Abstract— With system development becoming increasingly incremental, software reuse and stability stand out as two of the most desirable attributes of high-quality software. In this context, a key goal in contemporary software design is to simultaneously promote reuse and stability of the software modules. However, the achievement of this goal is far from trivial as it largely depends on the programming techniques being employed in the software implementation. While the use of a specific advanced mechanism can somehow contribute to modules’ reuse, it might also require developers to make various undesirable changes in their interfaces and implementation. In this context, there are a growing number of techniques for improving modularity, ranging from aspect-oriented and feature-oriented programming to composition filters. This paper presents an exploratory analysis of advanced programming techniques on how they make it possible to reach a better tradeoff of software reuse and stability. The evaluation was carried out based on 11 releases of 2 product lines, which were originally built to promote the stable reuse of common modules across different products. Our results revealed that a hybrid incarnation of feature-oriented and aspect-oriented programming seems to be the most promising programming technique. For instance, the combined use of virtual inner classes, mixin composition, and pointcut-advice tended to promote product-line modules with both superior stability and reusability.

Keywords - Modularity mechanisms, composition mechanisms, software stability, software reuse, software product lines.

I. INTRODUCTION

The simultaneous satisfaction of key quality attributes, such as reuse and stability, are main tenets in the development of long-living software systems, such as software product lines. It has been widely recognized that reuse of software systems is not useful if it is detrimental to software stability [1]. A software system or a particular module is considered stable [2] if its interface or implementation is not undesirably modified and ripple effects [2][3] do not manifest in the presence of changes. A software or a particular module is reused in a project if it is used in more than one context within the software system [4] 1. There are various Advanced Programming Techniques (APTs) for improving modularity of source code. These techniques have emerged over the last years and range from Aspect-Oriented Programming (AOP) [5] and Feature-Oriented Programming (FOP) [6] to Composition Filters [7]. There are also programming models [8][9][10] that support hybrid incarnations of these programming techniques. The expressive power of such contemporary programming techniques leads to the expectation that, by applying them, reuse and stability of software modules are both improved.

Ideally, reusing software modules should not require that invasive modifications are made in modules that are not the focus of a change. For instance, modifications should not be made to either interfaces or internal members of modules that are being reused in a different context as software evolves. Otherwise, the software instabilities, provoked by these harmful modifications, are likely to degrade software design over time [3]. In this fashion, positive (and negative) effects of adopting contemporary modularization techniques need to be systematically investigated. However, while proponents of these techniques claim that reuse and stability of software systems can be simultaneously improved [11][12], there is not much knowledge on whether they achieve this promise or not.

In fact, there is only very fragmented knowledge on the assessment of emerging programming techniques. The problem is that all empirical studies developed so far tend to carry out a narrow analysis, focusing solely on either modularity or stability [1][13]. For instance, Sven Apel et. al [9][10] have recently contributed with a hybrid programming model that combines AOP and FOP techniques. They have performed an initial evaluation of such a model, but it is limited from several perspectives. First, the evaluation involved only their own hybrid programming model and did not embrace multiple contemporary advanced mechanisms, such as composition filters. Second, they did not perform any tradeoff analysis of reuse and stability in their comparisons with traditional AOP models, such as Aspect. There are other initial assessments of such traditional models, but they either concentrate only on software stability [1][14] or on partial evaluation of reuse [15]. Hence, there is no investigation of how these techniques affect simultaneously reuse and stability. Another problem is that it is often assumed that modularity properties are the only characteristics that have a major impact on stable reuse of software modules.

This paper presents a first exploratory evaluation about the impact of advanced programming techniques on both software reusability and stability. The tradeoff analysis has only previously carried out in the context of applying object-oriented programming [1]. We also aim to identify if there are
specific advanced mechanisms – such as virtual classes, superimposition and inter-type declaration – that play a central role in promoting simultaneous satisfaction of reuse and stability. The advanced mechanisms investigated in this study (Section III) are supported by two programming languages, namely Compose* [16] and CaesarJ [17]. Compose* and CaesarJ implementations of two software product lines (Section II) were compared with existing AspectJ versions of these systems. Product lines were the target of our evaluation as both stability and reuse of their core modules are major driving requirements in these systems. The comparison with AspectJ was motivated by many reasons, such as: (i) it entails the most popular programming model that supports “conventional” mechanisms, such as pointcuts, advice, and intertype declarations, and (ii) most of the previous evaluations [1][14] focus on AspectJ.

Our analyses were based on previously-defined modularity [14], stability [3] and reuse metrics [13], so that we could test some relevant hypotheses (Section IV). We have found that the hybrid incarnation of FOP and AOP, seems to be the most promising programming technique to support stable reuse of modules. For instance, the combined use of virtual inner classes, mixin composition, and pointcut-advice tended to promote product-line modules with both superior stability and reusability. In addition, differently from what we expected, we have observed that modularity attributes were not the main factor that contributed to superior stability and reuse of the analyzed software systems. The explanations for these conclusions are given in analysis section (Section V). Furthermore, our study was compared to previous studies (Section VI) and threats to validity and our final considerations are presented in Section VII and VIII, respectively.

II. TARGET PRODUCT LINES

In order to promote a systematic evaluation of Compose* and CaesarJ, the selected cases were two software product lines (SPLs): (i) a large open-source software, called iBatis (110 KLOC), and (ii) an embedded mobile software, called MobileMedia (5 KLOC). These SPLs were chosen as they have been evolved and underwent various forms of changes (Section V) over a long period of time. Also, they are interesting to our study because they contain different categories of: (i) crosscutting concerns and crosscutting features, and (ii) domain-specific features and non-functional features. These characteristics would enable us to expose AspectJ, Compose* and CaesarJ implementations to a number of varied feature modularizations and changes. For instance, it would enable us to assess to what extent certain mechanisms, such as composition filters and collaboration interfaces, could help to minimize the modifications of existing modules through SPL changes.

