PCDiff: Attacking the Fragile Pointcut Problem

Christian Koppen (koppen@fmi.uni-passau.de)
Maximilian Stoerzer (stoerzer@fmi.uni-passau.de)
University of Passau

August 22, 2004

Abstract

Aspect oriented programming has been proposed as a way to improve modularity of software systems by allowing encapsulation of cross-cutting concerns. To do so, aspects specify *where* new functionality should apply using *pointcuts*.

Unfortunately todays mainstream aspect oriented languages suffer from pointcut languages where pointcut declarations result in a high coupling between aspect and base system. Additionally, these pointcuts are *fragile*, as non-local changes easily may break pointcut semantics. These properties are a major obstacle for program evolution of aspect oriented software. This paper introduces a pointcut delta analysis to deal with this problem.

1 Motivation

Aspect oriented programming (AOP), first introduced in [2], is a new paradigm in programming extending traditional programming techniques. Its basic idea is to encapsulate so called *cross-cutting concerns* influencing many modules of a given software system in a new kind of module called *aspect*. Aspects provides two constructs to specify new behaviour and where it should apply: *advice* and *pointcuts*. Advice is a method-like construct defining new functionality which is bound to a pointcut identifying a set of well-defined points during the execution of a program called *joinpoints*. So these pointcuts specify where advice should be executed. The *aspect weaver* finally combines aspect functionality with the base system producing an executable system.

For the remaining of this paper, we will use AspectJ [6] as an example, although the observations made are also valid for other currently available AspectJ-like languages. The pointcut language of AspectJ offers a set of *primitive pointcut designators*, like `call` specifying method call sites or `get/set` specifying field access. These primitive pointcut designators can be combined using logical operations (`||, &&, !`) forming (named) *pointcuts*. As a running example, we will define aspects for a simple shopping cart system, the main class `ShoppingCart` is shown in program 1.

Program 1 A simple class modeling a shopping cart

```
public class ShoppingCart {

  public final static int NEW=0,
   PROCESSED=1, FINISHED=2;

  private Set items;
  private int status;
  private double total;
  private Customer receiver;

  ShoppingCart(Customer receiver) {
    this.receiver = receiver;
    items = new HashSet(); total = 0.0;
    status = NEW;
  }

  public void addItem(Integer itemNr) {
    items.add(itemNr);
    total += Database.loadPrice(itemNr);
  }

  public void removeItem(Integer itemNr) {
    items.remove(itemNr);
    total -= Database.loadPrice(itemNr);
  }
  ...
}
```

The problem with current pointcut designators is that most of them explicitly specify their target location(s) by *naming* elements of their corresponding base program (see program 2). These explicit references obviously in-

troduce a *high coupling* between the base system and the aspect, making aspect reuse harder.

Program 2 A highly coupled pointcut

```
aspect ItemChanges {
  pointcut itemChanges(Customer c):
    this(c) &&
    ( call(* ShoppingCart.addItem(..)) ||
      call(* ShoppingCart.removeItem(..)));

  before(Customer c): itemChanges(c) {
    // do something
  }
}
```

AspectJ also offers *wild-cards* to reduce coupling. However, this introduces a new problem. If pointcuts use this mechanism they rely on *naming conventions*. As such conventions are not checked by a compiler, they are *never guaranteed*. As a result, programmers have to be very careful with their pointcuts to avoid spurious or missed matches. In program 3, the previous example has been rewritten using a wild-card expression to pick out all calls methods modifying the items in our shopping cart demo application.

Program 3 A pointcut using wildcards

```
aspect ItemChanges {
  pointcut itemChanges(Customer c):
    this(c) &&
call(* ShoppingCart.*Item(..)));
  ...
}
```

Assume we have a naming convention that all methods in class `ShoppingCart` modifying the set of items in the cart should end with `item`. In class `ShoppingCart` the wild-card pointcut expression matches the two methods complying to this naming convention. Additionally, an aspect programmer always has to make sure, that his pointcuts do not match methods accidentally complying to this name pattern.

