2.2 Program Visualisation and Metrics

Author: Michele Lanza

> *"Continuous visual displays allow users to assimilate information rapidly and to readily identify trends and anomalies. The essential idea is that visual representations can help make understanding software easier."*
> [BALL 96]

Although the object-oriented paradigm lets programmers work at higher levels of abstraction than procedural models, the tasks of understanding, debugging, and tuning large systems remain difficult. This has numerous causes: the dichotomy between the code structure as hierarchies of classes and the execution structure as networks of objects; the atomisation of functionality - small chunks of functionality dispersed across multiple classes; and the sheer numbers of classes and complexity of relationships in applications and frameworks. The fields of scientific visualisation and program visualisation have demonstrated repeatedly that the most effective way to present large volumes of data to users is with a continuous visual fashion [PAUW 93].

In this chapter we list some properties that a graphical representation of source code should possess to be useful for reverse engineering. We then see in what respect our approach fulfils those requirements and include a short scenario to explain our approach. We also list some problems concerning the visualisation of metrics, colors and issues concerning interactivity.

The central point of this chapter is to show how we merge the concepts of program visualisation, metrics and interactivity. These three aspects are the cornerstones of this work. The concepts that are explained here have been implemented in a single tool called CodeCrawler, which we present in the next chapter.

2.2.1 Graphs for Reverse Engineering

In this section we list some features that in our eyes graphs for reverse engineering should have. We emphasise that we use the term graph in a very broad sense: often we mean its picture or graphical representation on screen or on paper and not necessarily its scientific definition.

- *Simplicity and Quality*. The first important prerequisite is that the generated pictures of a graph have to be relatively simple and easy to grasp. The main reason for that is that too much displayed information overloads the viewer's perception. This tends to backfire and causes an unwanted information loss. A secondary aspect is that simple graphs are also easily reproducible, while complex techniques like hyperbolic trees [LAMP 95] are affected by a considerable complexity which is hard to grasp and reproduce. Many approaches have been discussed as to how a software entity could be represented for program visualisation ([BALL 96, PAUW 93, KLEY 88] to name but a few). We think that a graphical representation of an object oriented entity should be easy to grasp and not make use of a specific dictionary of shapes which has first to be learned. A graph should be able to transmit useful information to the viewer at first sight.
- *Quantity*. We have to be able to select how much of the subject system we want to display and at what level of granularity. Thus, we should be able to zoom in and out of such a graph and reduce the amount of displayed information at will.
- *Colors*. Program visualisation can be supported by colors, because they can attract the eye to interesting hot spots, while other parts of the graph which look less colorful can be ignored by the viewer. Colors have often been used in program visualisation [RIVA 98]. While colors are a good way to attract the attention of the eye, the usage of too many colors in a graph is not advised, since this results in an optical overload for the viewer of the graph. We also advise against the use of color conventions which have first to be learned by the viewer, as this lessens the impact of the colors.

- *Scalability*. As reverse engineering is especially crucial in very large systems, a visualisation should be scalable and work if possible at any level of granularity. The number of displayed entities should not affect the quality of the graph.
- *Interactivity*. A very important aspect of graphs is not only their layout algorithm but also that they can provide interactivity to the user through direct-manipulation interfaces. Making a static display of nodes and trying to extract information from the graph has clearly defined limits, which we discuss below in Section 2.2.2.
- *Metrics*. Although intangible in the physical sense, software *has* size. It can be measured, especially in object-oriented code we can assign numerical values (metric measurements) to its entities. Although the concept of software is abstract and often exists only in the head of the programmer, we can measure it. Once we can measure it, we can assign a size to it and represent this size graphically. We think that metrics enrich the semantic value of a graphical representation of a software entity, and discuss this below in Section 2.2.4.

2.2.2 Interactivity

A graph which lacks interactivity has certain drawbacks:

1. The user can't produce new views starting from a part of the graph.
2. The user can't find out secondary information (e.g. he can't inspect the nodes or browse through their source code).
3. The user can't reduce the amount of displayed data by either removing nodes by hand or by filtering out nodes through algorithms.

Those limits can be overridden if the graph is interactive:

- If we produce a view on a system and one particular node is drawing our attention, we'd like to know more about this node and the entity that it is representing. So we should be able to know its name, to have a look at its properties, to zoom in into the node, to have a list of all nodes that have a relationship with this node, or even to have a look at the source code behind the node (suppose the node is a method).
- Starting from a part of the graph or from one single node we'd like to be able to generate new views without having to go through the whole graph generation procedure again. The viewer should be able to 'navigate' around the code travelling from one point of interest to the next.
- Sometimes the relationship edges in a graph make the whole graph look like a cobweb. We should be able to switch off edges and switch them on again on demand depending on nodes we selected, etc.
- Suppose we have displayed a graph with a lot of nodes and edges. One particular node is of interest to us. But since there are too many edges in the graph it's hard to see how many times and to which other nodes the node in question is connected. So the graph should also be able to provide a 'highlighting feature' where we can display on top of all edges and nodes the connections of the node in question. It is important to note here that compared to the previous point we don't want to reduce the complexity of the displayed graph. We just want to have a better view on it.

It is an important point we are stating: The interactivity of a graph is *not just a nice feature* but one of its *most important aspects*.

2.2.3 The Use of Layout Algorithms

Perhaps the most difficult aspect of showing software through graphs involves the graph layout problem. The nodes and edges of the graph must be positioned in a pleasing and informative layout that clearly shows the underlying graph's structure. Many techniques have been proposed for laying out arbitrary graphs. Unfortunately, in practice, drawing informative graphs is exceedingly difficult, particularly for large systems. The resulting graphs, even when drawn carefully, are often too busy and cluttered to interpret [BALL 96].

The opposite case can also be true: sometimes elaborate layout algorithms can't ameliorate the user's perception or can do that only at the cost of algorithm complexity: There are various (and sometimes very complex) techniques to display a tree graph, but in the end it's still just a tree.

However, we don't want to minimise the importance of complex layout algorithms, on the contrary: we believe they could bring many more benefits than drawbacks. Good layout algorithms just were not part of the constraints of this work. But it is certainly a very promising field of research in this context. The layout algorithms used in this work are discussed in detail in Section 2.2.6.

2.2.4 The Use of Metrics in Graphs

In [BALL 96] the following statement is made: "Software is intangible, having no physical shape or size. Software visualisation tools use graphical techniques to make software visible by displaying programs, program artifacts, and program behaviour."

It is obvious that everything regarding metrics possible through their graphical display is also possible by just calculating and analysing the metric measurements. So the question arises why we should have a graphical display of them, since the information sought is in the metrics themselves. But in the same way one could think to listen to music by just reading the partiture of a song instead of using the sense normally designed for that (the hearing)[4]. *What changes is the perception and the impact of what is perceived.*

Our Idea. The whole concept is fairly easy: we map metric measurements of software entities on their graphical representation on the screen. As we said before we chose the entities to be represented by rectangles. Rectangles have a certain width and a certain height. They can be filled with a color. Their position can also bear a certain amount of information.

With this approach, in a two-dimensional graph consisting of nodes and eventually edges between the nodes, up to five metrics can be assigned to a node and rendered visually at the same time. These are:

1. The X coordinate of the position
2. The Y coordinate of the position
3. The width
4. The height
5. The color shade

This concept is rendered clearly in Figure 2.3, where we see where the metrics can be applied on a node.

Not every graph can make use of five the metrics at the same time. In a graph that does not have an origin (which defines an absolute coordinate system) it makes no sense using two metrics for the position of the

[4] A short comment on perception: the size of software can be seen through other means: if we scroll through the source code of a very large class, we probably have to either move the mouse or press some keys on the keyboard to scroll on. This physical act of scrolling can also transmit size and complexity to the viewer.

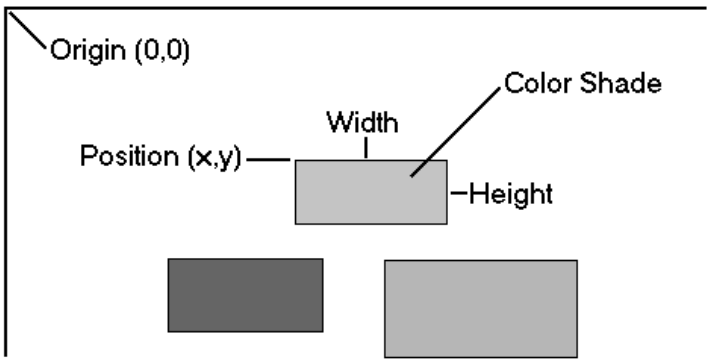

Figure 2.3: An example of nodes and their possible metrics.

nodes. A good example for such a graph is a tree graph, where the position on the nodes is implicitly defined by the logical position of the nodes in the tree. Another property which came up during our experiments was that sometimes the multiple use of the same metric (for example if we choose the same metric to reflect width and height) can emphasise certain parts of the graph and render them more clearly for the viewer.

2.2.5 A Concrete Graph Specification.

In our approach a *concrete graph*, this means the resulting displayed graph, is the combination of four factors :

1. **The Graph Type**. Its purpose is to render a certain aspect of a system: a tree graph is good for displaying hierarchical information, a circle for communication, a confrontation graph for dependencies, etc.

2. **The Layout Algorithm**. Starting from the original idea of the graph, variations refine the concrete display. The layout takes into account the following issues:

   - Display concerns (i.e. the fact that the complete graph should or not: fit into the screen, minimise the space used, sort the nodes according to certain criteria, etc.).
   - The entities and their relationships. This implies the choice of the represented entities (class, attribute and/or method) to be rendered as graphic elements and the logical link between the graphical elements and the metrics. For example in some graphs the position of the nodes reflects the size of the entities whereas in others this is the size of the node.

3. **The Metric Selection**. Once the layout algorithm stands, metrics are associated to the graph. This application depends on the specification of the previous step.

4. **The Interaction**. Since the goal of a graph is to support the reverse engineering of the application, the interaction that a user can perform should be specified. All the graphs support basic navigation functionality which allows one to access code elements. However, the interaction is refined for specific graphs, for example to walk through it, to highlight the edges, to zoom in/out, etc.

2.2.6 A Short Example

To make the whole idea of visualising software structures with the help of metrics a bit more understandable we included here a short example of our approach.

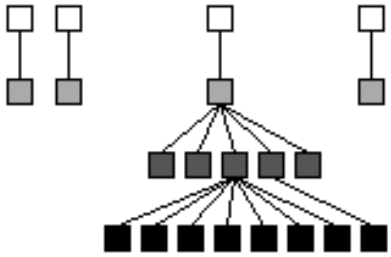

Figure 2.4: A simple inheritance tree.

Suppose we want to understand the inheritance hierarchy of a small system. The idea that comes up is to display the graph as a tree. The nodes in tree represent classes, the edges represent inheritance relationships.

The layout algorithm for displaying a tree is arbitrary, for reason of simplicity we chose a very simple one, which sometimes can make edges cross nodes, but it renders the whole concept nonetheless. Keep in mind that this layout part can also make use of very complicated algorithms for space optimisations, edge crossing reduction, etc.

Once we have displayed the tree as we see in Figure 2.4 we apply size and color metrics to the nodes. The use of position metrics is not possible here, as the position of the nodes is intrinsically defined by their logical position in the tree. As the nodes represent classes, we use class size metrics. The width and height of the nodes render the number of methods (NOM) while the color renders the number of attributes (instance variables).

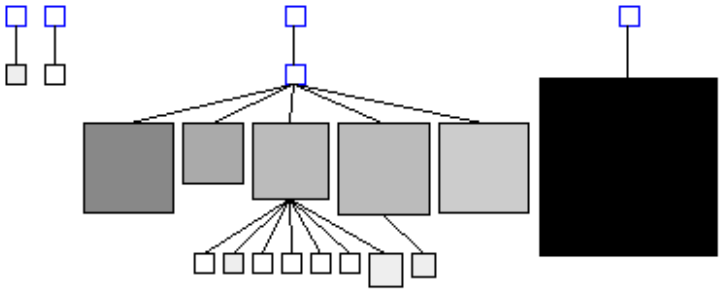

Figure 2.5: An inheritance tree that makes use of size and color metrics.

Once the tree is rendered as in Figure 2.5 we can start interacting with the graph. We can freely move nodes around, delete them, inspect them (i.e. browse the underlying classes), filter out parts, etc.

In fact, if we left out the interactive part, the amount of useful information that we could extract would be limited to the display in Figure 2.5.

Useful Graphs

This section is dedicated to the graphs which prove to be useful when it comes to the understanding of software systems and the detection of design problems using the approach discussed in this work. Although this may seem a little confusing, what in this chapter is called a 'useful graph' is not only its layout, but primarily the *combination of a layout with object-oriented metrics and the consequent extraction of information made by the viewer through interaction with the graph*.

