Augmenting Static Source Views in IDEs with Dynamic Metrics

David Röthlisberger[1], Marcel Härry[1], Alex Villazón[2], Danilo Ansaloni[2],
Walter Binder[2], Oscar Nierstrasz[1], Philippe Moret[2]

[1]Software Composition Group, University of Bern, Switzerland
[2]University of Lugano, Switzerland

Abstract

Mainstream IDEs such as Eclipse support developers in managing software projects mainly by offering static views of the source code. Such a static perspective neglects any information about runtime behavior. However, object-oriented programs heavily rely on polymorphism and late-binding, which makes them difficult to understand just based on their static structure. Developers thus resort to debuggers or profilers to study the system's dynamics. However, the information provided by these tools is volatile and hence cannot be exploited to ease the navigation of the source space. In this paper we present an approach to augment the static source perspective with dynamic metrics such as precise runtime type information, or memory and object allocation statistics. Dynamic metrics can leverage the understanding for the behavior and structure of a system. We rely on dynamic data gathering based on aspects to analyze running Java systems. By solving concrete use cases we illustrate how dynamic metrics directly available in the IDE are useful. We also comprehensively report on the efficiency of our approach to gather dynamic metrics.

Keywords: dynamic analysis, development environments, aspects, program comprehension

1 Introduction

Maintaining object-oriented systems is complicated by the fact that conceptually related code is often scattered over a large source space. The use of inheritance, interface types and polymorphism leads to hard to understand source code, as it is unclear which concrete methods are invoked at runtime at a polymorphic call site even in statically-typed languages. As IDEs typically focus on browsing static source code, they provide little help to reveal the execution paths a system actually takes at runtime. However, being able to, for instance, reconstruct the execution flow of a system while working in the IDE, can lead to a better understanding and a more focused navigation of the source space. In this paper we claim that developers can more efficiently maintain object-oriented code in the IDE if the static views of the IDE are augmented with dynamic information.

The importance of execution path information becomes clear when inspecting Java applications employing abstract classes or interfaces. Source code of such applications usually refers to these abstract types, while at runtime concrete types are used. However, locating the concrete class whose methods are executed at runtime when the source code just refers to interface types, can be extremely cumbersome with static navigation, since there may be a large number of concrete implementations of a declared type. Similarly, when examining source code that invokes a particular method, a large list of candidate method implementations may be generated. Static analysis alone will not tell you how frequently, if at all, each of these candidates is actually invoked. However, such information is crucial to assess the performance impact of particular code statements.

Developers usually resort to debuggers to determine the execution flow of an application. However, information extracted in a debugging session is volatile, that is, it disappears at the end of the session. Furthermore such information is bound to specific executions, so it cannot be used in general to tell which runtime types occur how often at a specific place in source code. To analyze and improve the performance of a system, developers typically use profilers, which suffer from the same drawbacks as debuggers: neither tool feeds aggregated information back to the IDE, thus developers use them only occasionally instead of benefiting from runtime information directly in the static IDE views.

This paper presents an approach to dynamically analyze systems, and to augment the static perspectives of IDEs with various dynamic metrics. To prototype this approach we implemented *Senseo*, an Eclipse plugin that enables developers to dynamically analyze Java applications. *Senseo* enriches the source views of Eclipse with several dynamic metrics such as information about which concrete methods a particular method invokes how often at runtime, which methods invoke this particular method, or how many objects or how much memory is allocated in particular meth-

ods. These dynamic metrics are aggregated over several runs of the subject system; the developer decides which runs to take into account. The paper contributes the following: (i) a technique to efficiently analyze systems dynamically, (ii) useful dynamic metrics for software maintenance, (iii) means to integrate these metrics in IDEs, and (iv) an evaluation of the efficiency and usefulness of the entire approach.

The paper is structured as follows: In Section 2 we present several concrete use cases highlighting the need for dynamic metrics available directly in the IDE. Section 3 illustrates our approach to integrate dynamic metrics in IDEs; we introduce *Senseo*, our prototype to augment Eclipse with dynamic metrics. In this section we also validate the practical usefulness by solving the use cases from Section 2 with *Senseo*. Section 4 explains our approach to gather dynamic metrics from running applications and validates this approach with efficiency benchmarks. Section 5 presents related work in the context of gathering dynamic data and its visualization. Finally, Section 6 concludes the paper.

2 Use Cases

Senseo aims primarily at supporting two typical use cases that arise when maintaining object-oriented software systems. In the first use case developers attempt to gain an understanding of the execution paths and runtime types of an object-oriented system employing complex hierarchies including abstract classes and interfaces. In the second use case, developers must improve the overall speed of the system due to critical performance issues. We summarize the difficulties arising in each use case when the developer is limited to using the plain static views of a typical IDE.

