Towards a Methodology for the Understanding of Object-Oriented Systems

Stéphane Ducasse — Michele Lanza

Software Composition Group, University of Berne
Neubrückstrasse 12, CH – 3012 Berne, Switzerland
{ducasse,lanza}@iam.unibe.ch
http://www.iam.unibe.ch/∼scg/

ABSTRACT. *The reverse engineering of object-oriented legacy systems presents a number of problems typically encountered in large-scale legacy systems: the lack of overview and the need to focus on interesting parts. To help in reverse engineering large object-oriented legacy systems, we proposed a hybrid approach combining the immediate appeal of visualisations with the scalability of metrics. However, our approach lacked of a methodology that guides the reverse engineer. In this paper we present a first methodology that we developed from our industrial experiments.*

RÉSUMÉ. *La phase de compréhension et d'analyse (reverse engineering) lors de la rétro-conception d'applications à objets rencontre les problèmes typiques des grands systèmes (large scale systems), à savoir le manque de vue d'ensemble et la nécessité de se focaliser sur les parties intéressantes. Afin d'aider cette tâche, nous avons proposé une approche hybride combinant l'attrait de la visualisation avec celui des métriques. Cependant, notre approche manquait d'une méthodologie qui guide le rétro-concepteur. Dans cet article, nous présentons une première méthodologie que nous avons élaborée et validée au cours de nos expériences industrielles.*

KEY WORDS : *Reverse Engineering, Program Understanding, Program Visualisation, Reengineering, Software Metrics, Object-Oriented Programming, CodeCrawler, Moose.*

MOTS-CLÉS : *Reverse engineering, compréhension de programmes, visualisation de programmes, rétro-conception, métriques, programmation à objets, CodeCrawler, Moose.*

1. Introduction

The ability to reverse engineer object-oriented legacy systems has become a vital matter in today’s software industry. Early adopters of the object-oriented programming paradigm are now facing the problem of transforming their object-oriented legacy systems into frameworks, hence they need to understand the inner workings of their legacy systems and identify potential design anomalies. However, since legacy systems tend to be large —hundreds of thousands lines of poorly documented code are not an exception— there is a definite need for approaches that help in program understanding and problem detection [CAS 98]. As shown by the following definitions, reverse engineering is a part of the reengineering process in the sense that it focuses on analysing the system and providing representations of a system.

“***Reengineering*** *is the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.*”
“***Reverse engineering*** *is the process of analysing a subject system to identify the system’s components and their relationships, and to create representations of the system in another form or at a higher level of abstraction.*”[CHI 90]

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

A Hybrid Approach. We proposed a hybrid approach to understand existing program structures and identify potential design anomalies [DEM 99b]. As one of our strong constraint is the replication of the approach by reengineers without our tool support, the approach is based on the combination of simple graph layouts and easy to compute code metrics. For example, a class hierarchy can be displayed by a tree where the size of its nodes reflects metric values of the classes they represent. This way, the graphs can be enriched by more than 30 metrics producing a huge number of possibilities. As only a subset of the enriched graphs is meaningful, we identified so-called *useful graphs* [LAN 99].

The Need for a Methodology. However, despite the finding of those useful graphs, there is a definite need for a methodology that guides the reverse engineer depending on the state of his reverse engineering. A methodology should state in a goal-oriented way which graph to apply depending on the context and the previous graphs applied.

Based on our previous work and industrial experiments, the current paper introduces a methodology to guide the reverse engineer during the application and analysis of the graphs. The paper starts with an overview of the hybrid reverse engineering approach we developed (section 2) and presents an overview of the useful graphs that formed the base of the approach. Then in section 3 the methodology is presented by defining *clusters* of useful graphs based on the reverse engineering state and the possible paths to other graphs. In section 4 the methodology is illustrated on an industrial case study. In section 5 we discuss CodeCrawler, the tool that supports the methodology. Finally, we provide an overview of the related work (section 6), and conclude (section 7).

2. Combining simple metrics and trivial graphs

"*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.*" [BAL 96]

Before presenting our approach in detail we would like to present the context in which this approach has been developed.

2.1. *Replication in an Industrial Context as a Constraint*

One of the major constraints that we imposed on ourselves is that graphs and the methodology presented here are applicable in an industrial context, where software reengineering faces many problems, i.e. short time constraints, little tool support, and limited manpower. It is for this reason that we limited ourselves to use *simple metrics* that are easily extracted from source code entities using for example Perl scripts and to use *simple graph layouts* that could be easily implemented using scriptable tools like Rigi [Mül 86].

2.2. *Principle*

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

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

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

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

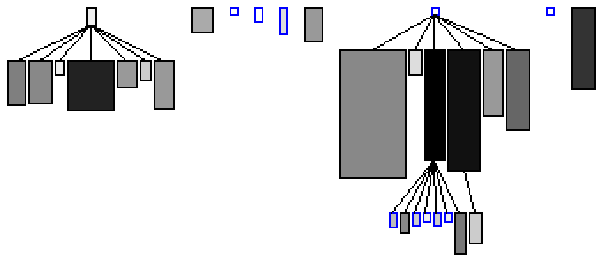

Figure 1. *Inheritance Tree; node width = NIV (number of instance variables), node height = NOM (number of methods) and colour = WLOC (lines of code)*

Figure 1 shows an example of an inheritance tree enriched with metrics information. The nodes represent the classes, the edges the inheritance relationships. The size of the nodes reflects the number of instance variables (width) and the number of methods (height) of the class, while the colour tone represents the lines of code of the class. The position of a node does not represent metrics in this case, as the position is given by the layout algorithm.

2.3. *Metrics*

We make extensive use of object oriented software metrics and measurements [FEN 97, HEN 96]. In the wide array of possible metrics that we could use, we selected so called *Design Metrics* [LOR 94], i.e. metrics that are *extracted from the source entities themselves*. These metrics are used to assess the size and in some cases the quality and complexity of software. Moreover, the ability to easily reproduce our approach lead us to select simple metrics. To this regard the work presented in this paper is on how to use simple metrics to support program understanding and not on the definition of metrics for code quality assessment.

We chose to use metrics that:

— can be easily extracted from source code entities,

— have a simple and clear definition, and

— are termed as *direct measurement* metrics [FEN 97], i.e. their computation involves no other attributes or entities.

As such we don't make use of *composite metrics* which raise the issue of dimensional consistency [HEN 96]. Furthermore we don't make use of *indirect measurement* where metrics are combined to generate new ones. A good example for such metrics are average and percentage metrics.

In Figure 1 we list all metrics mentioned in this article. The metrics are divided into three groups, namely class, method and attribute metrics, i.e. these are the entities that the metric measurements are assigned to.

Name	Description
Class Metrics	
HNL	Number of classes in superclass chain of class
WNMAA	Number of all accesses on attributes
WNOC	Number of all descendants
NCV	Number of class variables
WNOS	Sum of statements in all method bodies of class
Method Metrics	
NIV	Number of instance variables
LOC	Method lines of code
NMA	Number of methods added, i.e. defined in subclass and not in superclass
MHNL	Class HNL in which method is implemented
NME	Number of methods extended, i.e. redefined in subclass by invoking the same method on a superclass
MSG	Number of method message sends
NOP	Number of parameters
NMAA	Number of accesses on attributes
NMO	Number of methods overridden, i.e. redefined compared to superclass
NOS	Number of statements in method body
NOC	Number of immediate children of a class
Attribute Metrics	
NOM	Count all methods in class
AHNL	Class HNL in which attribute is defined
WLOC	Class lines of code
NAA	Number of times accessed

Figure 1. *Selected Measurements and Metrics*

2.4. *Actual Visualisation*

The actual visualisation depends on three factors:

1. **The graph type:** Its purpose is to emphasise those aspects of a system that are relevant for reverse engineering, e.g. a tree graph is well suited for showing the inheritance relationships in the system.