The following is structured as follow:

- **Graph Structure.** Section 2.2.7 presents the structure we adopted to describe a useful graph. Every useful graph is presented using this format.
- **Case Studies.** Section 2.2.8 is a short presentation of the two case studies we chose to apply each useful graph on.
- **Layout Algorithms.** Section 2.2.9 presents the layout algorithms we selected.
- **Useful Graphs.** In section 2.2.10 we present the useful graphs divided into the 4 distinct groups: *class*, *method*, *attribute* and *class internal*. The names indicate which kind of entities are displayed in the graphs. Class internal treats the special case where methods and attributes are displayed at the same time.

2.2.7 Graph Structure

For each graph which we treat in this chapter, we discuss the following properties:

Graph: Indicates what type of graph and layout has to be chosen, and whether a sorting of the nodes has to precede the display.

Scope: At what granularity level the graph can be applied. We differentiate between *full system*, *subsystem* and *single class*. Sometimes the subsystems are indicated as a single inheritance hierarchy. We also indicate if the graph is language specific.

Metrics: We list five metrics in the following order: width metric, height metric, color metric, horizontal position metric, vertical position metrics. When we write a dash (-), this means that the metric should not be set. In case we write an asterisk (*) this means that the metric can be set freely. In the case of class internal graphs we repeat the five metrics twice, once for the method nodes and once for the attribute nodes.

General idea: We write what the graph is all about and what ideas lie underneath it. We also indicate what the user should be searching for in the graph.

Results: Here we present the results obtained after applying the graph on our case studies.

Possible Alternatives: We list a few alterations that could be made regarding the metrics, so as to obtain slightly different graphs, and list also some eventual interactions that could be applied on the graph to increase its usefulness.

Evaluation: Some statements about the advantages and drawbacks of the graph.

Application	Refactoring Browser	Duploc
Classes	166	123
Methods	2365	2382
Attributes	365	386

Table 2.1: An overview of the size of our case studies.

2.2.8 Case Studies

This section contains a short overview of the systems we used as case studies for this work. Basically we use them to test the graphs listed in the remainder of this chapter. We chose these two case studies for the following reasons:

- Availability. Both case studies are public domain and can be downloaded freely. With this point we can ensure that the results are reproducible.
- Size. We chose two case studies which can be termed as being of an *average size* and are representative of medium-sized standalone applications. We think that very small applications can't reflect results properly because the purpose of most graphs is coping with complexity, which in such cases is not necessary. On the other hand, if we had chosen very big applications, it would have been hard to present results in a concise manner, because many graphs can be applied in various areas and at various levels of granularity.
- Level of maturity. We chose one very mature application which has gone through some refactorings and redesigns, and another one which has been developed in a rush and which has yet to undergo its first redesign. We did this to see if the results of our experiments would differ and in what way they would do that.

Refactoring Browser. The Refactoring Browser is a widely used tool for the implementation of Smalltalk programs [ROBE 97a]. We took it as a case study because it is an application which has gone through several refactorings and redesigns and has been written by some very experienced programmers. This quality of implementation should thus be visible in such a system. It is a medium sized application as we can see in table 2.1.

Duploc. Duploc is a tool for the detection of duplicated code [RIEG 98]. Duploc was the first application written in Smalltalk by its developer, Matthias Rieger and has yet to undergo its first major redesign. Thus we expect it to have some of the flaws which new systems tend have, like oversized classes and methods, obsolete attributes, etc.

2.2.9 Graphs

This section is dedicated to the graphs and layouts we have selected to implement in CodeCrawler. We discuss the properties, advantages and drawbacks of each one of them. We include this here because they are mentioned throughout the remainder of this chapter.

We discuss the original idea of a graph and the scope of its applicability. Each graph has at least one possible kind of layout and we discuss it with a regard for the metrics that can be applied for that special layout. Sometimes a sorting of the nodes has an influence on the usefulness of a graph and we discuss that as well as the general pros and contras for each graph.

In Table 2.2 we have an overview of all graphs and their properties supported by CodeCrawler.

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

Table 2.2: CodeCrawler's graph layouts.

The 'Metrics' column specifies how many metrics can be rendered by the graph. 5 means that the a single node can render 5 metrics at the same time. $3 + 3$ means that two separate groups of entities and metrics can be defined. The 'Entities' column refers to the kind of entities the graph can be applied upon: C for class, M for method and A for attribute[5]. The 'Scope' column specifies if the graph can be applied to the complete (sub)system or only to some entities like a class or a method. The 'Sort' column indicates if a sorting of the nodes according to a certain metric measurement can enhance the usefulness of the graph in question.

[5]The limitation to these three types of entity is due to the current implementation of the Moose model. Future implementations of it may include supplemental entities.

2.2.9.1 The Tree Graph

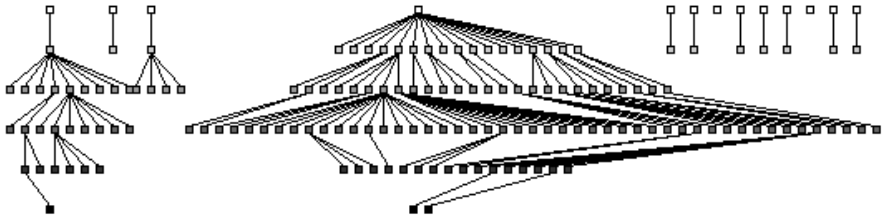

Figure 2.6: A tree graph of a system.

Overall Idea. A tree graph is useful for the display of hierarchical structures like inheritance hierarchies containing classes. The nodes represent classes, while the edges between the nodes represent inheritance relationships.

Scope. The scope of this graph ranges from very large systems to subsystems consisting of few classes. A requirement is that there is some usage of inheritance in the system. Otherwise the graph gets very flat and wide.

Layouts. We implemented three slightly different layout algorithms, which we simply called left, centered and right. Each one of them is based on recursion.

Metrics. The number of possible metrics that can be applied is 3. The two position metrics cannot be used, as the position of the nodes is defined by the layout algorithm. However, a virtual fourth metric is present, HNL. It is rendered by the layout algorithm through the vertical position of the nodes.

Sort influence. This graph is one of the few cases where a sorting of the nodes is not advised, as it disturbs the recursive layout algorithm.

Pro et contra. The advantage of this graph is that it can render a complex system in a very simple manner. Its only drawback is that because the position of the nodes is defined by the layout algorithm, this graph tends to get very large for big systems and will sometimes not fit on one single screen. The use of node shrinking can alleviate this problem.

2.2.9.2 The Correlation Graph

Figure 2.7: A correlation graph of method nodes using LOC and NOS as position metrics.

Overall Idea. This graph can render the relationship between two metrics when they are applied to entities. The two metrics are directly mapped onto the position coordinates of the nodes. This graph needs an absolute origin within a coordinate system, which in our case is the upper left corner of the graph. If the chosen metrics are in close relation to each other, the nodes are positioned along a certain correlation axis, which is defined by the metrics. If a node finds itself far away from this correlation axis, it means that its metric measurements are somehow abnormal compared to the other nodes and should be inspected. Very large measurements put a node far away from the origin, if one of the two position metric measurements is very small, the node finds itself near the left or top border of the graph.

Scope. This graph can be applied to any type of entity. The maximum number of displayable nodes is very big, as the expansion of the graph drawing depends on the outliers in the system and not on the number of displayed nodes. This involves an overlapping of nodes, which however is not negative, because we are mainly interested in the outliers (i.e. the extreme values).

Layouts. There is only one possible layout in this case, which directly maps the position metrics to the position of the nodes.

Metrics. The number of possible metrics that can be applied is 5. Indeed, each metric can be applied in this case. However, if we choose to select size metrics this involves that the nodes overlap, while without size metrics the nodes will either be positioned next to each other or cover up other nodes entirely. The overlap problem is especially acute when the chosen size metrics tend to have big values, like LOC.

Sort influence. A sort has no influence on the layout.

Pro et contra. The main advantage of this graph is its scalability. Another advantage is that we can pick out the outliers at one glance. The drawback is a certain loss of overview, because the nodes overlap. However, as we often do not make use of size metrics for this graph, we can circumnavigate this problem.

2.2.9.3 The Histogram

Figure 2.8: A horizontal histogram.

Figure 2.9: A horizontal histogram using the size addition layout

Overall Idea. A histogram provides a representation of the distribution of entities related to a certain metric. The distribution of the nodes can in turn give us general information about a system. For example if we use as vertical position metric LOC of methods, we are able to gather if the methods tend to be overlong or not, and if there are any significant outliers.

Scope. This graph can be applied to any type of entity, class, method or attribute. The number of displayable nodes is also very large. However, since a large part of the nodes distribute around a certain value, a few of the rows of this graph can get very large and eventually get bigger than the screen. This problem is sometimes acute if we use the size addition layout described below. One of the fields where its use is advised, is to make a distribution of the methods of single classes or of attributes of subsystems.

Layouts. There are two possible layouts. The first, called *horizontal*, ignores size metrics and displays every node with the same size. The second one, called *size addition*, makes use of the width metric, and puts the nodes next to each other, while taking their size in consideration. Only the horizontal layout can be considered to be a real histogram, the kind which is used in the field of statistics.

Metrics. The number of possible metrics depends on the used layout. The horizontal layout can make use of 2 metrics, namely the color and the vertical position.The size addition layout can also make use of the width metric.

Sort influence. In the case of the horizontal layout, a sort has a positive effect if we take the color metric as sort criterion. It makes the detection of color metric outliers easier. In the case of the size addition layout, a sort according to the width metric also has some positive effect for the detection of width metric outliers.

Pro et contra. This graph shows a good behaviour in terms of scalability. Its major drawback is that the vertical position metric needs to have a rather large measurement interval, otherwise the nodes will be distributed all near the same vertical position.

2.2.9.4 The Checker Graph

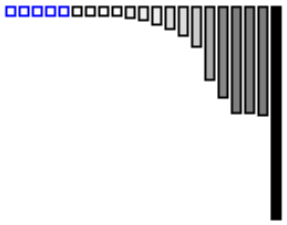

Figure 2.10: A checker graph using a sorted horizontal layout.

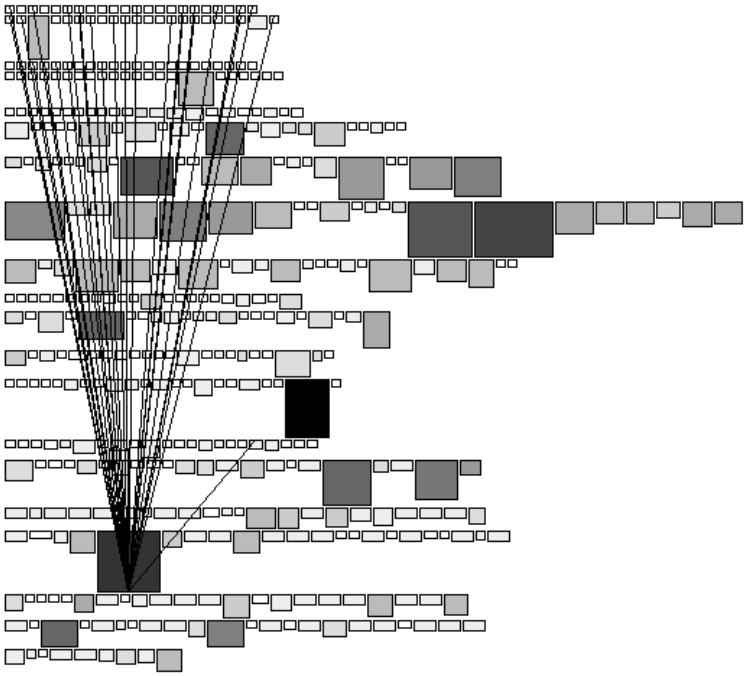

Figure 2.11: A checker graph using a quadratic layout with method nodes and invocation edges.

Overall Idea. The base idea for this kind of graph is simplicity. We want to lay out nodes without a special algorithm, we just place them one next to each other, to prevent them from overlapping.

Scope. This graph scales up quite well (especially if node shrinking is applied). Therefore it can be used for any kind on entity. However, it's not advisable to use edges in this graph, because it looks very chaotic, as they will cross the nodes.

Layouts. The first layout kind is called *horizontal* and *vertical*. We just place the nodes next to each other. We see such a layout in Figure 2.10[6]. Because this wastes a lot of space, we introduced the *quadratic* layout which tries to lay out the nodes to make them form a rectangle, whose width is dependent of the number of

[6]This figure suggests that a histogram is a special case of a checker graph. This is not true: a histogram makes use of a more complex layout algorithm which makes use of position metrics, as we see in the following subsections.