Understanding abstract class and interface hierarchies. As a case study we take the Eclipse JDT[1], a set of plug-ins implementing the Eclipse Java IDE. JDT encompasses interfaces and classes modeling Java source code artifacts such as classes, methods, fields, or local variables. Figure 1 shows an extract of the JDT interfaces and classes representing static artifacts of a class. Clients of this representation usually refer to interface types such as `IJavaElement` or `IJavaProject`, as the following code snippet found in `JavadocHover` illustrates:

```
IJavaElement element = elements[0];
if (element.getElementType() ==
    IJavaElement.FIELD) {
    IJavaProject javaProject = element.
    getJavaProject();
} else if (element. getElementType() ==
    IJavaElement.LOCAL_VARIABLE) {
    IJavaProject javaProject = element.
    getParent().getJavaProject();
}
```

[1] `http://www.eclipse.org/jdt`

There is a problem in this code: For some elements the resulting `javaProject` is wrong or undefined. The developer has the impression that the `if` conditions are not comprehensive or not correct at all. Another possibility is that the method `getJavaProject` is wrongly implemented for some element types. Thus the developer has several questions about this code:

1. Which `getJavaProject` methods are invoked?
2. Which types are stored in variable `element`; are all relevant cases covered with `if` statements?

To answer these questions purely based on static information, we can use the references and declarations search tool of Eclipse. For the first question, we search for all declarations of method `getJavaProject`. However, the JDT declares more than 20 methods with this name, most of which are not related to the representation of source code elements. We have to skim through this list to find out which declarations are defined in subtypes of `IJavaElement`. After having found those declarations, we still cannot be sure which are actually invoked in this code.

To address the second question, we first search for all classes implementing `IJavaElement` in the list of references to this interface. This yields a list with more than 2000 elements; all are false positives as `IJavaElement` is not supposed to be implemented by clients. We thus search for all sub-interfaces of `IJavaElement` to see whether those have implementing classes. After locating two direct sub-interfaces (`IMember` and `ILocalVariable`), each of which has more than 1000 references in JDT, we give up searching for references to indirect sub-interfaces such as `IField` or `IType`. It is not possible to statically find all concrete implementing classes of `IJavaElement`, in particular not those actually used in this code.

Thus we resort to use the debugger. We find out that `element` is of type `SourceField` in one scenario. However, we know that debuggers focus on specific runs, thus we still cannot know all the different types `element` has in this code. To reveal all types of `element` and all `getJavaProject` methods invoked by this polymorphic message send, we would have to debug many more scenarios, which is very time-consuming as this code is executed many times for each system run.

For all these reasons, it is much more convenient for a developer if the IDE itself could collect and show runtime information aggregated over several runs together with the static structure, that is, augmenting Eclipse's source code viewer to show precisely which methods are invoked at runtime and how often, optionally even displaying runtime types for receiver, arguments, and return values.

Assessing runtime complexity. Running this code shows that accessing the Java-project for some types of

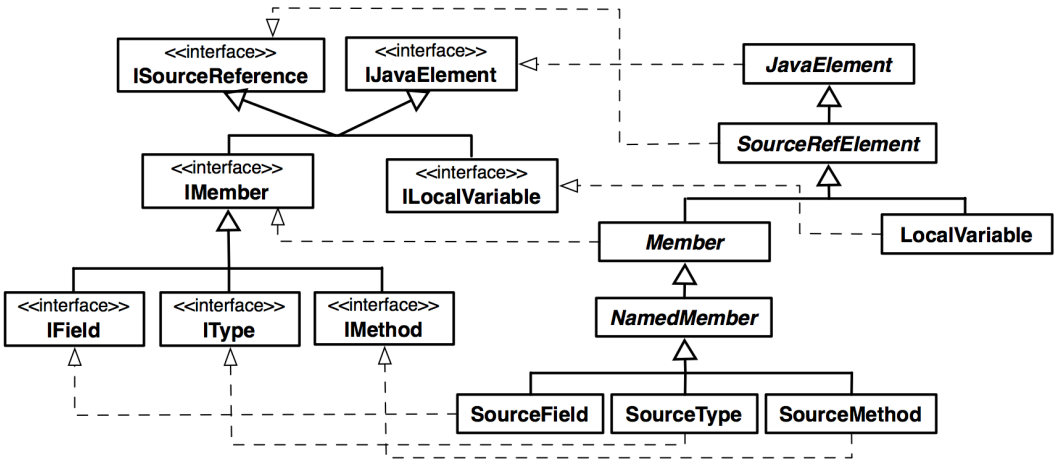

Figure 1. JDT interface and class hierarchies representing Java source elements (extract).

source elements is remarkably slow. Developers addressing the efficiency problem need to know for which types the implementation of `getJavaProject` is slow and why, that is, for instance, whether the inefficient implementations create many objects or repeatedly execute code.

We tackle these questions by profiling this code example. The profiler indeed gives us answers about the code's execution performance, but similarly to debuggers, profilers also focus on specific runs of a system and hence only show the efficiency of `getJavaProject` for the particular types used in a specific run. In the profiled run, all executed `getJavaProject` methods are reasonably efficient. However, it turns out that for very specific runs not reproducible while profiling, very slow implementations of `getJavaProject` are executed. The runtime performance of this method obviously depends heavily on the receiver of the message send.

