Representing and Integrating Dynamic Collaborations in IDEs*

David Röthlisberger and Orla Greevy
Software Composition Group, University of Bern, Switzerland
{roethlis, greevy}@iam.unibe.ch

Abstract

Static views of object-oriented source code as presented in a development environment (IDE) do not provide explicit representations of dynamic collaboration to describe how source artifacts communicate at runtime. Direct access within an IDE to explicit representations of dynamic collaborations would provide developers with useful insights into a system's behavior. In this paper we describe how we seamlessly integrate novel interactive visual representations of dynamic collaborations between static artifacts to complement traditional static concepts within the IDE. We motivate our work and introduce our enhancements in our prototype IDE (Hermion) and provide validation for our work by means of case studies and benchmarks.

Keywords: dynamic analysis, dynamic collaborations, development environments, partial behavioral reflection, program comprehension

1 Introduction

Object-oriented system behavior stems from the dynamic cooperation of interacting classes and methods, thus source code browsing alone does not answer many of the questions about how an object-oriented system behaves at runtime. There is a lack of solid support for behavioral information within existing development environments (IDEs), forcing developers to maintain a mental map of the dynamic relationships between source artifacts.

A common practice of object-oriented software development is to incorporate documented design patterns to solve well-known, recurring design problems. Design patterns are a kind of "micro architectures" consisting of static program artifacts and dynamic collaborations between them. The *Chain of Responsibility* pattern, for example, processes a series of objects, involving several different static source artifacts whose dynamic interaction is often difficult to reveal from the source code. While patterns may increase the flexibility of the system, they usually also introduce a level of complexity, making a system even more difficult to understand just by static source code browsing.

Developers typically focus on specific static artifacts, *e.g.*, key classes which they have identified as relevant for their current task, and have a need to study their dynamic relationships while source code browsing. Immediate access to visualizations of dynamic class relationships that evolve in synch with the static artifacts would provide the developer with the missing behavioral information.

IDEs such as Eclipse [4] provide plugins to generate visual representations of a system's behavior (*e.g.*, UML sequence diagrams). These are usually restricted to providing pure snapshot visualizations and lack interactive capabilities to support navigation directly within the IDE. To offer real added value to a developer, we believe that visualizations should provide a means to navigate through and browsing the source code of collaborating artifacts.

In a previous work [11] we argued the importance of having runtime information embedded in the source code view of the IDE to understand dynamically-typed object-oriented software systems, *e.g.*, by integrating dynamic type information for variables directly into the source code. In this work we go one step further: we propose the introduction of dynamic collaboration representations that are readily accessible in the IDE as interactive, navigable views. We focus on three key research questions:

- *Why is it crucial to understand how source artifacts dynamically communicate from within the IDE?*
- *How can we achieve the immediate availability of dynamic information in the IDE?*
- *How do we browse and reason about dynamic collaborations in an IDE?*

We address these questions in detail in Section 2. In Section 3 we contribute our working prototype IDE Hermion, to illustrate how to represent, visualize and navigate dynamic collaborations directly in the IDE. In Section 4 we validate the efficiency of this approach by conducting benchmarks. We provide an overview of related work in the context of our work in Section 5 and conclude in Section 6.

*In: *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE)*, 2008

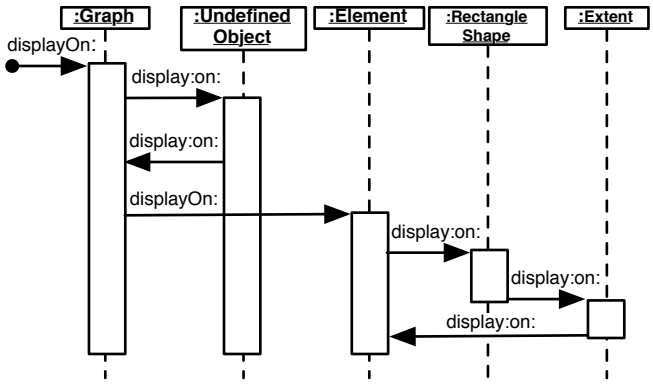

Figure 1. Sequence diagram in Mondrian to display a graph.

