Polymetric Views – A Lightweight Visual Approach to Reverse Engineering

Michele Lanza, Stéphane Ducasse

Preprint of the Transactions on Software Engineering Journal

Abstract

Reverse engineering software systems has become a major concern in software industry because of their sheer size and complexity. This problem needs to be tackled, since the systems in question are of considerable worth to their owners and maintainers. In this article we present the concept of a *polymetric view*, a lightweight software visualization technique enriched with software metrics information. Polymetric views help to understand the structure and detect problems of a software system in the initial phases of a reverse engineering process. We discuss the benefits and limits of several predefined polymetric views we have implemented in our tool CodeCrawler. Moreover, based on clusters of different polymetric views we have developed a methodology which supports and guides a software engineer in the first phases of a reverse engineering of a large software system. We have refined this methodology by repeatedly applying it on industrial systems, and illustrate it by applying a selection of polymetric views to a case study.

Keywords

Reverse Engineering, Object-Oriented Programming, Software Visualization, Software Metrics

I. Introduction

Reverse engineering large software systems is difficult due to their sheer size and complexity. However, it is a prerequisite for their maintenance, reengineering, and evolution. Chikofsky [1] defines reverse engineering as "the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction". Maintaining and evolving existing software systems is difficult for several reasons, among which are the accelerating turnover of developers, the increasing size and complexity of software systems, and the constantly changing requirements of software systems. These *legacy systems* are large, mature, and complex software systems, which are the result of a long-term investment effort of a company and must therefore be maintained and evolved, because new requirements must be fulfilled [2] [3], and because the company's investment has to pay back. Parnas [4] assessed that most legacy systems suffer from several typical problems, including original developers may be no longer available, outdated development methods and/or programming languages, and outdated, incomplete or missing documentation.

The maintenance and evolution of such systems is therefore, apart from being technically difficult, prohibitively expensive: Sommerville [5] and Davis [6] estimate that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system. Rewriting these systems from scratch is also problematic, because this would take vast amounts of time, money, and human resources.

Since legacy systems tend to be large – hundreds of thousands of lines of poorly documented code are no exception – there is a definite need for effective approaches which help in program understanding and problem detection. We focus on object-oriented legacy systems, mainly because most current systems are written using this paradigm, and because it is not *age* that turns a piece of software into a legacy system, but the *rate* at which it has been developed and adapted [7]. Moreover, since the object-oriented paradigm does not support a sequential reading order (*i.e.*, the domain model is distributed across classes, hierarchies, and subsystems), the reverse engineer needs to know where to look into the system to understand its structure.

Michele Lanza and Stéphane Ducasse are both members of the Software Composition Group at the University of Bern in Switzerland. Michele Lanza is the corresponding author. Contact Information Michele Lanza: Michele Lanza, Institut für Informatik und angewandte Mathematik, Neubrückstrasse 10, 3012 Bern, Switzerland. E-mail: lanza@iam.unibe.ch. Tel.: +41 31 631 4868. Fax: +41 31 631 3355. Contact Information Stéphane Ducasse: Stéphane Ducasse, Institut für Informatik und angewandte Mathematik, Neubrückstrasse 10, 3012 Bern, Switzerland. E-mail: ducasse@iam.unibe.ch. Tel.: +41 31 631 4903. Fax: +41 31 631 3355.

We are targeting the first phase (*e.g.*, the first week) of a reverse engineering process, because in this phase a reverse engineer has to form an initial mental picture of the system [8]. Our approach helps the reverse engineer get a mental picture by viewing the system by means of polymetric views, lightweight software visualizations enriched with software metrics.

We use *software visualization* in this context because visual displays allow the human brain to study multiple aspects of complex problems – like reverse engineering – in parallel [9]. Ware states that "Visualization provides an ability to comprehend huge amounts of data" [10]. However, software visualizations are often too simplistic and lack visual cues for the viewer to correctly interpret them [11]. In other cases the obtained visualizations are still too complex to be of any real value to the viewer.

We use *software metrics* because they can be used to assess the quality and complexity of a system and because they are known to scale up well. Furthermore, metrics are a good means to control the quality and the state of a software system during the development process [12]. However, metrics often come in huge tables that are hard to interpret, and this is even more difficult when metrics are combined to generate yet other metrics.

We propose a lightweight approach based on *the combination of software visualization and software metrics*, by enriching simple visualizations with metrics information. We refer to these lightweight combinations as *polymetric views*. Depending on the applied polymetric view, the viewer can visually (*e.g.*, by looking and interacting with the visualization) extract different kinds of information about the visualized system, *i.e.*, information about the structure of hierarchies, about the size of classes and methods, about the use of attributes, etc. The viewer can then verify his findings by inspecting the corresponding source code fragments (according to the program cognition model vocabulary proposed by Littman *et al.* [13] we support an approach of understanding that is *opportunistic* in the sense that it is not based on a *systematic* line-by-line understanding but as needed). Note that opportunistic code reading is also useful in a forward engineering context, *e.g.*, in such a context our approach helps in code browsing, but this goes beyond the scope of this paper and is part of our current research. In this article we describe several polymetric views in detail, and point out the idea, the strengths, and the weaknesses of each view.

To guide a reverse engineer in the beginning phases (*e.g.*, the first one or two weeks, depending on the size of the system) of a reverse engineering process we have developed a methodology based on the polymetric views, which we have extended and refined by applying it repeatedly on industrial case studies.

II. Object-Oriented Reverse Engineering

Chikofsky states that *"The primary purpose of **reverse engineering** a software system is to increase the overall comprehensibility of the system for both maintenance and new development"* [1]. Therefore, before starting a reverse engineering process, it is essential to decide which primary goals to pursuit and which ones are only of secondary importance. In the context of object-oriented legacy systems we settled on the following goals for getting a first impression and a mental model of a system:

- Assess the overall quality of the system and gain an overview of the system in terms of size, complexity and structure.
- Locate and understand the most important classes and inheritance hierarchies, *i.e.*, find the classes and hierarchies that represent a core part of the system's domain and understand their structure in terms of implementation and purpose in terms of functionality.
- Identify exceptional classes in terms of size and/or complexity compared to all classes in the subject system. These may be candidates for a further inspection or for the application of refactorings.
- Identify the possible presence of design patterns or occasions where design patterns could be introduced to ameliorate the system's structure.

The result of a reverse engineering process is therefore not only a list of problematic classes or subsystems, even if the identification of possible design defects is a valuable piece of information. Indeed, we are looking for the bad use as well as the good use of object-oriented design, *e.g.*, obtaining information about how a system has been implemented is important, independently from the quality of its implementation. Moreover,

in a reengineering context the fact that a class may have a design problem does not necessarily imply that the class should be modified or completely redesigned, as this would cost time and money. Indeed, if a badly designed class (*e.g.*, too many methods, inconsistent use of accessors, dead code, etc.) or subsystem accomplishes the work it has been assigned to, without having a negative impact on the overall working of the system, there is no point in changing it. However, being aware of such information is still valuable for getting a better mental model of the system.

We developed our approach in the context of the European Esprit project FAMOOS, whose main results have been summarized in a reengineering handbook [14] which was the basis for a book on object-oriented reengineering patterns [7]. The goal of the project was to reengineer several large industrial object-oriented software systems. The industrial setting of the FAMOOS project introduced the following constraints:

Simplicity. In software industry, reengineers face many problems, *i.e.*, short time constraints, little tool support, and limited manpower. It is for this reason that we wanted our results to be reproduceable by software engineers at their workplace, without having to rely on complex or expensive tools. Moreover, by choosing a lightweight approach, we were able to get results quickly, in order to evaluate whether certain ideas were viable or not.

Scalability. We wanted to make sure that our approach could handle the size of industrial systems, which can be of several millions of lines of code. The scalability is on one hand guaranteed by the use of software metrics, since metrics can be computed independently from the size of the system. On the other hand our approach allowed us to generate, test and accept/reject new ideas in short iteration cycles. After starting the development of our tools we constantly tested them in industrial settings to see whether they were actually viable and could indeed scale up.

Language Independence. In order to handle software systems written in different languages, we developed FAMIX [15], a language independent metamodel. Our implementation in Smalltalk of the FAMIX metamodel, called the Moose Reengineering Environment [16], is presented in detail in Section VI.

