AspectJ-based Idioms for Flexible Feature Binding

Rodrigo Andrade*, Henrique Rebêlo*, Márcio Ribeiro†, Paulo Borba*
* Informatics Center, Federal University of Pernambuco
Email: rcaa2,hemr,phmb@cin.ufpe.br
† Computing Institute, Federal University of Alagoas
Email: marcio@ic.ufal.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 base 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 new AspectJ-based idioms. Moreover, to evaluate our new 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 new idioms in terms of code reusability, changeability, and instrumentation overhead.**

Keywords: Software Product Lines; Aspect-Oriented Programming; Idioms; Flexible Feature Binding.

I. 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 [1].

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 [2]. 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 run or link time) to allow more flexibility, with the cost of performance and memory consumption. Furthermore, if the 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 [3], which we focus on this work, we proposed Layered Aspects [4]. This solution reduces several problems identified in a previous work [5], such as code cloning, scattering, and tangling [4], [6]. Although these goals are achieved to some extent, Layered Aspects still presents some deficiencies. It may introduce feature code scattering and instrumentation overhead to the flexible feature binding implementation. Additionally, applying Layered Aspects demands several changes, which could hamper the reuse of the flexible feature binding implementation.

Hence, to address the Layered Aspects issues and still have low rates of code cloning, scattering, and tangling, we define three new idioms based on AspectJ [7], 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 increment addresses part of the issues with the aid of Java annotations. The second increment uses the @AspectJ syntax [8] to address more Layered Aspects issues, although, because this syntax does not support intertypes, it may introduce problems, such as feature code scattering. In this context, due to AspectJ traditional syntax limitations, our final idiom uses an internal resource of AspectJ's compiler to address all the issues without introducing @AspectJ syntax problems.

To evaluate these idioms, we extract the code of a total of 13 features from four different product lines and we apply each new idiom plus Layered Aspects to implement flexible binding for these features. Then, to evaluate whether our new 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 five metrics: Pairs of Cloned Code, Degree of Scattering across Components [9], Degree of Tangling within Components [9], Source Lines of Code, and Vocabulary Size. Additionally, we discuss the four idioms regarding three factors: their code reusability, changeability, and instrumentation overhead based on our four product lines and also on our previous knowledge about this topic [4], [6]. As result of this evaluation, we conclude that our final new idiom incrementally addresses these three factors and does not present worse results regarding the software metrics.

In summary, the contributions of this paper are:

- We identify deficiencies in an existent idiom to implement flexible feature binding;
- We address these deficiencies by incrementally defining three new idioms to implement flexible feature binding;
- We apply these four idioms to provide flexible binding for 13 features from four case studies;
- We quantitatively evaluate the three new idioms plus the existent one with respect to code cloning, scattering, tangling, and size by means of software metrics;
- We discuss the idioms regarding reusability, changeability, and code instrumentation overhead;
- We provide the source code and sheet used in this paper [10].

At last, we structure the remainder of this paper as follows. In Section II, we present the motivation of our work, detailing the Layered Aspects issues. Section III introduces our three new idioms to address these issues. In Section IV, we present

the evaluation of Layered Aspects and our three new idioms regarding code cloning, scattering, tangling, and size as well as a qualitative discussion. Finally, Section V presents the threats to validity, Section VI discusses related work, and Section VII concludes this work.

II. Motivating Example

This section presents the Layered Aspects issues by showing the implementation of flexible binding for the Total optional feature of the 101Companies SPL. This product line is based on a Java version of the 101Companies project [11]. In this context, the Total feature represents the total salary of a given employee, the sum of all department salaries, or the sum of all company salaries. We do not use the same application from our previous work [4] as a toy example because we believe that the 101Companies project is best known nowadays, which makes the understanding easier. Besides that, the application we used before has not been supported for a long time[1].

As mentioned in the previous section, to implement flexible binding time, we could use the Layered Aspects idiom, 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.

Using the 101Companies Total feature as an example, in Listing 1, we illustrate part of this feature code. It consists of pointcuts (Line 3), advice (Line 6), intertype declarations (Line 11), and `private` methods, which we omit for simplicity. Albeit we do not show all the code in this paper, we provide it elsewhere [10]. To apply Layered Aspects, we need to change the `TotalFeature` aspect by including the `abstract` keyword in Line 1. This allows the concrete subaspect to inherit from `TotalFeature`, since only abstract aspects can be inherited in AspectJ [8].

