Advance Access publication on 31 July 2015
doi:10.1093/comjnl/bxv054

Assessing Idioms for a Flexible Feature Binding Time

Rodrigo Andrade[1]*, Márcio Ribeiro[2], Henrique Rebêlo[1], Paulo Borba[1], Vaidas Gasiunas[3] and Lucas Satabin[3]

[1]*Informatics Center, Federal University of Pernambuco, Recife, Brazil*
[2]*Computing Institute, Federal University of Alagoas, Maceió, Brazil*
[3]*Technische Universität Darmstadt, Darmstadt, Germany*
**Corresponding author: rcaa2@cin.ufpe.br*

In software product lines development, it is sometimes important to provide a flexible binding time for features such that developers can choose between static or dynamic feature activation. For example, software products designed for devices with constrained resources may use a static binding time to avoid the performance overhead introduced by dynamic binding time activation. However, other devices can exploit binding time flexibility to support products with a dynamic binding time for some of their features. To implement this kind of flexibility in a modular way, we can define AspectJ-based idioms. Researchers have proposed Edicts, an idiom based on AspectJ and design patterns. In this article, we argue that this idiom leads to an increase in code duplication, scattering, tangling and size, which can hamper code reuse, maintenance and understanding. To mitigate such issues, this paper proposes three idioms based on aspect-oriented programming to implement flexible feature binding. We apply our three idioms, along with Edicts, to implement a flexible binding time for features in four different applications. By doing so, we were able to assess the resulting implementations by using software metrics that judge code-quality factors. Our evaluation suggests that our idioms reduce the above-mentioned problems when implementing flexible feature binding for the selected features.

Keywords: flexible binding time; software product lines; software metrics; aspect-oriented programming

Received 11 February 2014; revised 24 June 2015
Handling editor: Mariangiola Dezani-Ciancaglini

1. INTRODUCTION

A software product line (SPL) is a family of software-intensive systems developed from reusable assets. By reusing assets, it is possible to construct a significant number of different products by applying compositions of different features [1]. These features define common characteristics as well as differences between the respective products within a family. They are also used to define whether these characteristics are mandatory, optional or alternative [2]. An SPL paradigm might offer several benefits regarding software development and maintenance, including improvements in the time to market, maintenance costs, productivity and product quality [1].

Depending on the requirements and composition mechanisms, features may be activated or deactivated flexibly, that is, at different times. In this context, features may be bound statically (i.e. during compile time or preprocessing) or dynamically (i.e. during run or link time). The benefit to the former is to facilitate an applications' customizability without any overhead at runtime [3]. Therefore, static feature binding is suitable for applications running on devices with limited resources, such as mobile phones. On the other hand, the latter allows for more flexibility in terms of performance costs and memory consumption [3]. Furthermore, if the developers are unaware of the set of features that should be activated before runtime, they can use dynamic feature binding to activate them on demand. Dynamic feature binding applies to Dynamic SPLs, which support late feature-composition changes to address requirements that vary at runtime [4, 5]. Thus, it is important to provide a flexible binding time for features so that their execution can be activated statically or dynamically [3, 6].

One way to support flexible feature binding is by means of aspect-oriented programming (AOP) [7], which allows features to be composed both statically and dynamically. Using aspects to achieve this goal is important because we can implement

these features in a modular way, that is, we can avoid problems, such as feature code tangling and scattering [7]. Thus, we focus on AOP solutions to support flexible binding time for features.

To the best of our knowledge, there is only one approach that implement flexible binding by means of aspects. Chakravarthy *et al.* [8] have proposed Edicts, an AOP idiom based on AspectJ [9] and design patterns [10]. Edicts provides a configurable binding time, meaning that developers can change the binding time (static or dynamic) without rewriting the source code and traceability. A flexible binding-time implementation is well modularized and easily identifiable, because it is located in only one package. The authors designed this idiom to improve static bind optimization and dynamic bind flexibility. Furthermore, they claim they successfully implemented flexible feature binding in a non-trivial product line, which Chakravarthy *et al.* [8] created by changing the JacORB middleware. In their paper, the authors provide evidence of the benefits of Edicts in this product line, what motivated us to explore the idiom in more detail.

Despite the claim made by the authors of Edicts that they have achieved these benefits with JacORB, we noticed that Edicts can lead to some problems when it is applied to static and dynamic binding for the same platform (i.e. exclusively for desktops or exclusively for mobiles). For example, duplicated feature code can be found in aspects that implement a static and dynamic binding time. This risks increasing the source code and unnecessarily making it difficult to understand, owing to the need to read replicated parts more than once [11]. Additionally, Edicts can scatter feature code across aspects and tangle different concerns, and this is evidence that feature implementations are not well modularized [12]. These issues might hamper code reuse, maintenance and understanding, and thereby might reduce development productivity. Moreover, as far as we know, there is no AOP-based solution in the literature that provides flexible feature binding in a well-modularized manner.

In this context, the problems introduced by Edicts motivated us to design solutions using AOP [13]. As a matter of fact, we use Edicts as a primary point of reference because the essence of our work is motivated by the problems beset by this idiom. Therefore, we propose three idioms to implement a flexible binding time for features. We use the term 'idiom' in a more liberal manner than it is commonly used in the design pattern community. Furthermore, we refer to 'idioms' because our increments are AspectJ- or CaesarJ-specific and they address a smaller and less general problem than patterns. We design our solutions to support flexible binding time for features by means of AOP. Despite structure similarity, our idioms are not Edicts-based and do not necessarily demand knowledge on Edicts.

The three idioms we propose are AOP-based, so we use AOP constructs to extract feature code and implement a static and dynamic binding time. To evaluate the advantages and disadvantages of Edicts and our idioms, we applied them to provide a flexible binding time for features in different product lines. In addition, we performed an assessment with respect to code duplication, scattering, tangling and source-code size by using a metric suite. In summary, our evaluation shows evidence that Edicts is subject to the problems just discussed, and suggests that our idioms mitigate these problems when implementing flexible feature binding for the selected features.

The main contributions of this work are the following:

(i) We identify deficiencies in Edicts, an existing idiom.
(ii) We address these deficiencies by defining three idioms for flexible feature binding.
(iii) We apply these 4 idioms (i.e. Edicts and the three we designed) to provide flexible binding for 18 features in 4 different product lines.
(iv) We assess these four idioms quantitatively, with respect to code cloning, scattering, tangling and size by means of software metrics.
(v) We discuss these implementations in terms of feature interaction and behaviour.

This paper improves our previous work [13] in four ways: by introducing a novel idiom, by evaluating several new features, by using a tool to compare the behaviour of the implementations and by considering two additional metrics to refine the evidence regarding the benefits and drawbacks of each idiom. In addition to considering features of a different nature, we have analysed implementations with alternative features (beyond merely optional, mandatory and 'OR' features [2]) and cases of feature interaction, which occur when a feature interacts structurally with another feature by modifying its source code [14]. This eliminates some threats to the external validity of our previous study. Besides a static evaluation by means of metrics, it is important to conduct a dynamic evaluation. Thus, we provide evidence that the feature code's execution behaviour does not change when using different idioms. To do so, we use the SafeRefactor tool [15] to generate a test suite for each flexible feature-binding implementation with each idiom. This is useful for detecting changes to the feature-code execution behaviour between implementations of flexible feature binding. In this manner, we can detect whether a feature code execution—which presents a flexible binding time implemented with one of the idioms—differs from the same feature code presenting a flexible binding time implemented with another idiom. Moreover, previous results [13] have led us to suspect that assessing the idioms by means of metrics at the level of operations (i.e. methods) will supplement and reinforce existing evidence concerning their advantages and drawbacks. Thus, to assess our idioms in a more comprehensive way, we considered two additional metrics in this article, allowing us to investigate code scattering (Degree of Scattering across Operations [16]) and tangling (Degree of Tangling within Operations [16]) at the level of operations (beyond the package and class levels). This will help us to better understand and explain differences between the idioms. In fact, some idioms are subject to a similar degree of scattering at the component level (i.e. classes or

aspects), whereas they exude a different degree of scattering at the level of operations. Moreover, to favour the replication of our work, we also provide additional information about the idioms [17].

We structured the remainder of this article as follows. In Section 2, we present the motivation for our work, detailing Edicts' problems. Section 3 introduces the three AOP-based idioms designed to mitigate these problems. In Section 4, we discuss how to apply an idiom to provide flexible feature binding in a scenario where features interact, and we show that these interactions are not harmful in that context. Following that, Section 5 presents the study settings and the selected applications. We also introduce the approach used to perform our evaluation and the assessment procedures we followed. Subsequently, Section 6 presents the evaluation results for Edicts and our three idioms regarding code cloning, scattering, tangling, size and feature behaviour. Finally, the remaining sections discuss threats to validity, related work, and conclusions.

2. MOTIVATION

In this section, we provide a brief introduction to Edicts [8]. More pertinently, we show the problems faced when applying this idiom, such as code cloning, scattering, and tangling.

As mentioned in the previous section, to implement a flexible binding time, we can use the Edicts idiom, which makes it possible to choose between feature binding when compiling (i.e. static binding) or during runtime (i.e. dynamic binding). The basic idea is to extract the feature implementation into AspectJ [9] aspects. The programmer applies the Edicts idiom using separate aspects for static and dynamic binding, as illustrated via the Unified Modelling Language (UML) in Fig. 1. The `AbstractAspect` contains concrete pointcuts that allow us to select join points in the base code where we should add feature behaviour when it is activated.

To implement the feature code, we use intertypes and pieces of advice. The intertypes represent structural changes that are necessary for a feature implementation that we do not need to deactivate because they are called only from within the feature implementation. Thus, the code is not executed when the feature is deactivated. Because we do not need to deactivate these declarations, we implement them in the `AbstractAspect`, avoiding feature code duplication across the concrete sub-aspects.

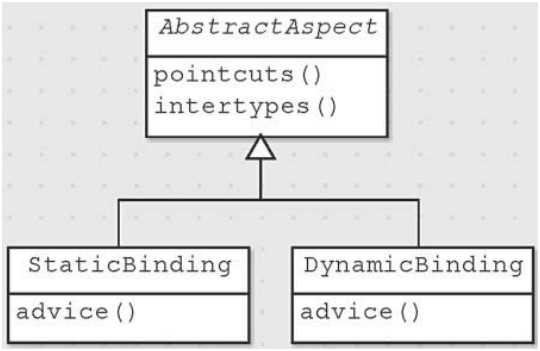

FIGURE 1. Structure of Edicts [8].

On the other hand, we implement the pieces of advice in two concrete sub-aspects. The `StaticBinding` and the `DynamicBinding` sub-aspects implement static and dynamic binding times, respectively. They differ because the `DynamicBinding` aspect implements a driver mechanism. This mechanism is used for dynamic deactivation of the feature. This can range from simple user-interface prompts to complex sensors that automatically determine whether a particular feature should be activated [18]. Thus, this mechanism allows features to be activated or deactivated at any time during execution or link time. In our case, the driver mechanism reads a property value from a properties file. For instance, to dynamically activate a given feature, we set `featurename=true` in the properties file. We do so for simplicity, because the complexity involved in providing information for feature activation is beyond of the scope of this article.

The structure in Fig. 1 shows only a single abstract aspect. We can apply this structure for each aspect defined in feature implementation. For instance, if a feature code is scattered across three aspects, we should abstract these three aspects and implement static and dynamic sub-aspects for each one.

Despite separating feature code and supporting a flexible binding time, Edicts might suffer from a number of issues, including code *cloning*, which is the presence of superfluous copies of the code, *scattering*, that is, code associated to a particular concern that appears in a number of the programme elements, and **tangling**, which is code associated to different concerns that nonetheless appears in the same program element.

To illustrate these problems, consider the *Checksum* optional feature from our BerkeleyDB product line. BerkeleyDB is an open-source database written in Java. The Checksum feature implementation detects data corruption when writing or reading a database page. Following Edicts' structure (Fig. 1), we have concrete pointcuts and intertypes in abstract aspects, and advice declarations in concrete sub-aspects. However, this design may lead to code *cloning* because we need to duplicate the pieces of advice between the concrete sub-aspects to implement a static and dynamic binding time. For the *Checksum* feature, 17 pieces of advice would require duplication. Listing 1 and 2 show a portion of the cloned code. The only difference appears in Lines 4, 9, 13 and 15 from Listing 2. These lines implement the aforementioned driver mechanism. Hence, the code between Lines 4 and 9, and 13 and 15 from Listing 2 is executed only if the driver activates the feature, which can be based on a user decision, for example. For simplicity, we have not shown the `ChecksumAbstract` implementation, which

declares the concrete pointcuts and intertypes related to the *Checksum* feature and referenced by the concrete sub-aspects.

In fact, applying Edicts to our selected features requires cloning each piece of advice between the concrete sub-aspects of 18 features, resulting in 177 pieces of cloned advice in our studies. Whereas it is not always the case that code cloning is harmful [19], in this situation it is clear that duplications will complicate code maintenance and evolution [20, 21]. We provide further details in Section 6.1.

Listing 1. Checksum static binding.

```
1  aspect ChecksumStatic extends ChecksumAbstract {
2    ...
3    after(LogManager lm) : readHeader(lm) {
4     if (lm.doChecksumOnRead) {
5       validator = new ChecksumValidator();
6       ...
7     }
8    }
9    after(FileReader fr)
10   : lastFileReaderConstructor(fr) {
11    fr.anticipateChecksumErrors = true;
12   }
13   ...
14 }
```

Listing 2. Checksum dynamic binding.

```
1  aspect ChecksumDynamic extends ChecksumAbstract {
2    ...
3    after(LogManager lm) : readHeader(lm) {
4     if (driver.isActivated("checksum")) {
5       if (lm.doChecksumOnRead) {
6         validator = new ChecksumValidator();
7         ...
8       }
9     }
10   }
11   after(FileReader fr)
12   : lastFileReaderConstructor(fr) {
13    if (driver.isActivated("checksum")) {
14      fr.anticipateChecksumErrors = true;
15    }
16   }
17   ...
18 }
```

In addition to cloning, the code in Listing 2 is not restricted to feature implementation. It is *tangled* with the code from the aforementioned driver mechanism. There is no separation of the driver and feature concerns, and this can lead to increased complexity in terms of any addition, removal or modifications to the driver code [22]. For example, we may need to change the driver mechanism owing to the configuration of a new environment where the application is running, such as one with less memory resources. In such a scenario, we must add a new driver condition into each piece of advice. Moreover, the situation is exacerbated when applying Edicts to large features that require multiple advice declarations, because this mechanism code is tangled with a feature implementation for each piece of advice. Even if we use an interface to reference the driver within the pieces of advice, the problem persists because these interfaces may vary as well. When applying Edicts, we use aspects to implement the feature code, although we do not use aspects to implement the driver mechanism code.

Furthermore, there are also issues to introducing the driver mechanism. This happens because `if` statements like the one in Lines 4 and 13 (in Listing 2) are *scattered* and **tangled** throughout the advice in order to support a dynamic binding time. Such code scattering is error-prone, because forgetting an `if` statement may give raise to runtime exceptions in the application, such as a `NullPointerException`. Changing the driver mechanism is also time consuming because we have to alter the driver code within each piece of advice.

The cloning issue can be mitigated by writing the feature code as methods in a separate aspect or class, one that is not a part of the application's logic. Indeed, we could decrease code cloning by duplicating only advice declarations and calling these methods in the advice. Thus, the cloning problem can be alleviated but not eliminated. Furthermore, this solution will often fail, because AspectJ does not support `proceed` calls outside the advice. Thus, it cannot be called from within the methods in a separate aspect. Moreover, we cannot pass arguments through these absent `proceed` calls, potentially leading to inconsistent object states. For instance, if we adopt this method to alter an object that is used in the base code, we would lose its new state. If we create a separate class, instead of an aspect, to implement the aforementioned methods, we ultimately worsen the problem, because classes do not support `privileged` access to non-public members, so we would have to change the visibility of non-public methods called within the advice body.

Furthermore, this approach to mitigating the cloning issue comes at the cost of increased code scattering, because it uses an additional component that contains feature code. For example, feature code would be scattered in 26 new aspects considering the 18 features we use in this work. Likewise, the *Checksum* feature implementation would need four aspects instead of three. The feature code is even more scattered when its implementation uses more aspects. Nevertheless, the tangling would remain the same, because the `if` statements (in Lines 4 and 13 from Listing 2) would still be implemented in the same way, insofar we would only extract feature code.

These problems are mostly related to the design of Edicts and its use of AspectJ. They are more evident when implementing dynamic feature binding, owing to the inclusion of a driver, which increases the difficulty of implementation, while allowing for dynamic product lines and late-feature compositions. These problems are detrimental in terms of code reuse, maintenance and understanding. Therefore, we propose three idioms to address Edicts' shortcomings in the next section.

3. FLEXIBLE BINDING TIME IDIOMS

In this section, we describe our idioms to provide flexible binding time for features with the aim of mitigating the discussed problems with Edicts. In Section 3.1, we introduce the Pointcut Redefinition idiom. It avoids cloning pieces of advice as well as

reduces feature and driver code tangling and scattering. However, this idiom may increase the implementation size because it redefines the pointcuts related to the feature. Hence, we describe the Layered Aspects idiom in Section 3.2. It also addresses the discussed problems with Edicts, but the implementation size and the driver scattering are smaller. Additionally, we introduce the Flexible Deployment idiom in Section 3.3 to evaluate an idiom which is not based on AspectJ. Unlike the others, this idiom relies on CaesarJ [23], which allows an improvement regarding the discussed problems, but on the other hand, it does not work in some cases that we describe in Section 6.9.

