Run-Time Information Visualization for Understanding Object-Oriented Systems

Roland Bertuli
I3S Laboratory
Sophia-Antipolis, France
bertuli@essi.fr

Stéphane Ducasse
Software Composition Group
University of Bern, Switzerland
ducasse@iam.unibe.ch

Michele Lanza
Software Composition Group
University of Bern, Switzerland
lanza@iam.unibe.ch

Abstract

Understanding object-oriented legacy systems is a complex task exacerbated by the presence of late binding and polymorphism. Moreover, the metaphor of message sending and the anthropomorphism promoted by object-oriented languages makes it difficult to statically identify the precise role the objects play at run-time. We propose a lightweight visualization approach enriched with run-time information which allows us to identify precise aspects of the objects lifetime such as the role played in the creation of other objects and the communication architecture they support. Our approach not only supports the run-time understanding of an application but also allows one to evaluate test understanding and test coverage.

Keywords: software visualization, reverse engineering, reengineering, dynamic information, object-oriented programming, program understanding

1. Introduction

Corbi [4] reported that during maintenance professionals spend at least half of their time analyzing software to understand it. Moreover, Sommerville [28] and Davis [5] estimate that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system. These facts show that understanding applications is one of the hardest tasks in the maintenance of software systems.

Nowadays these *legacy systems* are not only limited to procedural languages but are also written in object-oriented languages [7]. This situation has exacerbated the problems of understanding since in object-oriented systems the domain model of the application is distributed across the whole system and the behavior is distributed across inheritance hierarchies with late-binding [30] [2] [7].

Many approaches to help the understanding of object-oriented systems make use of static information. The information gathered this way is very valuable to understand the structure and the design of a system, but reveals nothing about the behavior of the system at run-time. In order to do so, people instrument the source code using various techniques (method wrapping, logging, etc.), and then run the system. The instrumented source code can be used to generate a *trace*, which contains information about the run-time behavior of the system. Such a trace typically contains information about which method is calling which other method, which objects are created at which time, etc.

The problem is that the low-level nature of the information contained in such a trace makes it hard for a software engineer to infer higher-level information about a software system. For example he may want to know which other objects a certain object is sending messages to, but he does not want to have to analyze and verify every single method invocation. Indeed, many approaches based on run-time information have as primary goal to reduce the complexity of the trace and to reveal certain aspects like collaboration between classes [27].

Our solution is based on lightweight visualizations enriched with measurements that we collect during the generation of run-time information [6] [22]. We collect the measurements by analyzing the run-time information and enriching a model of a software application with these measurements. We then use these measurements to enrich our visualizations in order to obtain a better understanding of the application's run-time behaviour.

2. Problems

Wilde and Huitt assessed that understanding an object-oriented application is difficult because of several reasons, such as:

- Polymorphism and late-binding make traditional tool analyzers like program slicers inadequate. Data-flow analyzers are more complex to build especially in presence of dynamically typed languages.

- The use of inheritance and incremental class definitions, together with the dynamic semantics of *self* and *this*, make applications more difficult to understand.
- The domain model of the applications is spread over classes residing in different hierarchies and/or subsystems and it is difficult to pinpoint the location of a certain functionality.
- Contrary to procedural systems, where a top-down reverse engineering approach can work because of the structured decomposition of an application, in the case of object-oriented systems the first question a reverse engineer has to answer is where to start the reverse engineering process.

Moreover, in an run-time context, De Pauw states that *"Numerous classes, complex inheritance and containment hierarchies, and diverse patterns of dynamic interaction all contribute to difficulties in understanding, reusing, debugging, and tuning large object-oriented systems"* [24].

Indeed, understanding an application written in an object-oriented language is a difficult task. Using dynamic information is one way to support the understanding process. In such a context the essential questions that have to be answered are the following ones:

- What are the most instantiated classes?
- What are the classes having tenured objects? From an architectural point of view having a singleton is also an important information.
- What are the classes that create objects? Detecting factories is important information.
- How do classes communicate with each other?
- Which percentage of the methods defined in a class are actually used?

2.1 Challenges and Constraints

The run-time analysis of object-oriented systems is challenging because of constraints such as:

- *Amount and density of information.* The execution traces generated for run-time analysis are packed with extremely large amounts of low-level information. Therefore they must be analyzed using techniques which reduce their complexity, *e.g.*, filtering, clustering, concept analysis, or visualization.
- *Granularity of information.* Execution traces contain large amounts of low-level information, *e.g.*, which methods invoke which methods, which methods access which attributes, which objects are created at what time, etc. It is difficult, using such pieces of information, to gain an understanding at a higher level. Our lightweight approach tries to use the minimal amount of information needed to support the understanding of the run-time behavior of an application.
- *Online vs. post-mortem analysis.* Several approaches analyze the generated execution traces after the application has been shut down again. In that sense we use the definition of *post-mortem* analysis. However, it is also thinkable to generate an analyze *on-the-fly* without having to shut down the application, *i.e.*, the information would be constantly generated.