2 Hidden Dynamic Collaboration

The dynamics of object-oriented software systems are hidden in IDEs. To highlight this, we take as example the task of trying to understand a concrete incarnation of the *Chain of Responsibility* design pattern in one of our case study systems, Mondrian [8], a graph rendering software. To fully comprehend the workings of this pattern, the developer needs to understand how the delegation between the participating objects works at runtime.

To define a graph of nodes and edges in Mondrian, a developer specifies the layout and styles of nodes and how they are connected to each other with edges.

IDEs can often generate UML class diagrams from source code. The class diagram reveals that Mondrian has a generic design around a central class *Element*. The entire graph is a subclass of *Element* as are nodes and edges. Different styles applied to nodes are composed and arranged in a *Chain of Responsibility*. After being applied, the first style passes to the next style in the chain until all of the defined styles have been applied. As an element can be a graph, a node or an edge, it is virtually impossible to tell from reading the code what methods are actually invoked at runtime. Elements can also have children, *e.g.*, in a class hierarchy graph the subclasses of a given class are modeled as child elements of the element representing that class.

The actual dynamics of Mondrian displaying a graph is illustrated in the sequence diagram in Figure 1: To render the whole graph the method *Element ≫ displayOn: canvas* is invoked on the graph object. This method triggers the traversing of the chain of styles. The last style in this chain triggers the displaying of the element itself, *i.e.*, invoking method *Element ≫ display: element on: canvas* which iterates over all children of that element and invokes method *Element ≫ displayOn: canvas* on them. The style of the graph itself is undefined. An instance of *UndefinedObject* hence directly invokes *display: element on: canvas* to iterate over the graph's children. Revealing the interplay of graph, nodes and styles together at runtime is not feasible by studying a static class diagram or by reading source code, in particular as both, graph and elements, are simply called *element* in the source code. Moreover, it is not obvious that relevant behavior is implemented in *UndefinedObject* intended to be invoked if an element's style is undefined.

This Mondrian example is typical of the type of challenges a developer faces when trying to reverse-engineer a system within the IDE. Relevant source entities may be identified but it is difficult to see how the static artifacts interact at runtime, as the dynamics of source entities, *e.g.*, collaboration between objects, are not explicitly available in IDEs.

IDE plugins for generating sequence diagrams exist, but they do not offer a means to browse and navigate the source artifacts to which the refer, as they are not embedded in IDEs. They also fail to provide the information immediately after system's execution. Moreover, sequence diagrams generally cannot deal with large amounts of runtime data.

3 Representing Dynamic Collaboration in the IDE

To represent dynamic collaboration in the IDE, we first need to execute the system to collect runtime information. Then we empower developers to reason about the runtime information within the IDE by representing the dynamic information as interactive and navigable views. Developers select arbitrary static artifacts within the IDE, *e.g.*, several classes of an application, whose dynamic collaboration they need to comprehend for a given usage scenario. Developers then execute the system (exercising specific features of interest), and the IDE takes care of the dynamic data collection and immediately presents this data in form of views to developers.

3.1 Collecting Dynamic Information

Analyzing the runtime behavior of applications using tracing tools is time-consuming and generates large amount of data making such tools inappropriate for integration in IDEs. Developers require immediate benefit from the results of dynamic analyzes. Partial behavioral reflection overcomes these problems as it supports fine-grained selection of dynamic parts of a system on which to reflect.

We build our approach on the *Reflectivity* framework presented in the work of Denker [2]. Internally, this framework represents method source code by an abstract syntax tree (AST).