For this trivial example a pointcut mismatch can easily be seen. However, aspects have been proposed for large or distributed system scenarios, where it is much harder to find spurious or missed matches. In general, the aspect programmer needs *global system knowledge* to assure that his pointcut works as expected. Additionally, humans tend not to look for *unexpected* things, and mismatches in general are unexpected.

If a programmer uses wild-cards or not, he still has to face another problem: pointcuts in general are *fragile*. Consider the following scenario. A programmer correctly specifies a pointcut, as in program 3. The corresponding aspect works as intended, all tests are successful. Now, the code is refactored, and we rename some methods. If we have references to these methods in our program, the compiler will tell us if we missed to update a reference.

Now if we consider the pointcut, we see that the set of joinpoints picked out by the – unchanged! – pointcut definition may be altered by a rename. In general there are several trivial non-local changes possibly modifying pointcut semantics in terms of actually matched joinpoints.

Rename: Renaming classes, methods or fields influences matching semantics of `call`, `execution` and `get/set` and other pointcuts. Wild-cards can only provide limited protection against these effects.

Move method/class: Pointcuts can pick out joinpoints by their lexical position, using `within` or `withincode`. Moving classes to another packages or methods to another class obviously changes matching semantics for such pointcuts.

Add/Delete method/field/class: Pointcut semantics is also affected by adding or removing program elements. New elements can (and sometimes should) be matched by available pointcuts, but in general pointcuts development cannot anticipate all possible future additions. Removal of program elements naturally results in 'lost' joinpoints.

Refactorings in general require to modify code referencing updated code[1]. Automated refactoring, as can also be seen in IBM's Eclipse IDE for Java or the Smalltalk Refactoring Browser might be a way to avoid breaking pointcuts in some cases, but for AspectJ refactorings are currently not available and might be problematic for dynamic joinpoints in general. More important, automated refactorings require that the user explicitly requests a refactoring. But this not necessarily address system evolution in general, just consider adding new methods, classes or packages due to new functionality.

So currently system evolution as well as refactorings are done manually. While this is also the case for several other languages, there is an important difference between "traditional" code and pointcuts. In general code affected

[1] Renaming a method for example requires modifying all calls to this method to match the new name.

by a refactoring – but not updated – will result in a *not compilable* program. So the compiler checks for a lot of potential errors introduced by refactorings (although semantic differences can occur in this context as well which are *not* revealed by the compiler).

For pointcuts, this problem is considerably more serious as changed pointcut semantics *in general are not visible* for the programmer. There is no support at all to alert programmers if refactorings change the set of joinpoints matched by their pointcuts. In our opinion, this is a major problem for program evolution of aspect oriented software.

Additionally, aspects influencing a given base class are not visible in the code as a consequence of the *obliviousness-property* [3] of AOP. As a result, a programmer refactoring e.g. a class of the base system is not aware of all the aspects possibly matching joinpoints in this class. Tool support lightens this problem [1], but in our opinion this does not resolve the problems for evolution of aspects, classes and their dependencies.

This also raises another interesting question for companies developing AO software: If failures due to changed pointcut semantics occur, who is responsible? The aspect developer or the base developer? Both answers are not satisfactory. The aspect programmer developed the aspect for a given version of the system using the program elements at hand, and cannot anticipate all potential refactorings of the system.

The base programmer in general should not be responsible to modify an affected aspect as an aspect might affect many other modules as well. So the base class programmer definitely is no expert to adapt an aspect potentially influenced by his code changes. This is especially true as aspects can even access non-public elements of a class[2]. So in general both programmers have to talk to each other but therefore they have to be aware of potential evolution problems.

We think the issues demonstrated here are crucial for current AspectJ-like pointcut languages. We refer to this problems as the *fragile pointcut problem*.

2 Language-based Improvements

The improvement of the pointcut definition mechanism is an important research topic today. Several approaches have been proposed to attack the fragile pointcut problem using improved pointcut languages.

[2]For example `get/set` pointcut designators allow to intercept any access to (potentially `private`) fields.

To reduce coupling, AspectJ invented *abstract aspects*. These aspects can contain abstract pointcuts which are defined by inheriting aspects. Thus all the advice code is encapsulated in the abstract aspect and can be reused. The aspect can be applied to concrete problem by inheriting from the abstract aspect and defining the pointcuts for the concrete base system. Unfortunately, although coupling is reduced, pointcuts in the concrete aspect still are fragile.

