*Journal of Universal Computer Science, vol. 20, no. 5 (2014), 692-719*
*submitted: 28/7/13, accepted: 15/2/14, appeared: 1/5/14 © J.UCS*

# Flexible Feature Binding with AspectJ-based Idioms

**Rodrigo Andrade**
(Federal University of Pernambuco, Recife, Brazil
rcaa2@cin.ufpe.br)

**Henrique Rebêlo**
(Federal University of Pernambuco, Recife, Brazil
hemr@cin.ufpe.br)

**Márcio Ribeiro**
(Federal University of Alagoas, Maceió, Brazil
marcio@ic.ufal.br)

**Paulo Borba**
(Federal University of Pernambuco, Recife, Brazil
phmb@cin.ufpe.br)

**Abstract:** In Software Product Lines (SPL), we can bind reusable features to compose a product at different times, which in general are static or dynamic. The former allows customizability without any overhead at runtime. On the other hand, the latter allows feature activation or deactivation while running the application with the cost of performance and memory consumption. To implement features, we might use aspect-oriented programming (AOP), in which aspects enable a clear separation between invariable code and variable code. In this context, recent work provides AspectJ-based idioms to implement flexible feature binding. However, we identified some design deficiencies. Thus, to solve the issues of these idioms, we incrementally create three AspectJ-based idioms. We apply these idioms to provide flexible binding for 16 features from five different product lines. Moreover, to evaluate our idioms, we quantitatively analyze them with respect to code cloning, scattering, tangling, and size by means of software metrics. Besides that, we qualitatively discuss our idioms in terms of code reusability, changeability, instrumentation overhead, behavior, and feature interaction. In conclusion, we show evidences that our idioms address the issues of those existing ones.
**Key Words:** Software Product Lines, Aspect-Oriented Programming, Idioms, Flexible Feature Binding
**Category:** D.1.m, D.2.8, D.2.13

## 1 Introduction

A Software Product Line (SPL) is a family of software-intensive systems developed from reusable assets. By reusing such assets, it is possible to construct a large number of different products applying compositions of different features [Pohl et al., 2005].

Depending on requirements and composition mechanisms, features should be activated or deactivated at different times. In this context, features may be bound statically, which could be, for instance, at compile time or preprocessing. The benefit of this approach is to facilitate the applications' customizability without any overhead at runtime [Rosenmüller et al., 2011a]. Therefore, this static feature binding is suitable for applications running on devices with constrained resources, such as certain mobile phones. On the other hand, features may be bound dynamically (e.g. at runtime) to allow more flexibility, with the cost of performance and memory consumption. Furthermore, if developers do not know, before runtime, the set of features that should be activated, they could use dynamic feature binding to activate features on demand.

To support flexible binding for feature code implemented using aspects [Kiczales et al., 1997]—which is the technique we focus on this work—we proposed Layered Aspects [Andrade et al., 2011]. This solution makes it possible to choose between compile or runtime binding for selected features. Moreover, it reduces several problems identified in a previous work [Chakravarthy et al., 2008], such as code cloning, scattering, and tangling [Andrade et al., 2011]. Although these goals are achieved to some extent, Layered Aspects still has some deficiencies. It may introduce feature code scattering and instrumentation overhead to the flexible binding implementation. Additionally, applying Layered Aspects demands several changes, which could hamper the reuse of the flexible binding implementation.

Hence, to address the Layered Aspects issues and still have low rates of code cloning, scattering, and tangling, we define three idioms [Andrade et al., 2013a] based on AspectJ [Kiczales et al., 2001], which we call increments, as they incrementally address the Layered Aspects issues. In our context, we use the terminology idiom instead of pattern because our increments are more AspectJ specific and address a smaller and less general problem than a pattern.

The first idiom addresses part of the issues with the aid of Java annotations. The second idiom uses the @AspectJ syntax [Laddad, 2009] to address more Layered Aspects issues. However, this syntax does not support intertypes, so it may introduce problems, such as feature code scattering. In this context, due to AspectJ traditional syntax limitations, our final idiom uses a resource of AspectJ's compiler to address these issues without introducing @AspectJ syntax problems.

To evaluate these idioms, we extract code of 16 features from five different product lines (101Companies, ArgoUML, Freemind, BerkeleyDB, and Sudoku) and apply each idiom plus Layered Aspects to implement flexible binding for these features. Then, to evaluate whether our idioms do not present worse results than Layered Aspects with respect to code cloning, scattering, tangling, and size, we quantitatively assess the idioms by means of software metrics. To this end, we use seven metrics: Pairs of Cloned Code, Degree of Scattering across Components [Eaddy, 2008], Degree of Scattering across Operations [Eaddy, 2008], Degree of Tangling within Components [Eaddy, 2008], Degree of Tangling within Operations [Eaddy, 2008], Source Lines of Code, and Vocabulary Size. Additionally, we discuss the three idioms plus Layered Aspects regarding four factors: their code reusability, changeability, instrumentation overhead, and behavior based on our five product lines and also on our previous knowledge about this topic [Andrade et al., 2011, Ribeiro et al., 2009]. As result of this evaluation, we conclude that our final idiom addresses these three factors and does not present worse results regarding the software metrics.

This paper extends our previous work [Andrade et al., 2013a] in three ways. First, our evaluation now considers two additional metrics that help to reinforce our assessment results. These metrics [Eaddy, 2008] are degree of scattering across operations (pointcuts, methods, or advice) and degree of tangling within operations. In this way, we could strengthen some of our results and obtain new insights not only regarding code scattering and tangling in the level of classes and aspects but also regarding pointcuts, methods, and advice. Thus, we could conclude that Layered Aspects scatters code in the level of operations, although it does not in the level of classes. Moreover, we also conclude that our second and third idioms do not increase the degree of scattering or tangling in the level of operations. Measuring these two new metrics is important because code scattering or tangling may hinder reusability, for example, also at the level of operations. Second, we consider a new case study with three features and a number of feature interaction cases [Calder et al., 2003]. Thus, we could apply our idioms to a feature interaction scenario, extending the scope of our earlier evaluation and the validity of our results in a different context. We consider such scenario because feature interaction may be damaging to application development and user expectations [Calder et al., 2003]. Third, we use the SafeRefactor tool [Soares et al., 2010] to gather evidence that using our idioms preserve feature behavior, so that they can likely be used as refactoring targets for existing systems.

In summary, the contributions of this paper are:

1. We identify deficiencies in an existing idiom (Layered Aspects) for flexible feature binding;

2. We address these deficiencies by incrementally defining three idioms for flexible feature binding;

3. We apply these four idioms to provide flexible binding for 16 features of five case studies;

4. We quantitatively assess the three idioms plus the existing one with respect to code cloning, scattering, tangling, and size by means of software metrics;

5. We discuss the idioms regarding reusability, changeability, code instrumentation overhead, behavior, and feature interaction;

At last, we structure the remainder of this paper as follows. In Section 2, we present the motivation of our work, detailing the Layered Aspects issues.

Section 3 introduces our three idioms to address these issues. In Section 4, we present the evaluation of Layered Aspects and our three idioms regarding code cloning, scattering, tangling, and size, a qualitative discussion and the threats to validity. Finally, Section 5 discusses related work, and Section 6 concludes this article.

## 2 Motivating Example

This section presents the Layered Aspects issues by showing the implementation of flexible binding for an optional feature of the 101Companies SPL. This product line is based on a Java version of the 101Companies project [Favre et al., 2012], which aims at developing a free, structured, wiki-accessible knowledge resource including an open-source repository. This project defines several features so that developers can implement them using different programming languages or technologies and share with everyone. The optional feature we consider in this section is called *Total* and represents the total salary of a given employee, the sum of all department salaries, or the sum of all company salaries. We omit further detail about this SPL because we only focus and use the *Total* optional feature throughout this section.

As mentioned in the previous section, to provide flexible feature binding, we could use the Layered Aspects idiom [Andrade et al., 2011], which makes it possible to choose between static (compile time) and dynamic (runtime) binding for features. Basically, the structure of this idiom includes three aspects. One abstract aspect implements the feature code whereas two concrete subaspects implement static and dynamic feature binding. Listing 1 illustrates part of the *Total* feature code implemented using aspects, consisting of pointcuts (Line 3), advice (Line 6), intertype declarations (Line 11), and `private` methods, which we omit for simplicity. To apply Layered Aspects, we need to change the `TotalFeature` aspect by including the `abstract` keyword in Line 1. This allows the concrete subaspects to inherit from `TotalFeature`, since only abstract aspects can be inherited in AspectJ [Laddad, 2009].