III. The Approach

A. The Principle

Our visualization tool CodeCrawler uses two-dimensional displays to visualize object-oriented software [17]. The nodes represent software entities or abstractions of them, while the edges represent relationships between those entities. This is a widely used practice in information visualization and software visualization tools. Ware claims that "other possible graphical notations for showing connectivity would be far less effective" [10]. We enrich this basic visualization method by rendering up to 5 metric measurements on a single node simultaneously, as we see in Figure 1.[A]. A list of some of the metrics we can enrich our visualizations with is given in Table I.

Node Size. The width and the height of a node can each render one metric measurement. The bigger these measurements are, the bigger the node is in one or both of the dimensions.

Node Color. The color interval between white and black can be used to render another metric measurement. The convention is that the higher the metric value is, the darker the node is. Thus light gray represents a smaller metric measurement than dark gray.

Node Position. The X and Y coordinates of the position of the node can also reflect two metrics measurements. This requires the presence of an absolute origin within a fixed coordinate system. Not all layouts can exploit position metrics, as some of them implicitly dictate the position of the nodes (*e.g.*, a tree layout).

In measurement theory, this procedure of rendering metrics on two-dimensional nodes is called *measurement mapping*, and fulfills the representation condition, which asserts that "a measurement mapping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations" [12]. In other words, if a number a is bigger than a number b, the graphical representation of a and b must preserve this fact.

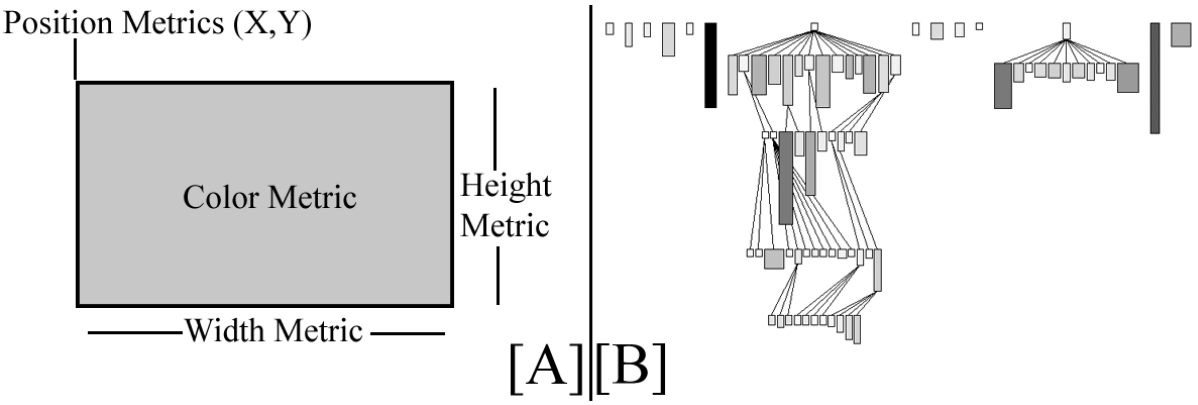

Fig. 1. [A]: Up to 5 metrics can be visualized on one node. The list of possible metrics is given in Table I. [B]: The SYSTEM COMPLEXITY view. This visualization of classes uses a tree layout. The edges represent inheritance relationships. The metrics we use to enrich the view are NOA (the number of attributes of a class) for the width and NOM (the number of methods of a class) for the height. The color shade represents WLOC (the number of lines of code of a class).

B. Software Metrics

We make extensive use of object-oriented software metrics. In the wide array of possible metrics [18] [19] [12] we selected *design metrics, i.e.*, metrics that can be extracted from the source code entities themselves. These metrics are usually used to assess the size and in some cases the quality and complexity of software. The metrics we use are termed *direct measurement* metrics because their computation involves no other attributes or entities [12]. We don't make use of *indirect measurement* where metrics are combined to generate new ones, because the *measurement mapping* presented in the previous section works best with direct measurements. Examples of indirect metrics include *programmer productivity*, *defect detection density* or *module defect density*, as well as more code-oriented ones like *CBO* and *RFC*, presented in [20]. We chose to use metrics that can be extracted from source code entities, and which have a simple and clear definition. As such we don't use *composite metrics*, which raise the issue of dimensional consistency [19].

In Table I (see appendix) we list all metrics mentioned in this article.

C. The Actual Visualization: A Polymetric View

An actual visualization of software in CodeCrawler depends on three ingredients: *a layout*, *a set of metrics*, and *a set of entities*.

1. **A layout.** A layout takes into account the choice of the displayed entities and their relationships and issues like whether the complete display should fit on the screen, whether space should be minimized, whether nodes should be sorted, etc. Some layouts make sense for all purposes, while others are better suited for special cases (*e.g.*, a tree layout is better suited for the display of an inheritance hierarchy than a circle layout).
As part of our lightweight approach, we chose to implement only simple layouts, although more advanced and powerful layouting techniques [21] are also interesting in this context. We use the following layouts, described in detail in the appendix of this article: tree, scatterplot, checker, and stapled.
2. **The metrics.** We incorporate up to 5 metrics selected from Table I into a view, as we have seen in Section III-A. The choice of the metrics heavily influences the resulting visualization, as well as its interpretation.
3. **The entities.** Certain views are better suited for small parts of the system, while others can handle a complete large system. The reverse engineer must choose which parts or entities of the subject system he wants to visualize. These choices are part of the methodology discussed in depth in Section IV.

Example. Figure 1.[B] shows a tree layout of nodes enriched with metrics information. The nodes represent classes, while the edges represent the inheritance relationships between them. The size of the nodes reflects the number of attributes (width) and the number of methods (height) of the classes, while the color tone represents the number of lines of code of the classes. The position of the nodes does not reflect metric

measurements in this case, as the nodes' position is implicitly given by the tree layout. In the figure we see that the visualized system is composed of two large inheritance hierarchies (one of which is quite deep) and some standalone classes.

The combination of the tree layout, the metrics mentioned above and the selection of classes as nodes and inheritance relationships as edges yields a polymetric view that we call SYSTEM COMPLEXITY view, whose properties are described in more detail in Section V-A.

Interpretation of a View. The polymetric views are revealers of *symptoms* which reside at a purely visual level, *i.e.*, they can be small dark nodes or large nodes, or even nodes at a certain position. These symptoms provide information about the subject system and support the decision process of which next view should be applied on which part of the system by the reverse engineer. Not all views lead to other views, but they may also result in specific reengineering actions that represent the next logical step after the detection of defects. For example detecting a "god class", defined by Riel [22] as a class that has grown over the years ending up with too many responsibilities, may lead to a necessary splitting of the class. For example, long methods can be analyzed to see if they contain duplicated code or if they can be split up into smaller, more reusable methods [23], etc. The interpretation of the views is based on heuristics which mainly come from the experience of the authors, but have also been documented by Riel [22], Fowler [24], Beck [25], and others.

Useful Views. Note that since our approach allows one to combine a subset of the metrics presented in Table I with a layout algorithm on any kind of software artifacts, there is a great number of possible views. However, many of those are similar to others (*e.g.*, by exchanging the width and height metrics) and many others do not help the reverse engineering process. We identified a number of *useful* views, *i.e.*, polymetric views that are useful for the reverse engineering process, and present a subset of them in this article.

IV. A LIGHTWEIGHT VISUAL REVERSE ENGINEERING METHODOLOGY

We have already presented a first version of our methodology [26], and now present an extended and elaborated version. Ideally such a methodology defines which views to apply, what the paths are between the different views, and on what parts of the system the next view should be applied. There are many challenges to the elaboration of such a methodology:

- There is no unique or ideal path through the views, since different views can be applied at the same stage depending on the current context.
- The decision to use a certain view most of the time depends on some interactions with the currently displayed view. Furthermore, the views can be applied to different entities implying some navigation facilities between the different views.
- A view displays a system from a certain perspective that emphasizes a particular aspect of the system. However, the view has to be analyzed and the code understood to determine if the details revealed by the view are interesting for further investigation.
- The views are heavily customizable. For instance exchanging two metrics is easy, yet it may yield completely different views. The reverse engineer must steer this process in order to apply and customize the useful views.