Thus, we need to have information about runtime complexity aggregated over multiple runs to pinpoint specific method executions being slow. We claim that it is best to have such aggregated information in the IDE, at a fine-grained method level to illustrate complexity, for instance, based on receivers of a message send, but also at a more coarse-grained level, for instance, entire packages or classes, to allow candidate locations in source code for performance issues to be quickly identified at a glance. The IDE itself should give general evidence about possible performance bottlenecks.

3 Integrating Dynamic Metrics in IDEs

In this section we present an approach to augment IDEs with dynamic metrics, towards the goal of supporting the understanding of runtime behavior of applications. We prototyped this approach in *Senseo*, a plugin for the Eclipse Java IDE integrating dynamic metrics in familiar Eclipse tools such as package explorer, source editor, or ruler columns (the vertical bars next to the editor view). We first present the basic architecture of *Senseo* and second discuss dynamic metrics that can leverage runtime understanding directly in the IDE. Third, we illustrate several practical integrations and visualizations of these dynamic metrics brought to Eclipse by *Senseo*. Eventually, we solve the use cases of Section 2 to validate how useful our approach is.

3.1 Architecture

Senseo uses *MAJOR*, an aspect weaving tool enabling comprehensive aspect weaving into every class loaded in a Java VM, including the standard Java class library, vendor specific classes, and dynamically generated classes. *MAJOR* is based on the standard AspectJ [10] compiler and weaver and uses advanced bytecode instrumentation techniques to ensure portability [3]. *MAJOR* provides aspects to gather runtime information of the application under instrumentation. The collected data is used to build calling context profiles containing different dynamic metrics such as number of created objects.

The application to be analyzed is executed in a separate application VM where *MAJOR* weaves the data gathering aspect into every loaded class, while the Eclipse IDE runs in a standard VM to avoid perturbations. While the subject system is still running, we periodically transfer the gathered dynamic data from the application VM to Eclipse using a socket. We do not have to halt the application to obtain its dynamic data. *Senseo* receives the transfered data, processes it and stores the aggregated information in its own storage system which is optimized for fast access from the IDE. Figure 2 gives an overview of the setup of our approach.

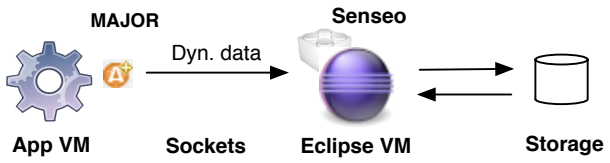

Figure 2. Setup to gather dynamic metrics.

To analyze the application dynamically within the IDE, developers have to execute it with *Senseo*. Before starting the application, developers can define which dynamic metrics should be gathered at runtime. By default, all packages and classes of the application are dynamically analyzed. However, developers can restrict the analysis to specific classes or even methods to reduce the analysis overhead if only specific areas need to be observed. Extending the analysis to also cover the standard Java class library is possible too. As soon as Eclipse receives dynamic data over the socket from *MAJOR*, *Senseo* directly displays the dynamic metrics extracted from the transfered calling context profiles. Section 3.2 proposes useful dynamic metrics to gather and Section 3.3 illustrates means to integrate these metrics in Eclipse. Usually *Senseo* aggregates dynamic data over all application runs executed with it, but developers can empty the storage and start afresh.

3.2 Dynamic Metrics

To support the use cases of Section 2, we integrate the following dynamic metrics into the IDE.

Message sending. In object-oriented programs, understanding how objects send messages to each other at runtime is crucial. We therefore extract the following information about message sends:

Invoked Methods. Often invoked methods are not implemented in the statically defined type (*i.e.*, class), but in its super- or subtypes, when inheritance, respectively dynamic binding, are used. Having information in the IDE about the methods invoked is intended to help developers better understand collaborations between objects and ease navigation of the runtime execution flow.

Optionally, developers can ask to gather more information about message sends when starting the application with *Senseo*. Such additional information includes:

- *Receiver types*. Often sub-types of the type implementing the method receive the message send at runtime. Knowing receiver types and their frequency thus further increases program understanding.
- *Argument types*. Information about actual argument types and their frequency increases the understanding for a method, *i.e.*, how it is used at runtime.
- *Return types*. As return values pass results from one method to another, knowing their types and their frequency helps developers to better understand communication between different methods, for instance, whether a `null` return value is to be expected.

Number of invocations. This dynamic metric helps the developer quickly identify hot spots in code, that is, very frequently invoked methods or classes containing such methods. Furthermore, methods never invoked at runtime become visible, which is useful when removing dead code or extending the test coverage of the application's test suite. Related to this metric is the number of invocations of other methods triggered from a particular method.

Number of created objects. By reading static source code, a developer usually cannot tell how many objects are created at runtime in a class, in a method or in a line of source code. It is unclear whether a source artifact creates one or one thousand objects — or none at all. This dynamic metric, however, is useful to assess the costs imposed by the execution of a source artifact, to locate inefficient code, or to discover potential problems, for instance inefficient algorithms creating enormous numbers of objects.