**Listing 1:** TotalFeature aspect

```
1   privileged aspect TotalFeature {
2
3     pointcut newAbstractView(AbstractView cthis) :
4      execution(AbstractView.new(..)) && this(cthis);
5
6     void around(AbstractView cthis) : newAbstractView(cthis) {
7      proceed(cthis);
8      cthis.total = new JTextField();
9     }
10
11    private JTextField AbstractView.total;
12    ...
13  }
```

To implement static binding, we define `TotalStatic`, which is an empty concrete subaspect that inherits from `TotalFeature` aspect. Thus, we are able to statically activate the feature execution by including both aspects in the project build.

Before explaining the dynamic feature (de)activation, we first need to introduce an important concept used in this article: the driver [Andrade et al., 2011]. This is the mechanism responsible for dynamically activating or deactivating feature code execution. It may vary from a simple user interface prompt to complex sensors, which decide by themselves whether the feature should be activated [Ribeiro et al., 2009]. In our case, the driver mechanism reads a property value from a properties file. For instance, to dynamically activate the *Total* feature, we would set `total=true` in the properties file. We do this for simplicity, since the complexity about providing information for feature activation is out of the scope of this work.

To implement dynamic binding for the *Total* feature, we define `TotalDynamic`, as showed in Listing 2. Line 3 defines an `if` pointcut to capture the driver's value. To allow dynamic feature binding, Lines 5-8 define an `adviceexecution` pointcut to deal only with `before` and `after` advice. Thus, it is possible to execute those pieces of advice defined in `TotalFeature` aspect (Listing 1) depending on the driver's value. For instance, the feature code within a `before` or `after` advice in `TotalFeature` aspect is executed if the driver condition is set to true in Line 3 of Listing 2. In this case, the `adviceexecution` pointcut does not match any join point in `TotalFeature` because the driver is negated in Line 6, and therefore, the feature code is executed. On the other hand, if the driver condition is false, the `adviceexecution` pointcut matches some join points. However, feature code is not executed because we do not call `proceed`. Additionally, returning `null` in Line 7 is not harmful when the feature is deactivated because Layered Aspects does not use the `adviceexecution` pointcut for `around` advice [Andrade et al., 2011].

**Listing 2:** Layered Aspects TotalDynamic aspect

```
1  aspect TotalDynamic extends TotalFeature {
2
3   pointcut driver() : if (Driver.isActivated("total"));
4
5   Object around() : adviceexecution() && within(TotalFeature)
6    && !driver() {
7     return null;
8   }
9
10  pointcut newAbstractView(AbstractView cthis) :
11   TotalFeature.newAbstractView(cthis) && driver();
12 }
```

Thereby, Layered Aspects design states that the pieces of `around` advice of the feature code must be deactivated one-by-one because the `adviceexecution` pointcut could lead to problems when the driver states the feature deactivation [Andrade et al., 2011]. For such a scenario, we would miss the invariable code execution, since the `around` advice matched by the `adviceexecution` would not be executed and, consequently, the `proceed()` of the `around` advice would not be executed either, which leads to missing the invariable code execution that is independent of the activation or deactivation of features.

Thus, to avoid this problem, Layered Aspects associates the driver with each pointcut related to an `around` advice defined in `TotalFeature` as showed in Lines 10 and 11. These lines redefine the `newAbstractView` pointcut and associate it with the driver. Thus, the code within the `around` advice defined in Listing 1 is executed only if the driver's is set to true, that is, the feature is activated. The redefinition of pointcuts for such cases is the reason why the `TotalDynamic` needs to inherit from `TotalFeature` [Andrade et al., 2011], so the latter needs to be an abstract aspect, since AspectJ does not provide a way to inherit from a concrete aspect.

In this context, we may observe three main issues when applying Layered Aspects to implement flexible feature binding. First, the `adviceexecution` pointcut unnecessarily matches all pieces of advice within the feature code, including `around` advice. As mentioned, the `adviceexecution` is used only for `before` and `after` advice. This issue may cause overhead in byte code instrumentation. Additionally, returning `null` within `adviceexecution` pointcut is not a very elegant solution, even though this situation is not error-prone, as mentioned.

The second issue is the empty concrete subaspect to implement static feature binding. We have to define it due to the AspectJ limitation, in which an aspect can inherit from another only if the latter is abstract. So this subaspect is imperative for static feature activation, since it allows feature code instantiation. This may increase code scattering because we need an empty subaspect for each abstract aspect that implements feature code. For instance, we had to implement 18 empty concrete aspects to implement static binding for our 16 selected features.

Another issue is the pointcut redefinition, which is applied when a pointcut within the feature code is related to an `around` advice. In this context, if there are a large number of `around` advice, we would need to redefine each pointcut related to them, which could lead to low productivity or even make the task of maintaining such a code hard and error-prone. Therefore, this issue could hinder code reusability and changeability.

Hence, we enumerate the main goals we try to address with the idioms:

1. To prevent `adviceexecution` pointcut to unnecessarily match `around` advice;
2. To avoid the empty concrete subaspect to implement static binding;
3. To eliminate the need of redefining each pointcut related to an `around` advice within the concrete subaspect to implement the dynamic binding.

We believe that defining idioms to address these issues may bring benefits, such as code scattering reduction, increase of reusability and changeability, and decrease of instrumentation overhead. We discuss these improvements throughout the next sections.

## 3 Idioms for flexible binding

In this section, we illustrate our three idioms [Andrade et al., 2013a]. To perform this, we apply each idiom to implement flexible binding for the *Total* feature from the 101Companies SPL. We point out the advantages and disadvantages of each increment and how they address the issues presented in Section 2. Although we conclude that the AroundClosure idiom is the best solution in Section 4, the other idioms are also complete solutions and can be used by developers, which should be aware of their limitations, as explained throughout this work.

Moreover, for the examples in the following sections, we consider the same 101Companies SPL source code. More specifically, we replicate this source code so that we could apply each idiom for the code of its features.

### 3.1 First increment: AnnotatedBind

For this increment, we try to prevent `adviceexecution` pointcut to match `around` advice within feature code, which corresponds to the first issue. To achieve that, we use an AspectJ 5 mechanism, which includes the support for matching join points based on the presence of Java 5 annotations [Laddad, 2009].

In this context, we create an `AroundAdvice` annotation and use it to annotate all pieces of `around` advice within the feature code, as depicted in Line 3 of Listing 3. In this way, we can prevent `adviceexecution` pointcut to match any of these annotated advice when applying dynamic binding.

**Listing 3:** Annotated around advice

```
1 abstract privileged aspect TotalFeature {
2   ...
3   @AroundAdvice
4   void around(AbstractView cthis) : newAbstractView(cthis) {
5    proceed(cthis);
6    cthis.total = new JTextField();
7   }
8 }
```

To implement the static feature binding, we include the `TotalFeature` and `TotalStatic` aspects plus the `Total` class in the project build. In its turn, to implement the dynamic feature binding, we change the `adviceexecution` pointcut by adding the `!@annotation(AroundAdvice)` clause. Thus, this pointcut does not match the pieces of `around` advice defined in `TotalFeature`. In Listing 4, we show the `adviceexecution` pointcut with the `!@annotation(AroundAdvice)` clause, which is the part that differs from Listing 2. Therefore, we resolve the first Layered Aspects issue. However, the other two issues remain open. To address them, we introduce more increments next.

**Listing 4:** TotalDynamic aspect with the AnnotatedBind idiom

```
aspect TotalDynamic extends TotalFeature {
  ...
 void around() : adviceexecution() && within(TotalFeature)
  && !@annotation(AroundAdvice) {
   if (Driver.isActivated("total")) { proceed(); }
 }
}
```

### 3.2 Second increment: @Proceed

For this increment, we try to address the second and third Layered Aspects issues, which correspond to avoiding the empty concrete subaspect to implement static binding and to eliminating the need of redefining each pointcut related to `around` advice, as explained in Section 2.

To achieve that, we use the new @AspectJ syntax [Laddad, 2009], which offers the option of compiling source code with a plain Java compiler. This syntax demands that the feature code elements are annotated with provided annotations, such as `@Aspect`, `@Pointcut`, and `@Around`. Listing 5 illustrates part of the `TotalFeature` class, which contains feature code similarly to Listing 1. The main differences are the annotations in Lines 1, 4, and 7, which are used in collusion with their parameters to define an aspect, pointcut, and advice, respectively.