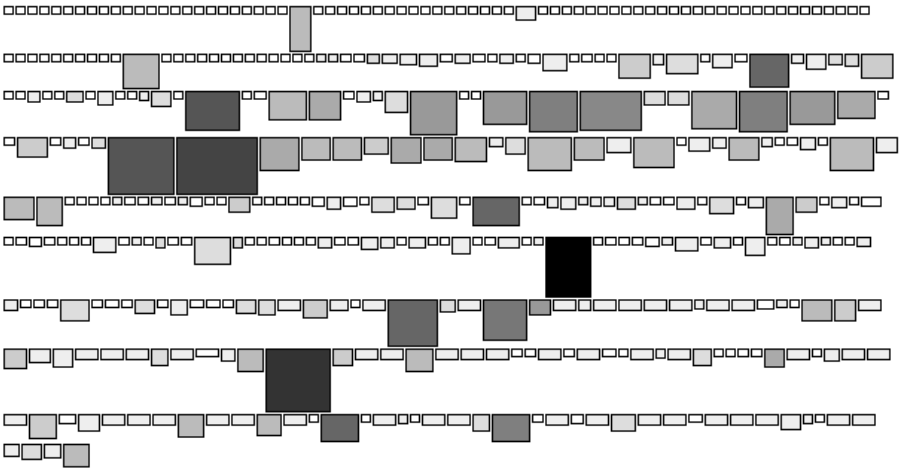

Figure 2.12: A checker graph using a maximal space usage layout.

displayed nodes. The graph which makes the best use of space is called *maximal space usage*, which tries to put as many nodes on the visible part of the drawing as possible.

Metrics. As the position metrics can't be used in this graph, we can only use size and color metrics.

Sort influence. The sort is essential for this graph. Indeed, if we don't make use of it, the nodes are placed randomly on the screen and it will be very hard to discern significant nodes. If we do make use of a sort according to a metric (especially the width metric), the detection of outliers will be very easy.

Pro et contra. The advantage is that we end up with a very easy to analyse layout. If the nodes are sorted, the detection of outliers is very easy, and the detection of suspicious node shapes is easy as well. This graph scales up well and several hundreds of nodes can be displayed at the same time without overlapping.

2.2.9.5 The Stapled Graph

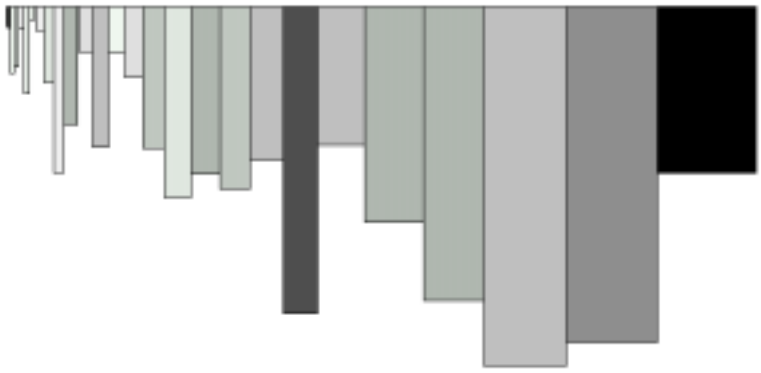

Figure 2.13: A stapled graph of class nodes.

Overall Idea. The idea for this graph came up when we tried to cure a small flaw in the horizontal checker layout: The width of the whole graph is defined by the summed widths of the nodes and cannot be influenced by the user. In such cases it often happens that the checker graph is wider than the screen. The stapled graph is thus a derivate: the user can indicate the maximum width of the graph he'd like to have, and all the node are accordingly shrunk in their width to make the graph fit the indicated space.

Scope. This graph can also display any kind of entity.

Layouts. A this time there is only one possible layout, which displays the nodes horizontally.

Metrics. The size and color metrics can be used, while this is not possible for the position metrics.

Sort influence. The sorting of nodes is essential for this graph to get some meaningful results. In fact it can be used for the detection of outliers regarding the height metric, if the nodes are sorted according to the width metric. If the two metrics are in close relation we often get a "staircase effect" because the nodes tend to get equally bigger in width and height. If this is not the case, the staircase effect breaks and we'll be able to easily detect those cases.

Pro et contra. One major drawback is that the width of a node will not directly reflect its metric, because it's being distorted by the graph width mapping function. Another drawback is that if the summed undistorted node widths of all nodes is bigger than the desired graph width, the nodes are shrunk in their width (otherwise they will be enlarged). If this shrinking is heavy, many small nodes will somehow disappear because they get very narrow, often only one pixel wide. The pro is obviously the intuitive detection of abnormal nodes which *don't* have to be outliers, but which stand out because two normally related metrics are not closely related in their case. Another pro is also that the graph will always fit the screen.

2.2.9.6 The Confrontation Graph

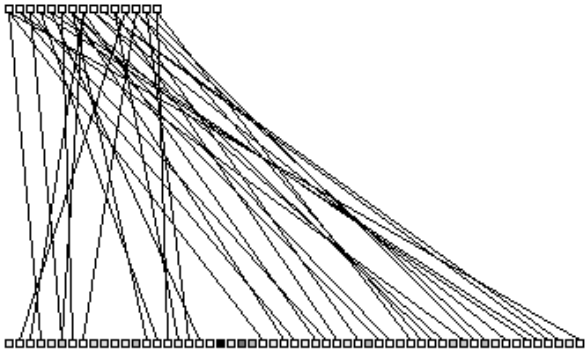

Figure 2.14: A confrontation graph using an horizontal layout

Overall Idea. This graph grew out of the necessity to display the access relationships between methods and attributes. An access is the only type of relationship between two entities of a different type.

Scope. This graph can only be applied on methods and attributes at the same time with accesses as edges. It's best used with the methods and attributes of one class.

Layouts. There are two possible displays. The first, called either *horizontal* or *vertical* displays on one row (column) the attributes and on the other one the methods. We can see such a layout in Figure 2.14. However, since in a class often the number of methods is much greater than the number of attributes, and the graph very soon gets larger than the screen, we introduced the *three row* layout. In this case the attributes are in the middle row, while the methods are in the upper and lower row.

Metrics. The size and color metrics can be used, while this is not possible for the position metrics.

Sort influence. A sort is advised for this graph. In the case of the method nodes it's especially useful according to the metrics LOC, NOS and NMAA. In case of the attribute nodes it's best to use NAA. If such a sort is applied, the number of edge crossings tends to drop and makes the graph look less cluttered.

Pro et contra. The major contra for this graph is that there is no special ordering of the nodes like clustering, except for a possible sort. However, it's the best graph to look at the internals of a class.

2.2.10 Useful Graphs: Class Graphs

In this section we list all graphs which display class nodes. We have noticed that the following graphs can be separated in two distinct groups. The graphs in the second group are normally applied after those in the first group, because they address more precise issues. We distinguish the following groups:

1. Those which serve primarily for system understanding. They work at a higher abstraction level, and in some cases can only return a general statement about the system. Problem detection is secondary in such graphs and in some cases not even possible. The following graphs fall under this category:
   - SYSTEM COMPLEXITY, Section 2.2.10.1.
   - SYSTEM HOT SPOTS, Section 2.2.10.2.
   - WEIGHT DISTRIBUTION, Section 2.2.10.3.
   - ROOT CLASS DETECTION, Section 2.2.10.4.
2. Those which primarily address problem detection, and secondarily program understanding. They must be applied on subsystems, rather than full systems. We list the following:
   - SERVICE CLASS DETECTION, Section 2.2.10.5.
   - COHESION OVERVIEW, Section 2.2.10.6.
   - SPINOFF HIERARCHY, Section 2.2.10.7.
   - INHERITANCE IMPACT, Section 2.2.10.8.
   - INTERMEDIATE ABSTRACT, Section 2.2.10.9.

2.2.10.1 System Complexity

Graph	Inheritance tree, without sort.	
Scope	Full system.	
Metrics		
Size	NIV (number of instance variables)	NOM (number of methods)
Color	WLOC (lines of code)	
Position	-	-

General Idea: This is one of the first graphs that should be applied to a system. It is an overview of the inheritance hierarchies of a whole system. This graph can give clues on the complexity and structure of the system (how many classes are present?), as well as information on the use of inheritance in the system (how deep do the hierarchies go and is the system in general flat or deep?). If we furthermore apply some class complexity metrics we can extract some more information. In this case we use as size metrics NIV and NOM, while for the color we choose WLOC. The detection of aberrant classes is now made easy: we can see if there are *very large classes*, *small classes* or even *empty classes*.

Results with the Refactoring Browser: In Figure 2.15 we see the SYSTEM COMPLEXITY graph applied on the Refactoring Browser. It shows few stand-alone classes and a few deep hierarchies. The first thing that strikes the eye is the class *BrowserNavigator* (A) which has a huge number of methods (175) and lines of code (1495) compared to the other classes present in the system. At the same time it only has one instance variable (this is the reason for its very narrow look). It may be a case for refactoring. If we take a look at the inheritance tree on the right side we can spot the class *BRStatementNode* (B) which is completely empty. When I asked the developers of the Refactoring Browser about this case, they told me that they were aware of the problem and that this class had been created to duplicate a hierarchy of another program. The same case can be spotted on one of the stand-alone classes *RefactoringError* (D) which is also empty. The next point of interest is the class *BRScanner* (C) which has the most instance variables (14) while it implements comparatively few methods (52). Perhaps this massive stand-alone class could be split up into subclasses. Another thing we can see is, that in the inheritance hierarchy in the middle of the graph, the root class *Refactoring* (E) is implementing by far the most methods, while there are quite a few very small classes deeper down the inheritance chain.

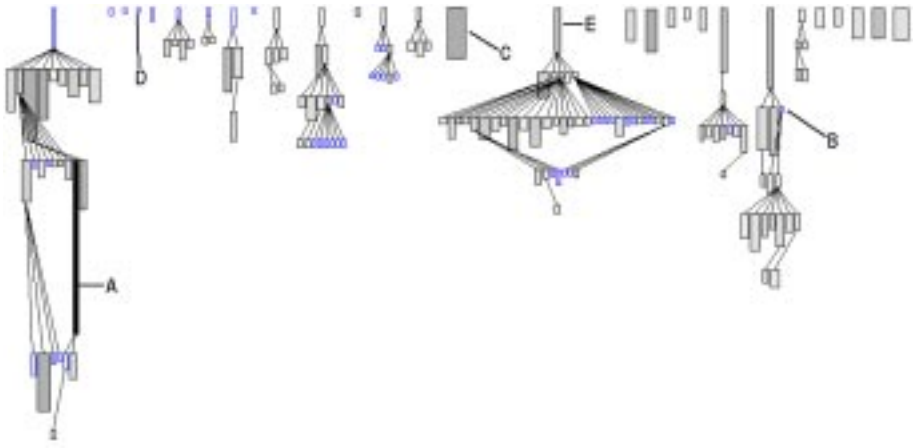

Figure 2.15: The system complexity graph applied on the Refactoring Browser using as size metrics NIV and NOM, and as color metric WLOC.

Results with Duploc: When we apply the SYSTEM COMPLEXITY graph on Duploc, we can spot the following in Figure 2.16: The system shows some very flat inheritance hierarchies, with many stand-alone classes which can have considerable sizes. This could mean that the system has not yet been refactored. There are three deep hierarchies, although in all three we can see that the main work is being done by the roots, which indicates top-heavy hierarchies. We also see that the main class *called DuplocApplication* (A) is very large and has only one very small subclass called it DuplocInformationMural[7]. Although *DuplocApplication* has the most methods and has the second most instance variables, the class with the most lines of code is *FastSparseCMatrix* (B). This class has only half the number of methods of *DuplocApplication* (74 vs. 130) but has nearly twice as much lines of code (1641 vs. 1060). Because of this we can already deduce that *FastSparseCMatrix* has some very long methods. The third point of interest are the classes on the left side (C): all of them are empty. These classes have become empty after being exported from the ENVY environment. The fourth eye-catch is the class *BinValueColoringModel* (D) on the right side. This class has the most instance variables (20), but only 52 methods. This may indicate that it is a service class which implements a lot of accessor methods. This supposition is being enforced by the light color value which is a sign for few lines of code (402), and is confirmed when we browse the source code of this class.

Figure 2.16: The system complexity graph applied on Duploc using as size metrics NIV and NOM, and as color metric WLOC.

Possible Alternatives: The color metric can be varied at will, especially class complexity metrics like NCV (number of class variables) prove to be useful.