By loosely grouping the polymetric views into clusters and by indicating alternative views and navigation possibilities between the views we think that these challenges can be overcome. Furthermore, although the views are customizable, our tool offers a set of predefined views, some of which are presented in this paper, that can be applied directly and without requiring the user to define them himself, unless he wishes to do so. We identified the following clusters:

First Contact. The first thing to do with a subject system is to gain a first overview. We would like to know how big and complex the system is and in which way it is structured. The views in this cluster provide answers to the following questions: How big is the system and how is it composed: only of standalone classes, or of some (maybe large / deep) inheritance hierarchies. Is the system composed of many small classes or are there some really big ones? Where in the system do these large classes reside? This cluster contains the views SYSTEM HOTSPOTS and SYSTEM COMPLEXITY.

Inheritance Assessment. Inheritance is a key aspect of object-oriented programming languages, and thus represents an important perspective from which to understand applications. Inheritance can be used in different ways, for example as pure addition of functionality in the subclasses or as an extension of the functionality provided by the superclasses. The views in this cluster help in the analysis of inheritance and provide answers to the following questions: How are inheritance hierarchies structured and how do they make use of inheritance? Are subclasses merely adding new functionality of redefining the functionality defined in the superclasses? This cluster contains the views INHERITANCE CLASSIFICATION and INHERITANCE CARRIER.

Candidate Detection. One of the primary goals of a reverse engineer is to detect candidates which may be either cases where further investigation is necessary or where code refactorings are needed. The views in this cluster help in this problem detection process and provide answers to the following questions: Where are the large (small) classes or methods? Are there methods which contain dead code or attributes which are never used? This cluster contains the views DATA STORAGE CLASS DETECTION, METHOD STRUCTURE CORRELATION and DIRECT ATTRIBUTE ACCESS.

Class Internal. Understanding classes is a key activity in object-oriented programming, since classes represent the primary abstractions from which applications are built. The main problem of this task is to quickly grasp the purpose of a class and its inner structure. We have performed extensive research on this subject [27], but to keep this paper within a certain scope, we do not present the views contained in this cluster, especially the CLASS BLUEPRINT view. However, note that several of the views belonging to the other clusters can easily be applied on single classes as well, such as those from the candidate detection cluster.

V. A REVERSE ENGINEERING SCENARIO

Reporting about a case study is quite difficult without sacrificing the exploratory nature of our approach. Indeed, the idea is that different views provide different yet complementary perspectives on the software. Consequently, a concrete reverse engineering strategy should be to apply the views in some specific order, although the exact order would vary depending on the kind of system at hand and the kind of questions driving the reverse engineering effort. Therefore, readers should read this case study report as one possible use case, keeping in mind that reverse engineers must always customize their approach to a particular reverse engineering project.

Some Facts about the Case Study. The system we report on is called Duploc (version 2.16a), which is a tool for the detection of duplicated code [28]. We have already done a preliminary case study on an older version from 1999 of Duploc [29], and are curious to see how Duploc has evolved in the meantime. Duploc has become a mature application, consisting of more than 300 classes. Duploc detects code duplication by means of a visualization of each line as a dot in a two-dimensional matrix.

A. Reverse Engineering a System

Reverse engineering a system is a non-linear procedure and is difficult to present as a sequential text. For reasons of simplicity we discuss the views of each of the clusters, show their application on the case study and put them into relation according to our methodology presented in Section IV, as well as depending on the situations encountered during the reverse engineering of Duploc.

SYSTEM HOTSPOTS VIEW

Description: **Layout:** Checker. **Target:** Classes. **Scope:** Full system. **Metrics:** Width: NOA. Height: NOM. Color: WLOC. **Sort:** Width. **Example:** Figure 2.[A].

This simple view helps to identify large and small classes and scales up to very large systems. It relates the number of methods with the number of attributes of a class. The nodes are sorted according to the former, which makes the identification of outliers easy.

Symptoms: (1) Large nodes represent voluminous classes that may be further investigated. (2) Tall, and nar-

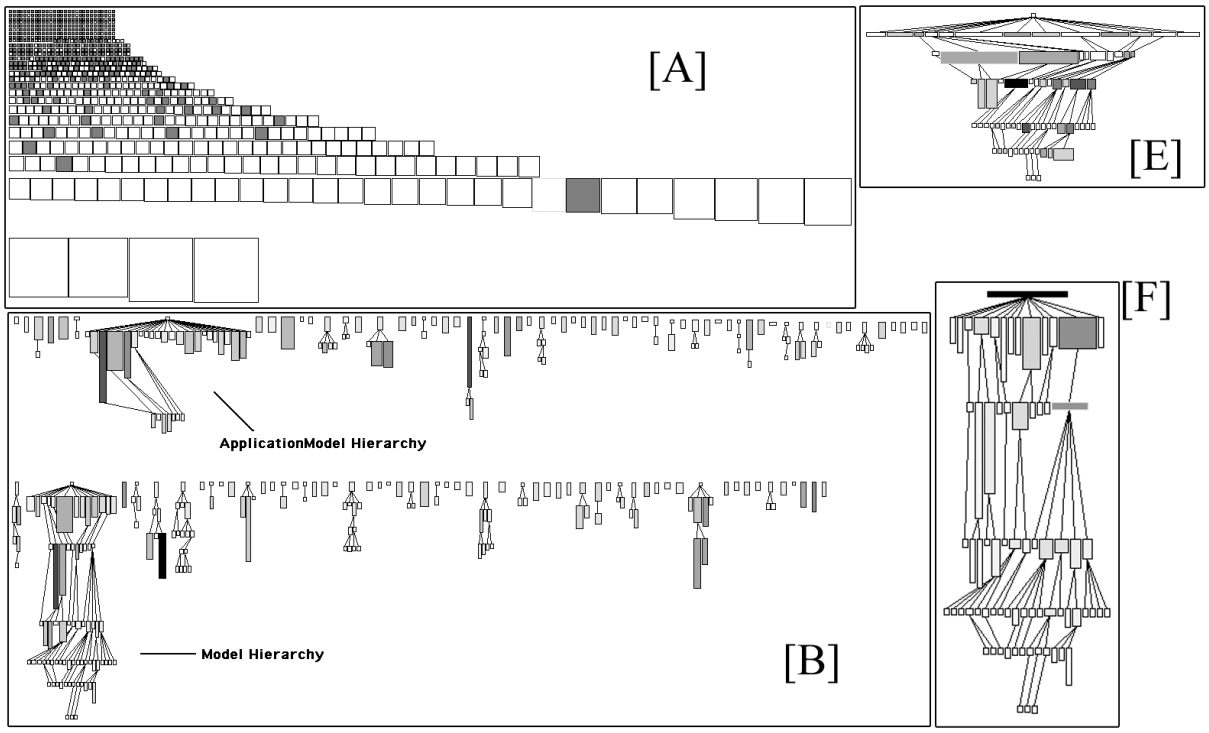

Fig. 2. [A]: A SYSTEM HOTSPOTS view (Variation 1 and 2) of Duploc. The nodes represent all the classes, while the size of the nodes represent the number of methods they define. They grey nodes represent metaclasses. [B]: A SYSTEM COMPLEXITY view on Duploc. The nodes represent the classes, while the edges represent inheritance relationships. As metrics we use the number of attributes (NOA) for the width, the number of methods (NOM) for the height and the number of lines of code (WLOC) for the color. [E]: An INHERITANCE CLASSIFICATION view of the *Model* hierarchy in Duploc. The width and height of the class nodes represents the number of added methods and the number of overridden methods, while the color represents the number of extended methods. [F]: An INHERITANCE CARRIER view of one hierarchy in Duploc. The width and the color of the class nodes represents the number of descendants, while the height represents the number of methods.

row nodes represent classes which define many methods and few or no attributes. (3) Wide nodes are classes with many attributes. When such nodes show a 1:2 width-height ratio it may represent a class whose main purpose is to be a data structure implementing mostly accessor methods. Further evidence can be gained from the color, which reflects the number of lines of code of a class. Should a tall class have a light color it means that the class contains mostly short methods.