[4] proposes a completely different pointcut language. In this language, a program is represented as a set of facts and pointcuts are defined in a Prolog like language as a query over these facts. However, this language is Turing-complete, thus pointcuts are more dynamic as in AspectJ-like languages and often cannot be evaluated at compile time.

An approach in-between these two extremes proposes *declarative pointcuts*, a set of *descriptive pointcut designators* which allows to specify joinpoints by their (semantic) properties [5]. This approach reduces the necessity to reference names or source locations and thus considerably lightens the problem with fragile pointcuts. Unfortunately, these pointcut designators are currently not available.

While we consider the improvement of pointcut languages important research, these languages will only lighten the problem in the future when the emerging constructs will become part of main stream languages. However, by then we assume that there is a considerable amount of code written in e.g. AspectJ where evolution suffers from the problems outlined above - even if the attempted refactoring is the renewal of the pointcut definitions with new, more declarative constructs. For this code we think our approach can be most valuable.

3 Pointcut delta analysis

Software written today using available pointcut languages with all the deficiencies outlined above potentially will be maintained for years. So a way to deal with this problem *for current languages* is needed.

In general, semantical differences introduced into a system are (hopefully) revealed by rerunning a regression test suite (failing test). Testing however only shows the presence of bugs, but can never prove their absence. For a failing test, the results have to be further analyzed to actually track down bugs.

Our solution to deal with the fragile pointcut problem is to provide a tool to detect differences in pointcut semantics. This tool should be used as follows: We have a working version of our system. Some time later, the sys-

tem evolves. Several edits (of base and aspects) produce an new version of this system. Unfortunately now some regression tests fail. We assume that a pointcut mismatch might be the reason and thus want to know *how the set of matched joinpoints has changed*.

If an aspect (or more specific its pointcuts) has not been modified, base code edits could be a reason for now experienced test failures. If the pointcut has been modified, we might expect differences in its matching behaviour. Anyway, a delta of the matched pieces of advice can considerably help to validate the expectations.

We propose the following pragmatic and straight forward analysis: We calculate the set of matched joinpoints for both versions of the program and compare the resulting sets, producing delta information for pointcut matching. This approach is possible for any AspectJ-like language where the set of matched pointcuts is (at least partly) statically computable.

Definition 3.1 (Pointcut Delta) *Let P be the program before, P' the program after the edits. Let $joinpoints(P)$ be a function calculating all joinpoints for the given program P, $advice(j,P)$ a function listing all pieces of advice at a given joinpoint j in program P (for $j \notin joinpoints(P)$ define $advice(j,P) = \emptyset$). Let*

$$JP = joinpoints(P') \cup joinpoints(P)$$

be the set of joinpoints in both program versions. Let

$$add(P,P') = \bigcup_{j \in JP} (advice(j,P') - advice(j,P))$$

be the set of additional advice matches *and*

$$del(P,P') = \bigcup_{j \in JP} (advice(j,P) - advice(j,P'))$$

the set of lost advice matches*. We are then interested in the set*

$$deltaPC(P,P') = \{(add(P,P') \cup (del(P,P'))\}$$

which represents exactly all pieces of advice now either applying additionally *or applying* no longer *(associated with the respective joinpoint j).*

The benefit of calculating the delta sets is that these sets tend to be *small* compared to the overall number of all pieces of advice in the system. If $deltaPC(P,P') = \emptyset$, a base programmer can assume that any applying aspect is not affected by changes he made. If $deltaPC(P,P')$ contains differences, these differences can easily be traced back to the affected aspects, so the aspect programmer can be notified of this change. A potential problem is detected *before delivery* and can thus be easily corrected.

This delta analysis has been implemented in an Eclipse plugin extending the AspectJ Development Tools (ajdt) to access relevant joinpoint match information. The current implementation only uses information which is available from the ajdt-plugin [1] and the AspectJ compiler (the *structure model*) and does not calculate any matching information itself.

