A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualisation

Serge Demeyer, Stéphane Ducasse, Michele Lanza
Software Composition Group, University of Berne
Neubrückstrasse 12, CH – 3012 Berne, Switzerland
{demeyer,ducasse,lanza}@iam.unibe.ch — http://www.iam.unibe.ch/∼scg/

Abstract

Surprising as it may seem, many of the early adopters of the object-oriented paradigm already face a number of problems typically encountered in large-scale legacy systems. Consequently, reverse engineering techniques are relevant in an object-oriented context as well. This paper investigates a hybrid approach, combining the immediate appeal of visualisations with the scalability of metrics. We validate such a hybrid approach by showing how CodeCrawler —the experimental platform we built— allowed us to understand the program structure of, and identify potential design anomalies in a public domain software system.

Keywords: Reverse Engineering, Program Visualisation, Software Metrics, Object-Oriented Programming, Code-Crawler

1. Introduction

"While the benefits of object-oriented technology are widely recognised, the indiscriminate use of object-oriented mechanisms and weaknesses in analysis and design methods are rapidly leading to a new generation of inflexible legacy systems." [3].

The ability to reverse engineer object-oriented legacy systems has become a vital matter in today's software industry. Early adopters of the object-oriented programming paradigm are now facing the problem of transforming their object-oriented legacy systems into full-fledged frameworks, hence they need to understand the inner workings of their legacy systems and identify potential design anomalies. However, since legacy systems tend to be big — hundreds of thousands lines of poorly documented code are not an exception— there is a definite need for approaches aiding in program understanding and problem detection.

Among the various approaches that exist today, two seem very interesting for large scale reverse engineering. One is *program visualisation*, often applied because good visual displays allow the human brain to study multiple aspects of complex problems in parallel (This is often phrased as "One picture conveys a thousand words"). Another is *metrics*, because metrics are often applied to help assess the quality of a software system and because they are known to scale up well.

This paper investigates a hybrid approach to understand existing program structures and identify potential design anomalies. The approach is based on the combination of *quite trivial graph layouts* and *easy to compute code metrics*. The aim for simplicity is driven by our long term goal, which is to identify reverse engineering techniques that can be easily incorporated in scriptable reengineering tool sets like Rigi [24, 31] or RainCode [34]. Having identified such simple techniques, one reverse engineer should be able to incorporate them in a reverse engineering tool within a very short amount of time — say, a couple of days. Afterwards, the whole reverse engineering team should be able to gain back that time by applying the tool in their daily working practices.

The paper starts with an overview of the hybrid reverse engineering approach (Section 2). Afterwards, we report on a case study (i.e., the SMALLTALK Refactoring Browser [29]), which illustrates the feasibility of the approach for program understanding and problem detection (Section 3). Next, we briefly present CodeCrawler, which is the platform we have developed to experiment with various combinations of metrics and program visualisations (Section 4). Then we discuss the advantages and limitations of our ap-

proach, including a sketch of the future work (Section 5). Finally, we provide an overview of the related work (Section 6), and end with the conclusion (Section 7).

2. Combining simple metrics and trivial graphs

Principle. We enrich a simple graph with metric information of the object-oriented entities it represents. Given a two-dimensional graph we can render up to five metrics on a single node simultaneously.

1. **Node Size.** The width and height of a node can render two measurements. We follow the convention that the wider and the higher the node, the bigger the measurements its size is reflecting.

2. **Node Position.** The X and Y coordinates of the position of the node can reflect two metric measurements. This requires the presence of an absolute origin within a fixed coordinate system, therefore not all layouts can exploit this dimension.

3. **Node Colour.** The colour interval between white and black can display yet another measurement. Here the convention is that the higher the value the darker the node is. Thus light gray represents a smaller metric measurement than dark gray.
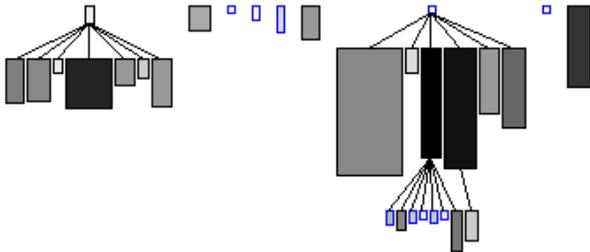

Figure 1. Inheritance Tree; node width = NIV, node height = NOM and colour = NCV.

Figure 1 shows an example of an inheritance tree enriched with metrics information. The nodes represent the classes, the edges represent the inheritance relationships. The size of the nodes reflects the number of instance variables (width) and the number of methods (height) of the class, while the colour tone represent the number of class variables. The position of a node does not reveal a metric as it is used to show the location in the inheritance tree.

Metrics. We constrain ourselves to apply metrics that are relatively simple to collect from source code as source code

is usually the most reliable source of information. The particular metrics included in our experiment are listed in Table 1.

Name	Description
Class Metrics	
HNL	Hierarchy nesting level, number of classes in superclass chain of class
NAM	Number of abstract methods
NCV	Number of class variables
NIA	Number of inherited attributes (defined in all superclasses)
NIV	Number of instance variables
NMA	Number of methods added, i.e. defined in subclass and not in superclass.
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.
NMO	Number of methods overridden, i.e. redefined compared to superclass.
NOC	Number of immediate children of a class.
NOM	Count all methods in class.
WLOC	Sum of all lines of code in all method bodies of class
WMSG	Sum of number of message sends in all method bodies of class.
WNAA	Number of times all defined attributes are accessed
WNI	Number of invocations of all methods
WNMAA	Number of all accesses on attributes
WNOC	Number of all descendants
WNOS	Sum of number of statements in all method bodies of class.
Method Metrics	
LOC	Lines of code in method body.
MHNL	Hierarchy nesting level of class in which method is implemented
MSG	Number of message sends in method body.
NI	Number of invocations of other methods
NMAA	Number of accesses on attributes
NOP	Number of parameters
NOS	Number of statements in method body.
Attribute Metrics	
AHNL	Hierarchy nesting level of class in which attribute is defined
NAA	Number of times accessed

Table 1. Selected Metrics.