Allocated memory. Different objects vary in memory size. Having many but very tiny objects might not be an issue, whereas creating a few but very huge objects could be a sign of an efficiency problem. Hence, we also provide a dynamic metric recording memory usage of various source artifacts such as classes or methods. This metric can be combined with the number of created objects metric to reveal which types of objects consume most memory and thus are candidates for optimization.

3.3 Enhancements to the IDE

All these dynamic metrics are seamlessly integrated with the static perspective provided by the Eclipse IDE. This ensures that the dynamic metrics augment the static view of a system. In the following we describe how the available dynamic metrics augment the IDE.

Source code enhancements. As a technique to complement source code without impeding its readability we opted to use *hovers*, small windows that pop up when the mouse hovers over a source element (a method name, a variable, *etc.*). Hovers are interactive, which means the developer can for instance open the class of a receiver type by clicking on it. We now describe the integration of dynamic metrics with *Senseo*:

Method header. The hover that appears on mouse over the method name in a method header shows (i) all senders invoking that particular method, (ii) all callees, that is, all methods invoked by this method, and optionally (iii) all

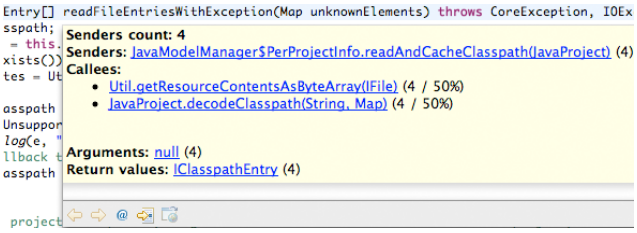

Figure 3. Hover appearing for a method name in its declaration.

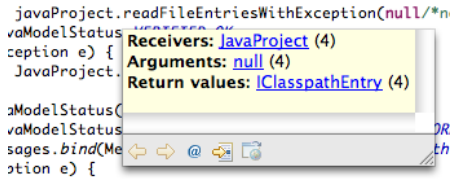

Figure 4. Hover for a message send occurring in a method.

argument and return value types. For each piece of information we also show how often a particular invocation occurred. For instance for a sender, we display the qualified name of the method containing the send (that is, the calling method) and the number of invocations from this sender. Optionally, we also display the type of object to which the message triggering the invocation of the current method was sent, if this is a sub-type of the class implementing the current method. For a callee we provide similar information: The class implementing the invoked method, the name of the message, and how often a particular method was invoked. Additionally, we can show concrete receiver types of the message send, if they are not the same as the class implementing the called method. Figure 3 shows a concrete method name hover for method `readFileEntriesWithException`.

In a method header, we can optionally show information about argument and return types, if developers have chosen to gather such data. Hovers presenting this information appear when the mouse is over the declared arguments of a method or the defined return type. These hovers also include numbers about how often specific argument and return value types occurred at runtime.

Method body. We also augment source elements in the method body with hovers. For each message send defined in the method, we provide the dynamic callee information similarly as for the method name, namely concretely invoked methods, optionally along with argument or return types that occurred in this method for that particular message send at runtime, as shown in Figure 4. Of course all these types listed are always accompanied with the number of occurrences and the relative frequency of the specific types at runtime.

Ruler columns. There are two kind of rulers next to the source editor: (i) the standard ruler on the left showing local information and (ii) the overview ruler on the right giving an overview over the entire file opened in the editor. In the traditional Eclipse IDE these rulers denote annotations for errors or warnings in the source file. Ruler (i) only shows the annotations for the currently visible part of the file, while the overview ruler (ii) displays all available annotations for the entire file. Clicking on such an annotation in (ii) brings the developer to the annotated line in the source file, for instance to a line containing an error.

We extended these two rulers to also display dynamic metrics. For every executed method in a Java source file the overview ruler presents, for instance, how often it has been executed in average per system run using three different icons colored in a hot/cold scheme: *blue* means only a few, *yellow* several, and *red* many invocations [21]. Clicking on such an annotation icon causes a jump to the declaration of the method in the file. The ruler on the left side provides more detailed information: It shows on a scale from 1 to 6 the frequency of invocation of a particular method compared to all other invoked methods, see Figure 5. A completely filled bar for a method denotes methods that have been invoked the most in this application. The dynamic metrics in these two rulers allow developers to quickly identify hot spots in their code, that is, methods being invoked frequently. The applied heat metaphor allows different methods to be compared in terms of number of invocations.

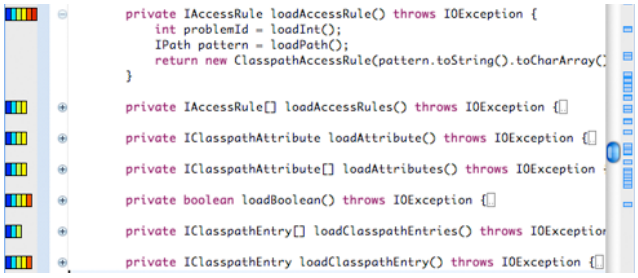

Figure 5. Rulers left and right of the editor view showing dynamic metrics.