**Listing 5:** Total feature with the @Proceed idiom

```
1  @Aspect
2  class TotalFeature {
3     ...
4    @Pointcut("execution(AbstractView.new(..)) && this(cthis)")
5    public void newAbstractView(AbstractView cthis) {}
6
7    @Around("newAbstractView(cthis)")
8    void around1(AbstractView cthis, ProceedingJoinPoint pjp) {
9     pjp.proceed();
10    cthis.total = new JTextField();
11   }
12 }
```

However, the @AspectJ syntax presents some disadvantages. First, there is no way to declare a `privileged` aspect [Laddad, 2009], which is necessary to avoid creating an access method or changing invariable code element's visibility, such as changing from `private` to `public` to be visible within `TotalFeature` class.

Indeed, we had to change or add `get` methods for eight program elements only within the *Total* feature code. Second, this new syntax does not support intertype declarations [Laddad, 2009]. Therefore, we need to define an additional aspect, using the traditional AspectJ syntax, containing the intertype declarations.

Despite these limitations, we could eliminate the empty concrete aspect to implement the static feature binding. Since `TotalFeature` of Listing 5 is a class rather than an `abstract` aspect, we are able to instantiate it without the concrete subaspect. In this way, to statically activate the *Total* feature, we need to include the `TotalFeature` and `Total` classes, and the `TotalFeatureInter` aspect, which is the aspect containing intertype declarations, as explained.

To implement the dynamic feature binding, we use an `adviceexecution` pointcut, which matches `before`, `after`, and `around` advice. Hence, we do not need to redefine pointcuts related to `around` advice. Therefore, we address the third Layered Aspects issue. Listing 6 illustrates how this increment deals with dynamic feature binding. Lines 4-14 define an `adviceexecution` pointcut using the @AspectJ syntax in a similar way to the one defined in Listing 2. Besides the syntax, the difference is dealing with scenarios that the feature is dynamically deactivated. Thus, we define the `proceedAroundCallAtAspectJ` method in a separate class and call it in Line 10, which allows us to call the `proceed` join point of the matched pieces of advice defined within `TotalFeature`. Hence, even if the *Total* feature is dynamically deactivated, the execution of other functionalities are not compromised [Andrade et al., 2011]. Additionally, the `adviceexecution` pointcut is used for `before`, `after`, and `around` advice. Therefore, it does not unnecessarily match pieces of advice as the Layered Aspects idiom does. In this way, the first Layered Aspects issue remains solved.

**Listing 6:** TotalDynamic class for @Proceed idiom

```
1  @Aspect
2  public class TotalDynamic {
3   @Around("adviceexecution() && within(TotalFeature)")
4   public Object adviceexecutionIdiom(JoinPoint thisJoinPoint,
5    ProceedingJoinPoint pjp) {
6    Object ret;
7    if (Driver.isActivated("total")) {
8     ret = pjp.proceed();
9    } else {
10    ret = Util.proceedAroundCallAtAspectJ(thisJoinPoint);
11   }
12   return ret;
13  }
14 }
```

Albeit we address the three Layered Aspects issues with our @Proceed idiom, it still presents some undesired points. First, the @AspectJ syntax is limited: it does not support `privileged` aspects, intertype declarations, and exception handling [Laddad, 2009]. Furthermore, the pointcut and advice definitions within the annotation statement are verified only at weaving time rather than compile time with the traditional syntax. This could hamper code maintenance and error finding. Therefore, in the next increment, we try to keep addressing the three Layered Aspects issues without using the @AspectJ syntax.

### 3.3 Final increment: AroundClosure

Now, we improve our previous increment by addressing all the three Layered Aspects issues presented in Section 2, but without introducing the @AspectJ syntax deficiencies. To achieve that, we still need to avoid these three issues and use the traditional AspectJ syntax.

The AroundClosure idiom does not demand any changes in the feature code implementation showed in Listing 1. Thus, to provide flexible binding to the Total feature with AroundClosure, we need `Total` class plus the `TotalFeature`, and `TotalDynamic` aspects, as showed in Listing 1, and 7, respectively.

In this context, since `TotalFeature` is not an abstract aspect like in Layered Aspects or our first increment (AnnotatedBind), it is not necessary to have an empty abstract aspect to implement static feature binding. We just include the `TotalFeature` aspect and `Total` class in the project build to statically activate the *Total* feature.

Further, to implement the dynamic feature binding, we define the `TotalDynamic` aspect, as illustrated in Listing 7. We define a generic advice using `adviceexecution` pointcut that works with `before`, `after`, and `around` advice. Hence, we do not need to redefine each pointcut within the feature implementation that is related to an `around` advice. Thereby, `TotalDynamic` does not extend `TotalFeature`, so the abstract aspect is no longer needed.

**Listing 7:** TotalDynamic aspect with AroundClosure

```
1 aspect TotalDynamic {
2  Object around() : adviceexecution() && within(TotalFeature) {
3   if (Driver.isActivated("total")) {
4    return proceed();
5   } else {
6    return Util.proceedAroundCall(thisJoinPoint);
7   }
8  }
9 }
```

More specifically, to deal with dynamic feature binding, we just call `proceed()` in Line 4, so the feature code within the advice defined in `TotalFeature` is executed. We have to define the `around` advice as returning an Object in Line 2 to make it generic, avoiding compilation errors when an `around` advice, that is not `void`, is present in the feature implementation.

On the other hand, it is not trivial to deal with the scenario in which the feature is dynamically deactivated due to `around` advice. This kind of advice uses a special form (`proceed`) to continue with the normal invariable code flow of execution at the corresponding join point. This special form is implemented by generating a method that takes in all of the original arguments to the `around` advice plus an additional AroundClosure object that encapsulates the invariable code flow of execution [Hilsdale and Hugunin, 2004], which has been interrupted by the pieces of advice related to the feature and afterwards interrupted by the `adviceexecution` pointcut. Thus, in Line 6, we call the `proceedAroundCall` method passing as argument `thisJoinPoint`, which contains reflective information about the current join point of the feature code advice that `adviceexecution` is matching.

To avoid missing the invariable code flow of execution when the feature is dynamically deactivated, Listing 8 defines part of the `proceedAroundCall` method. First, we obtain an array with the arguments of the matched advice through the `thisJoinPoint` information in Line 3. By means of this array we obtain the AspectJ `AroundClosure` object. Thus, we directly call the `AroundClosure` method `run` in Line 6, which executes the invariable code. This `run` method is automatically called under the hood by the `proceed` of each `around` advice. However, since we miss this `proceed` when the feature is dynamically deactivated, we need to manually call `run` so that we do not miss the invariable code execution.

As explained, this idiom uses the `AroundClosure` object, which is an internal resource of AspectJ's compiler. Therefore, to the correct operation of this idiom, the `AroundClosure` object must be present in the compiler. Although we focus only on AspectJ, other AOP-based compilers also include this object [Aracic et al., 2006, Avgustinov et al., 2005].

**Listing 8:** The proceedAroundCall method

```
1  static Object proceedAroundCall(JoinPoint thisJoinPoint) {
2   ...
3   Object[] args = thisJoinPoint.getArgs();
4   int i = (args.length - 1);
5   if (args[i] instanceof AroundClosure) {
6    return ((AroundClosure) args[i]).run(args);
7   }
8  }
```

At last, the AroundClosure idiom addresses the Layered Aspects issues without introducing the @AspectJ syntax problems. We evaluate our three idioms plus Layered Aspects in the next section.

## 4 Evaluation

In this section, we explain our evaluation. The Section 4.1 presents the selected case studies and the main procedures we follow to conduct our evaluation. In Section 4.2, we quantitatively evaluate our idioms and Layered Aspects in a similar way we did in our previous work [Andrade et al., 2011] to avoid bias. Besides that, we discuss our three idioms and Layered Aspects regarding code reusability, changeability, instrumentation overhead, behavior, and feature interaction in Section 4.3.