The AspectJ structure model works well for static pointcuts, but for pointcuts including dynamic joinpoints (`if`, `cflow`, ...), the model is problematic as it (conservatively) approximates possible matches (i.e. `if(..)` is approximated as `true`). So the model reports spurious matches. A comparison of supersets obviously might fail to report differences, both additional or lost matches.

Although we are currently not aware of any numbers illustrating how often dynamic joinpoints are used in practice, in our opinion this is a relevant problem of the delta approach presented here.

A simple way to deal with this problem is to trace back advice matches to the pointcut definition responsible for the match. If the pointcut definition contains dynamic pointcut designators, the system should mark up these matches to show that here the delta might include spurious information. The user then can interpret the information as either a reliable information (for matches statically computable) or as a heuristic hint requiring additional examination.

It is also possible to reduce the amount of spurious matches by further analysing dynamic joinpoints, but an exact calculation of matching information in general is not computable. As this is also a relevant problem for performance of AOP software, this is a current research topic[8].

Although this is a weakness of our approach, we think that even the simple plugin available now can considerable help programmers to track down bugs in evolving aspect-oriented software. However, this is outside the scope of this paper and considered future work.

Our tool allows users to take snapshots of arbitrary versions of the program and compare these snapshots with each other. Results of this comparison are presented in the task view and as text markers in the editor for the current version and in a tree-based comparison view showing differences on a per-file basis, comparing the program model of P and P'. The model of P' is adorned with the set of lost matches, additional matches appear in the model of P.

To represent changes of pointcut semantics in the source code, the plugin also uses the *Eclipse marker mechanism* to show additional or lost matches. Sometimes lost matches can no longer be displayed (if the target joinpoint respective its underlying resource has been deleted). In this case, the lost match is shown at the top of

the file. Additionally, the *task view* contains an entry for each affected file.

4 Example: A simple shopping cart application

We will demonstrate our plugin with our running example. It deals with a simple web-based shopping cart system as used in many places to keep track of articles and orders of customers. As time and technology advance the system evolves, so unfortunately introducing new flaws. We will demo how our tool can help to identify reasons for unexpected or faulty behaviour.

4.1 Initial system

The central class for our analysis is the class `ShoppingCart` (program 1). Customers may add or remove items to the cart, order the items in the cart and so on (not shown here).

The initial system contains the class *ShoppingCart* and a class modelling customers. Additionally, the aspect `Authentication` checks access to the shopping cart using `before`-advice; if a customer is not authorized to change the shopping cart content, an `AccessException` is thrown. Therefore, the pointcut `itemChanges` is used defining all joinpoint where item data is actually changed within a shopping cart.

Within our tool, we can take *snapshots* of every state of the system that might be of interest for later comparisons. As the initial state is always a good starting point for comparisons, we take a snapshot now (Figure 1).

Figure 1: Taking a Snapshot

4.2 Modification 1: Persistence

With the system evolving, the shopping cart shall be made persistent, i.e. its data shall be saved permanently, so that a customer can log off and log on again without losing the content of his shopping cart (see *Amazon*).

Program 4 `Customer` and `Authentication`.

```
public class Customer {
  Integer custNr;
  String name, address;
  ShoppingCart cart;

  public Customer(String name, String address){
    this.name = name; this.address = address;
    this.cart = new ShoppingCart(this);
    custNr = Database.newUniqueNr();
  }

  public void orderItem(Integer itemNr) {
    System.out.println("Adding item " + itemNr);
    cart.addItem(itemNr);
  }

  public void cancelItem(Integer itemNr) {
    System.out.println("Removing item "+itemNr);
    cart.removeItem(itemNr);
  }

  ShoppingCart getCart() { return cart; }

  ...
}

public abstract aspect ItemChanges {
  pointcut itemChanges(Customer c) :
    this(c) && call(* ShoppingCart.*Item(..));
}

public aspect Authentication
       extends ItemChanges {
    before (Customer c) : itemChanges(c) {
        // check if customer may change
        if (!mayAccess(c)) {
            throw new AccessException(
              "Illegal Access - denied.");
        }
    }

    private boolean mayAccess(Customer c) {
        return c.getName().length() % 2 == 0;
    }
}
```

To do so, a `Persistence` aspect is added. It contains an `after`-advice saving the shopping cart content after any modification. As `Persistence` and `Authentication` affect the same *joinpoints*, the pointcut `itemChanges` can be reused (persistence must be enforced if and only if an item within the cart has changed). The code is shown in program 5. An example output can be seen in figure 6.