3. Our Approach

In our approach we synthesize information from a software execution without necessarily keeping the complete execution trace. We focus our analysis upon a few pieces of relevant run-time information to have a global visualization of an execution.

3.1 The Principle of a Polymetric View

The baseline of our work is based upon the lightweight approach implemented in CodeCrawler [21] [6]. For understanding software systems with static analysis, we used *polymetric views*, lightweight software visualizations enriched with software metrics. In Figure 1 we see that, given two-dimensional nodes representing entities (*e.g.*, software artifacts) and edges representing relationships, we can enrich this simple visualizations with up to 5 metrics on the nodes:

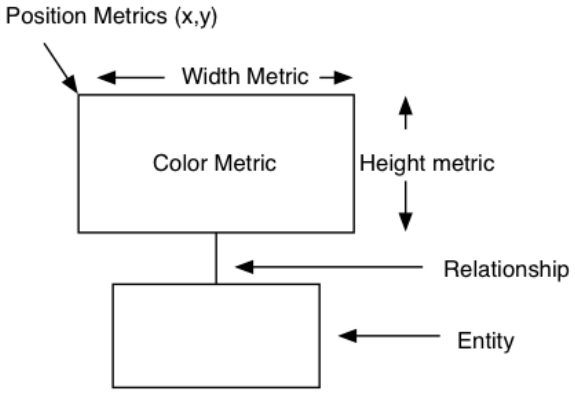

Figure 1. The principle of a polymetric view.

- *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.

- *Node Color.* The color interval between white and black can display a measurement. Here the convention is that the higher the measurement the darker the node is. Thus light gray represents a smaller metric measurement than dark gray.

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

In the previous work [21] [6] we only made use of static information and software metrics which could be gathered from a static analysis of a software system. In this article we also make use of the thickness of the edges to render measurements. This gives us information about the *weight* of an edge between two entities, *e.g.*, a thick invocation edge between two classes signifies there are many invocations between the two classes.

Note that as run-time information tends to be not linear and with extremely huge difference in scale, some of the views use a logarithmic scale to display the measurements. For example, a class may be invoked 50,000 times while another one 10.

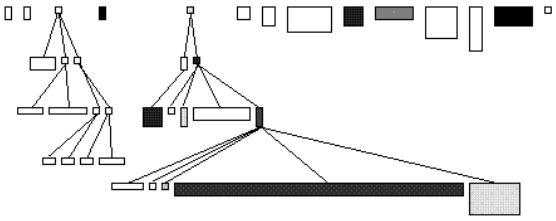

Figure 2. Enhanced Inheritance Tree.

Example. Figure 2 shows an example of an inheritance tree enhanced with run-time information. The nodes represent the classes of the analyzed application, the edges illustrate the inheritance relationships. In this example, the width of the nodes reflects the number of created instances, while the height represents the number of used methods during the execution. The color tone represents the number of method calls.

3.2 Run-Time Information Collection

Run-time information collection is a rich domain that goes from the wrapping of methods [1], the control of objects [3] [10] to the instrumentation of VM execution. A great body of work represents the run-time information in terms of a trace of events [14] [15] [26]. This trace representation is valuable information, however it consumes a lot of space (several megabytes per second of tracing, depending on the granularity level of the trace, *i.e.*, how much information is extracted) and requires a lot of abstractions and manipulation to extract information.

Our approach focuses on collecting some minimal information, *i.e.*, measurements, during the execution (number of invocations, number of object creations, number of used classes/method, etc.). We constrain ourselves to apply only relatively simple information. For example, we collect the number of method calls on a class during the execution.

The measurements we extract from an execution trace are listed in Table 1.

Name	Description
Class Run-Time Information	
NCM	Number of called methods
RCM	Rate of called methods
NMI	Number of method invocations on a class
NIMI	Number of internal method invocations on a class
NEMI	Number of external method invocations on a class
NCCM	Number of called class (static) methods
NCMI	Number of class (static) method calls on a class
NCI	Number of created instances
NCO	Number of created objects by the class instances
Method Run-Time Information	
TI	Total number of calls
ITI	Number of calls by Owner
ETI	Number of calls by Foreign

Table 1. A list of the measurements we extract from an execution trace.