### 4.1 Study Settings

We consider 16 features of five case studies: two features of 101Companies [Favre et al., 2012], eight features of BerkeleyDB [Kästner et al., 2007], one feature of ArgoUML [Tigris, 2013], two features of Freemind [Müller et al., 2013], and three features of Sudoku [Kästner, 2013], which is our new case study. Besides 101Companies and Sudoku, the other three case studies are the same of our previous work [Andrade et al., 2011]. This is important to show the gains obtained with the idioms on the top of the same features. In this way, we avoid biases such as implementing flexible binding for feature that present different degree of scattering or tangling. In Table 1, we map the 16 features to the respective case study. These case studies represent different sizes, purposes, architectures, granularity, and complexity. Moreover, the code of their features present different types, such as optional or alternative features [Kang et al., 1990].

**Table 1:** Case study and features

| Case study | Features |
|---|---|
| Freemind | Icons and Clouds |
| ArgoUML | Guillemets |
| 101Companies | Total and Cut |
| BerkeleyDB | EnvironmentLock, Checksum, Delete, LookAheadCache, Evictor, NIO, IO, and INCompressor |
| Sudoku | Solver, Undo, and Guesser |

To perform our evaluation, we follow four main procedures, as explained next. However, we do not execute the first and second procedures for the BerkeleyDB case study because it already existed, as we discuss in Section 4.4.

First, to create the product lines from the original code of these case studies, we assigned the code of their features by using the prune dependency rules [Eaddy et al., 2007], which state that "a program element is relevant to a feature if it should be removed, or otherwise altered, when the feature is pruned from the application". By following these rules, we could identify all the code related to the features. We chose this rule to reduce introducing bias while identifying feature code.

Second, we extracted part of the feature code that is tangled with invariable code into AspectJ aspects. However, the code within some classes is not extracted into aspects when the whole class is only relevant to the feature. Thus, this feature code is not tangled or scattered throughout the invariable code. Additionally, there are references to the elements of these classes only within the feature code. Each feature code is localized in a different and unique package, which contains aspects and, possibly, classes. In summary, there are two procedures: (i) we extract feature code that is tangled with invariable code into aspects and (ii) we move classes that contain only the feature code into the package created specifically for this feature.

Third, to evaluate our three idioms and Layered Aspects, we applied each one of our three idioms plus Layered Aspects to implement flexible binding for the 16 features of the five case studies.

For the 101Companies, we apply each one of our three idioms plus Layered Aspects to implement flexible binding for its two features. This product line has nearly 900 lines of code whereas 300 of code for the two selected features.

For BerkeleyDB, we apply the four idioms to implement flexible binding for eight features of the BerkeleyDB product line [Kästner et al., 2007]. This product line has around 32000 lines of code whereas the eight selected features sum up approximately 2300 lines of code. This allows us to test our AroundClosure idiom and the increments in a large and widely used application.

For ArgoUML, we create a product line by extracting the code of one feature into AspectJ aspects. Then, we apply the four idioms presented to implement flexible binding for these features. Our ArgoUML product line has nearly 113000 lines of code and 200 of feature *Guillemets* code.

For Freemind and Sudoku, we also extract the code of five features into AspectJ aspects. Then, we apply the four idioms to provide flexible feature binding for these features. The Freemind product line has about 67000 lines of code and both selected features have approximately 4000 lines. The Sudoku product line has 2100 lines of code and its two features sum up 250 lines.

Fourth, we collect the number of lines of code (LOC) of relevant components, such as feature or driver code, to provide as input to compute the metrics. We use the Google CodePro AnalytiX[1] to obtain the LOC and we use sheets to help the computation of the metrics. Moreover, we detail the selected metrics and results in Section 4.2.

### 4.2 Quantitative analysis

To drive the quantitative evaluation of our idioms, we follow the Goal-Question-Metric (GQM) design [Basili et al., 1994]. We structure it in Table 2. We use Pairs of Cloned Code in Section 4.2.1 to answer Question 1, as it may indicate a design that could increase maintenance costs [Baxter et al., 1998] because a change would have to be done twice to the duplicated code. To answer Question

[1] `https://developers.google.com/java-dev-tools/download-codepro`

2, we use Degree of Scattering across Components [Eaddy, 2008] and Degree of Scattering across Operations [Eaddy, 2008] in Section 4.2.2 to measure the implementation scattering for each idiom regarding driver and feature code. To answer Question 3, we measure the tangling between driver and feature code considering the Degree of Tangling within Components [Eaddy, 2008] and Degree of Tangling within Operations [Eaddy, 2008] metrics in Section 4.2.3. Furthermore, Source Lines of Code and Vocabulary Size are well known metrics for quantifying a module size and complexity. So, in Section 4.2.4, we answer Question 4 measuring the size of each idiom in terms of lines of code and number of components. Albeit we show only part of the graphs and data in this section, we provide them completely elsewhere [Andrade et al., 2013b].

**Table 2:** GQM

| | |
|---|---|
| **Goal** | |
| `Purpose` | Evaluate idioms regarding |
| `Issue` | cloning, scattering, tangling, and size of their flexible binding implementation |
| `Object` | for features |
| `Viewpoint` | from a software engineer viewpoint |
| **Questions and Metrics** | |
| **Q1- Do the idioms increase code cloning?** | |
| `Pairs of Cloned Code` | `PCC` |
| **Q2- Do the idioms increase driver and feature code scattering?** | |
| `Degree of Scattering across Components` | `DOSC` |
| `Degree of Scattering across Operations` | `DOSO` |
| **Q3- Do the idioms increase tangling between driver and feature code?** | |
| `Degree of Tangling within Components` | `DOTC` |
| `Degree of Tangling within Operations` | `DOTO` |
| **Q4- Do the idioms increase lines of code and number of components?** | |
| `Source Lines of Code` | `SLOC` |
| `Vocabulary Size` | `VS` |

#### 4.2.1 Cloning

To answer Question 1 and investigate whether our idioms increase code cloning, we use the CCFinder [Kamiya et al., 2002] tool to obtain the PCC metric results. CCFinder is a widely used tool [Kamiya et al., 2013] to detect cloned code [School and Rajapakse, 2005, Kapser and Godfrey, 2006, Bruntink et al., 2005]. Similarly to our previous work [Andrade et al., 2011], we use 12 as the token set size (TKS) and 40 as the minimum clone length (in tokens) to preset the tool, which means that to be considered cloned, two pairs of code must have at least 40 equal tokens.

In general, the four idioms present similar results. There is no code replication for 11 features out of 16 regarding the four idioms. Additionally, the idioms lead to low PCC rates for the code of these five features that present code replication [Andrade et al., 2013b]. Therefore, our idioms do not increase code cloning. This answers Question 1.

### 4.2.2 Scattering

To answer Question 2, we use DOSC and DOSO to analyze feature and driver code scattering for each idiom. Feature and driver are different concerns, so we analyze them separately. Although, the only way we could measure the driver code scattering is after an idiom is applied to provide flexible binding for the selected features. In this way, we discuss driver code scattering considering the four idioms applied to the selected features.

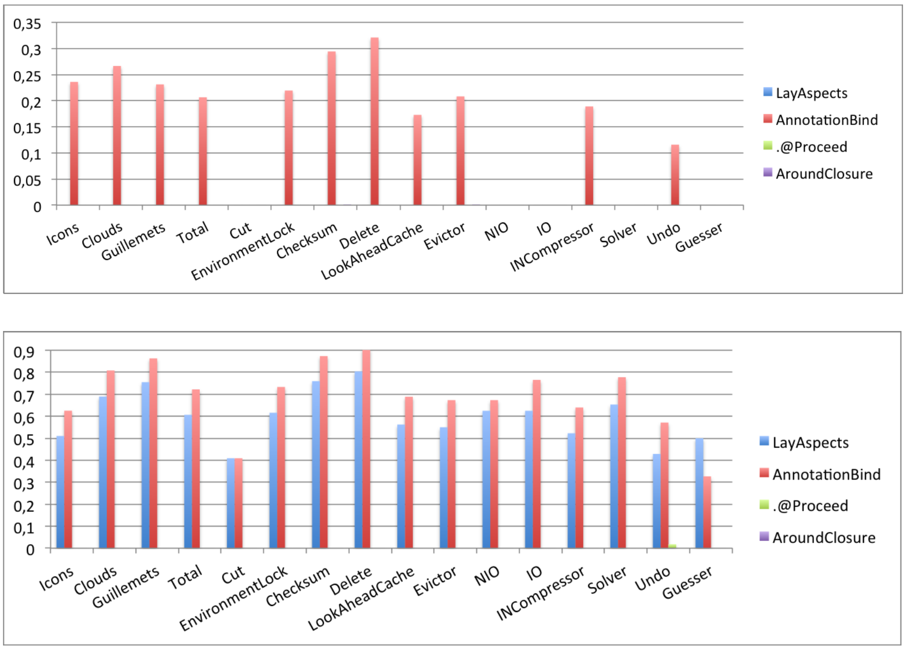


**Figure 1:** DOSC and DOSO for driver

**Driver**. On the upper side of Figure 1, we present the results regarding the DOSC metric. The only idiom that presents driver scattering is our Annotated-Bind idiom. This occurs due to the annotations we must add to `around` advice defined within the feature code, as explained in Section 3.1. This may hinder code reusability and changeability. However, the AnnotatedBind idiom reduces the byte code instrumentation, as we discuss in Section 4.3. Additionally, features *Cut* and *Guesser* do not present an `around` advice, therefore there is no `AroundAdvice` annotation in its code. The *NIO*, *IO*, and *Solver* features only present `around` advice, thus there is no need to add the `AroundAdvice` annotation, as only one pointcut redefinition implements the driver.

