example, the components-and-connectors view.

In another approach, we combined visualization, metrics and run-time information to offer a highly condensed view of the execution of applications. We developed DIVOOR, a metrics engine for dynamic information, and then used CODECRAWLER for the visualization [17]. The visualization consists of polymetric views of the static structure, where each node is annotated with dynamic information: number of instances created of a class, number of message passed to and from a class *etc.*

More recent work explores further the relationship between the dynamic information and the static information [31]. The approach is implemented in TRACESCRAPER and the aim is two-fold: characterize the structural entities with respect to features and characterize the features with respect to the structural entities. The approach is based on extending the meta-model with the trace as an explicit entity and on linking it with the structural entities (*e.g.,* classes, methods) that it passes through. The structural entities can be characterized according to their importance in the features. For example, a class can be characterized as infrastructural (when it is used in most of the features) or single feature (when it is used in only one feature). At the same time, the features can also be characterized as being related or not based on the code they use.

This model is further explored in combination with the HISMO meta-model to analyze the evolution of the role of the structural entities with respect to a given set of features [32]. History measurements are defined to detect structural entities that become more important within the system as they become parts of its infrastructure, or entities that lose importance as they are used by fewer features. Such information can be used, for example to detect the features that the development is currently focused on, or to detect refactorings or obsolete code.

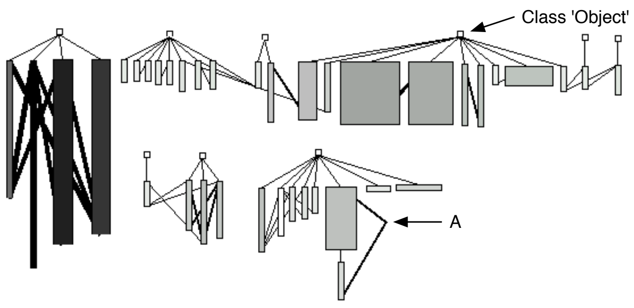


Figure 8: Internal and external duplication

3.5 Clone Detection

Duplicated code is one of the most common signs of code rot in legacy software systems. Contrary to initial expectations, however, duplicated code can be quite difficult to identify in large systems, since code is rarely copied outright, but is generally copied and modified in various ways. Formatting may be changed arbitrarily, and bits of code may be inserted or deleted at multiple locations. As a consequence, simple string-matching alone is unlikely to be very successful in identifying code clones.

DUPLOC is a tool that augments simple string-matching with noise elimination to remove the effects of reformatting, and post-filtering to identify only significant runs of duplication [21]. Surprisingly, the lightweight approach based on string-matching is highly effective, and yields high recall and acceptable precision compared with more sophisticated approaches based on, for example, comparison of syntax trees [19, 49].

DUPLOC is an example of a language-independent reverse-engineering tool that was not based on MOOSE, mainly because the string-matching approach did not require the level of detail offered by the MOOSE models of software.

Nevertheless, we were able to leverage the metrics visualization tools of MOOSE to gain additional insights into the nature of code duplication. In Figure 8 we see a System Model view of part of JBOSS– a polymetric view in which boxes represent classes of a hierarchy, the width of each box represents the degree of internal duplication (*i.e.,* within a class), the height represents the external duplication (*i.e.,* between different classes), and thick lines represent copied code [50]. (The small squares are classes outside of JBOSS.) Class A in the figure indicates code duplication between a subclass and its superclass.

3.6 Semantic Analysis

Trying to understand a software system by just analyzing its structure is a bit like trying to appreciate a painting with just a light meter. The structure of a software system tells us how parts of the code are related, but not what they mean. Important clues to the meaning of the system can be found in the source code, *i.e.,* in the names of identifiers and in the comments.