In addition, the two SPLs are from significantly-different domains, and well-designed implementations with Java and AspectJ are already available, facilitating the analysis of the APTs in this study. These existing implementations have prioritized both reuse and maintainability of crosscutting and non-crosscutting features. Distinguishing characteristics of the target applications are presented in the following.

iBatis. It is a Java-based open source framework for data mapping. It is composed by more than 60 releases incrementally implemented. Four releases were chosen and implemented using the AspectJ language. The following features were chosen to be refactored with modularity mechanisms of AspectJ, Compose* and CaesarJ: type mapping, error context, and design patterns. We chose these features for four main reasons: (i) they have undergone more frequent and heterogeneous changes through the iBatis history, (ii) their own crosscutting nature and characteristics (e.g. as occurs with type mapping and error context) call for the advanced mechanisms of the APTs being assessed in our study, (iii) they cover both broadly- and narrowly-scoped crosscutting concerns (e.g. design patterns) that enables to explore the different advanced mechanisms of APTs, and (iv) there were good practice guidelines on how to modularize these chosen features with the advanced mechanisms [8][16][17]. Thus, we also analyzed if the iBatis aspectization process followed such good implementation practices.

MobileMedia. It is a program family that provides support to manage (create, delete, visualize, play, send) different types of media (photo, music and video) on mobile devices. During the SPL development and evolution, the initial core architecture was systematically enriched with mandatory, optional and alternative features. Seven releases of the MobileMedia were analyzed. The core features are: create/delete media (photo, music or video), label media, and view/play media. Some varying features, amongst others, are: transfer photo via SMS, count and sort media, copy media and set favorites. We decided to re-implement with advanced mechanisms all the variabilities, whenever it made sense, those ones that are implemented with aspects in the original AspectJ implementation.

III. ADVANCED PROGRAMMING TECHNIQUES

This section presents the advanced mechanisms used in this study (Sections A and B) and supported by the languages CaesarJ and Compose*, respectively. These languages were mainly chosen because they support significantly-different advanced mechanisms – combination of AOP and FOP (CaesarJ) and composition filters (Compose*), while having similar goals of improving module reuse and stability. In addition, their compiler implementations are open and proved to be robust enough during our pilot assessments. Equally important, there are also public reports of their successful adoption in industrial software development projects [17]. It is important to point out that, from now on, the word “modules” and “components” are treated as synonymous.

A. Feature Modularity: Virtual Classes &Mixin Composition

Besides the support for pointcut-advice mechanisms (also supported by AspectJ), CaesarJ [17] supports other advanced mechanisms that enable other ways of decomposing software modules. In particular, virtual classes and mixin composition [18] are prominent mechanisms of CaesarJ. The use of virtual classes makes it possible to apply overriding and late binding to inner classes in a similar way of late-bound of virtual methods. In fact, all inner classes within a class of CaesarJ are considered as virtual classes, because they are
handled similarly to virtual methods. This means that a class or a
group of classes can be redefined in any subclass of the
enclosing class. In other words, the name of the class can be
dynamically bound to multiple, different classes, depending on
the specific instance of the enclosing CaesarJ class. Once this
class has been defined as abstract, its implementation can be
done later or just partially. This way, virtual classes comprise
the capability to treat inner, nested classes, polymorphically.
Thus, in CaesarJ, a module can be created by composing
several CaesarJ classes. Using mixin composition, different
components can be composed to build more complex
components without compromising the independence of each
component. Mixin composition is the way by which CaesarJ
creates complete components out of various parts.

Figure 1 demonstrates an example of feature
modularization with virtual classes. The cclass
MediaManagement implements the base functionality of the
MobileMedia application. The new cclass PhotoSorting
refines cclass MediaData adding the functionality of sorting
photos by highest viewing frequency. The code slice illustrates a
refinement (a so-called further-binding) of the virtual classes
(box #1 – lines 03, 04 and 05) that are tangled by the feature
PHOTOSORTING. In such a further-binding, we can override
inherited methods, add new methods or new state, as well as
add additional super-interfaces and super-classes. The addition
of methods is illustrated in Figure 1 (box #2). In order to
realize the photo sorting feature, three new methods were
added to the cclass MediaData.

Since a feature may need functionality of multiple features
it is important to have relied on some form of multiple
inheritance. The mixin composition mechanism is used to
address this, through which a final component is generated.
The composition is realized with the operator ‘&’. In the listing
1, class MAlbumManagement composes PhotoSorting with
other features such as PhotoLabel. The composition operator
realizes a variant of multiple inheritance that linearizes the
superclasses, thereby avoiding ambiguities such as duplication
of inherited state.

class MPhotoManagement extends
MAlbumManagement & PhotoSorting & PhotoLabel
{ ... }  
Listing 1: Example of Composition in CaesarJ

B. Feature Modularity with Composition Filters

Compose*[16] aims at enhancing modularity and
composition of modules through the composition filters model
[7]. The idea is that objects can send messages between each
other, e.g. in the form of method calls or events. In the
composition filters model, these messages can be filtered using
a set of filters. Each filter has a filter type (e.g. Dispatch, Meta,
After, Before and Error), which defines the behavior that
should be executed if the filter accepts the message and the
behavior that should be executed if the filter rejects the
message.

