Science of Computer Programming 96 (2014) 230–253

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines

Felipe Nunes Gaia [a,*], Gabriel Coutinho Sousa Ferreira [a,*], Eduardo Figueiredo [b,*], Marcelo de Almeida Maia [a,*]

[a] Federal University of Uberlândia, Brazil
[b] Federal University of Minas Gerais, Brazil

HIGHLIGHTS

- Variability mechanisms are systematically evaluated in the evolution of SPLs.
- FOP and AFM have shown better adherence to the Open-Closed Principle than CC.
- When crosscutting concerns are present, AFM are recommended over FOP.
- Refactoring at component level has important impact in AFM and FOP.
- CC compilation should be avoided when modular design is an important requirement.

ARTICLE INFO

Article history:
Received 1 April 2013
Received in revised form 28 February 2014
Accepted 10 March 2014
Available online 31 March 2014

Keywords:
Software product lines
Feature-oriented programming
Aspect-oriented programming
Aspectual feature modules
Variability mechanisms

ABSTRACT

Feature-Oriented Programming (FOP) and Aspect-Oriented Programming (AOP) are programming techniques based on composition mechanisms, called refinements and aspects, respectively. These techniques are assumed to be good variability mechanisms for implementing Software Product Lines (SPLs). Aspectual Feature Modules (AFM) is an approach that combines advantages of feature modules and aspects to increase concern modularity. Some guidelines on how to integrate these techniques have been established in some studies, but these studies do not focus the analysis on how effectively AFM can preserve the modularity and stability facilitating SPL evolution. The main purpose of this paper is to investigate whether the simultaneous use of aspects and features through the AFM approach facilitates the evolution of SPLs. The quantitative data were collected from two SPLs developed using four different variability mechanisms: (1) feature modules, aspects and aspects refinements of AFM, (2) aspects of aspect-oriented programming (AOP), (3) feature modules of feature-oriented programming (FOP), and (4) conditional compilation (CC) with object-oriented programming. Metrics for change propagation and modularity were calculated and the results support the benefits of the AFM option in a context where the product line has been evolved with addition or modification of crosscutting concerns. However a drawback of this approach is that refactoring components' design requires a higher degree of modifications to the SPL structure.

* Corresponding authors.
E-mail addresses: felipegaia@mestrado.ufu.br (F.N. Gaia), gabriel@mestrado.ufu.br (G.C.S. Ferreira), figueiredo@dcc.ufmg.br (E. Figueiredo), marcmaia@facom.ufu.br (M.A. Maia).

http://dx.doi.org/10.1016/j.scico.2014.03.006
0167-6423/

1. Introduction

Software Product Lines (SPLs) refer to an emerging engineering technique that aims to provide the systematic reuse of common core and shared modules in several software products [12]. Optional features define points of variability and their role is to permit the differentiation of products in a specific software product line (SPL). SPLs products share the same application domain and have points of variability among them. The adoption of SPLs presents as potential benefits the increased product's quality and development productivity, which are achieved through the systematic reuse of features in different products [12].

During an SPL life cycle, change requests are not only inevitable, but also highly frequent [22] since they target several different products. These change requests must be accommodated since they include demands from multiple stakeholders [16].

Variability mechanisms play a crucial role when considering evolving SPLs. They must guarantee the architecture stability and, at the same time, facilitate future changes in the SPL. Therefore, variability mechanisms should not degenerate modularity and should minimize the need of future changes. Ideally, all evolutionary tasks in an SPL should be conducted through non-intrusive and self-contained changes that favor insertions and do not require deep modifications into existing components. The inefficacy of variability mechanisms to accommodate changes might lead to several undesirable consequences related to the product line stability, including invasive wide changes, significant ripple effects, artificial dependencies between core and optional features, and the lack of independence of optional code [17,32].

In our previous study [15], we analyzed and compared variability mechanisms to evolve SPLs, using FOP, Design Patterns and Conditional Compilation. The evaluation was also based on change propagation measures and modularity metrics. In that work, the result was mostly favorable for FOP mechanism. It is important to consider that crosscutting concerns were not considered in the subject system analyzed in that study. This work has as main goal the better understanding of how contemporary variability mechanisms contribute to the mentioned SPLs evolution practices. To this aim, this paper presents two case studies that evaluate comparatively four mechanisms for implementing variability during the evolution of product lines: conditional compilation (CC), feature-oriented programming (FOP), aspect-oriented programming (AOP), and aspectual feature modules (AFM). This work is an extension of a previous work [19], which was carried out only with one SPL, called WebStore. In this work, we include five releases of another SPL called MobileMedia [17,45]. This SPL is larger than WebStore not only in terms of number of components but also with respect to the variety of change scenarios. Therefore, this new case study helped us to (i) increase the results reliability, (ii) come up with new findings, and (iii) reduce threats to study validity. Moreover, we extended significantly the amount of data, providing *quantitative* and *qualitative* analysis of the measured data in greater depth. The quantitative analysis refers to interpretation of collected measures related to *stability* and *modularity*. The analysis of stability considers measures of change impact [45], while the analysis of modularity relies on Separation of Concern metrics [39]. The qualitative analysis is concerned with the interpretation and reasoning of the possible factors that influenced the quantitative results.

The analysis presents novel results that support the benefits of choosing between AFM and FOP when an SPL has many optional features. In this case, class refinements adhere more closely to the Open-Closed principle [33]. In addition, these mechanisms cope well for features with no shared code and facilitate the instantiation of different products. However, FOP is not suitable for crosscutting concerns and design refactoring with AFM causes a higher number of modifications in SPL components. The results also demonstrate that CC may not be appropriate in SPL evolution context when modularity of features is an important concern. For example, the inclusion of new features usually increases tangling and scattering of others features.

In Section 2, the implementation mechanisms used in the case study are presented. Section 3 describes the study settings, including the target SPL and change scenarios. Section 4 analyzes changes made in the WebStore and MobileMedia SPLs and how they propagate through its releases. Section 5 analyzes the modularity of both SPLs with specific concern-related metrics. Section 6 provides an overall discussion of results. Section 7 presents some limitations of this work. Section 8 presents related work and points out directions for future work. Finally, Section 9 concludes this paper.

2. Variability mechanisms

This section presents some concepts about the four techniques evaluated in the study: conditional compilation (CC), feature-oriented programming (FOP), aspect-oriented programming (AOP), and aspectual feature modules (AFM). Our main goal is to compare the different composition mechanisms available to understand their distinct strengths and weaknesses. Although CC is not a new variability mechanism, we decided to include it in this study because it is still a state-of-the-practice option adopted in SPL industry, and can serve as a fair baseline for comparison [1,41,6].

2.1. Conditional compilation

The Conditional Compilation (CC) approach studied in this work is a well-known annotation-based technique for handling software variability [1,4,25]. It has been used in programming languages like C for decades and it is also available in object-oriented languages such as Java. Basically, the preprocessor directives indicate pieces of code that should be compiled or not, based on the value of preprocessor variables. The major advantage of this approach is that the code can be marked at different granularities, from a single line of code to a whole file.

The code snippet below shows the use of CC mechanisms by inserting pre-processing directives.

```
1 public class ControllerServlet extends HttpServlet {
2   public void init() {
3     actions.put("goToHome", new GoToAction("home.jsp"));
4     //#if defined(BankSlip)
5     actions.put("goToBankSlip", new
                GoToAction("bankslip.jsp"));
6     //#endif
7     //#if defined(Logging)
8     Logger.getRootLogger().addAppender(new ConsoleAppender(
         new PatternLayout("[%C{1}] Method %M
                                  executed with success.")));
9    //#endif
10   }
11 }
```

Listing 1: Example of variability management with conditional compilation.

In the example above, there are some directives that characterize the CC way of handling variability. On line 4 there is a directive `//#if defined (BankSlip)` that indicates the beginning of the code belonging to *BankSlip* feature. In line 6 there is a `#endif` directive that determines the end of the code associated to this feature. The identifiers used in the construction of these directives, in this case "BankSlip", are defined in a configuration file and are always associated with a boolean value. This value indicates the presence of the feature in the product, and consequently the inclusion of the bounded piece of code in the compiled product. The same reasoning applies to the bounded piece of code that belongs to *Logging* feature.

2.2. Feature-oriented programming

Feature-oriented programming (FOP) [38] is a paradigm for software modularization that considers features as a major abstraction. This work relies on AHEAD [11,9] which is an approach to support FOP based on step-wise refinements. The main idea behind AHEAD is that programs are constants and features are added to programs using refinement functions. Each refinement is composed on the base class in a certain order, increasing its behavior. The code snippets in Listings 2–4 show examples of a class and two class refinements used to implement variation points.

```
1 public class ControllerServlet extends HttpServlet {
2   public void init() {
3     actions.put("goToHome", new GoToAction("home.jsp"));
5   }
6 }
```

Listing 2: Example of variability mechanism with FOP (base class).

```
1 layer bankslip;
2 refines class ControllerServlet {
3   public void init() {
4     Super().init();
5     actions.put("goToBankSlip", new
                  GoToAction("bankslip.jsp"));
6   }
7 }
```

Listing 3: Example of variability mechanism with FOP (bankslip class refinement).

The example in Listing 2 shows an ordinary base class that implements a default action for *going to home* and Listing 3 presents the respective FOP class refinement that considers going to *bank slip payment* in checkout. Line 1 of Listing 3 is a clause that indicates a layer of the class refinements. The `bankslip` identifier in line 1 is used to compose the layers according to some pre-established order in the SPL configuration script that creates a specific product.

Listing 4 provides another class refinement to include the behavior of feature *Logging* in the class. This feature is designed to register successful execution of public methods.

2.3. Aspect-oriented programming

Aspect-oriented programming (AOP) has been proposed to modularize crosscutting concerns. The main mechanism of modularization is the aspect, which encapsulate a concern code that would be tangled with and scattered across the code of other concerns. An aspect is composed on the class that crosscut through the process of weaving. This work is based

```
1 layer logging;
2 refines class ControllerServlet {
3   public void init() {
4     Super().init();
5     Logger.getRootLogger().addAppender(new ConsoleAppender(
        new PatternLayout("[%C{1}] Method %M
                                executed with success.")));
6   }
7 }
```

Listing 4: Example of variability mechanism with FOP (logging class refinement).

in an extension of Java for AOP called AspectJ [27]. Listing 5 shows how an aspect can modularize the *Bankslip* feature. An aspect usually needs to provide interception points in the base code in order to get the code adequately weaved. Lines 2–3 show an example of intercepting the execution before the `init` method. Lines 4–6 show how and what will be executed after that interception point (pointcut). The aspect shown in Listing 7 (*Logging* feature) is added in a similar way.

```
1 public aspect BankSlipAspect {
2   pointcut init(ControllerServlet controller):
3     execution(public void ControllerServlet.init())&&
      this(controller) && args();
4   after(ControllerServlet controller): init(controller) {
5     controller.actions.put("goToBankSlip", new
                            GoToAction("bankslip.jsp"));
6   }
7 }
```

Listing 5: Example of variability mechanism with AOP (aspect).

2.4. Aspectual feature modules

Aspectual feature modules (AFMs) are an approach thought to explore the symbiosis of FOP and AOP [7,5,8]. An AFM encapsulates the roles of collaborating classes and aspects that contribute to a feature. In other words, a feature is implemented by a collection of components, e.g., classes, refinements and aspects. Typically, an aspect inside an AFM does not implement a role. Usually, a refinement is more adequate for this task. Aspects in AFM are usually used to do what they are good for, and in the same way of AOP: modularize a concern code that otherwise would be tangled with or scattered across many components. It is important to note that an aspect is a legitimate part of a feature module and so, is applied and removed together with the feature it belongs to. Generally, AFMs are created following a well-defined sequence of steps. First, enabled features are composed using AHEAD Tool Suite (ATS). After, aspects belonging to these features are weaved using AspectJ. Listing 6 shows a chain of mixins generated by ATS that creates redirection to payment pages function. After this process, the mixins layers are composed with aspects. Listing 7 shows an aspect that includes the behavior of feature *Logging*, similar to Listing 4. It is important to clarify that only the code of refinements and aspects was used for the analyses and the example of Listing 6 merely illustrates the functioning of the mechanism.

3. Case studies

This section describes the study based on the analysis of the evolution of two software product lines. WebStore SPL which was conceived to represent features of a real web store, such as order products and view products catalog. MobileMedia SPL that manages photos, videos and audio. We choose these SPLs because they were available to us and have been used in previous studies with similar purpose [15,17].

3.1. Research questions

The study was conducted to answer the following research questions.

RQ1. Does the use of AFM have smoother change propagation impact than using CC, FOP, or AOP?
RQ2. Does the use of AFM provide more stable design of the SPL features than using CC, FOP, or AOP during the evolution?