Evaluation: This is certainly one of the first graphs that should be applied to a system, as it can return information on the structure and complexity of the subject system. However, it suffers one small drawback, which shows in very large systems: Sometimes the number of classes we want to display is so large that this graph takes several screens of place. It is difficult then to discern the outliers in the systems at one glance. The system hot spots graph discussed in Section 2.2.10.2 can counter this problem.

[7]The InformationMural is a subapplication of Duploc included in a latter phase of development. Evidently the developer did not want to write an own main application class from scratch, but preferred to take the existing one, subclass it and override only some needed methods. This explains the small size of this class.

2.2.10.2 System Hot spots

Graph	Checker, quadratic, sort according to width metric.	
Scope	Full system.	
Metrics		
Size	NOM (number of methods)	NIV (number of instance variables)
Color	WLOC (lines of code)	
Position	-	-

General Idea: For very large systems it's hard to decide where to start looking for problems hot spots. One general rule is to look for very large or complex classes regarding their number of attributes and methods. The graph described here is a very simple display of all classes in the system sorted according to a certain metric. The nodes are placed next to each other to prevent overlapping. This graph detects outliers very easily because of the sorting. We distinguish the following:

- Large nodes at the bottom of the graph. These represent the biggest classes in the system.
- Small nodes at the top of the graph. These are the smallest classes which can sometimes even be empty.
- Very flat nodes. These nodes possess very few (if any) instance variables.
- Rather high nodes. This is seldom the case, as classes rarely have many attributes. Sometimes we can detect configuration classes like this.

Results with the Refactoring Browser: In Figure 2.17 we get a HOT SPOTS view on the Refactoring Browser. While in Figure 2.15 we had to search for the biggest and smallest nodes, this is made easy in this kind of graph because the nodes have been sorted: as before we can locate the class *BrowserNavigator* (A) and *BRScanner* (B). The sorting of the nodes makes it easy now to detect empty or very small classes, which find themselves at the top of the graph (D). Our attention is now also drawn to other classes like *BrowserApplicationModel* (C), which implements 38 methods while it defines no instance variable, which is visible by its flat shape. The view on the shape of the nodes is also facilitated, we can now detect classes like *MoveVariableDefinitionRefactoring* (E), which defines 6 instance variables while it implements only 7 methods (mainly accessors), giving it nearly a square shape.

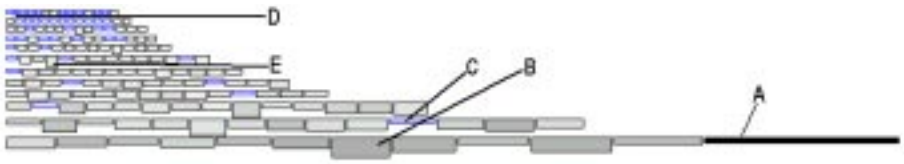

Figure 2.17: The system hot spots graph applied on the Refactoring Browser using as size metrics NOM and NIV, and as color metric WLOC. The nodes have been sorted according to NOM.

Results with Duploc: The HOT POTS view on Duploc reveals also some information which could not be seen at first sight in Figure 2.16, as we see in Figure 2.18. We see Duploc has either very large classes (A)(B), or very small ones (D). We can also locate some classes with many instance variables (C). Two classes which could be interesting for further investigation because of are *DuplocCodeReader* (F) (32 methods, 17 instance variables) and *DuplocProgressMeter* (E) (15 methods, 9 instance variables): both classes have many instance variables and few methods, which could indicate service classes apt for refactoring.

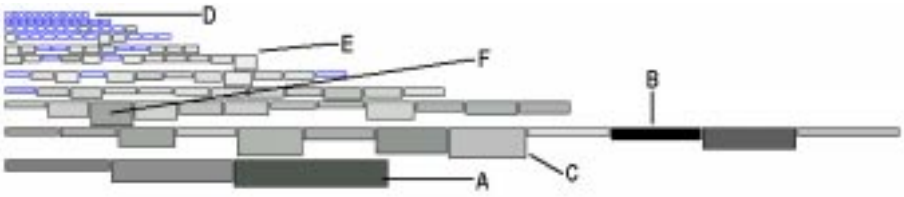

Figure 2.18: The system hot spots graph applied on Duploc using as size metrics NOM and NIV, and as color metric WLOC. Sort according to NOM.

Possible Alternatives: The color metric can be varied at will. A sort according to other metrics (especially WLOC and NCV) can also give interesting results which emphasise certain nodes.

Evaluation: The main drawback of the SYSTEM COMPLEXITY graph described in Section 2.2.10.1 is the fact that through the ordering of the nodes in tree structures we lose track of the size of the nodes all too easily. Only extreme cases strike our eyes. The SYSTEM HOT SPOTS graph described here makes this up through the sorting of the nodes and an ordering of them which reflects this sorting. However we lose the notion of inheritance in this case, since displaying the edges would mess up the view. A certain disadvantage of this graph is that the more nodes we display the more space is needed.

2.2.10.3 Weight Distribution

Graph	Histogram, size addition layout, sort according to width metric.	
Scope	Full system.	
Metrics		
Size	NOM (number of methods)	-
Color	HNL (hierarchy nesting level)	
Position	-	NOM

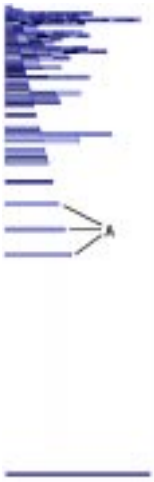

Figure 2.19: The weight distribution graph applied on the Refactoring Browser. As width and vertical position metric we use NOM, as color metric we use HNL.

General Idea: With this graph we are able to make a general assessment on the system we are investigating. The width and the vertical position of the nodes is reflected by NOM, the color represents their HNL. This means that the deeper down (in the graph) the class nodes are, the more methods these classes implement. A dark node on the other hand means that the class it represents has a deep hierarchy nesting level. The possible assessments we can now make are:

- The system is *top-heavy*. This means that the classes that implement the most functionality are high up in the inheritance hierarchies. Such a graph has big nodes (on the bottom of the graph) which have very light color values (because their HNL is small). Top-heavy systems suffer when it comes to subclassing and reusing, because their root classes do too much themselves.
- The system is *bottom-heavy*. The most functionality is implemented in classes deep down the inheritance hierarchies. Such a case displays dark, big nodes on the bottom of the graph. Bottom-heavy systems are sometimes the results of overzealous abstracting mechanisms.
- The system is *even*. This display looks somehow chaotic, because the dark and light nodes distribute themselves over the whole graph. This case balances the two cases described above.

Results with the Refactoring Browser: The Refactoring Browser is an evenly distributed system, as we see in Figure 2.19: It's not possible to locate a majority of the dark or the light nodes on a certain area of the graph, although we can see there are three big classes marked as (A) high up the hierarchy.

Figure 2.20: The weight distribution graph applied on Duploc. As width and vertical position metric we use NOM, as color metric we use HNL.

Results with Duploc: Duploc is clearly a top-heavy system, as we see in Figure 2.20: The dark nodes are all very small (small NOM) and thus located on the top region of the graph. The big classes on the bottom of the graph are all very light (high up in the hierarchy). The system is thus to be classified as top-heavy, which is mainly due to its young age: Duploc has not yet undergone a reengineering or refactoring. It should be analysed on whether it's possible to introduce a supplemental abstraction level high up in the hierarchy.

Possible Alternatives: The width metric can be varied, especially NIV (number of instance variables) can give some supplemental information on the complexity of the classes. The color metric can also be changed, especially WLOC (lines of code) shows a good behaviour.

Evaluation: This graph can make a general assessment about the system. Such an assessment may not be very useful and will most probably not involve a specific problem, but upon such statements about the subject system we can vary our approach. In fact, the more we know about the system before we dive into its details, the more precisely we can deploy the other graphs.

2.2.10.4 Root Class Detection

Graph	Correlation.	
Scope	Full system or very large subsystems.	
Metrics		
Size	*	*
Color	*	
Position	WNOC (total number of children)	NOC (number of children)

General Idea: In very large systems with many inheritance hierarchies it may be difficult to identify at once the classes which have the most impact on their subclasses. The impact of a class on its descendants can be measured with the number of direct subclasses and the total number of subclasses of a class: the more there are, the more the functionality implemented in a root class is used. This graph shows the correlation between WNOC (total number of subclasses) and NOC (number of direct subclasses).

The further away from the origin such a class node is, the bigger is its impact. The type of inheritance used for a class can also be identified with this graph:

- If a node is positioned on the right side of the graph, while holding a vertical position near the top, this means that while the underlying class has a great number of descendants its direct subclasses are few. This is often the case when directly below a root class a supplemental abstraction level of classes has been introduced.
- If the node finds itself on the 45 degrees axis (it can't be further left because WNOC is always at least equal to NOC) and far away from the top of the graph, this means that the underlying class has a lot of direct subclasses. This is what we call a *flying saucer hierarchy* because the inheritance tree of this class resembles one.
- If a class node is positioned exactly along the 45 degrees axis this means that all its subclasses don't have subclasses themselves, and thus are leaf node classes in an inheritance tree.

Results with the Refactoring Browser: To make the effect of this graph more visible, in Figure 2.21 we see on the top left the root class detection graph while on the bottom right we see a display of two major inheritance trees. We see the class *Refactoring* (A) which has 43 descendants and 5 direct subclasses as root of major inheritance tree on the right side of the correlation graph. The other root class, *BrowserApplicationModel* (B) can also be identified on the right side of the graph. Two classes, *MethodRefactoring* (C) and *VariableRefactoring* (D), which are the heads of minor flying saucer hierarchies (14 and 13 direct subclasses) can be identified near the 45 degrees axis.

Results with Duploc: The results of this graph are somewhat deceiving in the case of Duploc, as its inheritance hierarchies are very flat. We can detect however two root classes, namely *PresentationModel-ControllerState* (A) and *PMCS* (B). In Figure 2.22 we see where the detected root classes are located in one of the inheritance hierarchies of Duploc.

Possible Alternatives: We do not make use of the color and size metrics, which could add information to this graph.

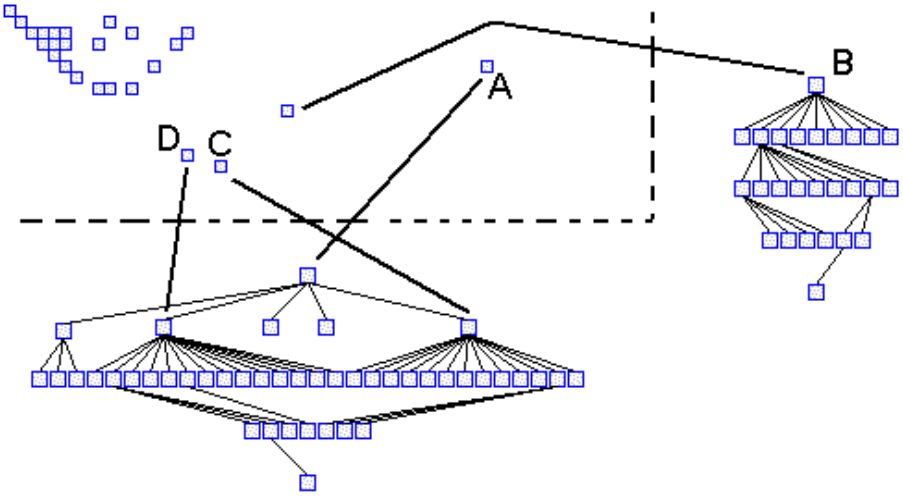

Figure 2.21: A root class detection graph applied on the Refactoring Browser. As position metrics we use WNOC and NOC.

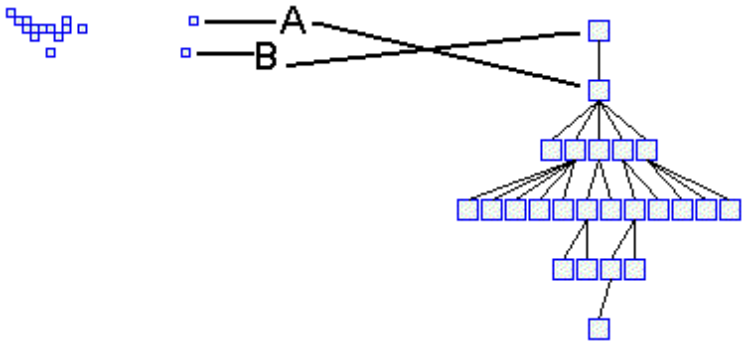

Figure 2.22: A root class detection graph applied on Duploc. As position metrics we use WNOC and NOC.