In order to illustrate how Compose* works, the same
example of PHOTOSORTING is shown in Figure 2, in terms of
composition filters. The Figure illustrates the concept of
concern. A concern defines the filter modules (lines 2 to 09)
and the superimposition block (lines 10 to 15). The filter
module of our example contains an internal declaration (line
04) and one filter definition named sort (line 07). Internals are
the object instances that are instantiated for each instance of a
filter module. Because of this, internals can be used in
situations where each instance of a filter module must have its
own instance of an object, for example to hold a state or for
inheritance by delegation. Furthermore, the declaration of an
internal creates a composition relation between the object on
which this filter module is superimposed and an instance of the
type declared in the internal. Also, the internal can be used in
filter definitions as a destination for the message. The filter
definition on line 07 creates a filter that forwards a message to
a new destination. In this case, messages matching
BaseController.showImage are forwarded to the internal
destination (line 07). In other words, the after filter will forward all
messages sent to the object to the SortingUtil instance,
when the message matches a signature in the class
SortingUtil.

A message consists of a target and a selector:
target.selector. The target is the object that receives the
message and the selector is the called method. The selector
definition (line 12) selects a collection of program elements; in
this example, it selects all classes with the name
BaseController and MediaUtil. The filter module
PhotoSorting is superimposed on all program elements
selected by the selector sorts on line 12. The filter definitions
of a filter module form an advice. The points are determined by
the selectors and message matching in the filter definitions.

The concern definition in this example creates a
composition between the classes BaseController and
MediaUtil and the class SortingUtil (line 04). The
SortingUtil class implements the logic for a Sorting type.
Each BaseController and MediaUtil is associated with a
SortingUtil instance. The BaseController and MediaUtil
classes are extended to contain the methods
defined in the filters. The execution of these methods is
delegated to the SortingUtil class. Thus the interface of the
class SortingUtil has been extended in a way similar to
AspectJ’s inter-type declarations. As far as the Compose*
versions are concerned, the varying features are realized using
filters and the superimposition mechanism. By means of them,
the tangled code common to the PHOTOSORTING is now
modularised in a unique place (SortingUtil) and managed
by the filters.

IV. STUDY SETTING

This section presents the methods and hypotheses used in
this study (Section A) and the set of metrics applied to the SPL
application code (Section B).

A. Procedures and Hypotheses

Procedures. To begin with, all SPL releases were analyzed
according to a number of programming alignment rules in
order to assure equal compliance to coding styles and included
functionality. Moreover, the implementations followed the
same design decisions in all implementations to ensure a high degree of module reuse and maintainability. The reviews of designs and implementations with CaesarJ and Compose* were also performed by other independent researchers, who are also experienced on the use of feature-oriented programming and composition filters. All these initial procedures were carried out to ensure that the comparison between the implementations was equitable and fair. Inevitably, some refactorings in the AspectJ, CaesarJ and Compose* versions had to be performed when misalignments were observed at the implementation or even at the design level. When these misalignments were discovered, the particular versions were modified accordingly.

As a second and final step, the implementation modularity, reuse and stability of the target SPLs (Section II) was measured and compared. To begin with, we examined the fundamental modularity properties through the releases of both SPLs. Later, we analyzed the degree of reuse of the SPL modules as the target SPLs evolved. The idea was to verify if the technique that has promoted higher reuse degree would be the same that produced more stable SPLs. Thus, the central goal of our third analysis was verify the degree of stability of the SPL modules.

**Hypotheses.** The objective of our study is to understand if the aforementioned APTs provide better simultaneous satisfaction between reuse and stability than conventional AOP models. We decided to develop an exploratory (non-controlled) study rather than a controlled experiment, because our goal was first to reveal potential benefits and drawbacks of APTs and AspectJ. This exploratory analysis will be useful to derive further controlled studies in the future that investigate more specific hypotheses related to specific mechanisms of APTs. In addition, the use of the advanced mechanisms supported by these techniques is not yet as popular as those supported by AspectJ. The execution of an experiment would require the selection of several subjects with significant expertise in specific APTs, which is a non-trivial, if not impossible, task since they are not so popular and broadly adopted.

Even though our study is of exploratory notion, we decided to make our working general hypotheses explicit in order to expose our research questions in a clearer way. We have defined three hypotheses based on common claims found in the literature [2][3]. Table I presents the null (0) and alternative (1) forms for each hypothesis. The first hypothesis (H1) is defined to evaluate whether the reusable SPL modules are modified in different degrees when they are developed using APTs and AspectJ. The second hypothesis (H2) was defined to compare CaesarJ and Compose* in terms of the number of modifications that are required in reusable modules. Finally, our last hypothesis evaluates whether reusable and non-reusable modules are equally modified or not. The detailed discussion of these hypotheses is presented in Section V.
Hypothesis

**H1.0:** Reusable modules implemented with APTs and AspectJ are equally modified.

**H1.1:** Reusable modules implemented with APTs are modified less than those implemented using AspectJ.

**H2.0:** The use of CaesarJ leads to the same number of modifications in reusable modules when compared with Compose*.

**H2.1:** The use of CaesarJ leads to a higher number of modifications in reusable software modules when compared with Compose*.

**H3.0:** Reusable and non-reusable software modules implemented using APTs are equally modified.

**H3.1:** Reusable software modules implemented using APTs are modified less than non-reusable ones.

B. Modularity, Reuse and Stability Metrics

In order to support the test of our hypotheses (Table I), some previously-validated metrics for modularity, reuse, and stability were used. Therefore, three groups of metrics were used: (i) modularity metrics, (ii) reuse metrics, and (iii) change propagation metrics. These metrics were applied to multiple SPL releases with the intention of respectively computing: (i) the constancy of pivotal modularity properties through SPL releases, (ii) the degree of reuse of software modules, and (iii) the undesirable SPL modifications (i.e. ripple effects [2][3]).