At first sight the difference between NCM, NMI, and RCM can be delicate to grasp. NCM represents the number of called *methods*, NMI represents the number of *invocations* on the methods of the class, while RCM represents the rate of called methods of the class. For example, if a class has 5 methods and during the execution 3 different methods have been invoked 500 times, then NCM equals to 3 while NMI equals to 500, and RCM equals to 0.6.

To remove the ambiguity between NCI and NCO, we have to well understand that NCI represents the number of created instances of a class, while NCO represents the number of created objects by class instances.

ITI represents the number of method invocation where the caller is the receiver of the invocation, while ETI represents the number of method invocations where the caller is different than the receiver.

4 Run-time Polymetric Views

In this section, we apply a series of views enriched with run-time information resulting of a software execution. Using these examples we see the contribution that our approach might give. The case study used for our examples is described below.

4.1 Case Study in a Nutshell

The particular software system used in our experiment is the Moose reengineering environment developed in Smalltalk [11] [12]. Moose serves as a foundation for other reverse engineering tools [20] [17]. It provides a language independent representation and manipulation of source code written in C++, Java, Cobol, and Smalltalk. To achieve this language independence it is based on the FAMIX meta-model [9], which describes how elementary source code elements such as attributes, methods, classes, and namespaces are represented [8]. Moreover, Moose describes meta-models as instances of its own meta-meta-model. This explicit description of meta-models supports the creation of generic model reader and writers.

To parse the source code of applications written in Java or C++, Moose interprets CDIF or XMI compliant files, while for extracting Smalltalk applications, Moose uses its own parser and analyzes the resulting abstract syntax trees to generate Moose models. Moose is a small case study as it consists of 137 classes and 2093 methods of Smalltalk code.

We run the Moose system during the analysis of a Smalltalk application: therefore a meta-model is created, a Smalltalk-specific source code model of the application is created, then the application is analyzed extracting some metrics and other source code analysis, finally the model was saved on file and reloaded using various external representation format.

4.2 The Instance Usage Overview

Instance Usage Overview Description	
Layout	Inheritance tree, without sort
Nodes	Classes
Edges	Inheritance
Scope	Full system
Metric Scale	Logarithmic
Node Width	NCI (*Number of created instances*)
Node Height	NCM (*Number of called methods*)
Node Color	NMI (*Number of method invocations on a class*)

View Intention. The *Instance Usage Overview* view shows which classes are instantiated and used during the system's execution. As shown by the view description above the node width represents the number of created instances, the height of a node represents the number of methods that have been used, and the color the total number of method invocations during the program execution.

Revealing Symptoms. Note that this view only considers instance method invocations and does not take into account class or static method invocations. While this view provides an overview of a complete application it also offers detailed information. Here is the list of graphical signs that this view may contain:

- Small, white, and square nodes represent classes that have not been instantiated, therefore not used.
- Narrow, lightly colored nodes represent classes whose methods have been invoked but having no or few instances. This can be the case of singletons or abstract classes which are not instantiated but their methods are used by means of inheritance.
- Flat, lightly colored nodes represent classes that are heavily instantiated but not often used as their number of invocation is low (denoted by the light color).
- Flat, dark nodes represent classes that are heavily instantiated with few used, but heavily invoked, methods.
- Large, dark nodes represent classes that have been heavily instantiated and used.

Case Study. Figure 3 shows a part of the Instance Usage Overview applied on our case study.

The dark, large node A is a CDIF scanner, which parses files written in the CDIF format, an industrial exchange format. An instance of the scanner is created each time a model is loaded into memory. It is heavily invoked since the scanning is a dense process putting in movement many small and specific methods.

The dark, large node B represents the Moose meta-meta-model *AttributeDescription* class which has been instantiated a high number of times. This meta-meta-model is instantiated to represent the current Moose meta-model. As Moose is a dynamic environment and meta-model can be extended, the current meta-model representation is created each time a model is loaded. This explains why there are a lot of created instances. However, the developers of Moose were really puzzled by the fact that those classes are heavily instantiated and used (350,000 calls and 3,500 instances).

The FAMIX meta-model classes (represented in the inheritance hierarchy C) which model the source code entities