The developer triggers the dynamic analysis of the entities of interest directly from within Hermion and then runs the system, either using recorded scripts such as test cases or by directly running the system as an end-user. Reflective behavior, which is introduced into the binary of methods, collects information about every message send occurring within the selected entities. For more details of how we build our IDE enhancements on partial behavioral reflection, we refer the reader to our previous work [12].

3.2 Explicit Dynamic Collaboration

Hermion, our prototype IDE encompassing dynamic information, enables the developer to browse dynamic collaborations between static source artifacts as soon as any dynamic information has been gathered. We refer to the tool that supports browsing and analysis of dynamic collaboration as the *interactive collaboration chart*. We embed these charts tightly in the IDE so they are directly accessible to a developer working on the source code: In the case of our Mondrian example, a developer selects the static entities of interest, *e.g.*, class *Graph*, *Element* and *Style* to observe their dynamic behavior, exercises specific features of Mondrian, and even while the system is running, she can open an *interactive collaboration chart* showing the dynamic communication occurring between instances of the selected classes.

3.2.1 Interactive Collaboration Charts

In this section we describe the details of our *Interactive Collaboration Charts*. All our views are graph representations of dynamic collaborations at different levels of granularity. Their interactive capabilities support navigation of source code artifacts. Furthermore we map information (*e.g.*, number of message sends) to edges and nodes, similar to the polymetric views for visualizing runtime information described in the work of Ducasse *et al.* [3].

Class Collaboration Chart. This chart (see Figure 2) is conceptually similar to UML's sequence diagram. It displays how messages are passed between classes. As sequence diagrams do not scale for larger applications with a deep nesting level of message sending involving many classes, we condense the information in the collaboration chart to show each message sent between selected classes only once. We take into account indirect communication, *i.e.*, if class A sends a message to a not selected class, but this class sends a message to any selected class, we show a dashed line between the two selected classes denoting indirect communication. The order of message sends is not preserved in this view (the same message send between two classes is displayed as one single edge, not matter how often it occurs). As a result the collaboration chart does not become too cluttered even with large systems. To further compress the dynamic information we adopt an approach similar to that described by Hamou-Lhadj and Lethbridge [5]: we ignore repeated message sends occurring as a result of loops or recursions when calculating the message send frequency. The views guide the developer to understand how selected artifacts interplay at runtime without being confronted with too much information.

In our Mondrian example, performed by the three classes *Graph*, *Element* and *Style*, we show the interactive collaboration chart in Figure 2. All messages exchanged by these classes, including *UndefinedObject*, are visually represented. The developer can convert the whole chart into an UML sequence diagram or select a specific message send and open a new class collaboration chart with this message send as a starting point. Additionally, it is also possible to open an interactive collaboration chart on a specific method, *i.e.*, seeing all collaboration between this method and other methods. Clicking on the edge *#displayOn:* leaving *Graph* for instance brings up the method collaboration chart shown in Figure 3.

Package and Method Collaboration Chart.

We provide a big picture view of dynamic collaboration at the package level, typically representing the entire system, if necessary even including system packages. The developer can study the collaboration between any two packages by clicking on the edge between the two packages to discover which classes actually communicate to each other. It is then possible to open a *Class Collaboration Chart* on any two collaborating classes to study the collaboration on a message sending level.

To reason about communication on a method level, we use a *method collaboration chart* focusing on a particular method. This graph shows all messages sent from within this method as edges that invoke other methods. Figure 3 presents the method collaboration chart for the method *Element* $\gg$ *displayOn: canvas* of our Mondrian example.

A key characteristic of our collaboration charts is that they are interactive and support browsing within the IDE. Clicking on any class in a class collaboration chart opens this class in source code. By clicking on a message send between two classes (*i.e.*, an edge) the user can, for example, see all methods being invoked by this send or open method collaboration charts having the invoked method as a root.