3.2. Infrastructure settings

The independent variable of this study is the variability mechanism used to implement the SPLs, namely, Conditional Compilation (CC), Feature-oriented programming (FOP), Aspect-oriented programming (AOP), and Aspectual Feature Modules (AFM). Two subject systems (SPLs) are used to analyze the behavior of the dependent variables: change propagation

```
1 abstract class ControllerServlet$$Base$refinements
      extends HttpServlet {
2   public void init() {
3     actions.put("goToHome", new GoToAction("home.jsp"));
4     actions.put("goToResponse", new
                GoToAction("response.jsp"));
5   }
6 }

7 public class ControllerServlet$$BasicPayment$refinements
      extends ControllerServlet$$Base$refinements {
8   public void init() {
9     super.init();
10    actions.put("goToPayment", new
                GoToAction("payment.jsp"));
11  }
12 }

13 public class ControllerServlet
      extends ControllerServlet$$BasicPayment$refinements {
14  public void init() {
15    super.init();
16    actions.put("goToBankSlip", new
                GoToAction("bankslip.jsp"));
17  }
18 }
```

Listing 6: Example of variability mechanism with AFM (mixin layers).

```
1 public aspect LoggingAspect {
2   pointcut publicMethods():execution(public * *.*(..));
3   after() returning: publicMethods() {
4     Logger.getRootLogger().addAppender(
      new ConsoleAppender(new PatternLayout("[%C{1}] Method %M
                                  executed with success.")));
5   }
6 }
```

Listing 7: Example of variability mechanism with AFM (aspect).

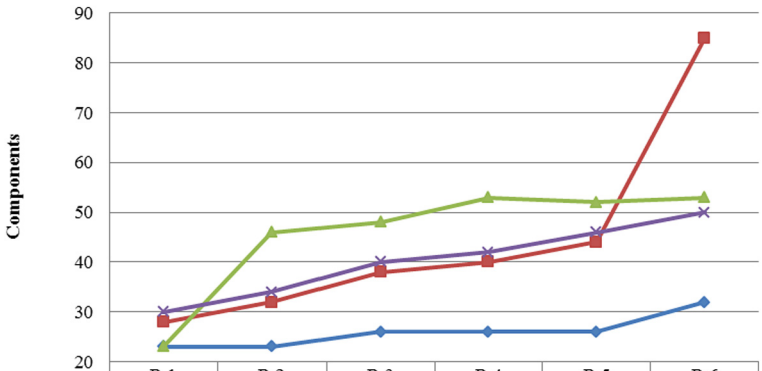
and modularity metrics. For each SPL, the study was organized in four phases: (1) construction of the subject SPL with complete releases that correspond to their respective change scenarios using the four aforementioned techniques for each release (2) identification and color shadowing of blocks addressing each feature source code, (3) measurement and metrics calculation, and (4) quantitative and qualitative analysis of the results. In the first phase, the first two authors implemented the WebStore SPL from the scratch using all different variability mechanisms. The authors also adapted the MobileMedia SPL using two different mechanisms, FOP and AFM. At end of this phase, 24 different releases of the WebStore SPL and 20 different releases of MobileMedia SPL were available. In the second phase, all code was marked according to each designed feature of both SPLs. The concrete result of this phase was text files, one for each code file, marked with the corresponding feature. In the third phase, change propagation [45] was measured and modularity metrics [17,39] were calculated. Finally, the results were analyzed in the fourth phase. The next sections present the analysis of both SPLs and discuss their change scenarios.

3.3. The evolved WebStore SPL

The first target SPL was developed to represent major features of an interactive web store system. It was designed for academic purpose, but focusing on real features available in typical web store systems. We have also designed representative changes scenarios (the same for all studied techniques – CC, FOP, AOP and AFM), considered important, that could exercise the SPL evolution.

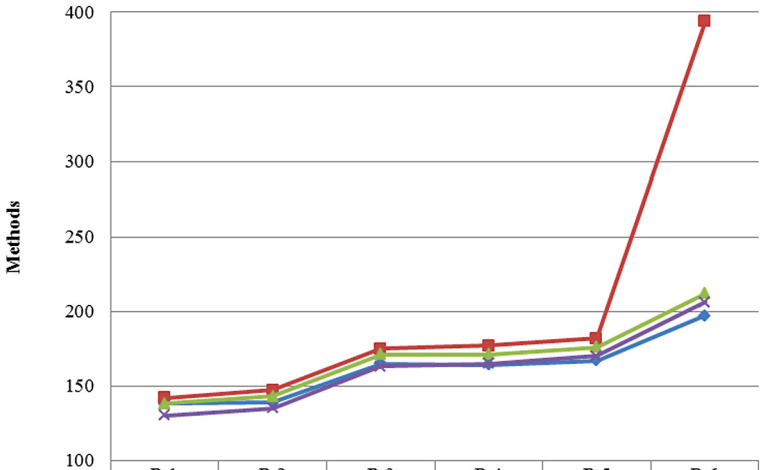
WebStore is an SPL for applications that manage products and their categories, show products catalog, control access, and payments. Figs. 1, 2 and 3 provide some measures about the size of the SPL implementation in terms of number of components, methods, and lines of source code (LOC). Classes, class refinements, and aspects were accounted as components. The number of components varies from 23 (CC) to 85 (FOP). The columns R.1 to R.6 represent the different releases of the SPL.

Fig. 4 presents a simplified view of the WebStore SPL feature model [10]. Examples of core features are *CategoryManagement* and *ProductManagement*. In addition, some optional features are *DisplayByCategory* and *BankSlip*. We use numbers in the top right-hand corner of a feature in Fig. 4 to indicate in which release the feature was included (see Table 1).



	R.1	R.2	R.3	R.4	R.5	R.6
CC	23	23	26	26	26	32
FOP	28	32	38	40	44	85
AOP	23	46	48	53	52	53
AFM	30	34	40	42	46	50

Fig. 1. WebStore SPL implementation (Components).



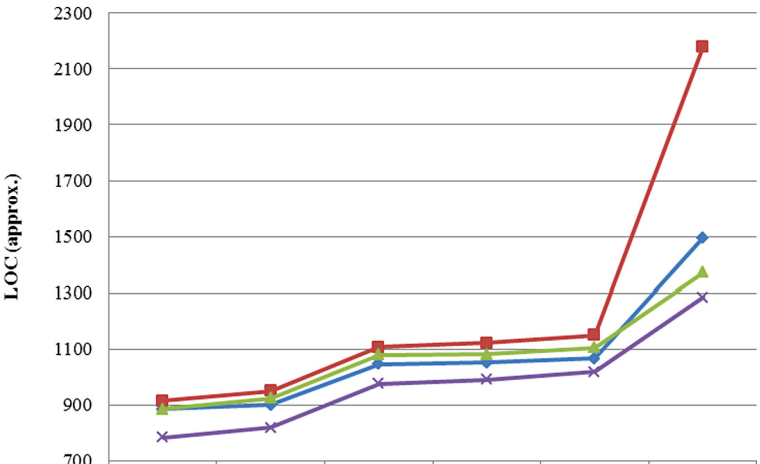
	R.1	R.2	R.3	R.4	R.5	R.6
CC	138	139	165	164	167	197
FOP	142	147	175	177	182	394
AOP	138	143	171	171	176	212
AFM	130	135	163	165	170	206

Fig. 2. WebStore SPL implementation (Methods).

The WebStore releases of each mechanism are very similar from the architecture design point-of-view. Even though they are implemented using four distinct variability mechanisms, in all different releases we could follow the MVC architectural pattern. In all mechanisms, the Release 1 contains the core of the target SPL. All subsequent releases were designed to incorporate the required changes in order to include the corresponding feature. For instance, the FOP mechanism was developed trying to maximize the decomposition of the product features. This explains why Release 1 in FOP contains more components than Release 1 that uses CC. All new scenarios were incorporated by including, changing, or removing classes, class refinements, or aspects.

3.4. The WebStore change scenarios

As aforementioned, we designed and implemented a set of change scenarios in the first phase of our investigation. The implementations of CC and FOP were already available for other study [15], but also been developed by the same people. A total of five change scenarios were incorporated into WebStore, resulting in six releases. Table 1 summarizes changes made in each release. The scenarios comprised different types of changes involving mandatory and optional features. Table 1 also presents which types of change each release encompassed. The purpose of these changes is to exercise the implementation of the feature boundaries and, so, to assess the design stability of the SPL.



	R.1	R.2	R.3	R.4	R.5	R.6
CC	885	900	1045	1052	1066	1496
FOP	915	950	1107	1121	1149	2181
AOP	885	924	1080	1081	1105	1371
AFM	784	819	976	990	1018	1284

Fig. 3. WebStore SPL implementation (LOC).

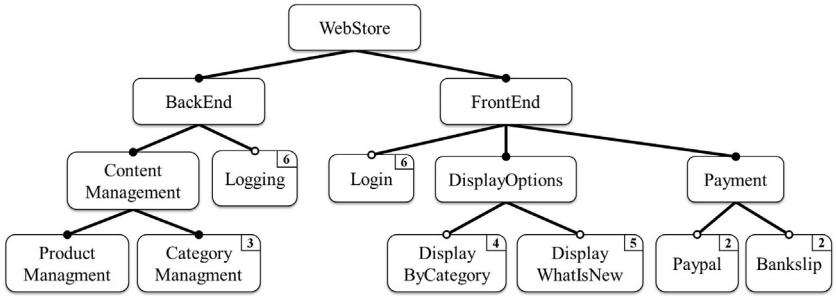


Fig. 4. WebStore basic feature model.

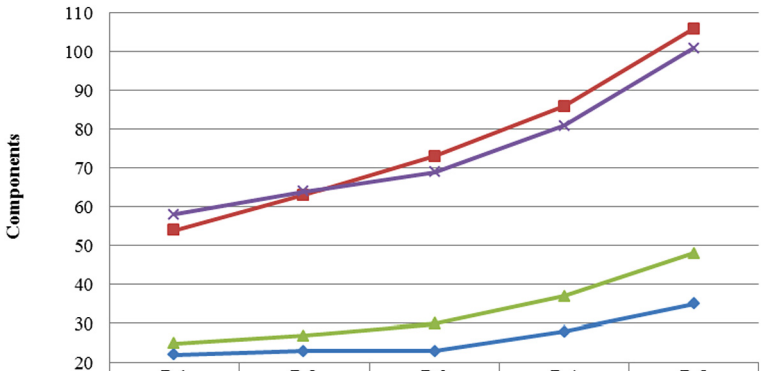
Table 1
Summary of change scenarios in WebStore.

Release	Description	Type of change
R1	WebStore core.	
R2	Two types of payment included. (*Paypal* and *BankSlip*)	Inclusion of optional feature
R3	New feature included to manage category.	Inclusion of optional feature
R4	The management of category was changed to mandatory feature and new feature included to display products by category.	Changing optional feature to mandatory and inclusion of optional feature
R5	New feature included to display products by nearest day of inclusion.	Inclusion of optional feature
R6	Two crosscutting features included. (*Login* and *Logging*)	Inclusion of optional feature

3.5. The evolved MobileMedia SPL

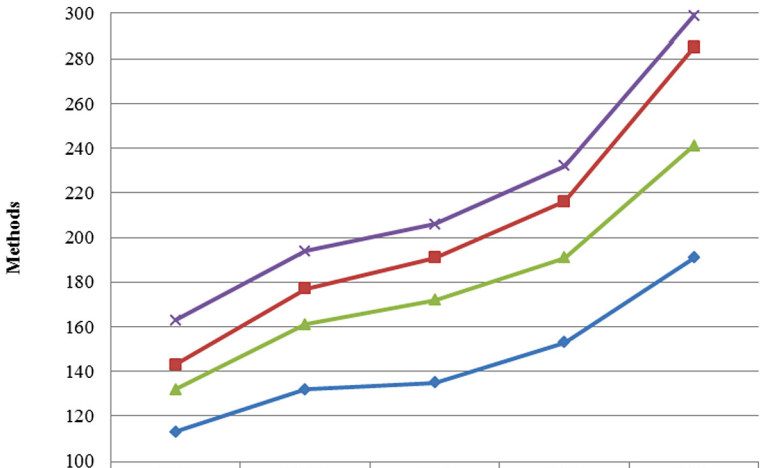
The second target SPL, called MobileMedia, was developed with the purpose to serve as reference for aspect-oriented programming studies [17]. It was designed with academic purpose, but including change scenarios with mandatory and optional features that could exercise its evolution. Although MobileMedia is not a case of the industry, was widely studied in several academic works [13,15,17,18].