2. **The layout variation:** Starting from the type of the graph, layout variations further customise the actual visualisation. The layout takes into account the choice of the displayed entities and their relationships plus issues like whether the complete graph should fit onto the screen, whether space should be minimised, whether nodes should be sorted, etc.

3. **The metric selection:** Metrics selected from Figure 1 are incorporated into the graph.

2.5. *Graph Types*

We selected a small set of graph types for their simplicity and simple layout algorithms. We list below the essential ones. For a more detailed description of the graphs and a discussion of their advantages and limitations, we refer to [LAN 99].

Name	Layout	Metrics	Sort	Scope
System Complexity	Inheritance Tree	NIV, NOM, LOC,- ,-	NO	Full System
Gives an overview based on the inheritance hierarchies of a whole system. It gives clues on the complexity and structure of the system.				
System Hot Spots	Checker	NOM, NIV, WLOC,- ,-	NO	Very Large System
Displays really simply all the classes according to their number of methods and instance variables.				
Weight Distribution	Histogram	NOM, -, HNL, - ,NOM	NO	Full System
Categorises systems as top-heavy, bottom-heavy or mixed.				
Root Class Detection	Correlation	-, -, -, WNOC, NOC	NO	Very Large Full System
Identifies important classes regarding their impact on their children.				
Service Class Detection	Stapled	NOM, WLOC, NOM,- ,-	width	Full System
Identifies data structure like class containing only read write accessors.				
Cohesion Overview Checker	Checker	NOM, WNAA, NIV,- ,-	width	Full System
Identifies possible god class candidates and highly cohesive classes.				
Hierarchy Carriers	Inheritance tree	WNOC, NOM, WNOC ,- ,-	NO	Subsystem: Inheritance Tree
Identifies classes having a big impact on their subclasses.				
Intermediate Abstract	Inheritance tree	NOM, NMA, NOC,- ,-	NO	Subsystem: Inheritance Tree
Detects abstract classes or nearly-empty classes which are located somewhere in the middle of an inheritance chain. Ideal for Smalltalk.				
Class Size Checker	Checker	LOC, LOC,NIV,-,-	width	Full System
Gives a raw overview of the system in terms of its physical size				
Inheritance Classification	Inheritance Tree	NMA,NMO, NME,-, -	NO	Subsystem
Qualifies inheritance relationships. Subclasses can be only overriding methods, adding functionality or specialising behaviour.				

Figure 2. *List of useful graphs which can be applied on a collection of classes or on complete systems*

Tree. Positions all entities according to some hierarchical relationship (example Figure 1).

Correlation. Positions entities in an orthogonal grid (origin in the upper left corner) according to two measurements. Entities with the same measurement will overlap. Useful for comparing two metrics in large populations (example Figure 7). The reader should note that the graph is not a real correlation graph in the mathematical sense. We overuse this term to convey the fact the entities are placed according to two metrics.

Histogram. Positions nodes along a vertical axis depending on one measurement. Nodes with the same measurement are then positioned in rows, one beside the other. Useful for analysing the distribution within a population.

Checker. Sorts nodes according to a given metric and then places them into several rows in a checkerboard pattern. It is useful for getting a first impression of rather small populations, especially for the relative proportions between measurements

Name	Layout	Metrics	Sort	Scope
Graphs applied to methods				
Identifies possible candidate methods for further refactorings.				
Coding Impact Histogram	Histogram	LOC, -, LOC, LOC, -	width	Small Subsystem or single class
Gives an overview of the class method distribution.				
Method Size Nest Level	Checker	LOC, NOS, MHNL, -, -	NO	Subsystem: Inheritance Tree
Graphs applied to attributes				
Direct Access Attribute Checker	Checker	NAA,NAA,NAA,-,-	NO	Full System
Categorises attributes in terms of their uses.				
Graphs applied to class internals				
Class Confrontation	Confrontation	method(LOC,NOS,LOC) attribute (NAA,NAA,NAA)	NO	Class
Presents class structure in terms of attribute accesses and identifies method clusters.				
Method Size Correlation	Correlation	-, -, LOC, NOS, LOC	NO	Full System
Gives an overview of the system from a method size point of view. It detects empty, strange, long or non-coherently formatted methods.				

Figure 3. *List of useful graphs applicable on methods and attributes*

(example Figure 5).

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

Confrontation. Visualises two different kinds of entities and the relationships between them. It is mainly used for analysing access patterns between attributes and methods. The two kinds of entities are positioned in two separate rows and then edges are drawn to represent the access relationships. Layout variations are achieved by sorting or splitting the rows (example Figure 11).

2.6. *Useful graphs*

Enriching graphs with the set of metrics we presented leads to a huge set of potential graphs out of which only a small number are meaningful. In [LAN 99] we identified some useful graphs, i.e. a combination of a graph type and the metrics necessary to enrich the graph. We present them in Figure 2 and 3. The first two entries are the name and the layout of the useful graph. The third entry describes its metrics described as follows, first the node size, then the node color and the node position: width node metric, height node metric, node color metric, x node metric, y node metric. '-' means that no metric is defined. Then the fourth entry describes if the graph is sorted, and if so according to which metric. The last entry describes the scope of graph.

3. A First Methodology

We applied our approach on several large industrial applications ranging from a system of approximately 1.2 Million lines of C++ from Nokia to a Smalltalk framework of approximately 3000 classes. Our experiments demonstrated the strength of our approach. We were able to quickly gain an overall understanding of the analysed application, identify some problems and point to classes or subsystems for further investigation.

Moreover we learned that the approach is preferably applicable during the first contact with the system, and provides maximum benefits during the one or two first weeks of the reverse engineering phase.

However the approach definitively lacked a methodology that would help a reverse engineer to deploy its full potential. Ideally such a methodology should define which graphs to apply depending on the goal of the reverse engineer, what the paths are between the different graphs, and on what selections the next graphs should be applied.

Difficulties. Such a methodology is difficult to elaborate for the following reasons:

— There is no one unique or ideal path through the graphs.

— Different graphs can be applied at the same stage depending of the analysis of the graphs.

— The decision to apply a graph most of the time depends on some interactions with the current graph.

— The graphs can be applied to different entities implying some back and forth between different graphs.

— A graph displays a system from a certain point of view that emphasises a particular aspect of the system. However, the graph has to be analysed and the code understood to determine if the symptoms revealed by the graph are interesting for further investigation.

The Purpose of a Methodology. We would like to stress that the results of a reverse engineering phase are not a list of problematic classes, even if the identification of possible design defects is a valuable piece of information. Understanding the overall structure of the application, gaining a better understanding of the inheritance relationships between classes, as well as gaining an overview of the methods and the way they are organised should be the result of a reverse engineering phase. We are looking for the bad use as well as the good use of object-oriented design. In that sense, knowing that an inheritance hierarchy is well designed is also valuable information.

Moreover in a reengineering context the fact that a class may have a design problem does not mean that the class should be redesigned. Indeed, if a badly designed class or subsystem accomplishes the work it has been assigned to, without impacting the reengineering process, there is no point in changing it. However, as most of the time reengineers are not the original developers of the system they are maintaining, being aware of such information is still valuable for getting a better mental model of the system.

3.1. *The Methodology Navigation Map*

Each of the graphs presented here produces some *symptoms*, like small dark nodes or wide flat nodes. Such symptoms provide information about the analysed system and also support the choice of the next graphs to further complete this understanding. Graph symptoms point to possible paths that can guide the reverse engineer from graph to graph. Depending on the symptoms the next graph can be applied on the same entities, on a subpart of currently displayed entities or on the structurally containing entities (a class for a method or an attribute). Not all the symptoms are leading to new graphs but also to some specific reengineering actions that represent the next logical step after the detection of defects. For example detecting a “god class” that Riel defines as a class that has grown over the years and has been assigned too many responsibilities, may lead to a split of the class [RIE 96], long methods may be analysed to see if they contain code duplication or be split up if they perform several tasks at the same time [ROB 97, FOW 99].