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, as we illustrate in Listing 2. Thus, we are able to statically activate the feature execution by including both aspects in the project build.

Listing 2: TotalStatic subaspect

```
1 aspect TotalStatic extends TotalFeature  {}
```

Before explaining the dynamic feature activation or deactivation, we first need to introduce an important concept used throughout this paper: the driver [4]. 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 [6]. In our case, the driver mechanism reads a property value from a properties file. For instance, if we want 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 3. 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 3. 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 [4].

Listing 3: 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 [4]. For such scenario, we would miss the base 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 base 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 value is 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` [4], and consequently the latter needs to be an abstract aspect, since AspectJ does not provide a way to inherit from a concrete aspect.

[1] http://kiang.org/jordan/software/tetrismidlet/

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 is necessary to allow 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 15 empty concrete aspects to implement static binding for our 13 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. Besides that, this could hinder code reusability and changeability. It may hinder the former because these pointcut redefinitions may hamper reusing the flexible binding implementation. Additionally, it hinders the latter when introducing a new driver for example, as we would need to associate the driver to each pointcut.

Hence, we enumerate the main goals we try to address in this work:
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 new idioms that 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.

III. New idioms

In this section, we illustrate our three new idioms. 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 II. The last increment corresponds to the AroundClosure in which we address all the issues.

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.

A. First increment

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 [8].

Listing 4: Around advice annotated

```
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 }
```

In this context, we create an `AroundAdvice` annotation and we use it to annotate all pieces of `around` advice within the feature code, as showed in Line 3 of Listing 4. Hence, we can prevent `adviceexecution` pointcut to match any of these annotated advice when applying dynamic binding.

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 5, we show the `adviceexecution` pointcut with the `!@annotation(AroundAdvice)` clause, which is the part that differs from Listing 3.

Therefore, we resolve the first Layered Aspects issue. However, the other two issues remain open. To address them, we continue demonstrating more increments next.

Listing 5: TotalDynamic aspect with the first increment

```
1 aspect TotalDynamic extends TotalFeature {
2  ...
3  void around() : adviceexecution() && within(TotalFeature)
4   && !@annotation(AroundAdvice) {
5   if (Driver.isActivated("total")) { proceed(); }
6  }
7 }
```

B. Second increment

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 II.

To achieve that, we use the new @AspectJ syntax [8], 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`, `@Around`, `@Before`, and `@After`. Listing 6 shows 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.

However, the @AspectJ syntax presents some disadvantages. First, there is no way to declare a `privileged` aspect [8], which is necessary to avoid creating an access method or changing base 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 [8]. Therefore, we need to define an additional aspect, which is implemented with the traditional AspectJ syntax, containing the needed intertype declarations.

Listing 6: Total feature with the second increment

```
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 }
```

Despite these limitations, we could eliminate the empty concrete aspect to implement the static feature binding. Since `TotalFeature` of Listing 6 is a class rather than an `abstract` aspect, we are able to instantiate it without the concrete subaspect. In this way, to statically activate 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 7 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 3. 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 [4].

Listing 7: TotalDynamic class with second increment

```
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 second increment, it still presents some undesired points. First, the @AspectJ syntax is limited: it does not support `privileged` aspects, intertype declarations, and exception handling [8]. 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.

C. Final increment: AroundClosure

Now, we improve our previous increment by addressing all the three Layered Aspects issues presented in Section II, 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 8, respectively.

In this context, since `TotalFeature` is not an abstract aspect like in Layered Aspects or our first increment, 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 8. 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.

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 normally. 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.

Listing 8: 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 }
```

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 base 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 base code flow of execution [12], 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.