To associate the continuous distribution of metric values to a discrete scale with for instance three representations (*e.g.*, red, yellow, and blue), we use the *k-means* clustering algorithm [13].

To see fine-grained values for the dynamic metrics, the annotations in the two columns are also enriched with hovers. Developers hovering over a heat bar in the left column

or over the annotation icon in the right bar get a hover displaying precise metric values, for instance exact total numbers of invocations or even number of invocations from specific methods or receiver types.

Furthermore, developers can choose between different dynamic metrics to be visualized in the rulers. Besides number of invocations of methods, we also provide metrics such as the number of objects a method creates, the number of bytecodes it executes, and the amount of memory it allocates, either on average or in total over all executions. Such metrics allow developers to quickly assess the runtime complexity of specific methods and thus to locate candidate methods for optimization. Changing the dynamic metrics to be displayed is done in the Eclipse preferences; the chosen metric is immediately displayed in the rulers.

Package Explorer. The package explorer is the primary tool in Eclipse to locate packages and classes of an application. *Senseo* augments the package explorer with dynamic information to guide the developer at a high level to the source artifacts of interest, for instance to classes creating many objects. For that purpose, we annotate packages and classes in the explorer tree with icons denoting the degree with which they contribute to the selected dynamic metric such as amount of allocated memory. A class for instance aggregates the metric value of all its methods, a package the value of all its classes. Similar to the overview ruler the metric values are mapped to three different package explorer icons: *blue*, *yellow*, and *red*, representing a heat coloring scheme [21].

3.4 Evaluation

In Section 2 we raised two questions about a typical code example from the Eclipse JDT. We show that with *Senseo* developers can answer such questions directly in the source perspective of Eclipse.

First, to determine the `getJavaProject` methods invoked in the given code example, developers hold the mouse over the call site written in source code to get a hover mentioning all distinct methods that have been invoked at runtime at this call site, along with the number of invocations. This hover saves us from browsing the statically generated list of more than 20 declarations of this method by showing us precisely the actually invoked methods. Second, to find out which types of objects have been stored in `element`, we can look at the method call to `getElementType`, whose statically defined receiver is the variable `element`. The hover can also show the runtime receiver types of a message send, which are all types stored in `element` in this case. It turns out that the types of `element` are `SourceField`, `LocalVariable`, but also `SourceMethod`, thus the `if` statements in this code have to be extended to also cover `SourceMethod` elements. We were unable to statically elicit this information.

To assess the efficiency of the various invoked `getJavaProject` methods, we navigate to the declaration of each such method. The dynamic metrics in the ruler columns reveal how complex an invocation of this method is, that is for instance how many objects an invocation creates in average, even depending on the receiver type. Thanks to these metrics we find out that if the receiver of `getJavaProject` is of type `LocalVariable`, the code searches iteratively in the chain of parents of this local variable for a defined Java-project. We can optimize this by searching directly in the enclosing type of the local variable.

4 Dynamic Metrics Collection

In this section we first explain our approach to dynamic metrics collection and afterwards investigate its overhead.

4.1 Aspect-based Calling Context Tree Construction

Senseo requires support for flexibly aggregating dynamic metrics in various ways. For instance, runtime type information is needed separately for each pair of caller and callee methods, while memory allocation metrics need to be aggregated for the whole execution of a method (including the execution of its direct and indirect callees). In order to support different ways of aggregating metrics, we resort to a generic datastructure that is able to hold different metrics for each executed calling context. The Calling Context Tree (CCT) [1] perfectly fits this requirement. Figure 6 illustrates a code snippet together with the corresponding CCT (showing only method invocation counts as metric).

Each CCT node stores dynamic metrics and refers to an identifier of the target method for which the metrics have been collected. It also has links to the parent and child nodes for navigation in the CCT. Our CCT representation is designed for extensibility so that additional metrics can be easily integrated.

CCT construction and collection of dynamic metrics can be implemented either with a modified Java Virtual Machine (JVM), or through a profiling agent implemented in native code using the standard JVM tool interface (JVMTI) [24], or with the aid of program transformation respectively bytecode instrumentation techniques. For portability and compatibility reasons, we chose the latter approach. However, instead of using a low-level bytecode engineering library for implementing the instrumentation, we rely on high-level aspect-oriented programming (AOP) [11] techniques in order to specify CCT construction and metrics collection as an aspect. This approach not only results in a compact implementation, but it ensures ease of maintenance and extension.