For all the three idioms, we adopt a project build strategy. The structure of these idioms introduce at least one aspect for feature code and two more aspects for static and dynamic binding. In this context, if a developer wants to provide static binding, she must include the aspect that contains feature code and the other that implements static binding in the project build. On the other hand, if she wants to provide dynamic binding, she must add to the project build the aspect that contains feature code and the other that contains dynamic binding implementation. Furthermore, the aspects that implement static and dynamic binding do not coexist in a single project build. We walk through the details in the following sections.

3.1. Pointcut Redefinition

The Pointcut Redefinition idiom uses inheritance of AspectJ aspects and redefinition of pointcuts to provide binding time flexibility. Essentially, we extract the feature code into an abstract aspect. To implement static binding, we define an empty concrete subaspect to permit feature code instantiation by inheriting the abstract aspect. This is necessary due to the AspectJ limitation, in which an aspect can inherit from another one only if the latter is abstract [24]. By compiling the application with this concrete subaspect, we activate all intertypes and advice declarations, so the application execution will run the feature code. On the other hand, if we do not compile this concrete subaspect, the application execution will not run the feature code. For dynamic binding, we define another concrete subaspect that redefines the pointcuts from the abstract aspect restricting them with the driver mechanism, that is, the new pointcuts we define in this aspect. In this way, when compiling the application with this concrete subaspect, we are able to dynamically decide whether the feature code is executed. In what follows, we provide details about this idiom.

3.1.1. Design

Pointcut Redefinition is implemented using AspectJ aspects. We define an abstract aspect, which may contain advice, pointcuts, and intertypes related to the feature code. Figure 2 illustrates an overview of the Pointcut Redefinition's structure.

This idiom provides static binding by an empty concrete subaspect to allow the `AbstractAspect` instantiation and

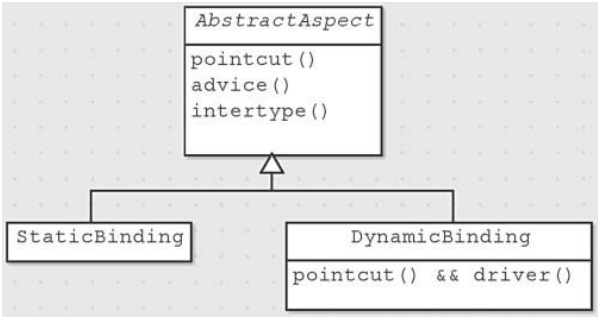

FIGURE 2. The structure of Pointcut Redefinition.

the feature code execution. For dynamic binding, we redefine the `AbstractAspect` concrete pointcuts and associate them with the driver code in the `DynamicBinding` aspect. The driver mechanism (or a set of mechanisms) is responsible for providing information about whether a feature should be executed at runtime. In this way, the pointcuts defined in the `AbstractAspect` intercept the corresponding join points only if the driver activates the feature execution. Moreover, static and dynamic binding time do not coexist at the same product. We include only the `StaticBinding` aspect or the `DynamicBinding` aspect in the compilation with the `AbstractAspect`. Therefore, we provide three different variability possibilities: feature dynamically bound or unbound, feature statically bound and feature statically unbound. This also applies to our idiom in Section 3.2.

The discussed structure of Pointcut Redefinition avoids cloning advice code since there is no need to duplicate pieces of advice with feature code between the concrete subaspects because they are in the abstract aspect. Moreover, it solves the tangling between driver and feature code because the driver mechanism is implemented in a separate subaspect (`DynamicBinding`). There is no feature code in the concrete subaspects and no driver code in the abstract aspect, so they are not scattered between the three aspects. However, the redefinition of pointcuts may increase the idiom's implementation size when several pointcuts are present, since each one of these pointcuts are redefined, which duplicates its size. This increase could be tiny comparing with the application code size, although it could be significant when comparing with the size of other idioms. This also leads to driver-code scattering in an operation level, since several pointcuts are associated with driver code.

3.1.2. Example

To explain the Pointcut Redefinition idiom more clearly, we use the same example from Section 2, the *Checksum* optional feature.

Feature implementation. The abstract aspect contains intertype and advice declarations as well as the pointcuts related to the

Checksum feature code. Listing 3 shows a simplified abstract aspect that implements the aforementioned elements. Lines 3 and 6 define pointcuts that match join points where the feature code should be executed. The pieces of advice defined between Lines 10 and 20, contain feature code that should be executed in the join points matched by those pointcuts. Moreover, Lines 22–24 define an intertype declaration that introduces the `validateChecksum` method in the `FileReader` class. In contrast to the code in the pieces of advice, the code of this method should not be related to a given join point, the own method is referenced by another member in the feature implementation, therefore we use intertype declaration.

Listing 3. ChecksumAbstract.

```
1  abstract aspect ChecksumAbstract {
2   ...
3   pointcut readHeader()
4   : call(void EntryHeader.readHeader());
5   ...
6   pointcut addPrevOffset(int entrySize)
7   : execution(ByteBuffer LogManager.addPrevOffset())
8   && args(entrySize);
9   ...
10  after() throws DatabaseException : readHeader() {
11   if (doChecksumOnRead) {
12    validator = new ChecksumValidator();
13    ...
14   }
15  }
16  ...
17  ByteBuffer around(int entrySize)
18  : addPrevOffset(entrySize) {
19   return proceed(entrySize);
20  }
21  ...
22  void FileReader.validateChecksum() {
23   ...
24  }
25 }
```

To avoid cloning the pieces of advice, we implement them in the abstract aspect differently from Edicts. We may have as many aspects as needed to extract code related to the feature, this is an engineering decision. Nevertheless, we apply the structure illustrated in Fig. 2 for each created aspect.

Implementing static binding time. As mentioned in Section 3.1.1, we create an empty concrete subaspect inheriting the `ChecksumAbstract` aspect to allow its instantiation. Making the `ChecksumAbstract` aspect concrete to avoid this empty aspect would not be possible because the `ChecksumDynamic` aspect needs to extend `ChecksumAbstract` in order to implement the dynamic binding time, since AspectJ aspects can only be inherited if they are abstract [24].

To statically activate the feature execution, we include the `ChecksumAbstract` and `ChecksumStatic` aspects in the project build. On the other hand, to statically deactivate the feature execution, we do not include any of these aspects.

Implementing dynamic binding time. Now we are ready to define the aspect responsible for dynamic binding time. Lines 3 and 4 of Listing 4 show the driver implementation. It uses the AspectJ `if` pointcut, which matches each join point where the boolean expression evaluates to `true`. In this case, the boolean expression checks if the checksum property corresponds to `true` or `false` in a properties file. If this property's value corresponds to `true`, the feature is activated and its code should be executed. In Lines 5–8, we redefine the pointcuts defined in `ChecksumAbstract` and associate the driver mechanism. Thus, the driver controls whether the pointcuts are dynamically applied. If the feature execution is activated, the pointcuts are applied and consequently the code within the pieces of advice is executed. On the other hand, if the feature execution is deactivated, the pointcuts are not applied and the feature code is not executed.

Listing 4. ChecksumDynamic.

```
1  aspect ChecksumDynamic extends ChecksumAbstract {
2
3   pointcut driver():
4   if(new Driver().isActivated("checksum"));
5
6   pointcut readHeader()
7   : ChecksumAbstract.readHeader() && driver();
8
9   pointcut addPrevOffset(int entrySize)
10  : ChecksumAbstract.addPrevOffset(entrySize)
11  && driver();
12 }
```

To observe the Pointcut Redefinition disadvantages, notice that when we define several pointcuts, the dynamic binding implementation may increase its size because we redefine all pointcuts. Additionally, we scatter the driver throughout the redefined pointcuts, as shown in Lines 3, 4, 7 and 11 of Listing 4. To mitigate these problems, we introduce the Layered Aspects idiom.

3.2. Layered Aspects

Now we propose the Layered Aspects idiom to implement flexible binding time for features. Similarly to Pointcut Redefinition, the basic idea of Layered Aspects is to implement feature code in an abstract aspect and two concrete subaspects exclusively to implement static and dynamic binding time. For the static binding time, we compile an empty concrete subaspect which inherits the feature code from the abstract aspect and allows its instantiation equally to the Pointcut Redefinition idiom and due to the same AspectJ limitation, which only allows aspect inheritance from an abstract aspect. For the dynamic binding time, we compile another concrete subaspect, which matches the execution of the pieces of advice that implement feature code defined in the abstract aspect to dynamically decide whether the feature code is executed. We implement this carefully using the `adviceexecution` pointcut provided by AspectJ, which matches these advice execution join points and allows the prosecution of the feature code execution or not. We adopt the Layered Aspects name because there is one aspect inheriting from another, or one affecting another through the `adviceexecution` pointcut. In the following, we show

this idiom's design and how it tries to address the problems of Edicts and Pointcut Redefinition.

3.2.1. Design

Similarly to Pointcut Redefinition, we implement the feature code in an abstract aspect. This aspect may contain pieces of advice, pointcuts and intertype declarations associated with the feature code. Then, we implement static and dynamic binding time within two concrete subaspects inheriting the abstract one, as illustrated in Fig. 3. An empty concrete aspect (`StaticBinding`) is necessary in the compilation to allow the `AbstractAspect` instantiation when a static feature binding occurs. For dynamic binding of features, we compile the `DynamicBinding` aspect, which implements code for dealing with different kinds of advice defined in `AbstractAspect`.

For `before` and `after` advice, the `adviceexecution` pointcut matches their join points and only proceeds their execution if the feature is activated.

For `around` advice, the `adviceexecution` pointcut does not work because when the driver states the feature deactivation, we must restore the base code overridden by the `around` advice. Since there is no way to access the `proceed()` join point of the advice intercepted by the `adviceexecution` pointcut, it is not possible to call it in a generic way. Thus, the pieces of `around` advice of the feature implementation must be deactivated one by one. For example, Fig. 4 illustrates this explanation. Since the advice defined in aspect `A` does not call `proceed()`, the advice in aspect `B` is not executed and consequently, `System.out.print("BaseCode")` is not executed either. Therefore, missing the base code execution is the reason we need to deactivate pieces of `around` advice one by one. Hence, we use aspect inheritance to redefine the pointcuts related to `around` advice declarations and associate them with the driver in the `DynamicBinding` aspect. In this way, `DynamicBinding` contains redefinitions of pointcuts that are related to `around` advice, similarly to Pointcut Redefinition. Thereby, we compile the application with the `DynamicBinding` aspect to provide dynamic binding time for features.

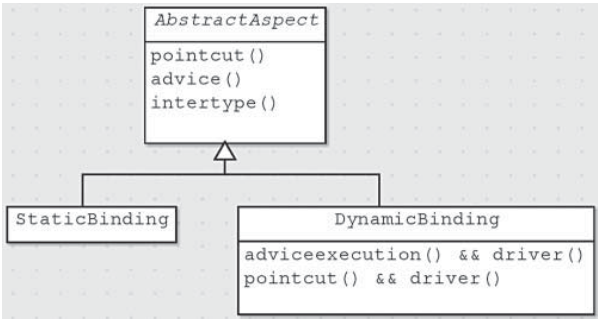

FIGURE 3. The structure of Layered Aspects.

Furthermore, one may argue why we deny the `driver()` and do not call `proceed()` within the advice defined in aspect `A`. Actually, if we do that, the feature code would always be executed. If the `driver()` was evaluated to `true`, the feature code would normally be executed. However, if the `driver()` was evaluated to `false`, then the `adviceexecution` pointcut would not intercept any piece of advice, so the feature code would be executed too.

The aforementioned structure of Layered Aspects avoids feature code cloning, tangling and scattering for the same reasons Pointcut Redefinition does. Layered Aspects does not duplicate advice with feature code between `StaticBinding` and `DynamicBinding`, since these pieces of advice are defined only in `AbstractAspect`. Additionally, it does not tangle driver and feature code because we implement the driver mechanism only in `DynamicBinding`, which does not contain feature code. Furthermore, the feature code is not scattered between the concrete subaspects because we implement it solely in `AbstractAspect`. However, Layered Aspects may increase its implementation size when several `around` advice are present due to the discussed `adviceexecution`

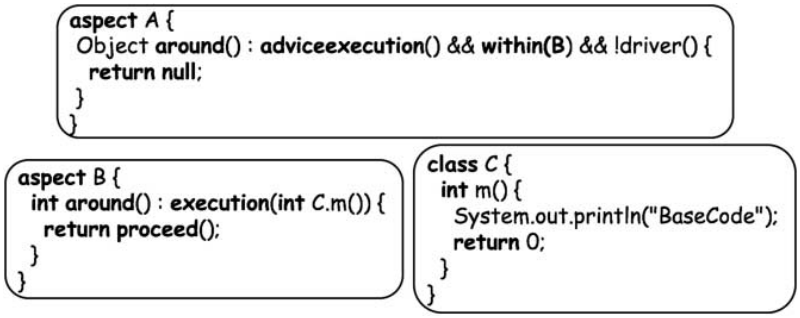

FIGURE 4. Around advice limitation.

pointcut. In the following, we provide more details about Layered Aspects.

3.2.2. *Example*

To better explain Layered Aspects, consider the *Checksum* feature introduced in Section 2. We show how to apply flexible binding time for this feature and how this idiom addresses the Edicts shortcomings. We go straight to the dynamic binding time explanation because the feature and static binding time implementations are identical to the one presented in Section 3.1.2. The difference of Layered Aspects consists of implementing the dynamic binding time, as follows.

Implementing dynamic binding time. Now we show how Layered Aspects allows dynamic feature activation. Listing 5 shows how we implement dynamic binding of features. Lines 3 and 4 define the driver, which is a pointcut that checks if the checksum property corresponds to `true` or `false` in a property file, equally to the driver defined in Lines 3 and 4 of Listing 4. The feature code is executed depending on the evaluation of this conditional. However, as mentioned in Section 3.1.2, the driver could be differently implemented. Additionally, for dynamic feature binding, Line 9 implements the `adviceexecution` pointcut to deal with `before` and `after` advice. If the `driver()` condition corresponds to `false`, the feature is deactivated, so this pointcut intercepts the pieces of advice defined in the `ChecksumAbstract` aspect, but the other advice defined in Lines 9–13 does not call the `proceed()` join point, so the feature code is not executed. On the other hand, if the `driver()` condition is `true`, the feature is activated, so this pointcut does not intercept any advice declaration, thus the feature code is executed.

In this context, we avoid infinite recursion [25] when using `adviceexecution` because we specify which aspect is advised by this pointcut in Line 10. Therefore, the `ChecksumDynamic` aspect cannot advise itself. Moreover, Lines 6 and 7 redefine the `addPrevOffset` pointcut, which is defined in the `ChecksumAbstract` aspect in order to associate it with the driver because this pointcut is related to an `around` advice. As explained in Section 3.2.1, such type of advice is handled separately, it is deactivated one by one. Therefore, we implement the dynamic binding following the Pointcut Redefinition structure for `around` advice.

Listing 5. ChecksumDynamic.

```
1  aspect ChecksumDynamic extends ChecksumAbstract {
2    ...
3    pointcut driver()
4    : if(new Driver().isActivated("checksum"));
5
6    pointcut addPrevOffset()
7    : ChecksumAbstract.addPrevOffset() && driver();
8
9    Object around()
10   : adviceexecution() && within(ChecksumAbstract)
11    && !driver() {
12     return null;
13    }
14 }
```

Furthermore, returning `null` in Line 12 (Listing 5) is not harmful when the feature is deactivated because we apply the `adviceexecution` pointcut only for `before` and `after` advice, so it does not intercept `around` advice. Therefore, when the feature is deactivated, the new pointcuts related to `around` advice are not applied and so the `adviceexecution()` does not intercept the execution of the advice defined in Lines 17–20 (Listing 3). If we remove the `null` statement, we may have a compilation error, since the `adviceexecution()` pointcut statically targets the pieces of advice defined in `ChecksumAbstract` that return an `Object` or a primitive type.

Although Layered Aspects reduces some issues, this idiom could scatter driver code when several `around` advice are defined because we redefine the pointcuts related to it, as discussed above. The Pointcut Redefinition idiom presents the same deficiency, however, it redefines all the pointcuts whereas Layered Aspects redefines only the pointcuts related to `around` advice. Hence, Layered Aspects design does not reduce the implementation size comparing with Pointcut Redefinition in such cases. To handle these issues, we present next the Flexible Deployment idiom.

3.3. Flexible Deployment

The Flexible Deployment idiom uses dynamic deployment of aspects provided by CaesarJ [23] to allow feature binding time flexibility. CaesarJ is an aspect-oriented language that extends Java with support for reusability by means of aspect-oriented constructs, such as pointcut and advice, as well as object-oriented modularization mechanisms, like CaesarJ classes. Furthermore, CaesarJ supports dynamic deployment of classes and advanced object-oriented modularization mechanisms.

We select this language to provide an additional idiom without introducing some of AspectJ limitations, such as the `adviceexecution` limitation, explained in Section 3.2. CaesarJ supports a dynamic aspect deployment mechanism to allow control over scope [26], which we use to implement dynamic flexible binding for features. Furthermore, AspectJ reuse mechanisms are limited to abstract aspects and aspect inheritance. In particular, CaesarJ promises to overcome these limitations. For example, CaesarJ supports the technique of coding interfaces, virtual types and mixin composition [26]. On the other hand, other aspect-oriented languages (e.g. ABC [27]) present the same AspectJ reuse limitations.