To prevent unwanted side-effects, the system is checked after this modification with our tool. Figure 2 shows the changes between the original and the modified system.

For every changed file, a marker appeared in the task view. When viewing one of the affected files in the editor, every change is shown as code markers in the editor; a green plus or red minus icon shows the kind of change, the change is further described in the marker description.

Program 5 The `Persistence` aspect

```
public aspect Persistence
  extends ItemChanges {
    after (Customer c): itemChanges(c) {
        // save new state
        Database.saveShoppingCart(c);
        System.out.println(
          "Saving Shopping Cart to DB");
    }
}
```

As expected, the persistence advice will be executed after every shopping cart modification.

Note that the programmer has a very focused view on the differences in matched joinpoints. He no longer has to filter anything which has not changed – only actual changes are displayed so greatly reducing the amount of data to check. Unwanted changes can be seen easily. However, the delta analysis also helps to reveal missed matches, as a smaller amount of matches has to be examined.

4.3 Modification 2: Allow reading the cart content

As we are fans of the "wish list" feature of Amazon, we also want to implement this (in a "light" version) for our shopping cart system. Therefore, the system is changed one more time to allow customers to inspect other customer's shopping carts, so that friends of a certain customer can see what present he would be interested in. This change is implemented by adding a method `showItem()` to the class `ShoppingCart`.

Unfortunately, this new functionality does not work as expected: users are not able to inspect a different customer's shopping cart; instead, an authorisation error occurs (see figure 6). We use our tool to analyze the system and find the error. We take a snapshot of the current state, and compare it to the last working state of the system (after adding persistence) with our tool.

As shown in figure 3 markers for some unexpected advice calls from within `Authentication` and `Persistence` referring to the pointcut `itemChanges` appeared.

Looking further at this pointcut, we see that it is also – erroneously – bound to the newly introduced method `showItem()`. Once knowing this, it is easy to modify the pointcut so that it matches only the methods which in fact change the shopping cart (and do not just read it).

To be sure that this modification is correct, the new system state is compared to the previous one. To achieve a better overview than markers could offer, we do not use the task / editor views for this, but the *PointcutDiff-TreeView*. Within this view, the user can choose any two snapshots he took from the system, and all changes between the two snapshots are shown by presenting two trees: the left tree shows all items that are no longer matched in the system, the right one displays the elements which are additionally matched in the new version. As shown in figure 4, the spurious matches for `showItem()` disappeared.

The system is now back in a consistent state. We can verify this by comparing the state after adding persistence with the final state – as expected, no differences in the set of matches joinpoints occur.

Finally, we show the output of our system for two versions of our system. The first version demonstrates the base functionality of our system. The second version shows the final functionality, after adding (but before fixing) the 'wish list'-feature.

```
Customer Maximilian (Foo Road 123/1) -- starting.
Adding item 1
Adding item 2
Removing item 1
Customer Maximilian (Foo Road 123/1) -- finished.
Customer Christian (Bar Street 321/2) -- starting.
Adding item 1
Illegal Access - denied.
Customer Christian (Bar Street 321/2) -- finished.
```

Figure 5: Version one: base system output.

```
Customer Maximilian (Foo Road 123/1) -- starting.
Adding item 1
Saving Shopping Cart to DB ...
Adding item 2
Saving Shopping Cart to DB ...
Removing item 1
Saving Shopping Cart to DB ...
Customer Maximilian (Foo Road 123/1) -- finished.
Customer Grandma (Foobar Av./2) -- starting.
Inspecting item 1
shoppingCart.AccessException: Illegal Access - denied.
    at shoppingCart.aspects.Authentication.ajc
      $before$shoppingCart_aspects_Authentication$c2(
      Authentication.java:12)
    at shoppingCart.Customer.inspectItem(Customer.java:28)
    at shoppingCart.testDrivers.ListOfWishesDemo.main(
      ListOfWishesDemo.java:24)
Exception in thread "main"
```

Figure 6: Version two: flawed wish list.