Evaluation: The detection of flying saucer hierarchies can of course be done through an inheritance tree display. The resulting tree graph has then to be searched for them. However, in some cases where the number of classes was very large, the resulting graph would become several screens big. In such cases it's not easy to detect flying saucers at once, and the graph described here comes into play. This graph can come in handy to see if there are some inheritance hierarchies upon which we want to apply inheritance specific graphs like intermediate abstract or inheritance impact.

2.2.10.5 Service Class Detection

Graph	Stapled, sort according to width metric.	
Scope	Subsystem or small full system.	
Metrics		
Size	NOM (number of methods)	WLOC (lines of code)
Color	NOM	
Position	-	-

General Idea: This graph has proven to be useful for the detection of so-called *service classes*. A service class is a class which mainly provides services to other classes. It often contains some tables and dictionaries which other classes can access for their purposes. Such classes often have an aberrant ratio between NOM and WLOC: they have very short methods which mainly access or return values. In this kind of graph we present a selection of some classes (a whole inheritance tree is often a good choice) as a stapled graph. The classes have been sorted according to their width, which represents NOM.

Because there tends to be a certain relation between NOM and WLOC, we should get a sort of staircase effect on the nodes the more we move to the right.

We can make out the following:

- If a class node breaks the staircase effect (by being too short) it is a candidate for a service class.
- This graph can also serve as detector for classes with overlong methods: If the class breaks the effect in the other direction (by being too tall) it's a candidate for method splitting, because this means that it has many lines of code (tall) and comparatively few methods (narrow, and because of the sorting pushed to the left side of the graph).

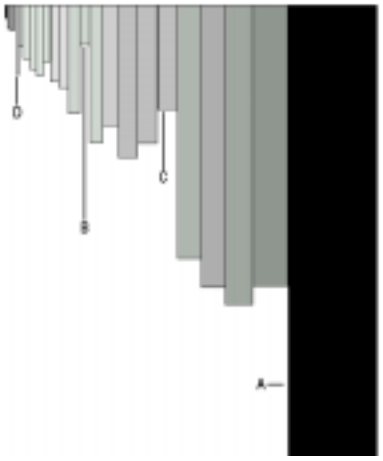

Figure 2.23: The service class detection graph applied on a subhierarchy of the Refactoring Browser. As width metric and sorting criterion we use NOM, the height metric is WLOC.

Results with the Refactoring Browser: In Figure 2.23 we selected a whole inheritance tree (26 classes) of the Refactoring Browser to be displayed in a SERVICE CLASS DETECTION graph. We see one huge class *BrowserNavigator* (A), which in fact is even bigger, but we cut it down because of space reasons. We see quite clearly that there is a certain tendency for a staircase which is severely broken in two places. The first

service class candidate is *CodeTool* (B), which has 22 methods and 49 lines of code. A closer inspection reveals that the methods are mainly get/set-methods (accessors). The second candidate is *CodeModel* (C) with 40 methods and 136 lines of code. The name itself already reveals the service function this class is intended to have. As method splitting candidate we detect the class *ClassCommentTool* (D) which has only 7 methods but 89 lines of code.

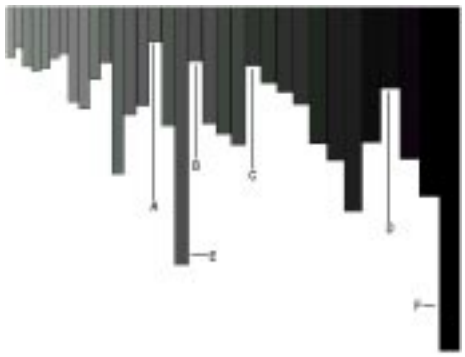

Figure 2.24: The service class detection graph applied on a subset of Duploc. As width metric and sorting criterion we use NOM, the height metric is WLOC.

Results with Duploc: We obtained the graph in Figure 2.24 by first applying the graph on the whole system and then by selecting a subset which looked interesting. We see there are some candidates for service classes: The class *CachedObservationData* (A) contains 20 methods for a total count of 50 lines of code. A closer inspection reveals it is truly a service class. Nearly the same ratio is visible in the classes *ComparisonMatrixBody* (B) (22/80), *PresentationModelControllerState* (C) (25/87) and *ObservationOnRawSubMatrix* (D) (30/122). Some classes tend to have overlong methods, namely *PMVSInformationMuralMode* (E) (22/396) and *DuplocCodeReader* (F) (32/530), and should be looked at for possible method splitting.

Possible Alternatives: Nearly the same results can be obtained if we use NIV (number of instance variables) instead of NOM: both NOM and NIV are closely related in service classes (because of the accessors). Sometimes abstract classes higher up the hierarchy tend to have the same properties as service classes, because their abstract nature makes them have several very short methods which are later overridden or extended by the subclasses. This can be alleviated if we use HNL (hierarchy nesting level) as color metric. Service class candidates which are true service classes tend then to have a darker color shade. Fake service classes like the abstract ones will have a lighter color shade because they are higher up the hierarchy.

Evaluation: As this graph addresses a special problem, it should be used in a second phase of reverse engineering. Experience has shown that it's advisable to apply it on subsystems, especially inheritance hierarchies.

2.2.10.6 Cohesion Overview

Graph	Checker, quadratic, sort according to width metric.	
Scope	Full system or subsystem.	
Metrics		
Size	NOM (number of methods)	WNAA (number of direct accesses on attributes)
Color	NIV (number of instance variables)	
Position	-	-

General Idea: In this graph the nodes differ greatly in shape and color. In the best case this graph can give us some clues on which classes we should inspect for a possible splitting. We distinguish the following:

- The flat nodes indicate that the methods of a class (the width indicates the number) do not access many times its instance variables. This is further emphasised by the small height (few instance variable accesses).
- The narrow and high nodes on the other hand, tend also to be very light colored. This case happens when the classes have many accesses but only few instance variables. This is mostly the case when the class defines an attribute which is then heavily accessed directly by its subclasses. This is not advisable because of the lacking encapsulation: a single access through an accessor which would then be invoked by other classes, instead of direct accesses on the attribute, would be much better.
- More or less rectangular nodes with darker color shades indicate a good cohesion inside those classes, although this is only provable after applying a class cohesion graph, which is described in Section 2.2.13.1.

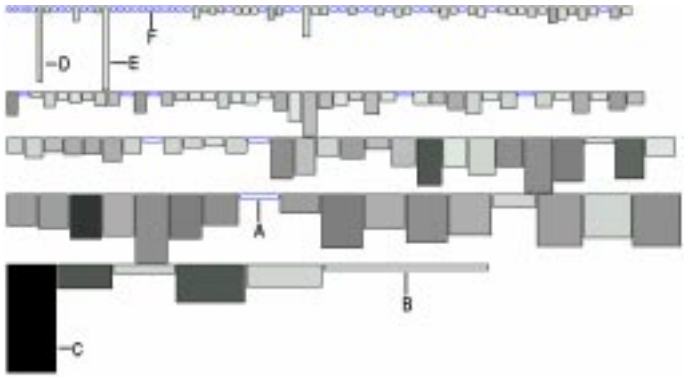

Figure 2.25: A cohesion overview graph applied on the Refactoring Browser. As size metrics we use NOM and WNAA. As color metric NIV is used.

Results with the Refactoring Browser: The resulting graph can be seen in Figure 2.25. The first thing we notice is that the nodes differ heavily in their shapes and colors. There are some white nodes that don't define instance variables (for example (A)) and because of this absence they can't have any instance variable access either. This is the reason for their flat shape. We also gather there are some empty or nearly empty ones (located around (F)). The class *BrowserNavigator* strikes once again the eye for its huge number of methods and its small number of instance variables (only one). The nodes (D) and (E) strike the

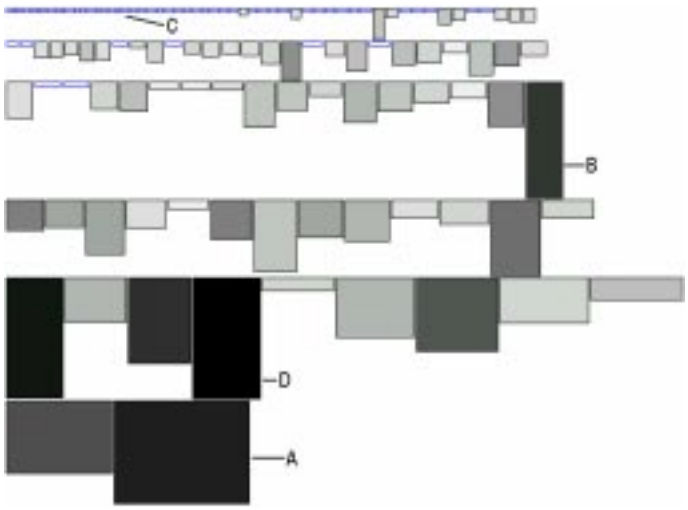

Figure 2.26: A cohesion overview graph applied on Duploc. As size metrics we use NOM and WNAA. As color metric NIV is used.

eye for their narrow shape and light color: Both have few methods and instance variables, (1,2) and (2,1) respectively, while at the same time they have a huge number of accesses. The reason for this is that their variables are directly accessed by their subclasses. The class *BRScanner* (C) shows a great complexity and heavy access.

Results with Duploc: The graph in Figure 2.26 shows a few characteristics of Duploc: Many empty or nearly-empty classes (C), quite a few heavy-access classes (B) and (D) and a few very large classes, for example *DuplocApplication* (A). We see there are quite a few classes that could be interesting for inspection with a class cohesion graph and do that for one special case, the class *DuplocApplication* in Section 2.2.13.1.

Possible Alternatives: None.

Evaluation: This graph can be seen as an *in-betweener*, because it comes after a graph for general overview and before a graph which treats class internals. The best result it can return is a collection of classes which we should further examine with a class cohesion graph, described in Section 2.2.13.1.

2.2.10.7 Spinoff Hierarchy

Graph	Inheritance tree, centered, without sort.	
Scope	Subsystem, especially inheritance hierarchies.	
Metrics		
Size	WNOC (total number of children)	NOM (number of methods)
Color	WNOC (total number of children)	
Position	-	-

General Idea: We have noticed that in inheritance hierarchies the notion of inheritance is often carried on only by one or two classes on each level of the inheritance tree. This means that when a class has some subclasses often only one of them is really carrying on the weight of the inheritance, while its siblings tend to be *spinoff* classes implementing only few functionalities. Although this is not a bad thing per se, an easy detection of such spinoff hierarchies could make us focus on the inheritance carriers, while we could save time by ignoring (at least at the beginning) the less important spinoff classes. Spinoff classes often implement few methods and have few or no subclasses at all.

We distinguish the following:

- Small, light colored nodes. These are the *spinoff classes* with few or no children and few methods.
- Large, dark colored nodes. These are the *inheritance carriers*.

Results with the Refactoring Browser: In Figure 2.27 we see all inheritance hierarchies that make up the Refactoring Browser. We filtered out all stand-alone classes to get a clearer overview. We detect two cases of spinoff hierarchies:

1. The one with the class *BrowserApplicationModel* (A) as root. We see two classes split up the second level of this tree, namely *CodeTool* (A21) and *Navigator* (A11). There are a few spinoff classes on this level, neither of them has subclasses. The same situation is present on the next level of this tree where the classes *BrowserTextTool* (A22) and *BrowserNavigator* (A12) carry on the weight of inheritance. A good example for spinoffs is visible between *CodeTool* (A21) and *BrowserTextTool* (A22): *CodeTool* has 7 subclasses but only one of them, *BrowserTextTool*, carries on the inheritance. Each one of its siblings is very small (keep in mind that the height reflects NOM) and is thus a spinoff.
2. The one with the class *Refactoring* (B) as root. Again two main inheritance threads are visible: The one consisting of *Refactoring* (B), *MethodRefactoring* (B11) and *ChangeMethodNameRefactoring* (B12). The other consists of *Refactoring* (B), *VariableRefactoring* (B21) and *RestoringVariableRefactoring* (B22).

The other inheritance trees in this display also show some property of a spinoff hierarchy, and could be a case of further investigation.

Results with Duploc: After removing the many stand-alone classes from Duploc, the remaining graph in Figure 2.28 can only show us the absence of spinoff hierarchies. Especially in the tree with the class *PresentationModelControllerState* (A) as root, we see that on the third level we have 5 siblings, 4 of which are all inheritance carriers, with only one tiny spinoff class with the meaningful name *PMCSDummyMode* (B).

