Chapter 18

CodeCrawler

Author(s): Michele Lanza and Stéphane Ducasse

CODECRAWLER is a tool that supports reverse engineering of large object-oriented projects. It combines the immediate appeal of visualisations with the scalability of metrics. Furthermore, it allows the user to tailor what information is presented as well as how it is presented.

18.1 Problem

The reverse engineering of large object-oriented legacy systems benefits greatly from an approach providing a fast overview and focusing on the problematic parts. Among the various approaches that exist today, two are particularly 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[1] [CONS 92], [KLEY 88], [LAMP 95], [MÜ 86], [PAUW 93], [JERD 97], [SAND 96], [STOR 95], [SUGI 81], [CROS 98], [BALL 96], [JERD 97]. Another is *metrics*, because metrics are known to scale up well [DEME 99a], [KONT 97], [LEWE 98a],[LORE 94],[MARI 98].

We propose an tool encompassing both graph visualisation and metrics combined in a simple approach where (a) the graph layout is very simple and (b) the extracted metrics are straight forward to compute. Indeed, our goal is to identify useful combinations of graphs and metrics that can be easily reproduceable by reverse engineers using a scriptable reengineering tool-set.

18.2 Principle and Tool

Combining Graph and Metrics. We enrich a simple graph like tree, boxes one besides the other, with metric information of the object-oriented entities it represents. In a two-dimensional graph we render up to five metrics on a single node at the same time. For a more in-depth discussion of this topic please read Section 2.2.4.

1. **Node Size**. The width and height of a node can render two measurements. This means 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 reflect two metric measurements. This requires the presence of an absolute origin within a fixed coordinate system. Note that not all layouts exploit this dimension.

[1] This is often phrased as "One picture conveys a thousand words".

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

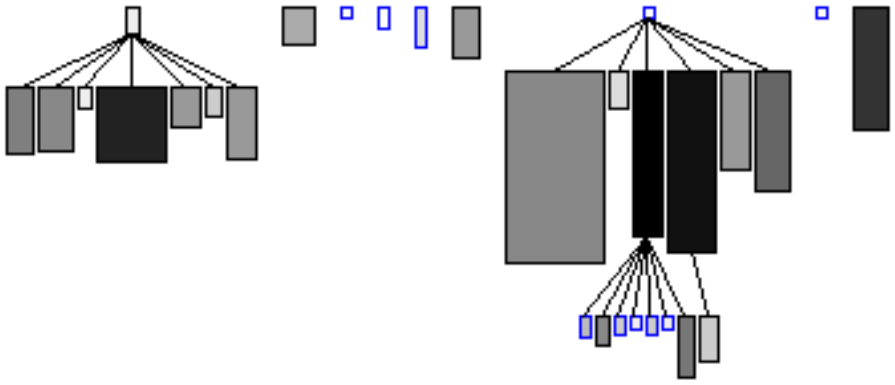

Figure 18.1: Inheritance Tree; node width = NIV, node height = NOM and colour = NCV.

Illustration. As an example Figure 18.1 shows an inheritance tree graph from CodeCrawler. The nodes represent the classes and 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. In this case the position of the nodes does not reflect metrics as the position is defined by the layout algorithm.

18.2.1 The Tool.

CODECRAWLER is an open platform providing a graphical representation of source code combined with object oriented metrics. Besides the pure combination of graph with metrics values, while building CODECRAWLER we were confronted with practical considerations such as the minimal size of a node or the size of the screen.

Graphical Considerations and Influences. For the node size, we chose to implement the mapping such as to really reflect the measurement in the size on the screen with a slight distortion in the case where 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 1 or 2 pixels wide is unecessarily difficult.

While the usage of different colours is a good way to attract the attention of the eye, the usage of too many colours should be avoided. It results in an optical overload that hinders rather than helps understanding. The solution with the colour tone has the advantage that numerical information can be transmitted by colours: 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 must notice that since the perception of a colour tone is less precise than the perception of size, the colour metric is only useful for the detection of extreme values.

Note that CODECRAWLER supports different distributions (linear, logarithmic...) to 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 exceed a certain threshold value.