Actual visualisation. The actual visualisation depends on three factors:

1. *The graph type.* Its purpose is to emphasise those aspects of a system that are relevant for reverse engineering. For instance, a tree graph is good for displaying hierarchical information; a circle graph is well suited for showing communication between components; and a confrontation graph is best for visualising dependencies between modules (see Table 2).

2. *The layout variation* Starting from the type of the graph, layout variations further customise the actual visualisation. The layout takes into account the choice of the displayed entities and their relationships plus

issues like whether the complete graph should fit onto the screen, whether space should be minimised, whether nodes nodes should be sorted.

3. *The metric selection.* Once the layout variation is chosen, metrics selected from Table 1 are incorporated into the graph.

Graph types. We selected a small set of graph types for their simplicity and easy layout algorithm. Table 2 lists their properties and below is a brief description of each of them. For a more detailed description of the graphs and a discussion of their advantages and limitations, we refer to [21].

Tree. Positions all entities according to some hierarchical relationship. Most often used to display class hierarchies. Layout variations are possible by aligning nodes relative to the position of their parent (left, center or right). [example Figure 3]

Correlation. Positions entities in an orthogonal grid (origin in the upper left corner) according to two measurements. Entities with the same measurement will overlap. Useful for comparing two metrics in large populations. [example Figure 7]

Histogram. Positions nodes along a vertical axis depending on one measurement. Nodes with the same measurement are then positioned in rows, one beside the other. Useful for analysing the distribution within a population. Layout variations result mainly from rendering additional metrics on the width of a node. [example Figure 6]

Checkers. (The result resembles a checkers board, hence the name of the graph.) Sorts nodes according to a given metric and then places them into several rows, starting a new row when the row length exceeds the square root of the total number of nodes. Useful for getting a first impression of rather small population, especially for the relative proportions between measurements. [example Figure 2)]

Stapled. Sorts nodes according to a given metric, renders a second metric as the height of a node and then positions nodes one besides the other in a long row. Is used to detect exceptional cases for metrics that usually correlate, because the stapling effect will normally result in a steady inclining staircase, yet exceptions will break the steady inclination. [see [21]]

Confrontation. Visualises two different kinds of entities and the relationships between them. Used mainly for analysing access patterns between attributes and methods. The two kinds of entities are positioned in two separate rows and then edges are drawn to represent the relationships. Layout variations are achieved by sorting or splitting the rows. [example Figure 9]

Circle. Distributes entities uniformly over a circle; then draws relationships as lines. Useful for displaying invocation relationships between methods. Layout variations are possible by rendering a metric on the radius for each entity and sorting the entities, which results into a *spiral*. Another variation is clustering entities according to the radius metric to achieve *concentric circles*. [see [21]]

Graph Type	Metrics	Entities	Sort	Scope
Tree	3	C		Global
Correlation	5	CMA		Global- Local
Histogram	3	CMA	+	Global- Local
Checkers	3	CMA	+	Global- Local
Stapled	3	CMA	+	Global- Local
Confrontation	3 + 3	MA	+	Local
Circle	3	CMA		Global- Local

Metrics specifies how many metrics can be rendered by the graph (3 + 3 stands for two separate groups of entities where each can render 3 metrics). *Entities* refers to the kind of entities the graph can be applied upon: C for class, M for method and A for Attributes. *Sort* indicates whether sorting the nodes enhances the graph. *Scope* specifies if the graph can be applied to a complete (sub)system or only to a local entity like a class or a method.

Table 2. Selected Graph Layouts.

Now that we have defined the metrics (Table 1) and the graphs (Table 2), the next section illustrates how we combine the two of them to obtain an initial overview of a software system and focus on potential design anomalies.

3. Case study

In this section, we apply a series of graph layouts enriched with metric information to reverse engineer an existing software system (Section 3.1). Afterwards, we evaluate how well the combination of metrics and visualisation techniques helps us to understand the system and identify potential problems (Section 3.2). Finally, to counter the potential critique that one case study is little evidence for supporting the claim, we say a few words on some other experiments with industrial systems.

The particular software system used in our experiment is the Refactoring Browser [29] which is well-known throughout the SMALLTALK community. To give an idea about the size of the system: the Refactoring Browser consists of 166 classes (not counting the meta-classes), 2365 methods, 365 instance variables, 2198 instance variable accesses and 9780 method invocations.

Reporting about a case study is quite difficult without sacrificing the explorative nature of the approach. Indeed, the idea is that different graphs provide different yet complementary perspectives. Consequently, a concrete reverse

engineering strategy should apply the graphs in some specific order, although the exact order should vary depending on the kind of system at hand and the kind of questions driving the reverse engineering project. Therefore, readers should read the case study report as one possible use case, keeping in mind that reverse engineers must customise their approach to a particular reverse engineering project.

3.1. Reverse engineering the Refactoring Browser

The report on the reverse engineering case study depicts various kinds of graphs, some of them providing overviews of the classes and methods in the system, others focusing on possible problems in the design.