The idea to provide flexible binding time is to define a CaesarJ class, which supports the definition of aspect-oriented constructs, to implement feature code and two additional CaesarJ classes to implement static and dynamic binding. For static binding, we define a statically deployed CaesarJ class that inherits another CaesarJ class containing the feature code. For dynamic binding, a deployed CaesarJ class implements the driver mechanism. It activates the feature by dynamically deploying the CaesarJ class that contains the feature code and

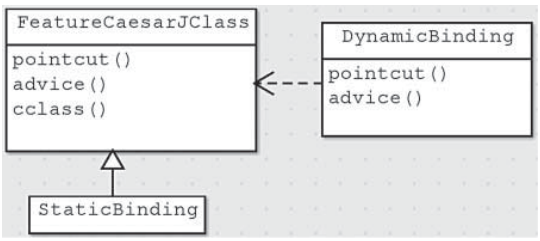

FIGURE 5. The structure of Flexible Deployment.

consequently allowing its execution. We provide more details about this idiom in the following sections.

3.3.1. *Design*

Flexible Deployment is implemented using CaesarJ classes. We define a CaesarJ class to implement a given feature. It may contain pointcuts, advice and wrapper classes associated with the feature implementation. These wrapper classes, which are provided by CaesarJ, dynamically extends other classes. They can introduce new fields and methods [23]. Unlike AspectJ intertype declarations, these wrapper classes do not introduce feature code in base code classes. Figure 5 shows the structure of Flexible Deployment.

Furthermore, we implement static binding by defining an empty deployed CaesarJ class (`StaticBinding`) that inherits from `FeatureCaesarJClass`. Thereby, when both CaesarJ classes are present in a build, the feature is statically activated. For dynamic binding, we define a separate CaesarJ class (`DynamicBinding`) which contains the driver mechanism implemented as a pointcut and an advice. In Section 3.3.2, we provide more details.

This idiom reduces code cloning, tangling, scattering and size. We implement feature code in a separate CaesarJ class. Therefore, there is no cloning of pieces of advice and there is no feature code scattering throughout the classes either. Since we implement the driver mechanism in a separate CaesarJ class, the driver code is not tangled with feature code. Moreover, the idiom implementation size does not increase when several pointcuts are present as Pointcut Redefinition does.

3.3.2. *Example*

Now we use the *Checksum* feature to describe how to implement flexible binding time using the Flexible Deployment idiom.

Feature implementation. The CaesarJ class that contains feature code may define pointcuts, advice and wrapper classes, as illustrated in Listing 6. Differently from AspectJ, CaesarJ does not support intertypes, so we define wrapper classes (Lines 23–28) in `ChecksumCaesarJClass` instead. The `FileReaderCaesarJ` wrapper class dynamically extends the `FileReader` class. It defines the `validateChecksum` method, which is part of the feature implementation (Lines 25–27). The Lines 3–8 define pointcuts that match certain join points in the base code where the feature code should be executed. Lines 10–21 define pieces of advice that contains this feature code.

Listing 6. ChecksumCaesarJClass

```
1  cclass ChecksumCaesarJClass {
2   ...
3   pointcut readHeader()
4   : call(void EntryHeader.readHeader());
5
6   pointcut addPrevOffset(int entrySize)
7   : execution(ByteBuffer LogManager.addPrevOffset())
8   && args(entrySize);
9
10  after() throws DatabaseException : readHeader() {
11   if (doChecksumOnRead) {
12    validator = new ChecksumValidator();
13    ...
14   }
15  }
16
17  ByteBuffer around(int entrySize)
18  : addPrevOffset(entrySize) {
19   ...
20   return proceed(entrySize);
21  }
22
23  cclass FileReaderCaesarJ wraps FileReader {
24  ...
25   void validateChecksum() {
26    ...
27   }
28  }
29 }
```

Implementing static binding time. In CaesarJ, instantiation does not automatically activate an aspect. It must be deployed in order to activate its pointcuts and advice [23]. This deployment may be done statically by introducing the key word `deployed` before the CaesarJ class declaration. In this context, for the static binding time, we define a `deployed` CaesarJ subclass extending `ChecksumCaesarJClass`, as illustrated in Listing 7. Therefore, we activate the feature code execution by including both CaesarJ classes in the project build. On the other hand, to deactivate feature code execution, we do not include any of these CaesarJ classes.

Listing 7. ChecksumStatic.

```
1 deployed cclass ChecksumStatic
2  extends ChecksumCaesarJClass {
3 }
```

Implementing dynamic binding time. Now we describe how the Flexible Deployment idiom allows dynamic feature activation. Listing 8 shows the driver mechanism implementation in a `deployed` CaesarJ class. In this case, we define a pointcut that intercepts the system main method execution in Lines 2 and 3. This allows the advice defined in Line 4 to dynamically deploy the CaesarJ class that contains feature code (`ChecksumCaesarJClass`) before the main method execution. Thereby, the feature code is executed depending on the driver mechanism. Furthermore, this driver mechanism is implemented according to the application requirements. Thus, it is not necessarily implemented as in Listing 8. As we

mentioned in Section 3.1.2, it could vary from simple GUIs to complex sensors.

Indeed, Flexible Deployment implementation does not clone or scatter feature code because we do not need to duplicate it in the classes that implement static and dynamic binding time, so we implement feature code only in one class. Moreover, this idiom does not tangle feature and driver code since we define a separate class to implement the driver mechanism. Finally, Flexible Deployment reduces the implementation size comparing to the other idioms because its size does not vary due to certain advice, as in Layered Aspects.

Listing 8. ChecksumDynamic.

```
1 deployed cclass ChecksumDynamic {
2   pointcut pc_jarmain()
3   : execution(* JarMain.main(..));
4   before() : pc_jarmain() {
5     if (new Driver().isActivated("checksum")) {
6       deploy new ChecksumCaesarJClass();
7     }
8   }
9 }
```

However, CaesarJ does no support some AspectJ constructs, such as `declare parents` [23]. Because of this disadvantage, we do not apply the Flexible Deployment idiom to provide flexible binding time for four features out of our 18 selected features. The flexible binding time implementation for these four features with the other idioms needs the `declare parents` construct, so we would need to do several changes to implement flexible binding time with Flexible Deployment, which could introduce bias in the evaluation. This is the reason why we do not show the metric results, in Section 6 for the features *NextPiece-Desktop*, *NextPiece-Mobile*, *Record-Desktop* and *Record-Mobile* concerning the Flexible Deployment idiom. We provide further details in Section 6.9.

At last, we provide a summary of the key discriminators between the four idioms presented so far in Table 1.

4. FEATURE INTERACTION

So far, we explained how to implement flexible binding time without considering features that interact. In this section, we explain how to apply Edicts into a feature interaction scenario. Moreover, our idioms might be applied in an analogous way. Thus, we omit their implementation in this section. However, we provide the corresponding source code [17].

A feature interaction occurs when one or more features modify or influence other features [28, 29]. As Liu *et al.* state, there are different ways in which feature interact [14], such as behavioural or structural interaction. For this work, we focus on interactions that are static and structural, that is, how the flexible binding time implementation for a feature may influence others when activating or deactivating this feature. Moreover, considering feature interactions is important because they can be damaging to application development and user expectations [30]. Besides that, the concepts of feature interaction used in this work are compatible with Sobering's [31].

In such feature interaction scenarios, we may need to apply flexible binding time as well. Therefore, we explain how to deal with feature interaction when applying the idioms discussed before.

4.1. Example of feature interaction

To better explain flexible binding time within feature interaction context, consider the *NextPiece* optional feature of our Tetris product line, which is our only case of interaction. Basically, this feature execution shows the next piece which is about to drop on the screen of a simple Tetris game. Moreover, we can run this game on *Mobile* and *Desktop* environments, which are mutually exclusive features. Thus, we need to implement *NextPiece* code specifically for each environment mainly concerning the user interface, which lead to an interaction between these features. We choose the Tetris application because its features interact and it is small and easy to understand. Further, we provide more details about this application in Section 5.1.1.

To exemplify feature interaction in Tetris, Table 2 illustrates a configuration knowledge of the feature interaction scenario. For example, if the *NextPiece* and *Mobile* features should be executed, we must include `Mobile.aj`, `NextPieceAbstract.aj`, `NextPieceStatic.aj`, `MobileNextPieceAbstract.aj` and `DesktopNextPieceDynamic.aj` aspects in the project build. Note that the environment where we can execute the *NextPiece* feature may vary, that is, we can run this feature on *Mobile*

TABLE 1. Summary of the idioms.

	Edicts	Pointcut Redefinition	Layered Aspects	Flexible Deployment
Language	AspectJ	AspectJ	AspectJ	CaesarJ
`adviceexecution`	No	No	Yes	No
Redefinition of pointcuts	No	Yes	Depends	No
Duplication of aspects	Yes	No	No	No
Feature and driver code tangling	Yes	No	Depends	No
Mobile and desktop platforms	Yes	Yes	Yes	No

TABLE 2. Tetris configuration knowledge and binding mode.

Feature expression	Binding mode	Functionality	Components
Mobile	Static	Provides support to run on mobile environment	Mobile.aj
Desktop	Static	Provides support to run on desktop environment	Desktop.aj
NextPiece	Static	Shows the next piece to drop on the screen	NextPieceAbstract.aj, NextPieceStatic.aj, and NextPieceDynamic.aj
NextPiece ∧ *Mobile*	Static	Provides support to see the next piece when running on mobile environment	MobileNextPieceAbstract.aj and MobileNextPieceStatic.aj
NextPiece ∧ *Desktop*	Dynamic	Provides support to see the next piece when running on desktop environment	DesktopNextPieceAbstract.aj and DesktopNextPieceDynamic.aj

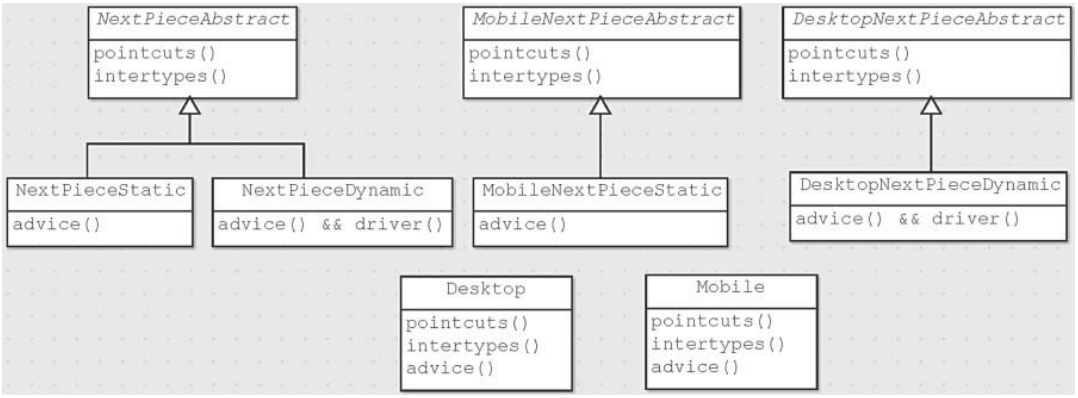

FIGURE 6. The structure of Edicts with feature interaction.

or *Desktop* environments. In this way, we observe the feature interaction scenario, since we need to define additional aspects (`MobileNextPieceAbstract.aj` and `MobileNextPieceStatic.aj` for *Mobile*, and `Desktop NextPieceAbstract.aj` and `DesktopNextPiece Dynamic.aj` for *Desktop*) to implement part of *NextPiece* code specifically to *Mobile* or *Desktop*. Furthermore, the *NextPiece* behaviour varies depending on these environments where it is running.

4.2. Implementing flexible binding time

First, we summarize the features and the corresponding binding time we take into account in Table 2. In this example, the mutually exclusive features *Mobile* and *Desktop* present only static binding time, since it does not make sense to dynamically change the environment where the application is running. However, we want to provide dynamic binding time for the *NextPiece* optional feature when it is running on *Desktop* environment and static binding time when it is running on *Mobile* environment. This decision is based on requirements and we provide further details in Section 5.1.1.

To explain how to apply Edicts in the context of feature interaction, we implement flexible binding time with this idiom for the *NextPiece* feature in both *Mobile* and *Desktop* environments. We omit the flexible binding time implementation with the other idioms because it would be repetitive and the application of their concepts is analogous to the application of Edicts concepts.

In Fig. 6, we illustrate an overview of Edicts structure in the context of feature interaction. Following the concepts of this idiom, the `NextPieceAbstract` aspect contains pointcuts and intertype declarations. Additionally, `NextPieceStatic` and `NextPieceDynamic` implement static and dynamic binding time, respectively. These three aspects concern exclusively to the *NextPiece* feature. On the other hand, the `Mobile` and `Desktop` aspects implement exclusive code of *Mobile* and *Desktop* features. As mentioned before, we do not implement dynamic binding for these

features. Moreover, the `MobileNextPieceAbstract`, `MobileNextPieceStatic`, `DesktopNextPiece Abstract` and `Desktop- NextPieceDynamic` aspects implement static and dynamic binding for the interaction between *NextPiece*, *Mobile* and *Desktop* features. To show the implementation of Edicts, we follow the same text structure of Section 3.

Listing 9. NextPieceBoxAbstract

```
1 abstract aspect NextPieceBoxAbstract {
2
3  NextPieceBox TetrisCanvas.nextPieceBox;
4
5  pointcut createNextPiece(TetrisCanvas cthis, ...)
6  :execution(*TetrisCanvas.createNextPieceBoxHook())
7  && this(cthis) && args(...);
8 }
```

Feature implementation. Differently from the explanation of *Checksum* (Section 2), we now deal with three features that interact. Therefore, by following the concepts of Editcs, we create at least three abstract aspects for each feature. Thus, Listing 9 contains pointcuts and intertypes corresponding exclusively to the *NextPiece* feature.

The Listings 10 and 11 illustrate two aspects that implement the *Mobile* and *Desktop* features. These aspects contain pointcuts, intertypes and pieces of advice corresponding solely to these features. Note that we do not have to create concrete subaspects in order to implement static and dynamic binding time because we just include one of these aspects in the project build according to the feature that should be statically activated. This justifies why we implement pieces of advice in these aspects. The `declare parents` in Line 3 determines which class `TetrisCanvas` extends, that is, `Canvas` for *Mobile* environment or `JPanel` for *Desktop* environment. On the other hand, the pointcuts and advice start the application for *Mobile* or *Desktop*.

Listing 10. Mobile

```
1 aspect Mobile {
2  ...
3  declare parents : TetrisCanvas extends Canvas;
4
5  pointcut startApp(TetrisMidlet cthis)
6  : execution(* TetrisMidlet.startApp(..))
7  && this(cthis);
8
9  after(TetrisMidlet cthis) : startApp(cthis) {
10  Display.getDisplay(cthis)
11  .setCurrent(cthis.gameCanvas);
12  }
13 }
```

Listing 11. Desktop

```
1 aspect Desktop {
2  ...
3  declare parents : TetrisCanvas extends JPanel;
4
5  pointcut startApp(TetrisMidlet cthis)
6  : execution(* TetrisMidlet.startApp(..))
7  && this(cthis);
8
9  after(TetrisMidlet cthis) : startApp(cthis) {
10  cthis.setVisible(true);
11  }
12 }
```

Last but not least, we define abstract aspects in Listings 12 and 13 to implement feature interaction code. First, `MobileNextPieceAbstract` defines pointcuts and intertypes related to the interaction between *Mobile* and *NextPiece* features. Therefore, this aspect contains *NextPiece* feature code to run only on the *Mobile* environment. We declare in Line 6 the `captionFont` variable by using intertype, which is defined with type `Font`. Besides this intertype declaration, we define in Line 8 the `paintInfoBoxes` pointcut, which uses a `Graphics` object as one of its parameters. Note that we import the objects `Font` and `Graphics` from Java Micro Edition libraries, which is different from Java Standard Edition libraries. On the other hand, in `DesktopNextPieceAbstract` aspect, we import `Font` and `Graphics` objects from Java Standard Edition libraries, as shown in Lines 1 and 2 of Listing 13. These *NextPiece* implementation differences between *Mobile* and *Desktop* environments could vary from simple imports to different use of variables or calculations of the position of objects in the user interface.

Listing 12. MobileNextPieceAbstract

```
1 import javax.microedition.lcdui.Font; import
2 javax.microedition.lcdui.Graphics;
3
4 abstract aspect MobileNextPieceAbstract {
5
6  Font NextPieceBox.captionFont;
7
8  pointcut paintInfoBoxes(TetrisCanvas cthis,
9  Graphics g)
10  : execution(* TetrisCanvas.paintInfoBoxes(..))
11  && this(cthis) && args(g);
12 }
```

Implementing static binding time. As we mentioned before, we implement static feature binding for *NextPiece* when the Tetris application is configured to run on *Mobile* environment.

Listing 13. DesktopNextPieceAbstract

```
1 import java.awt.Font; import java.awt.Graphics;
2
3 abstract aspect DesktopNextPieceAbstract {
4
5  Font NextPieceBox.captionFont;
6
7  pointcut paintInfoBoxes(TetrisCanvas cthis,
8  Graphics g)
9  : execution(* TetrisCanvas.paintInfoBoxes(..))
10  && this(cthis) && args(g);
11 }
```

Differently from the structure presented in Section 2, we define concrete subaspects to the feature we aim at implementing binding time as well as to the interaction between features. Therefore, Listings 14 and 15 illustrate these concrete subaspects. The `NextPieceStatic` aspect implements pieces of advice that alter the application behaviour by changing or adding feature functionality into the join points identified by the pointcuts defined in `NextPieceAbstract`, that is, *NextPiece* code without interaction. Additionally, the `MobileNextPieceStatic` aspect implements pieces of advice with the same purpose of those. However, these

refers to pointcuts that are part of the interaction between *Mobile* and *NextPiece* features. To statically bind these two features, we need to include `NextPieceAbstract`, `NextPieceStatic`, `MobileNextPieceAbstract`, `MobileNextPieceStatic` and `Mobile` aspects in the project build.