Code Sample

```
void f() {
  for(int i=1;i<=10;++i) {
    h();
    g(i);
  }
}
void g(int i) {
  for(int j=1;j<=i;++j) {
    h();
  }
}
void h() { return; }
```

Calling Context Tree

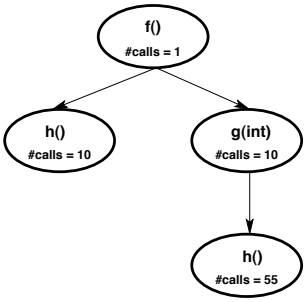

Figure 6. Sample code and its corresponding CCT

Our implementation leverages *MAJOR* [25, 26], an aspect weaver based on AspectJ [10] offering two distinguishing features. First, *MAJOR* allows for complete method coverage. That is, all methods executing in the JVM (after it has completed bootstrapping) can be woven, including methods in the standard Java class library. Second, *MAJOR* provides efficient access to complete calling context information through customizable, thread-local shadow stacks. Using the new pseudo-variables `thisStack` and `thisSP`, the aspect gets access to the array holding the current thread's shadow stack, respectively to the array index (shadow stack pointer) corresponding to the currently executing method.

Figure 7 illustrates three advices[2] of our aspect for CCT construction and dynamic metrics collection. In the `CCTAspect`, each thread generates a separate, thread-local CCT. The shadow stack is an array of `CCTNode` instances, representing nodes in the thread-local CCT. A special root node is stored at position zero. Periodically, after a configurable number of profiled method calls, each thread integrates its thread-local CCT into a shared CCT in a synchronized manner. This approach reduces contention on the shared CCT, yielding significant overhead reduction in comparison with an alternative solution where all threads directly update a shared CCT upon each method invocation. Because of space limitations, the details of periodic CCT integration are not shown in Figure 7.

The first advice in Figure 7 intercepts method entries and pushes the `CCTNode` representing the invoked method onto the shadow stack. To this end, it gets the caller's `CCTNode` instance from the shadow stack (i.e. at position `sp-1`) and invokes the `profileCall` method, which takes as argument an identifier of the callee method. We use static join points, accessed through AspectJ's `thisJoinPointStaticPart` pseudo-variable, to uniquely identify method entries; they provide information about the method signature, modifiers, *etc.* The `profileCall` method returns the callee's `CCTNode` instance and increments its invocation counter; if the same callee has not been invoked in the same calling context before, a new `CCTNode` instance is created as child of the caller's node.

[2]Aspects specify *pointcuts* to intercept certain points in the execution of programs (so-called *join points*), such as method calls, fields access, etc. *Before*, *after*, or *around* the intercepted join points specified *advices* are executed. Advices are methods that have access to some contextual information of the join points.

```
before() : execution(* *(..)) {
  CCTNode[] ss = thisStack;
  int sp = thisSP;
  ss[sp] = ss[sp-1].profileCall(
   thisJoinPointStaticPart);
  ss[sp].storeRcvArgsRuntimeTypes(
   thisJoinPoint);
}

after() returning(Object o) :
    execution(* *(..)) {
  CCTNode[] ss = thisStack;
  int sp = thisSP;
  ss[sp].storeRetRuntimeType(o);
  ss[sp] = null;
}

after() returning(Object o) :
    call(*.new(..)) {
  CCTNode[] ss = thisStack;
  int sp = thisSP;
  ss[sp].storeObjAlloc(o);
}
...
}
```

Figure 7. Simplified excerpt of the CCTAspect

The second advice in Figure 7 deals with normal method completion, popping the method's entry from the shadow stack. For simplicity, here we do not show cleanup of the shadow stack in the case of abnormal method completion throwing an exception [26].

The third advice in Figure 7 intercepts object instantiations to keep track of the number of created objects and of allocated memory for each calling context. The method `storeObjAlloc(Object)` uses the object size estimation functionality provided by the `java.lang.instrumentation` API (which is exposed to the aspect) to update the memory allocation statistics in the corresponding `CCTNode` instance.

In addition to the number of method invocations and to the object allocation metrics, the `CCTAspect` collects the runtime types of receiver, arguments, and result. For receiver and arguments, this functionality is implemented in method `storeRcvArgsRuntimeTypes` used upon method entry. It takes a dynamic join point instance,

which is accessed through AspectJ's `thisJoinPoint` pseudo-variable in the advice. The dynamic join point instance provides references to the receiver and to the method arguments. Upon method completion, the `storeRetRuntimeType` method stores the runtime type of the result. The result object is passed as context information (`returning(Object o)`) to the advice.

During the execution, the aspect also takes care of periodically sending the collected metrics to the *Senseo* plugin in the IDE. Upon metrics transmission, thread-local CCTs of terminated threads are first integrated into the shared CCT. Afterwards, the shared CCT is traversed to aggregate the metrics as required by *Senseo*. Finally, the aggregated metrics are sent to the plugin through a socket. Metrics aggregation and serialization may proceed in parallel with the program threads, since they operate on thread-local CCTs most of the time.

4.2 Performance

We evaluate our approach in two different settings. We use *MAJOR*[3] version 0.5 with AspectJ[4] version 1.6.2 and the SunJDK 1.6.0_13 Hotspot Server Virtual Machine.