Nevertheless, the new considered metric shows different results on the bottom side of Figure 1. By means of the DOSO metric, we identify that the Layered Aspects [Andrade et al., 2011] idiom implementation scatters driver code at the operation level. This happens because we need to associate driver code with the redefined pointcuts related to `around` advice. Therefore, the driver code could be present in many redefined pointcuts. Although, @Proceed and AroundClosure do not scatter driver code throughout methods, pointcuts, or advice.

Furthermore, @Proceed and AroundClosure do not present any driver code scattering, since their driver is implemented within a unique aspect and advice for each idiom.

**Feature**. Figure 2 illustrates the DOSC results considering features. In this context, our @Proceed idiom presents a disadvantage when compared to the others. This happens because the @AspectJ syntax, which is used by the @Proceed idiom, does not support intertype declarations. Thus, as explained in Section 3.2, this idiom needs an additional AspectJ aspect (traditional syntax) to implement the intertype declarations, which contributes to scatter feature code across at least two components. On the other hand, the features *NIO*, *IO*, and *Guesser* do not present intertype declarations within their implementation. Thus, our @Proceed idiom does not scatter feature code in these cases. Our AnnotatedBind idiom and Layered Aspects present similar results because the implementation of these two idioms are similar regarding feature code. Additionally, the AroundClosure idiom only presents feature code scattering when more than one aspect is used to implement feature code, which is the case of the *Delete* feature.

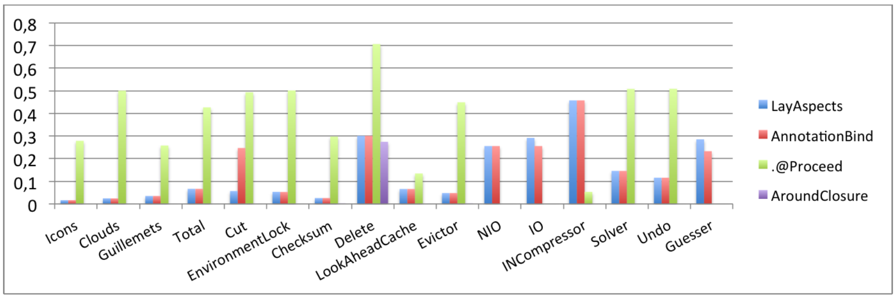


**Figure 2:** DOSC for feature

Furthermore, the DOSO results for features are equal to the four considered idioms. Since applying the idioms does not change the feature code, the idiom implementation needs the same number of pieces of advice, which contain the same feature code. Therefore, we omit the graph with the DOSO metric results.

At last, we answer Question 2 saying that AnnotatedBind increases driver code scattering whereas our @Proceed idiom increases feature code scattering. However, our final solution (AroundClosure) does not present driver scattering. Additionally, it does not increase feature code scattering.

### 4.2.3 Tangling

This section answers Question 3 by investigating the extent of tangling between feature and driver code. According to the principle of separation of concerns [Parnas, 1972], one should be able to implement and reason about each concern independently.

Similarly to our previous work [Andrade et al., 2011], we also assume that the greater the tangling between feature code and its driver code, the worse the separation of those concerns. Thus, we measure the Degree of Tangling within Components (DOTC) and the Degree of Tangling within Operations (DOTO).

On the upper side of Figure 3, we show the DOTC metric results. Only the AnnotatedBind idiom presents tangling between two concerns: driver and feature. This happens due to the `AroundAdvice` annotation included within the aspects that implement feature code. On the other hand, @Proceed and AroundClosure present no tangling between driver and feature code. For example, Listings 6 and 7 contain only driver code by following the prune dependency rule, that is, the code defined within `TotalDynamic` class and aspect is relevant only to the driver concern. In this way, these idioms comply with the results obtained for Layered Aspects. The features *Cut*, *Undo*, and *Guesser* do not present `around` advice and therefore none `AroundAdvice` annotation. On the other hand, the features *IO*, *NIO*, and *Solver* only present `around` advice, thus there is no need to introduce the `AroundAdvice` annotation since there is no code instrumentation overhead.

Moreover, we reinforce our findings [Andrade et al., 2013a] with the DOTO results showed in the bottom side of Figure 3. As illustrated, only the AnnotatedBind idiom presents tangling between feature and driver code at the operation level. This happens for the same reason explained before for the DOTC results.

Thus, we conclude that our AnnotatedBind idiom increases the tangling between driver and feature code. However, @Proceed and AroundClosure does not present tangling at all. This answers Question 3.

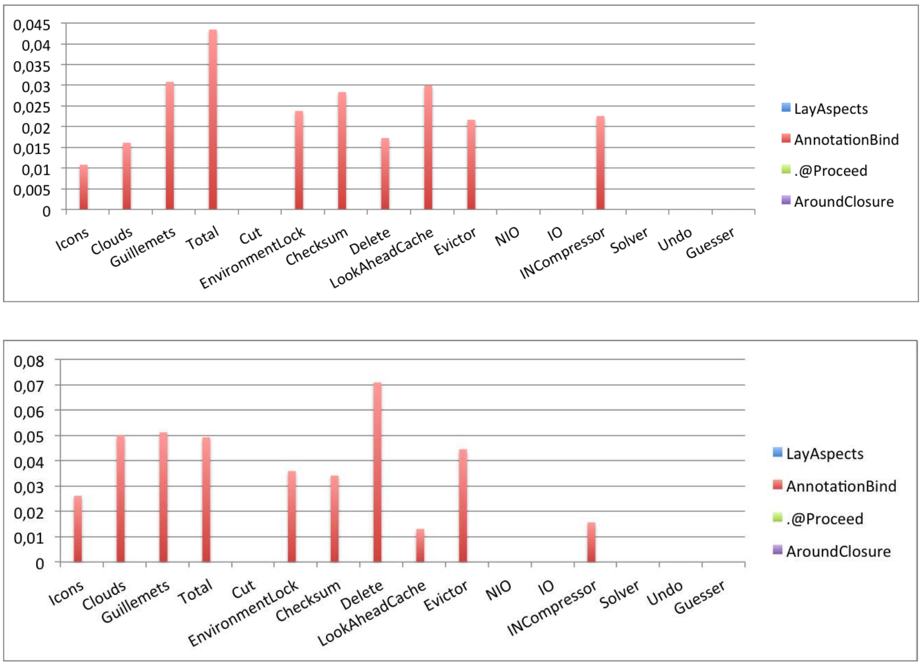


**Figure 3:** DOTC and DOTO

#### 4.2.4 Size

To identify the idiom that increases the size of its implementation, we try to answer Question 4. For this purpose, we use the SLOC and VS metrics.

In this context, the differences between the four idioms is insignificant for SLOC and VS metrics. For instance, the *Icons* feature presents between 2155 and 2186 source lines of code for the smallest and largest idiom implementation, respectively. This represents a difference of only 1.41% of the feature implementation. Similarly, the differences between the four idioms for the VS metric results are also insignificant. Therefore, we answer Question 4 stating that our idioms do not increase lines of code and number of components.

### 4.3 Qualitative discussion

In this section, we qualitatively discuss Layered Aspects and our three idioms in terms of code reusability, changeability, and instrumentation overhead. Furthermore, we discuss the use of SafeRefactor [Soares et al., 2010] to check any behavioral changes between the flexible feature binding using different idioms and a scenario in which features interact.