1. Class size overview: checkers graph. One of the first impressions of the system that reverse engineers desire is a feeling for the raw physical measures of a system. For that purpose, we generate a so called *checkers graph* with lines of code as node size, the number of instance variables as colour tone and we sort the nodes using lines of code as criterium (see Figure 2).
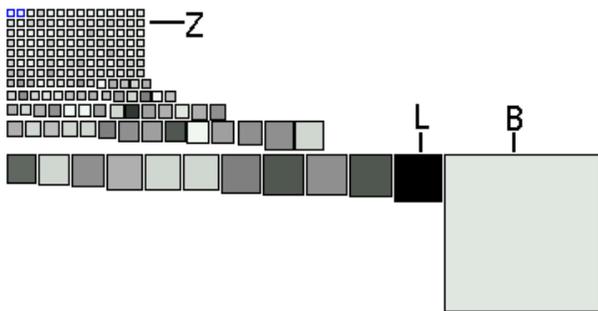
Figure 2. Class Size Overview via Checkers Graph; node size = LOC, and colour = NIV.

Interpretation. The checkers graph is useful for showing relative proportions between the system elements. In this particular case it shows the proportions among the classes of the software system in terms of lines of code. Through sorting it is easy to identify the largest and smallest classes. In this graph the biggest node represents the class BrowserNavigator with 1495 lines of code marked as B. The second biggest class with 441 lines of code is called BRScanner marked as L. We are also able to see that many classes in the system (marked as Z) are very small, and that there are some empty classes positioned on the upper left corner. (This last detail is only visible on the screen and not in the paper version, because metric measurements equal to zero render the nodes with a blue border).

2. Inheritance overview: trees. To assess the size and complexity of the system, we build an *inheritance tree*. We

use as node size the number of instance variables (width) and the number of methods (height), while the colour tone represents the lines of code of the class (see Figure 3).
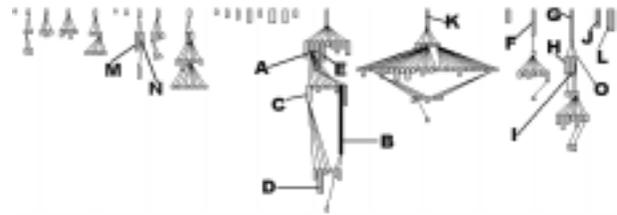
Figure 3. Inheritance Overview via Tree; node width = NIV, node height = NOM and colour = WLOC.

Interpretation. We observe a few main hierarchies with a high proportion of very small classes. Then, using the number of methods per class as a criterium we identify some candidate classes for further investigation: (1) the smallest class (O) is completely empty and (2) 13 classes are quite large (between 40 and 175 methods) (B).

These 13 classes can be classified according to their position in the inheritance tree: being a leaf (D, I), being on top of a hierarchy (F,G,K), being in the middle of the hierarchy (A, B) or being alone (E,J,L). Large sibling classes like (H, I) and (N, M) are good candidates for refactoring analysis because some of the code may be moved up in their common superclass.

An example of possible further investigation is the large class called BrowserNavigator that implements 175 methods (marked B) whereas its superclass Navigator (A) already implements 70 methods. Another interesting case is the class called BRScanner (named L), which implements 49 methods and defines 14 instance variables.

3. Focus on subclass relationships: more trees. Following the analysis of Figure 3, we want to use a graph as a problem detection tool. More precisely, we want to understand the relationship between some of the previously identified classes (A, F, G) and their subclasses. For that purpose we display a portion of the inheritance tree with metrics NMA (number of methods added to the ones defined in the superclass), NMO (number of methods overridden in the subclass) and NME (number of methods extending superclass methods) to assess the corresponding ratios overridden (see Figure 4).

Interpretation. The fact that the classes A, F and G are flat nodes is normal: they define much functionality (width), but can override only little (height) because they are at the root of their respective hierarchies. For the subclasses two different situations occur: either the subclasses are flat (B), or they are tall (H, I and the subclasses of F).
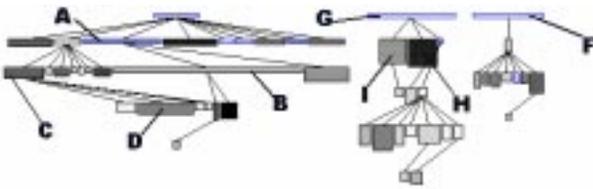
Figure 4. Focus on the Inheritance Tree; node width = NMA, height = NMO, and colour = NME.

This graph shows that the inheritance relationship can somehow be qualified: the subclasses of *A* add a lot of methods and override very little, whereas the subclasses of *G* tend to override more methods than they add. So we can consider *G* as a class designed to be partly redefined whereas *A* can be seen as a complete piece of functionality to be reused without modification.

4. Method sharing between (sub)classes: even more trees. A second example of a graph that may be used as a problem detection tool is one that analyses whether some classes in the system influence their subclasses in terms of method sharing. To evaluate this influence, we display the main inheritance hierarchies using the number of all descendants as node height, the number of methods as size metrics for the nodes and the number of immediate children as colour metric (see Figure 5).

Interpretation. The hierarchies rooted at classes *a* and *b* seem to represent a good factorisation of methods, because all the nodes are taller than they are wide (thus, lots of classes will inherit the methods defined) and because nodes become smaller when we come near to the leaves (thus, deeper in the hierarchy, less methods are defined). In contrast, the classes *c* and *d* do not introduce a lot of functionality that is shared with their subclasses, because these nodes are very flat. We also observe that the hierarchies with *a* and *e* as root classes are top-heavy, because the biggest part of the functionality is implemented in them (this is reflected by their height). Also, the classes *c* and *d* can be identified as intermediate abstract classes: their size shows very little functionality, while their dark colour reflects that they have many children. These classes introduce a new level of abstraction, which can be also gathered by their names: *c* is called MethodRefactoring, while its children are named AddMethodRefactoring, ExtractMethodRefactoring, etc.