MobileMedia [17] was developed based on a previous SPL, called MobilePhoto [45]. Figs. 5, 6 and 7 provide some measures about the size of the SPL implementation in terms of number of components, number of methods and number of lines of source code (LOC). Classes, class refinements and aspects were accounted as components. LOC were accounted without considering blank lines. The average number of components varies from 22 (CC) to 106 (FOP). As occurred in WebStore, FOP requires more components to implement MobileMedia features. Moreover MobileMedia AFM-based solution uses more lines of code than the FOP implementation.



	R.1	R.2	R.3	R.4	R.5
CC	22	23	23	28	35
FOP	54	63	73	86	106
AOP	25	27	30	37	48
AFM	58	64	69	81	101

Fig. 5. MobileMedia SPL implementation (Components).



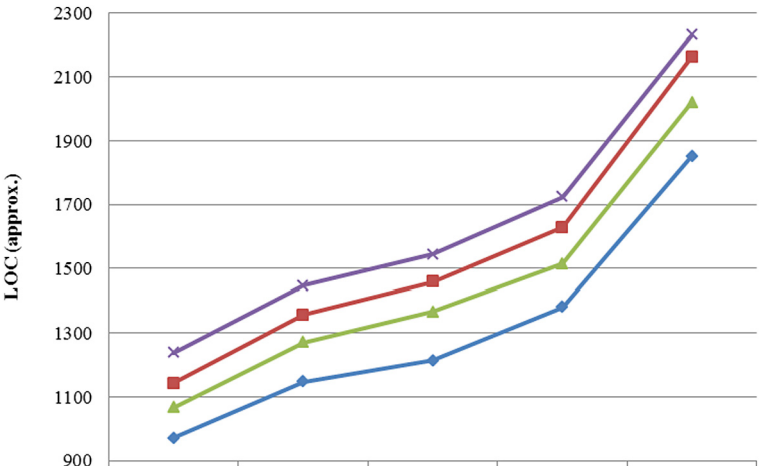
	R.1	R.2	R.3	R.4	R.5
CC	113	132	135	153	191
FOP	143	177	191	216	285
AOP	132	161	172	191	241
AFM	163	194	206	232	299

Fig. 6. MobileMedia SPL implementation (Methods).

Fig. 8 presents a simplified view of the MobileMedia SPL feature model. Examples of core features are *AlbumManagement* and *PhotoManagement*. In addition, some optional features are *Favourites*, *Sorting* and *CopyPhoto*. Similar to Fig. 4, numbers on the top right-hand corner of a feature in Fig. 8, were used to indicate in which release the feature was included (see Table 2).

3.6. The MobileMedia change scenarios

Unlike WebStore, which has been developed from scratch, we have a full CC and AOP implementation of MobileMedia available to us [17]. However, we had to design and implement the corresponding set of change scenarios in FOP and AFM. Four change scenarios were considered in MobileMedia, resulting in five releases. Table 2 summarizes changes of each release. The scenarios comprised different types of changes involving mandatory and optional features. Table 2 also presents which types of change each release encompassed. These changes were defined so that some crosscutting features were present, such as, Sorting and Favorites. These features have code tangled and scattered across several other features, hindering the modularization of their code. The purpose of both crosscutting features is exercise scenarios, in principle, more appropriated to aspect-based mechanisms (AOP and AFM) in the context of software product line evolution.



	R.1	R.2	R.3	R.4	R.5
CC	971	1147	1214	1380	1852
FOP	1142	1356	1458	1629	2163
AOP	1066	1270	1367	1516	2020
AFM	1238	1447	1545	1724	2232

Fig. 7. MobileMedia SPL implementation (LOC).

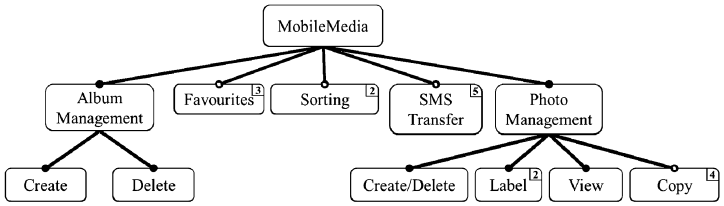


Fig. 8. MobileMedia basic feature model.

Table 2
Summary of change scenarios in MobileMedia.

Release	Description	Type of change
R1	MobileMedia core.	
R2	New feature included to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature included to edit the photo`s label.	Inclusion of optional and mandatory features
R3	New feature included to allow users to specify and view their favorite photos.	Inclusion of optional feature
R4	New feature included to allow users to keep multiple copies of photos.	Inclusion of optional feature
R5	New feature included to send photo to other users by SMS.	Inclusion of optional feature

4. Change propagation analysis

This section presents a quantitative analysis to answer RQ1. In particular, we are interested to know how different variability mechanisms affect changes in software product line evolution. The quantitative analysis uses traditional measures of change impact [45], considering different levels of granularity: components, methods, and lines of source code (Figs. 1, 2 and 3). A general interpretation of these measures is that a lower number of components modified and removed suggests a more stable solution, possibly supported by the variability mechanisms. In the case of additions, we expect that a higher number of additions of components indicates the conformance with the Open-Closed principle [33]. In this case, a lower number of additions may suggest that the evolution is not being supported by non-intrusive extensions.

4.1. Change propagation analysis for WebStore

Firstly, we perform the analysis of change propagation at component level. Fig. 9 shows the number of components added, removed and changed, respectively, in Releases 2 to 6 of the WebStore SPL.

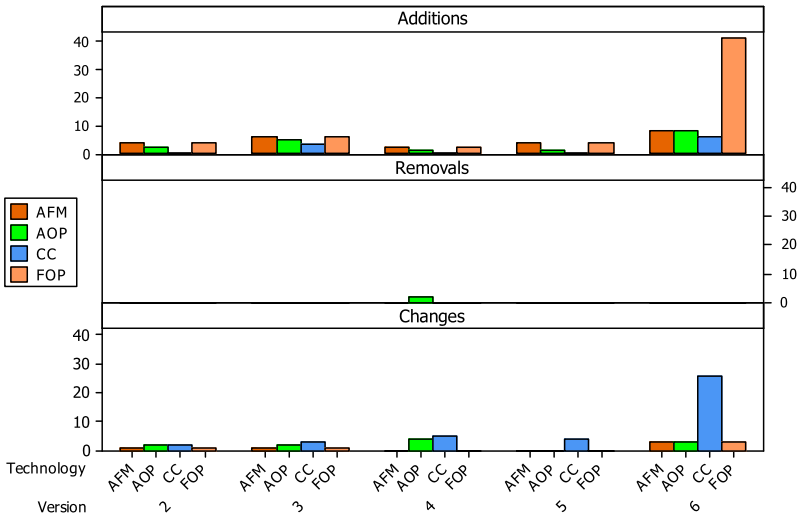


Fig. 9. Component additions, removals and changes in WebStore releases.

Considering the addition of components shown in the upper plot of Fig. 9, we observe that the CC mechanism has the lowest number of components added compared to the other approaches. Considering some specific releases, we could observe that the addition of crosscutting concerns with FOP is a painful task. For instance, in Release 6, there is a great number of component additions for FOP. This can be explained because the implementation of the crosscutting feature *Logging* requires a class refinement for each component of the SPL, which makes the number of components to double.

Considering the removal of components shown in the middle plot of Fig. 9, we observe that Release 4 using AOP has a significant difference compared to the other mechanisms, because it was necessary to remove components. This occurred because the feature *Category Management* changed from optional to mandatory, so it was necessary to remove the aspect components that allowed the enabling of this feature and code distribution of it throughout the system.

Considering the change of components shown in the bottom plot of Fig. 9, we observe that the AFM and FOP mechanism have lower number of modified components than AOP (except in Releases 5 and 6) and CC. The changes in components and methods were due to insertions, necessary to implement the desired features, which occurred inside them. This is an interesting finding: CC releases have consistently lower number of added components than the others mechanisms and also have consistently higher number of changed components than the others mechanisms. Considering that there is no notably difference considering all releases, we conclude that CC does not adhere as closely to the Open-Close principle as the other mechanisms do. This occurs because instead of facilitating the addition of new components to support new features, CC typically requires changing existing components.

Considering the difficulty of FOP in handling crosscutting concerns, we could clearly see the importance of AFM to cope with this situation. On the other hand, aspects did not work well when transforming an optional feature into mandatory because of the necessity to remove the aspects.

Figs. 10, 11 and 12 show fine-grained change propagation data, i.e., in the method and LOC level. In each of those graphs, we show boxplots for the additions, changes, and removals of methods and LOC. Boxplots are interesting because they show the tendency of the data and also the general dispersion as well. A boxplot shows the division of the dataset in four groups, each comprising a quarter of the data. The box represents the two quarters around the median, which is marked inside the box. The asterisks are considered outliers. The way we group the boxplots reveal the variation between different mechanisms in a same release.

In general, concerning the number of methods and lines of code, there is no sharp difference and variation between the measures of the four mechanisms. An important exception is for the implementation of Release 6. In that case, there are a substantial higher number of method additions within FOP (Fig. 10 – M A – Release 6). This can be explained because the implementation of the *Logging* feature required more method additions for the required new refinements. The median of added LOC and methods for CC in Release 6 is also lower than for the others (Fig. 10 – LOC A – Release 6 – CC) contributing with the finding that CC changes are more prominent than additions (Fig. 11 – M C – Release 6 – CC) undermining the Open Closed principle. We can also observe in Fig. 11 that AFM and FOP are less prone to changes.

Concerning the removals of lines and methods, we have the same tendency of components. In Release 4 using AOP had a significant difference on the number of methods and lines removed, which was significantly higher than in other approaches. This occurred because the removal of components resulted in removal of LOC and methods.

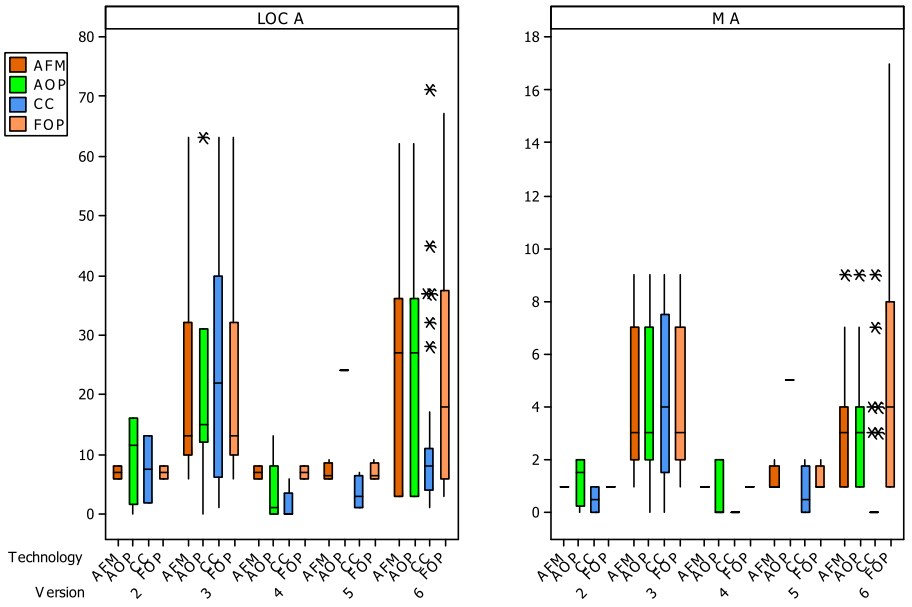


Fig. 10. LOC and method additions in WebStore releases.

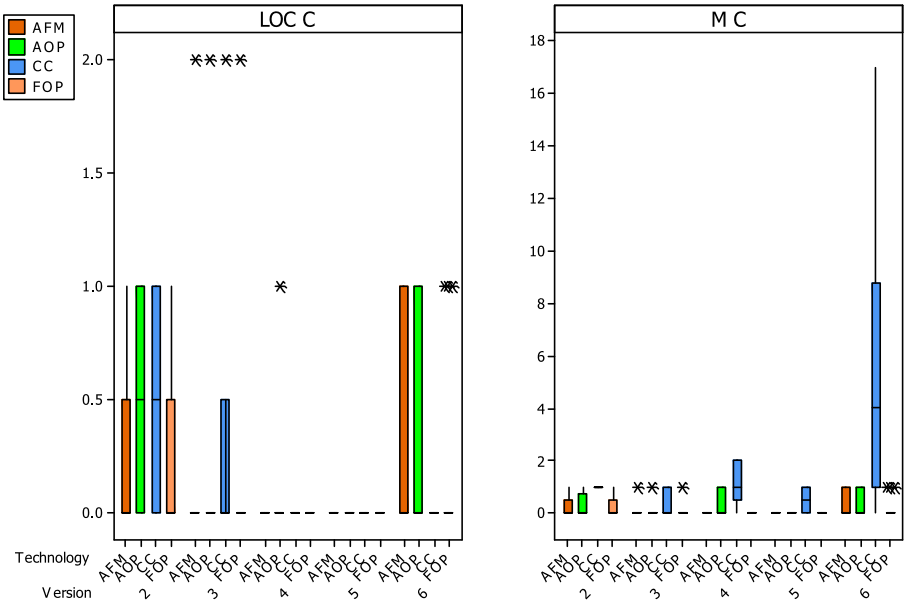


Fig. 11. LOC and method changes in WebStore releases.