Listing 9: 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 }
```

To avoid missing the base code flow of execution when the feature is dynamically deactivated, Listing 9 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 base 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 base 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. Albeit our focus is only AspectJ, other AOP-based compilers also include this object [13], [14].

At last, by means of our new AroundClosure idiom, we address the Layered Aspects issues without introducing the @AspectJ syntax problems. Therefore, we recommend developers to use AroundClosure instead of LayeredAspects, although the other increments could be used as idioms as well. Furthermore, AroundClosure brings some advantages with respect to code reusability, changeability, and byte code instrumentation. Besides that, it does not worsen the metrics results with respect to code cloning, scattering, tangling, and size. We discuss these matters in Section IV.

IV. Evaluation

In this section, we explain our evaluation. In Section IV-A, we quantitatively evaluate our new idioms and Layered Aspects in a similar way we did in our previous work [4] to avoid bias. Besides that, we discuss our three new idioms and Layered Aspects regarding code reusability, changeability, and instrumentation overhead in Section IV-B.

In this context, we consider 13 features of four case studies: two features of 101Companies [11], eight features of BerkeleyDB [15], one feature of ArgoUML [16], and two features of Freemind [17]. Besides 101Companies, the other three case studies are the same from our previous work [4]. This is important to show the gains obtained with the new 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 I, we map the 13 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 regarding feature model, such as optional, alternative, and mandatory features [18].

To perform our evaluation, we follow four main procedures. However, we only execute the third and fourth procedures for the BerkeleyDB case study, as we discuss in Section V-A.

TABLE I: 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

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 [19], 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 choose this rule to reduce introducing bias while identifying feature code.

Second, we extracted part of this code into AspectJ aspects. We say part because some feature code do not need to be extracted into aspects, as it could be localized in classes. The code within these classes is not extracted into aspects because the whole class is only relevant to the feature, so it should not exist in the base code by following the prune dependency rules. 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.

Third, to evaluate our three new idioms and Layered Aspects, we applied each one of our three new idioms plus Layered Aspects to implement flexible binding for the 13 features of the four case studies.

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

For BerkeleyDB, we apply the four idioms to implement flexible binding for eight features of the BerkeleyDB product line [15]. This product line has around 32000 lines of code whereas the eight features sum up approximately 2300 lines of code. This allows us to test our new AroundClosure idiom and the increments in a large and wide 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 code.

For Freemind, we also extract the code of two features into AspectJ aspects. Then, we apply the four idioms to provide flexible feature binding for these two features. The Freemind product line has about 67000 lines of code and both features have approximately 4000 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[2] to obtain the LOC and we use sheets to auxiliary the computation of the metrics. Moreover, we detail the selected metrics and results in Section IV-A. At last, we provide all the source code and sheets elsewhere [10].

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

A. Quantitative analysis

To drive the quantitative evaluation of our idioms, we follow the Goal-Question-Metric (GQM) design [20]. We structure it in Table II. We use Pairs of Cloned Code in Section IV-A1 to answer Question 1, as it may indicate a design that could increase maintenance costs [21] because a change would have to be done twice to the duplicated code. To answer Question 2, we use Degree of Scattering across Components [9] in Section IV-A2 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 [9] metric in Section IV-A3. Furthermore, Source Lines of Code and Vocabulary Size are well known metrics for quantifying a module size and complexity. So, in Section IV-A4, 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 [10].

TABLE II: 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 new idioms increase code cloning?	
`Pairs of Cloned Code`	`PCC`
Q2- Do the new idioms increase driver and feature code scattering?	
`Degree of Scattering across Components`	`DOSC`
Q3- Do the new idioms increase tangling between driver and feature code?	
`Degree of Tangling within Components`	`DOTC`
Q4- Do the new idioms increase lines of code and number of components?	