The presented methodology is based on clusters that group the useful graphs depending on the problem encountered by the reverse engineer and the information provided by the graphs. Each of these clusters is presented in detail in the subsequent section. We identify four clusters:

1. **First Contact**, which provides different overviews of the system.
2. **Inheritance Assessment**, which qualifies the inheritance relationships and the role played by the classes.
3. **Candidate Detection**, which identifies potential class candidates for future investigation.
4. **Class Internals**, which analyses the classes themselves.

Figure 4 presents a map that summarises the main paths between the different the graphs, the clustered graphs and their symptoms.

3.2. *First Contact*

The first thing to do with an unknown system is to gain an overview. We should know how complex the system is and in which way the system is organised. The graphs proposed in this cluster provide answers to the following questions: How is the system composed: only of standalone classes, or of some (maybe deep) inheritance hierarchies? Is the system composed of a lot of small classes and some outliers or only of big classes? What are the biggest entities?

Class Size Checker is based on the metric LOC and provides an overview of a

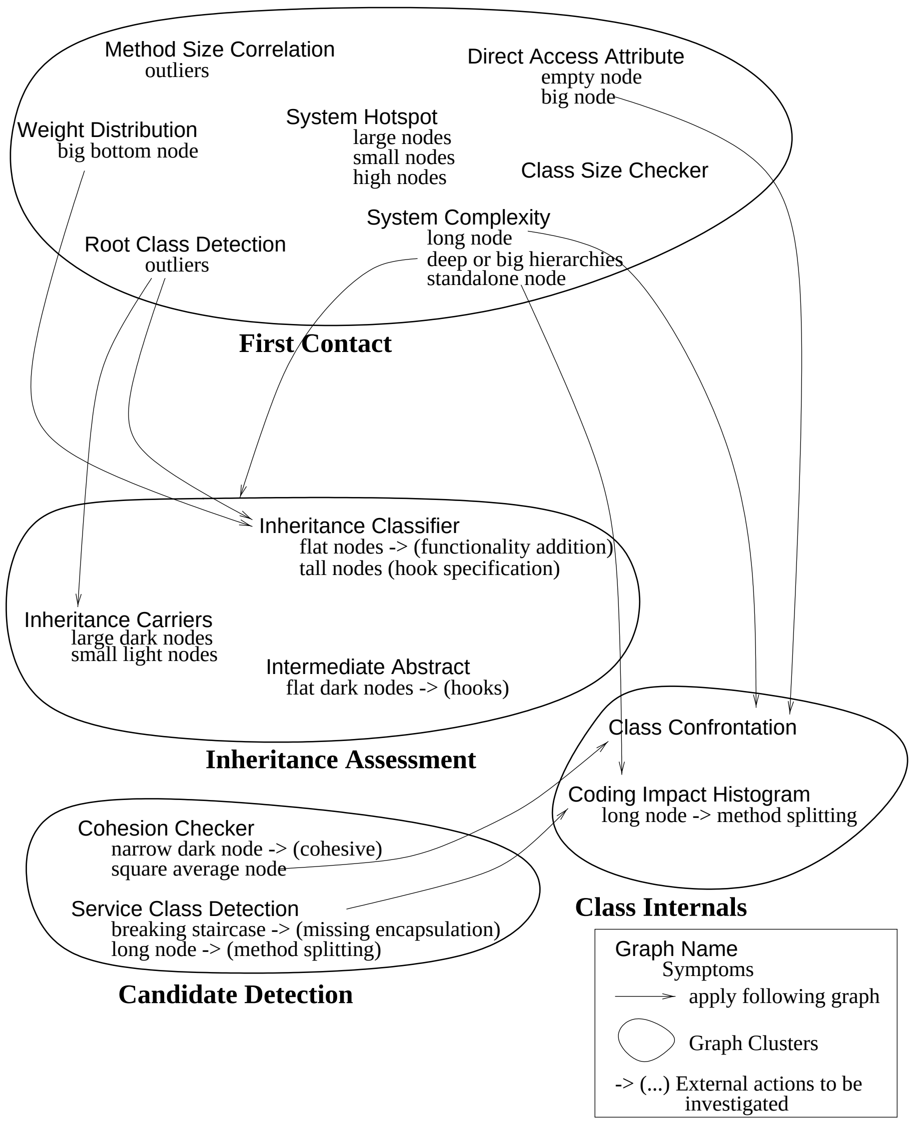

Figure 4. *The methodology is summarised by a navigation map that identifies the graphs and their symptoms and relates the clusters themselves and the graphs*

system application in terms of its raw measure. In particular, it reveals the overall proportion of the classes and allows one to understand if the system is composed by few big classes and lot of small ones or is only composed of average classes (see Figure 5). This graph scales up well to very large systems.

— *Big nodes* represent big classes that may be worth further analysis, and can be investigated by applying **Class Confrontation**.

System Hot Spots helps to identify big and small classes and scales up well to very large systems. It relates the number of methods with the number of attributes of a class.

— *Large nodes* represent voluminous classes that may be further investigated, applying for example **Class Confrontation**.

— *High nodes* are classes that define a lot of methods and few or no attributes. Applying **Inheritance Classification** and **Hierarchy Carriers** may help to understand their relationship with their descendants.

— *Wide nodes* are classes defining a lot of instances variables. When such a class shows a 2:1 size ratio it may represent a class whose main purpose is to be a data structure implementing only get and set methods.

Variations of **Class Size Checker** and **System Hot Spots** are useful to understand how the programmers made use of static behaviour. For example, for Smalltalk code, changing the color of the metaclasses allows us to see if metaclasses have not been overused. Too much behaviour on the static or metaclass side are a sign that the class has too many responsibilities or that it is a facade or a bridge to other classes.

Weight Distribution gives a qualitative overview of a system by categorising it as *top-heavy*, *bottom-heavy* or *mixed*.

— *Big dark nodes at the bottom of the graph* represent classes defining a lot of methods deep inside the hierarchy. Applying **Inheritance Classification** on the hierarchy containing this class helps in understanding the repartition of the methods. Applying **Class Confrontation** on them helps to understand their structure.

System Complexity which is based on the inheritance hierarchies of a whole system gives clues on the complexity and structure of the system (see Figure 6). For really huge systems and depending of the complexity of the inheritance hierarchies, this graph should be applied on subsystems.

— *Tall, narrow nodes* represent classes with few instance variables and a lot of methods. Applying **Class Confrontation** helps to understand their internal structure. When such long nodes appear in a hierarchy, applying **Inheritance Classification** helps to qualify the semantics of the inheritance relationship in which the class is involved.

— *Deep or big hierarchies* are definitively subsystems on which the graphs of the **Inheritance Assessment** cluster help to refine the understanding.

— *Big standalone nodes* represent classes having a lot of attributes and a lot of methods, yet without any subclasses. After further investigation it may be worth to apply **Class Confrontation** to understand the internal structure of the class and learn if the class is well structured or could be decomposed or reorganised.

Method Size Correlation provides a view at the method level (see Figure 7). It helps to assess if the application is composed of classes principally defining data structure and implementing little behaviour or not. By relating the method lines of code with the number of statements in the context of the entire system, it helps to identify empty methods, overlong methods, or methods not consistently formatted according to the method majority.

— *First raw* displays empty methods that can represent commented or hook methods.

— *Outliers* represents methods being too long and that may be the candidates for splitting or methods not following formatting rules adopted by the rest of the application.

Root Class Detection helps in identifying "root" of inheritance tree in the sense of classes having a lot of children. **Root Class Detection** is really useful for really large applications where **System Complexity** may have problem to represent on screen huge and multiple inheritance trees.

— *Outliers* are root classes. In languages like C++ where there is no concept of a common root, root classes and their subclasses are good candidates for all the graphs of the cluster **Inheritance Assessment**.