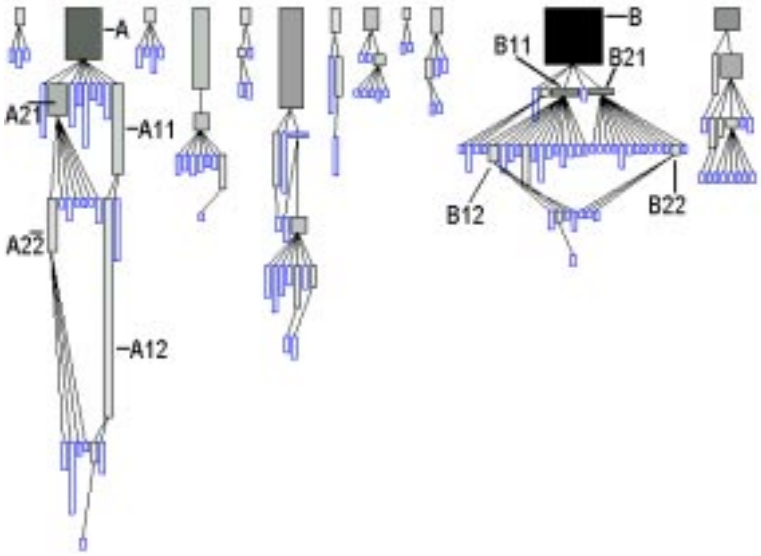

Figure 2.27: The spinoff hierarchy graph applied on the inheritance hierarchies of the Refactoring Browser. As size metrics we use WNOC and NOM, as color metric WNOC.

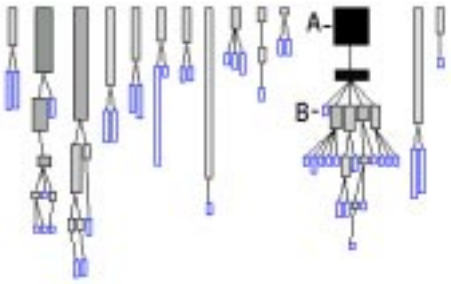

Figure 2.28: The spinoff hierarchy graph applied on Duploc. As size metrics we use WNOC and NOM, as color metric WNOC.

Possible Alternatives: We have to emphasise that a preprocessing consisting of filtering out all stand-alone nodes is advised for this graph, as they add unnecessary complexity to the displayed graph. This graph does not have real alternatives, as it addresses a special problem.

Evaluation: This graph should come into play in a later phase of the reverse engineering, as it addresses a special problem which may not be present at all in the system. The detection of an inheritance carrier could be important, as it is the place which should be checked out because subclasses depend on it. The spinoff classes on the other hand, can be examined for possible push-ups of functionality.

2.2.10.8 Inheritance Impact

Graph	Inheritance tree, without sort.	
Scope	Subsystem, especially inheritance hierarchies.	
Metrics		
Size	NMO (number of methods overridden)	NME (number of methods extended)
Color	NOM (number of methods)	
Position	-	-

General Idea: This graph is able to tell us if there has been made an improper or suspect use of inheritance: it can tell us if a class that implements many methods does not make use of method overriding or method extension, or uses it only rarely. Overriding and extending methods is one of the powerful properties of object-oriented languages and should be used if possible.

Nodes that override or extend a lot are bigger, nodes that implement many methods are dark. We are looking for dark nodes (many methods) which are at the same time very small (no use or rare use of overriding and extension).

Results with the Refactoring Browser: One of the hierarchies of the Refactoring Browser seems to have one such class which should certainly be further investigated: In Figure 2.29 we can detect the class *BrowserNavigator* (A) which implements many methods (175), while it only overrides one and extends two methods.

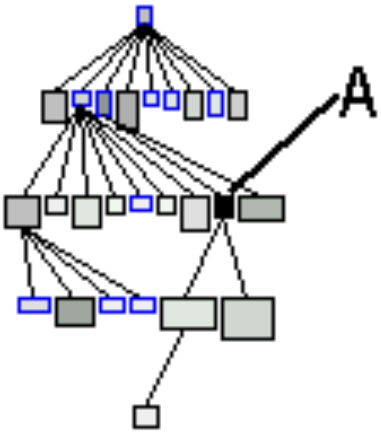

Figure 2.29: The inheritance impact graph applied on an inheritance tree of the Refactoring Browser. As size metrics we use NMO and NME, as color metric NOM.

Results with Duploc: This graph returns no meaningful results if applied on Duploc.

Possible Alternatives: No real alternatives, as it addresses a specific problem. This graph is often obtained after filtering out all stand-alone classes and all inheritance hierarchies which show no sign we are looking for.

Evaluation: A graph which addresses a very special problem. It's not always useful, but if it can detect something, it can be an important discovery which can affect a whole inheritance hierarchy.

2.2.10.9 Intermediate Abstract Class

Graph	Inheritance tree, without sort.	
Scope	Subsystem, especially inheritance hierarchies.	
Metrics		
Size	NOM (number of methods)	NMA (number of methods added)
Color	NOC (number of children)	
Position	-	-

General Idea: This graph is useful for the detection of abstract classes or nearly-empty classes which are located somewhere in the middle of an inheritance chain. Often they tend to have a superclass which implements a lot of methods. The programmer then started to subclass this class. The number of direct subclasses would soon be too big so an attempt was made to logically group several subclasses under an abstract intermediate class.

Such an intermediate subclass would normally have many children, while at the same time its size is very small (because it is abstract or nearly empty). We thus have to look for small, dark nodes in the middle of inheritance hierarchies.

The dark color comes from the greater number of direct subclasses, while the small size from the small functionality implemented. We chose NMA as height metric to reflect the fact that often such intermediate abstract classes don't override superclass methods, which in turn means that is we use NOM as width metric, the node is square (no functionality implemented, or if there is a bit of implemented functionality, then it doesn't come from the superclass). Intermediate abstract classes are of some interest, because often we can try to push up some functionalities of its subclasses into it, thus concentrating them in one place, instead of spreading the functionality all over the subclasses, risking to obtain duplicated code.

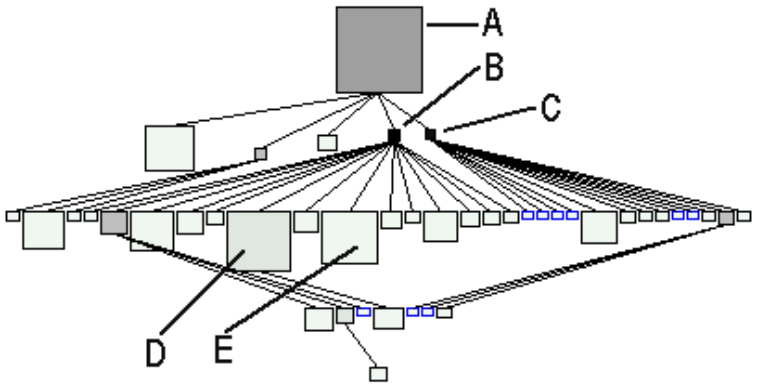

Figure 2.30: The intermediate abstract class graph applied on a subset of the Refactoring Browser. As size metrics we use NOM and NMA, as color metric NOC.

Results with the Refactoring Browser: The Refactoring Browser harbours in one of its inheritance hierarchies two intermediate abstract classes, as we see in Figure 2.30. The root class *Refactoring* (A) implements quite a few methods, while we can spot the two intermediate abstract classes as *MethodRefactoring* (B) and *VariableRefactoring* (C). These two classes implement themselves very few methods (2 and 1 respectively) and are the roots of smaller subhierarchies. In the case of *MethodRefactoring* we see that its

subclasses are implementing several methods, as we see in *InlineMethodRefactoring* (D) and *MoveMethodRefactoring* (E). Perhaps an attempt could be made to extract duplicated code and push it up into the intermediate abstract class.

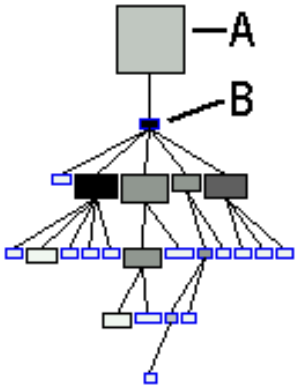

Figure 2.31: The intermediate abstract class graph applied on an inheritance hierarchy of Duploc. As size metrics we use NOM and NMA, as color metric NOC.

Results with Duploc: One of Duploc's inheritance hierarchies also contains an intermediate abstract class, as we see in Figure 2.31: The subclass *PMCS* (B) of the root class *PresentationModelControllerState* (A) implements only 4 methods and is obviously an intermediate abstract class. The subclasses of *PMCS* should be searched for duplicated code which could be pushed up into *PMCS*.

Possible Alternatives: None.

Evaluation: The detection of abstract classes is very important: several object oriented languages either directly provide a declaration or support a standard idiom for identifying abstract classes. Abstract or nearly abstract classes can be seen as the hinges of the system, upon which several classes depend. It's where the common functionality is defined and where we should start to look at source code if we want to understand the logic of their subclasses.

2.2.11 Useful Graphs: Method Graphs

Method graphs can work at any level of granularity most of the time. However, the more method nodes we display, the harder it is to make out outliers. Methods are the entities which are responsible for the action in a system. This implies that every graph which uses method nodes is often followed by an examination of the actual underlying source code. This means that the graphs listed here have a very concrete context.

In this section we list the following graphs:

- METHOD EFFICIENCY CORRELATION, Section 2.2.11.1.
- CODING IMPACT HISTOGRAM, Section 2.2.11.2.
- METHOD SIZE NESTING LEVEL, Section 2.2.11.3.

2.2.11.1 Method Efficiency Correlation

Graph	Correlation.	
Scope	Full system, subsystem or single class.	
Metrics		
Size	NOP (number of parameters)	NOP
Color	*	
Position	LOC (lines of code)	NOS (number of statements)

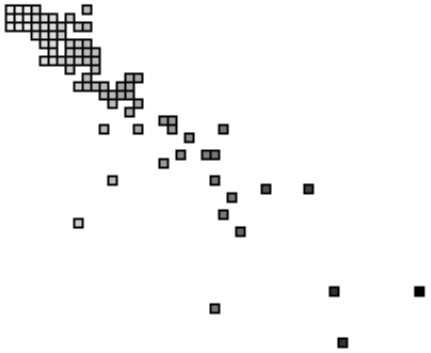

Figure 2.32: A method efficiency correlation graph.

General Idea: This graph is a good way to locate the *freaky entities* inside a group of methods, when it comes to their efficiency. By efficiency we mean how many statements are put on each line. By displaying the nodes in the correlation graph (as in Figure 2.32), we see that most of the nodes are near a certain correlation axis. However, there are a few which do not adhere to this rule.

The methods that are not near the correlation axis may have some problems, which may be

1. High LOC (lines of code) and low NOS (number of statements). This is for example the case with ”forgotten methods”, that at some point have been commented out and then been forgotten. This may also be the case for overzealous line indentation, when a single parenthesis is put on a line of its own or when many blank lines have been used.
2. Low LOC and high NOS. This can be the case when the methods are written without indentation and several statements are on the same line, which is a bad thing too, since this decreases the readability, and it may also break the law of Demeter [LIEB 89].
3. Long methods (high LOC and high NOS). Normally a case for redesign, since long methods should be split up in smaller, better understandable and reusable ones [BECK 97].
4. Empty methods. These nodes position themselves on the top left of the graph. Although they can be viewed there by selecting and moving, the overlapping of the nodes which is characteristic for this graph makes it hard to see those empty methods at one glance. A better graph for the detection of empty methods is the Coding Impact Histogram described in Section 2.2.11.2.

Other hot spots can be detected by looking at the size of the nodes:

- Big nodes have many parameters. Although it’s hard to define a threshold on the number of parameters, we think that methods taking more than 5 parameters should be looked at.

- Very small nodes on the outskirts of the graph should be looked at: these are very long methods which do not take any input parameter. Perhaps they could be split up easily.

The interesting property of this graph is its scalability. Since most of the nodes overlay each other, and those nodes are of no real interest to us, because they have average metric measurements, we can display several thousands of nodes at the same time. Our interest is drawn by the nodes which find themselves on the outer skirts of the graph, and which do not suffer overlaying, as their position is defined by their non-average metric measurements. The size of this graph is not affected by the number of displayed nodes, but on the maximum values for the position metrics.