5. Overview method size: histogram. After having inspected the classes, it is time to turn to the finer grained elements of a system, the methods. Just as with classes (see Figure 2), reverse engineers first desire an impression of the raw physical measures of a system, but this time must deal with the large number of items resulting from the finer
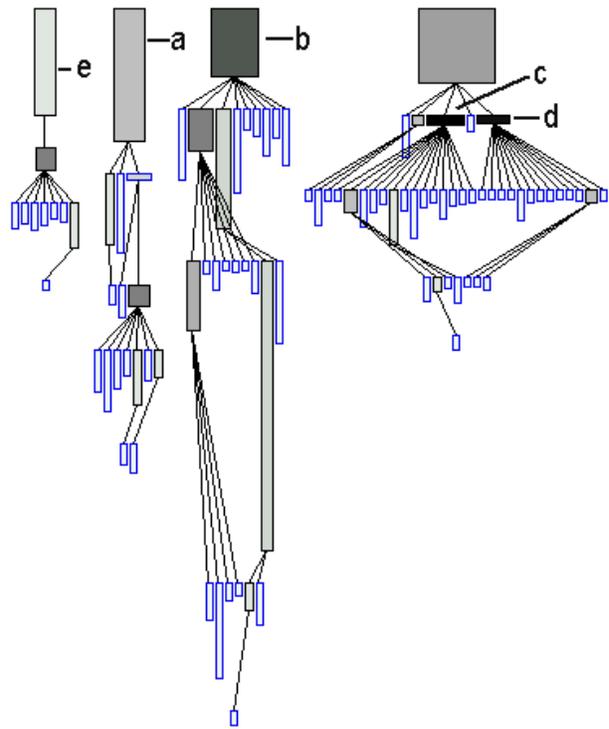

Figure 5. Method Sharing in the Inheritance Tree; node width = WNOC, node height = NOM and colour = NOC.

grained analysis. Therefore, a *histogram* is more appropriate, as it gives a good overview of the distribution of the elements with respect to a certain metric. To get a feeling for the distribution of method sizes within a system, we use a histogram showing the lines of code (see Figure 6).
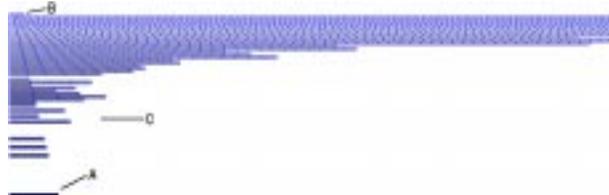

Figure 6. Method Size Overview via Histogram; node width = LOC, vertical position = LOC and colour = LOC.

Interpretation. By inspecting the histogram, we learn that less than 30 methods on the total of 2365 methods have more than 29 lines of code (*C*). The biggest method has 65 lines of code (*A*), which is a lot when the average method size in SMALLTALK is around 6 lines. We can also locate a few empty methods (*B*) which should be inspected in further detail.

6. Overview method size: correlation. While a histogram is good to get a feeling for the distribution of system elements according to one metric, it is not optimal for analysing a system as it emphasises a single metric only. In contrast, the *correlation graph* is one of the graphs that may render up to five different metrics per node. Nevertheless, we usually restrict ourselves to three metrics to achieve the effect of all nodes having uniform size.

In Figure 7, the position metrics chosen are LOC (lines of code) for the x coordinate and NOS (number of statements) for the y coordinate. The colour also reflects the lines of code to emphasise this aspect. Note that in a correlation graph, the metrics may overlay each other, which is the case for the nodes in the upper left corner.
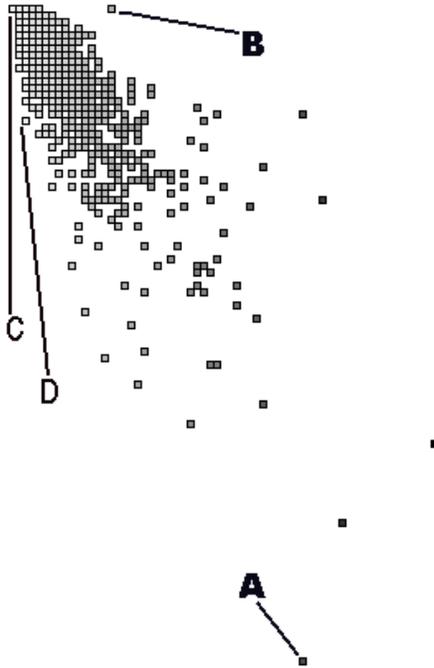

Figure 7. Method Size Overview via a Correlation Graph; node position = (LOC,MSG) and colour = LOC.

Interpretation. The first nodes to be considered are the ones on the topmost horizontal line, because their method bodies do not include any statements (*C*). Most of them are empty methods, which often represent hook methods —i.e. places where to plug in additional behaviour— thus are interesting for reverse engineering. However, some of them may represent dead code. For instance, node *B* represents a method that has 16 lines of code but no statements at all. Checking the corresponding source code, we see that the whole body of the method has been commented out.

The second interesting region of the correlation graph are the leftmost vertical columns, especially the bottom part of

them (*D*). The nodes there tend to have much more statements than lines of code, which seems strange. These usually correspond with unusual formatting of code, hence are worth to explore further as the developers treated them in a special way.

The most interesting nodes are on the ones on the outer edges of the correlation graph, representing methods that have high metric values. We identify for example a method with 99 statements on 45 lines of code (*A*) which seems a good candidate for a split.