Listing 14. NextPieceStatic

```
1  aspect NextPieceStatic extends NextPieceAbstract {
2
3   after(TetrisCanvas cthis, ...)
4   : createNextPieceBoxHook(cthis, ...) {
5    cthis.nextPieceBox = new NextPieceBox(
6    TetrisConstants.COLOR_BLACK,
7    TetrisConstants.COLOR_LIGHT_GREY,
8    cthis.font, ...);
9   }
10 }
```

Listing 15. MobileNextPieceStatic

```
1  aspect MobileNextPieceStatic
2   extends MobileNextPieceAbstract {
3
4   after(TetrisCanvas cthis, Graphics g)
5   : pc_paintInfoBoxes(cthis, g) {
6    if (cthis.nextPieceBox.setPieceType(
7    cthis.game.getNextPieceType())) {
8     cthis.nextPieceBox.paint(g);
9    }
10  }
11 }
```

Implementing dynamic binding time. We implement dynamic binding for the *NextPiece* feature when the Tetris application is configured to run on *Desktop* environment.

Analogously to static binding time implementation, we define two aspects to implement dynamic binding of *NextPiece* feature. Listing 16 shows the implementation of dynamic binding time for code related exclusively to the *NextPiece* feature whereas Listing 17 shows the implementation of dynamic binding time for code related to the interaction between *NextPiece* and *Desktop*. The only difference of Listings 16 and 17 over Listings 14 and 15 is the driver implementation in Line 6. As mentioned before, this is a mechanism to activate features dynamically.

Listing 16. NextPieceDynamic

```
1  aspect NextPieceDynamic
2   extends NextPieceAbstract {
3
4   after(TetrisCanvas cthis, ...)
5   : createNextPieceBoxHook(cthis, ...) {
6    if (driverNextPiece) {
7     cthis.nextPieceBox = new NextPieceBox(
8     TetrisConstants.COLOR_BLACK,
9     TetrisConstants.COLOR_LIGHT_GREY,
10    cthis.font, ...);
11   }
12  }
13 }
```

Last but not least, even Edicts might be applied to provide flexible binding within feature interaction scenarios. However, we observe the same problems presented in Section 2, that is, code cloning, scattering, and tangling. Furthermore, we show that Pointcut Redefinition and Layered Aspects achieve better results regarding these issues for the features where interaction occurs (*NextPiece-Desktop* and *NextPiece-Mobile*) in Sections 6.2 and 6.3.

Listing 17. DesktopNextPieceDynamic

```
1  aspect DesktopNextPieceDynamic
2   extends DesktopNextPieceAbstract {
3
4   after(TetrisCanvas cthis, Graphics g)
5   : pc_paintInfoBoxes(cthis, g) {
6    if (driverNextPiece) {
7     if (cthis.nextPieceBox.setPieceType(
8     cthis.getGame().getNextPieceType())) {
9      cthis.nextPieceBox.paint(g);
10    }
11   }
12  }
13 }
```

5. STUDY SETTINGS

To evaluate the idioms showed in Sections 2 and 3, we perform an empirical assessment focusing on code reuse, maintenance and understanding. Therefore, we explain how we conduct our assessment in this section.

First, we explain the selected applications discussing general characteristics and the selected features in Section 5.1. Secondly, we discuss the Goal-Question-Metric (GQM) design [32] in Section 5.2. To guide our evaluation, we present the goals we aim to achieve, which consists of assessing idioms to implement flexible binding times for features regarding code reuse, maintenance and understanding. Additionally, we present the research questions we intend to investigate, such as which idiom helps to reduce code cloning. Further, we outline the metrics we use to answer these questions. Third, we explain the assessment procedures we follow to do this work in Section 5.3.

5.1. Selected applications

This section presents the selected applications and their features that we provide flexible binding time. We implement the four idioms discussed for every feature presented in this section.

BerkeleyDB, which is one of our selected application, was originally refactored by Kästner *et al.* [33]. Therefore, we reviewed and refactored 10 feature implementations we select of this application to comply with the way we implement the other eight selected features of the other applications. One example of this refactoring consists of separating pointcuts from advice declarations. Leaving them together would alter the evaluation and possibly introduce bias, such as increasing code cloning for the Edicts idiom because by following its structure, presented in Section 2, we would have to duplicate pointcuts and advice instead of only pieces of advice between the concrete subaspects. Besides BerkeleyDB, we consider three other applications: Tetris [34], Freemind [35] and ArgoUML [36]. They are plain object-oriented applications written in Java.

We extract the code of some of their features into aspects to create an SPL before applying the idioms. In what follows, we provide more details about these applications.

5.1.1. Tetris

Tetris [34] is an open-source implementation in Java ME (Micro Edition) of the well-known Tetris game. It is designed to run on mobile environment. First, we create a product line by extracting the code related to Java ME into aspects. Thus, we were able to code and add Java SE (Standard Edition) support as well. In this way, we can run Tetris for both platforms. Figure 7 shows the feature model of our Tetris SPL. Note that *Mobile* and *Desktop* are alternative features that represent the platforms we can run this application. Additionally, we implement flexible binding time for *Record* and *NextPiece* optional features. The circles in Fig. 8 illustrate both features in the application. The *Record* feature shows to the user what is the highest score any player has already achieved. The second feature, named *NextPiece*, shows the next piece which is about to drop on the screen. This product

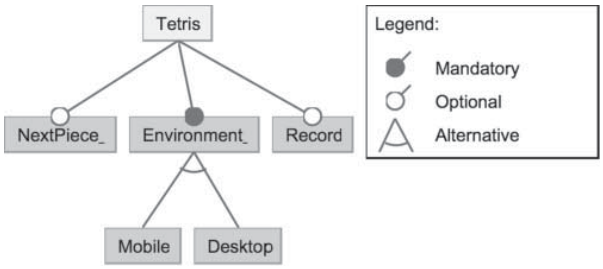

FIGURE 7. Tetris SPL feature model.

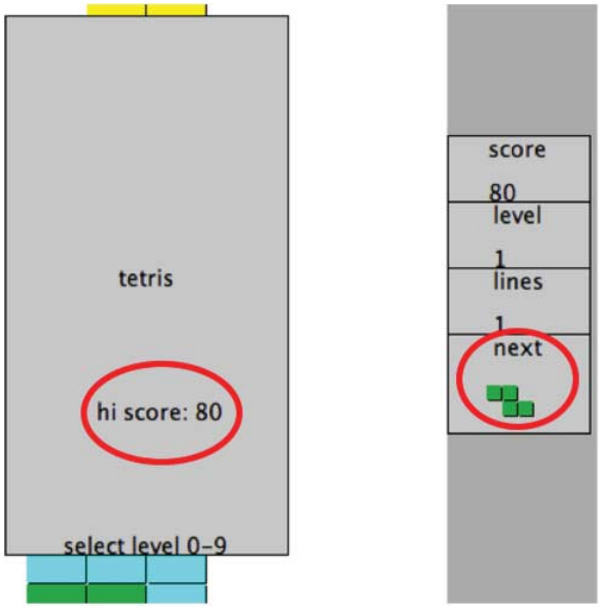

FIGURE 8. Tetris features.

line has about 1500 lines of code. The *Record* and *NextPiece* features have nearly 400 lines of code within aspects and classes. Additionally, the features represent a scenario of feature interaction, as we explained in Section 4.

For the Java SE platform, we implement a dialog box to appear in the beginning of the execution and let the user choose what features should be activated. Thus, this dialog box represents the driver mechanism in this application. It allows dynamic binding time for *NextPiece* and *Record*. On the other hand, for the Java ME platform, it is desirable to avoid overhead introduced by dynamic binding [37] due to restrictions of performance, so we provide a simple user interface without this dialog box to statically activate or deactivate the features.

Therefore, it is possible to generate different products: (i) features dynamically bound for the Java SE platform and (ii) features statically bound or (iii) unbound for the Java ME platform. The *Record* and *NextPiece* features do not present constraints regarding their composition. Therefore, we may activate or deactivate them independently. Thereby, both may be activated or deactivated, or one may be deactivated while the other is activated, or *vice versa*.

5.1.2. Freemind

Freemind [35] is an open-source system used to construct diagrams to organize ideas by using mind maps. It is written in Java and runs on the Java SE platform. We select this application because it is a widely used one and presents code related to particular functionalities scattered throughout several layers and classes. In this context, we extracted the code of two features to create a product line and provide flexible binding time. Figure 9 illustrates the feature model of our resulting SPL. The *Clouds* and *Icons* features are optional and do not present constraints, which means they may have any combination.

Figure 10 illustrates a mind map in Freemind that organizes information of our application. The circles represent the two features. The *Clouds* optional feature (right-hand side circle) may alert an important node from the mind map. For instance, we use a cloud to call our attention about what platforms Tetris runs on. The *Icons* optional feature decorates these nodes beside their names. The Tetris and Freemind nodes contain icons.

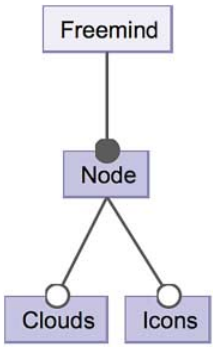

FIGURE 9. Freemind SPL feature model.

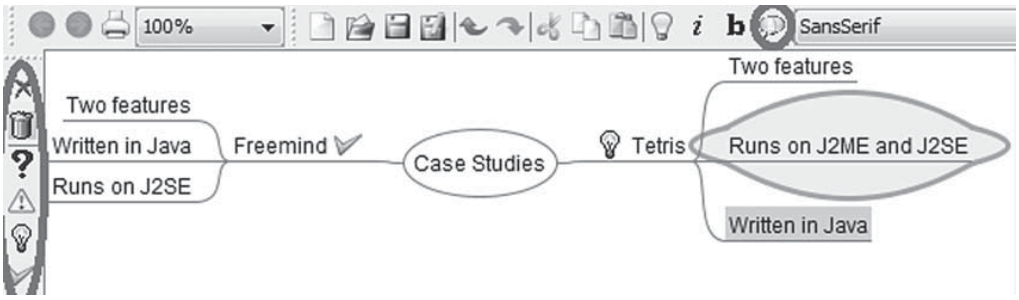

FIGURE 10. Mind map constructed in Freemind.

Differently from Tetris, *Clouds* and *Icons* are crosscutting, scattered and tangled throughout different architecture layers. We also use several different AspectJ constructs to implement these features. Albeit we consider these kinds of feature code in Freemind, we also select less scattered feature code in the other source code.

The Freemind product line has about 67 000 lines of code and both features have ∼4000 lines. For this SPL, we can generate different products with dynamic and static feature bind or static feature unbind.

5.1.3. *ArgoUML*

ArgoUML [36] is an open-source UML modelling tool written in Java that includes support for standard UML 1.4 [38] diagrams. We select this application because the its code is separated into subsystems that have different responsibilities and are organized in layers. Thus, the feature code should be scattered within this component instead of the whole application as in Freemind, which turns its feature code different from the other SPLs. In this context, we create a product line by extracting the code of two features into aspects and introducing flexible binding time for both.

For the first feature, we focus on the notation subsystem which defines the *Notation* language used in UML diagrams. ArgoUML provides two of them: *UML 1.4* and *Java*. This is an OR feature, so we may have only *UML 1.4*, only *Java* or both. We also consider the *Guillemets* optional feature. It is responsible for showing the symbols "«" "»" to accommodate the stereotypes of classes in the diagrams. Despite the simplicity, the *Guillemets* feature code is scattered throughout many modules of ArgoUML.

The ArgoUML product line has nearly 113 000 lines of code and 470 of feature code. Figure 11 shows the feature model of this product line. Note that we may have any combination between the two selected features. Therefore, we can generate different products with dynamic and static feature bind and static feature unbind, equally to the Freemind product line.

5.1.4. *BerkeleyDB*

BerkeleyDB [39] is an open-source database written entirely in Java. It uses the Java Environment advantages to simplify

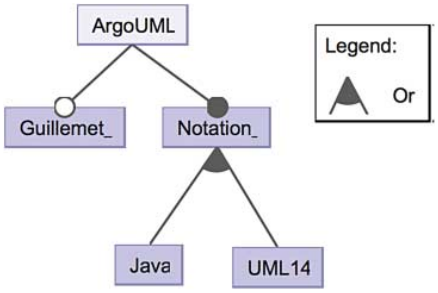

FIGURE 11. ArgoUML SPL feature model.

development and redeployment, and provides simple store key/-value pairs of arbitrary data. For this work, we use a product line version [33], which has 38 optional or alternative features. We select ten features to cover different characteristics, such as crosscutting and scattered code.

Moreover, we do not consider all the 38 features because we subjectively analysed their code and found that their code is similar. We checked the number of pointcuts and pieces of advice, which might influence metric results. Besides that, we verified what kinds of advice were implemented within feature code. They may be important to determine feature code scattering and tangling. Thus, we selected a set of features that represents different sizes and presents different kinds of advice. The 10 selected features represent different sizes and crosscutting characteristics present in BerkeleyDB. Size and crosscutting code are the main characteristics that can alter our metric results. Since these characteristics are similar among the other 28 features, our results presented in Section 6 would not change or bring new insights with respect to the selected metrics.

Table 3 explains the responsibility of each feature. The optional features have constraints between each other. For example, *Truncate* depends on *Delete* to delete the database before creating a new empty one. In other words, *Delete* must be present in the product when *Truncate* is activated. We discuss it later in Section 6.9. Due to the quantity of features, we can have several number of product configurations.

This product line has ∼32 000 lines of code. All the 10 selected features sum up ∼4000 lines of code. Figure 12 illustrates the feature model of this product line. The *IO* and *NIO* features are alternatives, which means that only one may be present at a time. The other features are optional. Differently from the other SPLs, we cannot have some combinations of features. As aforementioned, some implementation of features depend on other implementation of features to execute correctly. This limits the possible products we may generate.

TABLE 3. BerkeleyDB features.

Feature	Definition
Checksum	Checksum read and write validation of persistence subsystem.
Delete	Deletes database.
EnvironmentLock	Prevents two instances on the same database directory.
Evictor	Reduces memory consumption by evicting non-persistent nodes from the database tree.
INCompressor	Removes delete entries and empty nodes from the tree.
IO	Classic I/O implementation.
Look ahead cache	Keeps track of memory used, and when full (over budget), the node offsets should be queried and removed.
Memory budget	Calculates the available memory for the database and how to apportion it between cache and log buffers.
NIO	New I/O implementation.
Truncate	Deletes the database and creates a new one without the previous data.

5.2. Goal-Question-Metric

We use a GQM [32] design to drive the evaluation process. It is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data. The resulting measurement model includes the Goal, Question and Metric. We structure it in Table 4. We try to answer which idiom brings benefits with respect to code reuse, maintenance and understanding. To achieve that, we consider characteristics such as code cloning, scattering, tangling and size. Furthermore, most metrics we choose have already been defined and successfully used to measure quality factors in several works [40–45]. All of these metrics are defined to be used at the implementation level, which we focus on this work.

Cloning

(i) *Pairs of Cloned Code* (*PCC*). It measures the number of pairs of duplicated code based on tokens. To measure this metric, we use CCFinder [46], which is a token-based tool to detect code duplication. To be considered a pair of duplicated code, there must be at least 40 consecutive duplicated tokens, we provide further details in Section 6.1.

Scattering

(i) *Degree of Scattering across Components* (*DOSC*). It measures how distributed is a concern code across components (classes or aspects). It varies from 0 to 1. If DOSC is 0, then the code of a concern is in a single component. On the other hand, if DOSC is 1, then the code of a concern is equally divided among all considered components [16].

(ii) *Degree of Scattering across Operations* (*DOSO*). Similarly to DOSC, DOSO measures how distributed is a concern across methods and advice. It varies from 0 to 1. If DOSO is 0, then the code of a concern is in a single method or advice. On the other

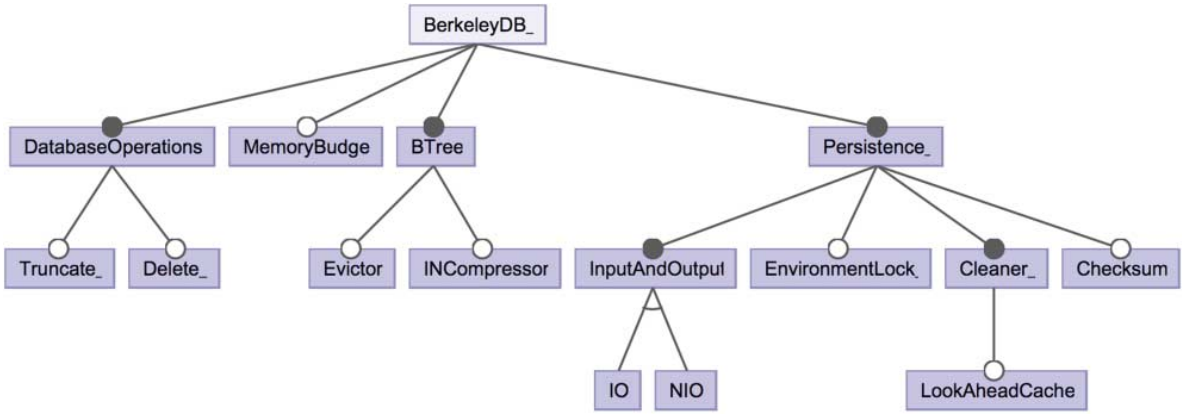

FIGURE 12. BerkeleyDB SPL feature model.

TABLE 4. GQM to assess flexible binding time implementation.