Modularity Metrics. The modularity metrics [14] were used to enable us to analyze to what extent a certain modularity principle remained constant through the SPL evolution. It was also useful to compare the degree of modularity achieved by each advanced programming technique. The modularity metrics were used to quantify the following modularity properties: separation of concerns (SoC), cohesion, coupling, and code conciseness metrics. For instance, the concern metrics enabled to capture whether the localization degree of each feature was modularized in each SPL release. Concern metrics were used to quantify the concern diffusion at the level of components (metric CDC) and operations (metric CDO). They were also used to compute the level of tangling of a feature with other features (metric CDLOC). As we are focused here on using these metrics to quantify feature locality, they are also from herein called feature metrics. The attributes cohesion, coupling, and code size were measured with the following metrics: Coupling between Modules (CBC), Lack of Cohesion in Operations (LCO), and Lines of Code (LOC). It is important to highlight that, in the particular case of CaesarJ, we have considered each inner class as a different module as they can be instantiated apart. In Compose* was carried out is described in [19].

Stability Metrics. Change propagation metrics were used with the purpose of quantifying the degree of modifications on code implementation. For instance, they enable us to check if a change, originally targeted at adding a new feature, also affected the other features and/or core modules of a SPL. The used metrics were defined to quantify two complementary forms of modification: (i) refactorings – when the change is aimed at improving the system structure while preserving the existing code semantics, and (ii) alterations – when functionalities are added, removed, or modified through the system modules. For the first case, we used a metric, called Refactoring of Modules (RoM). This metric is used to quantify structural changes in classes, aspects, concerns, filters and/or in their respective internal elements. For the second case, a metric, named Alterations in Code Elements (ACE), was used to compute the number of increments, deletions, and actual modifications in code elements. Examples of these elements can be a class, a method, an aspect, an advice, a pointcut, or a filter.

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H1.0</strong></td>
<td>Reusable modules implemented with APTs and AspectJ are equally modified.</td>
</tr>
<tr>
<td><strong>H1.1</strong></td>
<td>Reusable modules implemented with APTs are modified less than those implemented using AspectJ.</td>
</tr>
<tr>
<td><strong>H2.0</strong></td>
<td>The use of CaesarJ leads to the same number of modifications in reusable modules when compared with Compose*.</td>
</tr>
<tr>
<td><strong>H2.1</strong></td>
<td>The use of CaesarJ leads to a higher number of modifications in reusable software modules when compared with Compose*.</td>
</tr>
<tr>
<td><strong>H3.0</strong></td>
<td>Reusable and non-reusable software modules implemented using APTs are equally modified.</td>
</tr>
<tr>
<td><strong>H3.1</strong></td>
<td>Reusable software modules implemented using APTs are modified less than non-reusable ones.</td>
</tr>
</tbody>
</table>

**B. Modularity, Reuse and Stability Metrics**

In order to support the test of our hypotheses (Table I), some previously-validated metrics for modularity, reuse, and stability were used. Therefore, three groups of metrics were used: (i) modularity metrics, (ii) reuse metrics, and (iii) change propagation metrics. These metrics were applied to multiple SPL releases with the intention of respectively computing: (i) the constancy of pivotal modularity properties through SPL releases, (ii) the degree of reuse of software modules, and (iii) the undesirable SPL modifications (i.e. ripple effects [2][3]).

Modularity Metrics. The modularity metrics [14] were used to enable us to analyze to what extent a certain modularity principle remained constant through the SPL evolution. It was also useful to compare the degree of modularity achieved by each advanced programming technique. The modularity metrics were used to quantify the following modularity properties: separation of concerns (SoC), cohesion, coupling, and code conciseness metrics. For instance, the concern metrics enabled to capture whether the localization degree of each feature was modularized in each SPL release. Concern metrics were used to quantify the concern diffusion at the level of components (metric CDC) and operations (metric CDO). They were also used to compute the level of tangling of a feature with other features (metric CDLOC). As we are focused here on using these metrics to quantify feature locality, they are also from herein called feature metrics. The attributes cohesion, coupling, and code size were measured with the following metrics: Coupling between Modules (CBC), Lack of Cohesion in Operations (LCO), and Lines of Code (LOC). It is important to highlight that, in the particular case of CaesarJ, we have considered each inner class as a different module as they can be instantiated apart. In Compose* was carried out is described in [19].

Stability Metrics. Change propagation metrics were used with the purpose of quantifying the degree of modifications on code implementation. For instance, they enable us to check if a change, originally targeted at adding a new feature, also affected the other features and/or core modules of a SPL. The used metrics were defined to quantify two complementary forms of modification: (i) refactorings – when the change is aimed at improving the system structure while preserving the existing code semantics, and (ii) alterations – when functionalities are added, removed, or modified through the system modules. For the first case, we used a metric, called Refactoring of Modules (RoM). This metric is used to quantify structural changes in classes, aspects, concerns, filters and/or in their respective internal elements. For the second case, a metric, named Alterations in Code Elements (ACE), was used to compute the number of increments, deletions, and actual modifications in code elements. Examples of these elements can be a class, a method, an aspect, an advice, a pointcut, or a filter.

**TABLE I. RESEARCH HYPOTHESES**

<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H1.0</strong></td>
<td>Reusable modules implemented with APTs and AspectJ are equally modified.</td>
</tr>
<tr>
<td><strong>H1.1</strong></td>
<td>Reusable modules implemented with APTs are modified less than those implemented using AspectJ.</td>
</tr>
<tr>
<td><strong>H2.0</strong></td>
<td>The use of CaesarJ leads to the same number of modifications in reusable modules when compared with Compose*.</td>
</tr>
<tr>
<td><strong>H2.1</strong></td>
<td>The use of CaesarJ leads to a higher number of modifications in reusable software modules when compared with Compose*.</td>
</tr>
<tr>
<td><strong>H3.0</strong></td>
<td>Reusable and non-reusable software modules implemented using APTs are equally modified.</td>
</tr>
<tr>
<td><strong>H3.1</strong></td>
<td>Reusable software modules implemented using APTs are modified less than non-reusable ones.</td>
</tr>
</tbody>
</table>

**B. Modularity, Reuse and Stability Metrics**

In order to support the test of our hypotheses (Table I), some previously-validated metrics for modularity, reuse, and stability were used. Therefore, three groups of metrics were used: (i) modularity metrics, (ii) reuse metrics, and (iii) change propagation metrics. These metrics were applied to multiple SPL releases with the intention of respectively computing: (i) the constancy of pivotal modularity properties through SPL releases, (ii) the degree of reuse of software modules, and (iii) the undesirable SPL modifications (i.e. ripple effects [2][3]).