`Source Lines of Code`	`SLOC`
`Vocabulary Size`	`VS`

1) Cloning: To answer Question 1 and try to determine that our new idioms do not increase code cloning, we use the CCFinder [22] tool to obtain the PCC metric results. CCFinder is a widely used tool to detect cloned code [23], [24], [25]. Equally to our previous work [4], we use 12 as the token set size (TKS), and use 40 as the minimum clone length (in tokens), 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 eight features out of 13 regarding the four idioms. Additionally, the idioms lead to low PCC rates for the code of these five features that present code replication [10]. Therefore, our new idioms do not increase code cloning. This is the answer for Question 1.

2) Scattering: To answer Question 2, we use DOSC to analyze feature and driver code scattering for each idiom. Feature and driver are different concerns, so we analyze them separately.

Driver. In Figure 1, we present the results regarding the DOSC metric. The only idiom that presents driver scattering is our new first increment. This occurs due to the annotations we must add to `around` advice defined within the feature code, as explained in Section III-A. This may hinder code reusability and changeability. However, our first increment reduces the byte code instrumentation, as we discuss in Section IV-B. Additionally, feature *Cut* does not present an `around` advice, therefore there is no `AroundAdvice` annotation in its code. The *NIO* and *IO* features only present one `around` advice each, thus there is no need to add the `AroundAdvice` annotation, as only one pointcut redefinition implements the driver.

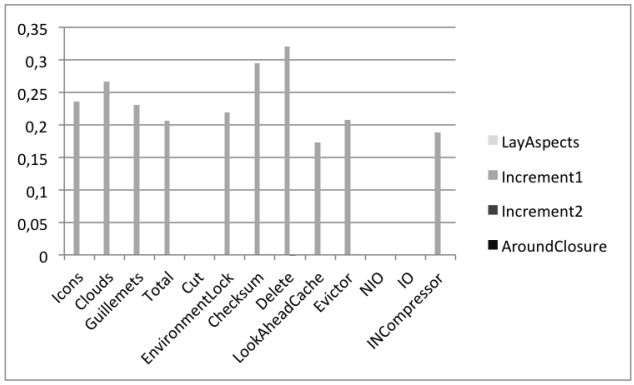

Fig. 1: DOSC for driver

Furthermore, Layered Aspects [4] and our new second increment do not present any driver code scattering, since their driver are implemented within a unique separate component for each idiom. Specially, our AroundClosure idiom, which is the best solution proposed, do not present driver code scattering either due to the same reasons.

Feature. Figure 2 illustrates the DOSC results. In this context, our new second increment presents a disadvantage comparing to the others. This happens because the @AspectJ syntax, which is used by the second increment, does not support intertype declarations. Thus, as explained in Section III-B, 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, features *NIO* and *IO* do not present intertype declarations within their implementation. Thus, our second increment does not scatter feature code in these cases.

Differently from driver code scattering, the implementation of these two idioms are similar regarding feature code. Thus, our first increment and Layered Aspects present similar results. 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.

At last, we answer Question 2 saying that our first increment increases driver code scattering whereas our second increment increases feature code scattering. However, our final solution (AroundClosure) does not present driver scattering and lower scattering rates for feature.

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 [26], one should be able to implement and reason about each concern independently.

Equally to our previous work [4], we also assume that the greater is the tangling between feature code and its driver code,

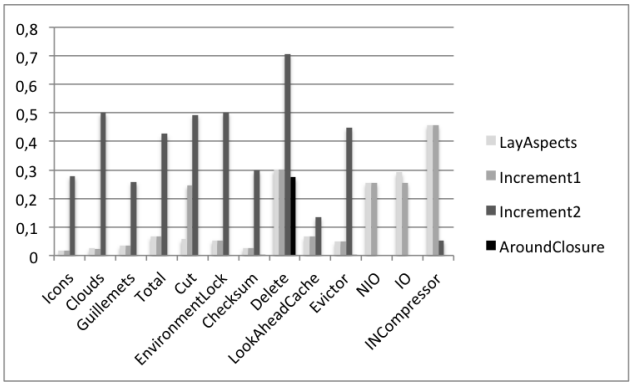

Fig. 2: DOSC for feature

the worse is the separation of those concerns. In this way, we measure the Degree of Tangling within Components (DOTC).

In Figure 3, we show the DOTC metric results. Only our first increment 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, our second increment and AroundClosure present no tangling between driver and feature code. For example, Listings 7 and 8 contain only driver code by following the prune dependency rule, that is, the code defined within `TotalDynamic` class and aspect is relevant only to driver concern. In this way, these idioms comply with the results obtained for Layered Aspects.

At last, we conclude that our first increment increases the tangling between driver and feature code. However, our second increment and AroundClosure does not present tangling at all. This answers Question 3.

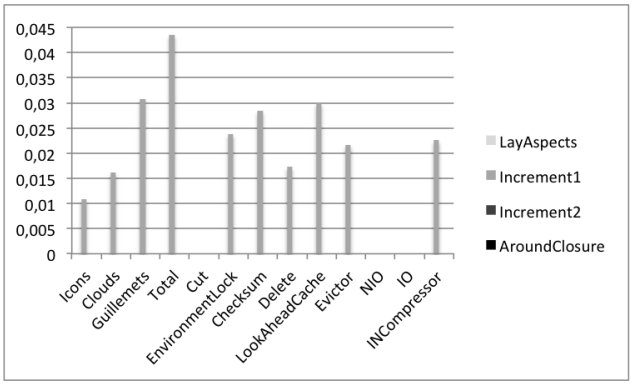

Fig. 3: DOTC

4) Size: To identify the idiom that increases the size of its implementation, we try to answer Question 4, which is related to the size of each idiom in terms of lines of code and the number of components. 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 new idioms do not increase lines of code and number of components. We provide all data and graphs elsewhere [10].

B. Qualitative discussion

In this section, we qualitatively discuss Layered Aspects and our three new idioms in terms of code reusability, changeability, and instrumentation overhead. Several works choose similar approaches to qualitatively discuss their implementations [27], [28], [29], [30].

1) Reusability: The reusability concerns 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 First increment. 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 first increment redefine the pointcuts related to `around` advice, which hinders reuse since these redefined pointcuts are associated to a particular feature. Hence, this compromises the overall reusability of the implementation of both idioms.