HAPAX implements Latent Semantic Indexing (LSI), an

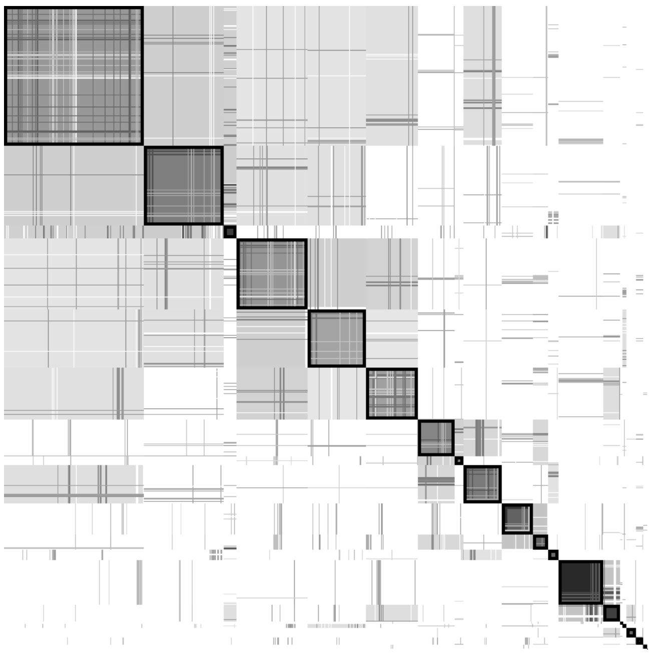


Figure 9: HAPAX showing the semantic similarity correlation matrix for the classes of JEdit. The same visualization engine is also used by CHRONIA.

information retrieval technique to analyze the space formed by documents and terms. As an application to software reengineering, HAPAX considers structural entities to be documents and the identifiers to be terms.

Searching is a straightforward application of LSI, so HAPAX is well-suited to searching for the most relevant entities for a given text query. A more ambitious application for reverse engineering is to cluster the entities based on their semantic similarity [36]. Figure 9 shows a correlation matrix visualization of all the classes of JEdit. The visualization engine used is shared with CHRONIA (Section 3.3). The matrix displays the classes both on the rows and on the columns. The grayness of a dot in the matrix is determined by the similarity between the corresponding classes. The dark borders delimit the detected clusters.

LSI is also used to determine the most relevant terms for the given clusters, thus the approach also indicates what the clusters are about. For example, the top-left clusters are characterized by the terms: "cell", "renderer", "pane", "scroller" and "frame". This indicates that clusters refer to the user interface.

4. LESSONS LEARNED

MOOSE appears to violate the principle that research prototypes are just that – prototypes that serve as a proof-of-concept to achieve a research result, but not more than that. Common wisdom states that one should invest no more into research prototypes than is necessary to publish a paper or get the next research grant.

Although the total effort invested in MOOSE over the years may be considerable, in fact the effort invested at any point in time was not more than would have been devoted to an *ad hoc* tool. The key difference is that MOOSE has been refined, extended and reengineered over the years rather than simply being thrown away. In the end, MOOSE is still relatively small, but it is far more useful as a research medium than its original FAMIX ancestor.

Make your meta-model explicit. A meta-model describes the way the system can be represented, that is, it provides bricks for reasoning. A rapid prototype built using existing tools is often a slave to the paradigms and models implicit in those tools. By making the meta-model explicit, we were able to develop tools that cooperate better, and so obtain benefits by combining results of multiple tools. Without this, each tool would have been a standalone prototype without any potential for supporting further experiments. By developing our own meta-model we were able to establish a minimal infrastructure for integrating the experimental tools that we built in the early days of the FAMOOS project.

Control your meta-model. One premise of MOOSE is not just to provide a static meta-model, but to provide for the infrastructure to extend the meta-model according to the needs of the analysis. Thus, our aim is to take an active stand towards the nature of data, and not to be governed by the meta-models provided by different tools available. For example, CVS is a widely used versioning system and it offers text-based information. Instead of directly analyzing the data provided by CVS, we define HISMO, our history meta-model to be able to express analyses at different levels of abstractions.

Leverage tools that combine techniques. At the start of FAMOOS, we were content to experiment by combining various off-the-shelf tools and prototypes to quickly obtain initial results. As it became clear that certain combinations of techniques were especially productive, the investment in home-grown tools became worthwhile. In particular, the combination of software metrics and visualization offered by CODECRAWLER could not be achieved by merely combining existing tools. The meta-modeling framework offered by MOOSE gave us the possibility to experiment with numerous reverse and reengineering techniques that would have been much more difficult without a modeling infrastructure in place.

Use of a dynamic programming environment. MOOSE and most of the tools built on top of it have been developed using VisualWorks Smalltalk [58]. We strongly believe that Smalltalk has contributed to the success of MOOSE in several important ways.

In Smalltalk, everything is an object, and all objects live in a persistent image. Code is incrementally compiled, so there is effectively no physical or temporal separation between "run-time" and "compile-time". As a consequence, all objects are always available, and it is possible to send any message to any object at any time. In particular, since querying is a central part of analysis, we could directly use the Smalltalk language as a query language, instead of having to develop a specialized, *ad hoc* query language.