Goal	
Purpose	Evaluate idioms regarding
Issue	cloning, scattering, tangling, and size of their flexible binding time implementation
Object	for features
Viewpoint	from a software engineer viewpoint
Questions and Metrics	
Q1- Which idiom reduces the code duplication when implementing binding time flexibility?	
Pairs of Cloned Code	PCC
Q2- Which idiom reduces the driver and feature code scattering?	
Degree of Scattering across Components	DOSC
Degree of Scattering across Operations	DOSO
Concern Diffusion over Components	CDC
Q3- Which idiom reduces the tangling between the driver and feature code?	
Degree of Tangling within Components	DOTC
Degree of Tangling within Operations	DOTO
Q4- Which idiom reduces the lines of code and number of components?	
Source Lines of Code	SLOC
Vocabulary Size	VS

hand, if DOSO is 1, then the code of a concern is equally divided among all considered methods and advice [16].

(iii) *Concern Diffusion over Components* (*CDC*). Number of components that include code related to a feature plus the number of other components that access them [45].

Tangling

(i) *Degree of Tangling within Components* (*DOTC*). It measures how dedicated a component (class or aspect) is to one or more concerns under consideration. Like DOSC and DOSO, it varies from 0 to 1. If DOTC is 0, then the code of a component is totally dedicated to one concern. On the other hand, it is 1 if the code of a component is dedicated to all concerns under consideration [16].

(ii) *Degree of Tangling within Operations* (*DOTO*). It measures how dedicated a method or advice is to one or more concerns under consideration. It varies from 0 to 1. If DOTO is 0, then the code of a method or advice is totally dedicated to one concern. On the other hand, it is 1 if the code of a method or advice is dedicated to all concerns under consideration [16].

Size

(i) *Source Lines of Code* (*SLOC*). Number of source lines of a component (e.g. classes or aspects).

(ii) *Vocabulary Size* (*VS*). Number of program components (e.g. classes or aspects);

We use PCC in Section 6.1 to answer Question 1, as it may indicate a design that could increase maintenance costs [47] because a change would have to be replicated to the duplicated code as well. To answer Question 2, we use CDC, DOSC and DOSO in Section 6.2 to measure the implementation scattering for each idiom. Thus, we measure code scattering through different perspectives, such as number of components in which the code is scattered, and how dedicated the code is with respect to components or methods. To answer Question 3, we measure the tangling between driver and feature code considering the DOTC and DOTO metrics in Section 6.3. Hence, our evaluation considers code tangling within components and methods. Additionally, SLOC and VS are well-known metrics for quantifying a module size and complexity. So we answer Question 4 measuring the size of each idiom in terms of lines of code and number of components in Section 6.4. We provide more details about how DOSC, DOSO, DOTC and DOTO are defined in Appendix A. However, these details are not crucial to understand our assessment.

5.3. Assessment procedures

To obtain results for the metrics and perform our assessment, we follow the following procedure. We detail it in five steps, which we follow in the order presented.

(1) *Application selection.* In order to generalize the results of our study to other contexts, we consider 18 features selected from the 46 total features contained in the 4 product lines discussed in Section 5.1.

four different applications and 18 different features.

However, we do not choose these source code arbitrarily. First, BerkeleyDB has been minutely studied before [33]. Thus, we could check what code changes we should make in an existing source code (including aspects) in order to apply flexible feature binding. Furthermore, we select Tetris because it is a simple and easy to understand code as well as it includes feature interaction and it runs over different platforms, which makes it diverse from the others. In contrast to Tetris, Freemind and ArgoUML represent widely used software and a large source code. We select them because their feature code presents interesting characteristics. Freemind and ArgoUML features are highly tangled and scattered throughout the code, which leads to a challenging scenario when extracting their code into aspects. Additionally, the selected features have different granularity and complexity. Moreover, they have different types regarding feature model, such as optional, alternative, OR and mandatory [2].

(2) *Feature code identification and assignment.* After choosing the feature, we apply Prune Dependency rules [48] to identify scattered feature code throughout the application. These rules state that '*a program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned from the application*'. By following these rules, two different people can identify the same code related to a given concern. We choose this rule to reduce introducing bias while identifying feature code. We use comments to manually assign the programme elements related to each feature. In this way, two different researchers could identify and assign our feature code.

(3) *Feature code extraction.* For this phase, we extract the assigned feature code from the classes to aspects, except for BerkeleyDB, which its feature code was extracted before. Thus, we aim at separating the feature and core code. This allows the creation of product lines from the applications. After this phase, the application should be independent from the feature code. For example, the application may compile and execute properly without the optional feature code.

(4) *Flexible binding time implementation.* We then apply the idiom for the selected features that we wish to provide binding time flexibility. We use the chosen idiom to control whether the feature code is executed. In this work, we duplicate the SPLs in order to apply each idiom for each feature. However, in practice, a developer only needs to choose one idiom to provide flexible feature binding. The same authors (first and second) were in charge of feature code extraction and flexible binding time implementation.

(5) *Evaluate idioms.* In this phase, we use the GQM design to drive our evaluation of the four idioms. Our goal is to assess the implementation of the idioms in the selected applications. We elaborate four questions with respect to points that we want to investigate about the idioms. Then, we answer the questions by analysing the measures obtained for the selected metrics. To collect the required information to compute these metrics, we use the Metrics tool [49], which measures the number of lines of code of components, such as classes and aspects. Additionally, to compute some of the selected metrics, we need to know the number of lines of code of the feature and the driver. To do that, we manually count the corresponding lines. This information and the formulas of the metrics are stored in a sheet, which we used to obtain our results [17]. Moreover, we investigate possible changes concerning the idiom's behaviour. For example, we seek changes in the behaviour of certain feature code with respect to its implementation using Edicts and Layered Aspects. We provide details in Section 6.5.

6. RESULTS

In this section, we discuss the results regarding code quality metrics, such as code cloning, scattering, tangling and size. Therefore, we assess the implementations of the four idioms (Edicts and the three we define) to confirm that we are able to mitigate the problems with Edicts that we have identified. We answer each of the questions outlined in Section 5.2, so that we answer each question in the following subsections. In some of the subsequent discussion, we provide the metric results for only some of features, when they are demonstrably sufficient to drive our explanation. However, the complete material produced in this work is available on our website [17]. We provide an association of features and the corresponding application in Table 5. Furthermore, we do not present results from the Flexible Deployment idiom for the Tetris product line, because CaesarJ does not support some AOP constructs needed to extract the code of Tetris' features. Further details are provided in Section 6.9. In an effort to confirm that there is no difference in the execution of a given feature code implemented by

TABLE 5. Features and application.

Application	Features
Freemind	Icons and clouds
ArgoUML	Notation and guillemets
Tetris	NextPiece and Record
BerkeleyDB	EnvironmentLock, Checksum, Truncate, Delete, LookAheadCache, Evictor, NIO, MemoryBudget, IO and INCompressor

different idioms, we seek to determine whether the execution behaviour changes between the four idioms. To do so, we use the SafeRefactor tool [15] to compare the behaviour of the same features implemented with two different idioms in Section 6.5. Moreover, Sections 6.6–6.8 discuss some threats to validity. Finally, we discuss the advantages and disadvantages of our idioms in Section 6.9.

6.1. Cloning

To answer Question 1 and to determine which idiom best reduces code cloning, we use the CCFinder [46] tool to obtain the results for the PCC metric. CCFinder is a widely used tool to detect cloned code, and several researchers have used it for this purpose [50–54]. We stipulated 40 as the minimum clone length (in tokens). That is, to be considered a clone, two pairs of code must have at least 40 equal tokens. Thus, pairs of similar code that have, say, 39 tokens are not considered cloned, and they are disregarded by CCFinder. We decided to disregard results with fewer than 40 similar tokens owing to the frequent appearance of pseudo-clones, such as package names. On the other hand, we did not apply a higher minimum clone length because, in doing so, we would omit some relevant PCC. We used a token-based detection method because representing a source code as a token-sequence facilitates the detection of clones with different line structures, which cannot be detected by a line-by-line algorithm [55]. This allows us to detect more clones in our product lines. In addition, we stipulated 12 as the token set size (TKS). At 12, the TKS will not consider a cloned-code fragment as a simple statement, such as a series of variable declarations, which is often present in classes and irrelevant in most cases. On the other hand, the TKS is not so high as to omit some interesting duplications in the code. Table 6 summarizes the results of the PCC metric.

As expected, Edicts had a higher PCC rate. It frequently duplicated feature code in concrete sub-aspects. Particularly, Edicts contained more cloned code in large features that define many pieces of advice, such as the *Memory Budget* and *Icons* features. However, for features such as *Evictor, NIO* and *IO*, Edicts did not result in cloning, because of limitations to the CCFinder tool—about which further details are provided in Section 6.7.

On the other hand, Pointcut Redefinition significantly decreased the PCC rate for some features, such as *Clouds, Delete* and *Memory Budget*. Moreover, it eliminated code cloning altogether in *EnvironmentLock* and *LookAheadCache*. The PCC rate was still high for the *Memory Budget* feature, however, because it is constituted by a relatively large code containing many pointcuts. Hence, following the Pointcut Redefinition idiom design, these pointcuts were redefined in the concrete sub-aspects, and, consequently, their signatures were cloned.

In addition, Layered Aspects decreased the PCC rate even more for *Delete*, *Memory Budget* and *INCompressor*. This happened because not all pointcuts needed to be redefined, unlike with Pointcut Redefinition. However, for *Memory Budget*, which presents some pieces of `around` advice, Layered Aspects must still redefine pointcuts related to the `around` advice, as explained in Section 3.2.

TABLE 6. PCC metric results.

	Edicts	PR	LA	FD
Icons	19	4	4	4
Clouds	5	1	1	1
Notation	9	9	9	8
Guillemets	4	0	0	0
NextPiece-Desktop	0	0	0	–
NextPiece-Mobile	0	0	0	–
Record-Desktop	0	0	0	–
Record-Mobile	0	0	0	–
EnvironmentLock	2	0	0	3
Checksum	5	0	0	0
Truncate	2	2	2	5
Delete	9	3	2	0
LookAheadCache	2	0	0	0
Evictor	0	0	0	0
NIO	0	0	0	0
MemoryBudget	34	17	7	1
IO	0	0	0	0
INCompressor	2	2	1	1

To summarize these results, the Flexible Deployment idiom drastically reduced the PCC rate for the *Memory Budget* feature. As explained in Section 3.3, Flexible Deployment uses the dynamic deployment of aspects to implement a dynamic binding time, such that it is unnecessary to redefine pointcuts or duplicate feature code between the concrete aspects. Nonetheless, some interesting code cloning is apparent. The PCC rate for *EnvironmentLock* and *Truncate* is worse than it is in other idioms, owing to the need for wrapper classes, which sometimes have long names. These are examples of code cloning that, despite being identified by CCFinder, is irrelevant, as we explain in Section 6.7.

Furthermore, the *NextPiece* and *Record* features, which fall within the context of feature interaction, do not contain clones because they have only static binding times in the mobile version, and only dynamic binding in the desktop version. Thus, the code differs depending on the platform. In addition, *Evictor*, *NIO* and IO are especially small features with few tokens in their program elements. Whereas we know that Edicts clones at least the advice code between the concrete aspects, the CCFinder tool was not able to detect it. Moreover, the *Notation* feature is a subsystem that activates and deactivates its code in a manner similar to the four idioms.

We conclude that Edicts may be harmful in terms of code maintenance. The reason for drawing this conclusion is that

Edicts clones all the advice within a feature implementation [56]. Therefore, in maintaining the application code, a programmer will need to alter the same code for static and dynamic binding-time implementations. Although our idioms clone less code, they still duplicate some, and this may be harmful to code maintenance in specific cases. Nevertheless, we contend that Edicts is the worst idiom insofar as it has the highest cloning rate overall. Whereas Kapser and Godfrey [19] state that code cloning is not always harmful, we did not detect any similarities in the type the code duplication nor patterns of non-harmful code replication. Therefore, the code duplication we uncovered may be harmful in terms of code reuse and understanding.

6.2. Scattering

As mentioned in Section 5.2, we used DOSO, DOSC and CDC to analyse feature and driver-code scattering for each idiom. Feature code and driver code represent different concerns, so we analysed them separately. By using these metrics, we pursued an answer to Question 2 in Section 5.2. In the following sections, we show the results from only some features, an abridgement that we believe is sufficient for the purposes of our discussion. However, we have provided the full results in our online appendix [17].

6.2.1. Driver

We turn now to a discussion of driver-code scattering. Figure 13 presents the results from applying the DOSO metric. Edicts and Pointcut Redefinition reported the most scattering because they scatter driver code throughout many program elements. For example, the Edicts idiom introduces driver code into several pieces of advice. Moreover, Pointcut Redefinition introduces driver code into the redefined pointcuts in the concrete sub-aspect. In contrast, Layered Aspects increases the DOSO results only in cases where pointcuts related to the around advice are redefined (see Section 3.2). Accordingly, Flexible Deployment had less driver scattering, owing to the dynamic deployment of aspects that implement the driver mechanism in only one pointcut and advice procedure. The *Truncate* feature reported a DOSO value of zero, because it does not implement advice. The *Notation* feature had an equivalent result in all the idioms because its implementation is similarly independent of the idiom. This happens because the *Notation* feature consists of a subsystem in the architecture of ArgoUML. Hence, this feature is not scattered throughout the source code and we only need to deactivate the subsystem. The *IO* and *NIO* features have only one piece of advice, resulting in a DOSO of zero for them as well, since there is only one `if` statement for each piece of advice. We should also observe that *NextPiece-Desktop* and *NextPiece-Mobile* have identical results, owing to feature interaction. Indeed, as revealed by the source code for our Tetris application [17], applying driver code to interactive features is no different from applying it to features that do not interact.

To assess scattering from another perspective, we also considered the DOSC metric. This allows us to identify cases where code scattering occurs at a class or aspect level. Although driver-code scattering is common when considering pointcuts and advice, it is unusual in classes or aspects, because some features are small and do not require more than a single aspect for their implementation. Therefore, the driver code is implemented only in a concrete sub-aspect. Figure 14 shows the results from applying the DOSC metric. The *Delete*, *Memory Budget* and *INCompressor* features use more than one aspect to implement the feature code. Therefore, their respective DOSC results will vary when the driver is also implemented in more than one aspect, because we must implement static and dynamic binding in concrete sub-aspects defined for each abstract aspect. This is especially detrimental for Edicts and Pointcut Redefinition, because driver code is introduced in the concrete sub-aspects to implement a dynamic binding time. On the other hand, the DOSC is zero for the other features because

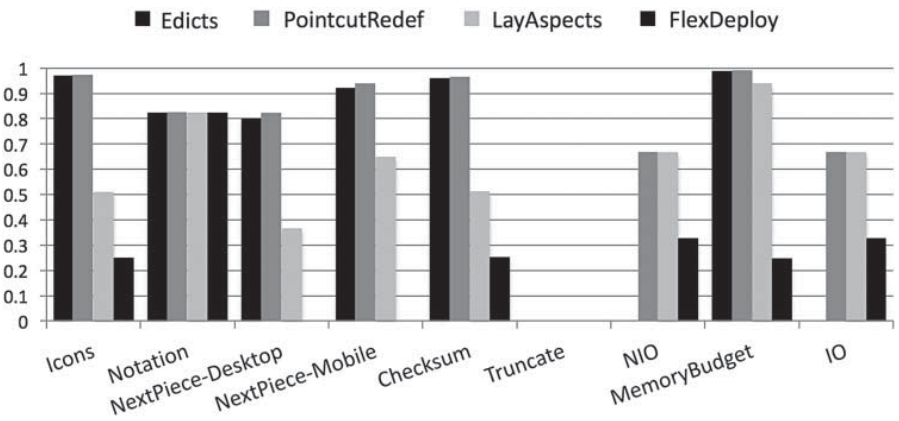

FIGURE 13. DOSO Driver metric results.

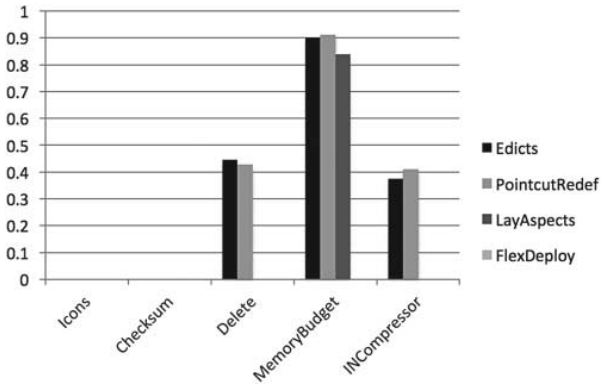

FIGURE 14. DOSC Driver metric results.

the driver code is implemented in only one aspect. Therefore, it is not scattered throughout the components that implement the feature code.

In this context, we can observe that DOSO and DOSC draw different conclusions. Scattering at the level of operations depends on the number of elements, such as the number of pointcuts or pieces of advice. Therefore, on account of their design, as explained in Sections 2 and 3, Edicts and Pointcut Redefinition are generally worse at the level of operations. On the other hand, scattering at the component level depends on the number of elements such as classes and aspects. In this way, only feature implementations that contain more than one abstract aspect lead to code scattering. This is why *Icons* resulted in driver-code scattering at the level of operations (with the driver code in more than one operation), whereas it is absent at the component level (with feature code in more than one aspect).