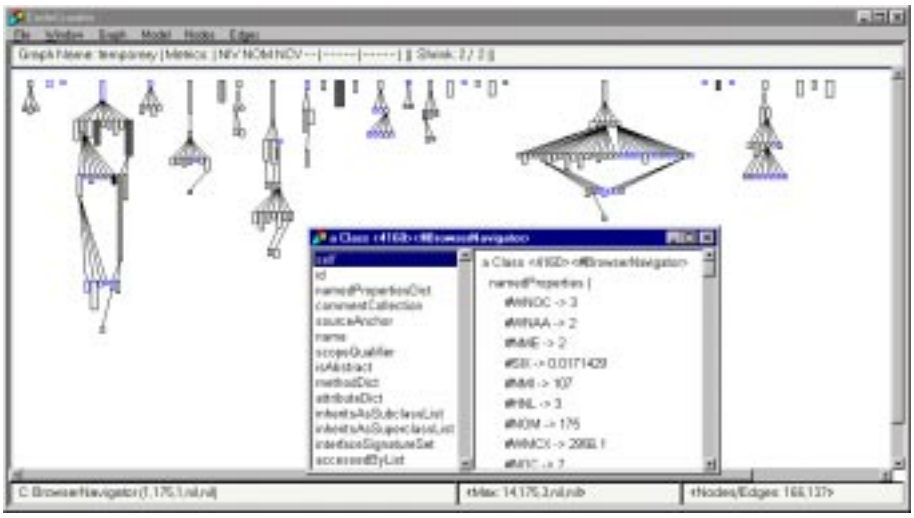

Figure 18.2: The CodeCrawler platform at work: (1) an inheritance tree with x node size = NIV, y node size = NOM, colour = NCV and (2) inspecting the data of a represented entity.

Supporting Reverse Engineering. Besides the definition of concrete graphs and operations on the graphs themselves like highlighting all the edges arriving at a specific node, following a particular edge, applying a new graph to a specific node . . . , CODECRAWLER provides a number of other features that greatly enhance reverse engineering activities. Most noteworthy are (1) the query of the graph to identify a node according to some criteria such as its name and (2) code navigation via the graph. 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 its metrics.

Moreover, as shown in Figure 18.2 CODECRAWLER displays the information of the current displayed graph (top border) and the information related to the entity under which is the mouse (bottom border). In Figure 18.2 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.

18.3 A Scenario

The scenario itself consists of 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.

The scenario conveys well how customisable and exploratory a hybrid approach can be. Indeed, the idea is that different graphs provides different yet complementary perspectives. Consequently, a concrete reverse engineering strategy is to apply the graphs in some order, although the exact order may vary depending on the kind of system at hand and the kind of questions driving the reverse engineering project.

The particular software system used for our scenario is the Refactoring Browser [ROBE 97a] 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 metaclasses), 2365 methods, 365 instance variables, 2198 instance variable accesses and 9780 method invocations. In that respect we consider it a small to medium case study.

Note that we have run other experiments on industrial case studies implemented with C++ and SMALLTALK . Unfortunately, due to non-disclosure agreements with the case study providers, we cannot publish the results of these studies here.

18.3.1 Understanding the Refactoring Browser

1. Class Size Overview: Checker 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 *checker 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 criterion (see Figure 18.3).

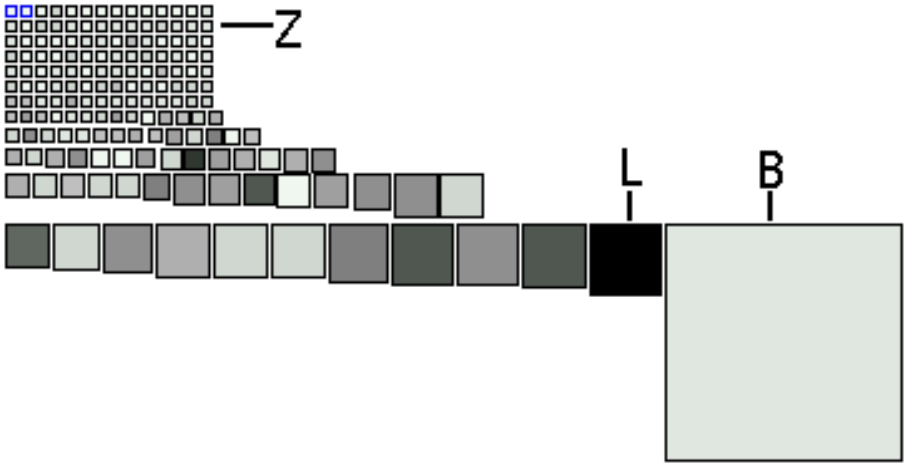

Figure 18.3: Class Size Overview via Checker Graph; node size = LOC, and colour = NIV.