In the first setting, we assess the overhead caused by the collection of dynamic metrics using the standard DaCapo benchmark suite[5]; we also explore the different sources of overhead and their contributions to the overall overhead. We focus only on the performance of *MAJOR* and exclude the overhead of communication with the *Senseo* plugin. *MAJOR* is configured to ensure complete method coverage, profiling also execution within the Java class library. In the first setting, our measurement environment is a 16-core machine running RedHat Enterprise Linux 5.2 (Intel Xeon, 2.4GHz, 16GB RAM). We chose a high-end machine in order to speed up the benchmarking process, and each measurement represents the median of the overhead factor of 15 runs within the same JVM process. In the second setting, we measure end-to-end performance of our system (i.e., including *MAJOR* and *Senseo*), as experienced by the user. In this setting, we use a typical developer machine (Intel Core 2 Duo, 2.16Ghz, 2GB RAM).

Setting 1. Figure 8 illustrates the overhead due to the `CCTAspect`. We explore and seggregate the overhead contributions of shadow stack maintenance, of CCT construction, and of collecting memory allocation metrics.

The geometric mean of the overhead for all the benchmarks is factor 3.98, while the maximum overhead is factor 8.75 for "xalan". The biggest part of this overhead stems from CCT creation, which is particularly expensive in the presence of recursions. The collection of memory allocation metrics causes relatively little overhead, because object alloction is far less frequent than method invocation and the corresponding advice can directly access the `CCTNode` that has to be updated on the top of the shadow stack.

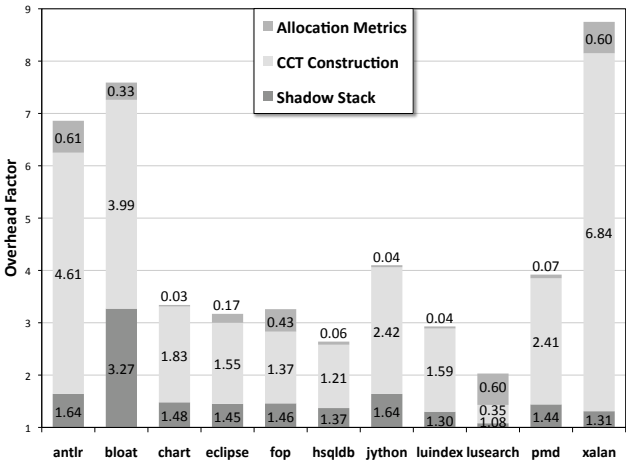

Figure 8. Overhead of shadow stack creation, CCT construction, and collection of memory allocation metrics

Setting 2. In order to assess end-to-end performance of our system, we measure the overhead of running an application with *Senseo*. As a case study, we ran Eclipse itself as target application and analyzed the usage of the JDT core. Without *Senseo*, starting and terminating Eclipse takes 53 seconds; executing the same scenario with *Senseo* takes 188 seconds, i.e., the overhead is 255%. This overhead includes the collection of all dynamic metrics discussed in this paper within all JDT core classes, metrics aggregation, data transmission through a socket, and visualization of the data in the developer's Eclipse environment. Eclipse is conveniently usable while being analyzed. Considering that the JDT is a large application consisting of more than 1000 classes and approximately 16000 methods, our measurements confirm that our approach is practical also for large workloads.

5 Related Work

5.1 Metrics Collection

JFluid exploits dynamic bytecode instrumentation and code hotswapping to collect dynamic metrics [5]. JFluid uses a hard-coded, low-level instrumentation to collect gross time for a single code region and to build a Calling Context Tree (CCT) augmented with accumulated execution time for individual methods. In contrast, we use a flexible, high-level, aspect-based approach to specify CCT construction and dynamic metrics collection, which eases

[3] `http://www.inf.unisi.ch/projects/ferrari`
[4] `http://www.eclipse.org/aspectj/`
[5] `http://dacapobench.org/`

customization and extension. In addition, JFluid relies on a customized JVM, whereas our tools are fully portable and compatible with standard JVMs. Similar to *Senseo*, JFluid runs the application under instrumentation in a separate JVM, which communicates with the visualization part through a socket and also through shared memory. JFluid is a pure profiling tool, whereas *Senseo* was designed to support program understanding and maintenance. The JFluid technology is integrated into the NetBeans Profiler [15].

Dufour *et al.* [6] present a variety of dynamic metrics for Java programs. They introduce a tool called *J [7] for metrics measurement. In contrast to our fully portable approach, *J relies on the Java Virtual Machine Profiler Interface (JVMPI) [23], which is known to cause high performance overhead and requires profiler agents to be written in native code. Other profilers based on the JVMPI or its successor, the JVM Tool Interface (JVMTI) [24], such as JProfiler [6] or JProbe [18] also suffer from platform dependence and from limited extensibility.

PROSE [17] provides aspect support within the JVM, which may ease the collection of certain dynamic metrics with aspects, thanks to the direct access to JVM internals. PROSE combines bytecode instrumentation and aspect support at the just-in-time compiler level. It does not support aspect weaving in the standard Java class library, a distinguishing feature of *MAJOR*.