The Smalltalk environment also offers a generic inspector which allows one to explore the contents of any given object. As a result, whenever our own GUI was not expressive enough we could always drop down and use the inspector. In this way we could quickly test out ideas without investing a great deal in coding, and thus evaluate whether it would be worthwhile to invest in a certain direction.

Focus on problems, not solutions. In the end, the research process is not about building tools, but about exploring ideas. In the context of reengineering research, however, one must build tools to explore ideas. Crafting a tool requires engineering expertise and effort, which consumes valuable research resources.

The art is to keep the research process *agile* in the sense that one concentrates on the research problems, while still keeping an eye out for the possible benefits of generalization.

As an example, CHRONIA and HAPAX are recently-built tools that share the same visualization engine. It is only now, however, that the shared interest has been identified, that it becomes worthwhile to invest in factoring out this visualization engine to make it more readily available to other efforts.

Exploit new opportunities. Research opportunities constantly arise, if one is prepared to recognize them.

As an example, our research into code duplication led us to explore language mechanisms to enable fine-grained factoring of common functionality into *traits* [20, 52, 53].

Similarly, the observation that software evolution is difficult to manage in programming languages that require global consistency at all times led to the development of *classboxes*, a module system in which various parts of a software system may see different, inconsistent extensions of shared modules [5, 6].

A platform for research is a fine thing, but one should also be prepared at any time to strike out in new directions.

5. CONCLUSIONS

Over the space of several years, MOOSE has evolved from a simple meta-model for integrating various experimental reverse- and re-engineering tools, to a platform offering various common services, such as software metrics evaluation and visualization, generic GUI services for querying, browsing and grouping, and support for multiple, concurrent models. Although MOOSE has evolved considerably, it has remained relatively small, thanks to reengineering efforts being applied to MOOSE itself, and care taken to ensure that only the essential common services find their way to the core of MOOSE.

Whereas most research prototypes typical have a very short half-life, MOOSE has survived mainly because it serves to support various other research activities, particularly those which can benefit from combinations of techniques, such as metrics and visualization, or static and dynamic analysis.

MOOSE continues to evolve in new directions. The 2D visualization framework of CODECRAWLER is being extended to support 3D visualization and animation of dynamic traces. We are developing a bridge to popular version control systems so that version histories can be directly explored from the perspective of the MOOSE meta-model. We have been extending the metrics framework to explore new kinds of metrics that better capture notions of coupling and cohesion for component frameworks, thus taking usage scenarios into account. We are exploring the use of Formal Concept Analysis to identify refactoring opportunities. We have also experimented with other clustering techniques to automatically categorize classes according to their internal calling patterns [34]. In each case, MOOSE has accelerated the research activity by allowing us to leverage our investment in meta-modeling, metrics, visualization, and other complementary techniques.

Finally, the research activities surrounding MOOSE have indirectly led us to explore new directions in programming languages design, in the search for language mechanisms and features that ease the task of developing systems that can gracefully evolve over time [43].

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

6. REFERENCES

[1] G. Arévalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, Jan. 2005.

[2] G. Arévalo, F. Buchli, and O. Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*, pages 122–131. IEEE Computer Society Press, Nov. 2004.

[3] G. Arévalo, S. Ducasse, and O. Nierstrasz. X-Ray views: Understanding the unternals of classes. In *Proceedings of ASE '03 (18th Conference on Automated Software Engineering)*, pages 267–270. IEEE Computer Society Press, Oct. 2003. Short paper.

[4] G. Arévalo, S. Ducasse, and O. Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *Proceedings of CSMR '05 (9th European Conference on Software Maintenance and Reengineering)*, pages 62–71. IEEE Computer Society Press, Mar. 2005.

[5] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, New York, NY, USA, 2005. ACM Press. To appear.

[6] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, May 2005.

[7] W. R. Bischofberger. Sniff: A pragmatic approach to a c++ programming environment. In *C++ Conference*, pages 67–82, 1992.

[8] C. T. Committee. CDIF framework for modeling and extensibility. Technical Report EIA/IS-107, Electronic Industries Association, Jan. 1994. See http://www.cdif.org/.

[9] S. Demeyer and S. Ducasse. Metrics, do they really help? In J. Malenfant, editor, *Proceedings LMO '99 (Languages et Modèles à Objets)*, pages 69–82. HERMES Science Publications, Paris, 1999.