Interpretation. The checker graph is useful for showing relative proportions between the system elements, in this particular case it shows the proportion 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 identified as B. The second biggest class with 441 lines of code is called BRScanner identified as L in Figure 18.3. 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 which are positioned on the upper left corner of Figure 18.3. (This last detail is only visible on the screen and not on the paper version, because metric measurements equal to zero make the nodes to be displayed with a blue border and this colour is not rendered in this outprint).

2. Inheritance Overview: Trees. To assess the size and complexity of the system, we request for an *inheritance tree*. We use as node size the number of instance variables and the number of methods, while the colour tone represents the lines of code of the classes (see Figure 18.4).

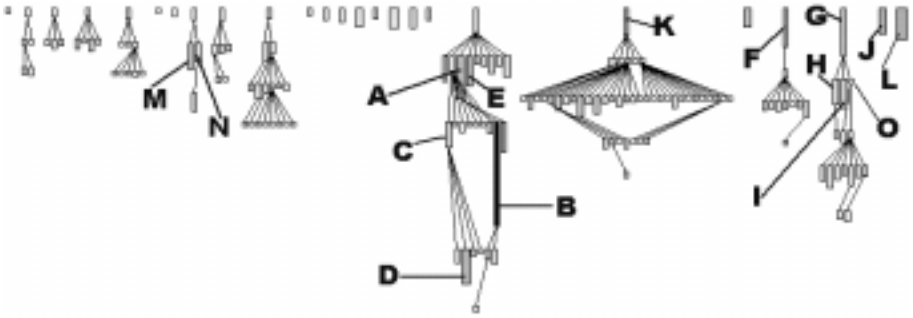

Figure 18.4: Inheritance Overview via Tree; node width = NIV, node height = NOM and colour = WLOC.

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 criterion we identify some candidate classes for further investigation: (1) the smallest class (O) is completely empty and (2) 12 classes have from 40 methods per class to 175 at the maximum (B).

These 12 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). Sibling classes like (H, I) and (N, M) are good candidates for refactoring analysis to see if some of the code could not be refactored up in their superclass.

An example of possible further investigation is the huge class called BrowserNavigator that implements 175 methods (named B in Figure 18.4) whereas its superclass Navigator (A) already implements 70 methods. Another interesting class without subclasses is the class called BRScanner (named L), that implements 49 methods and defines 14 instance variables.

3. Focus on Class Cohesion: Confrontation Graph. Given the analysis of Figure 18.4, we will focus on the class BRScanner (L). 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 access done by the method. The resulting graph is shown in Figure 18.5. The instance variables are the middle row of nodes and the methods are the top and bottom row of nodes.

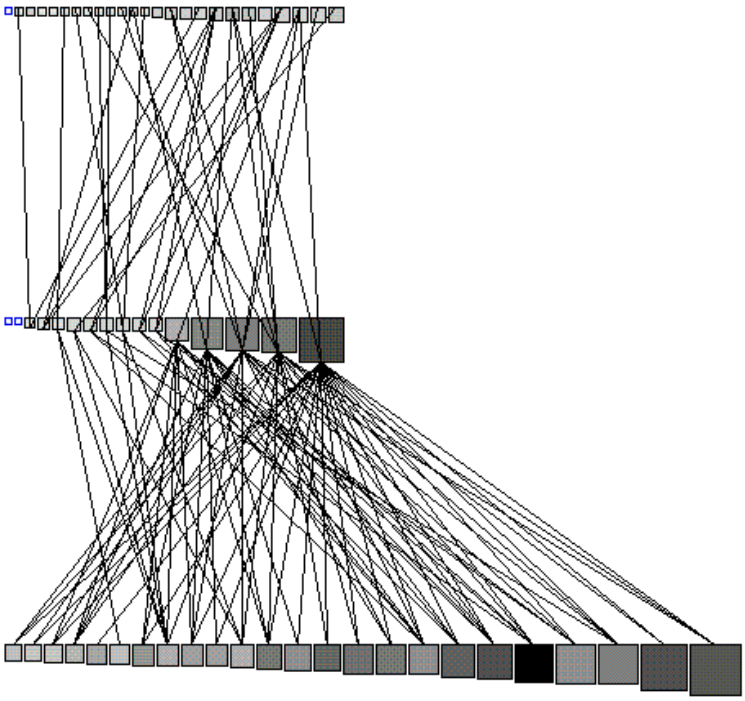

Figure 18.5: Focus on Class Cohesion of BRScanner via a Confrontation Graph; method node height and width = NOS and colour = LOC; attribute node height and width and colour = NAA.