Results with the Refactoring Browser: The method efficiency correlation graph shows some interesting results when applied to the Refactoring Browser. In Figure 2.33 we display all 2365 methods of the Refactoring Browser. We can spot several cases which should be looked into. The first nodes to meet the eye are those on the right edge of the graph (A). These three methods are very long (45, 51 and 65 lines of code) compared to the other methods in the system, which does not have a great distribution, thus signifying that the system is homogeneous related to the method lengths. The opposite case can be seen on the top left side of the graph (B). Upon closer inspection (by selecting and moving the nodes) we can see that the RefactoringBrowser contains 20 empty methods. The next point of interest is the method marked (C): this method takes 7 input parameters which is of course very much. The method *reInstallInterface* (D) on the top of the graph is also a case of closer study: While it has 16 lines of code it contains no statements. If we browse its source code, we see that the whole body of the method has been commented. The method *needsParenthesisFor:* (E) on the other hand contains 31 statements in only 19 lines of code and should perhaps be reformatted. The group of methods marked as (F) should also be looked into, since all of them contain comparatively few statements in long method bodies.

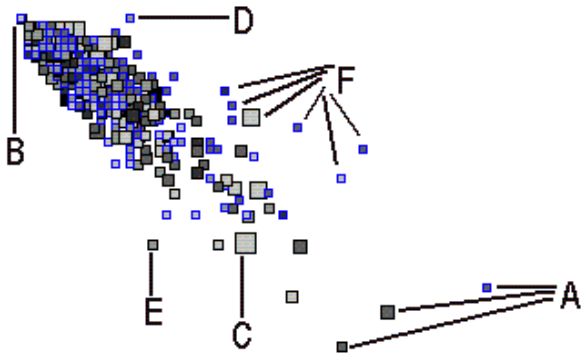

Figure 2.33: The method efficiency graph applied on the Refactoring Browser, using as position metrics LOC and NOS, as color metric HNL, and as size metric NOP.

Results with Duploc: When this graph is applied to Duploc, as we see in Figure 2.34, the first thing to strike the eye is the large distribution of the nodes. Duploc obviously does have some very long methods. The second thing that meets the eye is that the main correlation axis has a different angle compared to the Refactoring Browser in Figure 2.33. The method *putPerlCode:* (A) is 201 lines long but does have only 2 statements. Upon closer inspection we see that its purpose is to print out a very long string. We have some other very long methods, (B) with 135 lines, (C) with 95 lines, (D) with 109 lines. We have some method that are far away from the system correlation axis, like (A), (C), (E), (F) and (G). (E) for example has 64 lines of code with only 1 statement. A closer inspection reveals its body is mainly commented code for testing purposes, i.e. when the system is tested some parts of the method body are uncommented. (F) reveals the same situation, where the 18 lines long method body doesn't contain any statements at all. (G) has 32 statements packed in 14 lines of code. Reformatting makes it more readable. The empty methods

can of course be detected as (H), while we should also note the nodes around (I), which seem to be very short and at the same time badly formatted methods. The two methods (J) also draw attention due to their considerable size, which reflects the fact that they take 9 input parameters each.

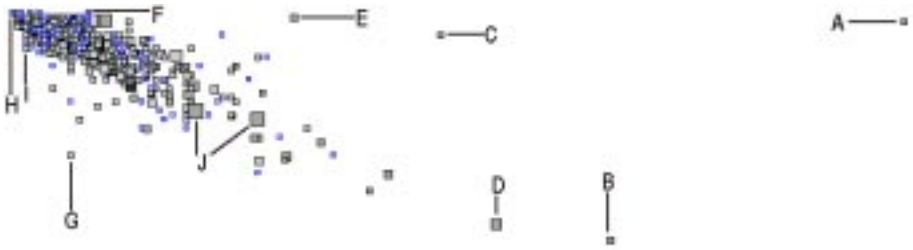

Figure 2.34: The method efficiency graph applied on Duploc, using as position metrics LOC and NOS, as color metric HNL, and as size metric NOP.

Possible Alternatives: We chose the size of the nodes to be represented by NOP (number of parameters). Since the distribution tends to get sparse the more we move to the right and to the bottom, we can see the methods which take many parameters more clearly, since it's normally the large methods that take more parameters. Generally in this graph the size metrics can be chosen freely, although it's advisable to use metrics which tend to have small measurements. Otherwise the nodes get very big and clutter up the view. The color metric can also be used freely. We chose HNL (hierarchy nesting level) in this case, but since the nodes in this graph tend to be very small, the color node metric doesn't really matter.

Evaluation: This is one of the few graphs which works very well at any level of granularity. As such it can be used anytime. We saw it can be useful to apply it on a subsystem before we dive into its details. At class level it can help to detect problem cases for a concrete reengineering.

2.2.11.2 Coding Impact Histogram

Graph	Histogram, size addition layout, sort according to width metric.	
Scope	Single class or small subsystem.	
Metrics		
Size	LOC (lines of code)	-
Color	LOC	
Position	LOC	-

General Idea: This graph shows the coding impact of methods and where the most coding has happened. While the normal histogram can only tell us how methods are distributed regarding their lines of code, this graph (Figure 2.35) can reveal where the real programming effort has been made: Writing 20 methods each one line long is easier than writing one method 20 lines of code long. It shows if there are any aberrant methods that are too long or if the system is unbalanced because of too long and complex methods. As a nice side-effect we can also grasp at one glance if there are any empty methods (those at the very top of the graph). A good design should have a lot of tiny methods so this is where the biggest columns in the graph should be. Methods not following this rule should be analysed as possible ”split candidates” which could be broken down into smaller pieces. While this graph is inefficient on whole systems because of the huge number of methods, it has proven to be very useful when applied to the methods of one single class. It should also be noted that the average length of a method implemented in typical industrial Smalltalk applications is around 6 lines [BECK 97].

Figure 2.35: A coding impact histogram.

Results with the Refactoring Browser: Since this is one of the graphs which can hardly be applied on whole systems, but rather on specific small subsystems or singular classes, we do not compare the systems from our case studies with each other, but we rather show a few illustrative examples taken out randomly[8] from the Refactoring Browser. We selected only the two classes (*BrowserNavigator* (B) and *BRProgramNode* (A)) with the most methods for this graph. We see in Figure 2.36 that each class has its own coding impact topography. We see that *BrowserNavigator* (B) has many methods which tend to be overlong, and especially 6 very long ones which isolate themselves (B1) from the others. On the other hand *BRProgramNode* has an irregular topography with many accessors (A2) and one very long method (A1).

Possible Alternatives: This graph knows many useful mutations, especially those which keep LOC as vertical position metric, but use other size and color metrics and a different sort criterion. In these cases, especially NI (number of invocation) and NMAA (number of accesses on attributes) showed good behaviour.

[8]This randomness should also express the interactive approach of such systems, which is guided by intuition rather than a systematic methodology, although experience has shown that at the beginning of a reverse engineering experiment we tend to apply a certain fixed set of graphs. This reflects the fact that the graphs address each a different level of abstraction.

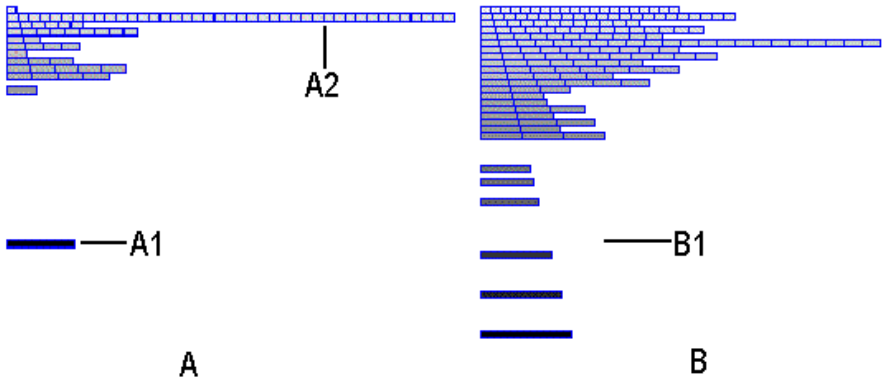

Figure 2.36: The coding impact graph applied on two classes of the Refactoring Browser. The width metric, as well as the color and vertical position metric is LOC.

Evaluation: This graph is very useful to *get a feeling* for certain classes or subsystems. It can show us what kind of implementation lies behind the subject entities and in certain cases what we should continue to explore.

2.2.11.3 Method Size Nesting Level

Graph	Checker, quadratic, sort according to width metric.	
Scope	Subsystem, especially inheritance hierarchy. No stand-alone classes.	
Metrics		
Size	LOC (lines of code)	NOS (number of statements)
Color	MHNL (hierarchy nesting level)	
Position	-	-

General Idea: A general rule is that big methods should be split up [BECK 97] into smaller chunks to increase their reusability and to make them easier to understand. This is especially true for methods that are implemented in classes deep down the inheritance hierarchy: perhaps parts of those big methods could be extracted and put up into a higher class to reuse them across several subclasses. The method size nesting level graph can help us to detect large methods deep down the inheritance hierarchy: It's a checker graph of methods with LOC and NOS as size metrics and MHNL as color metric. The nodes are sorted according to LOC, which puts the larger methods on the bottom area of the graph.

Since the color reflects the MHNL of the methods, we should be looking for big, dark nodes in the bottom area of the graph: these are possible split candidates. We call such methods split-and-push-up candidates.

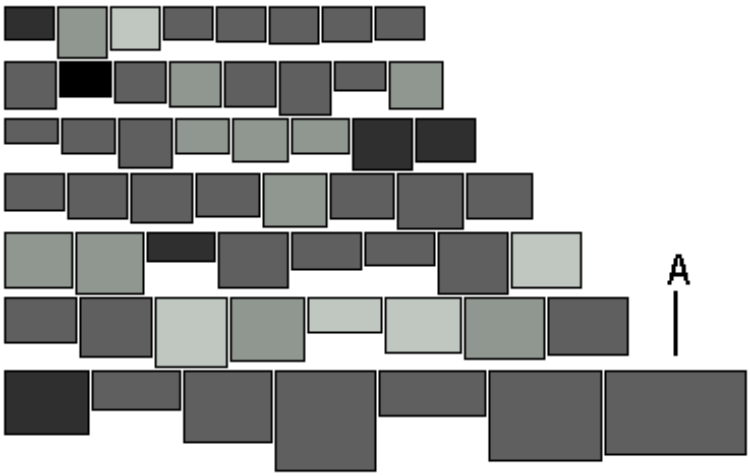

Figure 2.37: The method size nesting level graph applied on the largest Refactoring Browser methods. Size metrics: LOC, NOS. Color metric: MHNL.

Results with the Refactoring Browser: The Refactoring Browser shows in Figure 2.37 that is has been refactored itself a few times: there remain very few large methods, after filtering out all those with a LOC measurement smaller than 20. Yet, there are some large methods which also have medium MHNL values like those in the last row (A). Their lengths vary from 65 to 37 lines, which makes them also possible split-and-push-up candidates.

Results with Duploc: We display in Figure 2.38 only the methods that have more than 20 LOC and belong to non-stand-alone classes. The resulting graph shows us there are several very large methods, which on one hand don't have big MHNL values, but since they're not methods belonging to root classes either, are all the same split-and-push-up candidates. The biggest methods (A) have 201, 135 and 109

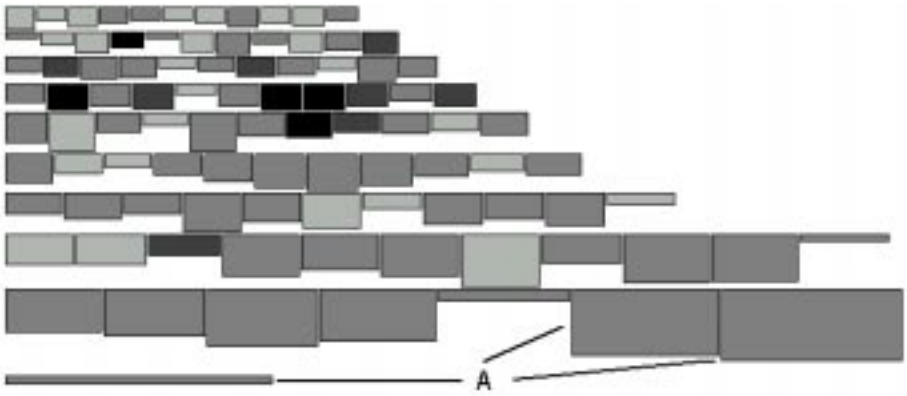

Figure 2.38: The method size nesting level graph applied on several Duploc methods. Size metrics: LOC, NOS. Color metric: MHNL.

LOC, which is way too much for Smalltalk methods. This excessive size is again due to the fact that most of them have never been refactored and written in one pull.

Possible Alternatives: The same graph using only LOC as size and color metric can be applied on whole systems (including stand-alone classes). In such a case the graph serves to easily detect very large methods which could be split up.

Evaluation: Since this graph is useful for classes belonging to inheritance hierarchies, it should primarily be used to get insights into such structures as to where the methods are which could be reengineering candidates.