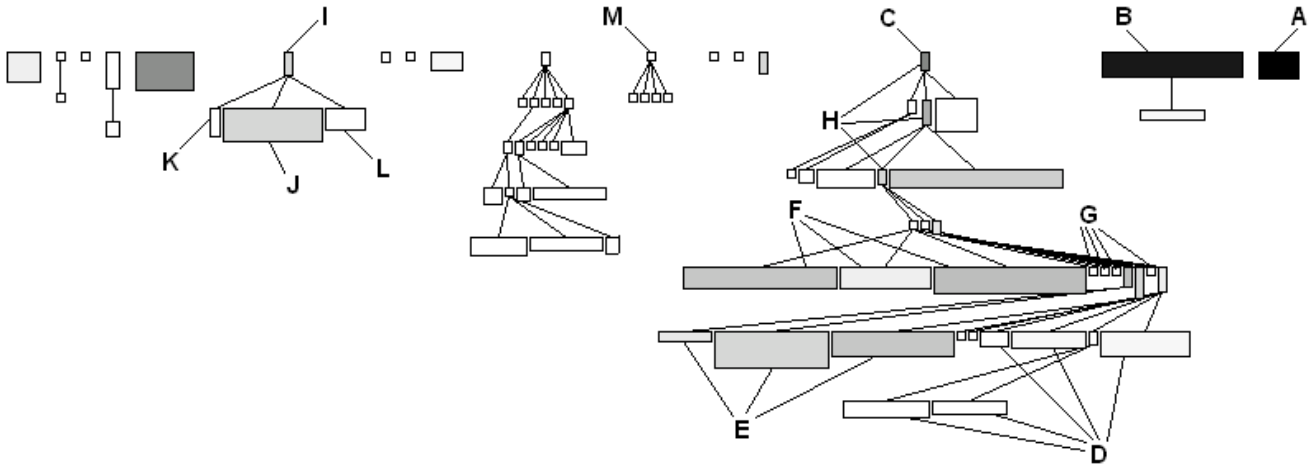

Figure 3. The Instance Usage Overview view.

are flat, lightly colored nodes. Indeed the models loaded into Moose during the tests are simple models containing only a couple of classes. Hence these classes are not the most instantiated as would be case with the loading of large models. This inheritance hierarchy contains the three following shapes:

1. The flat nodes are the information extracted from Smalltalk code (Classes, Methods, Attributes, Inheritances, ...), and they occur always as leaves of the tree. The white classes (D) that model instance and local variables are less instantiated and used compared to variable access and method invocation (E, F).

2. The small, square leaf nodes (G) represent classes that are defined in the language independent meta-model but that are not relevant in Smalltalk (Includes, SourceFile, Function). Therefore these classes have not been instantiated.

3. The narrow nodes in the middle of the hierarchy (H) represent abstract classes as they are not instantiated but their methods is invoked by subclass instances.

The small hierarchy (I) represents the visitor [13] parse tree that extracts the FAMIX meta model from the Smalltalk source code. The class *VWParseTreeEnumerator* [1] (J) is invoked each time a model is created from Smalltalk source code while the other two Visitors, which are *VWParsetree-MetricCalculator* (K) and *VWParseAnnotator* (L), are dedicated to analysis that is only performed on demand.

Finally the small hierarchy (M) is not covered at all by our execution. In fact, these classes represent a part the *graphical user interface* of Moose that have not been used during the execution.

Discussion. The *Instance Usage Overview* view is one of the first that should be applied to a system. It gives an overview of the run-time behavior of a whole application. It gives clues on the classes used in the system in the context of superclass code reuse. This view has the double advantage of combining static (inheritance shape of the system, number of classes) and run-time information for each class (number of created instances, number of method calls, number of invoked methods). The class assessment identifies *large instanced classes*, *not instantiated classes*, *very used classes*, and *not used classes*

4.3 The Communication Interaction View

Communication Interaction Description	
Layout	Embedded Spring Layout
Nodes	Classes
Edges	Invocations
Scope	Full system
Metric Scale	Linear
Node Width	NCM (*Number of called methods*)
Node Height	NCM (*Number of called methods*)
Node Color	NMI (*Number of method invocations on a class*)
Edge Width	Number of Invocations Between two Classes

View Intention. The *Communication Interaction* view shows the communication between classes of the system during its execution. As described above the size of a node

[1] VW stands for VisualWorks, a Smalltalk distribution.