4.2. Change propagation analysis for MobileMedia

In this section, we perform the same analysis for MobileMedia as conducted in the previous section. Firstly, we perform the analysis of change propagation at the component granularity level.

The upper graph in Fig. 13 shows that, similarly to the WebStore SPL, the CC mechanism has the lowest number of added components compared to the other approaches. Moreover, the approaches AFM and FOP have similar number of insertions, which are higher than the other approaches. This finding contributes with the hypothesis that AFM and FOP adheres more to the Open-Closed Principle because the introduction of features is promoted by the insertion of components.

The middle plot of Fig. 13 shows the number of removed components in Releases 2 to 5 of the MobileMedia SPL. Release 4 using FOP and AFM had a significant difference from the others. This occurred because in this release, the developers restructured the design to facilitate the modifications required in Release 5. So, several refinements had to be removed, similarly both in FOP and in AFM.

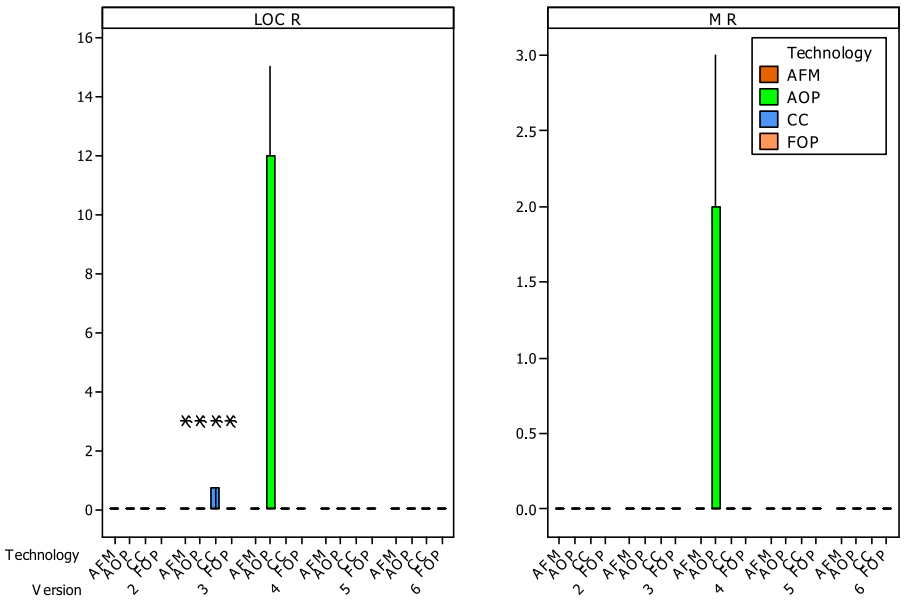


Fig. 12. LOC and method removals in WebStore releases.

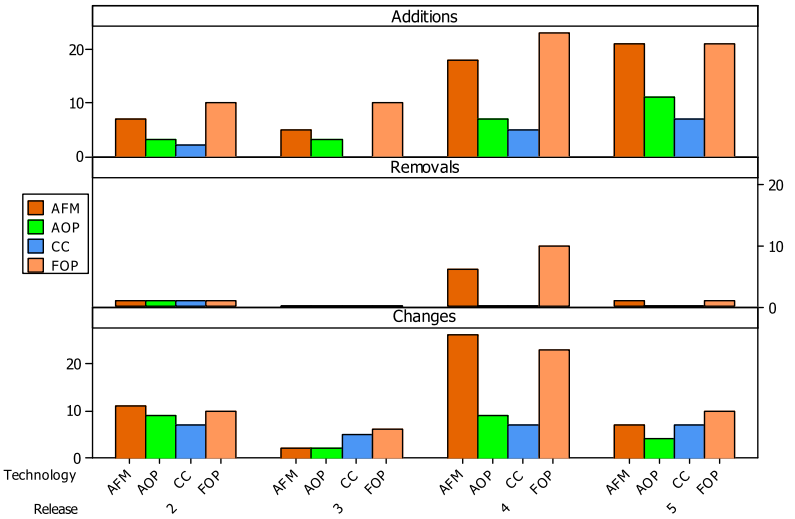


Fig. 13. Component additions, removals and changes in MobileMedia releases.

The bottom plot of Fig. 13 shows the number of changed components. Interestingly, CC has presented a lower number of modified components. However, this situation can be explained because CC releases have systematically lower number of components than the other approaches. So, if we consider the number of changed components relative to the total number of components of the respective release, we can observe that the relative number of components changed with CC is the highest in Releases 3 and 5. In Release 4, FOP and AFM still have the highest number of changed components. We explain this higher number of FOP and AFM because of the restructuring previously mentioned.

The change propagation analysis at the granularity level of lines of code and methods is shown in Figs. 14, 15 and 16. All these graphs show that concerning the number of methods and lines of code, there is no sharp difference between the measures of the four mechanisms. However, in Fig. 14, it is possible to see that AOP has slightly more line and method additions than the others, except in Release 4 where CC had more additions. This is can be explained by the fact that several components refactoring forced the addition of new pointcuts and advices. Differently from the WebStore SPL, FOP and AFM have shown a lower number of line and method additions.

In Fig. 15, we can observe that CC has high degree of LOC changes in Releases 2, 3 and 5 specially if compared to AFM and FOP. In Release 4, AFM and FOP do not perform well concerning LOC changes if compared to CC. The explanation for this result is the fact that the components' design was restructured in this release for FOP and AFM. The impact of

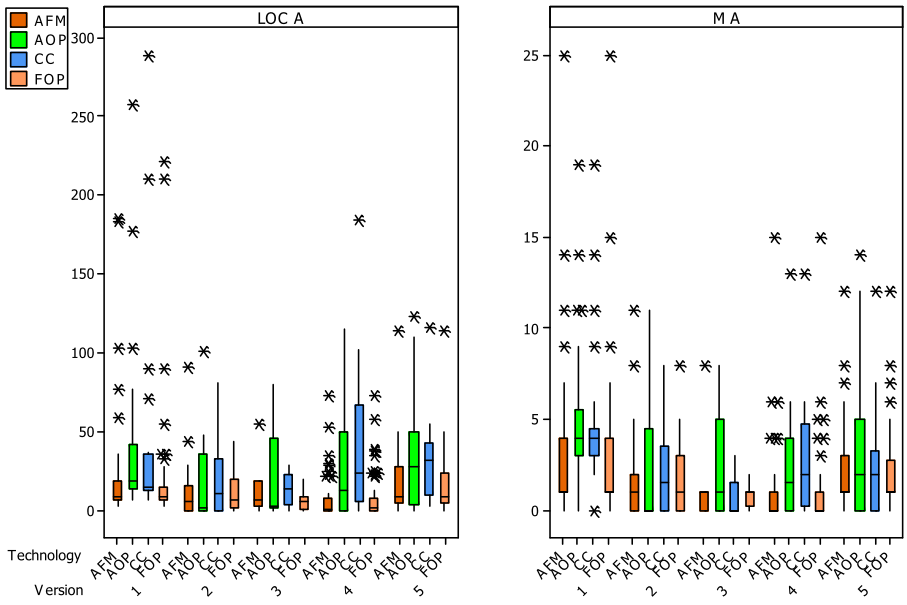


Fig. 14. LOC and method additions in MobileMedia releases.

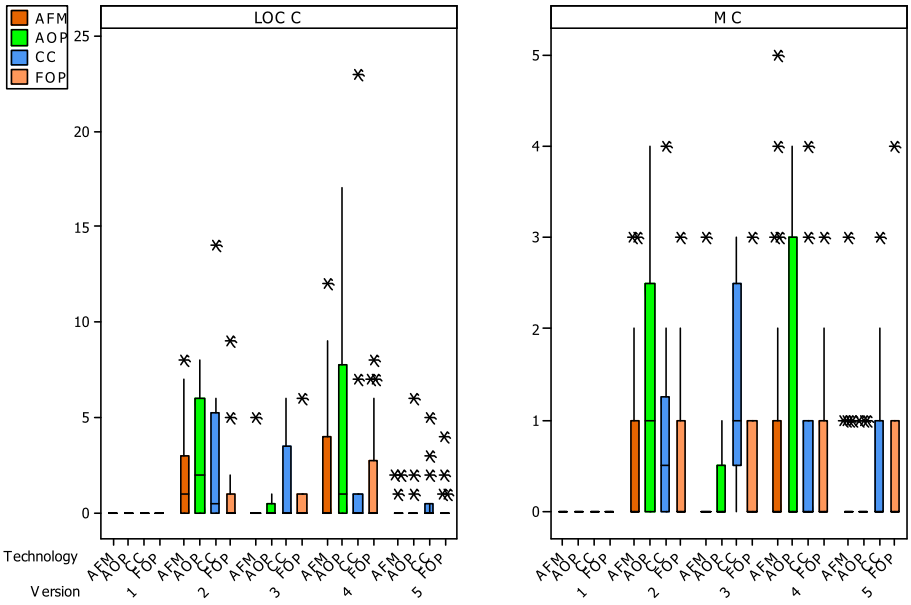


Fig. 15. LOC and method changes in MobileMedia releases.

LOC change has also been higher for AOP, except in Release 3, where CC has highest impact and in Release 5 where no significant difference could be observed (Fig. 15 – leftmost graph). The same pattern could be observed for method changes where Releases 2 and 4 have shown higher values for AOP (Fig. 15 – rightmost graph).

In Fig. 16, we can observe no sharp distinction between the different mechanisms. Nonetheless, we can see some outliers in Release 4 for all approaches. These outliers have occurred in a fundamental "controller" class, called *BaseController*, which was an important restructuring target.

The previous results from WebStore and MobileMedia have provided some main conclusions about the four different mechanisms:

- CC has presented higher degree of LOC changes and lower degree of component addition, suggesting that new features are typically introduced by changing existing code, which is considered harmful from maintenance point of view.

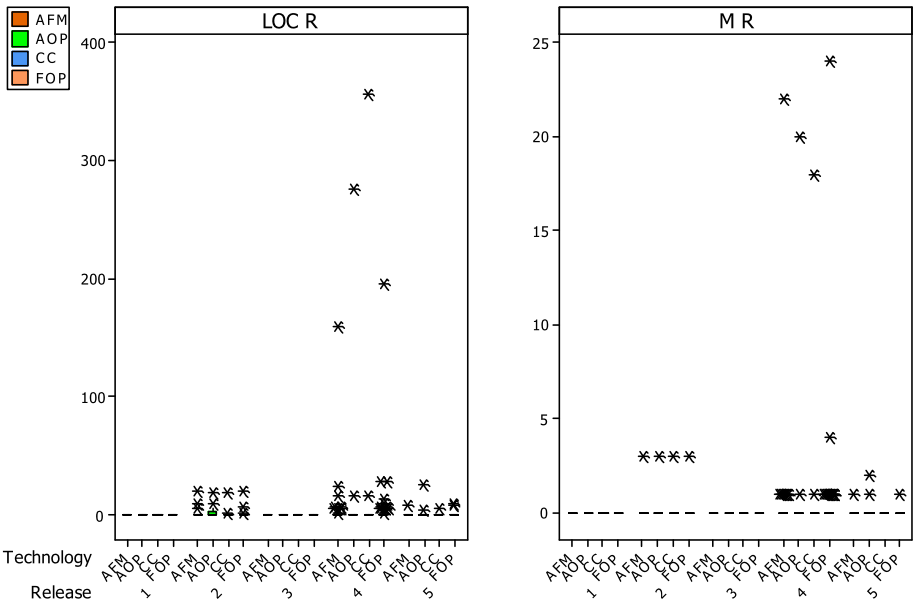


Fig. 16. LOC and method removals in MobileMedia releases.

- FOP and AFM have consistently higher degree of component additions, suggesting that new features are typically introduced by inserting new components, which is the ideal scenario from maintenance point of view.
- The similar pattern in component additions showed by FOP and AFM reveals that fine-grained refinement modules are used as the main construction to implement new features, as opposed to additions or refactoring of coarse-grained modules.
- AOP alone has not presented significant better indicators than CC. This result may suggest that significant changes are required to modularize existing code to new aspects.