Finally, it is important to observe that driver-code scattering is potentially harmful to code maintenance. Indeed, because the implementation of Edicts and Pointcut Redefinition leads to a higher DOSO and DOSC for the driver code, there is a potential need to change the same concern code throughout several elements. For Layered Aspects, only one aspect must be addressed, whereas for Flexible Deployment, one aspect and exactly two elements must be addressed. Thus, maintaining the driver code should be easier with the latter two idioms. Because Edicts and Pointcut Redefinition frequently scatters the driver code, it is difficult with both idioms to reuse a driver implementation. However, Layered Aspects and Flexible Deployment permit for easier driver-code reuse, because the corresponding code is localized in one component and few elements.

6.2.2. *Feature*

Next, we discuss feature-code scattering. The DOSC metric depends on the number of components related to feature implementation. Therefore, we consider scattering from two perspectives. From a package perspective, the feature code is well localized, owing to its implementation in a single package. Therefore, there is no scattering at the package level. From the perspective of components (i.e. classes or aspects), a single feature implementation may employ multiple aspects, leading to a scattering of the feature code. Deciding the optimal number of aspects to implement feature code is an engineering matter [8]. However, we used an equivalent number of aspects for a given feature. Thus, all of the flexible binding time implementations for the four idioms are consistent. The number of aspects may vary as a result of the idiom's particularities, but not as a result of the feature implementation.

According to the results shown in Fig. 15, Edicts scatters the feature code because its design leads to the implementation of feature code in the abstract aspect as well as in the concrete sub-aspects. In contrast, the other idioms scatter feature code only when multiple aspects are used in their respective implementations, as *Memory Budget*. For these features, we use more aspects to implement their code because they are large, so each aspect is responsible for a particular feature concern. In other cases, Layered Aspects, Pointcut Redefinition and Flexible Deployment resulted in low DOSC results. This was expected, because the concrete empty aspect is used to allow for the feature code's instantiation, as explained in Section 3.1.

In addition, we applied another metric to measure the scattering, to analyse it from a different perspective. The CDC metric allows us to identify potential differences between the idioms (Fig. 16). Differences between the idioms appear when we use more than one aspect to implement the feature code. Notice that Edicts and Pointcut Redefinition have the worst results, owing to their design. These idioms define two concrete sub-aspects to implement static and dynamic binding for each abstract aspect. On the other hand, Layered Aspects and Flexible Deployment implementations contain fewer components because there is no need to define the concrete sub-aspects for each abstract aspect.

Finally, DOSC and CDC show that our idioms generally reduce feature-code scattering relative to the Edicts idiom. The CDC metric complements DOSC because it shows that our idioms also reduce feature code scattering when we use multiple aspects to implement the feature, as demonstrated by the *Memory Budget* feature. In addition, and analogous to driver-code scattering, the Edicts idiom contains more crosscutting feature code, which may hampers code reuse [57].

6.3. Tangling

This section answers Question 3 by investigating the extent of tangling existing between the feature and driver code. According to the principle of Separation of Concerns [58], one should be able to implement and reason about each concern independently.

In this work, we assume that *the greater is the tangling between the feature code and its driver code, and the worse is the separation of those concerns*. As mentioned in Section 5.2,

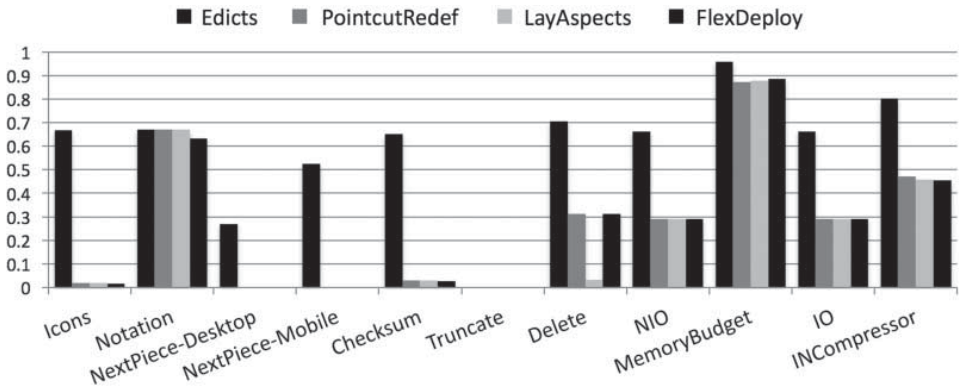

FIGURE 15. DOSC feature metric results.

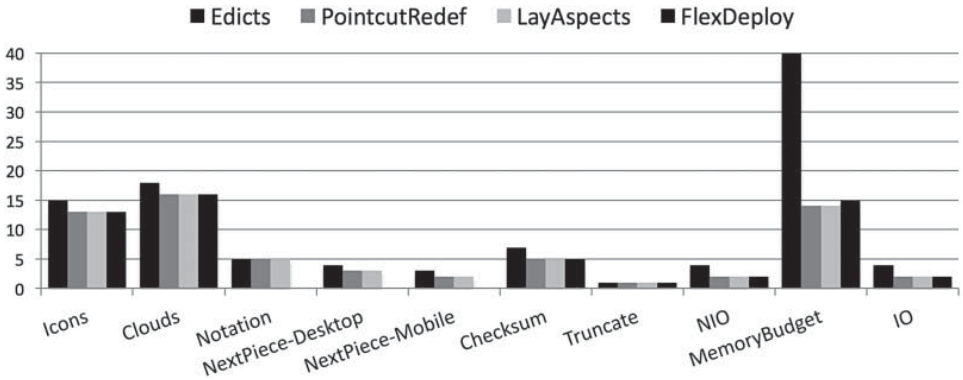

FIGURE 16. CDC metric results.

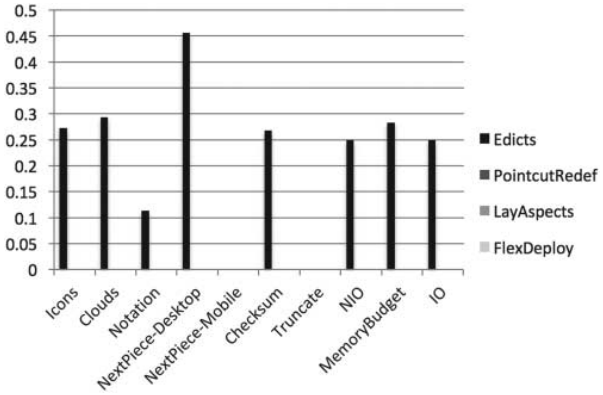

FIGURE 17. DOTO metric results.

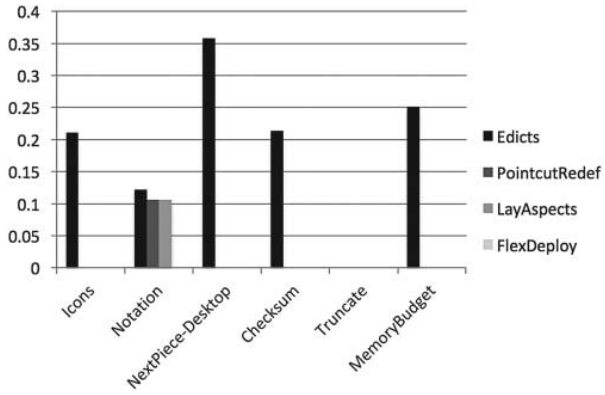

FIGURE 18. DOTC metric results.

we measure the DOTO and DOTC. Figures 17 and 18 show the DOTO and DOTC results, respectively.

Edicts has a high DOTO rate because we implemented the driver code for each piece of advice. As a consequence, the feature and driver code are tangled in the concrete sub-aspect that implements a dynamic binding time. The other three idioms do not tangle the driver and feature code, because the abstract aspect contains only feature code and the concrete

sub-aspect that implements dynamic binding contains only the driver implementation. Hence, there is no advice or pointcut with the feature code defined in the Pointcut Redefinition, Layered Aspects or Flexible Deployment implementations that are associated with the driver code.

The feature interaction presented in features *NextPiece-Desktop* and *NextPiece-Mobile* influences the DOTO results. As seen in Fig. 17 the former feature shows a high degree of tangling (DOT), whereas the latter does not. This happens because *NextPiece-Desktop* interacts with *Desktop*, which demands dynamic feature binding, as explained in Sections 4 and 5.1.1. Therefore, it must tangle the feature and driver code with the Edicts idiom. On the other hand, *NextPiece-Mobile* interacts with the *Mobile* feature, which demands only static feature binding. Thus, it does not require a driver. Furthermore, the*NextPiece-Desktop* feature resulted in the highest DOTO because its implementation contains only advice and, following Edicts' design, all of these pieces of advice are tangled with the driver code. As we explained in Section 2, the driver code is equally distributed among the pieces of advice. That is, they contain the same `if` statement, resulting in each piece of advice having the same quantity of driver code. On the contrary, features that present a low DOTO have few pieces of advice that contain feature and driver code compared to the total number of operations in this feature implementation.

Similarly, Edicts' implementation tangles the driver and feature code with respect to the aspects, as illustrated in Fig. 18. At least one concrete sub-aspect resulted in code tangling, owing to Edicts' structure. The situation is more egregious for feature implementations that use more than one aspect, such as *Memory Budget*. On the other hand, the other three idioms implement a separated aspect that contains only the driver mechanism. Therefore, for them, the DOTC is zero, except in the case of the *Notation* feature, which presents a similar implementation for all idioms, as we explained in Section 6.2. Notice that DOTO and DOTC are zero for *NextPiece* and *Record* because we implemented them to run on mobile environments, which binds the feature statically. As a result, there is no driver code. In addition, the DOTO and DOTC are also zero for the *Truncate* feature, because it does not implement pieces of advice, and consequently there is driver code for it either.

In this context, using Edicts for implementations with a flexible binding time is more difficult to maintain and reuse than using other idioms. Because driver and feature codes are highly tangled, and because they both represent different concerns, it would be difficult and error-prone to modify the driver, for instance. Another example of this difficulty is found in implementing new pieces of advice, owing to new requirement or change requests and neglecting to introduce an `if` statement containing the driver code. This risks hindering the execution of the feature code.

TABLE 7. SLOC metric results.

	Edicts	*PR*	*LA*	*FD*
Icons	2198	2180	2127	2031
Clouds	2015	1919	1897	1833
Notation	172	172	172	153
Guillemets	320	208	181	178
NextPiece-Desktop	516	515	506	–
NextPiece-Mobile	450	449	435	–
Record-Desktop	511	507	387	–
Record-Mobile	460	335	455	–
EnvironmentLock	160	119	116	118
Checksum	476	469	441	456
Truncate	157	157	157	178
Delete	460	358	338	358
LookAheadCache	140	146	134	134
Evictor	452	466	460	470
NIO	61	52	52	52
MemoryBudget	1891	1397	1397	1285
IO	74	66	66	62
INCompressor	592	584	584	570

6.4. Size

To answer the fourth question, we must identify the idiom that most efficiently reduces the size of its implementation, which is related to the size of each idiom in terms of the lines of code and the number of components. For this purpose, we used the SLOC and VS metrics.

As explained in Section 2, to support the flexible binding time of a feature, the Edicts idiom introduces two additional concrete sub-aspects for each aspect implementing the feature code. This situation leads to higher SLOC rates, mainly because of code duplication introduced by dynamic and static aspects. Table 7 presents the SLOC metric results. In general, the three idioms we defined achieved better results than Edicts, on account of the relative reduction in feature and driver code duplications. The difference is more conspicuous when features are large and contains many pieces of advice that are duplicated in the concrete sub-aspects. For example, the SLOC for Edicts is high for the *Memory Budget* feature compared to the other idioms, because it is large and contains many pieces of advice. In contrast, the *Truncate* feature does not contain any advice and, consequently, the Edicts idiom does not duplicate them, resulting in a relatively lower SLOC for Edicts with respect to this feature.

We analysed the implementation size by using the VS metric from the perspective of the number of components. The results are tabulated in Table 8. Unlike in the other graphs, there is a discernible difference between the product lines as a result of applying this metric. Most SPLs use a comparable number of components to implement features. However, differences occur because the Flexible Deployment implementation contains

TABLE 8. VS metric results.

	Edicts	*PR*	*LA*	*FD*
Freemind	550	551	549	557
ArgoUML	1622	1621	1621	1629
Tetris	21	21	21	–
Berkeley	345	338	345	350

wrapper classes (Section 3.3.2), resulting in several more components than the other idioms for Freemind, ArgoUML and BerkeleyDB. The Tetris product line has fewer aspects to the implementation of its features, explaining how the VS results are equivalent for Edicts, Pointcut Redefinition and Layered Aspects.

Finally, because the Edicts implementation is larger in size on account of the code duplication, its code is more difficult to maintain and understand. The increase in the SLOC forces a programmer to review more lines of code in order to perform changes during maintenance. In this context, we observe that maintaining an Edicts implementation is difficult owing to its size. However, the evolution of selected applications must be investigated before we can safely conclude that this observation is correct.

6.5. Behaviour

To reliably ensure that the execution of an implementation with a flexible binding time does not change when implementing the same feature code with different idioms, we used the SafeRefactor [15] tool. This tool uses static analysis to identify common methods in two SPLs, and it generates unit tests for the identified methods using Randoop [59]. Subsequently, SafeRefactor runs the generated test suite for both SPL versions in order to identify inconsistencies between the results from executing the two test suites.

In this context, we provided two different versions of the same application as inputs for SafeRefactor. These versions differed in the idioms used to implement a flexible binding time for the features of the application. For **example**, we provided one version of the BerkeleyDB product line with Edicts, and another version using Layered Aspects. SafeRefactor then generated and executed a test suite to determine whether there were behavioural changes in the two versions. Table 9 summarizes our results. Note that SafeRefactor generated several unit tests for each SPL. In addition, it generated the same number of tests for the source and target projects. For example, it generated 2005 tests for Freemind implemented with Edicts (i.e. the source) and 2005 tests for Freemind implemented with Pointcut Redefinition (i.e. the target). Then, SafeRefactor looks for differences between the results of the two sets of test executions. The results are equivalent when no behavioural changes were found.

TABLE 9. Number of unit tests generated by SafeRefactor.

	Edicts and Pointcut Redefinition	Edicts and Layered Aspects	Pointcut Redefinition and Layered Aspects
Freemind	2005	2072	2072
ArgoUML	1443	1443	1443
Tetris	2394	2380	2384
BerkeleyDB	3362	3141	3141

We did not find behavioural changes in the Freemind and Tetris product lines. However, SafeRefactor found some behavioural changes in the BerkeleyDB product line. Based on these results, we were able to determine that we accidentally removed a line of code within a constructor defined in the `FileManager` class of the base code. Thereby, we could fix this inconsistency prior to the execution of our assessment.

Unfortunately, SafeRefactor does not support the CaesarJ language. Hence, we could not test the Flexible Deployment implementation. However, we have not detected behavioural changes when using the applications implemented with this idiom.

6.6. Threats to internal validity

Threats to the internal validity concern the fact that the assessment affects the results [60]. Therefore, in our work, these threats suggest the introduction of bias resulting from the selection of favourable procedures, such as the manner of feature code assignment. In addition, we can expose decisions that risk introducing errors to our work and the means for circumventing them.

BerkeleyDB refactoring. Unlike Tetris, Freemind and ArgoUML, the feature code for our BerkeleyDB was previously extracted into aspects by Kästner *et al.* [33]. However, their extraction differs from how we extracted the other features. For example, their resulting code uses anonymous pointcuts that are defined where they are used as part of the advice [24]. Because we used named pointcuts [24] in our feature code, we refactored Kästner *et al.*'s code by changing the anonymous pointcuts to named pointcuts. That is, we separated pointcuts from advice declarations. Therefore, we avoid bias in our metric results, because we can count the number of source code lines that contains pointcuts in the same way we did for the other named pointcuts. Otherwise, features containing anonymous pointcuts would present less feature code.

Therefore, the code of BerkeleyDB product line's features comply with the other feature implementations. Additionally, all the refactoring is performed within the existing aspects. We do not re-assign feature code in the base code.

Feature code identification and assignment unreliability. We cannot ensure that our extraction of selected features does not introduce any bias, because the task of identifying and

assigning feature code is in a certain way subjective. This may be a hindrance to researchers who try to replicate our work. Indeed, there could be discord in the same feature code assigned by different researchers [61].

However, we have tried to minimize such unreliability in two ways. First, we used the Prune Dependency Rules [48] to identify and assign the feature code. These rules define some of the procedures that the researcher should follow in order to avoid introducing bias to the resulting extracted feature code, as we explain in Section 5.3. Secondly, only two researchers (*viz.* the first and second authors) identified and assigned the implementation of the selected features. Most activities were performed in pairs. We believe that our work is more reliable as a result of restricting the number of people and encouraging communication between them.

Behavioural differences between flexible binding time implementations. Our flexible binding time implementations might differ in their behaviour. We cannot guarantee that refactoring the base code to extract the feature code and refactoring upon implementing the four idioms will not result in nominal differences.

Nevertheless, we used the SafeRefactor tool [15] to create tests to validate the implementation of the idioms. As explained in Section 6.5, after applying this tool, we could fix an inconsistency between two idioms. Furthermore, in using all four applications before and after implementing a flexible binding time, we do not observe any differences in their behaviour.

Data-collection errors. We did not employ a tool or any automated means for collecting the data to compute our metrics. Rather, we prepared a digital sheet that contained the formulae of the metrics we selected to perform our assessment. This document included templates of these formulae, so that we could input the data required by a given metric corresponding to the feature implementation we were evaluating. Because this process is subject to errors resulting from the momentary inattention of the person performing the evaluation, our metric results may include unnoticed errors.

Nevertheless, we performed a pair review regarding the results of our metrics. That is, one researcher collected the data for a given metric, while another researcher sought inconsistencies to the metric's results and the source code where the data was obtained. Alas, we did discover some errors, such as an incorrect number of source code lines required as input for the calculus of a given metric. These errors were accordingly revised and corrected.