Modularity Metrics. The modularity metrics [14] were used to enable us to analyze to what extent a certain modularity principle remained constant through the SPL evolution. It was also useful to compare the degree of modularity achieved by each advanced programming technique. The modularity metrics were used to quantify the following modularity properties: separation of concerns (SoC), cohesion, coupling, and code conciseness metrics. For instance, the concern metrics enabled to capture whether the localization degree of each feature was modularized in each SPL release. Concern metrics were used to quantify the concern diffusion at the level of components (metric CDC) and operations (metric CDO). They were also used to compute the level of tangling of a feature with other features (metric CDLOC). As we are focused here on using these metrics to quantify feature locality, they are also from herein called feature metrics. The attributes cohesion, coupling, and code size were measured with the following metrics: Coupling between Modules (CBC), Lack of Cohesion in Operations (LCO), and Lines of Code (LOC). It is important to highlight that, in the particular case of CaesarJ, we have considered each inner class as a different module as they can be instantiated apart. In Compose* was carried out is described in [19].

Stability Metrics. Change propagation metrics were used with the purpose of quantifying the degree of modifications on code implementation. For instance, they enable us to check if a change, originally targeted at adding a new feature, also affected the other features and/or core modules of a SPL. The used metrics were defined to quantify two complementary forms of modification: (i) refactorings – when the change is aimed at improving the system structure while preserving the existing code semantics, and (ii) alterations – when functionalities are added, removed, or modified through the system modules. For the first case, we used a metric, called Refactoring of Modules (RoM). This metric is used to quantify structural changes in classes, aspects, concerns, filters and/or in their respective internal elements. For the second case, a metric, named Alterations in Code Elements (ACE), was used to compute the number of increments, deletions, and actual modifications in code elements. Examples of these elements can be a class, a method, an aspect, an advice, a pointcut, or a filter.

**TABLE II. CODE ELEMENTS PER LANGUAGE**

<table>
<thead>
<tr>
<th>Language</th>
<th>Code Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ</td>
<td>classes, aspects, operations, pointcut and advice</td>
</tr>
<tr>
<td>CaesarJ</td>
<td>classes (java class and cclass), operations</td>
</tr>
<tr>
<td>Compose*</td>
<td>classes, concerns, operations and filters</td>
</tr>
</tbody>
</table>

**TABLE III. REUSE METRICS**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse Percentage (RP)</td>
<td>Ratio of the number of reused lines of code to the total number of lines of code.</td>
</tr>
<tr>
<td>Reuse Level (RL)</td>
<td>Ratio of the number of reused elements to the total number of code elements.</td>
</tr>
<tr>
<td>Reuse Ratio (RR)</td>
<td>Similar to Reuse Level, but also considers partially changed elements as reused.</td>
</tr>
</tbody>
</table>

We chose all these metrics, presented above, for several other reasons. First, the stability metrics had already been used and considered relevant in many other stability studies [1][19]. Second, the modularity metrics capture a number of internal program attributes, which have been empirically found to exert a direct impact on stability and reuse in software artifacts. Third, these internal attributes are more neutral and not specific to certain decomposition mechanisms (e.g. DIT – depth of inheritance – is a metric that should be avoided because it measures coupling in specific context of inheritance usage), thereby allowing us to achieve a more equitable comparison of AspectJ, CaesarJ and Compose*.
V. ANALYSIS OF EVOLVING SPLS

Program evolution is driven by regular updates of modules in order to accomplish the new requirements. The evolution process is extremely improved if software designs are modular, but, more importantly, also asks for few or any modifications of its reusable or concrete modules. Therefore, the evolution of the target SPLs (Section II) was analyzed under the perspective of three quality attributes: modularity, reusability and stability. Each of them is individually discussed in Sections A, B, and C, respectively. The discussion is supported by the metrics defined in Section IV.B. Then, we present a broader evaluation respectively. The discussion is supported by the metrics defined in Section IV.B. Then, we present a broader evaluation.

A. Modularity Analysis

As far as modularity is concerned, the use of APTs against AspectJ was evaluated based on the results of a metrics suite. The metrics were used to quantify four fundamental attributes of modularity, namely separation of concerns, coupling, cohesion and size (Section IV). Following a broad perspective, we have considered the modularity attributes of all the releases associated with both SPLs. We have observed that the advanced mechanisms of CaesarJ and Compose* led to more modular code than the implementations with AspectJ. The superiority ranged from 12% to 17%. In fact, for most of the quality attributes considered the advantage was approximately 15%. However, when we compare the modularity measures of Compose* and CaesarJ, the difference was almost insignificant. Also, for most of the modularity attributes, Compose* is in the lead with approximately 5% of advantage in relation to CaesarJ. In order to support our discussion, Figure 3 illustrates the results of modularity metrics for: (i) the release 6 of the MobileMedia system, and (ii) the release 4 of the iBatis system. Both releases were chosen because they suffered several changes and, at the same time, explored several opportunities of module reuse. Furthermore, the same measurement patterns were observed through all the release implementations. Hence, the results in these releases are also representative of measures observed in the other releases.