5. Modularity analysis

This section presents and discusses the results for the analysis of the stability of the SPLs design throughout the implemented changes.

Separation of concerns (SoC) is a fundamental principle related to the decomposition mechanisms used both in design as in implementation. Concerns are the main way for decomposing software into smaller parts, and at the same time more manageable and comprehensible. A feature is a functionality increment in software [9] and this concept is closely related to that of concerns – some researchers regard them as equivalent [36].

To support our analysis, we used a suite of metrics for quantifying concern modularity [39]. We choose these metrics because they have been used and validated in several previous empirical studies [13–15,18,21,24].

This suite measures the degree to which a single feature of the system maps to:

(i) components (i.e., classes, class refinements and aspects) – based on the metric Concern Diffusion over Components (CDC) [39]. This metric quantifies the degree of feature scattering considering the granularity level of components. It counts the number of classes and interfaces that contributes to the implementation of a feature.
(ii) operations (i.e., methods and advices) – based on the metric Concern Diffusion over Operations (CDO) [39]. This metric quantifies the degree of feature scattering considering the granularity level of methods. It counts the number of methods and constructors realizing a feature.
(iii) lines of code – based on the metrics Concern Diffusion over Lines of Code (CDLOC) [39] and Number of Lines of Concern Code (LOCC) [17]. CDLOC computes the degree of feature tangling. For instance, given a certain feature F, this metric counts the number of "switches" between F and lines of code realizing other features [39]. A switch occurs when a code block realizing F is followed by a code block realizing another feature, and vice-versa. LOCC counts the total number of lines of code that contribute to the implementation of a feature.

We adapted these metrics considering the ratio of the measured value to the total value on that release, for instance, CDC was calculated as the ratio of classes that contributes to the implementation of a feature to the total number of classes, in addition, our relative CDC represents the percentage of classes that are used to implement the feature. This relative metrics enabled us to analyze together the set of metric values for all features. For all the employed metrics, a lower value implies a better result.

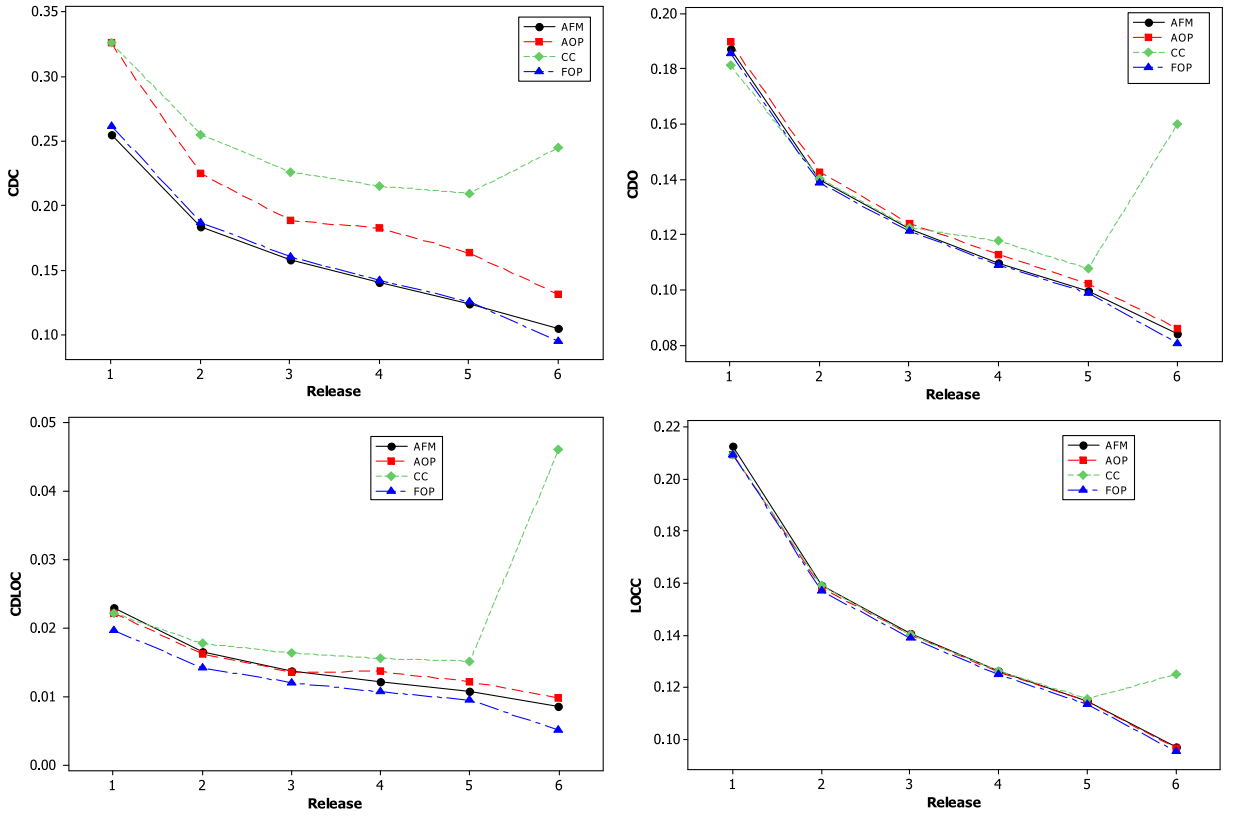


Fig. 17. Metrics values through WebStore evolution.

This metrics suite has a common characteristic that distinguishes them from traditional software metrics [24]. They capture information about the realization of features cutting across one or more components, i.e., these metrics are used for quantifying Separation of Concerns (SoC) [24,33]. They can be applied to any kind of software component in either object-oriented or feature-oriented programs. Although these metrics were originally proposed to quantify concern properties, they can also be used to quantify features properties. The terms concern and feature are used without distinction in this study.

In the next sections, we provide an overall analysis with the mean value of each metric for each mechanism/release. Because the overall analysis is based only on mean values, it does not cope with the variability of the data, so we conduct a second analysis using probability plots that reveals details about the overall dispersion of the metric values.

5.1. Overview of the modularity analysis for WebStore

The data was collected and organized in one spreadsheet for each metric. For WebStore, each sheet of one studied metric has 8435 lines, i.e., one line for each combination of feature, release, technique (AFM, AOP, CC, FOP) and component (classes, refinements, aspects). For example, the first line corresponds to the following combination: (*Base, 1, CC, ControllerServlet*). Considering this example for metric CDC, we can interpret that the component *ControllerServlet* of Release 1 using Conditional Compilation has some value with metric for feature *Base*.

We provide an overall view of the modularity data aggregating metrics' values with the mean function applied on the group of components from each release implemented with each mechanism (AFM, AOP, CC, FOP).

Fig. 17 presents CDC, CDO, CDLOC and LOCC mean values for each release of the subject SPL.

The CDC mean values for FOP and AFM were consistently the lowest in all releases. The values for AOP stayed in between, while CC had the highest values. These lower CDC values for FOP and AFM reveal that FOP and AFM are better suited to encapsulate features within the modules because lower CDC values means lower scattering of features across the modules. Moreover, the values for FOP and AFM being almost the same suggests that FOP refinements are responsible for that better encapsulation.

The CDLOC mean values for FOP were also consistently the lowest in all releases, meaning that FOP contributes to lower concern scattering not only considering the component level, but also in the LOC granularity level. The CDLOC mean values for AFM were slightly lower (better) than AOP in Releases 4, 5, and 6, due to the influence of FOP refinements in AFM. CDLOC values for CC were the worst ones, especially in Release 6. The highest CDC and CDLOC values for CC together contribute to reveal the inability of the CC mechanism to modularize the SPL features.

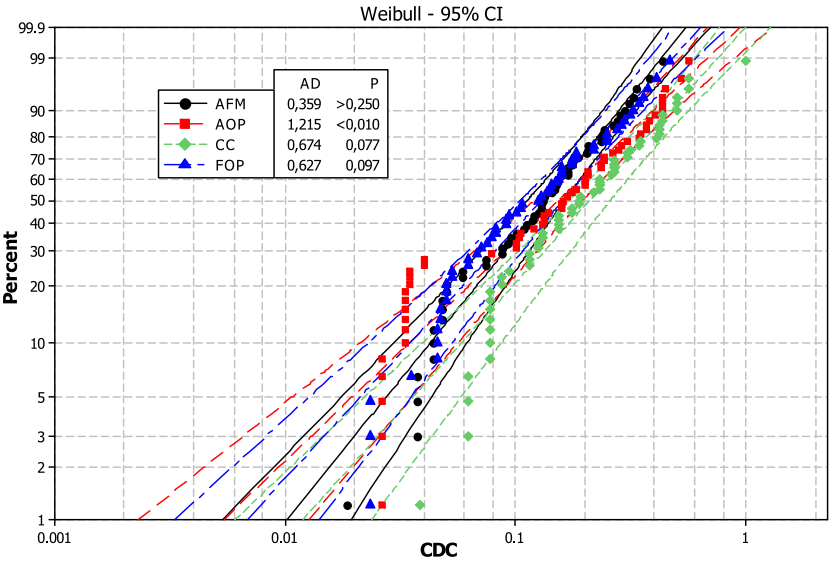


Fig. 18. Probability plot for CDC.

For CDO and LOCC there was no significant difference between releases or techniques, except in Release 6 where the CC values were significantly the worst ones.

5.2. Dispersion analysis of modularity metrics for WebStore

In the previous section, the graphs have shown only the mean value of the metrics. These graphs were useful to gain an overall insight, but they are limited for analyze the dispersion of the data. Because the mean value can be significantly skewed by outliers and do not represent the tendency of most part of the data, we conduct an analysis of the dispersion of the data using probability plots. The goal of this analysis is to gain confidence on the previous results based on the mean values and probably reveal more details on the data.

Probability plots are well suited to compare two or more data sets. The data are plotted against a theoretical distribution in such a way that the points ideally should form approximately a straight line. A theoretical distribution should be chosen, among known distributions, in such a way that it fairly fits the data. We have tested the data with the set of distributions of Minitab16© that could be applied to grouped data and did not require parameterization. Those distributions were Normal, Lognormal, Exponential, Smallest Extreme Value, Weibull, Largest Extreme Value, Logistic, and Loglogistic. The criterion to select the distribution that best fitted our data was to verify the Anderson–Darling – AD goodness-of-fit statistic and its associated p-value. The lowest mean AD for all curves (one for each mechanism) defined the best fitted distribution. In all probability plots, a box with the AD value and their respective p-value is provided. If the p-value is greater than 0.05 the fitness is significant. A not-significant fitness does not impact our analysis. We actually do not require significant fitness because we just want to compare the curves of different mechanisms for different or similar behavior.

Following, we will describe how we can analyze a probability plot using the one shown in Fig. 18, which is the probability plot for the CDC values of the WebStore releases. The other probabilities plots can analyzed in the same way. In this probability plot each point is a CDC value for a feature in a specific release using a specific mechanism (AFM, AOP, CC, FOP). The x-axis corresponds to the CDC value and the y-axis corresponds to the percentage number of plotted points from the bottom up. Lower CDC values are plotted firstly from the bottom up. For example, the green diamond point at the lower part of the graph represents a CDC value 0.03846 and the y-axis indicates that 1.21% of the points (in fact one point) were plotted. In order to interpret the graph, we can observe that 100% of the CDC values for CC are further to the right. This fact reveals that not only the mean value for CDC is higher for CC mechanism (as shown in the previous section), but also that the CDC value is consistently higher than for all other mechanisms. For the other mechanisms, we can see that AFM and FOP are almost coincident, reflecting the similar curve presented in the previous section. For AOP, we can observe that it presents the highest frequency of lower values considering the 27% lowest CDC values, being further to the left. However, in the range of the 28% to 99% of the CDC highest values, AOP presents lower values than AFM and FOP. This explains why in the previous section AOP is in between CC and AFM-FOP. In this case, we could see that differently from CC, CDC values for AOP is not always higher than AFM and FOP. This can be explained by the fact that some features could be well modularized in AOP. Examples of these features are *Bankslip*, *DisplayByCategory*, *DisplayWhatIsNew*, *Logging*, and *PayPal* in Release 6. On the other hand, the *Logging* feature produced high scattering both in FOP and CC, confirming the hypothesis that both FOP and CC do not cope well with crosscutting features.

