Instance Pointcuts for Program Comprehension

Christoph Bockisch
University of Twente
P.O. Box 217
7500 AE Enschede, the Netherlands
c.m.bockisch@cs.utwente.nl

Kardelen Hatun
University of Twente
P.O. Box 217
7500 AE Enschede, the Netherlands
k.hatun@cs.utwente.nl

Abstract
The dynamic behavior of programs generally cannot be fully observed via the source code, but dynamic tools, e.g., debuggers, have to be used. Comprehending dynamic behavior entails observing object interactions during runtime. The class structure is not always sufficient to understand these interactions since objects of the same type can be used in various places, just as objects of different types can be used in similar places. Our novel concept, instance pointcuts, groups objects based on the events they participate in, introducing a flexible way of creating object categories; a category contains objects that are used in the specified way. This paper proposes an application of instance pointcuts to the program comprehension domain. We illustrate the usefulness of our approach through three comprehension scenarios.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging—Code inspections and walk-throughs; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Design, Languages

Keywords program comprehension, debugging, instance pointcuts, object categories, object-oriented programming, aspect-oriented programming

1. Introduction
Complex software systems are difficult to understand because they consist of many elements with complicated dependencies. A factor further complicating the comprehensibility is the dynamicity of execution semantics: In object-oriented programming languages, the dynamic type of the receiver object determines which implementation of the called method will be executed. It is generally not possible to determine the dynamic type of an expression just by looking at the source code. For this reason, often dynamic tools are used to observe the program execution for comprehending a program. As such tools are mainly used during debugging, they are often called debuggers; we will also use this term throughout this paper to refer to tools which in general allow observing and possibly interacting with program executions at runtime.

In object-oriented programming, on the source code level the main abstractions are classes (object-oriented languages which are not class-based provide an equivalent like prototypes in delegation-based languages). At runtime, however, the main building blocks are objects which are instances of classes. Comprehending the behavior of a program means to comprehend the interplay of objects. In general, there can be arbitrarily many instances of each class, which makes the comprehension difficult as their connections and dependencies can be manifold.

A natural technique to increase comprehensibility is introducing a limited number of categories, grouping all objects into categories and considering only the categories during the comprehension task. This is even well supported by object-oriented languages, since classes already form such categories and there is an easily observable relation between an object and its class, and vice versa. However, the categorization offered by the class structure of the program is not always the best fit for the comprehension task at hand. On one hand, instances of the same class may be used in different ways throughout the program execution; on the other hand, instances of different classes may be used in the same or a similar way.

For these reasons, we claim that a more powerful and flexible way for dynamically categorizing objects is needed to guide the task of program comprehension. In our ongoing research, we are developing the language concept of instance pointcuts as an extension to the AspectJ language. The original motivation is to facilitate flexibly extending program behavior and composing independently developed architectural components. Recently, we have found the abstraction provided by instance pointcuts also to be useful to support comprehension of highly dynamic object-oriented programs.

In the following section, we will briefly introduce the concept of instance pointcuts. By walking the reader through a few particular comprehension tasks in section 3, we will illustrate the applicability of instance pointcuts; throughout this section we will sketch a user interface that visualizes instance pointcuts. In section 4 we will discuss challenges in realizing the outlined concepts. We present related work in section 5 and conclude our position in section 6.

2. Instance Pointcuts
Instance pointcut is a declarative language construct that is used to reify and maintain a set of objects of a specified type, with the ability to select them over a period marked by events in the usage of the object, modularizing the object selection concern and making it declarative. We have implemented a prototype by extending AspectJ. Besides defining instance pointcut based on events in the usage of objects, our language also provides means to compose instance pointcuts from existing ones. This feature is mainly useful in application programming where instance pointcuts may be reused and may evolve along with the application; in comprehension tasks, we expect instance pointcuts be used in an ad-hoc manner and therefore do not discuss these advanced features in detail here. In our language prototype, instance pointcuts are static members of
aspect modules. For the purpose of this paper we present them as stand-alone constructs, which can be defined through the user interface of the debugger. Detailed information on using instance pointcuts as a general-purpose language feature for composing software components is given in our previous paper [5].

1) ip ::= ‗instance pointcut‘ name ‗<‘ type ‗>‘ ‗:‘
2) ip−expr (UNTIL ip−expr)?
3) ip−expr ::= aft−event ‗[‘ bef−event
4) | bef−event ‗[‘ aft−event
5) | bef−event
6) bef−event ::= ‗after‘ ‗(pointcut−expression‘
7) bef−event ::= ‗before‘ ‗(pointcut−expression‘)

Listing 1. Grammar definition for instance pointcuts

A concrete instance pointcut definition (the grammar is shown in Listing 1) consists of a name and type, and an instance pointcut expression. The latter selects the desired events at which the instance pointcut set must be maintained and binds the object which is to be added to or removed from the maintained set. An instance pointcut expression consists of two sub-expressions that are separated by the UNTIL keyword. The first sub-expression, the so-called add expression, selects events which expose the instances to be added to the instance pointcut set. After the UNTIL clause an optional sub-expression, called the remove expression, can be defined, which specifies when to remove which object. When the remove expression is not defined, the object is kept in the set until it dies.