Second increment 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, our second increment 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 7 and 8) 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 8, 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.

2) Changeability: 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 First increment. 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 first increment also demands these pointcut redefinitions and we need to introduce the annotations in the `around` advice, as explained in Section III. This could require a lot of changes.

Second increment and AroundClosure. Applying these idioms demands few changes to implement flexible binding for

a feature. As explained in Section III, the second increment and AroundClosure do not redefine pointcuts. Hence, neither major changes nor altering feature code are needed.

3) 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 runtime 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.

First increment. Our first increment 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.

Second increment 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.

At last, we also provide all the source code used in this work and useful information for researchers to replicate our work [10].

V. 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.

A. Threats to internal validity

Threats to internal validity are concerned with the fact that the assessment leads to the results [31]. Therefore, in our work, these threats encompass the introduction of bias due to the selection of certain procedures, such as the way of feature code is assigned. Additionally, we discuss decisions that might introduce errors in our work and how we tried to circumvent them.

BerkeleyDB refactoring. Our BerkeleyDB case study was originally refactored by Kästner et al. [15]. 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 [32].

However, we tried to minimize such a unreliability in two ways. First, we used the prune dependency rules [19] 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 IV. Second, only one researcher identified the implementation of the selected features. We believe that restricting the number of people decreased unreliability.

B. Threats to external validity

Threats to external validity are concerned with generalization of the results [31].

Selected software product lines limitations. To perform our assessment, we selected four SPLs. They are written in Java and the feature code is implemented using aspect-oriented programming. 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 together in SPLs, which reinforce the significance of our new idioms. So the increments presented could be applied to other SPLs that comply with the technologies we considered.

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 [33]. This robot changes its configuration dynamically 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. Even considering a complex boolean expression, its evaluation could be only true or false, and this is what our driver implementation needs to know. Nevertheless, we plan to study these scenarios in future work.

AspectJ compiler dependence. As explained in Section III, 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 [13] and ABC [14] also include the resource needed to apply AroundClosure idiom. Thus, we believe our idiom covers at least three popular compilers.

VI. Related work

Besides Layered Aspects, we point out other researches regarding flexible binding times as well as studies that relate aspects and product line features. Additionally, we discuss how our work differs from them.

Rosenmüller et al. propose an approach for statically generating tailor-made SPLs to support dynamic feature binding time [34]. Similarly to part of our work, they statically choose a set of features to compose a product that supports dynamic binding time. 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 [35]. In contrast, our contribution is restricted to applications written mostly in Java, since we use AspectJ to provide flexible feature binding. In this way, 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 [33]. 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 [36]. They apply their approach to generate an industrial real-time television SPL. However, they do not not consider crosscutting features, which is very common based on our experience. In contrast, our approach works with crosscutting features.

Dinkelaker et al. [37] propose an approach that uses a dynamic feature model to describe variability and uses a domain-specific language for declaratively implementing variations and their constraints. Their work has mechanisms to detect and resolve feature interactions dynamically by validating an aspect-oriented model at runtime.

Marot et al. [38] propose OARTA, which is a declarative extension to the AspectBench Compiler [14], 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 IV, our AroundClosure idiom does not present such issue.