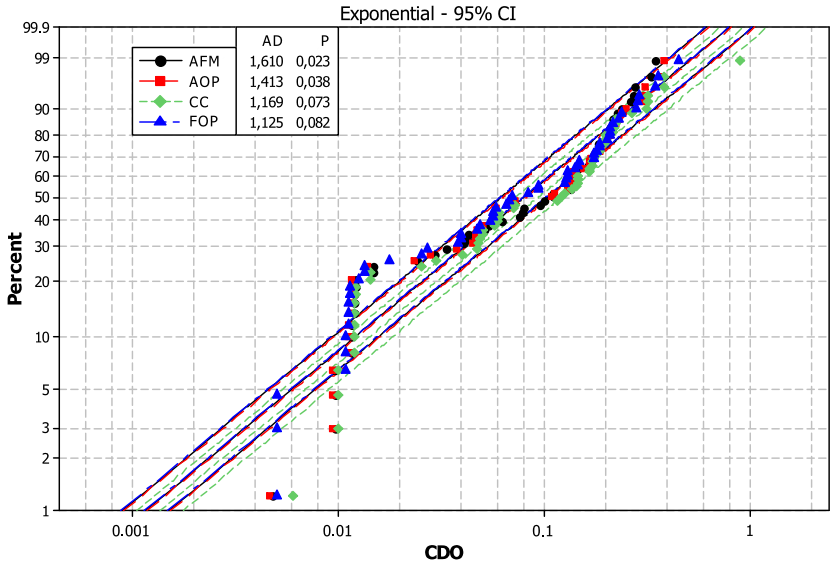


Fig. 19. Probability plot for CDO.

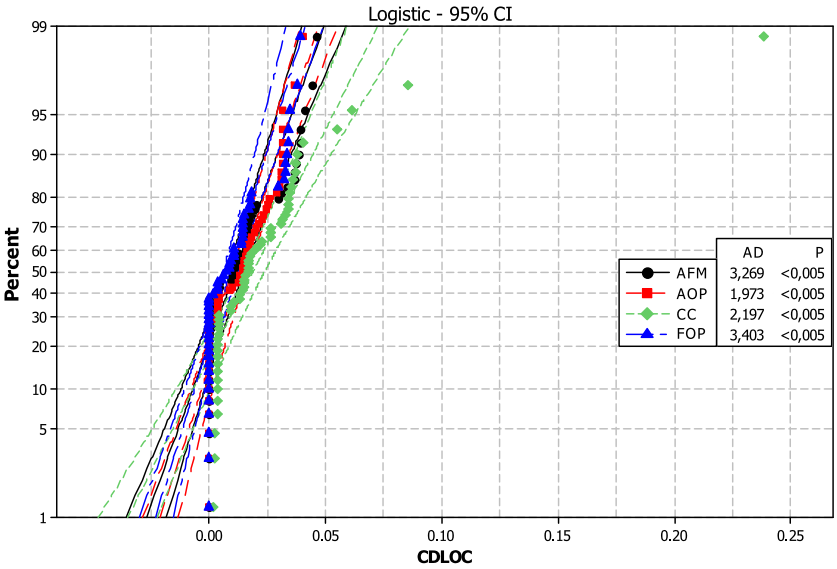


Fig. 20. Probability plot for CDLOC.

Fig. 19 shows the probability plot for CDO in the WebStore SPL. This plot confirms the mean CDO shown in Fig. 17, that there is no sharp difference between the different mechanisms.

Fig. 20 shows the probability plot for CDLOC in the WebStore SPL. In this plot we can observe that the tangling with CC is consistently the highest among all mechanisms because the data points for are consistently further to the right. The tangling with FOP was mostly the lowest, except in the cases where AOP was better, which was in the crosscutting features. AFM followed the FOP tendency.

Fig. 21 shows the probability plot for LOCC in the WebStore SPL. This plot has the same behavior of CDO plot, i.e., there is no significant dominance of any approach. This plot is consistent with the mean plot of Fig. 17. Nonetheless, we can observe that the *Logging* feature are highly scattered in FOP and CC, compared to their correspondents in AFM and AOP.

5.3. Modularity analysis for MobileMedia

The data was collected and organized in one spreadsheet for each metric. For MobileMedia, each sheet of one studied metric has 13 738 lines, i.e., one line for each combination of feature, release, variability mechanism (AFM, AOP, CC, FOP) and components (classes, refinements, aspects).

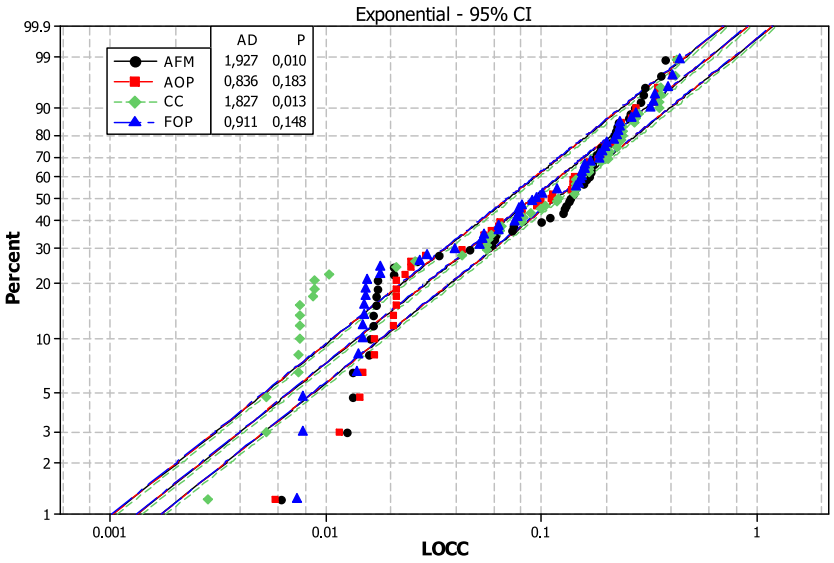


Fig. 21. Probability plot for LOCC.

Fig. 22 presents CDC, CDO, CDLOC and LOCC mean values for each release of the MobileMedia SPL. Similar to the results observed in the WebStore SPL, the CDC mean values for FOP and AFM were consistently the lowest among all releases, followed by AOP and CC, which have presented the highest values. The correspondence of CDC mean values between the target SPLs is an important finding, enhancing the consistency on the scattering over components.

Considering the CDO mean values, differently from the WebStore SPL, we can observe that AFM and FOP values are consistently lower, although the difference is not as sharp as it is in the CDC plot. For AOP, the CDO mean values stayed in between, following the tendency of CDC.

Considering the CDLOC mean values, we could also observe a different behavior when comparing to the results obtained in WebStore SPL. The CDLOC values for AFM and FOP were the lowest in all releases, but the values approximate to the AOP values in the last release. Also considering CDLOC, AOP is in between as the plots for CDC and CDO, with the exception of the values presented in Release 1.

Considering that Fig. 22 shows only mean values, we conducted similar analysis as in WebStore SPL with probability plots.

Fig. 23 shows the probability plot for CDC, where we can observe that AFM and FOP consistently present lower values than AOP and CC. AOP is mostly better than CC, but there are some points where AOP and CC had a similar behavior regarding scattering. In general, we can observe a similar pattern for the MobileMedia and WebStore concerning the CDC metric.

Fig. 24 shows the probability plot for CDO, where we could not observe significant differences between the variability mechanisms. Although these differences are not significant, we observed a lower value of mean for CDO in AFM and FOP compared to AOP and CC, caused by the values lying above the seventieth percentile. Nonetheless, for lower CDC values, AOP, and to a lesser extent CC, have lower values than FOP and AFM. In this case, the graph of Fig. 22, helps to explain that the features introduced in Releases 4 and 5 were benefited from the AOP mechanisms.

Fig. 25 shows the probability plot for CDLOC. In this plot, we can see that the tangling of AFM and FOP are typically lower than in AOP and CC. Interestingly, this behavior is similar to the one observed in the WebStore SPL. Moreover, in MobileMedia, we can observe that AOP is lower than in CC in the lowest values, but are similar in the highest values confirming that the features introduced in Releases 4 and 5 were benefited from the mechanisms of AOP that reduce tangling.

Fig. 26 shows the probability plot for LOCC. As in the plot with mean values, we can see no significant difference between the mechanisms. This result is consistent with the mean value plot and also consistent with the WebStore SPL.

We can observe that the main findings of the previous results are:

- CC presented clearly higher scattering and tangling of features. CDC and CDLOC were fundamental to reveal this situation.
- AFM and FOP have presented better behavior concerning the scattering and tangling of features. AFM has presented slightly better numbers over FOP concerning scattering and tangling in MobileMedia, where crosscutting features had more impact. The aspectual modules were the responsible for this behavior.

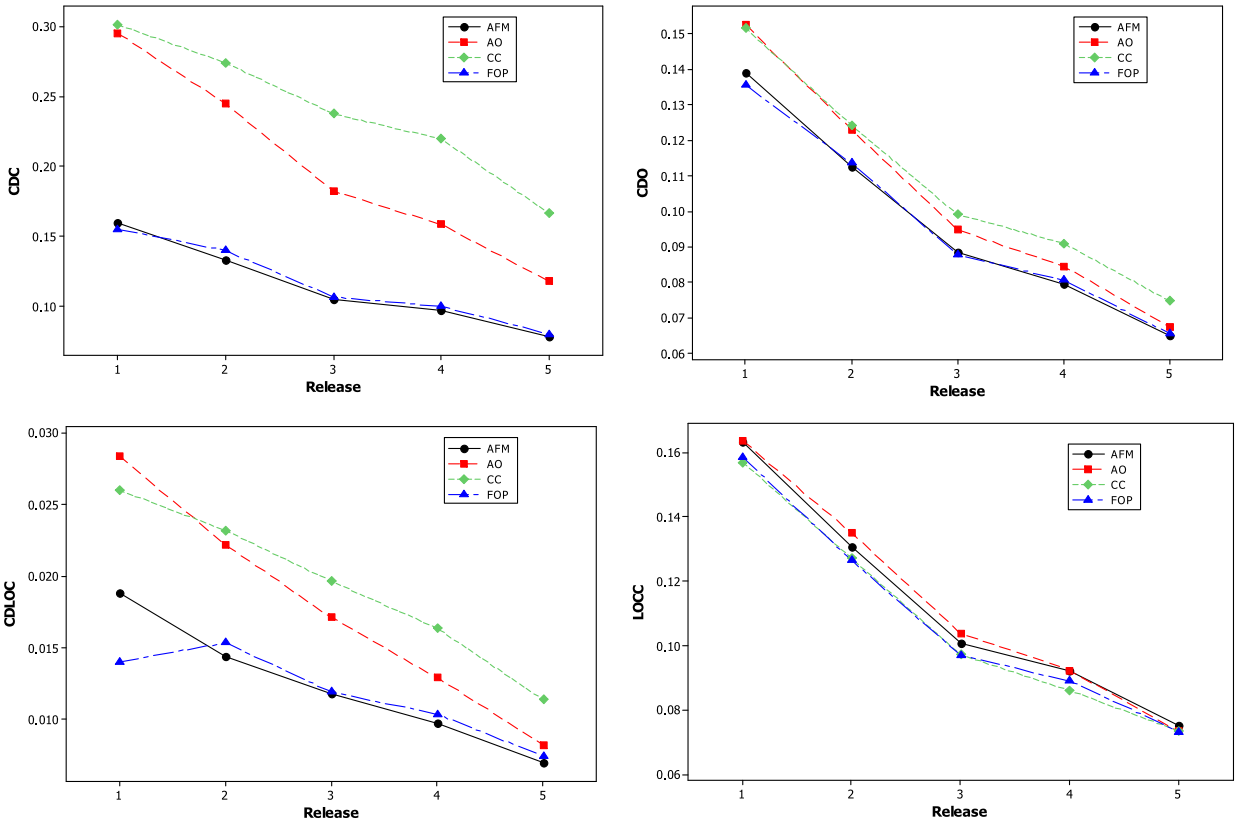


Fig. 22. Metrics values through MobileMedia evolution.

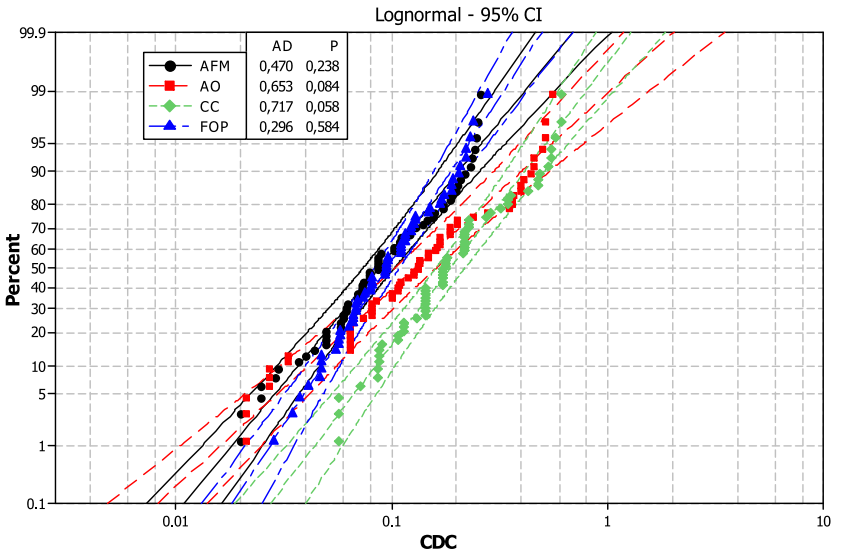


Fig. 23. Probability plot for CDC.