Direct Access Attribute Checker shows all the attributes of a system represented with the number of times they are accessed. This graph helps to understand if some coding conventions have been applied e.g. whether accessor methods have been used systematically. It helps also to identify violations of such conventions.

— *Small nodes* represent attributes that are not accessed and may point to dead code.

— *Big nodes* represent attributes heavily accessed. These nodes can be the starting point of understanding the functionality and structure of the class defining them.

3.3. *Inheritance Assessment*

Inheritance is a privileged way to structure object-oriented applications. Inheritance relationships are thus an interesting point of view to understand applications. However, inheritance can be used in different ways, like for example the pure addition of functionality in the subclasses or specialisation of the root class functionality. The graphs defined in this cluster help in the analysis of inheritance related aspects, e.g. identifying classes which have a large impact on their subclasses, identifying nearly empty classes or qualifying the inheritance relationships.

Inheritance Classification qualifies the inheritance relationships by displaying the amount of added methods relative to the number of overridden methods and the extended methods (see Figure 8).

— *Flat light nodes* represent classes where a lot of methods have been added but where few methods have been overridden or extended. In such a case, the semantic of the inheritance relationship is an addition of functionality.

— *Tall nodes* represent classes where a lot of methods have been overridden. They can represent classes that have specialised hook methods. When the nodes are dark this means that a lot of method have been extended.

Hierarchy Carriers helps to detect classes having a certain impact on their subclasses (see Figure 9).

— *Large dark nodes* represent classes that define a lot of behaviour and have a lot of descendants. They represent classes having a certain impact on the application.

— *Flat light nodes* represent classes defining little behaviour but shared by a lot of descendants. They can represent the ideal place to factor out code from the subclasses to the superclass.

Intermediate Abstract identifies classes that are nearly empty in the middle of a hierarchy.

— *Flat dark nodes* represent classes having few methods being added but that have subclasses. Further investigation may reveal that the classes implement some abstract behaviour.

3.4. *Candidate Detection*

This group of graphs helps in identifying potential classes for further analysis or refactorings, e.g. data structure like classes potentially needing a better encapsulation or interface, long methods or “god classes” [RIE 96] needing to be split.

Cohesion Overview Checker presents the number of methods, the total number of attributes accesses and the number of attributes of all the classes of a system (see Figure 10). It helps in identifying on one hand highly cohesive classes and on the other hand possible “god classes”.

— *Narrow and high dark nodes* represent classes having few methods which heavily access the attributes defined in the classes.

— *Square average node* represent medium to big classes with proportional attribute accesses. These classes may be further investigated by applying **Class Confrontation**.

Service Class Detection is based on relating the number of methods and the lines of code of a class and interpreting this information in the context of the complete application.

— *Long nodes* represents classes having long methods compared to the application context.

— *Breaking staircase effect nodes* represent nodes that given a certain number of methods do not have the expected length in terms of LOC. Such classes often lack from encapsulation and are service classes. Service classes may point to sets of coupled classes being brittle to changes.

3.5. *Class Internals*

The graphs of this cluster provide insight on the classes or small subsystems themselves.

Coding Impact Histogram reveals the shape of a class or a subsystem in terms of the methods size distribution. Methods that are too long or strangely structured classes are identified.

Class Confrontation reveals the structure of a class in terms of method use and attribute accesses by showing the dependencies between the methods and the attributes (see Figure 11). It detects classes which are candidates for splitting into smaller and more encapsulated classes.

4. Case study

“*The primary purpose of **reverse engineering** a software system is to increase the overall comprehensibility of the system for both maintenance and new development.*”[CHI 90]

Introduction. Before applying our methodology on the case study, we list the results that we expect to obtain. Here are our main expectations:

— Gain an overview of the system in terms of number of classes and their proportion.

— Locate and understand the most important class and inheritance hierarchies.

— Assess the overall quality of the system.

— Identify exceptional classes in terms of size and/or complexity.

— Comprehend the internal organisation of the classes.

— Identify the possible use of design patterns or occasions where design patterns could be used.

Reporting about the case study. In this section we illustrate the proposed methodology by showing a selection of the graphs we obtained while reverse engineering an industrial system.

Reporting about a case study is quite difficult without sacrificing the exploratory nature of the approach. Indeed, the idea is that different graphs provide different yet complementary perspectives. Consequently, a concrete reverse engineering strategy should be to apply the graphs in some specific order, although the exact order would vary depending on the kind of system at hand and the kind of questions driving the reverse engineering project. Therefore, readers should read the case study report as one possible use case, keeping in mind that reverse engineers must customise their approach to a particular reverse engineering project. For the sake of simplicity, we chose to present some selected useful graphs following the presented clusters and not a possible path through the graphs.

Some Facts about the Case Study. The system we report on here is VisualWorks 3.0 [HOW 95], an industrial framework developed in Smalltalk that consists of 528 classes (not counting the meta-classes), 10794 methods, 1674 attributes, 535 inheritance relationships, 32591 method invocations and 9537 attribute accesses.

VisualWorks is a good case study because (a) it is freely available so the results presented here can be reproduced and (b) it is a complex system large enough to show the benefits of our approach. Note that besides this case study, we have run other experiments on industrial systems implemented in C++ and Smalltalk. However due to non-disclosure agreements, we cannot publish the results of these experiments.

4.1. *First Contact Cluster*

As this cluster contains more graphs than the other, we present three different and complementary graphs that provide overviews of the case study.

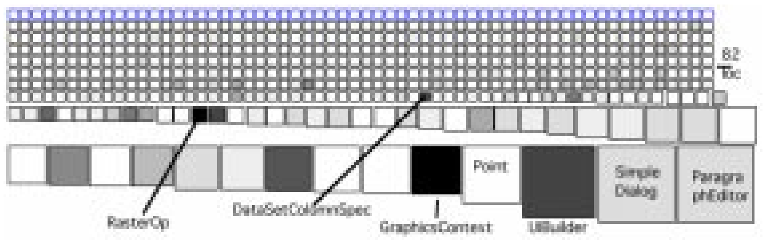

Figure 5. *First Contact: Class Size Checker; node size = LOC, and colour = NIV*

Class Size Checker. One of the first impressions of the system that reverse engineers desire is a feeling for the raw physical measures of a system. For that purpose, we generate a **Class Size Checker** (see Figure 5). Checker graphs are useful for showing relative proportions between the system elements. In this particular case it shows the proportions among the classes of the software system in terms of lines of code. Through sorting, it is easy to identify the largest and smallest classes.

Interpretation. In the displayed graph, we remove the class Object and a class called InverseColorMapInitializer that contains 72 attributes to get a better distribution of the colour metrics over the other nodes. In this graph we see there are many very small classes (around 400) and that there are some empty classes positioned on the upper left corner, and also a great number of big classes not bigger than 2000 LOC. T The empty classes are only visible on the screen and not in the paper version, because metric measurements equal to zero render the nodes with a blue border. The biggest one with 2087 lines of code is the class ParagraphEditor. We see there are many others which are also very big and have sizes around 2000 lines of code. In decreasing order: ParagraphEditor (2087 LOC), SimpleDialog (2068), UIBuilder (1959), Point (1572). Then the darker nodes represents classes having a lot of attributes: this is the case of

GraphicsContext (1302 LOC and 22 attributes) and some smaller classes RasterOp (422 LOC and 22 attributes) and DataSetColumnSpec (247 LOC and 19 attributes).