A careful analysis of measures and implementations seems to confirm that the superiority of CaesarJ and Compose* is thanks to the fact they address some expressiveness shortcomings of AspectJ. For instance, feature-oriented programming mechanisms of CaesarJ enable further modular decomposition of the base code beyond the use of OO mechanisms. Therefore, the opportunity for advising join points is significantly increased in CaesarJ code. Moreover, some of the AspectJ shortcomings were caused often by the stronger coupling between feature implementations (within aspects) and the base program. An example is the definition of a crosscutting feature by enumerating the join points by name or according to certain naming conventions in AspectJ. In Compose*, this problem is minimized since it offers a uniform programming model in which the feature decomposition is based on concerns. In CaesarJ, a new feature is a result of a polymorphic composition by means of virtual class and mixin composition. This means that there were fewer opportunities for pointcut fragility scenarios in CaesarJ. These two strategies are illustrated in Figures 1 and 2.

![Figure 3. Modularity Metrics for both SPLs](image)

Each inner class in CaesarJ was considered as a different module as they can be instantiated apart (Section IV.B). This decision led us to treat each inner class similarly to an aspect in AspectJ. Thus, when compared with Compose*, this kind of decomposition of SPL features makes the coupling among inner classes increase and the cohesion decrease. Figure 3 shows the measures that represent this issue. It is important to highlight that in terms of cohesion lower values indicates more cohesive. In Compose*, each filter usually corresponds to an aspect in the AspectJ and it is necessary to consider the class which implements some functionality of the filters. Thus, decomposition leads to almost the same number of modules (see Figure 3). However, the coupling occurs among a smaller number of modules as illustrated in Figure 3. Furthermore, Compose* supports specification of multiple inheritance in a single module by relying on the exclusive use of superimposition mechanisms. It is true that multiple inheritance in CaesarJ is successfully supported by mixin composition mechanisms. However, the main factor that contributes to the lower modularity is the existence of a higher number of virtual classes. This involves a high degree of coupling and cohesion thanks to the higher number of overridden methods.

With these observations, we come to the conclusion that similar modularity results are both achieved with Compose* and CaesarJ. Comparing all the measures, both techniques tend to exhibit almost the same level of modularity, which was clearly better than the level achieved with AspectJ. The difference among all the Compose* and CaesarJ measures does not exceed 5%; the exceptions are for the metrics LOC and the number of operations required to implement a feature (CDO), which presents a difference of 7% in favor of Compose*.

B. Reuse Analysis

A set of reuse metrics (Table III) was used in order to quantify what is the benefit on the use of APTs when compared with AspectJ. Basically, the reuse was measured in terms of absolute values of reusable lines of code, total number of lines of code, code elements without any modification and elements partially modified (Section IV). Our findings have evidenced
that the use of advanced mechanisms supported by CaesarJ and Compose* have led to a significant increase of reuse of code elements (Figure 4).

AspectJ’s reuse mechanisms are limited to abstract aspects and aspect inheritance. The limitation of AspectJ’s reuse mechanism is handled by CaesarJ by means of virtual class. For example, the method getNumberOfViews() is added to the class MediaData, providing total reuse of the cclass MediaData. This means that the reuse level (RL) is increased since, whenever the total of reused items is calculated, the class MediaData is considered. The addition of the method getNumberOfViews() is performed in AspectJ through intertype declarations (Figure 6). However, the aspect SortingAspect (Figure 6) is not considered as a reused element, since its body is modified by the definition of the method getNumberOfViews(). Thus, the reuse percentage (RP) for CaesarJ is increased. In Compose* the same method is superimposed on the class MediaData (Figure 2). However, only its total number of lines is considered as reusable since it was defined into another class (SortingUtil) and only this total is reused. Furthermore, the total of lines of code is higher since it was necessary to define a concern, namely Sorting, and the SortingUtil class. Figure 4 presents the reuse measures for both SPL applications. This way, the superiority of both CaesarJ and Compose* implementation in terms of reuse indicates that use of their advanced mechanisms increases the reusability of the code.

The superiority in terms of reusability supported by the CaesarJ’s advanced mechanisms has enabled a reuse percent (RP) of approximately 50%. More specifically, this high percent is a result of the high number of code elements reused (RL) since the addition of new features is provided by the overriding of classes and methods. If we take into consideration that the reused code elements also encompass the partially modified elements, this percent is to a certain extent improved. In this case, the reuse ratio (RR) is substantially increased since the number of partially modified elements is also considered. In CaesarJ, partial modification in code occurs mainly because the occurrence of rename operations. The method showImageList was partially changed in order to incorporate a rename operation. Figure 5 illustrates the code before the modification. The final code is illustrated in Figure 7.

![Figure 4. Reuse Metrics for both SPLs](image)

![Figure 5. Modularization of features with virtual classes](image)

To sum up, the APTs implementations provide better reuse for all the releases of both SPLs with a considerable advantage. However, between the two APTs, CaesarJ is in the lead with a large advantage of 22% and 32% in the worst and best cases when compared with the second placed (Compose*). When compared with AspectJ, the advantage of CaesarJ jumps to 27% and 42% in the worst and best cases.

C. Stability Analysis

The more changes are required to realize a new software evolution scenario, the more unstable the system design is likely to become. In this manner, we categorized the nature of changes that occurred more frequently in Compose*, CaesarJ and AspectJ implementations. Table IV summarizes the change classification observed in both MobileMedia and iBatis in terms of the stability metrics (Section IV). Table IV relies on the following notation pattern: AspectJ labeled “AJ”, CaesarJ labeled “CJ” and Compose* labeled “CS”.