Variations: (1) If we use the lines of code (WLOC) or the number of methods (NOM), as we see in Figure 2.[A] for rendering both the width and height of the nodes, we obtain a slightly different view which helps to assess the whole system in terms of raw measure: are there any big classes and how big are they actually? (2) In the case of Smalltalk classes, we can color metaclasses differently and check how they distribute themselves across the display. Should there now be large, colored nodes at the bottom of the display, it may be a sign that these metaclasses have too many responsibilities or that they function facades or as bridges to other classes [30].

Scenario: In Figure 2.[A] we see all the Duploc classes. The classes in the bottom row contain more than 100 methods and should be further investigated. They are *DuplocPresentationModelController (107 methods), RawMatrix (107), DuplocSmalltalkRepository (116) and DuplocApplication (117 methods).* We have colored the nodes representing metaclasses with grey. Note the bottom-most grey node which is the metaclass *DuplocGlobals* with 59 methods. This class, as suggested as well by the name, is a holder for global values. However, instead of using the metaclass, one suggestion to the developer is to apply the singleton design pattern instead [30].

SYSTEM COMPLEXITY VIEW

Description: **Layout:** Tree. **Target:** Classes. **Scope:** Full system. **Metrics:** Width: NOA. Height: NOM. Color: WLOC. **Sort:** -. **Example:** Figure 2.[B].

This view is based on the inheritance hierarchies of a subject system and gives clues on its complexity and structure. For very large systems it is advisable to apply this view first on subsystems, as it takes quite a lot of screen space. The goal of this view is to classify inheritance hierarchies in terms of the functionality they represent in a subject system. If we want to understand the inner working at a technical level of inheritance hierarchies we apply the views of the inheritance assessment cluster.

Symptoms: (1) Tall, and narrow nodes represent classes with few attributes and many methods. When such nodes appear within a hierarchy, applying the INHERITANCE CLASSIFICATION view or the INHERITANCE CARRIER view helps to qualify the semantics of the inheritance relationships in which the classes are involved. (2) Deep or large hierarchies are definitively subsystems on which the views of the inheritance assessment cluster help to refine understanding. (3) Large, standalone nodes represent classes with many attributes and methods without subclasses. It may be worth to have a look at the internal structure of the class to learn if the class is well structured or if it could be decomposed or reorganized. (4) Flat, light nodes with a width:height ration of 1:2 often represent data storage classes that define several attributes and for each attribute implement two accessor methods. The light color often denotes that a class has very short methods as is the case for accessors.

Scenario: Before showing this view we perform a manual preprocessing which consists in removing the class *Object*, which is the root class of the Smalltalk language, *e.g.*, every class inherits from it. We do this in order to focus on the use of inheritance within Duploc: Since many classes inherit directly from *Object* this view would be distorted if we included it in our view. We see the resulting SYSTEM COMPLEXITY view in Figure 2.[B]. We can see now that Duploc is in fact mainly composed of classes which are not organized in inheritance hierarchies. Indeed, there are some very large classes which do not have subclasses. The largest inheritance hierarchies are five and six levels deep. Noteworthy hierarchies seem to be the ones with the following root classes: *AbstractPresentationModelControllerState, AbstractPresentationModelViewState, DuplocSourceLocation*. The first one, with the root class *AbstractPresentationModelControllerState* with 31 descendants, seems to be the application of the *state* design pattern [30] for the controller part of an MVC pattern. Such a complex hierarchy within Duploc is necessary, since Duploc does not make any use of advanced graphical frameworks, but uses the basic standard VisualWorks GUI framework. Following this track of investigation we look for the other signs of the MVC pattern and find a hierarchy with *AbstractPresentationModelViewState* as root class with 12 descendants, which seems to constitute the view part of the MVC pattern.

INHERITANCE CLASSIFICATION VIEW

Description: **Layout:** Tree. **Target:** Classes. **Scope:** Subsystem. **Metrics:** Width: NMA. Height: NMO. Color: NME. **Sort:** -. **Example:** Figure 2.[E].

This view qualifies the inheritance relationships by displaying the amount of added methods relative to the number of overridden or extended methods. By extended methods we mean methods which contain a super call to a method with the same signature defined in one of the superclasses.

Symptoms: (1) Flat, light nodes represent classes where a lot of methods have been added but where few methods have been overridden or extended. In this case the semantic of the inheritance relationship is an addition of functionality by the subclasses. (2) Tall, possibly darker nodes represent classes where a lot of methods have been overridden and/or extended. They may represent classes that have specialized hook methods [30]. If the nodes are dark, it means that many methods have been extended, which hints at a higher degree of reuse of functionality.

Scenario: We have selected only one hierarchy, the one indicated as the *Model* hierarchy in Figure 2.[B], to demonstrate the application of this view. We see in Figure 2.[E] that the *Model* hierarchy is mainly composed of flat, lightly colored nodes: these classes mainly add functionality (denoted by their width) without really

overriding or extending functionality defined in the superclasses. We also see there are some exceptions: the subclasses of the two widest class nodes (*RawMatrix* and *AbstractRawSubMatrix*) with 96 and 72 added methods define several methods which are then overridden or extended by their subclasses. For example the two subclasses (*SymmetricRawMatrix* and *AsymmetricRawMatrix*) of *RawMatrix* heavily override functionality, as is indicated by their tall, narrow shape: both override 33 methods and add only 4, respectively 9, methods.

INHERITANCE CARRIER VIEW

Description: **Layout:** Tree. **Target:** Classes. **Scope:** Subsystem. **Metrics:** Width: WNOC. Height: NOM. Color: WNOC. **Sort:** -. **Example:** Figure 2.[F].

This view helps to detect classes with a certain impact on their subclasses in terms of functionality, *i.e.*, it helps to see which classes transmit the most functionality to their subclasses.

Symptoms: (1) Tall, dark nodes represent classes that define a lot of behavior and have many descendants. Therefore these classes have a certain importance for the (sub)system in question. (2) Flat, light nodes represent classes with little behavior and few descendants. (3) Flat, dark nodes represent classes with little behavior and many descendants. They can be the ideal place to factor out code from to the subclasses.

Scenario: Figure 2.[F] shows this view for the *Model* hierarchy. It shows that the classes which are carrying the weight of the implementation in this hierarchy are first of all the classes *AbstractPresentationModelControllerState* and *PMCS*, where the latter is the sole subclass of the former. These classes are emphasized in this view because of their darker color.