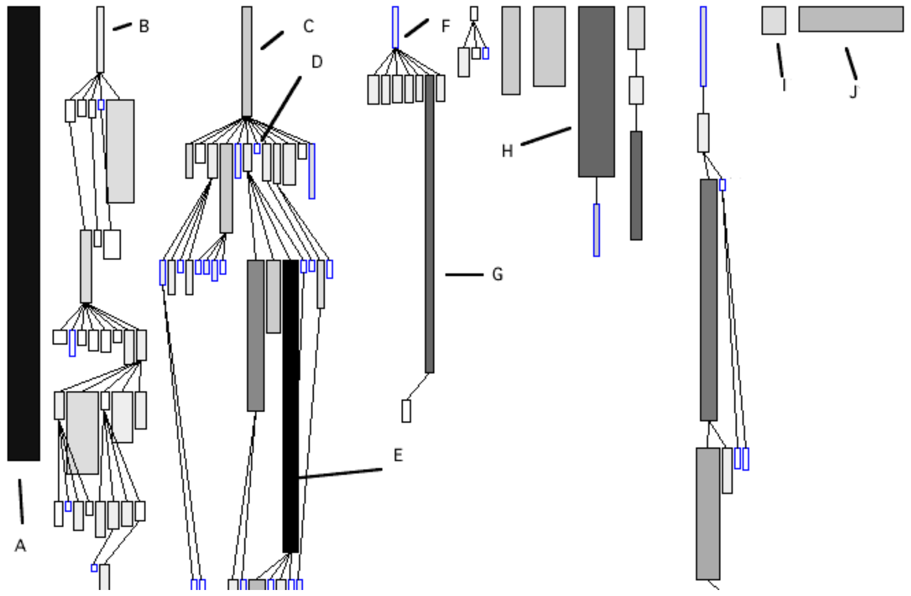

Figure 6. *First Contact: System Complexity; node width = NIV, height = NOM and colour = LOC*

System Complexity. To assess the size and complexity of the system, we apply **System Complexity** on the complete system. The fact that in Smalltalk all the classes inherit from the class Object blurs the identification of relevant inheritance trees. That's why to get a better understanding of the inheritance hierarchies we removed the class Object. Due to the space limits of the paper, we selected some interesting hierarchies. Figure 6 presents then a partial view that we obtained.

Interpretation. The system is composed by several inheritance trees of various size (B is the inheritance tree of UISpecification, C the one of Controller) and some standalone classes (A, K and L). The big left standalone class (A) is the UIBuilder class that is one of the cornerstones of the framework as mentioned in [HOW 95]. This class implements a lot of functionality. The second hierarchy from the left (B) is the UISpecification hierarchy that represents all the functionality related to the way the widget are specified. The third left inheritance graph (C) is the inheritance of the class Controller of the MVC triad. The big dark node (E) is the class ParagraphEditor with 2087 LOC. The small class D, named NoController, with only three methods is the result of the application of the Null Object pattern [MAR 98b]. It represents the default behaviour of not having a controller. The fourth inheritance tree (F) is a part of the Geometric class tree. The big narrow node (G) is Rectangle. Its big relative size in terms of methods is confirmed by the use of Rectangle in Smalltalk. H represents the class GraphicsContext a nearly standalone class. The two rightmost big standalone classes are InverseCoplorMapInitializer (J) with 72 attributes and InverseColorMap (I) with 16 attributes. After manual browsing, we identified that the first class is in fact an algorithm that uses attributes as temporary variables and that it is used by the second class.

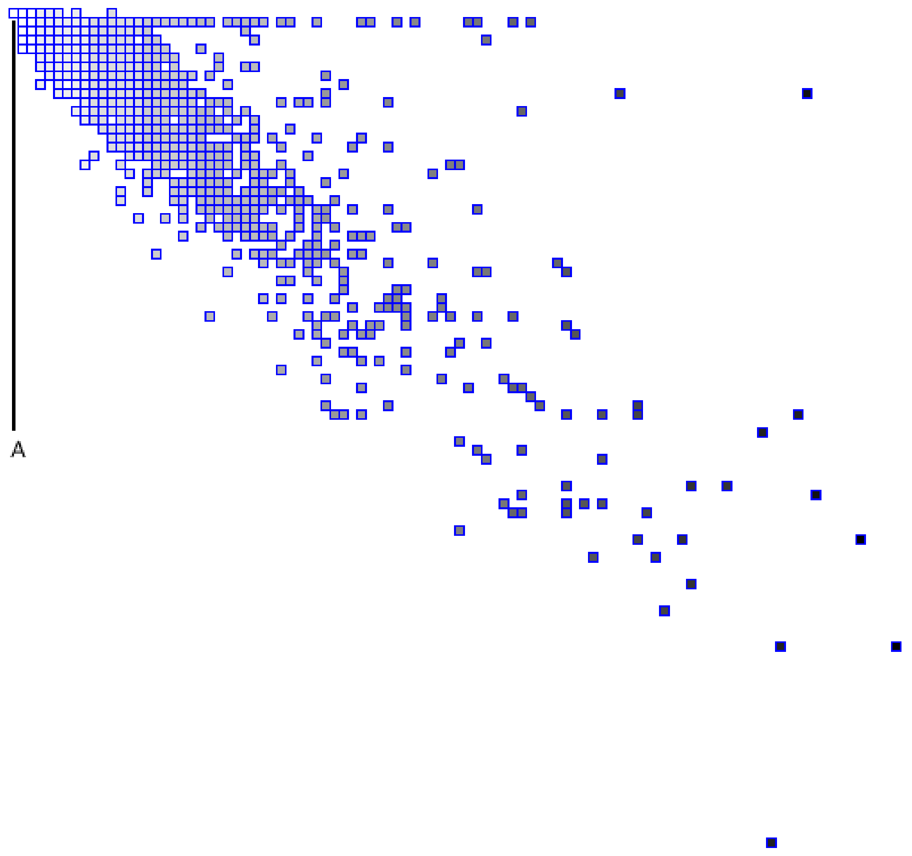

Figure 7. *First Contact: Method Size Correlation; node position = (X = LOC, Y = NOS) and colour = LOC*

Method Size Correlation. While a histogram is good to get a feeling for the distribution of system elements according to one metric, it is not optimal for analysing a system as it emphasises a single metric only. In contrast, the correlation graph is one of the graphs that may render up to five different metrics per node at the same time. Nevertheless, we usually restrict ourselves to three metrics to achieve the effect of all nodes having uniform size for better understanding. We apply **Method Size Correla-**

tion on the full system and obtain Figure 7. Note that in a correlation graph, the nodes may overlay each other, which is the case for many nodes in the upper left corner. However, this is not a problem as only the outliers are interesting to us.

Interpretation. The first line of nodes are interesting because their bodies contain no statement at all. Most of them represent hooks methods like hasFocus or performUpdate methods. Some of them are positioned there because they contain only comments (remember that we count comment lines as well). The rightmost nodes on the second line represent methods having a lot of lines of code but few statements (one in this case). In this system they represent most of the time primitives on classes Screen, GraphicsDevice that included quite a lot of comments. The left most top most node is a primitive with up to 14 arguments.

The other interesting nodes are the ones on the outer edges of the correlation graph. Looking at these methods helps to understand what are the classes performing some costly operations. Some of such a method are potential for a split like print:on:using: of the class NumberPrintPolicy that is 86 LOC long. However one of the danger with this graph is to dive too much into the details.

Another insight which can come from this graph is a general assessment of the system. During our experiments we have seen that the methods tend to align themselves along a certain correlation axis. Depending on the age of the system the axis changes its angle with time: methods are written and corrected all time, slowly getting messy with many statements on few lines.

4.2. *Inheritance Assessment Cluster*

We present two complementary graphs that help to understand inheritance relationships.

Inheritance Classification. Inheritance is a key mechanism of object-oriented programming. That's why having a better understanding of the inheritance relationships between classes is important. By changing the shape and the colour of the nodes according to their numbers of methods added, overriding or extended, **Inheritance Classification** provides such information. Figure 8 presents the results we obtained on the class VisualComponent that is crucial in the VisualWorks system.

Interpretation. Such a graph contains three kind of nodes: the horizontal ones, the vertical ones and the small blue square ones. Each of them describes a different kind of inheritance relationship.

A horizontal box means that a lot of method have been added. For example, the classes Image, TableView, SelectionView are adding new behaviour. Classes WidgetWrapper, DataSetView and VisualPart are adding and overriding behaviour.