2.2.12 Useful Graphs: Attribute Graphs

Attributes define the properties of classes. As such, it's mandatory that to understand the purpose of an attribute, we have to understand the class in which it is defined. This implies that very soon after applying one of the following graphs, we have to look at the source code of the class.

In this section we list the following graphs:

- DIRECT ATTRIBUTE ACCESS, Section 2.2.12.1.
- ATTRIBUTE PRIVACY, Section 2.2.12.2.

2.2.12.1 Direct Attribute Access

Graph	Checker, quadratic, sort according to width metric.	
Scope	Full system or subsystem.	
Metrics		
Size	NAA (number of times accessed directly)	NAA
Color	NAA	
Position	-	-

General Idea: This is a graph of all attributes of a system or subsystem. As metrics we use NAA (number of times accessed) for the size and the color. We then also sort the nodes according to NAA. What we get is a clear display of which attributes are accessed the most in a system. These attribute nodes are positioned at the bottom of this graph. The largest nodes should be a case for closer inspection. The general rule should be that attributes which are accessed directly can break the system if the inner implementation of the attribute changes. This can be avoided by using an accessor method which returns the value(s) of the attribute. An accessor on such an attribute can provide a defensive wall of protection against such changes. There may also be some attributes which are never accessed and which may have been forgotten in the system and thus only add unnecessary complexity to it. They could be removed from the system. Such attribute nodes are positioned on top of the graph.

Results with the Refactoring Browser: In Figure 2.39 we notice at once that there is the attribute *class* (A) defined in the class *MethodRefactoring* which is directly accessed 86 times. We also see there are some never accessed attributes which should also be further investigated (B).

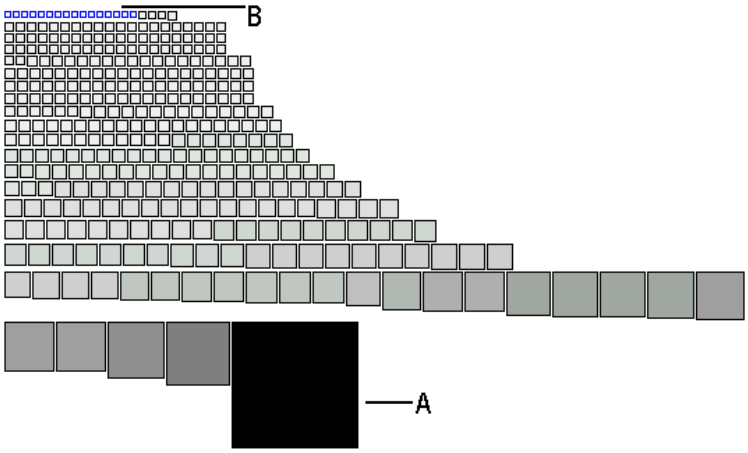

Figure 2.39: The direct attribute access graph applied on the Refactoring Browser. The size, color metric and sort criterion is NAA.

Results with Duploc: In Figure 2.40 we see that while in Duploc there are no attributes which are heavily accessed (the maximum is 31 direct accesses for the attribute *region* (A) defined in the class *AbstractRawSubMatrix*) there are many attributes which are never accessed (B) and which should be looked into for possible removal.

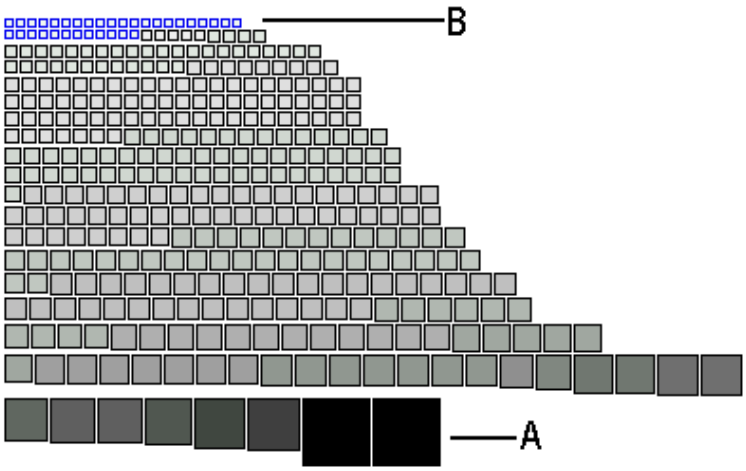

Figure 2.40: The direct attribute access graph applied on Duploc. The size, color metric and sort criterion is NAA.

Possible Alternatives: An interaction with interesting nodes is necessary to see if accessors have been implemented for them and if those accessor methods are used all the time.

Evaluation: A graph which works at every level of granularity. The next step which has to follow such a graph is to examine the classes in which the outlier attributes are defined. Note that this graph takes only the direct accesses into account. If an attribute is accessed very often through the use of an accessor method this will not show in this graph. Note that the quality of this graph depends heavily on the quality of the metamodel. Especially when building a model out of a CDIF file we have often seen that sometimes accesses are left out. This can lead us to wrong conclusion on never accessed attributes. Again, a check against the code has to be done to be sure.

2.2.12.2 Attribute Privacy

Graph	Checker, quadratic, sort according to width metric.	
Scope	Full system or subsystem. Better performance with C++ or Java.	
Metrics		
Size	NAA (number of times accessed directly)	NCM (number of classes which access this attribute)
Color	*	
Position	-	-

General Idea: Attributes may be directly accessed several times in a system. As we said in Section 2.2.12.1 such a situation is not ideal and can be detected with the graph described there. Apart from the number of times an attribute is accessed, another metric may prove to be useful for a similar graph: NCM, the number of classes which have methods that directly access a certain attribute. The attribute privacy graph is a checker graph which uses as size metrics NAA and NCM.

We are looking for wide, high nodes: such nodes are directly accessed a lot of times by many classes and should have an accessor at all costs, because the system easily breaks if such an attribute is tampered with.

Very wide but shallow nodes should also be looked at: although they are directly accessed a lot, it's by few or often only one class. If it's the case of only one accessing class, it should be checked it the attribute in question is private. If not, it can be made private without impact on the rest of the system.

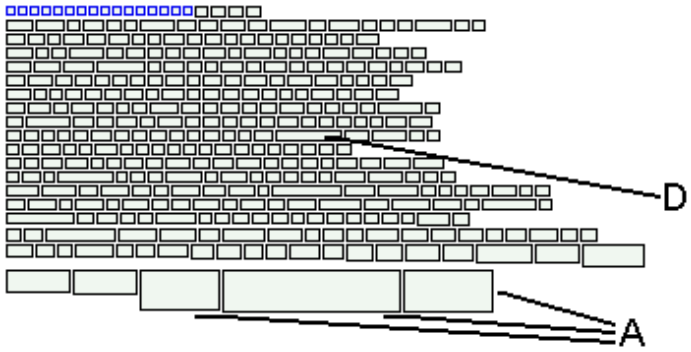

Figure 2.41: The direct attribute access graph applied on the Refactoring Browser. The size metrics are NAA and NCM.

Results with the Refactoring Browser: In Figure 2.41 we can spot some heavily accessed attributes marked as (A) which are accessed by many classes. We also see there are some very flat but wide nodes which are attributes heavily accessed by only 1 or very few classes.

Results with Duploc: In Figure 2.42 we can see that as a difference to the Refactoring Browser, Duploc has attributes which are seldom accessed by more than one class. The maximum NCM value is 3. We deduce from that that the implementor of Duploc keeps an eye on encapsulation[9].

Possible Alternatives: None.

[9]The implementor of Duploc used to implement a lot in C++, which could be a reason for the tight encapsulation.

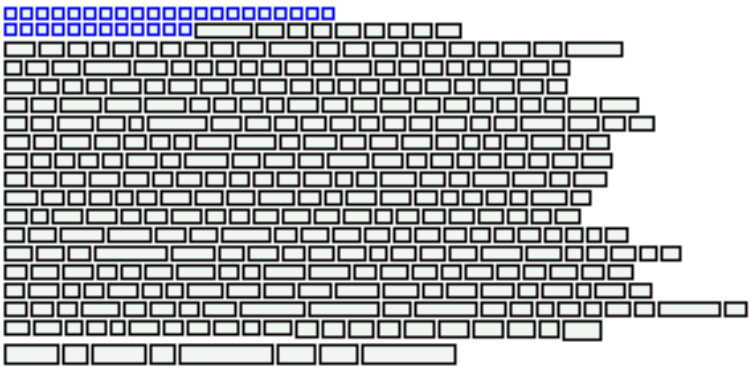

Figure 2.42: The direct attribute access graph applied on Duploc. The size metrics are NAA and NCM.

Evaluation: A graph whose purpose is to find attributes which have to be examined. Since such an examination takes place at textual level, it's a graph which can help find problems at once. The results are incomplete in this case: the last step after detecting wide and flat nodes would be to check if the attributes concerned are defined as private. If not they could be made private. However, this does not work in Smalltalk, so we had to leave that part out with our case studies.

2.2.13 Useful Graphs: Class Internal Graphs

A class internal graph treats the special case where the components of a class are displayed at the same time: methods and attributes.

In this case we find ourselves at a low level of abstraction, the source code is only one step away and it's necessary to look at it after applying a class internal graph.

In this section we list the following graph:

- CLASS COHESION, Section 2.2.13.1.

2.2.13.1 Class Cohesion

Graph	Confrontation graph, nodes sorted according to their width metrics..	
Scope	Single class.	
Metrics (Method Nodes)		
Size	LOC (lines of code)	NOS (number of statements)
Color	LOC	
Position	-	-
Metrics (Attribute Nodes)		
Size	NAA (number of times accessed directly)	NAA
Color	NAA	
Position	-	-

General Idea: This graph is a confrontation graph where the edges represent instance variable accesses between methods and attributes. This graph can indicate us how strong the internal cohesion of a class is. If a class has many accesses and looks very chaotic, this means that the class is difficult to split. On the other hand, if we can make out two or more separate clusters in this display, this is an indication that the class is a good split candidate. If the root class of an inheritance hierarchy shows such characteristics it is a sign that the hierarchy tends to be top-heavy. If the class shows sparse attribute accesses it could be easier to subclass.

Results with the Refactoring Browser: In Figure 2.43 we displayed the methods and attributes of the class *BRScanner* which has been identified as (C) in Figure 2.25. We gather at once that this class is heavily coupled internally and that splitting such a class is next to impossible.

Figure 2.43: A class cohesion graph applied on the class BRScanner. The method nodes (in the lower row) use as size metric NOS and as color metric LOC. The attribute nodes (in the upper row) use as color and size metric NAA.

Results with Duploc: We obtained some impressive results when we applied this graph to some classes of Duploc. We show only one here: the class *DuplocApplication*. After filtering out all methods that never accessed attributes, we got the graph displayed in Figure 2.44[10]. We clearly see two distinct clusters of attribute and method nodes. This class is thus certainly a split candidate. This suspect was confirmed afterwards when I asked the implementor of Duploc about this class. He confirmed that this class was to be split up during the next redesign of the system.

[10]Note that the graph resulted like this after direct manipulation of the graph (i.e. moving around nodes) and not because of a layout algorithm that can identify clusters. However, we included into CODECRAWLER the functionality to help us quickly identify such clusters.

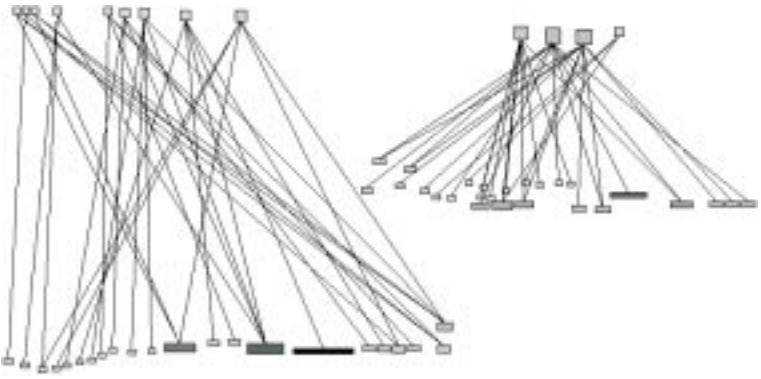

Figure 2.44: A class cohesion graph applied on the class DuplocApplication.

Possible Alternatives: We advise the user to remove all stand-alone nodes from the graph, as they are of no use in this case. The metrics, especially the color metric in the method nodes can be varied freely.

Evaluation: This graphs needs some interaction before it can express its full potential. However, its usefulness is indisputable: Up to this moment we haven't seen a technique which can detect split candidates with such an easy and quick method.