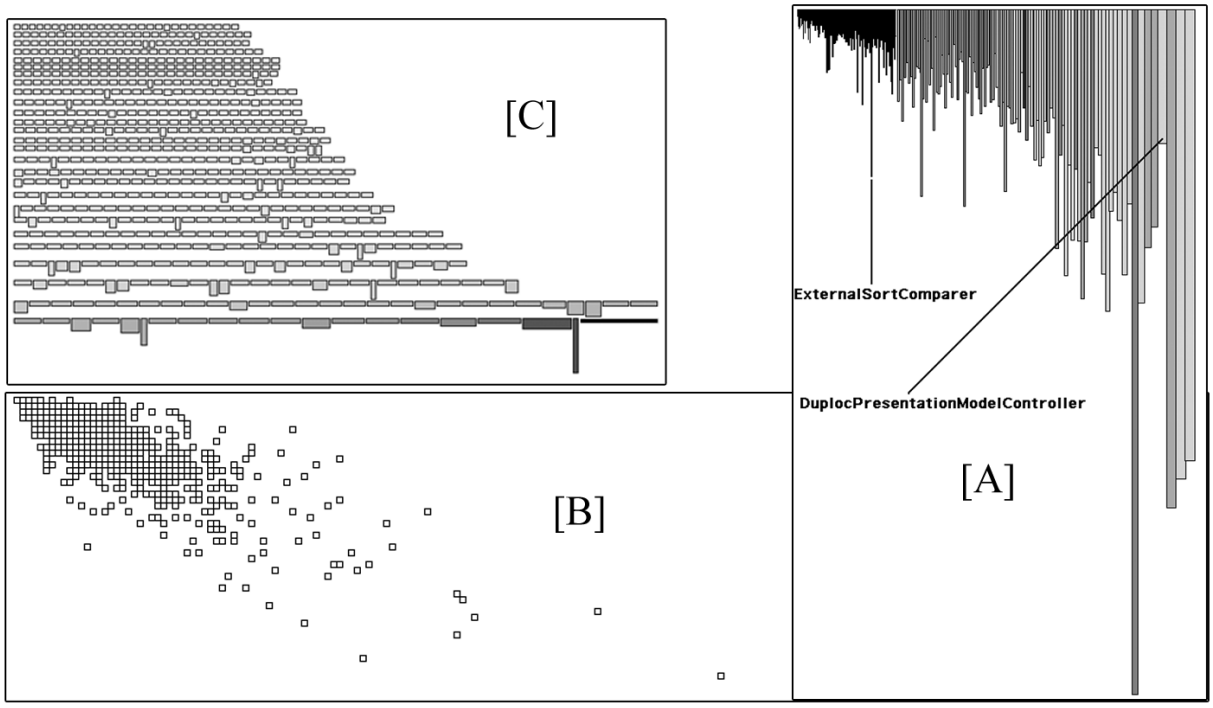

Fig. 3. [A]: A DATA STORAGE CLASS DETECTION view on the largest classes in terms of number of methods of Duploc. The color and height metrics represents the number of lines of code of each class, while the width represents the number of methods. The nodes are sorted according to their width. [B]: A METHOD STRUCTURE CORRELATION view of Duploc. As horizontal position metric we use the lines of code, while for vertical metrics we use the number of statements. [C]: A DIRECT ATTRIBUTE ACCESS view (Variation 3) of Duploc. The width of each attribute node represents the number of direct local accesses from within its defining class (NLA). The height of each node represents the number of accesses from outside of its class (NGA), while the color represents the number of total direct accesses. The nodes are sorted according to the color metrics.

DATA STORAGE CLASS DETECTION VIEW

Description: **Layout:** Stapled. **Target:** Classes. **Scope:** Subsystem. **Metrics:** Width: NOM. Height: WLOC. Color: NOM. **Sort:** -. **Example:** Figure 3.[A].
This view relates the number of methods (NOM) with the lines of code (WLOC) of classes and interprets this information in the context of a subsystem or small system. Ideally this view should return a staircase pattern from left to right, since the nodes are sorted according to the first metric and the two metrics are related. Note that this view works in any setting, *i.e.*, since it puts two values in relation it doesn't matter how big the actual measurements are.
Symptoms: (1) The staircase effect is broken by nodes which are too tall. These represent classes which have long methods compared to the classes which comply with the staircase pattern. (2) The staircase pattern is broken by nodes which are too short. These classes, given a certain number of methods, do not have the expected length in terms of lines of code. Such classes are often data storage classes, *i.e.*, classes which have short, simple methods, possibly only accessor methods. Data storage classes may point to sets of coupled classes being brittle to changes.
Variations: (1) To enhance the detection of data storage classes we use the number of attributes (NOA) as color metric, because data storage classes often have many attributes.
Scenario: We see in Figure 3.[A] that the fourth class from the right, *DuplocPresentationModelController* is very short (265 line of code) compared to the great number of methods (107) indicated by the position on the right. Upon closer inspection we see that the class contains dozens of one-line methods which return constant values. We also see the inverse case for the first tall class on the left named *ExternalSortComparer* which contains 12 methods for a total length of 330 lines. This class contains methods which can be refactored by splitting them up in smaller, more reusable pieces.

METHOD STRUCTURE CORRELATION VIEW
Description: **Layout:** Scatterplot. **Target:** Methods. **Scope:** Full system. **Metrics:** Position (X): LOC. Position (Y): NOS. **Sort:** -. **Example:** Figure 3.[B].
This very scalable view shows all methods using a scatterplot layout with the lines of code (LOC) and the number of statements (NOS) as position metrics. As the two metrics are related (each line may contain statements) we end up with a display of all methods, many of which align themselves along a certain correlation axis.
Symptoms: (1) Nodes to the right of the display represent long methods and should be further investigated as candidates for split method refactorings [24] [25]. (2) Nodes to the very left and top of the display represent empty methods. (3) Nodes to the top of the display, but not necessarily to the left, represent methods containing commented lines or possibly dead code. (4) Nodes to the left and more to the bottom of the display represent methods which are probably hard to read, as they contain several statements on each line. In this case one should check whether there are formatting rules within the application which are being violated.
Variations: (1) This view can be enriched using size metrics as well. One useful variation is using the number of parameters (NOP) for the size of the nodes, which reveals not only long methods but methods with many input parameters as well.
Scenario: We can see in Figure 3.[B] how well this view scales up: the figure shows nearly 5000 of Duploc's methods. Several method nodes seem to be good candidates for further investigations. All the methods longer than a certain number of lines (for example 30 or 50, depending on the average length of methods in the subject system) should be inspected. Note in this regard that the average length of Smalltalk methods is around 7 lines [31]. We can also see that there are many methods at the top of the display which therefore do not contain many statements. Upon closer inspection we can see this is partly due to code which is commented out (in some cases dead code), partly this is also due to very long comments written by the developer to explain what the methods are actually doing. Another insight which can come from this view is a general assessment of the system. We have seen that the methods tend to align themselves along a certain correlation axis. Depending on the age of the system the axis changes its angle: methods are written and corrected all the time, and slowly get messy with many statements on few lines. In this regard Duploc can still be considered

a young system.

DIRECT ATTRIBUTE ACCESS VIEW
Description: **Layout:** Checker. **Target:** Attributes. **Scope:** Full system. **Metrics:** Width: NAA. Height: NAA. Color: NAA. **Sort:** -. **Example:** Figure 3.[C].
This view uses the number of direct accesses (NAA) for the width, height and color of each attribute node, and sorts the nodes according to this metric.
Symptoms: (1) Small nodes at the top of the display represent attributes which are never accessed and may point to dead code. (2) Large, dark nodes at the bottom point to attributes which heavily directly accessed, which may lead to problems, in case the internal implementation changes. For such nodes one should also check whether accessor methods have been defined, and if yes why they are not always being used.
Variations: (1) Instead of using as size and color metric the number of direct global accesses, we use either the number of accesses via accessor methods to reveal how heavily these accessor methods are actually used. (2) We use as size and color metric the number of direct accesses by subclasses, in order to reveal coupling aspects of classes within inheritance hierarchies. (3) We use the number of local accesses (NLA) (from within the class where the attribute resides) for the width and the number of global accesses (NGA) (from outside of the class) for the height. Normally the attributes rendered like this should be as flat as possible, and in cases where this does not apply, a deeper inspection could be useful, since tall, narrow nodes represent attributes which are heavily accessed from outside of its defining class by means of direct accesses.
Scenario: In Figure 3.[C] we use a slight variation of the regular view definition and render for the width and the height the number of local, respectively the number of non-local accesses, while the color renders the total number of direct accesses. We see that Duploc uses a considerable number of attributes. The top row contains 11 attributes which are never accessed, and can therefore be removed. The bottom row contains the most heavily accessed attributes. For example the attribute *bvcm* belonging to class *BinValueColorerInterface* is directly accessed 77 times. Upon closer inspection we see that in fact the class defines accessor methods, but they are not consistently used, which may be risky [31]. Note also the tall, narrow attribute node at the bottom of this view. This attribute is heavily accessed directly from outside of its containing class. In such a case we suggest to define accessor methods and invoke them instead of directly accessing the attribute.

B. Evaluation

Case study. Our approach provided us with an initial understanding of the case study and helps us to identify some of the key classes without having to focus on the details. The developer of Duploc confirmed several of our findings and was surprised that we obtained our results in less than two days. Indeed, one of the major problems with large systems is to get an overview and some initial understanding at the beginning without getting lost in their intrinsic complexity [7]. The methodology based on clusters of views helps to stay focused at the different levels of understanding we want to gain. We cannot present all the results we obtained during this case study, as this would go beyond the scope of this paper. We rather limit ourselves to draw some specific conclusions on the major findings obtained during this case study, and some general conclusions on other case studies we have performed.

First Contact Views. The views in this cluster help us to get a first feeling for the size and structure of the system, and in more detail to see how a system's major hierarchies are composed and where larger classes are located. In the present case, we have seen that Duploc is composed of several standalone classes, and that a major part of Duploc is dedicated to the management of the graphical user interface. A first list of prominent classes and hierarchies of the system is useful to get an orientation. Especially on very large case studies, this cluster's views help to obtain results quickly.