A vertical box says that more methods have been overridden than added. This is the case of the class Wrapper. Indeed this class is a Decorator that is why it contains a lot of extended and overridden methods and few added ones [ALP 98].

Small blue square nodes identify classes that potentially are not defining new methods, thus only specialising the superclass behaviour. We have still to check that the first measurement is really zero. This is mainly the case for the classes denoted by A. Note that this shows that the hierarchy is good.

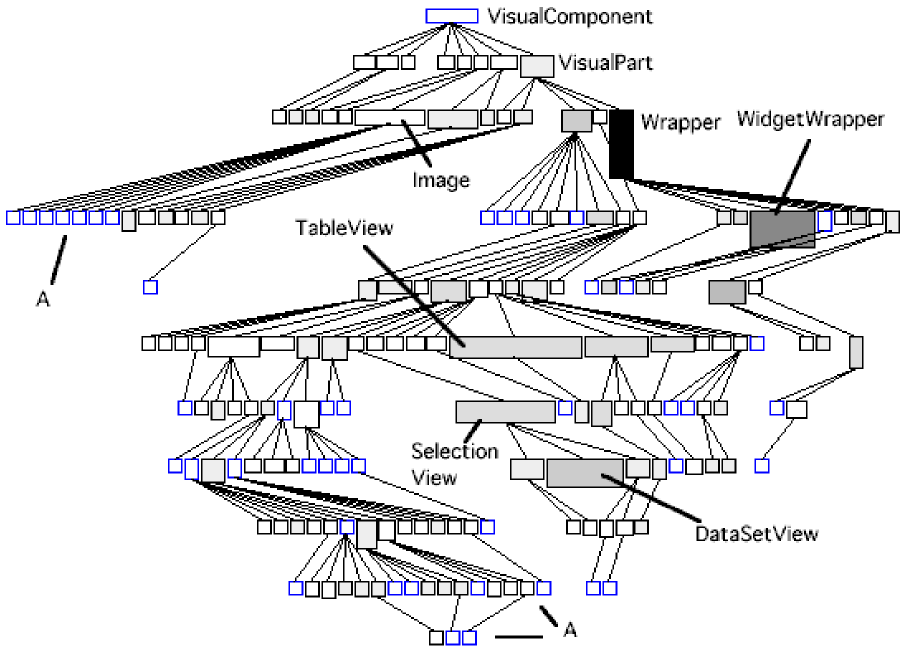

Figure 8. *Inheritance Assessment: Inheritance Classification; width node =NMA, height node = NMO and colour = NME*

Hierarchy Carriers. Inheritance being the essential mechanism of reuse, it is interesting to identify classes having a big impact on their subclasses. Figure 9 is the application of **Hierarchy Carriers** on the previous inheritance tree.

Interpretation. Large square or large horizontal rectangle nodes represent classes having a lot of children while in the same time defining a lot of methods. Such classes share code among a large set of classes. For example, the classes VisualComponent, VisualPart, DependentPart, View and SimpleView defines important behaviour (like associating a model and a controller to the visual element) that is shared among sub-hierarchies.

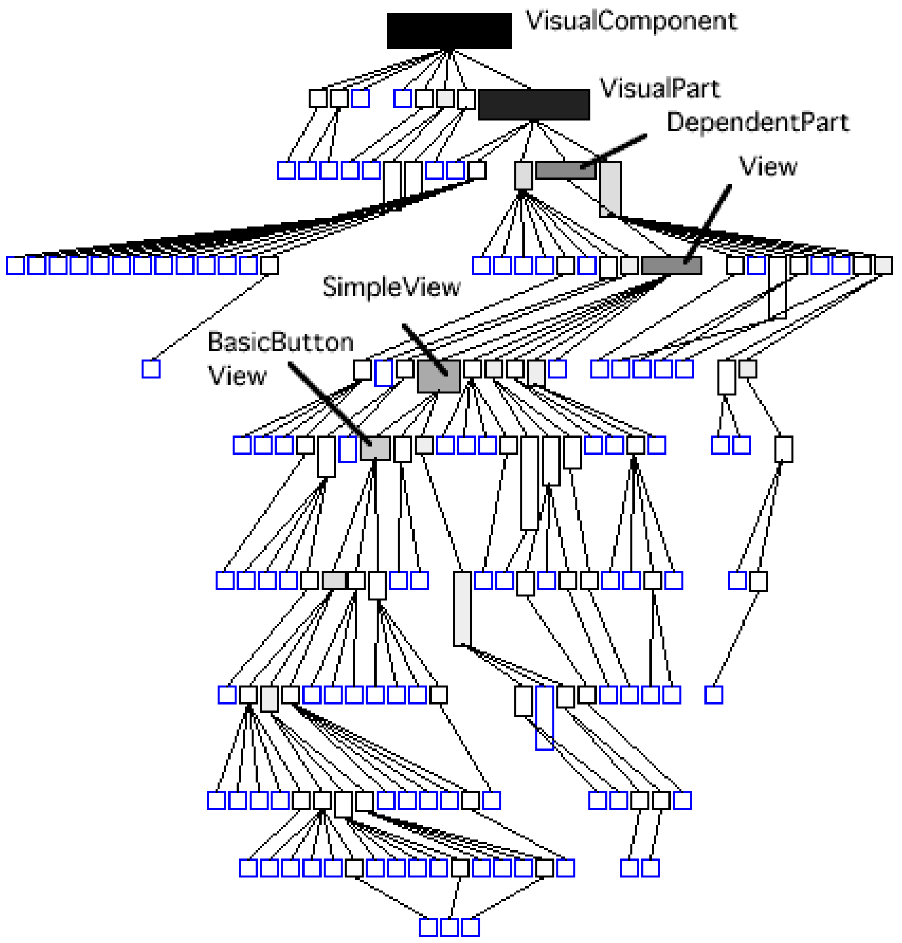

Figure 9. *Inheritance Assessment: Hierarchy Carriers; node width = WNOC, node height = NOM and colour = WNOC*

4.3. *Candidate Detection Cluster*

Cohesion Overview Checker. One of the generic principles in designing software systems is to maximise the cohesion within and minimise the coupling between the systems components. **Cohesion Overview Checker** helps in assessing if the classes are cohesive or possible god classes. Such classes are candidates for splitting.

Interpretation. In Figure 10, we see some narrow nodes like RasterOp, Controller that are cohesive. Narrow nodes means classes having few methods accessing a lot the attributes. Applying **Class Confrontation** on these classes produces dense graphs

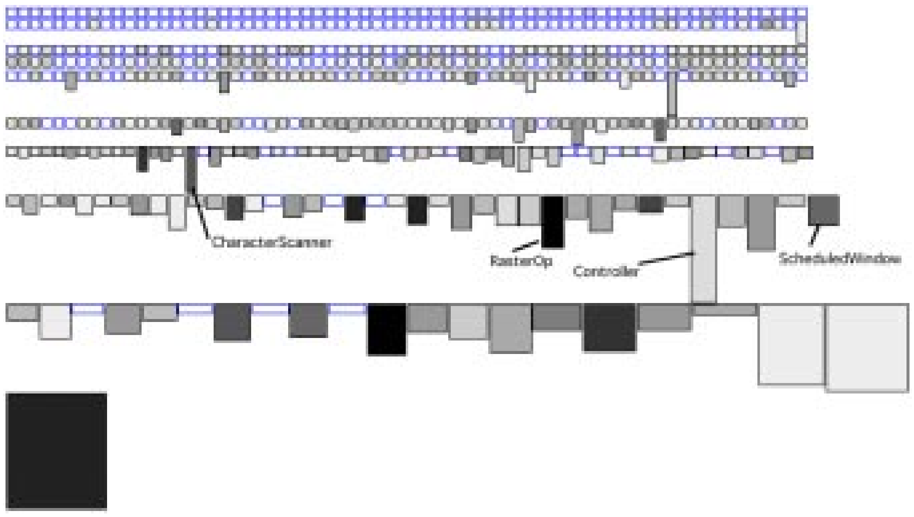

Figure 10. *Candidate Detection: using Cohesion Overview Checker to assess class cohesion; width node = NOM, height = WNAA, colour = NIV.*

where the attributes are heavily accessed by the methods. Flat light nodes, like the last bottom node of the graph, represent classes having few attributes, then few accesses to these attributes and a lot of methods.