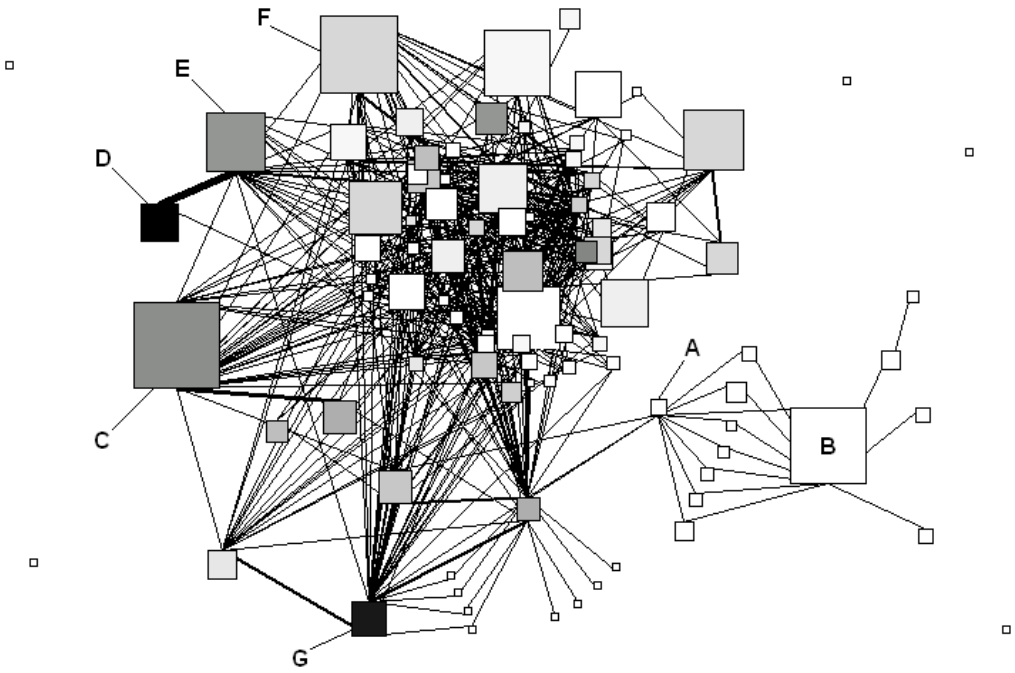

Figure 4. The Communication Interaction view.

represents the number of methods used and the color the number of methods invocations. The *Communication Interaction* view takes advantage of the embedded spring layout: it weighs the springs so that classes heavily communicating with each other will aggregate themselves. In addition, the width of the edges represents the class-to-class communication.

Revealing Symptoms. This view only considers instance level method invocations and not direct class references or instance creations between classes. This view may contain:

- Unconnected, tiny, square nodes representing classes whose methods do not invoke other methods or get invoked by other methods.
- Connected, tiny, square nodes represent classes whose methods are rarely invoked. Note that such classes can still have methods heavily invoking other methods when the node is dark.
- Large, white, square nodes represent classes having a considerable number of method used but which are rarely invoked during the execution.
- Dark, square nodes represent heavily used classes. Small, dark, square nodes in addition represent classes whose few methods are heavily used.
- Groups of nodes loosely connected to the view core represent classes communicating via a funnel [20] [22] to the rest of the system.

Case Study. Figure 4 shows the application of *Communication Interaction* view on our case study. However, we manually modified the box positions to help the interpretation of this view.

A group of classes is clearly disconnected from the rest of the core of the view, meanwhile it joins the biggest part of the view through a class (A). This group of classes implements the XMI file production based on a MOF compliant interface. This group of classes is not connected because the XMI/MOF producer is an independent package that is simply used by Moose to produce XMI model files. The big class (B) is the XMI producer which uses MOF interface objects that communicate via a bridge class (A) with the FAMIX-compliant meta-model. The XMI producer is rarely used because Moose favors the CDIF exchange format. This explains why the class is not colored.

The big class (C) is the central repository storing all the analyzed models. Moreover it acts as a main entry for querying the models. That is why this class is connected to all the classes modelling the Smalltalk source code. The medium sized dark class node (D) is the CDIF scanner that is mainly invoked by the importer class (E) which loads models into memory. The importer has the responsibility to populate a model and as such to transform textual rep-

resentations (from a CDIF text file) into objects. Note that the Moose developers learned that this class was also invoked by another one as shown in Figure 4. The big class (F) represents the class *MSEClass* that models classes in Moose. The class (G) is the class representing the meta-meta-model of Moose which is then instantiated to represent the FAMIX meta-model. This class is used by all the FAMIX classes as they describe themselves automatically and by the input/output tools as they provide meta-model independent functionality.

Discussion. The *Communication Interaction* view identifies heavily invoked classes, however it is less scalable than *Instance Usage Overview* as when the classes communicates heavily a naive spring layout has difficulties to create well identified groups of classes. Note also that in our approach we took into account has invocations self send between classes and subclasses which make the underlying view much denser. Another way to reduce such a high coupling would be to group all the classes within a common hierarchy.

4.4 The Creation Interaction View

Creation Interaction Description	
Layout	Embedded Spring Layout
Nodes	Classes
Edges	Instantiation
Scope	Full system
Metric Scale	Logarithmic
Node Width	NCO (*Number of created objects by the class*)
Node Height	NCI (*Number of created instances*)
Node Color	NCI (*Number of created instances*)
Edge Width	Number of Creation Between two Classes

View Intention. The *Creation Interaction* view shows the instance creations between classes of the system during the execution. As described above the width and the color of a class node represents the number of instances created by the class and the height represents the number of instances of the represented class. The *Creation Interaction* view takes also advantage of the embedded spring layout as it weighs the springs so that classes heavily instantiating other classes will aggregate themselves. In addition, the width of the edges represents the amount of class-to-class instantiation.