Although this example is considerable simple, we think it can give a good idea how our tool can help to find program flaws introduced due to accidentally matched joinpoints. Note that the tool captures differences due to

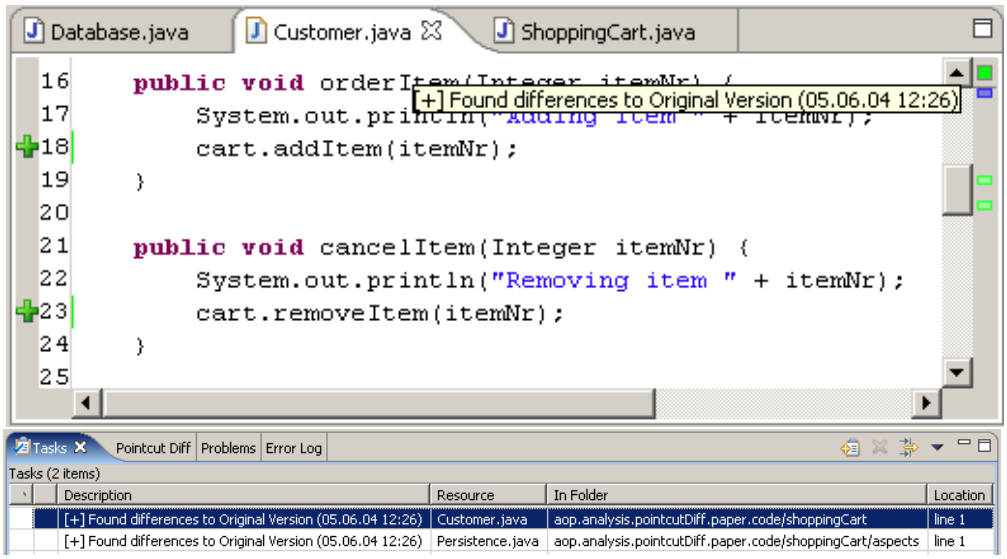

Figure 2: Changes between the original and the modified version

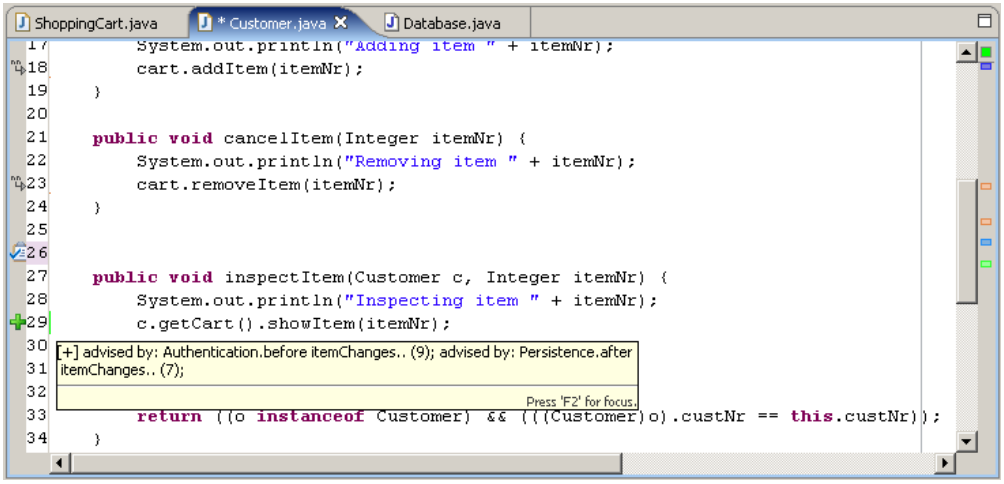

Figure 3: Unexpected advice calls

modified pointcut definitions as well as differences due to changes in the base code.

5 Conclusion

In this paper we claimed that current mainstream pointcut languages are not satisfactory, as they suffer from the fragile joinpoint problem. Although improvement of pointcut languages is a research topic and might well solve this problem one day, we proposed an analysis to deal with this problem for current languages, based on a comparison of the sets of matched joinpoints for two program versions.