Inheritance Assessment Views. The views in this cluster are useful for the easy understanding of the complex mechanisms related to inheritance. We can classify inheritance relationships and detect important classes in large hierarchies. Especially for larger hierarchies, which however this case study did not contain, this cluster's views reduce the time to understand complete inheritance hierarchies. In one special case, we

reverse engineered a system which contained very large inheritance hierarchies, with several hundreds of classes and where in one case the root class, had 97 direct subclasses. The views obtained after visualizing this hierarchy led us to coin the term *flying saucer* hierarchy, because of its very flat shape.

Candidate Detection Views. The views in this cluster help us to identify many candidates for closer examination. The problem with those candidates is that their number can be large. The reverse engineer can easily produce long lists of suspicious code fragments, classes, methods, etc., but the usefulness of such an approach is doubtful: in the end it is the software company that decides on which parts of their system they want to spend time and money for reengineering. In the case of Duploc, together with its developer we inspected several candidates and he confirmed our findings, but it is difficult to present the results in detail, because this would go beyond the scope of this article.

Industrial Experiences. We applied our approach on several large industrial applications ranging from a system of approximately 1.2 Million lines in C++ to a Smalltalk framework of approximately 3000 classes (600 kLOC). During our experiments we were able to quickly gain an overall understanding of the analyzed applications, identify problems, and point to classes or subsystems for further investigation. Moreover we learned that the approach is preferably applied during the first contact with a software system, and provides maximum benefit during the first one or two weeks of the reverse engineering process. However, due to non-disclosure agreements with the industrial partners, we cannot deliver a detailed report on those experiences.

We applied and refined our reverse engineering methodology by using our tool on industrial applications in an explorative way. The common point about these experiences was that the subject systems were of considerable size and that there were narrow time constraints (maximum 4 days). This led us to mainly get an understanding of the system's overall structure (subsystems, important hierarchies and their purpose) and produce overviews. We were also able to point out potential design problems (overly large classes, unused classes, dead code, overlong methods, unused attributes) and on medium-sized case studies we even had the time to propose possible redesigns of the system (for example in one case we suggested to inverse the order of the classes in a hierarchy and increase its stability by introducing the template method design pattern, which resulted in a considerable reduction of complexity of the hierarchy).

Taking the time constraints into account (none of the case studies lasted more than a few days) we obtained very satisfying results. The – often initially sceptical – developers were surprised that we had not only gained an overview over such large systems, but had also uncovered many design flaws in such a short time. Even though they were aware of at least half of the problems we found, many developers *saw* the complete software system they were working on for the first time. The typical result of each case study was a report containing a presentation of polymetric views of the system and a list of possible problems and errors, for example suspicious classes, overlong methods, unused attributes, etc. The developers liked our overviews (for example the SYSTEM COMPLEXITY view) and even used them for documentation purposes. The list of problems and design flaws we delivered was also well accepted, and although during the final discussion with the developers we saw that they were keen on examining them, we do not know to which extent this has been done, since the software companies in question were very protective regarding such information. We consider this protectiveness as harmful, as it would allow us to further improve our approach.

Moreover, our goal is to provide expressive views on a system, which can easily be complemented with code browsing. The time it takes the reverse engineer to go from a visualization to the source code level, must be kept as short as possible. In this context we speak about *opportunistic code reading*, *e.g.*, the polymetric views do not replace code reading, they support it and point a reverse engineer to the spots where code reading is needed. Combining the polymetric views with manual code browsing thus proved to be a good way to get the results [32]. The obvious conclusion is that tools are necessary but not sufficient on their own.

VI. IMPLEMENTATION

A. Moose, a Language Independent Reengineering Environment

Moose [16] is a language independent reengineering environment written in Smalltalk. It is based on the FAMIX metamodel [15], which provides for a language independent representation of object-oriented source

code and contains the required information for the reengineering and reverse engineering tasks performed by our tools. It is *language independent*, because we need to work with legacy systems written in different implementation languages. It is *extensible*, since we cannot know in advance all information that is needed in future tools. Since for some reengineering problems (*e.g.*, refactorings [33]) the tools might need for language specific information, we allow for language plug-ins that extend the model with language specific features. Next to that we also allow the tool plug-ins to extend the model with tool specific information.

The core FAMIX metamodel comprises the main object-oriented concepts – Class, Method, Attribute and Inheritance – plus the necessary associations between them – Invocation and Access. Note that the complete FAMIX metamodel includes many more aspects of the object-oriented paradigm, and contains source code entities like formal parameters, local variables, functions, etc. We opted against the use of UML because it is not sufficient for modeling source code for the purpose of reengineering, since it is specifically targeted towards OOAD and not at representing source code as such [34].

B. CodeCrawler

CodeCrawler uses Moose for representing software and uses the HotDraw framework [35] for the visualization part. In the remainder of this section we discuss some implementation issues and design decisions.

Extensibility. One lesson learned during all case studies we made, is that none of them is typical or normal. Every case study posed certain problems: size, use of atypical language constructs, use of domain specific aspects, to list just a few. A reverse engineering tool cannot be prepared for every situation which may arise and must therefore be easily extended and adapted to the current context. CodeCrawler provides for an easy way to integrate new kinds of entities and relationships. In the occasion of a research project dealing with a legacy system written in Cobol [36], the changes needed to enable visualizations of Cobol code were performed in a few hours. Thus, it doesn't necessarily need Moose entities, but can handle and visualize any kind of entity. The most recent examples include visualizations from the domain of concept analysis and visualizations of Prolog statements. CodeCrawler also exploits the properties of the entities within a Moose model. These properties, implemented as a dictionary, are freely and easily extensible, and are heavily used for example to add new metric measurements or to add new annotations, for example package affiliation, comments, etc. In this context we treat Moose models not as a read-only facility, but enrich it during the reverse engineering process with additional information which can in turn be used by CodeCrawler.

Interactivity. Software visualization needs to offer interactive facilities to the user. As most visualizations can be heavily parameterized, one must offer an easy way to do so by means of *direct manipulation idioms* [37], which give the user the freedom to directly manipulate the resulting visualization by means of zooming, scaling, deletion, elision, etc. [9]. In the case of the systems we reverse engineered, direct manipulation was necessary to reduce complexity by providing basic navigation support, as well as to cut down latency times between visualizations. In this context we claim that our visualizations do not merely represent source code, as in the case of static visualizations (*e.g.*, static pictures which cannot be manipulated), but they *are* the source code. This is further emphasized by another aspect, which we call *code proximity*, discussed below. Since we use the size and color of the nodes to render metric measurements, it is not possible to display the name or any other information in the nodes themselves, although CodeCrawler supports this as well (of course in this case the size does not render metric measurements anymore, but is dictated by the displayed contents). Since the user wants to know what he is interacting with, CodeCrawler provides the requested information in several ways: (1) By means of a tool-tip figure (2) By displaying it in the status field at the top (3) By displaying it in a separate window. Another issue in the context of interactivity is the definition of the measurement mapping function, whose principles we have presented in Section III-A: in order for the user to click on a desired node, the node must have a certain size. On the other hand the size of the node must render the underlying metric measurement as truthfully as possible. The first idea which comes to mind is a direct one-to-one mapping. However, this idea must be rejected because if a measurement is zero, the node will have no dimension. We have considered several possible solutions [29] for this problem, and have finally settled on defining a minimal node size (MNS) value, to which the metric measurements are directly added.

The MNS has the experience value of 4, it can however easily be changed by the user. Therefore we obtain a mapping function (width/height = MNS + metric measurement) that maps the metric measurement 0 on 4, 1 on 5, 2 on 6, 5 on 9, 10 on 14, 100 on 104, etc.

Code Proximity. Providing easy and fast access to the original source code can greatly reduce latency times. Since our goal is to provide alternative, yet expressive, views of a system, it is necessary to quickly verify the correctness of a visualization (*i.e.*, is that class really that big?), but is also important to "get down" from the visualization to the source code level. CodeCrawler provides access to the source code represented by the nodes in two ways. For Smalltalk code it can directly access and open a browser on the corresponding classes or methods. In the case of non-Smalltalk code we gain access to the right location in the right file by means of a *source anchor*, which is defined for every entity.

Scalability. One of the major issues software visualization tools are confronted with is scalability. CodeCrawler can visualize at this time ca. 1 Million entities. Note that we keep all the entities in memory.