Rosenmüller et al. present an approach that supports static and dynamic composition of features from a single base code [39]. They provide an infrastructure for dynamic instantiation and validation of SPLs. Their approach is based on FOP [40] whereas our work uses AOP. Additionally, they use an extension called FeatureC++ [35] to automate dynamic instantiation of SPLs. However, the usage of C++ as a client language may introduce some specific problems. Static constructs when using dynamic composition, virtual classes, semantic differences when comparing static and dynamic compositions are examples of such problems [39]. Albeit our work uses only Java as a client language, we did not observe these problems in our implementations.

An alternative proposal considers *conditional compilation* as a technique to implement flexible feature binding [41]. 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 times. 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.

Another proposal to implement flexible feature binding, which is also our previous work, considers aspect inheritance [6]. It defines an idiom that relies on aspect inheritance through the abstract pointcut definition. This solution states that we have to create an abstract aspect with feature code and an abstract pointcut definition, then we associate this driver with the advice. Furthermore, we create two concrete subaspects inheriting from the abstract one in order to implement the concrete driver. Despite of the fact that this solution enhances some weaknesses found in the literature, it is worst than the idioms presented in our previous work [4].

Chakravarthy et al. present Edicts [5], 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 [4] 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 issues with the AroundClosure idiom proposed in this work.

VII. 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 a new 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 new idioms, we perform an assessment regarding code cloning, scattering, tangling, and size. Furthermore, we qualitatively discuss these idioms with respect to code reusability, changeability, and instrumentation overhead. To achieve our conclusions, we base our analysis in 13 features of four different product lines and in our knowledge acquired during our research and previous work.

VIII. Acknowledgments

We would like to thank colleagues of the Software Productivity Group (SPG) for several discussions that helped to improve the work reported here. 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

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

[2] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake, "Flexible feature binding in software product lines," *Automated Software Engineering*, vol. 18, no. 2, pp. 163–197, 2011.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect–oriented programming," in *Proceedings of European Conference on Object–Oriented Programming*. Jyväskylä, Finland: Springer, Berlin, 9-13 June 1997, pp. 220–242.

[4] R. Andrade, M. Ribeiro, V. Gasiunas, L. Satabin, H. Rebelo, and P. Borba, "Assessing idioms for implementing features with flexible binding times," in *Proceedings of the European Conference on Software Maintenance and Reengineering*. Oldenburg, Germany: IEEE Computer Society, Washington, 1-4 March 2011, pp. 231–240.

[5] V. Chakravarthy, J. Regehr, and E. Eide, "Edicts: Implementing features with flexible binding times," in *Proceedings of the International Conference on Aspect-Oriented Software Development*. Brussels, Belgium: ACM, New York, 1-4 April 2008, pp. 108–119.

[6] M. Ribeiro, R. Cardoso, P. Borba, R. Bonifácio, and H. Rebêlo, "Does AspectJ provide modularity when implementing features with flexible binding times?" in *Latin American Workshop on Aspect-Oriented Software Development*, Fortaleza, Brazil, 04-05 October 2009, pp. 1–6.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting Started with AspectJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.

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

[9] M. Eaddy, "An empirical assessment of the crosscutting concern problem," Ph.D. dissertation, Graduate School of Arts and Sciences, Columbia University, New York, 2008.

[10] R. Andrade, M. Ribeiro, H. Rebêlo, V. Gasiunas, L. Satabin, and P. Borba, "Flexible binding time," 03 August 2013. [Online]. Available: http://preview.tinyurl.com/mkcaf5k

[11] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich, "101companies: A community project on software technologies and software languages," vol. 7304, pp. 58–74, 2012.

[12] E. Hilsdale and J. Hugunin, "Advice weaving in aspectj," in *Proceedings of the international conference on Aspect-oriented software development*. Lancaster, UK: ACM, New York, 22-26 March 2004, pp. 26–35.

[13] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An overview of caesarj," *Transactions on Aspect-Oriented Software Development I*, vol. 3880, pp. 135–173, 2006.

[14] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "ABC: An extensible AspectJ compiler," in *Proceedings of the International Conference on Aspect-Oriented Software Development*. Chicago, USA: ACM, New York, 14-18 March 2005, pp. 87–98.