Revealing Symptoms. This view only considers instance level object creations, and may contain:

- Unconnected, tiny, square nodes represent classes that have not been instantiated, are therefore not used during the system's execution.
- Connected, tiny, white square nodes represent classes with few instances. Note that such classes can still instantiate other classes.
- Flat, lightly colored nodes represent classes that heavily create instances, but are not often instantiated themselves. A few objects of these classes create a lot of other objects. Note that we can have an abstract class that still creates a lot of objects simply due to the fact that its methods are used by instance subclasses.
- Narrow, dark nodes represent classes that have been instantiated many times. But their instances create few other instances.
- Wide, dark nodes represent classes that have been heavily instantiated and used as the number of methods used and the number of invocations are high.

Case Study. Figure 5 shows the application of the *Creation Interaction* view on a execution of Moose. Four big groups of classes are identified:

1. The group on the top of the view, which is composed of a narrow dark node (A) and flat nodes (B), has an interesting shape. The narrow class node represents the class *AttributeDescription*, a meta-meta-model entity which has been instantiated during the initialization of the system by all the FAMIX meta-model entities. The small flat nodes represent the FAMIX meta-model classes that are not Smalltalk specific but that still have been creating instances of the class *AttributeDescription* to represent themselves during the creation of the FAMIX meta-model.
2. Extracting a source code model is done in two different phases by two different entities: (1) the *VWImporter* (C) which uses the reflective API of Smalltalk to query simple structural information such as classes, methods, attributes, and (2) *VWParseTreeEnumerator* (D) which is a Visitor extracting from the AST more detailed information.
3. The group on the top left of the view represents the first extraction phase where we identify the fact that the big class (C) creates a lot of entities of the surrounding classes (E). The opposite group in the view describes the second phase where the *VWParseTreeEnumerator* (D) creates a lot of instances of the *Access* and *Invocation* classes (F) which are the most numerous entities in our meta-model.

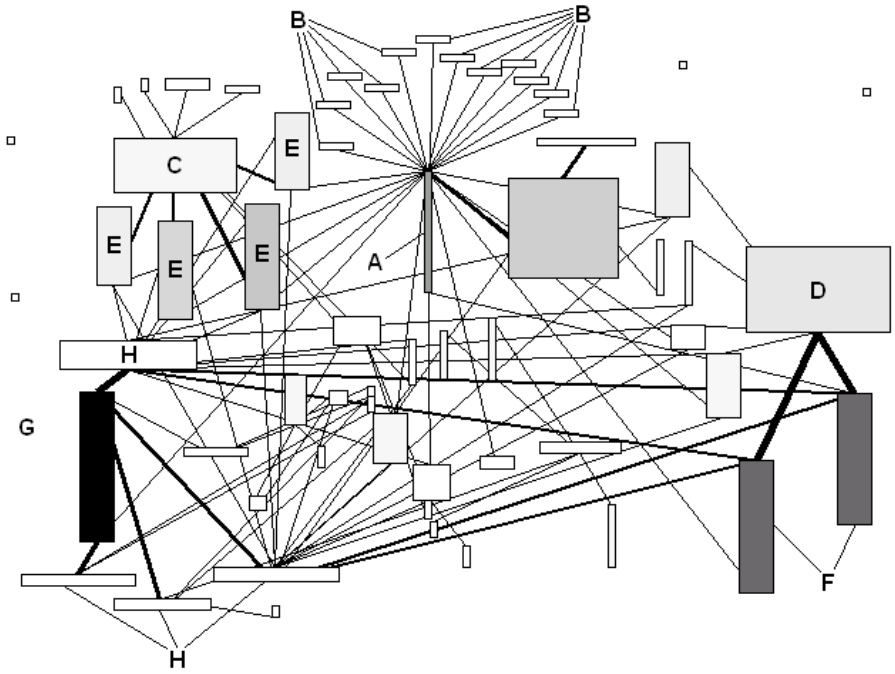

Figure 5. The Creation Interaction view.

4. Finally the group on the bottom left reveals an interesting aspect of the system. The big dark node (G) represents the class *Measurement*. Measurements which represents source code metrics are the most numerous entities created during a model analysis. As such they are not represented in memory, but they are stored on file. Instances of *Measurement* are then created during the loading of a file like the other entities but a second phase removes them from memory by means of garbage collection. What the picture shows is the fact that during the loading/saving of a source code model, instances of *Measurement* are created. The classes surrounding it are the various classes responsible for the loading and saving (H).