6.7. Threats to construct validity

Threats to construct validity cover issues related to the design of the assessment and its capacity to answer the research questions [60]. These threats concern the limitations of our cloning results and metrics that we did not use in our work.

Limitations to the cloning results. In collecting the PCC metric results, we did not use a manual filter to eliminate the cloned code that the CCFinder [46] tool detected. Nevertheless, such a filter would not have resulted in interesting or pertinent results. At most, we might have detected some uninteresting clones, such as method overloading. Because of this limitation, the CCFinder's results did not include some interesting code cloning that we might have observed by reading the code, especially for the Edicts idiom. However, this does not change our conclusion concerning the harmful effects of the Edicts idiom regarding code duplication.

To compensate for this issue, we analysed the source code of each idiom. The CCFinder tool reveals the classes and aspects in the code where duplication is present. Thus, besides gathering the metric rates, we manually compared the idioms by looking at the cloned code from their respective classes and aspects. Indeed, it was apparent that Edicts presented more cloned code than the other idioms.

Metrics limitations. We did not consider metrics such as code coupling and cohesion. These metrics could help to reveal opportunities for improving the quality of our flexible binding time implementations. We could compute the former by applying the coupling between components metric [62], which counts the number of classes or aspects declaring methods, and the advice, constructors, or fields that can be called or accessed by another class or aspect. Moreover, we could compute the latter by applying the lack of cohesion over operations [63] metric, which counts the number of operation pairs (i.e. methods or advice) working on fields with different classes and aspects (non-statically) minus the operation pairs working on common fields.

However, we gathered our conclusions based on a set of metrics that considered eight indices with respect to cloned code, scattering, tangling and size. In addition, no previous work in the literature has evaluated the implementation of a flexible binding time regarding these issues.

External attributes. We discussed code reuse, maintenance and understanding, based on our metrics, knowledge and intuition. Because internal attributes, such as code tangling and scattering measurements, are insufficient for empirically tracing external attributes, we can only conjecture about increases or decreases in these quality factors when considering idioms and their implementations. Nevertheless, we could explore these internal and external attributes by generating new research questions, for instance.

6.8. Threats to external validity

Threats to the external validity concern a generalization of the results [60]. Thus, we must discuss how our selected SPLs might be generalized, and we limit our results to applications that consider only one driver mechanism.

Limitations to the selected SPLs. To perform our assessment, we selected systems that we transformed into SPLs. They were written in Java, and variability was implemented using AOP. Therefore, we cannot generalize the results presented

in this work for distinct contexts such as other programming paradigms or languages. For example, we would be able to reduce the DOSO for the Layered Aspects idiom if AspectJ supported `proceed` outside advice, because we would not need to redefine the pointcuts in order to implement the bind time for `around` advice, as explained in Section 3.2.1. Moreover, the results from the token-based cloning metric are tied in a certain way to the language in which the SPLs were implemented (*viz.* Java). Indeed, we could alter the results by changing the language, which might require a different number of tokens to implement the same program elements.

Nevertheless, the combination of Java and AspectJ can be used together in SPLs to reinforce the significance of our findings. The results are likely to apply to other SPLs that conform to the technologies we considered. On the other hand, we must emphasize that the collected results are specific to the type of technology we used. We have no evidence that the results may be generalized to other contexts.

Finally, we selected a subset of 10 features out of the existing 38 in BerkeleyDB. We attempted to select a representative subset of features, so that the results would not be repetitious. However, our choice was subjective; we based it on the code's characteristics. Consequently, we may have missed some interesting features.

6.9. Discussion

In this section, we qualitatively discuss the idioms. The Edicts idiom may experience problems in several cases. Edicts clones the feature code because it duplicates advice in the sub-aspects of its structure. Further, it scatters and tangles driver code throughout the advice. Besides that, its implementation size tends to be larger, just because it clones parts of the feature code. These problems are detrimental to software maintenance. For instance, if programmers forget to introduce the driver mechanism to a piece of advice, a runtime exception is possible when the feature is deactivated. Maintaining the feature code will also be time consuming and error-prone, owing to the code cloning and scattering. For example, the same problem must be twice-repaired in two concrete sub-aspects.

In contrast, Pointcut Redefinition does not clone the feature code, because it is localized in the abstract aspect. Therefore, it is not duplicated in the sub-aspects. However, it does scatter the driver code throughout the pointcuts that are redefined in the concrete sub-aspects. This can be harmful when adding, updating, or removing the driver mechanism. If the feature implementation has several pointcuts, the driver code must be changed at several pointcuts. Nevertheless, Pointcut Redefinition does not tangle the feature and driver code, because the concrete sub-aspect that implements the driver mechanism does not contain the feature code.

The Layered Aspects idiom does not clone the feature code either. Unlike Pointcut Redefinition, however, this idiom only scatters the driver code when `around` advice is present. Thus, we mitigated the aforementioned problem regarding changes to the driver mechanism. However, Layered Aspects does not resolve this issue completely, because it must scatter driver code throughout the redefined pointcuts. Yet, this idiom does not tangle the feature and driver code, because neither of the concrete sub-aspects contain any feature code. The feature code is implemented only in the abstract aspect, which does not contain the driver code.

To summarize, Flexible Deployment does not clone, scatter or tangle the feature and driver code. Nevertheless, CaesarJ has some disadvantages, insofar as it does not support some AOP constructs we use to extract feature code. For example, we faced problems implementing two features from Tetris because a class that is inherited from another may vary between desktop and mobile platforms. Because CaesarJ does not provide mechanisms to define the inheritance of classes, such as the `declare parents` statement in AspectJ, we could not implement these features without introducing bias to the evaluation, as a result of numerous changes to Tetris. Moreover, CaesarJ does not support privileged access for non-public members. Thus, we must change the visibility of some methods in order to access them from within the CaesarJ virtual classes. On the other hand, the CaesarJ language has the advantage of supporting dynamic virtual-class deployment, which we used to implement dynamic feature binding in a concise and localized manner, as we explained in Section 3.3. Furthermore, we developed Flexible Deployment with one of the authors of CaesarJ [23]. Thus, we could again collaborate to enhance the language by identifying some of its deficiencies.

The implementations of a flexible binding time using the four idioms was performed by the same person to avoid introducing a bias. Consequently, the feature code was extracted into aspects in the same way, and each idiom was applied following its design for the features of the selected applications.

Nevertheless, our idioms may suffer some deficiencies. For example, the activation or deactivation of a feature may require activating or deactivation another, owing to the optional feature problem [64]. For dynamic binding, if we do not validate feature compositions when applying changes, runtime errors are possible, compromising the programme's execution. In this work, we did not validate the composition of features dynamically. Therefore, our idioms may reveal deficiencies when (de)activating features at runtime. This is a possibility in the BerkeleyDB product line. For example, the activation of the *Truncate* feature implies the activation of the *Delete* feature. Hence, if one of our idioms dynamically deactivates *Delete*, the *Truncate* feature will not work properly. We plan to circumvent this issue in future work.

Moreover, previous work has suggested that the execution of systems for static binding is more efficient than dynamic binding [37, 65]. Our work does not include an evaluation of performance and memory consumption. Thus, it is unclear whether our idioms improve or deteriorate the performance of feature-code executions. Nevertheless, we did not use

techniques or constructs that might harm performance or memory consumption [8].

Finally, we used only aspect-oriented-based idioms to implement a flexible binding time for features in applications written in Java. Thus, some disadvantages to our idioms may appear as a result of these technologies. For example, in Section 3.2.1, we explain that AspectJ does not provide access to the `proceed` join-point of the advice. This is a disadvantage, because we must deactivate each piece of advice one-by-one, rather than all at once. Therefore, our results might be exclusive to an AOP context or to our idiom's implementation, as we discuss in Section 6.8.

7. RELATED WORK

Besides Edicts, we point out other work regarding flexible binding times as well as studies that relate aspects and product line features. Additionally, we discuss how our work differs from them. In Section 7.1, we present related work that provides only dynamic binding time. Therefore, their main difference to our work is that our approach provides both static and dynamic binding time. On the other hand, we discuss related work concerning flexible binding time in Section 7.2, which includes approaches that support static and dynamic binding time.

7.1. Dynamic binding time

Rosenmüller *et al.* [66] propose an approach for statically generating SPLs to support dynamic feature binding time. 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 code base and evaluate it with concern to reconfiguration

In contrast, we saw opportunities to enhance a previous work, as explained in Section 2. Thus, we opted to try to fix weaknesses of Edicts by following the same approach, which is based on idioms instead of generative or other related mechanism.

Lee *et al.* [67] 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. 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. The authors use a home service robot product line as an example of their dynamically reconfigurable product. Nevertheless, differently of us, they do not evaluate their approach considering code quality.

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 [68]. They apply their approach to generate an industrial real-time television SPL.

This work provides only dynamic binding time whereas we support flexible binding time. It aims at adapting systems to changing requirements. Thereby, they do not consider the performance overhead that dynamic binding should introduce [37]. Further, there is no evaluation of their approach.

Dinkelaker *et al.* [5] 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.* [69] propose OARTA, which is a declarative extension to the AspectBench Compiler [27], which allows dynamic weaving of aspects. OARTA extends the AspectJ language syntax so that a developer can name a pointcut, 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. However, they do not perform an assessment considering quality of code factors of their approach.

Oliveira *et al.* introduce a modular mechanism to dynamic feature binding by means of composition of Object Algebras [70]. The Object Algebras are composed dynamically. They relay on feature-oriented programming [71]. Another recent work proposes dynamic feature binding in the context of Delta-oriented programming [72]. Damiani *et al.* provide dynamic composition of features using a reconfiguration automaton to specify which configurations are safe. Their environment automatically checks whether a reconfiguration is safe. In [73], they provide a formal foundation for dynamic DOP and a type system to ensure the safety of dynamic reconfiguration.

7.2. Flexible binding time

Rosenmüller *et al.* [37] present an approach that supports static and dynamic composition of features from a single base code. They provide an infrastructure for dynamic instantiation and validation of SPLs. Their approach is based on FOP [71] whereas our work uses AOP. Further, they use an extension called FeatureC++ [74] to automate dynamic instantiation of SPLs.

The usage of C++ as a client language introduces some specific problems. Static constructs when using dynamic composition, virtual classes, semantic differences when comparing static and dynamic compositions are examples of such problems

[37]. Despite of our work uses only Java as a client language, we did not observe these problems in our implementations. Furthermore, the authors only evaluate their approach regarding performance, applicability and memory consumption. In our work, we focus on a different evaluation regarding code cloning, scattering, tangling and size, as showed in Section 6.

An alternative proposal considers *conditional compilation* as a technique to implement features with flexible binding times [6]. This work discusses how to apply *conditional compilation* in real applications like operating systems. Likewise 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 binding time. Hence, for complex variation points, the situation becomes even worse.

Another proposal to implement flexible binding time into features, which is also our previous work, considers aspect inheritance [18]. 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. Differently from Edicts, Aspect Inheritance avoids feature code duplication. Despite of the fact that this solution enhances some weaknesses found in Edicts, it is worst than the three idioms presented in this work. Thereby, presenting this idiom again would turn this work unnecessarily repetitious.

Additional AspectJ idioms for flexible binding time appear in recent work [75]. There we compare the additional new idioms with Layered Aspects following a similar assessment method. Here we compare Layered Aspects and other idioms with the Edicts idiom, bringing evidence that we created better idioms with respect to code quality factors. Furthermore, our assessment considers the Tetris product line, which is also developed for *Mobile* applications. This allowed us to apply and evaluate the idioms in a non-*Desktop* environment. Moreover, our evaluation here considers more metrics (DOSO, DOTO and CDC), so that we could analyse the binding time implementations in the operation level rather than only in class and aspects level. Besides that, we use the SafeRefator tool [15] to bring evidence that the binding time implementations do not interfere in the execution behaviour. In this way, both are complementary in the sense that they contribute to the body of evidence on idioms for flexible binding time.

8. CONCLUSION

In this work, we discussed the implementation of a flexible binding time for features. We introduced three idioms designed to address the shortcomings in an idiom called Edicts. Our idioms mitigated some issues, such as code cloning, scattering and tangling.

Our first idiom, Pointcut Redefinition, uses the inheritance of AspectJ aspects and a redefinition of pointcuts to implement a flexible binding time. We extracted feature code for an abstract aspect and implemented a static and dynamic binding time for different concrete sub-aspects. The sub-aspect that implements static binding is empty and inherits the abstract aspect in order to prompt its instantiation. The sub-aspect that implements dynamic binding redefines the pointcuts that were defined in the abstract aspect and associates them with the driver mechanism. This idiom mitigates code cloning and tangling, issues that are beset by Edicts. However, it does not significantly reduce driver-code scattering, because the pointcuts must be redefined one-by-one and associated with the driver mechanism.

The second idiom, Layered Aspects, is based on AspectJ. We extracted the feature code for an abstract aspect and implemented static binding in the same manner as we did with Pointcut Redefinition. To implement a dynamic binding time, we defined a concrete sub-aspect that uses AspectJ `adviceexecution` in association with a redefinition of the pointcuts. Layered Aspects reduces code cloning, scattering, tangling and size, when compared with Edicts

The third idiom, Flexible Deployment, uses a dynamic deployment mechanism provided by CaesarJ to implement binding-time flexibility. We extracted feature code for a CaesarJ class and implemented static and dynamic binding times in different CaesarJ classes. To implement static binding, we defined an empty CaesarJ subclass to allow for the instantiation of the class that contains the feature code. To implement dynamic binding, we defined a CaesarJ class that contains one pointcut and one advice to implement the dynamic deployment mechanism. We could not apply Flexible Deployment to 4 of our 18 features, because the idiom does not support some AOP constructs. Nevertheless, Flexible Deployment exhibited satisfactory results when subjected to the metrics It reduces code cloning, scattering and tangling, when compared with Edicts.

This work evaluated these idioms by means of metrics. To achieve representative results, we used the idioms to implement a flexible binding time for 18 features in 4 different product lines. Our evaluation is based on the GQM design. Thus, we defined the goals, questions and metrics to assess the idioms for each of the 18 features. We also discussed qualitative aspects, such as the advantages and disadvantages of each idiom.

In future work, we shall consider safe dynamic composition of features. Our applications may present some problems when activating or deactivating features. For example, *NIO* and *IO* from BerkeleyDB are alternating features. Thus, if *NIO* is already activated when we activate *IO*, BerkeleyDB will not work correctly because only one of the features can be activated at a given time. Hence, we intend to support safe dynamic composition of features. We will consider two alternatives to resolve this issue. The first is to attempt the integration of the domain-specific language proposed by Dinkelaker *et al.* [5]

discussed in Section 7. The second is to create a new technique for our idioms. We will explore both alternatives to decide which one fits better in our context.

Besides safe dynamic composition of features, we also intend to adopt auxiliary tools to support feature extraction such as CIDE [76]. It would be interesting to compare this extraction with ours, based on the prune-dependency rule.

Another potential project involves comparing static and dynamic binding times in terms of performance and memory consumption, considering different idioms. To do so, we can analyse the consumption of working memory by observing the size of particular instances of aspects that implement features and binding times. Furthermore, we can compare static and dynamic binding times, and the respective idioms in terms of performance. The results could lead to new conclusions about which idioms to adopt. Thus, our evaluation would be more complete insofar as it not only considers code quality, but also code execution.

Moreover, we intend to define an idiom that is not based on AOP. We could address problems inherent to such a paradigm, such as the impossibility of accessing the `proceed` join-point in a generic way, as discussed in Section 3.2.1. Moreover, our product lines were written in Java, and we have not considered other languages. Hence, we might also implement a new idiom based on feature-oriented programming and C++, similar to Rosenmüller *et al.* do in their work [66]. Because related work has not been evaluated with respect to the metrics we used, we may achieve some interesting results.

Finally, we can add other metrics to our suite. Sant'Anna *et al.* define the Coupling Between Components [77] metric for AOP. It could be used to count the number of classes or aspects declaring methods, constructors, or fields that can be called or accessed by another aspect or class. Likewise, Garcia *et al.* [45] define a metric suite that can be pertinent in our context. We have already adopted some of the same metrics. Yet, we intend to apply others: weighted operations per component will offer an enhanced study of the source-code size; concern diffusion over operations and concern diffusion over SLOC can help to evaluate code scattering; the lack of cohesion over operations metric [63] can determine which idiom has the lowest code cohesion. Consequently, we can better identify difficulties in terms of the reuse, maintenance, and understanding of the code. Moreover, Lopez-Herrejon *et al.* [78] define some interesting metrics that we can use in further evaluations, such as the feature crosscutting degree, which counts the number of classes that are crosscut by a feature.

ACKNOWLEDGEMENTS

We would like to thank colleagues of the Software Productivity Group (SPG)[1] for several discussions that helped to improve the work reported here.

[1] http://www.cin.ufpe.br/spg

FUNDING

This work was supported by Centro Nacional de Desenvolvimento Científico e Tecnológico [grant numbers 141590/2013-0 to R.A., 560256/2010-8 and 484860/2011-9], Fundação de Amparo à Ciência e Tecnologia do Estado de Pernambuco [grant number IBPG-0573-1.03/09 to R.A.], Coordenação de Aperfeiçoamento de Pessoal de Nıivel Superior [grant number 347/10] and the Instituto Nacional de Ciência e Tecnologia para Engenharia de Software [grant numbers 573964/2008-4 and APQ-1037-1.03/08].