Sampling-based profiling techniques, which are often used for feedback-directed optimizations in dynamic compilers [2, 27], help significantly reduce the overhead of metrics collection. However, sampling produces incomplete and possibly inaccurate information, whereas *Senseo* requires complete and exact metrics for all executed methods. Hence, we rely on *MAJOR* to comprehensively weave our aspect into all methods in the system.

Dynamic analyses based on tracing mechanisms traditionally focus on capturing a call tree of message sends, but existing approaches do not bridge the gap between dynamic behavior and the static structure of a program [8, 28]. Our work aims at incorporating the information obtained through dynamic analyses into the IDE and thus connecting the static structure with the dynamic behavior of the system.

With static analysis, especially with static type inference [16], it is possible to gain insights into the types that variables assume at runtime. However, static type inference is a computationally expensive task and cannot always provide precise results in the context of object-oriented languages [19]. Furthermore, static analysis does not cover dynamically generated code, although dynamic bytecode generation is a common technique, for instance used in the "jython" benchmark of the DaCapo suite.

[6] `http://www.ej-technologies.com/products/jprofiler`

5.2 Augmenting IDEs

Reiss [20] visualizes the dynamics of Java programs in real time, *e.g.*, the number of message sends received by a class. Löwe *et al.* [14] follows a similar approach by merging information from static analysis with information from dynamic analysis to generate visualizations. The visualizations of these two approaches are not tightly integrated in an IDE though, but are provided by a separated tool. Thus, it is not directly possible to use these analyses while working with source code. We consider it as crucial to incorporate knowledge about the dynamics of programs into the IDE to ease navigating within the source space.

Other approaches ease and support the navigation of large software systems by different means than program analysis. For instance, NavTracks [22] keeps track of the navigation history of software developers. Using this history, NavTracks forms associations between related source files (*e.g.*, class files) and can hence present a recommendation list of entities related to the current selected source file, that is, source files developers browsed in the past along with the currently selected file. Mylar [9] computes a degree-of-interest value for each source artifact based on historical navigation. The relative degree-of-interest of artifacts is highlighted using colors — interesting entities are assigned a "hot" color. We also use heat colors to denote the degree of dynamic metrics such as number of invocations, but base our model on dynamic data.

6 Conclusions

In this paper we motivated the integration of dynamic information into IDEs such as Eclipse to ease maintaining object-oriented applications written in languages such as Java. We presented *Senseo*, an Eclipse plugin enabling developers to dynamically analyze their applications from within Eclipse. *Senseo* uses aspect-oriented programming techniques to gather runtime data from the system. *Senseo* presents gathered dynamic metrics such as execution flow information, runtime type information, numbers of method invocations, or amount of memory executed by augmenting the familiar static source views (package explorer, source editor, *etc.*). For instance, *Senseo* integrates dynamic metrics in hovers, in the ruler columns of the editor, or by annotating icons in the package explorer. We evaluated our work by solving concrete, practical use cases with *Senseo* and by thoroughly conducting performance benchmarks, as efficiency is most critical when it comes to dynamic analysis. *Senseo* as a tool is described in more detail in a separate tool demonstration paper. We plan to exploit the dynamic data gathered with *Senseo* by other means than augmenting the source perspective, for instance by visualizing the data in software maps [12].

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[3] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM Press.

[4] M. Desmond, M.-A. Storey, and C. Exton. Fluid source code views. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 260–263, 2006. IEEE Computer Society.

[5] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. *SIGSOFT Softw. Eng. Notes*, 29(1):139–150, 2004.

[6] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.

[7] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–307, 2003. ACM Press.

[8] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[9] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, 2005. ACM Press.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[12] A. Kuhn, P. Loretan, and O. Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 209–218, 2008. IEEE Computer Society Press.

[13] S. P. LLoyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[14] W. Löwe, A. Ludwig, and A. Schwind. Understanding software - static and dynamic aspects. In *17th International Conference on Advanced Science and Technology*, 2001.

[15] NetBeans. The NetBeans Profiler Project. Web pages at `http://profiler.netbeans.org/`, 2008.

[16] J. Pleviak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of OOPSLA '94*, pages 324–340, 1994.

[17] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, New York, NY, USA, 2003. ACM Press.

[18] Quest Software. JProbe. Web pages at `http://www.quest.com/jprobe/`.

[19] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery. Dynamic type inference to support object-oriented reengineering in smalltalk, 1998. Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS).

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

[21] D. Röthlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes. Supporting task-oriented navigation in IDEs with configurable heatmaps. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC 2009)*. To appear.

[22] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, Washington, DC, USA, sep 2005. IEEE Computer Society.

[23] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`, 2000.

[24] Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. Web pages at `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, 2006.

[25] A. Villazón, W. Binder, and P. Moret. Aspect Weaving in Standard Java Class Libraries. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 159–167, New York, NY, USA, Sept. 2008. ACM.

[26] A. Villazón, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-Oriented Programming. In *AOSD '09: Proceedings of the 8th International Conference on Aspect-oriented Software Development*, pages 63–74, Charlottesville, Virginia, USA, Mar. 2009. ACM.

[27] J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, June 2000.

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