Discussion. The view *Creation Interaction* is clearly more scalable than the *Communication Interaction* view. This is normal as a class has a higher probability to invoke more other classes than to create instances of other classes.

4.5 The Method Call Origin View

View Intention. The *Method Call Origin* view displays how the methods are called, *e.g.*, by an internal way of the owner class or by another instance. This view can be applied on a whole system or smaller parts to understand the usage of methods during an execution. The methods are laid out using a scatterplot with a logarithmic scale where the X coordinate represents the number of internal calls and the Y coordinate represents the number of external calls for the owner class. The color shade represents of the total number of calls.

Method Call Origin Description	
Layout	Scatterplot
Nodes	Methods
Edges	-
Scope	Full system, subsystem, or single class
Scale	Logarithmic
X Coordinate	ETI (*Number of calls by Foreign*)
Y Coordinate	ITI (*Number of calls by Owner*)
Node Color	TI (*Number of calls*)

Revealing Symptoms. Below is the list of graphical signs that this view may contain:

- Nodes close to the top left of the graph represent methods that were not used during the execution.
- Nodes close to the horizontal axis represent often called methods by foreign objects. They play the role of interface in its class.
- Nodes close to the vertical axis represent often called methods by its owner. They play the role of internal behavior of its class.

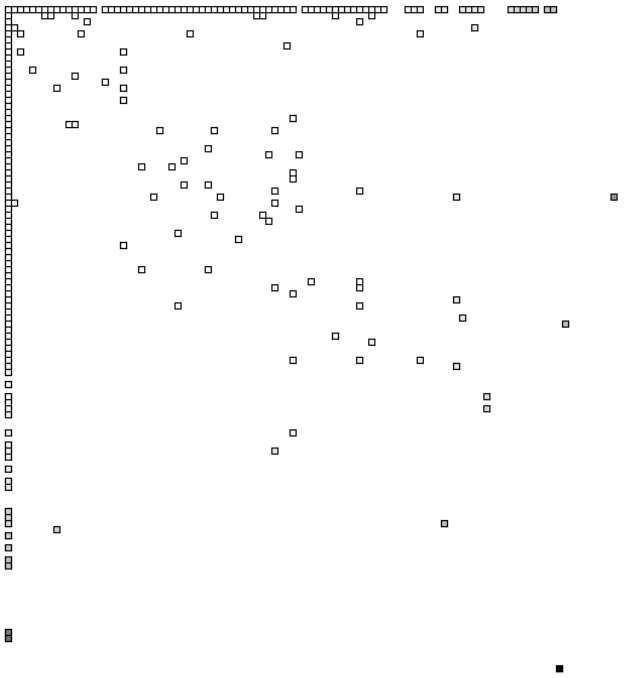

Figure 6. The Method Call Origin view.

- Nodes in the middle of the view represent hybrid methods, called either by their owner or by foreigners.
- Dark nodes and therefore far from the view's origin represent very often used methods.

Case Study. The view (see Figure 6) shows that even in a system written in Smalltalk where methods are all public, a large number of methods are mainly used internally. In the case study the methods in the middle cloud are mainly accessors methods that were used internally and externally following a coding convention.

Discussion. A scatterplot layout is good to get a feeling of the distribution of system elements according to two metrics. Indeed, in this way two entity characteristics are well illustrated, even with a huge number of entities. To grasp the kind of method calls origin of a method, we use this view to visualize three metrics. The chosen position metrics are *Number of Calls by Owner* for the X coordinate and *Number of Calls by Foreign* for the Y coordinate. The color reflects the *Total Number of Calls* to emphasize this aspect.

5. Discussion

The approach while based on a minimal amount of runtime information has proven to be successful to provide insights about application run-time. The presented views are rich as they have multiple facets revealing different information about the run-time of an application. Moreover the approach by its reduction of dynamic information is then applicable to systems that should not be disturbed and for which generating a trace would lead to extremely huge amount to data. The approach is also incremental in the sense that the collected information can be cumulated which supports the previous point. Finally the views provide overviews as well as in some cases finer information.

The drawbacks of the approach are its advantages. It does not support fine-grained run-time information at the sequence of interaction level such as offered by Jinsight-like tools [26]. Moreover, the spring layout shows some limits when applied on densely communicating systems.

6. Related Work

In the past, a great body of research has been conducted to support the understanding of object-oriented applications [16] [19] [23]. Among the various approaches to support understanding of software behavior that have been proposed in the literature, graphical representations of software execution have long been accepted as comprehension aids. Various tools provide quite different software execution visualizations.