REFERENCES

[1] Pohl, K., Bockle, G. and van der Linden, F.J. (2005) *Software Product Line Engineering*. Springer, Berlin.
[2] CMU/SEI-90-TR-021 (1990) Feature-oriented domain analysis (FODA) feasibility study. Technical Report. Software Engineering Institute, Pittsburgh, USA.
[3] Rosenmüller, M., Siegmund, N., Apel, S. and Saake, G. (2011) Flexible feature binding in software product lines. *Automat. Softw. Eng.*, **18**, 163–197.
[4] Gomaa, H. and Hussein, M. (2004) Dynamic Software Reconfiguration in Software Product Families. *Proc. Int. Workshop on Product Family Engineering*, Siena, Italy, November 4–6, pp. 435–444. Springer, Berlin.
[5] Dinkelaker, T., Mitschke, R., Fetzer, K. and Mezini, M. (2010) A Dynamic Software Product Line Approach using Aspect Models at Runtime. *Proc. Workshop on Composition and Variability*, Rennes and Saint Malo, France, March 15–19.
[6] Dolstra, E., Florijn, G. and Visser, E. (2003) Timeline Variability: The Variability of Binding Time of Variation Points. *Proc. Workshop on Software Variability Management*, Portland, USA, May 3–10.
[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. (1997) Aspect–Oriented Programming. *Proc. European Conf. Object–Oriented Programming*, Jyväskylä, Finland, June 9–13, pp. 220–242. Springer, Berlin.
[8] Chakravarthy, V., Regehr, J. and Eide, E. (2008) Edicts: Implementing Features with Flexible Binding Times. *Proc. Int. Conf. Aspect-Oriented Software Development*, Brussels, Belgium, April 1–4, pp. 108–119. ACM, New York.
[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. (2001) Getting started with AspectJ. *Commun. ACM*, **44**, 59–65.
[10] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston.
[11] Göde, N. and Koschke, R. (2009) Incremental Clone Detection. *Proc. European Conf. Software Maintenance and Reengineering*, Kaiserslautern, Germany, March 24–27, pp. 219–228. IEEE Computer Society, Washington.
[12] Lieberherr, K., Lorenz, D.H. and Ovlinger, J. (2003) Aspectual collaborations: Combining modules and aspects. *Comput. J.*, **46**, 542–565.
[13] Andrade, R., Ribeiro, M., Gasiunas, V., Satabin, L., Rebelo, H. and Borba, P. (2011) Assessing Idioms for Implementing

Features with Flexible Binding Times. *Proc. European Conf. Software Maintenance and Reengineering*, Oldenburg, Germany, March 1–4, pp. 231–240. IEEE Computer Society, Washington.

[14] Liu, J., Batory, D.S. and Nedunuri, S. (2005) Modeling Interactions in Feature Oriented Software Designs. *Feature Interactions in Telecommunications and Software Systems*, Leicester, UK, June 28–30, pp. 178–197. IOS Press, Washington.

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

[16] Eaddy, M. (2008) An empirical assessment of the crosscutting concern problem. PhD Thesis at Graduate School of Arts and Sciences, Columbia University, New York.

[17] Andrade, R., Ribeiro, M., Rebêlo, H., Gasiunas, V., Satabin, L. and Borba, P. (2013) Flexible binding time. http://tinyurl.com/kgleczd (accessed July 22, 2015).

[18] Ribeiro, M., Cardoso, R., Borba, P., Bonifácio, R. and Rebêlo, H. (2009) Does AspectJ Provide Modularity When Implementing Features with Flexible Binding Times? *Latin American Workshop on Aspect-Oriented Software Development*, Fortaleza, Brazil, October 4–5.

[19] Kapser, C.J. and Godfrey, M.W. (2008) Cloning considered harmful considered harmful: patterns of cloning in software. *Empirical Softw. Eng.*, **13**, 645–692.

[20] Krinke, J. (2001) Identifying Similar Code with Program Dependence Graphs. *Proc. 8th Working Conf. Reverse Engineering*, Stuttgart, Germany, 2–5 October, pp. 301–310. IEEE Computer Society, Washington.

[21] Ducasse, S., Rieger, M. and Demeyer, S. (1999) A Language Independent Approach for Detecting Duplicated Code. *Proc. IEEE Int. Conf. Software Maintenance*, Florence, Italy, November 6–10, pp. 109–118. IEEE Computer Society, Washington.

[22] Clarke, S., Harrison, W.H., Ossher, H. and Tarr, P.L. (1999) Separating Concerns Throughout the Development Lifecycle. *Proc. Workshop on Object-Oriented Technology*, Lisbon, Portugal, June 14–18, pp. 299–302. Springer, Berlin.

[23] Aracic, I., Gasiunas, V., Mezini, M. and Ostermann, K. (2006) An overview of caesarj. *Trans. Aspect-Oriented Softw. Dev. I*, **3880**, 135–173.

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

[25] Bodden, E., Forster, F. and Steimann, F. (2006) Avoiding Infinite Recursion with Stratified Aspects. *Proc. of the NODe–Objects, Aspects, Services, the Web*, Erfurt, Germany, September 18–21, pp. 49–64. GI LNI Lecture Notes in Informatics, Bonn Germany.

[26] Aracic, I., Gasiunas, V., Klose, K. and Bartolomei, T.T. (2013) Caesarj project. http://caesarj.org/ (accessed July 22, 2015).

[27] Avgustinov, P. *et al.* (2005) ABC: An Extensible AspectJ Compiler. *Proc. Int. Conf. Aspect-Oriented Software Development*, Chicago, USA, March 14–18, pp. 87–98. ACM, New York.

[28] Jackson, M. and Zave, P. (1998) Distributed feature composition: a virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.*, **24**, 831–847.

[29] Gibson, J.P. (1997) *Feature Requirements Models: Understanding Interactions*. IOS Press, Canada.

[30] Calder, M., Kolberg, M., Magill, E.H. and Reiff-Marganiec, S. (2003) Feature interaction: a critical review and considered forecast. *Comput. Netw.: Int. J. Comput. Telecommun. Netw.*, **41**, 115–141.

[31] Sobernig, S. (2010) Feature Interaction Networks. *Proc. Symp. Applied Computing*, Sierre, Switzerland, March 22–26.

[32] Basili, V., Caldiera, G. and Rombach, D.H. (1994) The goal question metric approach. In Marciniak, J.J. (ed.), *Encyclopedia of Software Engineering*, pp. 528–532. Wiley, NJ.

[33] Kästner, C., Apel, S. and Batory, D. (2007) A Case Study Implementing Features using AspectJ. *Proc. 11th Int. Software Product Line Conf.*, Kyoto, Japan, September 10–14, pp. 223–232. IEEE Computer Society, Washington.

[34] Kiang, J. (2013) Tetris. http://tinyurl.com/yepgrg3 (accessed July 22, 2015).

[35] Müller, J., Polansky, D., Novak, P., Foltin, C. and Polivaev, D. (2013) Free mind mapping software. http://tinyurl.com/5qrd5 (accessed June 20, 2015).

[36] 2013) Argouml. http://argouml.tigris.org/ (accessed July 22, 2015).

[37] Rosenmüller, M., Siegmund, N., Saake, G. and Apel, S. (2008) Code Generation to Support Static and Dynamic Composition of Software Product Lines. *Proc. Int. Conf. Generative Programming and Component Engineering*, Nashville, TN, USA, October 19–23, pp. 3–12. ACM, New York.

[38] Booch, G. (2005) *The unified modeling language user guide*. Pearson Education, India.

[39] Oracle (2013). Berkeley. http://tinyurl.com/26fr5a (accessed July 22, 2015).

[40] Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N. and Aho, A.V. (2008) Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, **34**, 497–515.

[41] Bonifácio, R. and Borba, P. (2009) Modeling Scenario Variability as Crosscutting Mechanisms. *Proc. Int. Conf. Aspect-Oriented Software Development*, Charlottesville, USA, March 2–6, pp. 125–136. ACM, New York.

[42] Apel, S. (2007) The role of features and aspects in software development. PhD thesis at School of Computer Science, University of Magdeburg. Magdeburg.

[43] Figueiredo, E., Garcia, A., Sant'anna, C., Kulesza, U. and Lucena, C. (2005) Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. *Proc. Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Glasgow, UK, July 25–29.

[44] Basili, V.R., Briand, L.C. and Melo, W.L. (1996) A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, **22**, 751–761.

[45] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C. and von Staa, A. (2005) Modularizing Design Patterns with Aspects: A Quantitative Study. *Proc. Int. Conf. Aspect-Oriented Software Development*, Chicago, USA, March 14–18, pp. 36–74. ACM, New York.

[46] Kamiya, T., Ohata, F., Kondou, K., Kusumoto, S. and Inoue, K. (2013) CCfinder Official Site. http://www.ccfinder.net/ (accessed July 22, 2015).

[47] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L. (1998) Clone Detection using Abstract Syntax Trees. *Proc. Int. Conf. Software Maintenance*, Bethesda, USA, November 16–20, pp. 368–377. IEEE Computer Society, Washington.

[48] Eaddy, M., Aho, A. and Murphy, G.C. (2007) Identifying, Assigning, and Quantifying Crosscutting Concerns. *Proc. Int. Workshop on Assessment of Contemporary Modularization Techniques*, Minneapolis, USA, May 20–26, pp. 2–7. IEEE Computer Society, Washington.

[49] Sauer, F. (2013) Metrics tool. http://metrics.sourceforge.net/ (accessed June 10, 2015).

[50] School, D.R. and Rajapakse, D.C. (2005) An Investigation of Cloning in Web Applications. *Proc. Special Interest Tracks and Posters of the Int. Conf. World Wide Web*, Sydney, Australia, July 27–29, pp. 252–262. Springer, Berlin.

[51] Kapser, C.J. and Godfrey, M.W. (2006) Supporting the analysis of clones in software systems: a case study. *J. Softw. Maintenance Evol.: Res. Practice*, **18**, 61–82.

[52] Bruntink, M., van Deursen, A., van Engelen, R. and Tourwe, T. (2005) On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, **31**, 804–818.

[53] Al-Ekram, R., Kapser, C., Holt, R. and Godfrey, M. (2005) Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems. *Proc. Int. Symp. Empirical Software Engineering*, Noosa Heads, Australia, 17–18 November.

[54] Deissenboeck, F., Hummel, B., Jürgens, E., Schätz, B., Wagner, S., Girard, J.-F. and Teuchert, S. (2008) Clone Detection in Automotive Model-Based Development. *Proc. Int. Conf. Software Engineering*, Leipzig, Germany, May 10–18, pp. 603–612. ACM, New York.

[55] Kamiya, T., Kusumoto, S. and Inoue, K. (2002) Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, **28**, 654–670.

[56] Bruntink, M., van Deursen, A., Tourwe, T. and van Engelen, R. (2004) An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. *Proc. Int. Conf. Software Maintenance*, Chicago, USA, September 11–17, pp. 200–209. IEEE Computer Society, Washington.

[57] Soule, P. (2010) *Autonomics Development: A Domain-Specific Aspect Language Approach*. Birkhäuser, Basel.

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

[59] Pacheco, C., Lahiri, S., Ernst, M. and Ball, T. (2007) Feedback-Directed Random Test Generation. *Proc. Int. Conf. Software Engineering*, Minneapolis, USA, May 19–27, pp. 75–84. ACM, New York.

[60] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B. and Wesslén, A. (2000) *Experimentation in Software Engineering: An Introduction*. Kluwer Academic, Boston.

[61] Lai, A. and Murphy, G.C. (1999) The Structure of Features in Java Code: an Exploratory Investigation. *Workshop on Multidimensional Separation of Concerns*, Denver, USA, November 1–5.

[62] Chidamber, S.R. and Kemerer, C.F. (1994) A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, **20**, 476–493.

[63] Ceccato, M. and Tonella, P. (2004) Measuring the Effects of Software Aspectization. *Proc. Workshop on Aspect Reverse Engineering*, Delft, The Netherlands, November 8–12.

[64] Kästner, C., Apel, S., ur, Rahman, Batory, D. and Saake, G. (2009) On the Impact of the Optional Feature Problem: Analysis and Case Studies. *Proc. Int. Software Product Line Conf.*, San Francisco, USA, August 24–28, pp. 181–190. Carnegie Mellon University, Pittsburgh.

[65] Günther, S. and Sunkle, S. (2010) Dynamically Adaptable Software Product Lines using Ruby Metaprogramming. *Proc. Int. Workshop on Feature-Oriented Software Development*, Eindhoven, The Netherlands, October 10–23, pp. 80–87. ACM, New York.

[66] Rosenmüller, M., Siegmund, N., Pukall, M. and Apel, S. (2011) Tailoring Dynamic Software Product Lines. *Proc. Int. Conf. Generative Programming and Component Engineering*, Portland, USA, October 22–23, pp. 3–12. ACM, New York.

[67] Lee, J. and Kang, K.C. (2006) A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. *Proc. Int. Software Product Line Conf.*, Baltimore, USA, August 21–24, pp. 131–140. IEEE Computer Society, Washington.

[68] Trinidad, P., Cortés, A.R. and Benavides, D. (2007) Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. *Proc. Int. Softw. Product Line Conf.*, Kyoto, Japan, September 10–14, pp. 51–56. Kindai Kagaku, Tokyo.

[69] Marot, A. and Wuyts, R. (2010) Composing Aspects with Aspects. *Proc. Int. Conf. Aspect-Oriented Software Development*, Rennes and Saint-Malo, France, March 15–19, pp. 157–168. ACM, New York.

[70] Oliveira, B., van der, Storm and Cook, W. (2013) Feature-Oriented Programming with Object Algebras. *Proc. European Conf. Object–Oriented Programming*, Montpellier, France, July 1–5, pp. 27–51. Springer, Berlin.

[71] Prehofer, C. (1997) Feature-Oriented Programming: A Fresh Look at Objects. *Proc. European Conf. Object-Oriented Programming*, Jyväskylä, Finland, June 9–13, pp. 419–443. Springer, Berlin.

[72] Damiani, F. and Schaefer, I. (2011) Dynamic Delta-Oriented Programming. *Proc. Int. Software Product Line Conf., Volume 2*, Munich, Germany, August 21–26, pp. 1–8. ACM, New York.

[73] Damiani, F., Padovani, L. and Schaefer, I. (2012) A Formal Foundation for Dynamic Delta-Oriented Software Product Lines. *Proc. Int. Conf. Generative Programming and Component Engineering*, Dresden, Germany, September 26–27, pp. 1–10. ACM, New York.

[74] Apel, S., Leich, T., Rosenmüller, M. and Saake, G. (2005) FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. *Proc. Int. Conf. Generative Programming and Component Engineering*, Tallinn, Estonia, September 29–1, pp. 125–140. Springer, Berlin.

[75] Andrade, R., Rebêlo, H., Ribeiro, M. and Borba, P. (2013) AspectJ-Based Idioms for Flexible Feature Binding. *Proc. of the Brazilian Symp. Software Components, Architectures, and Reuse*, Brasilia, Brazil, October 1–4, pp. 59–68. IEEE, San Francisco.

[76] Kästner, C., Apel, S. and Kuhlemann, M. (2008) Granularity in Software Product Lines. *Proc. Int. Conf. Software Engineering*, Leipzig, Germany, May 10–18, pp. 311–320. Springer, Berlin.

[77] Sant'anna, C., Garcia, A., Chavez, C., Lucena, C. and von Staa, A.v. (2003) On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. *Proc. of the Brazilian*

Symp. Software Engineering, Manaus, Brazil, October 6–10, pp. 19–34. ACM, New York.

[78] Lopez-Herrejon, R. and Apel, S. (2007) Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. *Proc. of Fundamental Approaches to Software Engineering*, Braga, Portugal, May and April 24–1, pp. 423–437. Springer, Berlin.

[79] Wong, W.E., Gokhale, S.S. and Horgan, J.R. (2000) Quantifying the closeness between program components and features. *Journal of Systems and Software*, **54**, 87–98.

APPENDIX A

METRICS DEFINITION

In this appendix, we present the definition of DOSC [16], DOSO [16], DOTC [16] and DOTO [16] metrics. In addition, we provide each metric's formulas. Due to their simplicity, we omit CDC, SLOC and VS details. Additionally, we also omit PCC details since we use the CCFinder tool to obtain its results, as explained in Section 6.1.

A.1. DOSC and DOSO

Concentration (CONC) measures how many of the source lines related to a feature s are contained within a specific component t [79], which in our case are classes, aspects or operations.

$$CONC(s,t) = \frac{\text{SLOCs in component } t \text{ related to feature } s}{\text{SLOCs related to feature } s} \tag{A.1}$$

The degree of scattering (DOS) is a measure of the variance of the concentration of a feature over all components with respect to the worst case (i.e. when the feature code is equally scattered across all components) [48]. We measure DOSC considering s as classes and aspects, whereas DOSO is measured selecting s as pieces of advice and methods.

$$DOS(s) = 1 - \frac{|T| \sum_t^T (CONC(s,t) - 1/|T|)^2}{|T| - 1} \tag{A.2}$$

where T is the set of components and $|T| > 1$.

A.2. DOTC and DOTO

Dedication (DEDI) measures how many of the source lines contained within a component t are related to concern s [79].

$$DEDI(s,t) = \frac{\text{SLOCs in component } t \text{ related to feature } s}{\text{SLOCs in component } t} \tag{A.3}$$

The DOT is a measure of the variance of the tangling of a component to every feature with respect to the worst case (i.e. when the component is equally tangled to all features) [48]. We measure DOTC considering s as classes and aspects, whereas DOTO is measured selecting s as pieces of advice and methods.

$$DOT(t) = \frac{|S| \sum_s^S (DEDI(s,t) - 1/|S|)^2}{|S| - 1} \tag{A.4}$$

where S is the set of features and $|S| > 1$.