We implemented this analysis in a tool which is available as an Eclipse plugin extending ajdt[3]. Results from our example are promising. We hope that the pointcut delta analysis is a useful tool to help programmers find bugs introduced into their software by breaking pointcuts, either directly or by modifying the base system.

Future work clearly has to address dynamic joinpoints.

[3]The tool is available under the terms of the CPL via our Eclipse Update Site: www.infosun.fmi.uni-passau.de/st/staff/stoerzer/PCDiff. Feedback is welcome!

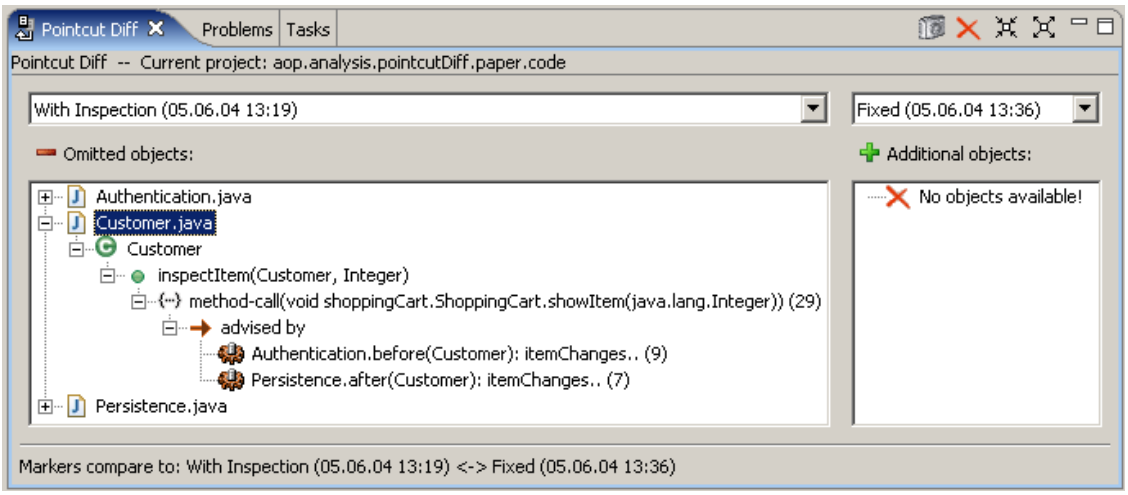

Figure 4: Advice no longer affects `showItem()`

We only addressed this topic in a footnote, but highly dynamic pointcuts can greatly reduce the value of deltas, as currently the comparison used a very conservative approximation for dynamic pointcuts: `cflow` and `if` primitive pointcuts are always approximated with `true`. This can definitely be improved.

We see our work related to many other efforts to improve program understanding. We addressed the fragile pointcut problem in an earlier paper [9]. The ajdt-development tools [1] also clearly address these topics, although the current version does not contain any support for pointcut deltas. But while ajdt statically analyzes a single program version to provide valuable feedback for the user, we are using two (or more) versions to analyze *their differences* to support system evolution and by that are rather related to other Change Impact Analysis approaches.

An approach to better approximate the `cflow` pointcut is presented in [8]. Partial evaluation [7] may also be useful to better approximate dynamic joinpoints.

References

[1] Adrian Colyer Andy Clement and Mik Kersten. Aspect-oriented programming with ajdt. In *Proceedings of AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003*, July 2003.

[2] Gregor Kiczales et. al. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[3] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness, 2000.

[4] Kris Gybels. Using a logic language to express cross-cutting through dynamic joinpoints.

[5] Gregor Kiczales. The fun has just begun. Keynote AOSD 2003, Boston, March 2003.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[7] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation Semantics of Aspect-Oriented Programs. In *Proc of workshop Foundations Of Aspect-Oriented Languages (FOAL) held in conjunction with AOSD 2002*. 2002.

[8] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 30–39. ACM Press, 2003.

[9] M. Störzer. Analytical problems and AspectJ. In *Proc. 3rd German Workshop on Aspect-Oriented Software Development, Essen, Germany*, March 2003.