7. Class cohesion overview: checkers graph. One of the generic principles in designing software systems is to maximise the cohesion within and minimise the coupling between the systems components. Consequently, we want to use some graphs for checking the coupling and cohesion. In particular, we take a closer look at the class cohesion in Figure 8 via a *checkers graph* with node size NOM (number of methods) and WNAA (number of accesses on attribute defined in the class) and colour tone NIV (number of instance variables).
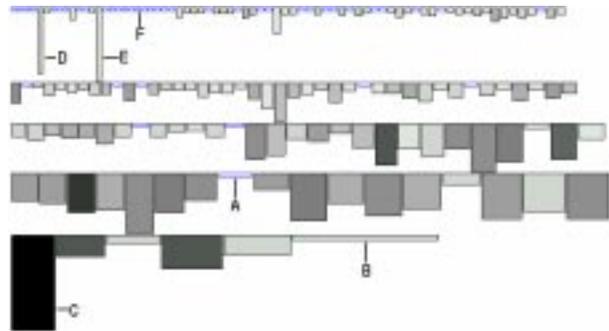

Figure 8. Class Cohesion Overview via Checkers Graph; node width = NOM, node height = WNAA, and colour = NIV.

Interpretation. Quite noteworthy in this graph is that the nodes differ heavily in shape and colour. The white nodes do not have instance variables (*A*), therefore can not have any accesses on them, hence their flat shape. The nodes *D* and *E* are special because of their narrow shape and light colour; both have few methods and instance variables, while at the same time their instance variables are accessed by a large number of methods. The reason for this is that their instance variables are directly accessed by their subclasses; the use of accessor methods might be more advisable for reasons of encapsulation. The class BrowserNavigator (*B*) strikes once again for its large number of methods and its small number of instance variables. Finally, the class which looks most suspicious from a cohesive point of view is the class BRScanner (*C*): it has the largest number of instance variables (it is black), it does not have that many methods,

6

yet it has a lot of accesses to the instance variables (it is shown as a long narrow rectangle).

8. Focus on class cohesion: confrontation graph. Given the analysis of Figure 8, we now focus on the class BRScanner (*C*). More precisely we want to understand the internal coupling of the class by looking at the way the methods access its instance variables. Therefore, we apply a *confrontation graph*: a graph where an edge between an instance variable and a method represents an instance variable accessed by the method. The resulting graph is shown in Figure 9. The instance variables are the middle row of nodes and the methods are the top and bottom row of nodes.
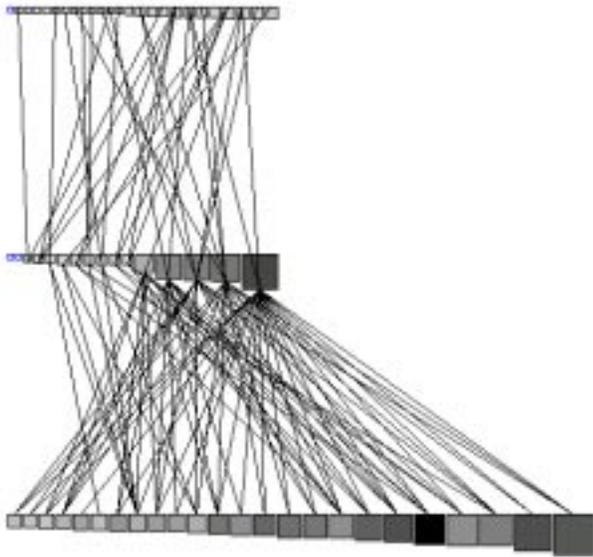

Figure 9. Focus on Class Cohesion of BRScanner via a Confrontation Graph; method node height and width = NOS and colour = LOC; attribute node height and width and colour = NAA.

Interpretation. The confrontation graph reveals that there are no apparent clusters in the way methods access instance variables. This is a sign that this class is quite cohesive, thus that a split is going to be difficult if not impossible. However, it also implies that subclassing will be quite difficult. Note that in other experiments, we have —by means of the very same confrontation graph— discovered classes that can easily be split. Unfortunately for our case study, but good for the Refactoring Browser, we did not find such a situation in the case study described here.

3.2. Case study evaluation.

Results. Using a hybrid approach combining metrics and graph layouts on the Refactoring Browser provided us with an initial understanding of the system without having to dive into the details. We also have been able to reveal some design anomalies, such as classes and methods that may be split, or suspicious class hierarchies. Due to the limited space of this paper we could not show all of the system with all of the graph layouts and metrics (Table 1 and Table 2), nor could we include other useful graphs. Interested readers may consult [21] for a more complete description.

Below is a more detailed overview of the lessons learned from the case study.

- **Initial overview**. The *checkers graph* provided us with a feeling for the proportions in terms of code size and helped us to identify extreme cases.

- **Qualify inheritance hierarchies.** The inheritance overviews helped us to identify and qualify the main hierarchies: the graphical navigation and tool classes like BrowserNavigator, the refactoring classes, the abstract syntax tree representation, the parser and the scanner. More than just displaying hierarchies, Code-Crawler helped us to understand the quality of the hierarchies: for example the refactoring hierarchy whose root is *K* in Figure 3 is composed by a high number of small classes whereas the abstract syntax tree hierarchy, whose root is *G* in Figure 3, is composed by some more substantial classes.

- **Insight in inheritance quality.** The complementary perspectives on the inheritance tree allowed us to have a better understanding of inheritance relationships. We found that some superclasses were defining functionality that should be specialised by their subclasses, whereas others were defining functionality that was reused without having to be specialised.