**Reusability** is related to how easily we can reuse the flexible binding implementation using an idiom. Therefore, we are interested in checking what we need to do to reuse a given idiom code when applying it to another feature.

*Layered Aspects and AnnotatedBind*. We may have to perform several changes to reuse the code of the implementation of these idioms. Only if the features we aim at applying flexible binding do not present any `around` advice within its implementation, then we would perform few changes to reuse the code of these idioms between the features, since the `adviceexecution` pointcut is reused as it applies to all `before` and `after` advice. However, Layered Aspects and our AnnotatedBind redefine the pointcuts related to `around` advice, which hinders reuse since these pointcuts are associated to a particular feature. Hence, this compromises the overall reusability of the implementation of both idioms.

*@Proceed and AroundClosure*. Few changes are needed to reuse the code of both idioms. The `adviceexecution` pointcut matches all the pieces of advice within the feature implementation, it does not matter whether they are `before`, `after`, or `around`. Thus, @Proceed and AroundClosure are easily reused, since the difference between one dynamic feature binding to another is only the aspect that the `adviceexecution` pointcut should apply (`within` clause in Listing 6 and 7) and the input to the driver. For example, if we want to apply the AroundClosure idiom to the *Cut* feature, we could reuse the code of this idiom used in *Total* feature. In Listing 7, we would alter `TotalFeature` to `CutFeature` in Line 2, which corresponds to the aspect that contains the *Cut* feature code and "`total`" to "`cut`" in Line 4, which represents the *Cut* feature property in the properties file used for the driver in our case.

**Changeability** is related to the amount of changes we need to perform in the application or in the idiom to implement flexible feature binding. Hence, we are interested in how difficult or time consuming the task of applying a flexible feature binding implementation through an idiom is.

*Layered Aspects and AnnotatedBind*. Applying these idioms demands several changes to implement flexible binding for a feature. For Layered Aspects, all pointcuts related to an `around` advice defined within the feature implementation are redefined in the aspect that implements dynamic feature binding. Hence, if the 101Companies SPL is being modified to support flexible binding, we need to change the aspect containing feature code (`TotalFeature`) to support pointcut redefinition and we would need to redefine each pointcut related to `around` advice in order to associate it with driver code. Similarly, our AnnotatedBind idiom demands these pointcut redefinitions and we need to introduce the annotations in the `around` advice, as explained in Section 3. This could require a lot of changes.

*@Proceed and AroundClosure*. Applying these idioms demands few changes to implement flexible binding for a feature. As explained in Section 3, the @Pro-

ceed and AroundClosure idioms do not redefine pointcuts. Hence, neither major changes nor altering feature code are needed.

**Instrumentation overhead (CIO)**. Now, we are interested in avoiding pointcuts that unnecessarily match join points. If we can exclude all the unnecessary instrumentation, we may gain in performance due to the less instrumentation provided by the AspectJ compiler.

*Layered Aspects*. Implementing flexible feature binding with this idiom may lead to instrumentation overhead because its `adviceexecution` pointcut matches more join points than necessary. The code of this idiom instruments all the pieces of advice within the feature implementation. However, the pieces of `around` advice are handled by the redefined pointcuts. This may lead to an overhead in the runtime as well.

*AnnotatedBind*. Our AnnotatedBind idiom annotates the `around` advice in collusion with the `!@annotation(AroundAdvice)` in the `adviceexecution` to avoid instrumentation overhead. In this way, the `advicexecution` pointcut only matches `before` and `after` advice, which eliminates the unnecessary instrumentation caused by the use of Layered Aspects.

*@Proceed and AroundClosure*. This increment and AdviceClosure do not present instrumentation overhead because their `adviceexecution` pointcut matches all the pieces of advice within the feature implementation only once. Hence, there is no unnecessary instrumentation.

**Behavior**. It is important to try to guarantee that using our idioms does not change the behavior of the feature code execution. Thus, to bring evidence that the execution of one flexible feature binding implementation presents the same behavior using any of the four idioms, we use the SafeRefactor tool [Soares et al., 2010], which receives two source code as input. It generates and executes unit tests and reports whether there are behavioral differences between the execution of these two sources. In our context, we use the SafeRefactor to detect behavioral changes between flexible feature binding implemented with different idioms. Therefore, we analyze two versions of the same case study (i.e. 101Companies), although each one using a distinct idiom to provide flexible binding for their features.

In this context, the SafeRefactor reports that our implementations do not present behavioral changes. That is, there is no difference in the execution of the generated tests when comparing the same feature with two different idioms. In Table 3, we illustrate details of the SafeRefactor report for the 101Companies and Freemind, which are sufficient for our discussion. However, we provide the full results in our online appendix [Andrade et al., 2013b]. For example, we analyze the 101Companies with Layered Aspects against the 101Companies with our AnnotatedBind idiom. The tool generates and executes 141 tests, reporting no behavioral changes between the code execution of *Total* feature using

Layered Aspects or our AnnotatedBind idiom. It is important to note that the SafeRefactor tool exercises more methods and execution paths when more tests are generated. Moreover, the number of generated tests may vary as shown in Table 3. This variation occurs due to the number of methods and advice of the source code and the number of methods and advice in common between the two source code. In some cases, we need to change a method's modifier from `private` or `protected` to `public`, specially for our @Proceed idiom, which does not support privileged aspects due to the @AspectJ syntax limitations, as explained in Section 3.2.

We already expected such results since our idioms are designed to change the least feature code possible. As a matter of fact, the differences between the idioms are focused on the way to implement static and dynamic feature binding, which leads to few changes in the feature code itself. Unfortunately, we could not compare our flexible binding implementations with the original code because SafeRefactor identifies common methods between Source and Target projects to generate the tests, and these common methods must have the same modifier, parameters, return, name, and be defined in the same class. Therefore, after refactoring the feature code into aspects, SafeRefactor would not identify these common methods.

**Table 3:** SafeRefactor results

| Source | Target | Number of tests | Changes? |
|---|---|---|---|
| 101Companies-LayAspects | 101Companies-AnnotatedBind | 141 | No |
| 101Companies-LayAspects | 101Companies-@Proceed | 180 | No |
| 101Companies-LayAspects | 101Companies-AroundClosure | 119 | No |
| 101Companies-AnnotatedBind | 101Companies-@Proceed | 135 | No |
| 101Companies-AnnotatedBind | 101Companies-AroundClosure | 145 | No |
| 101Companies-@Proceed | 101Companies-AroundClosure | 157 | No |
| Freemind-LayAspects | Freemind-AnnotatedBind | 241 | No |
| Freemind-LayAspects | Freemind-@Proceed | 229 | No |
| Freemind-LayAspects | Freemind-AroundClosure | 247 | No |
| Freemind-AnnotatedBind | Freemind-@Proceed | 275 | No |
| Freemind-AnnotatedBind | Freemind-AroundClosure | 275 | No |
| Freemind-@Proceed | Freemind-AroundClosure | 215 | No |

**Feature interaction.** Now, we illustrate how we apply our idioms in case features interact. A feature interaction occurs when one or more features modify or influence other features [Calder et al., 2003, Liu et al., 2005]. This interaction could be structural [Liu et al., 2005] or behavioral [Calder et al., 2003]. We could implement our idioms and Layered Aspects in such a scenario without additional effort. To illustrate a feature interaction scenario we consider a new case study called Sudoku in this work.

For instance, the features *Solver* and *Guesser* of the Sudoku case study interact structurally and behaviorally. The former uses the latter to guess possible solutions in a Sudoku board. Moreover, part of the *Guesser* code is defined within the *Solver* code. The *Solver* feature is responsible for providing a solution for a certain board of the Sudoku game whereas the *Guesser* feature implements an algorithm to guess possible solutions for Sudoku. In Listing 9, we show part of the *Solver* feature implementation. The `solve` method (Lines 2-10) is defined as an intertype because this whole method was extracted from the `BoardManager` class, as it concerns only to the *Solver* feature. This method also contains *Guesser* code. In order to provide a join point inside a method that can be extended by the aspect where the *Guesser* feature is implemented, we create a hook method (Lines 12-16). Hook methods are empty methods placed in the code for later extension [Kästner et al., 2007].