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 making a split 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 same confrontation graph— discovered classes that can easily be split. Unfor-

tunately for our scenario, but good for the Refactoring Browser, we did not find such case in the case study described here.

18.3.2 Scenario evaluation.

Results. Using a hybrid reverse engineering approach combining metrics and graph layouts to reverse engineer the Refactoring Browser provides us with a quick understanding of the system without having to dive into the details of the system. Due to space limitation we could not show system of all graph layouts and metrics, nor could we include more useful graphs. Interested readers may consult [LANZ 99] for a more complete description.

- **Fast overview**. The *checker graph* provided us with a intuition for the proportions in terms of code size and helped us to quickly identify extreme cases. 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, CODECRAWLER helped us to understand the quality of the hierarchies: for example the refactoring hierarchy whose root is K in Figure 18.4 is composed by a high number of small classes whereas the abstract syntax tree hierarchy, whose root is G in Figure 18.4, is composed by more substantial classes.
- **Insight on inheritance quality**. The complementary perspectives on the inheritance tree provided 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 specialisation.
- **Identification of exceptional classes**. Even if the Refactoring Browser is quite well-designed we did identify some exceptional classes that would benefit from a refactoring.
- **Overview of the methods**. We quickly identified possible singular methods and got a first view of the overall method quality. Only a few outliers possessed overly high LOC counts.
- **Internal class coupling**. Using confrontation graphs, we were able to have a fast idea of the coupling between a class and identify clusters of instance variables.

18.4 Implementation Information

CODECRAWLER is developed within the VISUALWORKS SMALLTALK environment, relying on the HotDraw framework [JOHN 92] for its visualisation. Moreover, it uses the facilities provided by the VISUALWORKS environment for the SMALLTALK code parsing, whereas for other languages like C++ and Java it relies on Sniff+ to generate code representation coded using the FAMIX Model [TICH 98](see below).

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). But such a limit is not linked with the approach but with the libraries used for the implementation of CODECRAWLER and the available memory.

The Underlying Data Model. CODECRAWLER is based on MOOSE a language independent representation of object-oriented source code, based on FAMIX (FAMoos Information eXchange model, see [TICH 98]) and 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 18.6).

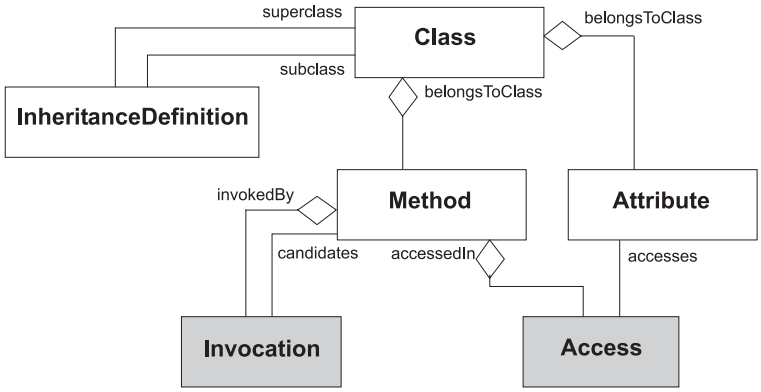

Figure 18.6: A simplified view of the FAMIX Data Model.

18.5 Contact Information

CODECRAWLER has been developed by Michele Lanza (lanza@iam.unibe.ch) during his master thesis at the University of Berne in the Software Composition Group. CODECRAWLER is based on the MOOSE language independent representation of object-oriented source code implemented in VISUALWORKS by Serge Demeyer and Stéphane Ducasse.
The authors can be contacted at {demeyer,ducasse,lanza}@iam.unibe.ch or http://www.iam.unibe.ch/scg.
CODECRAWLER is available at http:://www.iam.unibe.ch/∼scg/Archive/Software/CodeCrawler .