- **Identification of exceptional classes**. Even if the Refactoring Browser is quite well-designed we did identify some exceptional classes that would benefit from refactoring.

- **Overview of the methods**. We identified possible hook methods and got a first view of the overall method quality. Only a few outliers possessed overly high lines of code.

- **Internal class coupling**. Using confrontation graphs, we were able to have a rough idea of the coupling between a class and identify clusters of instance variables.

7

3.3. Other Experiments.

Besides the case study described in this paper, we have run other experiments on industrial systems implemented in C++ and SMALLTALK. Due to non-disclosure agreements, we cannot publish the results of these experiments. However, one experiment is particularly worthwhile to mention, because it provides anecdotal evidence for how well a hybrid but simple approach may outperform more specialised and complex approaches.

During the course of one week, a team of reverse engineers went for an on-site visit. The assignment was to use some reverse engineering tools to learn as much as possible about a particular C++ system. On the last day, each of the reverse engineers was asked to report their findings to the original developers, to evaluate the usefulness of the tools. Besides the hybrid approach described here, there was one pure metrics tool and one pure visualisation tool. Comparing our approach with the pure metrics and visualisation approaches, we made two observations. First, the simplicity of our hybrid approach helped a lot to get early results — the other two tools needed at least two days configuration time to be able to parse the system, because their specialised analysis required quite detailed knowledge about the source code. In contrast, our simple approach does not need a full C++ parser and we were able to display some graphs already during the first day. As a result, we were more productive during this short time span, hence were able to raise more questions and point to more problems than the other two reverse engineers using more specialised tools. Second, based on the feedback of the development team during the fifth day, the results obtained were highly appreciated. Moreover, the development team confirmed us that the little we understood from the system was correct and that the potential design anomalies we discovered were either known problems, or issues that would be further investigated.

4. Validating the approach: CodeCrawler

We based our experiment on the hypothesis that an approach which combines the immediate appeal of program visualisations with the scalability of metrics is applicable for program understanding and problem detection. However, to validate such a hypothesis, we immediately faced the problem of the vast number of possible combinations to explore. Indeed, given a list of 28 metrics (see Table 1) and knowing that one program visualisation is able to display up to 5 measurements simultaneously, there are hundreds of possible combinations per visualisation.

Rather than trying out all possibilities, we developed an exploratory tool, named CodeCrawler, to gain experience with combinations that provide good results. CodeCrawler is an open platform providing a graphical representation of source code combined with object oriented metrics. Figure 10 shows a screen dump of the tool in action.
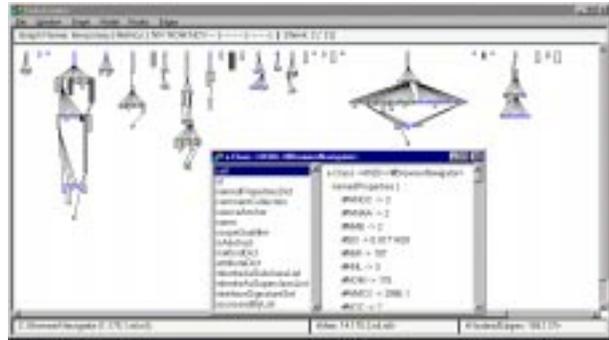

Figure 10. The CodeCrawler platform at work: (1) an inheritance tree with x node size = NIV, y node size = NOM, colour = NCV and (2) inspecting the data of a represented entity.

Practical considerations. Besides testing the combinations of graphs with metrics values, we have been confronted with practical considerations like the minimal size of a node or the size of the screen. These considerations have influenced the graph definitions, hence we report them here.

For the node size, we chose to implement the mapping such as to accurately reflect the measurement in the size on the screen with a slight distortion in case the measurement drops below a certain threshold. A minimal node size is a purely practical issue that is necessary when we want the graph to be interactive, since clicking with the mouse pointer on nodes only one or two pixels wide is difficult.

For the node colour, we chose to avoid optical overload by limiting the use of colours and using gray tone. Of course, the usage of different colours is a good way to attract the attention of the eye, but too many of them results in overload. The solution with gray tones has the advantage that numerical information can be transferred by gray values: we map numerical values (e.g. the metric measurements) into a colour interval ranging from white to black. Although this is a good way to display a supplemental metric, we experienced that the perception of a gray tone is less precise than the perception of size. Thus, the gray tone is only useful for the detection of extreme values.

Note that CodeCrawler supports different distributions (e.g., linear, logarithmic) represent the size of the nodes plus different modes like the shrinking of the graphs to fit the graphs into the size of a screen. It is also able to mark nodes whose metrics exceeds a certain threshold value.

Supporting reverse engineering. CodeCrawler provides a number of features that greatly enhance reverse engineering activities. First of all, a reverse engineer may configure and save sets of graph parameters. Next, CodeCrawler shows all

the information of the current displayed graph (top border) and the information related to the entity currently selected (bottom border). In Figure 10 the metrics are NIV, NOM and NCV applied on class entities, the last investigated class is BrowserNavigator that has 1 instance variable, 175 methods and 1 class variable.

Once the graph is displayed, several operations are possible. These include highlighting all the edges arriving at a specific node, following a particular edge, applying a new graph to a specific node. It is also possible to query the graph to locate nodes via their name. Finally, each graph entity is linked to the code entity that it represents, so the reverse engineer can browse the code related to the displayed entity as well as inspect its metrics.

Implementation. CodeCrawler is developed within the VisualWorks SMALLTALK environment, relying on the HotDraw framework [16] for its visualisation. It uses the facilities provided by the VisualWorks environment for the SMALLTALK code parsing. For other languages like C++ and Java it relies on Sniff+ to generate code representation encoded using the FAMIX Model [33] (see below).