- AOP alone helped to reduce scattering and tangling compared to CC, but not as significantly as AFM and FOP revealing that the aspectual modules have their importance at specific points of the SPL and the refinement modules are still responsible for the major impact in scattering and tangling.

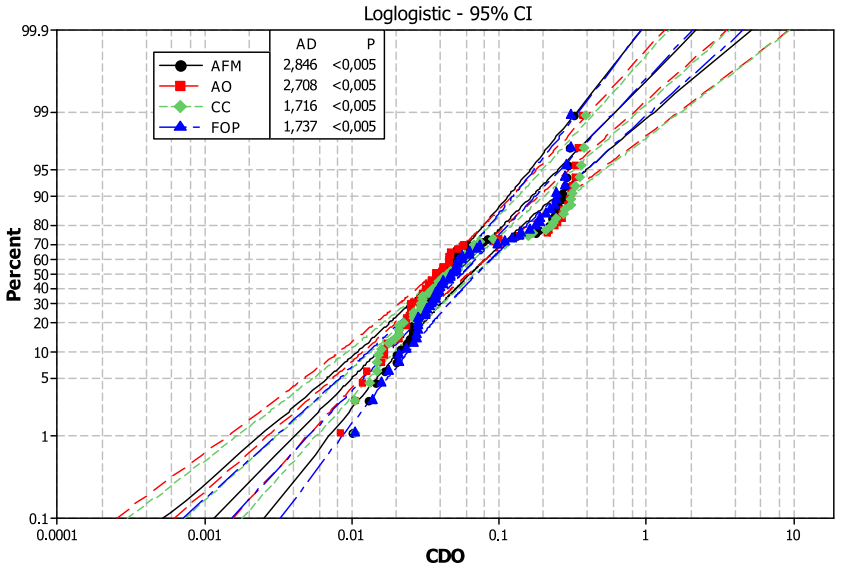


Fig. 24. Probability plot for CDO.

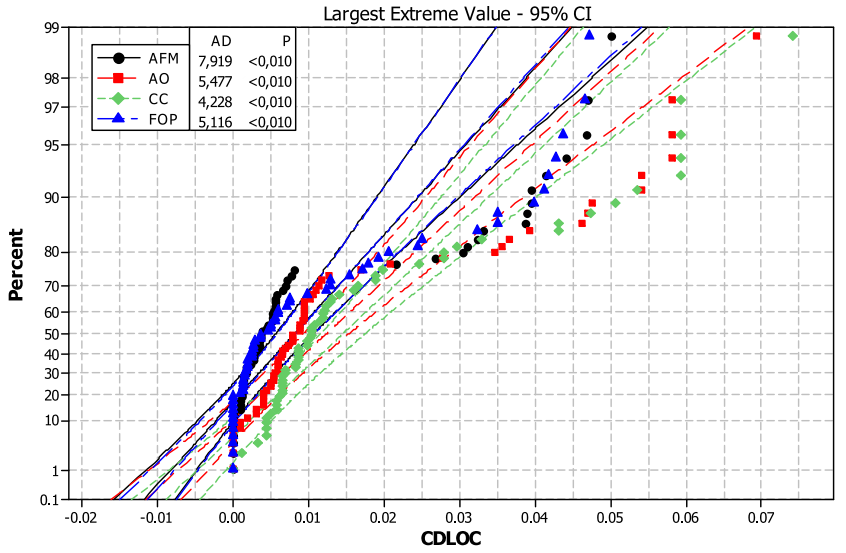


Fig. 25. Probability plot for CDLOC.

6. Discussion

From the analysis carried out in Sections 4 and 5, four interesting situations, discussed below, naturally emerged with respect to which type of modularization mechanism presents superior modularity and stability in specific conditions.

AFM and FOP succeed in features with no shared code. This situation was observed with three optional features of WebStore SPL (*Bankslip*, *Paypal*, and *DisplayWhatIsNew*) and six optional feature of MobileMedia SPL (*CreateAlbum*, *DeleteAlbum*, *CreatePhoto*, *DeletePhoto*, *EditPhotoLabel* and *ViewPhoto*). In these cases, the code for these features were independent (no sharing, i.e., there is no common code implementing more than one feature) and then, AFM and FOP solutions presented lower values and superior stability in terms of tangling, specially FOP w.r.t. (CDLOC) and to scattering over components (CDC) which explain the previous data. The results of the other metrics (CDO and LOCC) did not follow the same trend of the CDC metric. This situation occurs because whether the granularity of methods and lines of code is lower than granularity of components, then the distribution of features is proportionally in all mechanisms. On the other hand, since the granularity of components is higher, the impact on modularity metrics is higher too.

The efficiency of AFM and FOP mechanisms to isolate such feature that are implemented with independent code can be explained by the use of class refinements, which increment the base behavior in a pluggable way. The CC mechanism

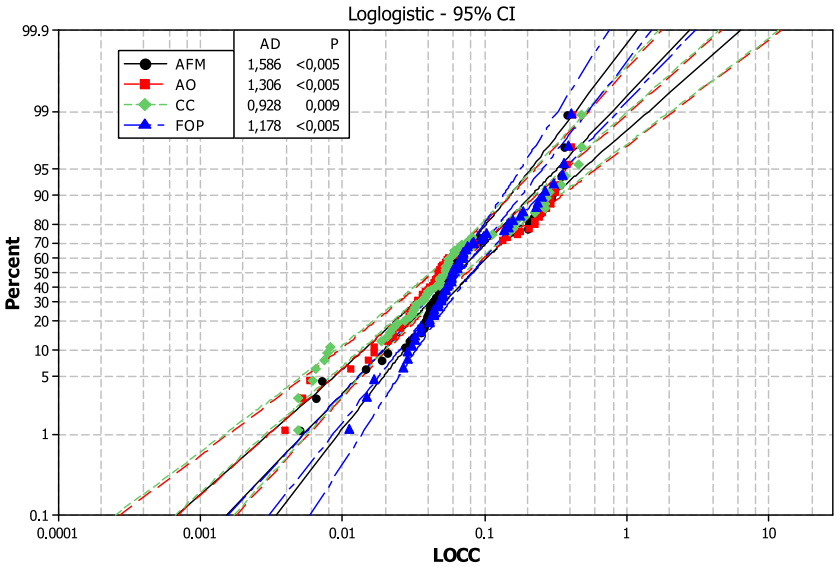


Fig. 26. Probability plot for LOCC.

has no capability to insert refinements, thus the developer needs to implement these features intrusively changing existing components, by introducing *#ifdefs* where features are tangled.

Even in the case of features that are implemented with shared code between each other, AFM and FOP have presented good modularity results. Nonetheless, the gain was not as observable as in the case of features with no shared code. The feature *Logging* is an exception, because even if it has no shared code with other features, this feature would be highly scattered in FOP solution. This result was expected considering the crosscutting nature of this feature, as discussed below.

When crosscutting concerns are present in the solution AFM are recommended over FOP. Another interesting finding that emerged from our analysis is that FOP does not cope well with crosscutting concerns. In this case, AFM provided an adequate solution, because it did not force the use of aspects to modularize features with no shared code, but still did not force painful scattered refinements to implement a crosscutting feature.

The difficulties of FOP mechanisms to implement crosscutting concerns were observed with the feature *Logging*, but they could not be observed with the feature *Login*, even though both are crosscutting features from WebStore SPL. The same situation was not observed with features *Favourites* and *Sorting* from the MobileMedia SPL. In the feature where this situation was observed, it was necessary to define several refinements within the FOP solution.

Even if this excessive refinement has not been observed in other FOP implementations of crosscutting features, some core features have benefited from the use of aspect, such as, *BasicBackEndDefinitions* and *BasicFrontEndDefinitions* from the WebStore SPL and features *AlbumManagement* and *PhotoManagement* from the MobileMedia SPL. Thus, even indirectly, aspects also have advantages in the implementation of non-crosscutting features. This can be explained because the code of crosscutting features would be tangled with these features, but using aspects it does not happen. As these features are bigger in terms of LOC, then the results are more expressive. Thus, the use of aspects also improves the modularity of features affected by crosscutting concerns.

Refactoring in design at component level has important impact in AFM and FOP. Component refactoring that impacts the architectural design tends to affect more AFM and FOP than the other mechanisms, because not only base classes but also the refinements are affected. In the case of AFM, this impact can be even worse because aspects can also be affected. This situation was observed in Release 4 of the MobileMedia, where some components were split and renamed with a specific goal to prepare the SPL for future releases. The change propagation analysis conducted in Section 4 has supported this finding.

CC compilation should be avoided when modular design is an important requirement. Although conditional compilation is still widely used in large scale projects [41], our data have shown that its use did not produce a stable architecture and should be avoided specially in situations where changes are frequent. This situation was observed in most analyses performed in Sections 4 and 5. On the other hand, this mechanism still could have some advantages that could influence its adoption that were not considered in this work, for example, the ease of understand the mechanism and the availability of robust tools that support the development [1,41].

7. Threats to validity

The validity evaluation of the results depends to a large extent on how well threats have been handled. Four types of validity threats [43] are considered: *conclusion validity*, *internal validity*, *external validity* and *construct validity*.

Conclusion validity concerns the relationship between treatment and outcome. That is, it affects the ability to draw correct conclusions based on data [43]. For the reliability of the measurement process, 33 740 data points were collected. A data point is a measurement on a single member of a statistical population [43]. An independent author who did not collect the respective data applied statistical analysis. Finally, conclusions are based on cross-discussion among all authors of this paper. Therefore, we believe this threat has been mitigated and our conclusions sound.

Threats to internal validity refer to matters that may affect an independent variable with respect to causality, without the knowledge of the researcher. They threat the conclusion about a possible causal relationship between treatment and outcome [43]. A possible threat to internal validity in this study is the fact that most releases of the SPLs were developed by some of the authors. In addition, there is a reasonably large space for different designs and alternative design options would produce different results. To mitigate this threat, all designs of WebStore were carefully constructed both to take the best practices of each implementation technique and to maintain a similar division of components. The know-how acquired in similar previous studies by the authors [15,19], associated with developers who work in the software industry, help to reduce the influences in results. In the MobileMedia case, the base application, formerly called MobilePhoto [45], was developed by independent researchers and it was also assessed in previous empirical studies with different purposes [17,18].

Threats to external validity may limit the ability to generalize the experimental results. Some factors can be considered in this case, such as, the special purpose of the subject SPLs and the choice of the evolution scenarios. In fact, it is hard to select enough representative samples of all application domains in a study like ours. Aware of that, our choices were carefully discussed in order to have representative scenarios of typical maintenance tasks in SPLs. The implementation languages and tools we used also limit the generalization since there are many alternatives. We decided to use AHEAD and AspectJ, for instance, because they are popular languages and also have common features of other FOP and AOP languages.

Finally, concerning the construct validity, one issue is on how much support modularity metrics offer to produce robust answers to the design stability problem. As a matter of fact, we concluded that these metrics offer a limited view on the overall design's quality. They are mostly related to the modularization of features, which are notably important for stable SPLs. The scope of this study has been narrowed to SPLs systems in order to cope with this issue. Although alternative metrics are available, we choose the mostly used and validated ones [21,24].

8. Related work

Recent research work has also analyzed stability and reuse of SPLs [13,17]. For instance, Figueiredo et al. [17] performed an empirical study to assess modularity, change propagation, and feature dependency of two evolving SPLs. Their study focused on aspect-oriented programming while we analyzed variability mechanisms available in feature-oriented programming in this study.

Gomes and Monteiro [23] also studied two different AOP languages, namely Object Teams/Java [43] and AspectJ. They investigate the impact of specific composition mechanisms on design pattern implementations [20]. Their results have shown that Object Teams/Java is highly recommended over AspectJ when the goal is to support the development of complex architectures and to provide enhanced evolvability. This study, on the other hand, focuses on a feature oriented programming language and our results are not limited to implementations of design patterns.