**Listing 9:** Solver feature

```
1  aspect SolverFeature {
2   List BoardManager.solve(Board board) {
3    List solutions = new LinkedList();
4    if (!board.isSolved()) {
5     hookguesser(board, solutions);
6    } else {
7     solutions.add(board);
8    }
9    return solutions;
10  }
11
12  void BoardManager.hookguesser(Board board, List solutions) {
13   Guesser guesser = new Guesser();
14   List guessed = guesser.guess(board);
15   solutions.addAll(solve(((Board) guessed.get(i))));
16  }
17 }
```

**Listing 10:** Guesser feature

```
1  aspect GuesserFeature {
2   pointcut hookguesser(BoardManager cthis, Board board,
3    List solutions) : execution(* BoardManager.hookguesser(..))
4    && this(cthis) && args(...);
5
6   before(BoardManager cthis, Board board, List solutions)
7   : hookguesser(cthis, board, solutions) {
8    Guesser guesser = new Guesser();
9    List guessed = guesser.guess(board);
10   solutions.addAll(solve(((Board) guessed.get(i))));
11   }
12 }
```

In this context, we also use pointcuts and advice to implement the *Guesser* feature within aspects, although we extracted the *Guesser* code from the aspects that implement the *Solver* feature instead of the invariable code. Listing 10 illustrates the *Guesser* feature implementation in an aspect. Lines 2-4 define a pointcut that matches the `hookguesser` method. We remove the *Guesser* feature code in Lines 13-15 of Listing 9 and implement it in an advice that corresponds to the *Guesser* feature implementation (Lines 8-10 of Listing 10).

At last, we observe that it is possible to apply our idioms to feature interaction scenario similar to the one presented. Furthermore, the quantitative evaluation results for *Solver* and *Guesser* comply with the results considering the other features (Section 4.2). Thus, this feature interaction scenario does not decrease the quality or our idioms regarding code cloning, scattering, tangling, or size.

### 4.4 Threats to validity

In this section, we discuss some threats to the validity of our work. We divided it in threats to internal and external validity.

**Threats to internal validity** are concerned with the fact that the assessment leads to the results [Wohlin et al., 2000].

*BerkeleyDB refactoring.* Our BerkeleyDB case study was originally refactored by Kästner et al. [Kästner et al., 2007]. The code of its features was extracted into aspects. However, this extraction was not in accordance with the way we extracted the implementation of features of the other case studies. Therefore, we refactored the code of BerkeleyDB product line's features so as to comply with the other feature implementations. Indeed, we followed the same procedures in order to refactor these implementations, such as the prune dependency rule.

*Feature code identification.* We cannot assure that the extraction of our selected features does not present bias because the task of identifying feature code is in a certain way subjective. This could be a hindrance to researchers that might try to replicate our work. Indeed, there could be unconformities between feature code identified by different researchers [Lai and Murphy, 1999].

However, we tried to minimize this threat in two ways. First, we used the prune dependency rules [Eaddy et al., 2007] to identify feature code. These rules define some procedures that the researcher should follow to avoid introducing bias in the resulting extracted feature code, as we mentioned in Section 4. Second, only one researcher identified the implementation of the selected features. We believe that restricting the number of people decreased unreliability.

**Threats to external validity** regard the generalization of the results [Wohlin et al., 2000].

*Selected software product lines limitations.* To perform our assessment, we selected five applications. The first and third authors were the ones who extracted the code of the 16 features. Thus, we defined five new SPLs based on these five applications. Although these SPLs are used only for academic purposes, their code covers different characteristics such as distinct complexities, architectures, granularities, purposes, and sizes.

Furthermore, the SPLs are written in Java and the feature code is implemented using AOP. Therefore, we cannot generalize the results presented here for other contexts, such as different programming paradigms or languages. Nevertheless, the combination of Java and AspectJ can be used in SPLs, which reinforce the significance of our idioms. So the increments presented could be applied to other SPLs that comply with the technologies we considered.

*Feature interaction scenario limitations.* Another threat concerns the feature interaction scenario, which is presented only in the new Sudoku case study. We cannot assure that our idioms work the same way to all feature interaction scenarios. However, we believe that extracting parts of feature code by means of aspects would not cause problems in other feature interaction examples. Although, we should need to refactor the feature code before extracting it to aspects. For instance, there is no way to extract a single parameter from a method by using aspects. Thus, such case demands refactoring before the extraction. We plan to find and investigate these scenarios in future work.

*Multiple drivers absence.* In this work, we only consider applying one driver at a time. However, we realize that some applications may depend on several conditions to activate or deactivate a certain feature. For instance, Lee et al. utilize a home service robot product line as case study [Lee and Kang, 2006]. This robot dynamically changes its configuration depending on the environment brightness or its remaining battery. It would demand at least two drivers to (de)activate some of its features in our context. Furthermore, the driver related boolean expression could become complex and hard to maintain, since simple boolean operations such as AND or OR may not work. Therefore, we reinforce that the mechanism that provides information to the driver is out of the scope of this work. Our proposal is to abstract the way our idioms receive this information. However, even the evaluation of a complex boolean expression could be only true or false, and this is what our idioms need to know. Nevertheless, we plan to study these scenarios in future work.

*AspectJ compiler dependence.* As explained in Section 3, our AroundClosure idiom depends on an internal resource of AspectJ's compiler. Thereby, this idiom may not work when applied in scenarios where a different compiler is used. However, besides AspectJ compiler, which is popular, other well-known compilers, such as the ones used for CaesarJ [Aracic et al., 2006] and ABC [Avgustinov et al., 2005] also include the resource needed by AroundClosure idiom. Thus, we believe our idiom covers at least three popular compilers.

## 5 Related work

Besides Layered Aspects [Andrade et al., 2011], which is an idiom we developed to fix some problems in existing solutions for flexible binding, we point out other researches regarding flexible binding as well as studies that relate aspects and product line features.

Rosenmüller et al. propose an approach for statically generating tailor-made SPLs to support dynamic feature binding [Rosenmüller et al., 2011b]. Similarly to part of our work, they statically choose a set of features to compose a product that supports dynamic binding. Furthermore, the authors describe a feature-based approach of adaptation and self-configuration to ensure composition safety. In this way, they statically select the features required for dynamic binding and generate a set of binding units that are composed at runtime to yield the program. Additionally, they implement their approach in one case study and evaluate it with concern to reconfiguration performance at runtime. Their contribution is restricted to applications based on C++, since they use the FeatureC++ language extension [Apel et al., 2005]. In contrast, our contribution is restricted to applications written mostly in Java, since we use AspectJ to provide flexible feature binding. Thus, our contribution applies to a different set of applications.

Lee et al. propose a systematic approach to develop dynamically reconfigurable core assets, which lies in the management of dynamic binding time regarding changes during the product execution [Lee and Kang, 2006]. Furthermore, they present strategies to manage product composition at runtime. Thus, they are able to safely change product composition (activate or deactivate features) due to an event occurred during runtime. However, the authors only provide conceptual support for a reconfiguration tool with no actual implementation.

Trinidad et al. propose a process to generate a component architecture that is able to dynamically activate or deactivate features and to perform some analysis operations on feature models to ensure that the feature composition is valid [Trinidad et al., 2007]. They apply their approach to generate an industrial real-time television SPL. However, they do not consider crosscutting features, which is very common based on our experience. In contrast, our approach works with crosscutting features.

Dinkelaker et al. [Dinkelaker et al., 2010] propose an approach that uses a dynamic feature model to describe variability and a domain-specific language for declaratively implementing variations and their constraints. This work has mechanisms to dynamically detect and resolve feature interactions at runtime.

Marot et al. [Marot and Wuyts, 2010] propose OARTA, which is a declarative extension to the AspectBench Compiler [Avgustinov et al., 2005], which allows dynamic weaving of aspects. OARTA extends the AspectJ language syntax so that a developer can name an advice, which allows referring to it later on. It is possible that aspects weave on other aspects. Therefore, they exemplify how to dynamically deactivate features in runtime situations (e.g. features competing for resources, which may be deactivated to speed up the execution). By using AspectJ, we would have to add an `if()` pointcut predicate to the pointcut of the advice that contains feature code. This may lead to a high degree of driver code scattering. Thus, as shown in Section 4, our AroundClosure idiom does not present such an issue.

An alternative proposal considers *conditional compilation* as a technique to implement flexible feature binding [Dolstra et al., 2003]. This work discusses how to apply *conditional compilation* in real applications like operating systems. Similarly to what we describe in our work, developers need to decide what features should be included to compose the product and their respective binding. However, the work concludes that, in fact, conditional compilation is not a very elegant solution to provide flexible feature binding. Hence, for complex variation points, the situation becomes even worse.