Taking the Table IV into consideration, we can observe that the number of changes was lower in the CaesarJ and Compose* implementations when compared with AspectJ. For instance, when refactoring names of classes happens, all the
implementations suffer with the changes. However, some conventional mechanisms of AspectJ such as intertype declarations used by AspectJ requires more changes than superimposition mechanisms provided by Compose* and virtual classes of CaesarJ. In Figure 1 (line 26 – box #2) the method `getNumberOfViews()` was added to the class `MediaData` overriding it since the class `MediaData` is virtual. Realize that no modification was required. The same operation is illustrated in Figure 6 using intertype declarations. In this case, the aspect `SortingAspect` was modified and as this method cannot be overridden it will be constantly changed along the SPL evolution process.

```
01 public aspect SortingAspect{
02 ...
03 void MediaData.getNumberofViews{ }{}
04 ...
05 }
```

Figure 6. Reuse using intertype declarations

In Compose* the method `getNumberOfViews()` is defined in the class `SortingUtil` (Figure 2 – line 4) and superimposed on the class `MediaData` (Figure 2 – line 12). The advantage of the superimposition mechanism is that the method can be superimposed on other classes just including their names in the list of names (Figure 2 – line 12). However, any modification in this method needs to be realized directly in the class `SortingUtil`.

<table>
<thead>
<tr>
<th>Table IV. Changes in SPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changed Type</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>ACE</td>
</tr>
<tr>
<td>RoM</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

This superiority of CaesarJ takes place mainly due to the use of virtual classes (Section III), which promotes the evolution based on inheritance mechanism and thus many methods and cclasses are overridden instead of being intrusively partially modified. Figure 7 shows an example of this situation. Realize that the method `showMediaList()` was overridden by the addition of cclass `Favourite`, which is in charge of allowing users to specify and view their favourite photos. The same technique was used to add the other features across the SPL applications. In a nutshell, we can say that with the use of virtual classes the degree of stability was ameliorated 59% when compared with AspectJ. Even presenting improvements when compared to AspectJ, the degree of Compose* stability over the versions is lower than CaesarJ’s stability degree in almost 34%. In fact, a part from some code elements such as filters and concerns, Compose* works mainly in terms of the superimposition of methods. This way, a considerable number of modifications in classes are needed. On the other hand, classes in CaesarJ can be partially overridden.

In addition, in terms of reusable and non-reusable modules, our investigation showed that modules are 47% modified for CaesarJ implementation against 52% for Compose* one. From the total number of modification for CaesarJ, 37% of the modifications were done in reusable modules and 63% in non-reusable ones. In Compose* the difference between the reusable and non-reusable modifications represented only 4%. 52% of changes were performed in non-reusable modules against 48% in reusable ones. Our analysis of the changes revealed that the main reason for this result is that; (i) non-reusable modules have more external interfaces than the reusable ones for both APTs, and (ii) the use of advanced mechanisms, such as virtual classes, tends to maximize the rate of code reuse.

```
01 cclass MListControl {
02 ...
03 void showMediaList(. . .){ . . .}
04 ...
05 }
06
07 cclass Favourite extends MListControl{
08 ...
09 void showMediaList(. . .){
10 if (favorite) {
11 if (medias[i].isFavorite())
12 mediaList.append(. . .);
13 }
14 }
15 }
```

Figure 7. Slice of code partially modified in CaesarJ

D. Discussion: Modularity vs. Stability vs. Reuse

According to our findings, the use of APTs tends to produce SPLs with a high degree of reuse and stability. Table V summarizes the analysis. In the first column the quality attributes are listed, the worst and best percentage of superiority in relation to the second position is given in the second column and in the third column the name of the APT which presented superiority is given. As we can see, CaesarJ is the lead in terms of reuse and stability. However, in terms of modularity the measures benefit Compose* (Section A). The lower modularity of CaesarJ code, when compared to Compose*, are mainly due to (Section A): (i) the use of virtual classes as components that can be overridden by means of inheritance mechanisms, and (ii) the possibility of instantiating them apart are considered.

<table>
<thead>
<tr>
<th>Table V. Quality Attributes vs. Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality Attribute</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Modularity</td>
</tr>
<tr>
<td>Reuse</td>
</tr>
<tr>
<td>Stability</td>
</tr>
</tbody>
</table>

Figure 8 presents a noticeable trend involving the three quality attributes: reuse, stability and modularity. This trend applies for both SPL cases: iBatis (iB) and MobileMedia (MM). Modularity is expressed through the measures of coupling as it represents the behavior of the other modularity metrics as well. It is clear that when the degree of stability increases, the reuse also becomes more visible. These curves represent the general scenario for both target SPLs implemented in CaesarJ. As illustrated in Figure 8, the degree of modularity along the releases is almost the same. In the...
MobileMedia, the modularity was being improved as new features were been added. In iBatis, the behavior was a little bit different. The modularity decreases along the releases. However, the percentage of variation was less than 5% and thus it can be considered constant.

As far as our hypotheses (Section IV.A) are concerned, we are able to trace some conclusions. The first hypothesis (H1) was partially accepted. Whereas H1.0 was rejected, H1.1 was accepted. The use of APTs promotes benefits in terms of reuse and stability in different degrees as illustrate in Table IV and Figure 4. Figure 8 illustrates the behavior of each quality attribute. As mentioned before, when the stability of the product line decreases, the degree of reuse in general also decrease and vice-versa. This relationship between them becomes more evident since that the values of stability are associated with the reusable code elements even for APTs. In other words, the information associated with reuse in Figure 8 was calculated from the reusable elements. This indicates the interplay between them. This result is interesting as the relation of stability and reuse was investigated only in the context of OO programming techniques [1].