The other interesting kind of nodes are the approximately average squared nodes of average colour. These nodes are potential god classes[RIE 96]. Applying **Class Confrontation** on these classes may lead to identify classes needing to be split. Figure 11 shows the application **Class Confrontation** on the node representing the class ScheduledWindow.

The current graph interpretation is limited by the fact that this graph only takes into account the direct accesses to attributes. This means that it cannot detect god classes or non-cohesive classes that are using accessors instead of directly accessing the attributes.

4.4. *Class Internals Cluster*

For this cluster we present the application of **Class Confrontation** as the following step of the application of **Cohesion Overview Checker** (see 4.3).

Class Confrontation. After applying the **Cohesion Overview Checker** graph, we apply a **Class Confrontation** on the class ScheduledWindow (in Figure 10) to gain a better understanding of its internal structure.

Interpretation. As shown by the Figure 11 we identify two clusters of methods accessing two distinct sets of attributes. This suggests that the class could be split. Due

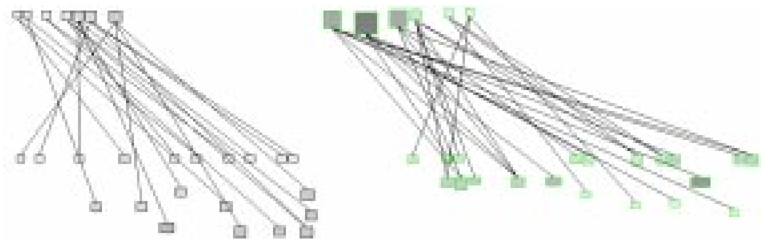

Figure 11. *Class Internals: Focus on Class Cohesion of ScheduledWindow using a Class Confrontation; method node height and width = NOS and colour = LOC; attribute node height and width and colour = NAA. The top nodes represents attributes, the bottom nodes the methods and the edges the attribute accesses.*

to the good state of VisualWorks we were surprised to find such a situation. In fact the left group of attributes deals with internal aspects such as the minimum size, maximum size, label... of the window, whereas the right set of attributes represents the connection with the external objects such as the controller, the component the master window and the model associated with the window.

4.5. *Case study evaluation*

Using a hybrid approach combining metrics and graph layouts on this industrial framework provided us with an initial understanding and identify some of the key classes without having to dive into the details. During the time we allocated to the system understanding, three days in the present case, the methodology guided us to steer our tasks by structuring the possible branches we could follow depending on the various signs the graph revealed. One of the major problems with large systems is to get an overview and get some initial understanding without getting lost in their intrinsic complexity. The clusters helped to stay focus at the different levels of understanding that we wanted to gain, and we were able to maintain a breath first overview instead of a deep first that would have led us to deal with too detailed understanding.

In the three days we allocated to this experiment we also have been able to:

— See that the framework is composed from a few big classes like UIBuilder, and ApplicationModel and a lot of small classes.

— See that metaclasses were used in a good way. Metaclass methods implement class behavior and not other services.

— Identify key hierarchies like the VisualComponent one in which we found the application of the Decorator pattern.

— Identify strange classes merely implementing algorithms.

— Identify many very long methods.

— Understanding the relative importance of key classes like the Controller class.

Possible Improvements. Up to now the graphs defined do not handle coupling or collaboration between classes. We missed this aspect while trying to understand how classes in different hierarchies were linked or when trying to understand what class represent configuration object or a central place in the system. Moreover, the understanding of the class internals is limited in the sense that it only shows attribute accesses and that method invocations is not really presented in a meaningful way. Improving these aspects is one of our major goals in the future.

5. Tool Support: Moose and CodeCrawler

CodeCrawler is the tool that we developed to identify the useful graphs and to validate our methodology. CodeCrawler is freely available at http://www.iam.unibe-.ch/∼famoos/. CodeCrawler is based on Moose for the source code entity meta model and the metric extraction, and on the HotDraw framework [JOH 92] for its visualisation.

Moose is a language independent and extensible reengineering environment built in VisualWorks Smalltalk and it has the following characteristics:

— Language independence – Moose supports reengineering of applications developed in different object-oriented languages such as Java, Smalltalk and C++, as its core model is *language independent* [TIC 98].

— Extensible – New entities like associations or groups can be added.

— Analysis support – Moose supports reengineering by providing facilities for analysing and storing multiple models, for refactoring and support for analysis methods such as metrics and inference of source code entity properties.

Measurements and Metrics in Moose. Moose defines a large set of metrics which is constantly being extended. The supported metrics are mainly simple metrics or measurements that are easily extractable from source code and whose definition is relatively simple as shown in Table 2.3. Moreover Moose supports language independent as well as language dependent metrics. By language dependent we mean metrics which can be defined only for a certain language. For example NOMP, the number of method protocols of a class, is meaningful for Smalltalk classes.

CodeCrawler. Being based on Moose, CodeCrawler is language independent – it has been used to validate our approach during the reverse engineering of applications written in Java, C++ and Smalltalk, extensible from the entity and metric point of view– we are currently extending the approach to introduce groups of entities (like module or package) and to define new metrics for such entities.

6. Related work

Program visualisation. Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids. Various tools provide quite different program visualisations: Graphtrace [KLE 88], Rigi[Mül 86], Hy+ [CON 92, CON 93], SeeSoft [BAL 96], ISVIS [JER 97]and JInsight [PAU 93].

Powerful algorithms have been developed to support such huge visual program representations: the Sugiyama algorithm to optimise hierarchical layouts [SUG 81], hyperbolic geometry to navigate through large hierarchies [LAM 95], Shrimp views to optimise layouts in general [STO 95], libraries providing ranges of algorithms [SAN 96], ternary diagrams to track dynamic interactions between system modules [HAY 97], mural techniques to provide large overviews [BAL 96], [JER 97].

Metrics. Metrics have long been studied as a way to assess the quality of large software systems [FEN 97] and recently this has been applied to object-oriented systems as well [MAY 96], [KON 97], [MAR 98a], [LEW 98], [NES 88]. However, a simple measurement is not sufficient to assess such complex thing as software quality [HEN 96], not to mention the reliability of the results [DEM 99a].

Some of the metric tools visualise information via typical algorithms for statistical data, such as histograms and Kiviat diagrams. Datrix [MAY 96], TAC++ [FIO 98a, FIO 98b], and Crocodile [LEW 98] are tools that exhibit such visualisation features. However, in all these approaches, the visualisation is a mere side-effect of having a lot of numbers to analyse. In our approach, the visualisation is an inherent part of the approach, hence we do not visualise numbers but constructs as they occur in source code.

Methology. To the best of our knowledge none of the approaches we referenced elaborate a methodology. Most of the times, metrics definitions, tools or techniques are indeed presented, but no process or methodology is discussed with which the tools or techniques can be applied.

7. Conclusion

We presented a hybrid approach for the reverse engineering of large object-oriented systems. It combines the immediate appeal of program visualisation with the scalability of metrics to support program understanding and problem detection. However, contrary to other researches that limit themselves to present metrics definition or tool functionality, we elaborated a methodology to guide the reengineers. The contributions of this paper are:

— The elaboration of a methodology that helps to apply and analyse the graphs we identified in our previous work. The methodology is based on the definition of simple graphs that once applied act as revealer by showing or not symptoms that the reengineer should check in the code. The methodology is based on the definition of clusters that group graphs depending on the goal of the reengineer. The methodology also states the possible paths from the graphs themselves or from the clusters depending on the symptoms revealed by the graph application.