Lopez-Herrejon and others [29] evaluated the support for features in five different advanced modularization techniques, namely AspectJ, Hyper/J, Jiazzi, Scala and AHEAD. They have investigated properties such as feature definition and composition capabilities of different language constructions such as aspects, units, refinements and traits. Their conclusion has shown that none of the studied mechanisms completely backup efficient SPL construction, and further studies about these and other mechanisms are required to understand their full potential. Our study complements Lopez-Herrejon's work by providing in-depth analysis based on quantitative data.

Dantas and his colleagues [13] also conducted an exploratory study to analyze the support of new modularization techniques to implement SPLs. Their study aimed at comparing the advantage and drawbacks of aspect-based techniques in terms of stability and reuse. Their work suggests a minimal advantage of CaesarJ [34] over the other mechanisms. Our work complements Dantas' by adding an analysis of different languages (AHEAD [11] and AspectJ [27]) and an evaluation of other SPL quality properties.

Apel and others [7], who proposed the Aspectual Feature Modules approach [5,8], have also used size metrics to quantify the number of components and lines of code required by their approach in an SPL implementation. Their study, however, did not consider a significant suite of software metrics and did not address SPL evolution and stability.

In another work Greenwood et al. [24] used similar suites of metrics to ours to assess the design stability of an evolving application. However, their work focused mostly on analyzing how Object-oriented (OO) and AO designs are affected by change scenarios that target extensibility (i.e., perfective and refactoring changes). In contrast to our work, they did not target at assessing the impact of evolutionary changes of core and variable features in SPLs.

Another advanced modularization technique, called Delta-oriented programming (DOP) [40] has gained attention in SPL community. DOP is a superset of FOP, designed specifically to increase feature expressiveness and flexibility in SPLs development. The major abstractions of DOP are delta modules, which are responsible for encapsulate program deltas (additions, modifications and removals) and complex constraints between them that enable safe and unambiguous combination of features. Although there are differences between DOP and FOP language constructions, we believe that if the same analyses regarding modularity and stability were applied to DOP, the results would not show significant differences when compared to FOP. Because of the nature of delta modules, the results would probably present slightly lower number of components and higher tangling of features. However, we acknowledge that DOP could overcome FOP in terms of enabling type-safe composition, handling optional feature problem, and improving SPL evolution means. These advantages are due to the natural capabilities of delta modules, namely *after* clauses to express dependencies, combination of features to avoid code duplication and the possibility to add new self-contained delta modules instead of performing intrusive refactoring on existing code. Further studies are required to compare these two development options in depth.

Other studies focused on challenges in the software evolution field [22,30,32]. These works have in common the concern about measuring different artifacts through software evolution, which relies directly on the use of reliable software metrics [26]. Furthermore, there is a shared sense about software metrics on engineering perspective: they are far from being mature and are constantly the focus of disagreements [26,31,44].

Several studies have investigated variability management on SPLs [2,3,28,37,42]. Batory et al. [9] have reported an increased flexibility in changes and significant reduction in program complexity measured by the number of methods, of lines of code, and of tokens per class. Simplification in evolving SPL architecture has also been reported in [35,39], as consequence of variability management.

9. Concluding remarks and future work

This study evolved two SPLs in order to assess the capabilities of contemporary variability mechanisms to provide SPL modularity and stability in the presence of change requests. The study with more than one SPL allowed us to analyze what results were evident, and thus infer that they are repeatable and scalable. Such evaluation included two complementary analyses: change propagation and feature modularity. The use of variability mechanisms to develop SPLs largely depends on our ability to empirically understand their positive and negative effects through design changes.

Some interesting results emerged from our analysis. First, the AFM and FOP designs of the studied SPLs tend to be more stable than the other approaches. This advantage of AFM and FOP is particularly observable when a change targets optional features. Second, we observed that AFM and FOP class refinements adhere more closely the Open-Closed principle. Furthermore, such mechanisms usually scale well for dependencies that do not involve shared code and facilitate multiple different product instantiations. However, FOP does not cope well when crosscutting concerns must be addressed. In this case, AFM provides a better scenario concerning the propagation of changes.

Our results also indicate that conditional compilation (CC) may not be adequate when feature modularity is a major concern in the evolving SPL. For instance, the addition of new features using CC mechanisms usually causes the increase of feature tangling and scattering. These crosscutting features destabilize the SPL architecture and make it difficult to accommodate future changes.

For future work, the investigation of other advanced modularization techniques and alternative metrics to assess other quality attributes in SPLs could be the natural next steps. For instance, it would be possible to check the support of more recent mechanisms, the amount of reused code, and properties related to robustness like coupling and cohesion.

We also aim to replicate this study with SPLs from different domains and richer change scenarios. In particular, we still interested in studying scenarios that involve the inclusion of alternative features and the modification of features that interact.

Acknowledgements

This work was partially supported by FAPEMIG, grants APQ-02376-11, APQ-02532-12, APQ-0286-11 and CNPq grants 485235/2011-0 and 475519/2012-4.

References

[1] B. Adams, W. De Meuter, H. Tromp, A.E. Hassan, Can we refactor conditional compilation into aspects?, in: 8th ACM International Conference on Aspect-Oriented Software Development, AOSD '09, ACM, Virginia, New York, 2009, pp. 243–254.
[2] C. Adler, Optional composition – a solution to the optional feature problem?, Master thesis, University of Magdeburg, Germany, February 2011.
[3] M. Ali Babar, L. Chen, F. Shull, Managing variability in software product lines, IEEE Softw. 27 (2010) 89–91.
[4] V. Alves, A.C. Neto, S. Soares, G. Santos, F. Calheiros, V. Nepomuceno, D. Pires, J. Leal, P. Borba, From conditional compilation to aspects: a case study in software product lines migration, in: First Workshop on Aspect-Oriented Product Line Engineering (AOPLE), Portland, USA, 2006.
[5] S. Apel, D. Batory, When to use features and aspects? A case study, in: GPCE, Portland, Oregon, 2006.
[6] S. Apel, C. Kästner, Virtual separation of concerns – a second chance for preprocessors, J. Object Technol. 8 (6) (September 2009) 59–78.
[7] S. Apel, T. Leich, G. Saake, Aspectual mixin layers: aspects and features in concert, in: Proceedings of ICSE'06, Shanghai, China, 2006.
[8] S. Apel, T. Leich, G. Saake, Aspectual feature modules, IEEE Trans. Softw. Eng. 34 (2008) 162–180.

[9] D. Batory, Feature-oriented programming and the AHEAD tool suite, in: 26th International Conference on Software Engineering, ICSE'04, IEEE Computer Society, Washington, 2004, pp. 702–703.
[10] D. Batory, Feature models, grammars, and propositional formulas, in: Proceedings of the 9th International Software Product Line Conference (SPLC), 2005, pp. 7–20.
[11] D. Batory, J. Sarvela, Rauschmayer: scaling step-wise refinement, IEEE Trans. Softw. Eng. 30 (6) (2004) 355–371.
[12] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.
[13] F. Dantas, A. Garcia, Software reuse versus stability: evaluating advanced programming techniques, in: 23th Brazilian Symposium on Software Engineering, SBES'10, 2010.
[14] M. Eaddy, T. Zimmermann, K.D. Sherwood, V. Garg, G.C. Murphy, N. Nagappan, A.V. Aho, Do crosscutting concerns cause defects?, IEEE Trans. Softw. Eng. 34 (2008) 497–515.
[15] G. Ferreira, F. Gaia, E. Figueiredo, M. Maia, On the use of feature-oriented programming for evolving software product lines – a comparative study, Sci. Comput. Program. (2013).
[16] E. Figueiredo, C. Sant'Anna, A. Garcia, T.T. Bartolomei, W. Cazzola, A. Marchetto, On the maintainability of aspect-oriented software: a concern-oriented measurement framework, in: Proc. of European Conf. on Soft. Maint. and Reeng (CSMR), Athens, 2008.
[17] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, F. Dantas, Evolving software product lines with aspects: an empirical study on design stability, in: 30th International Conference on Software Engineering, ICSE '08, Leipzig, Germany, 2008, pp. 261–270.
[18] E. Figueiredo, C. Sant'Anna, A. Garcia, C. Lucena, Applying and evaluating concern-sensitive design heuristics, in: 23rd Brazilian Symposium on Software Engineering (SBES), Fortaleza, Brazil, 2009.
[19] F. Gaia, G. Ferreira, E. Figueiredo, M. de Almeida Maia, A quantitative assessment of aspectual feature modules for evolving software product lines, in: Proceedings of the 16th Brazilian Symposium on Programming Languages, Natal, Brazil, 2012, pp. 134–149.
[20] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.
[21] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa, Modularizing design patterns with aspects: a quantitative study, in: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD'05, ACM, New York, NY, USA, 2005, pp. 3–14.
[22] M. Godfrey, D. German, The past, present, and future of software evolution, in: Frontiers of Software Maintenance, 2008, pp. 129–138.
[23] J.L. Gomes, M.P. Monteiro, Design pattern implementation in object teams, Softw. Pract. Exp. 43 (12) (December 2013) 1519–1551.
[24] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, A. Rashid, On the impact of aspectual decompositions on design stability: an empirical study, in: ECOOP, Berlin, 2007.
[25] Y. Hu, E. Merlo, M. Dagenais, B. Lague, C/C++ conditional compilation analysis using symbolic execution, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM), 2000.
[26] C. Jones, Software metrics: good, bad and missing, Computer 27 (1994) 98–100.
[27] C. Kästner, S. Apel, D. Batory, A case study implementing features using AspectJ, in: International SPL Conference, 2007.
[28] K. Lee, K.C. Kang, E. Koh, W. Chae, K. Bokyoung, B.W. Choi, Domain-oriented engineering of elevator control software: a product line practice, in: Proceedings of the First Conference on Software Product Lines: Experience and Research Directions, Kluwer Academic Publishers, 2000, pp. 3–22.
[29] R.E. Lopez-Herrejon, D. Batory, W.R. Cook, Evaluating support for features in advanced modularization technologies, in: ECOOP Lecture Notes in Computer Science, Springer, 2005, pp. 169–194.
[30] J. Maletic, H. Kagdi, Expressiveness and effectiveness of program comprehension: thoughts on future research directions, in: Frontiers of Software Maintenance, 2008, pp. 31–40.
[31] T. Mayer, T. Hall, A critical analysis of current OO design metrics, Softw. Qual. J. 8 (1999) 97–110.
[32] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfield, M. Jazayeri, Challenges in software evolution, in: IWPSE'05: Proceedings of the Eighth International Workshop on Principles of Software Evolution, IEEE Computer Society, 2005, pp. 13–22.
[33] B. Meyer, Object-Oriented Software Construction, 1st ed., Prentice-Hall, Englewood Cliffs, 1988.
[34] M. Mezini, K. Ostermann, Conquering aspects with Caesar, in: 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, 2003.
[35] M. Mezini, K. Ostermann, Variability management with feature-oriented programming and aspects, in: 12th ACM SIG-SOFT Twelfth International Symposium on Foundations of Software Engineering, SIGSOFT'04/FSE-12, ACM, New York, NY, USA, 2004, pp. 127–136.
[36] G.C. Murphy, A. Lai, R.J. Walker, M.P. Robillard, Separating features in source code: an exploratory study, in: Proceedings of the 23rd International Conference on Software Engineering (ICSE), IEEE Computer Society, 2001, pp. 275–284.
[37] U. Pettersson, S. Jarzabek, Industrial experience with building a web portal product line using a lightweight, reactive approach, in: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2005, pp. 326–335.
[38] C. Prehofer, Feature-oriented programming: a fresh look at objects, in: ECOOP 1997, 1997, pp. 419–443.
[39] C. Sant'Anna, A. Garcia, C. Chavez, A. von Staa, C. Lucena, On the reuse and maintenance of aspect-oriented software: an assessment framework, in: Brazilian Symposium on Software Engineering (SBES), 2003, pp. 19–34.
[40] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: Proceedings of Software Product Line Conference, SPLC'10, 2010, pp. 77–91.
[41] A. Sutton, J.I. Maletic, How we manage portability and configuration with the C preprocessor, in: Proceedings of 23rd International Conference on Software Maintenance (ICSM), IEEE, 2007, pp. 275–284.
[42] M. Svahnberg, J.v. Gurp, J. Bosch, A taxonomy of variability realization techniques, Softw. Pract. Exp. (2005) 705–754.
[43] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, Boston, MA, USA, 1999.
[44] S.S. Yau, J.S. Collofello, Design stability measures for software maintenance, IEEE Trans. Softw. Eng. 11 (9) (1985) 849–856.
[45] T.J. Young, Using AspectJ to build a software product line for mobile devices, MSc dissertation, Master's thesis, University of British Columbia, Department of Computer Science, 2005.