Tool Usability. Our tool has been downloaded over 2000 times, and although we haven't performed a user survey yet, from personal and e-Mail discussions with the users, we have learned that after a short learning time they know what they can get out of each view. However, although the views are easily editable, we have also learned that most of the users apply the predefined views and seldom create new views of their own.

VII. Related Work

Since our approach is a mixture of two already present approaches, we discuss first the work performed in those two areas, before focusing on the methodological aspects of related approaches.

Software Visualization. The graphical representations of software used in the field of software visualization, a subarea of information visualization [10][38], have long been accepted as comprehension aids to support reverse engineering. Indeed, software visualization itself has become one of the major approaches in reverse engineering. Price *et al.* have presented an extensive taxonomy of software visualization, with several examples and tools [39].

Many tools make use of static information to visualize software, like Rigi [40], Hy+ [41], SeeSoft [42], ShrimpViews [43], TANGO [44] and the FIELD environment [45], to name but a few prominent examples.

Substantial research has also been conducted on runtime information visualization, called program visualization. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [46], Jinsight and its ancestors [47] and Graphtrace [48]. Various approaches have been discussed like in [49] where interactions in program executions are being visualized. In our current approach we do not exploit dynamic information. Richner has conducted research on the combination of static and dynamic information [50], where the static information is provided by the Moose Reengineering Environment.

Several systems make use of the third dimension by rendering software in 3D. Brown and Najork explore three distinct uses of 3D [9], namely (1) expressing fundamental information about structures that are inherently two-dimensional, (2) uniting multiple views of an object, and (3) capturing a history of a two-dimensional view. They exemplify these uses by showing screen dumps of views developed with the Zeus algorithm animation system [51]. However, they also state that "the potential use of 3D graphics for program visualization is significant and mostly unexplored". Some of the systems cited above make both use of 2D and 3D visualizations. Until now we have refrained from using 3D for our visualizations, mainly because it would contradict the lightweight constraint. However, we consider the exploration of the use of 3D as possible future work.

Metrics. Metrics have long been studied as a way to assess the quality and complexity of software [12], and recently this has been applied to object-oriented software as well [18][19]. Metrics profit from their scalability and, in the case of simple ones, from their reliable definition. However, simple measurements are hardly enough to sufficiently and reliably [52] assess software quality. Some metric tools visualize information using diagrams for statistical analysis, like histograms and Kiviat diagrams. TAC++ [53] and Crocodile [54] are tools that exhibit such visualization features. However, in all these tools the visualizations are mere side effects of having to analyze large quantities of numbers. In our case, the visualization is an inherent part of

the approach, hence we do not visualize numbers, but constructs as they occur in source code.

Methodology. To the best of our knowledge none of the approaches we reference in this paper presents a reverse engineering methodology, which can help a reverse engineer to apply a certain tool or technique. Storey *et al.* present in [8] some basic ideas on how to build a mental model during software exploration, but do not provide the much-needed, yet difficult to obtain, empirical evidence. We suppose this is because of the *ad hoc* nature of reverse engineering tools (including ours) and because software industry has not yet adopted such tools as concrete aids for their development process.

VIII. Conclusions and Future Work

In this article we have presented the *polymetric views*, lightweight visualizations enriched with software metrics. Furthermore we have presented a reverse engineering methodology based on clusters of the *polymetric views*. This methodology enables to quickly gain insights into the inner structure of large software legacy systems and helps to detect problems.

Furthermore, we have shown CodeCrawler, a reverse engineering tool which was built on the principles of using simple ideas and applying them constantly on industrial case studies. CodeCrawler has been successfully used for reverse engineering several large industrial software systems.

Finally, we have used our methodology by applying different views and have reverse engineered a case study. We have been able to understand different aspects of the case study, among which an overview of the application, a discussion on the used inheritance mechanisms, the detection of design patterns, the detection of several places where in-depth examinations are needed, as well as propositions on where possible refactorings could be applied.

We have also seen that reverse engineering is not a systematic process, but that the understanding of a system non-linear and complemented by *opportunistic* code reading. This corroborates the work of Mayrhauser and Vans [32] in which they show that the understanding resides at all level of astractions.

Our lightweight approach is especially useful in the first phase (one to two weeks) of a reverse engineering process. We believe it can be combined with more complex and traditional approaches (code reading being one of the simplest), because our approach can point out where these complementary approaches can/should be used. However, we also believe the approach could be used iteratively during the complete reverse engineering process to generate *snap shots* of the system at the different stages. This has however not been tried out systematically and is part of our future work.

A. Future Work

Language specific views. Since FAMIX is language-independent we have focused on developing views in this context. We believe there are views which exploit language specific information, for example modifier information in languages like C++ and Java or metaclasses in Smalltalk.

New entities and relationships. The introduction of new entities and relationships, which may but do not need to have an equivalent in software could help to generate new views based on these new artifacts. A way to interactively generate these artifacts can be supported by grouping mechanisms, similar to the ones implemented in Rigi [40], which group entities and relationships according to certain rules (*i.e.*, naming conventions, types, etc.).

Usability and navigation. The extensive use of direct-manipulation idioms [37], especially those relevant to the reverse engineering process, should further increase the malleability and flexibility of our tools. The introduction of navigation mechanisms can further increase the efficiency of the reverse engineering process.

3D. The use of the third dimension (see for example [55]) can help to exploit and visualize more semantic information, although we believe that using such techniques generates results which cannot be classified as "lightweight" anymore.

Forward engineering. We are convinced the presented approach has also many benefits for forward engineering, code browsing, and programming environments. We are currently integrating CodeCrawler with the VisualWorks Smalltalk development environment.

APPENDIX

I. SOFTWARE METRICS

In the context of this article we make use of the software metrics described in Table I. The metrics are divided into three groups, namely class, method and attribute metrics, *i.e.*, these are the entities the metric measurements are assigned to. Note that our metrics engine is able to compute many more metrics, which we have omitted in the table, as they are not mentioned within this article. Since one of our main constraints is to reengineer systems written in different object-oriented languages we have chosen to include in our metrics engine metrics whose computation does not depend on any language-specific features, but can be based directly on our language-independent metamodel, which we present in Section VI.

TABLE I
A LIST OF THE SOFTWARE METRICS USED IN THIS PAPER.

Name	Description
Class Metrics	
HNL	Number of classes in superclass chain of class
NME	Number of methods extended, *i.e.*, redefined in subclass by invoking the same method on a superclass
NMI	Number of methods inherited, *i.e.*, defined in superclass and inherited unmodified by subclass
NMO	Number of methods overridden, *i.e.*, redefined compared to superclass
NOA	Number of attributes (NOA = NIV + NCV)
NOC	Number of immediate subclasses of a class
NOM	Number of methods
WLOC	Sum of LOC over all methods
WNOC	Number of all descendant classes
Method Metrics	
LOC	Method lines of code
MSG	Number of method message sends
NOP	Number of (input) parameters
NI	Number of invocations of other methods within method body
NMAA	Number of accesses on attributes
NOS	Number of statements in method body
Attribute Metrics	
NAA	Number of times directly accessed. Note that NAA = NGA + NLA
NGA	Number of direct accesses from outside of its class
NLA	Number of direct accesses from within its class

II. LAYOUTS

In the context of this article we make use of the following layouts:

- *Tree.* It positions all entities according to some hierarchical relationship. See Figure 1.[B] for an example. This layout is essential to visualize hierarchical structures. In the case of object-oriented programming languages this applies especially for classes and their inheritance relationships.
- *Scatterplot.* It positions nodes in an orthogonal grid (origin in the upper left corner) according to two measurements. Entities with two identical measurements will overlap. This algorithm is useful for comparing two metrics in large populations. See Figure 3.[B] for an example. This layout is very scalable, because the space it consumes is due to the measurements of the nodes and not to the actual number of nodes.
- *Checker.* It sorts nodes according to a given metric and then places them into several rows in a checkerboard pattern. It is useful for getting a first impression, especially for the relative proportions between the measurements of the visualized nodes. See Figure 2.[A] for an example. This layout's advantage is that is uses little space to layout large numbers of nodes. Moreover, since the nodes are sorted according to a certain metric, it can also be used to easily detect outliers.
- *Stapled.* It sorts nodes according to the width metric, renders a second metric as the height of a node and then positions nodes one besides the other in a long row. This layout is used to detect exceptional cases for metrics that usually correlate, because it normally results in a steady declining staircase, while exceptions break the steady declination. See Figure 3.[A] for an example.