Additional dynamic information is available on demand in collaboration charts, *e.g.*, the average execution time of a message send, the number of times a message has been sent, or the number of instances of a class. Such information may prove useful to assess or identify performance bottlenecks. We also provide visual means to quickly identify frequent communication paths by displaying the edges in this path thicker, as shown in Figure 3. Finally, the charts are dynamically modifiable, *i.e.*, the developer can for instance

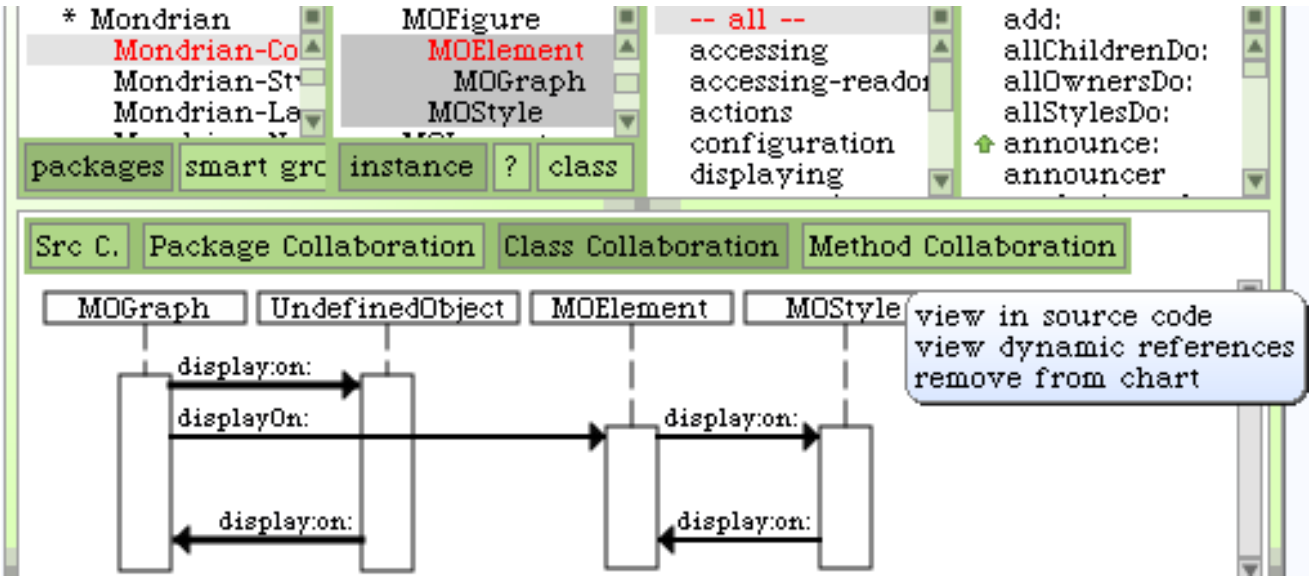

Figure 2. Integration of a class collaboration chart in the Squeak Smalltalk IDE

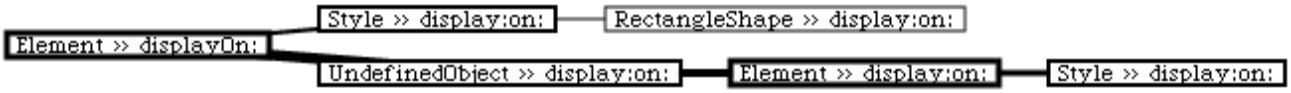

Figure 3. Method Collaboration Chart generated by the IDE

remove static artifacts from the chart or add additional artifacts (*e.g.*, classes) that should also be taken into account when rendering the chart.

In the following, we explain how these charts are tightly embedded in the IDE, *i.e.*, how these charts integrate with the static view on source artifacts such as packages, classes or methods.