Murphy *et al.* have developed, in AVID, an approach that allows software engineers to specify a high-level model of a system [29]. The software execution can be visualized using these models. Their visualization is oriented towards the liveness of objects and their number. Their work is directed more towards static, architectural models, while our work is more focused on the visualization of different kinds of interactions between classes of a software system during its execution.

Lange *et al.* with their Program Explorer are focused on views of classes and objects [19] [18]. The authors have developed a system for tracking function invocation, object instantiation, and attribute access. The views show class and instance relationships (usually focused on a particular instance or class), and short method-invocation histories. It is not intended as a global understanding tool. The users must know what they are interested in before they start, whereas our approach is made for covering a whole system.

Jerding *et al.* have created their own interaction diagrams to visualize the entire software execution [14] [15]. The purpose of their tool ISVis is to be able to visualize all the method calls between the classes. They can extract and recognize execution patterns, but its drawback is its lack of flexibility in the analysis. It has a good scalability for large numbers of messages, but not for a huge number of classes. In the latter case the visualization becomes less useful.

De Pauw *et al.* gave two different approaches. In their

tool Jinsight, they are focused on interaction diagrams [26]. This way, all messages between objects can be visualized. The extraction of execution patterns is also one of its main purposes. However, with an large execution trace it becomes difficult to understand class roles during execution. Earlier on, De Pauw, with its *class call clusters* and *class call matrix* [24] [25], was closer to our approach. These visualizations are simple, they have a good scalability, but they only present a small facet of an object-oriented application.

Except the last approach, all of them have in common that they visualize program executions by applying sophisticated diagrams to keep the whole execution trace. In contrast, we extract from the execution trace information which we then condense in a few metrics to enrich our visualizations.

7. Conclusion and Future Work

In this paper we presented a new way of presenting run-time information that is not based on a trace of a system, but on a minimal and compact information extracted from its execution. The approach is based on polymetric views, simple layout algorithms enriched with measurements [22].

The views proposed while been based on simple principles and a minimal run-time information still provide rich insight and multiple facets of the run-time behavior of a system.

The advantages of our approach is the fact that it can be applied to systems for which a trace generation would be difficult to extract or too big to efficiently analyze such as webservers or other applications running 24 hours a day.

Our approach can also be plugged dynamically while a system is running. It is linked to the wrapping technology we use [1] that allows one to dynamically and safely control any method but also to the minimal run-time information it requires.

In the future we plan to extend our current approach in the following ways:

- Attribute dynamics. Understanding how attributes are used during the life time of an objects or a class, its use frequency is an axis we want to explore.
- Object life-time. Objects do not have the same life time over an execution and it would be interesting to identify the different kind of objects.
- Test coverage. Understanding and assessing tests is a problem that with the emergence of test-driven methodologies is getting more and more crucial. We plan to apply our approach to understand and estimate tests and their quality.

References

[1] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP'98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[2] E. Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 10(8):45–52, January 1998.

[3] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, volume 707 of *LNCS*, pages 483–502, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[4] T. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[5] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[6] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[8] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In B. Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*, Kaiserslautern, Germany, Oct. 1999. Springer-Verlag.

[9] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 – the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[10] S. Ducasse. Evaluating message passing control techniques in smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[11] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[12] S. Ducasse, M. Lanza, and S. Tichelaar. The moose reengineering environment. *Smalltalk Chronicles*, Aug. 2001.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[14] D. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. In *Proceedings WCRE*, pages 56 – 65. IEEE, 1997.

[15] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing Message Patterns in Object-Oriented Program Executions. Technical Report GIT-GVU-96-15, Georgia Institute of Technology, may 1996.

[16] M. F. Kleyn and P. C. Gingrich. Graphtrace – understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88*, pages 191–205, Nov. 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.

[17] G. G. Koni-N'sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Bern, June 2001.

[18] D. Lange and Y. Nakamura. Program explorer: A program visualizer for C++. In *Proceedings of Usenix Conference on Object-Oriented Technologies*, pages 39–54, 1995.

[19] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA'95*, pages 342–357. ACM Press, 1995.

[20] M. Lanza. Codecrawler - lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, page to be published. IEEE Press, 2003.

[21] M. Lanza. *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Bern, 2003.

[22] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, page to be published, 2003.

[23] A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software - Concepts and Tools*, 16:170–182, 1995.

[24] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pages 326–337, Oct. 1993.

[25] W. D. Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP'94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[26] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[27] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of ICSM'2002 (International Conference on Software Maintenance)*, Oct. 2002.

[28] I. Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[29] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*, ACM SIGPLAN, pages 271–283. ACM, Oct. 1998.

[30] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.