ACKNOWLEDGMENTS

We would like to thank Oscar Nierstrasz and the TSE review commission for giving valuable advice on the writing of this paper.

REFERENCES

[1] E. J. Chikofsky and J. H. Cross, II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, pp. 13–17, January 1990.
[2] E. Casais, "Re-engineering object-oriented legacy systems," *Journal of Object-Oriented Programming*, vol. 10, no. 8, pp. 45–52, January 1998.
[3] S. Rugaber and J. White, "Restoring a legacy: Lessons learned," *IEEE Software*, vol. 15, no. 4, pp. 28–33, July 1998.
[4] D. L. Parnas, "Software aging," in *Proceedings of International Conference on Software Engineering*, 1994.
[5] I. Sommerville, *Software Engineering*, Addison Wesley, sixth edition, 2000.
[6] A. M. Davis, *201 Principles of Software Development*, McGraw-Hill, 1995.
[7] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002.
[8] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Software Systems*, vol. 44, pp. 171–185, 1999.
[9] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, Eds., *Software Visualization - Programming as a Multimedia Experience*, The MIT Press, 1998.
[10] C. Ware, *Information Visualization*, Morgan Kaufmann, 2000.
[11] M. Petre, "Why looking isn't always seeing: Readership skills and graphical programming," *Communications of the ACM*, vol. 38, no. 6, pp. 33–44, June 1995.
[12] N. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, International Thomson Computer Press, London, UK, second edition, 1996.
[13] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," in *Empirical Studies of Programmers, First Workshop*, 1996, pp. 80–98.
[14] S. Ducasse and S. Demeyer, Eds., *The FAMOOS Object-Oriented Reengineering Handbook*, University of Bern, October 1999, See http://www.iam.unibe.ch/~famoos/handbook.
[15] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1 – the FAMOOS information exchange model," Tech. Rep., University of Bern, 2001.
[16] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an extensible language-independent environment for reengineering object-oriented systems," in *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.
[17] S. Demeyer, S. Ducasse, and M. Lanza, "A hybrid reverse engineering platform combining metrics and program visualization," in *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. October 1999, IEEE.
[18] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*, Prentice-Hall, 1994.
[19] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, 1996.
[20] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994.
[21] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tolls, *Graph Drawing - Algorithms for the visualization of graphs*, Prentice-Hall, 1999.
[22] A. J. Riel, *Object-Oriented Design Heuristics*, Addison Wesley, 1996.
[23] D. Roberts, J. Brant, and R. E. Johnson, "A refactoring tool for Smalltalk," *Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 4, pp. 253–263, 1997.
[24] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.
[25] K. Beck, *Smalltalk Best Practice Patterns*, Prentice-Hall, 1997.
[26] S. Ducasse and M. Lanza, "Towards a methodology for the understanding of object-oriented systems," *Technique et science informatiques*, vol. 20, no. 4, pp. 539–566, 2001.
[27] M. Lanza and S. Ducasse, "A categorization of classes based on the visualization of their internal structure: the class blueprint," in *Proceedings of OOPSLA 2001*, 2001, pp. 300–311.
[28] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings ICSM'99 (International Conference on Software Maintenance)*. September 1999, pp. 109–118, IEEE.
[29] M. Lanza, "Combining metrics and graphs for object oriented reverse engineering," Diploma thesis, University of Bern, October 1999.
[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison Wesley, Reading, Mass., 1995.
[31] E. J. Klimas, S. Skublics, and D. A. Thomas, *Smalltalk with Style*, Prentice-Hall, 1996.
[32] A. von Mayrhauser and A. Vans, "Identification of dynamic comprehension processes during large scale maintenance," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 424–437, June 1996.
[33] S. Tichelaar, *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*, Ph.D. thesis, University of Berne, December 2001.
[34] S. Demeyer, S. Ducasse, and S. Tichelaar, "Why unified is not universal. UML shortcomings for coping with round-trip engineering," in *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, Kaiserslautern, Germany, October 1999, vol. 1723 of *LNCS*, Springer-Verlag.
[35] R. E. Johnson, "Documenting frameworks using patterns," in *Proceedings OOPSLA '92*, October 1992, pp. 63–76, Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.
[36] S. Ducasse, M. Lanza, O. Nierstrasz, M. Rieger, and S. Tichelaar, "Beoc analysis report," Tech. Rep., University of Bern, 2000.
[37] A. Cooper, *About Face - The Essentials of User Interface Design*, Hungry Minds, 1995.
[38] S. K. Card, J. D. Mackinlay, and B. Shneiderman, Eds., *Readings in Information Visualization - Using Vision to Think*, Morgan Kaufmann, 1999.
[39] B. A. Price, R. M. Baecker, and I. S. Small, "A principled taxonomy of software visualization," *Journal of Visual Languages and Computing*, vol. 4, no. 3, pp. 211–266, 1993.

[40] S. R. Tilley, H. A. Müller, M. J. Whitney, and K. Wong, "Domain-retargetable reverse engineering," in *Proceedings of CSM '93 The Conference on Software Maintenance*. September 1993, pp. 142–151, IEEE Computer Society.
[41] M. P. Consens and A. O. Mendelzon, "Hy+: A hygraph-based query and visualisation system," in *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, 1993, pp. 511–516.
[42] S. G. Eick, J. L. Steffen, and S. Eric E., Jr., "SeeSoft—A Tool for Visualizing Line Oriented Software Statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, November 1992.
[43] M.-A. D. Storey and H. A. Müller, "Manipulating and documenting software structures using shrimp views," in *Proceedings of the 1995 International Conference on Software Maintenance*, 1995.
[44] J. T. Stasko, "Tango: A framework and system for algorithm animation," *IEEE Computer*, vol. 23, no. 9, pp. 27–39, September 1990.
[45] S. P. Reiss, "Interacting with the field environment," *Software - Practice and Experience*, vol. 20, pp. 89–115, 1990.
[46] D. B. Lange and Y. Nakamura, "Interactive visualization of design patterns can help in framework understanding," in *Proceedings of OOPSLA'95*. 1995, pp. 342–357, ACM Press.
[47] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides, "Visualizing the behavior of object-oriented systems," in *Proceedings OOPSLA '93*, October 1993, pp. 326–337.
[48] M. F. Kleyn and P. C. Gingrich, "Graphtrace – understanding object-oriented systems using concurrently animated views," in *Proceedings OOPSLA '88*, November 1988, pp. 191–205, Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.
[49] D. J. Jerding, J. T. Stansko, and T. Ball, "Visualizing interactions in program executions," in *Proceedings of ICSE'97*, 1997, pp. 360–370.
[50] T. Richner and S. Ducasse, "Recovering high-level views of object-oriented applications from static and dynamic information," in *Proceedings ICSM'99 (International Conference on Software Maintenance)*. September 1999, pp. 13–22, IEEE.
[51] M. H. Brown, "Zeus: A system for algorithm animation and multi-view editing," in *Proceedings of the 1991 IEEE Workshop on Visual Languages*, October 1991, pp. 4–9.
[52] S. Demeyer and S. Ducasse, "Metrics, do they really help?," in *Proceedings LMO'99 (Languages et Modèles à Objets)*. 1999, pp. 69–82, HERMES Science Publications, Paris.
[53] F. Fioravanti, P. Nesi, and S. Perli, "Assessment of system evolution through characterization," in *ICSE'98 Proceedings (International Conference on Software Engineering)*. 1998, IEEE Computer Society.
[54] C. Lewerentz and F. Simon, "A Product Metrics Tool Integrated into a Software Development Environment," in *Object-Oriented Technology Ecoop'98 Workshop Reader*, 1998, vol. 1543 of *LNCS*, pp. 256–257.
[55] J. Rilling and S. P. Mudur, "On the use of metaballs to visually map source code structures and analysis results onto 3d space," in *WCRE 2002 Proceedings*. IEEE, 2002, pp. 299 – 308, IEEE Computer Society.