[10] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA '2000 (International Conference on*

Object-Oriented Programming Systems, Languages and Applications), pages 166–178, 2000.

[11] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2002.

[12] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[13] S. Ducasse and S. Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook.* University of Bern, Oct. 1999.

[14] S. Ducasse, T. Gîrba, and J.-M. Favre. Modeling software evolution by treating history as a first class entity. In *Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 71–82, 2004.

[15] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a Collaborative and Extensible Reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55 – 71. Franco Angeli, 2005.

[16] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[17] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of CSMR 2004 (Conference on Software Maintenance and Reengineering)*, pages 309 – 318, 2004.

[18] S. Ducasse, M. Lanza, and L. Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05).* IEEE Computer Society, 2005. To appear.

[19] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance: Research and Practice*, 2005. To appear.

[20] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, 2005. To appear.

[21] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings ICSM '99 (International Conference on Software Maintenance)*, pages 109–118. IEEE Computer Society, Sept. 1999.

[22] S. Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *International Journal on Software Maintenance: Research and Practice*, 15(5):345–373, Oct. 2003.

[23] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[24] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[25] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[26] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 40–49. IEEE Computer Society Press, 2004.

[27] T. Gîrba, S. Ducasse, R. Marinescu, and D. Raţiu. Identifying entities that change together. In *Ninth IEEE Workshop on Empirical Studies of Software Maintenance*, 2004.

[28] T. Gîrba, J.-M. Favre, and S. Ducasse. Using meta-model transformation to model software evolution, 2004.

[29] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How Developers Drive Software Evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE).* IEEE Computer Society Press, 2005. to appear.

[30] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of European Conference on Software Maintenance (CSMR 2005)*, 2005.

[31] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering.* IEEE Computer Society Press, 2005.

[32] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, Sept. 2005. to appear.

[33] A. Hassan and R. Holt. Predicting change propagation in software systems. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293. IEEE Computer Society Press, Sept. 2004.

[34] M.-P. Horvath. Automatic recognition of class blueprint patterns. Diploma thesis, University of Bern, Oct. 2004.

[35] R. Kazman, S. Woods, and S. Carriére. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of WCRE '98*, pages 154–163. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.

[36] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering, 2005. submitted.

[37] M. Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization.* PhD thesis, University of Berne, May 2003.

[38] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[39] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[40] M. Lanza and S. Ducasse. Codecrawler - an extensible

and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74 – 94. Franco Angeli, 2005.

[41] M. M. Lehman and L. Belady. *Program Evolution – Processes of Software Change*. London Academic Press, 1985.

[42] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 350–359. IEEE Computer Society Press, 2004.

[43] O. Nierstrasz, A. Bergel, M. Denker, S. Ducasse, M. Gälli, and R. Wuyt. On the revival of dynamic languages. In T. Gschwind and U. Aßmann, editors, *Proceedings of Software Composition 2005*. LNCS, 2005. Invited paper. To appear.

[44] D. L. Parnas. Software Aging. In *Proceedings of ICSE '94 (International Conference on Software Engineering)*, pages 279–287. IEEE Computer Society / ACM Press, 1994.

[45] D. Raţiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *Proceedings of CSMR 2004 (European Conference on Software Maintenance and Reengineering)*, pages 223–232, 2004.

[46] T. Richner. *Recovering Behavioral Design Views: a Query-Based Approach*. PhD thesis, University of Berne, May 2002.

[47] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings ICSM '99 (International Conference on Software Maintenance)*, pages 13–22. IEEE Computer Society Press, Sept. 1999.

[48] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of ICSM '2002 (International Conference on Software Maintenance)*, Oct. 2002.

[49] M. Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Berne, June 2005.

[50] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *Proceedings of WCRE 2004 (11th Working Conference on Reverse Engineering)*. IEEE Computer Society Press, Nov. 2004.

[51] A. J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[52] N. Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, Feb. 2005.

[53] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[54] A. Schlapbach. Generic XMI support for the MOOSE reengineering environment. Informatikprojekt, University of Bern, June 2001.

[55] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[56] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Dec. 2001.

[57] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A Meta-model for Language-Independent Refactoring. In *Proceedings of ISPSE '00 (International Conference on Software Evolution)*, pages 157–167. IEEE Computer Society Press, 2000.

[58] Cincom Smalltalk, Sept. 2003. http://www.cincom.com/scripts/smalltalk.dll/.