Scale. During our experiments the maximum number of entities we loaded in CodeCrawler was 198301 (3268 classes, 35538 methods, 5420 attributes, inheritance relationships 3266, 123066 method invocations and 27743 attribute accesses). We emphasise that this limit is not linked to the approach but to the libraries used to implement CodeCrawler plus the available memory while running the system.

5. Discussion and future work

Data model. CodeCrawler is based on a language independent representation of object-oriented source code, named FAMIX (FAMoos Information EXchange model, see [33] and [7]). FAMIX is defined in the context of the FAMOOS project, which investigates tools and techniques for transforming object-oriented legacy systems into frameworks. See http://www.iam.unibe.ch/∼famoos/ for more information. FAMIX exploits meta-modelling techniques to make the data model extensible.

A simplified view of the FAMIX data model comprises the main object-oriented concepts —namely Class, Method, Attribute and InheritanceDefinition— plus the necessary associations between them — namely Invocation and Access (see Figure 11).

- **Advantage:** Due to the language independent nature of FAMIX, CodeCrawler has already been applied to software systems developed in C++ and SMALLTALK.

- **Limitation:** In practice, we must limit ourselves to languages that can be parsed and translated into a
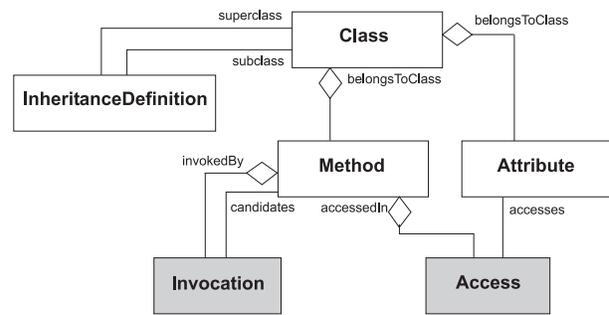

Figure 11. A simplified view of the FAMIX Data Model.

FAMIX representation. At the time of writing, these are SMALLTALK, Java and C++. Ada parsing is currently being implemented.

Metrics. The measurements of given source code entities are attached to the FAMIX counterparts (see Figure 11). Thus, a class entity knows about its number of methods and number of attributes; a method entity knows about its number of statements, etc. Most of the metrics defined in Table 1 can be derived from the data model itself, thus are language independent. However, a few of them (i.e.; number of statements in method body - M-NOS; number of methods overridden & extended - NMO & NME) require a language dependent interpretation.

- **Advantage:** Since most of the metrics applied in our approach are language independent in nature, a lot of the identified graphs can be used across different implementation languages.

- **Limitation:** A considerable part of the reverse engineering capabilities —especially analysing the quality of the inheritance tree— is based on the language dependent metrics. Thus, if one wants to reuse a hybrid metrics-visualisation tool across implementation languages, some language dependent customisation will be required.

5.1. Limitations

We have seen that while maintaining the initial constraints (simple graphs and simple metrics) we can use the hybrid approach for program understanding and problem detection. However, the hybrid approach has some inherent limitations.

1. It is purely based on static information and often the dynamic behaviour of a system is also crucial for understanding [26], [20], [15], [28].

9

2. The simplicity of the approach is also one of its limits. Especially, automatic clustering of entities before generating the graph could be a great enhancement to support reverse engineering activities [24].

3. Reverse engineering larger software systems requires other information besides the simple FAMIX data model in Figure 11. In particular, we must know about the packages or subsystems composing the software system, the files containing the code or the processes involved [1]. Incorporating new entities into the data model is definitively an issue in the future development of the tool, yet this will cost in terms of extra complexity.

5.2. Future work

The following issues will be addressed in the future:

- **Exploiting Type Information.** The first experiments we made were done using software systems developed in SMALLTALK which is a dynamically typed language. The later experiments with Java and C++ systems continued on this basis and did not exploit the availability of type information. We plan do develop graphs expressing the communication and coupling between classes based on type information.

- **Uniform Accessors and Accesses.** The fact that a programmer can access directly an attribute or invoke an accessor method has an impact on the interpretation of some graphs. We would like to uniformly take into account the accessors as accesses.

6. Related work

Program visualisation. Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids. Various tools provide quite different program visualisations: Graphtrace [17], Rigi [24], Hy+ [5, 4], SeeSoft [2], ISVIS [15], Jinsight [26, 27], Duploc [8], Gaudi [28].

Powerful algorithms have been developed to support such visual program representations: the Sugiyama algorithm to optimise hierarchical layouts [32], hyperbolic geometry to navigate through large hierarchies [19], Shrimp views to optimise layouts in general [31], libraries providing ranges of algorithms [30], ternary diagrams to track dynamic interactions between system modules [12], mural techniques to provide large overviews [2], [15].

All of these have in common that they seek to visualise programs by applying quite sophisticated layout algorithms.

In some cases, these algorithms even have copyrights on them! In contrast, we seek to simplify layout algorithms and try to obtain added value by exploiting metrics.

Metrics. Metrics have long been studied as a way to assess the quality of large software systems [9] and recently this has been applied to object-oriented systems as well [14], [18], [23], [22], [25]. However, a simple measurement is not sufficient to assess such complex thing as software quality [13], not to mention the reliability of the results [6].

Some of the metric tools visualise information via typical algorithms for statistical data, such as histograms and Kiviat diagrams. Datrix [14], TAC++ [10, 11], and Crocodile [22] are tools that exhibit such visualisation features. However, in all these approaches, the visualisation is a mere side-effect of having a lot of numbers to analyse. In our approach, the visualisation is an inherent part of the approach, hence we do not visualise numbers but constructs as they occur in source code.