In AspectJ, join points mark regions in time [8]; together with the keywords for defining an advice (before or after), events are selected. As we need to identify events for our approach, we combine pointcut expressions with advice specifiers to obtain expression elements. Each expression element contains a (nearly) regular AspectJ pointcut expression and the advice specifier (before or after) to select events. Both add and remove expressions are composed of expression elements. A sub-expression (add/remove expression) contains at least one expression element and at most two.

The pointcut expressions used in instance pointcuts are almost plain AspectJ pointcut expression with a few restrictions and only one addition: Every pointcut expression must contain exactly one binding predicate (args, target, etc.) that binds the (implicit) instance parameter. We extend the list of binding predicates with a new one, namely returning. The returning clause binds the returned variable by a method or a constructor and may only be used inside an after expression element. In AspectJ the result value can only be bound by the advice specifier after returning; since we only use pointcut expressions, we had to extend the pointcut language.

The way how the instance pointcut’s object set is maintained bears two peculiarities. First, instance pointcuts do not keep objects alive. This means: independently of the optional remove expression, objects are removed from the maintained set when they are not referenced by the rest of the program anymore and are garbage collected. Second, actually multisets are maintained. Thus, an object can be multiply bound by the add-expression and has to be bound equally often by the remove-expression to eventually be eliminated from the set.

In our full language prototype, we provide several operators to redefine and compose instance pointcuts. For example, new instance pointcuts can be defined based on existing ones by refining the type constraint or by composing them through the set operations union and intersect. Furthermore, properties of instance pointcuts can be checked, e.g., if the content of the instance pointcut’s set will always be empty, a warning can be issued.

3. Example Walkthrough

In this section, we will describe an existing system and several comprehension scenarios where instance pointcuts can be useful. Intertwined with these scenarios, we will discuss a hypothetical extension to the Eclipse debugger for using instance pointcuts.

We made the observation that instance pointcuts can support program comprehension while working on an extension to the Jikes Research Virtual Machine [1]. In particular, we were extending the optimizing just-in-time (JIT) compiler of this virtual machine. The optimizing compiler works in multiple phases, whereby the first one creates an object-based intermediate representation (IR) from the Java bytecode of the method which is currently compiled. This IR, basically a linked list of Instruction objects, is iteratively analyzed and rewritten until eventually machine code is emitted. In our extension we manipulated this IR by inserting instructions and adding rewrites. When we made a mistake, runtime failures occurred and we had to comprehend the impact of our extension on the further processing of the JIT compiler, a very complex system.

All elements in the IR are instances of the same class (Instruction) which are configured with an Operator object determining the behavior of the instruction. Thus, at first sight it is not clear which instruction is represented by an Instruction object. Besides the referenced Operator object, also the creation site of the Instruction object allows to draw conclusions about the purpose of an Instruction: There is a factory class for each kind of instruction. For instance, if we are only interested in instructions representing a function invocation, we can focus our comprehension task on those Instruction objects that are returned by a create method in the class Call.

3.1 Scenario 1

A View for Instance Pointcuts An instance pointcut, referred to as callInstructions, selecting the objects described above can be defined as seen in Figure 1. The figure depicts a view for defining instance pointcuts. The new view allows to add instance pointcuts which are maintained during the execution of the program, and to inspect the content of the defined instance pointcuts while the program execution is suspended at a breakpoint. The left column shows the names of the defined instance pointcuts. A row with an instance pointcut can be expanded, for example in the figure the instance pointcut callInstructions is expanded and its elements are listed. When a row containing the name of an instance pointcut is selected, the detail pane (at the bottom) shows its definition. When a row is selected which represents an element of an instance pointcut, the toString() of the value is shown.

![Figure 1. The Instance Pointcuts View](image)

1) For brevity we only show simple class names.
**Linking the Instance Pointcut View** The left-most icon of the toolbar in Figure 2 allows linking the Instance Pointcut View with other debugging-related views. When linking is turned on and an object is selected on another view, e.g., in the Variables view, all instance pointcuts are highlighted which contain the selected object. Figure 2 shows the instance pointcut callInstructions in green and bold, which means that the user has currently selected an object contained in callInstructions. In the outlined comprehension task, this feature makes it simple to identify for each Instruction object, if it represents an instruction relevant for the task, i.e., for which an instance pointcut was defined.

**Using Instance Pointcuts in Conditions and Expressions** Besides using instance pointcuts during inspection when the execution is suspended at a breakpoint, defined instance pointcuts can also be used for defining conditional breakpoints. As an example consider the task of comprehending a late compiler phase. At this time, many Instructions have been mutated for optimization purposes or for injecting runtime services. But many Instructions still simply reflect the functionality directly specified by the Java bytecode instructions. Assume we want to comprehend the impact of injected runtime services on the liveness analysis, a phase necessary for generating meta-data needed by the garbage collector. Figure 4 shows a screenshot of the breakpoint properties dialog where the condition refers to the instance pointcut defined in Figure 3. The execution is only suspended at this breakpoint when the local variable inst is selected by the instance pointcut describing the Instructions that realize runtime services. In the same way, watch expressions entered in Eclipse’s Expression View can refer to instance pointcuts, simply treating them as java.util.Set.