[15] C. Kästner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *Proceedings of the 11th International Software Product Line Conference*. Kyoto, Japan: IEEE Computer Society, Washington, 10-14 September 2007, pp. 223–232.

[16] Tigris, "Argouml," 03 August 2013. [Online]. Available: http://argouml.tigris.org/

[17] J. Müller, D. Polansky, P. Novak, C. Foltin, and D. Polivaev, "Free mind mapping software," 03 August 2013. [Online]. Available: http://preview.tinyurl.com/5qrd5

[18] K.-C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA). feasibility study," Tech. Rep. CMU/SEI-90-TR-21, 1990.

[19] M. Eaddy, A. Aho, and G. C. Murphy, "Identifying, assigning, and quantifying crosscutting concerns," in *Proceedings of the International Workshop on Assessment of Contemporary Modularization Techniques*. Minneapolis, USA: IEEE Computer Society, Washington, 20-26 May 2007, pp. 2–7.

[20] V. Basili, G. Caldiera, and D. H. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*. Wiley, New Jersey, 1994, pp. 528–532.

[21] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*. Bethesda, USA: IEEE Computer Society, Washington, 16-20 November 1998, pp. 368–377.

[22] T. Kamiya, F. Ohata, K. Kondou, S. Kusumoto, and K. Inoue, "CCfinder Official Site," 03 August 2013. [Online]. Available: http://www.ccfinder.net/

[23] D. R. School and D. C. Rajapakse, "An investigation of cloning in web applications," in *Proceedings of the Special Interest Tracks and Posters of the International Conference on World Wide Web*. Sydney, Australia: Springer-Verlag, Berlin, 27-29 July 2005, pp. 252–262.

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

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

[26] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[27] H. Rebêlo, R. Lima, and G. T. Leavens, "Modular contracts with procedures, annotations, pointcuts and advice," in *Proceeding of the Brazilian Symposium on Programming Languages*, Sao Paulo, Brazil, 29-30 September 2011.

[28] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *Proceedings of the international conference on Software engineering*. St. Louis, USA: ACM, New York, 15-21 May 2005, pp. 49–58.

[29] C. A. Cunha, J. a. L. Sobral, and M. P. Monteiro, "Reusable aspect-oriented implementations of concurrency patterns and mechanisms," in *Proceedings of the international conference on Aspect-oriented software development*, 2006, pp. 134–145.

[30] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*. Seattle, USA: ACM, New York, 4-8 November 2002, pp. 161–173.

[31] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Boston: Kluwer Academic, 2000.

[32] A. Lai and G. C. Murphy, "The structure of features in Java code: an exploratory investigation," in *Workshop on Multidimensional separation of concerns*. Denver, USA: ACM, New York, 1-5 November 1999.

[33] J. Lee and K. C. Kang, "A feature-oriented approach to developing dynamically reconfigurable products in product line engineering," in *Proceedings of the International Software Product Line Conference*. Baltimore, USA: IEEE Computer Society, Washington, 21-24 August 2006, pp. 131–140.

[34] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel, "Tailoring dynamic software product lines," in *Proceedings of the international conference on Generative programming and component engineering*. Portland, USA: ACM, New York, 22-23 October 2011, pp. 3–12.

[35] S. Apel, T. Leich, M. Rosenmuller, and G. Saake, "FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming," in *Proceedings of the International Conference on Generative Programming and Component Engineering*. Tallinn, Estonia: Springer-Verlag, Berlin, 29-1 September 2005, pp. 125–140.

[36] P. Trinidad, A. R. Cortés, J. Peña, and D. Benavides, "Mapping feature models onto component models to build dynamic software product lines," in *Proceedings of the International Software Product Line Conference*. Kyoto, Japan: Kindai Kagaku, Tokyo, 10-14 September 2007, pp. 51–56.

[37] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini, "A dynamic software product line approach using aspect models at runtime," in *Proceedings of the Workshop on Composition and Variability*, Rennes and Saint Malo, France, 15-19 March 2010.

[38] A. Marot and R. Wuyts, "Composing aspects with aspects," in *Proceedings of the International Conference on Aspect-Oriented Software Development*. Rennes and Saint-Malo, France: ACM, New York, 15-19 March 2010, pp. 157–168.

[39] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel, "Code generation to support static and dynamic composition of software product lines," in *Proceedings of the International Conference on Generative Programming and Component Engineering*. Nashville, TN, USA: ACM, New York, 19-23 October 2008, pp. 3–12.

[40] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *Proceedings of the European Conference on Object-Oriented Programming*. Jyväskylä, Finland: Springer, Berlin, 9-13 June 1997, pp. 419–443.

[41] E. Dolstra, G. Florijn, and E. Visser, "Timeline variability: The variability of binding time of variation points," Tech. Rep. UU-CS-2003-052, 2003.