— The methodology supports the reverse engineer process by helping to map some simple goals to some specific graph application e.g. for example which are the graphs that support class internal understanding or system overview.

— The presentation of an industrial case study where we show that the methodology helped us to understand differents aspects of the case study, e.g. the overview of the application, the qualification of inheritance, the presence of design patterns.

Note that the definition of new graphs expressing the coupling between entities (classes, methods, modules, applications...) could improve the presented methodology. Such new graphs may lead to the definition of new clusters and paths between graphs. This is part of our future work.

Acknowledgements

This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT program Project no. 21975.

8. Bibliography

[ALP 98] ALPERT S. R., BROWN K. et WOOLF B., *Design Patterns in Smalltalk*. Addison-Wesley, 1998.

[BAL 96] BALL T. et EICK S., « Software Visualization in the Large ». *IEEE Computer*, p. 33–43, 1996.

[CAS 98] CASAIS E., « Re-Engineering Object-Oriented Legacy Systems ». *Journal of Object-Oriented Programming*, vol. 10, n^{o} 8, p. 45–52, January 1998.

[CHI 90] CHIKOFSKY E. et CROSS II J., « Reverse Engineering and Design Recovery: A Taxonomy ». *IEEE Software Engineering*, p. 13–17, January 1990.

[CON 92] CONSENS M., MENDELZON A. et RYMAN A., « Visualizing and Querying Software Structures ». In *Proceedings of the 14th International Conference on Software Engineering*, p. 138–156, 1992.

[CON 93] CONSENS M. et MENDELZON A., « Hy+: A Hygraph-based Query and Visualisation System ». In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, p. 511–516, 1993.

[DEM 99a] DEMEYER S. et DUCASSE S., « Metrics, Do They Really Help? ». In MALENFANT J., Ed., *Proceedings LMO'99 (Languages et Modèles à Objets)*, p. 69–82. HERMES Science Publications, Paris, 1999.

[DEM 99b] DEMEYER S., DUCASSE S. et LANZA M., « A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization ». In BALMAS F., BLAHA M. et RUGABER S., Eds., *WCRE'99 Proceedings (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.

[FEN 97] FENTON N. et PFLEEGER S. L., *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second édition, 1997.

[FIO 98a] FIORAVANTI F., NESI P. et PERLI S., « Assessment of System Evolution Through Characterization ». In *ICSE'98 Proceedings (International Conference on Software Engineering)*. IEEE Computer Society, 1998.

[FIO 98b] FIORAVANTI F., NESI P. et PERLI S., « A Tool for Process and Product Assessment of C++ Applications ». In *CSMR'98 Proceedings (Euromicro Conference on Software Maintenance and Reengineering)*. IEEE Computer Society, 1998.

[FOW 99] FOWLER M., *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[HAY 97] HAYNES P., MENZIES T. et COHEN R., « *Software Visualization* », Chapitre Visualisations of Large Object-Oriented Systems. World-Scientific, 1997.

[HEN 96] HENDERSON-SELLERS B., *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[HOW 95] HOWARD T., *The Smalltalk Developer's Guide to VisualWorks*. SIGS Books, 1995.

[JER 97] JERDING D. et RUGABER S., « Using Visualization for Architectural Localization and Extraction ». In BAXTER I., QUILICI A. et VERHOEF C., Eds., *Proceedings Fourth Working Conference on Reverse Engineering*, p. 56 – 65. IEEE Computer Society, 1997.

[JOH 92] JOHNSON R. E., « Documenting Frameworks using Patterns ». In *Proceedings OOPSLA '92 ACM SIGPLAN Notices*, p. 63–76, October 1992.

[KLE 88] KLEYN M. F. et GINGRICH P. C., « GraphTrace – Understanding Object-Oriented Systems Using Concurrently Animated Views ». In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, p. 191–205, November 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.

[KON 97] KONTOGIANNIS K., « Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics ». In BAXTER I., QUILICI A. et VERHOEF C., Eds., *Proceedings Fourth Working Conference on Reverse Engineering*, p. 44 – 54. IEEE Computer Society, 1997.

[LAM 95] LAMPING J., RAO R. et PIROLLI P., « A Focus + Context Technique Based on Hyperbolic Geometry for Visualising Larges Hierarchies ». In *Proceedings of CHI'95*, 1995.

[LAN 99] LANZA M., « Combining Metrics And Graphs for Object Oriented Reverse Engineering ». Master's thesis, University of Bern, 1999.

[LEW 98] LEWERENTZ C. et SIMON F., « A Product Metrics Tool Integrated into a Software Development Environment ». In *Object-Oriented Technology Ecoop'98 Workshop Reader*, LNCS 1543, p. 256–257, 1998.

[LOR 94] LORENZ M. et KIDD J., *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[Mül 86] MÜLLER H., « *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications* ». PhD thesis, Rice University, 1986.

[MAR 98a] MARINESCU R., « Using Object-Oriented Metrics for Automatic Design Flaws in Large Scale Systems ». In *Object-Oriented Technology Ecoop'98 Workshop Reader*, LNCS 1543, p. 252–253, 1998.

[MAR 98b] MARTIN R., RIEHLE D. et BUSCHMANN F., Eds., *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[MAY 96] MAYRAND J., LEBLANC C. et MERLO E., « Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics ». In *International Conference on Software System Using Metrics*, p. 244–253, 1996.

[NES 88] NESI P., « Managing OO Project Better ». *IEEE Software*, July 1988.

[PAU 93] PAUW W. D., HELM R., KIMELMAN D. et VLISSIDES J., « Visualizing the Behavior of Object-Oriented Systems ». In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, p. 326–337, October 1993.

[RIE 96] RIEL A. J., *Object Oriented Design Heuristics*. Addison-Wesley, 1996.

[ROB 97] ROBERTS D., BRANT J. et JOHNSON R. E., « A Refactoring Tool for Smalltalk ». *Theory and Practice of Object Systems (TAPOS)*, vol. 3, n^{o} 4, p. 253–263, 1997.

[SAN 96] SANDER G., « Graph Layout for Applications in Compiler Construction ». Rapport technique, Universitaet des Saarlandes, February 1996.

[STO 95] STOREY M.-A. D. et MÜLLER. H. A., « Manipulating and documenting software structures using SHriMP views ». In *Proceedings of the 1995 International Conference on Software Maintenance*, 1995.

[SUG 81] SUGIYAMA K., TAGAWA S. et TODA M., « Methods for Visual Understanding of Hierarchical System Structures ». *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-11, n^{o} 2, February 1981.

[TIC 98] TICHELAAR S. et DEMEYER S., « An Exchange Model for Reengineering Tools ». In DEMEYER S. et BOSCH J., Eds., *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543. Springer-Verlag, July 1998.

Article reçu le 18 Novembre 1999
Version révisée le 6 Juin 2000
Rédacteur responsable : Christophe DONY

***Stéphane Ducasse** is first assistant at the University of Bern in the Software Composition Group led by Prof. Nierstrasz. Since 1996, he has a PhD from the I3S laboratory of the University of Nice-Sophia Antipolis. He likes teaching and supervising students. His research interests are: metaprogramming, reflective systems and languages, meta object protocol definitions, component specification and composition, metamodeling, frameworks building, code understanding and analysis, refactorings, object oriented reverse engineering and reengineering, and teaching programming to kids. He participated to the FAMOOS Esprit project and is currently participating to the PECOS IST project. He is involved in the Smalltalk and Squeak communities.*

Michele Lanza *is currently doing his Ph.D. at the University of Bern in the Software Composition Group led by Professor Nierstrasz. His main interests lie in reengineering and reverse engineering as well as program understanding. He likes programming and software engineering in general, although at the same time he is trying to keep an open view on things for the sake of personal flexibililty. He got involved in software reengineering by participating to the FAMOOS Esprit Project. At this time he is shifting his focus in research towards software evolution.*