**3.3 Scenario 3**

**Instance Pointcut Watchpoints** To demonstrate the remove expression feature of instance pointcuts, consider the two scenarios above. Since mutation changes the operation represented by an Instruction object, mutated Instructions must be removed from the instance pointcut representing a specific operation as mentioned in the first scenario. Figure 5 shows the extended instance pointcut definition.

Furthermore, the language concept of instance pointcuts allows to advise the joint points when a new instance is added to an instance pointcut and when an instance is removed, respectively. Instance pointcuts internally maintain multisets; we specify that only the initial addition and the final removal of an object are join points, i.e., when the cardinality of an object in the multiset changes from 0 to 1 and from 1 to 0, respectively. These join points can also be used to define watchpoints. The two right-most columns in Figure 5 represent this feature. In the example, the watchpoint for adding an object to the instance pointcut callInstructions is enabled; the breakpoint for removing an object is disabled.

**4. Challenges**

Obviously, the expressiveness of the pointcut language determines how precisely sets of objects can be selected. Since in our prototype we use the AspectJ pointcut language underlingly, we are limited to selecting the supported events and specifying supported restrictions. In the example of the optimizing compiler of the Jikes virtual machine, this can be a significant limitation: long switch statements identify different ways of processing instructions and it is often relevant in the context of which case an object was used. However,
since AspectJ pointcuts cannot refer to switch cases, they cannot be used in the add or remove expressions of instance pointcuts.

We are used to adding breakpoints or watch expressions dynamically, during the runtime of the debugged program. With instance pointcut, this is not possible or at least risky. As is always the case with dynamic deployment of aspects, it may be that some relevant join points have already passed at the time of deployment. Thus, it may be that objects, which should have been selected by an instance pointcut, are not selected; or not all of the recursive additions have been tracked and an object is removed from the multiset too early. For these reasons, we do not suggest to support dynamic addition of instance pointcuts, even though this may require to restart the program during one comprehension task, if new relevant categorizations for objects are discovered during runtime.

5. Related Work
A field related to instance pointcuts is object query languages which are used to query objects in an object-oriented program [4]. However object query languages do not support event based querying as presented in our approach, but only allow to select objects based on the current heap structure.

So-called omniscient debuggers [7, 11] trace all relevant events of the whole execution of programs, including changes to the heap. After the program execution, they allow to perform queries on the execution trace, e.g., to localize defects in the code, but also to generally comprehend the program behavior. But current omniscient debuggers limit the programmer to using predefined (although parameterized) queries for searching the execution trace. Extensions to omniscient debugging [10, 12] support more advanced queries over the execution trace of a program, Kho et al. [6] even support scripting queries. Nevertheless, our approach is more flexible and declarative in selecting object sets and can be applied in online-debugging.

Using pointcut-based techniques for specifying breakpoints in a debugger has been proposed before. Ng et al. [9] discuss using aspects for program comprehension in general. Chern and De Volder [3] have proposed to define breakpoints based on the current control flow, similar to AspectJ’s cflow pointcut designator. Bodden has proposed stateful breakpoints [2], which allows programmers to specify the order in which different events must occur to lead to suspension of the program execution. Yin et al. have presented a language for specifying breakpoints [13] which is also based on a pointcut language and already allows specifying breakpoints based on object relations. Nevertheless, the approach presented in our paper is the first to actually maintain a set of objects, which relate by how they have been used in the past, and to expose these sets to different debugging facilities.

6. Conclusion
In this paper we proposed an application of the instance pointcuts approach to the program comprehension domain, as an extension to the Eclipse debugger. Our vision is that by using instance pointcuts during debugging, we will be able to identify objects based on how they are used which is not observable by the class structure. Inspired by the challenges encountered while debugging an extension of the Jikes VM’s optimizing compiler, we devised three scenarios (section 3). In the first scenario we explained how the instance pointcuts concept is useful during debugging; we can easily identify objects that are relevant to the debugging task by looking if they are contained by a certain instance pointcut. In the second scenario we explored the use of instance pointcuts with conditional breakpoints; instance pointcuts can be referenced as Sets and we can observe if an object is added to the instance pointcut set by means of a conditional expression. In the third and the final scenario we have utilized the removal feature of instance pointcuts; here instance pointcuts are used to maintain a set of objects starting from the time they were created until they were mutated. We also showed how the built-in joinpoints for add and remove operations can be used to define watchpoints.

Despite some limitations, instance pointcuts are beneficial for program comprehension since they provide the means to create meaningful runtime categories. We think that this extra dimension brings practicality for understanding dynamic behavior. In future work, we will develop this idea further and explore different event selection languages to overcome limitations introduced by AspectJ.

References