3.3 Enhancing Existing IDE Tools

Typically, an IDE provides means to browse source code entities in a top-down manner, *i.e.*, going down from packages, classes to single methods, *e.g.*, by using a tree view. In the Squeak Smalltalk IDE [6] packages, classes, method categories and methods are navigated in columns in this order, as visible in Figure 2. We enhance this source code view in Squeak Smalltalk with a means to select several static artifacts of the same kind. The IDE instruments these selected entities on demand with partial behavioral reflection as described in Section 3.1 to gather dynamic information, without having to halt the system if it is already running. The developer then exercises a feature of interest in the system or runs a particular test case to generate runtime data. Within the IDE the developer chooses the class collaboration chart to view this chart based on the previously gathered information.

Such a scenario is illustrated in Figure 2 where the classes *Graph*, *Element* and *Style* have been selected. After the program, *i.e.*, Mondrian, has been executed, selecting the tab "Class Collaboration" brings up the class collaboration chart for these three classes

In all charts, static artifacts are navigable, *e.g.*, selecting a class opens this class in parallel to the chart, or clicking on edges in class charts brings up the method being invoked. These artifacts are modifiable using editors. Modifying source code marks all charts involving this source code as obsolete, *i.e.*, these charts need to be updated by a new run of the system.

4 Validation

We validate our approach by reporting on the efficiency and performance of the dynamic information gathering and the generation of the charts.

We ran a benchmark evaluating the generation of *Class Collaboration Charts*: First, we selected the three classes of our Mondrian example, (*Element*, *Style* and *Graph*), to generate a chart highlighting how these classes interact at runtime. To collect runtime data, we let Mondrian generate ten complex graphs each with a hundred nodes. We mea-

Action	Measured time (ms)
Execution w/o data collecting	7246
Execution w/ data collecting	20725 (overhead 186%)
Chart rendering	0.2

Figure 4. Time to gather data and render a class collaboration chart for Mondrian

sured the time to execute our scenario once with and once without dynamic data collection activated. We also measured the time to render the charts after having collected the dynamic information. We present the results in Table 4.

The figures we obtained from our benchmark lead us to conclude that our technique is efficient enough for most practical use cases where only a limited number of classes are to be analyzed.

Drawing the charts is very efficient even for large charts, as illustrated by the results of both benchmarks.

5 Related Work

Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [7] and Jinsight [1]. Richner and Ducasse described a Collaboration Browser [10] which represents a program's behavior in terms of collaboration patterns. Unlike our approach these approaches are only accessible in tools distinct from the IDE or do not integrate seamlessly with IDE navigation.

Reiss' [9] *Jive* tool focuses on visually representing runtime activity in real time. The goal of this work is to support software development activities such as debugging and performance optimizations. We focus on providing insights and interactive navigational aids for developers while working with the source code. Our aim is not solely to boost understanding for the software, but to use the visualization of runtime collaboration to evolve or maintain a system in the IDE.

6 Conclusions and Future Work

We described using concrete examples from case studies how we explicitly represent dynamic collaborations between static artifacts in our Hermion IDE to enhance program comprehension. We emphasized that an important prerequisite to such an representation of collaboration in IDEs is an efficient and effective technique to gather dynamic information, which is fulfilled by sub-method partial behavioral reflection [2] as we illustrated by means of conducted benchmarks.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

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

[2] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):231–251, Oct. 2007.

[3] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[4] Eclipse Platform: Technical Overview, 2003. http://www.-eclipse.org/whitepapers/eclipse-overview.pdf.

[5] A. Hamou-Lhadj and T. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proceedings of 1st International Workshop on Dynamic Analysis (WODA)*, May 2003.

[6] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997.

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

[8] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[9] S. P. Reiss. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, pages 57–66, 2003.

[10] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02)*, page 34, Los Alamitos CA, Oct. 2002. IEEE Computer Society.

[11] D. Röthlisberger, M. Denker, and É. Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.

[12] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Exploiting runtime information in the ide. In *Proceedings of the 2008 International Conference on Program Comprehension (ICPC 2008)*, 2008. To appear.