Chakravarthy et al. present Edicts [Chakravarthy et al., 2008], which is an AspectJ-based idiom to implement flexible feature binding. The idea is to scatter feature code across one abstract aspect and two concrete subaspects. Then, the programmer implements the driver by adding `if` statements within the pieces of advice. However, our previous work [Andrade et al., 2011] identified issues regarding code cloning, scattering, tangling, and size when applying Edicts to provide flexible feature binding. In this way, we reduce these issues with Layered Aspects and moreover, we fix the Layered Aspects limitations with the AroundClosure idiom proposed in this work.

## 6 Conclusion

In this work, we identify deficiencies in an existing AspectJ-based idiom to implement flexible feature binding in the context of software product lines. To improve this idiom, we incrementally define an idiom called AroundClosure. The creation of AroundClosure is performed increment-by-increment, which means that every increment corresponds to an improved idiom. To evaluate our idioms, we perform a quantitative assessment regarding code cloning, scattering, tangling, and size. Furthermore, we qualitatively discuss these idioms with respect to code reusability, changeability, instrumentation overhead, and behavior. Our evaluation results show that AroundClosure idiom brings advantages with respect to both quantitative and qualitative assessments. To achieve our conclusions, we base our analysis in 16 features of five different product lines and in our knowledge acquired during our research and previous work.

## Acknowledgments

We would like to thank colleagues of the Software Productivity Group (SPG) for helping to improve this work. Specially Gustavo Soares who helped us with SafeRefactor. Besides that, we would like to acknowledge financial support from

CNPq, FACEPE, CAPES, and the Brazilian Software Engineering National Institute of Science and Technology (INES).

## References

[Andrade et al., 2011] Andrade, R. et al. (2011). Assessing idioms for implementing features with flexible binding times. In *Proc. of the European Conf. on Software Maintenance and Reengineering*, pages 231–240.

[Andrade et al., 2013a] Andrade, R. et al. (2013a). AspectJ-based idioms for flexible feature binding. In *Proc. of the Brazilian Symposium on Software Components, Architectures, and Reuse*, pages 59–68.

[Andrade et al., 2013b] Andrade, R. et al. (2013b). Online appendix, http://tinyurl.com/mkcaf5k.

[Apel et al., 2005] Apel, S. et al. (2005). FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proc. of the Int. Conf. on Generative Programming and Component Engineering*, pages 125–140.

[Aracic et al., 2006] Aracic, I. et al. (2006). An overview of caesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173.

[Avgustinov et al., 2005] Avgustinov, P. et al. (2005). ABC: An extensible AspectJ compiler. In *Proc. of the Int. Conf. on Aspect-Oriented Software Development*, pages 87–98.

[Basili et al., 1994] Basili, V., Caldiera, G., and Rombach, D. H. (1994). The goal question metric approach. In *Encyc. of Software Engineering*, pages 528–532.

[Baxter et al., 1998] Baxter, I. et al. (1998). Clone detection using abstract syntax trees. In *Proc. of the Int. Conf. on Software Maintenance*, pages 368–377.

[Bruntink et al., 2005] Bruntink, M., van Deursen, A., van Engelen, R., and Tourwe, T. (2005). On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31:804–818.

[Calder et al., 2003] Calder, M. et al. (2003). Feature interaction: a critical review and considered forecast. *Computer Networks: The Int. Journal of Computer and Telecommunications Networking*, 41(1):115–141.

[Chakravarthy et al., 2008] Chakravarthy, V., Regehr, J., and Eide, E. (2008). Edicts: Implementing features with flexible binding times. In *Proc. of the Int. Conf. on Aspect-Oriented Software Development*, pages 108–119.

[Dinkelaker et al., 2010] Dinkelaker, T., Mitschke, R., Fetzer, K., and Mezini, M. (2010). A dynamic software product line approach using aspect models at runtime. In *Proc. of the Workshop on Composition and Variability*.

[Dolstra et al., 2003] Dolstra, E. et al. (2003). Timeline variability: The variability of binding time of variation points. Technical Report UU-CS-2003-052.

[Eaddy, 2008] Eaddy, M. (2008). *An Empirical Assessment of the Crosscutting Concern Problem*. PhD thesis, Columbia University, New York.

[Eaddy et al., 2007] Eaddy, M., Aho, A., and Murphy, G. C. (2007). Identifying, assigning, and quantifying crosscutting concerns. In *Proc. of the Int. Workshop on Assessment of Contemporary Modularization Techniques*, pages 2–7.

[Favre et al., 2012] Favre, J.-M. et al. (2012). 101companies: A community project on software technologies and software languages. 7304:58–74.

[Hilsdale and Hugunin, 2004] Hilsdale, E. and Hugunin, J. (2004). Advice weaving in aspectJ. In *Proc. of the Int. Conf. on Aspect-oriented software development*, pages 26–35.

[Kamiya et al., 2002] Kamiya, T. et al. (2002). Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transaction on Software Engineering*, 28(7):654–670.

[Kamiya et al., 2013] Kamiya, T. et al. (2013). CCFinder tool, http://www.ccfinder.net/.

[Kang et al., 1990] Kang, K.-C. et al. (1990). Feature-oriented domain analysis (FODA). Technical Report CMU/SEI-90-TR-21.

[Kapser and Godfrey, 2006] Kapser, C. J. and Godfrey, M. W. (2006). Supporting the analysis of clones in software systems: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:61–82.

[Kästner, 2013] Kästner, C. (2013). Sudoku, http://tinyurl.com/oelty38.

[Kästner et al., 2007] Kästner, C. et al. (2007). A case study implementing features using AspectJ. In *Proc. of the Int. Software Product Line Conf.*, pages 223–232.

[Kiczales et al., 1997] Kiczales, G. et al. (1997). Aspect–oriented programming. In *proc. of European Conf. on Object–Oriented Programming*, pages 220–242.

[Kiczales et al., 2001] Kiczales, G. et al. (2001). Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65.

[Laddad, 2009] Laddad, R. (2009). *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications.

[Lai and Murphy, 1999] Lai, A. and Murphy, G. C. (1999). The structure of features in Java code: an exploratory investigation. In *Workshop on Multidimensional separation of concerns*.

[Lee and Kang, 2006] Lee, J. and Kang, K. C. (2006). A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Proc. of the Int. Software Product Line Conf.*, pages 131–140.

[Liu et al., 2005] Liu, J. et al. (2005). Modeling interactions in feature oriented software designs. In *Feature Interactions in Telecommunications and Software Systems*, pages 178–197.

[Marot and Wuyts, 2010] Marot, A. and Wuyts, R. (2010). Composing aspects with aspects. In *Proc. of the Int. Conf. on Aspect-Oriented Software Development*, pages 157–168.

[Müller et al., 2013] Müller, J. et al. (2013). Freemind, http://tinyurl.com/5qrd5.

[Parnas, 1972] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

[Pohl et al., 2005] Pohl, K., Bockle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering*. Springer-Verlag, Berlin.

[Ribeiro et al., 2009] Ribeiro, M. et al. (2009). Does AspectJ provide modularity when implementing features with flexible binding times? In *Latin American Workshop on Aspect-Oriented Software Development*, pages 1–6.

[Rosenmüller et al., 2011a] Rosenmüller, M. et al. (2011a). Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197.

[Rosenmüller et al., 2011b] Rosenmüller, M. et al. (2011b). Tailoring dynamic software product lines. In *Proc. of the Int. Conf. on Generative programming and component engineering*, pages 3–12.

[School and Rajapakse, 2005] School, D. R. and Rajapakse, D. C. (2005). An investigation of cloning in web applications. In *Proc. of the Special Interest Tracks and Posters of the Int. Conf. on World Wide Web*, pages 252–262.

[Soares et al., 2010] Soares, G., Gheyi, R., Serey, D., and Massoni, T. (2010). Making program refactoring safer. *IEEE Software*, 27:52–57.

[Tigris, 2013] Tigris (2013). ArgoUML, http://argouml.tigris.org/.

[Trinidad et al., 2007] Trinidad, P. et al. (2007). Mapping feature models onto component models to build dynamic software product lines. In *Proc. of the Int. Software Product Line Conf.*, pages 51–56.

[Wohlin et al., 2000] Wohlin, C. et al. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic, Boston.