7. Conclusion

We presented a hybrid approach for the reverse engineering of object-oriented source code. It combines the immediate appeal of program visualisations with the scalability of metrics in order to aid in program understanding and problem detection.

By means of a case study, we have illustrated how our approach helps to get an initial overview of a software system and to focus on potential problems. We conclude that the combination of metrics with program visualisation is a promising approach, in particular because the possibility to rule out sophisticated layout algorithms may be countered by the combination with metrics.

In the near future, we will continue to expand CodeCrawler—the tool we have built to support our approach— to explore other layout algorithms and metrics. Next, we will exploit the language independent nature of the CodeCrawler platform to perform more case studies and gain more experience. Finally, we want to use our experience to develop a real reverse engineering method, based on sequences of program visualisations that have provided good results.

earlier drafts and who provide a splendid atmosphere to work in.

References

[1] A.S.Yeh, D. Harris, and M. Chase. Manipulating recovered software architecture views. In *ICSE'97 Proceedings (International Conference on Software Engineering)*. IEEE Computer Society, 1997.

[2] T. Ball and S. Erick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, Apr. 1996.

[3] E. Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 3(4):233–301, Jan. 1998.

[4] M. Consens and A. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *SIGMOD'93 Proceedings (International Conference on Management Data)*, pages 511 – 516. ACM Press, 1993.

[5] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *ICSE'92 Proceedings (International Conference on Software Engineering)*, pages 138 – 156. IEEE Computer Society, 1992.

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

[7] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal ? UML shortcomings for coping with round-trip engineering. In B. Rumpe, editor, *UML'99 Proceedings (The Second International Conference on The Unified Modeling Language)*, LNCS. Springer-Verlag, 1999.

[8] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM'99 Proceedings (International Conference on Software Maintenance)*. IEEE Computer Society, 1999.

[9] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.

[10] F. Fioravanti, P. Nesi, and S. Perli. Assessment of system evolution through characterization. In *ICSE'98 Proceedings (International Conference on Software Engineering)*. IEEE Computer Society, 1998.

[11] F. Fioravanti, P. Nesi, and S. Perli. A tool for process and product assessment of C++ applications. In *CSMR'98 Proceedings (Euromicro Conference on Software Maintenance and Reengineering)*. IEEE Computer Society, 1998.

[12] P. Haynes, T. Menzies, and R. Cohen. Visualisations of large object-oriented systems. In *Software Visualization*. World-Scientific, 1997.

[13] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[14] M. Jean and F. Coallier. System acquisition based on software product assessment. In *ICSE'96 Proceedings (International Conference on Software Engineering)*. IEEE Computer Society, 1996.

[15] D. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. In *WCRE'97 Proceedings (Working Conference on Reverse Engineering)*, pages 56 – 65. IEEE Computer Society, 1997.

[16] R. E. Johnson. Documenting frameworks using patterns. In *OOPSLA'92 Proceedings (Object-Oriented Programming Systems, Languages and Applications)*, pages 63 – 76. ACM Press, 1992.

[17] M. F. Kleyn and P. C. Gingrich. Graphtrace – understanding object-oriented systems using concurrently animated views. In *OOPSLA'88 Proceedings (Object-Oriented Programming Systems, Languages and Applications)*, pages 191–205. ACM Press, 1988.

[18] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *WCRE'97 Proceedings (Working Conference on Reverse Engineering)*, pages 44 – 54. IEEE Computer Society, 1997.

[19] J. Lamping, R. Rao, and P. Pirolli. A focus + context technique based on hyperbolic geometry for visualising large hierarchies. In *CHI'95 Proceedings (Computer Human Interaction)*. ACM Press, 1995.

[20] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *OOPSLA'95 Proceedings (Object-Oriented Programming Systems, Languages and Applications)*, pages 342 – 357. ACM Press, 1995.

[21] M. Lanza. Combining metrics and graphs for object oriented reverse engineering. Master's thesis, University of Bern, 1999.

[22] C. Lewerentz and F. Simon. A product metrics tool integrated into a software development environment. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543, pages 256 – 257. Springer-Verlag, 1998.

[23] R. Marinescu. Using object-oriented metrics for automatic design flaws in large scale systems. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543, pages 252–253. Springer-Verlag, 1998.

[24] H. Muller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[25] P. Nesi. Managing OO project better. *IEEE Software*, July 1988.

[26] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *OOPSLA'93 Proceedings (Object-Oriented Programming Systems, Languages and Applications)*, pages 326 – 337. ACM Press, 1993.

[27] W. D. Pauw and J. Vlissides. Visualizing object-oriented programs with jinsight. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543, pages 541 – 542. Springer-Verlag, 1998.

[28] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM'99 Proceedings (International Conference on Software Maintenance)*. IEEE Computer Society, 1999.

[29] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for smalltalk. *Journal of Theory and Practice of Object Systems (TAPOS)*, 3(4):253 – 263, 1997.

[30] G. Sander. Graph layout for applications in compiler construction. Technical report, Universitaet des Saarlandes, February 1996.

[31] M.-A. D. Storey and H. A. Mueller. Manipulating and documenting software structures using shrimp views. In *ICSM'95 Proceedings (International Conference on Software Maintenance)*. IEEE Computer Society, 1995.

[32] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on systems, man and cybernetics*, SMC-11(2), Feb. 1981.

[33] S. Tichelaar and S. Demeyer. An exchange model for reengineering tools. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543, pages 82 – 84. Springer-Verlag, 1998.

[34] www.raincode.com.