Our second hypothesis (H2) was also partially accepted. The use of CaesarJ produces reusable code elements more stable than Compose*. Thus, the hypothesis H2.0 was rejected whereas H2.1 was accepted. This takes place because what defined the degree of modification in reusable and non-reusable elements was the decomposition mechanisms used. This is an interesting result as modularity was not the major factor affecting the degree of stable reuse. All studies of APTs take for granted that modularity analysis is always enough for evaluating stability dimensions of source code. We observed here that the composition specificities played a major role in both stability and reuse. For instance, in terms of virtual classes the reused code is less modified (Section C). However, in the light of the Compose* implementation the non-reused elements were less modified since the reuse by means of composition filters occurs in terms of methods and filters only. In this case, for instance, the utility class (Figure 2 – line 04) which is part of a reusable element is constantly modified. This was presented in Table IV the superiority of CaesarJ was notable even in the worst case.

The use of APTs tends to produce non-reusable code elements less stable than reusable ones. This way our third null hypothesis (H3.0) was rejected and H3.1 was accepted. As the reusable modules were implemented though the APTs’ decomposition and these mechanisms provided a better degree of reuse (Section B), it was expected that non-reusable elements were less stable. It was possible to identify that CaesarJ implementations ask for few modifications in explicitly-reusable code units, such as abstract modules and interfaces. In terms of its compositions, the number of modifications was almost zero since the mechanism of mixin composition has a polymorphic power and was used to build each final composition.

Finally, we can conclude that the use of APT improves the simultaneous satisfaction between reuse and stability of SPL applications. In particular, it seems that the combination of FOP and AOP, as supported by CaesarJ, is more indicated than composition filters (Compose*) to implement SPL applications with higher degree of reuse and stability.

VI. RELATED WORK

This section will list the prior works and constraints that somehow have inspired this study or have guided it. The mechanisms provided by the CaesarJ language have been studied in the direction of SPL development [8][10][12]. However, these studies do not evaluate CaesarJ in terms of software stability and reuse. Yet in the domain of advanced mechanisms, Roo et. al [20] proposed the language Compose* on the top of composition filters studies. However, none of them (21)[22], similar to CaesarJ studies, was focused on stability of reusable code elements in SPLs.

Suri and Garg [13] have enumerated the various metrics of software to evaluate the reusability of modules. However, we have opted for selecting a specific group of reuse metrics so that we could perform an in-depth analysis. In addition, reuse and stability are two attributes of software quality that have not been studied together in the context of assessing advanced mechanisms. Recently, Mohagheghi et. al [23] described the results of an empirical study. In their study some hypotheses about the impact of reuse on defect-density and stability are assessed, using historical data on defects, modification rate, and software size of a large-scale telecom system developed by Ericsson. Similarly to our findings, they concluded that reused components are less modified (more stable) than non-reused ones between successive releases. However, their focus was on object-oriented systems. Our study compares different advanced mechanisms used for implementing and evolving SPLs. Besides, other studies evaluated SPL stability and reuse separately.

VII. THREATS TO VALIDITY

In our study, the conclusion validity threats are related to the implementation treatments. The versions were implemented by one AOP developer, who has a good knowledge on the programming languages involved in the study, namely AspectJ, CaesarJ, and Compose*. Even though the developer has not developed the AspectJ original versions of the SPLs, they had a partial knowledge of the possible modifications in future releases. However, we tried to minimize this threat by involving independent CaesarJ and Compose* researchers in order to check the quality of the design and code produced.

A threat to construct validity includes the suite of metrics used for quantifying changes, reuse and modularity properties. We used the concern metrics that allowed us to evaluate the modularity properties from each feature point of view. Additionally, coupling and cohesion metrics were used because...
they allowed us to evaluate the dependencies of the core/variable modules. We adopted these metrics because they were all empirically found to have correlation with design stability [1][4][19].

Threats to internal validity reside on alignment rules used to implement the CaesarJ and Compose\* versions. To reduce this threat, we performed a detailed analysis of the AspectJ code of the SPLs in order to reduce the inconsistencies in the pointcut interfaces and not propagating problems from the original AO implementations. It was necessary to ensure the quality of design in all versions and to do a fair and equitable comparison. Threats to external validity are conditions that allow results generalization. In order to minimize this threat, we chose a SPL that had already been used in another empirical study [1] and another well-known case study in the development of Java-based applications. These applications are representative and have a significant size. Besides, they contain a series of (non-) crosscutting mandatory, optional and alternative features. This way, they enabled us to observe the differences among the results. However, it is still necessary to conduct other evaluations with other SPLs to be able to provide more evidences regarding our conclusions.

VIII. FINAL CONSIDERATIONS

Gathering knowledge to identify which advanced modularity techniques achieve a better tradeoff of reuse and stability is particularly important due to many reasons. First, software engineers need to be better informed about which modularity mechanisms can maximize both stability and reuse of a system. Second, some of modularity mechanisms supported by these techniques, such as superimposition, inter-type declarations and virtual classes can be considered competitive. Finally, it is important to know which of these particular mechanisms tend to promote positive and negative effects over reusability and stability of software. In this context, this paper reported a comparative assessment of AspectJ, CaesarJ and Compose\* in the context of SPL development. Our analysis provided evidence that the synergistic use of AOP and FOP, as incarnated in CaesarJ, appears as a promising solution to implement software product lines. Our study confirms that the use of CaesarJ collaboration interfaces and virtual classes tends to increase the degree of reuse of methods.

We have also observed a number of new interesting outcomes as discussed through Section V. For instance, we found that modularity properties, as fostered by advanced programming techniques, do not seem to be the key factor to determine the degree of stable reuse of modules. According to our experience in both SPL cases, the different composition mechanisms exerted the main influence on the stable reuse superiority (or inferiority) of a technique. This justifies why Compose\* presented the best modularity results, but CaesarJ was superior in terms of reuse and stability. As future work, we believe that this initial work can be further improved in many directions, including (but not limited to): (i) the same investigation can be carried out in the context of other systems or using rigorous forms of empirical methods, such as controlled experiments, (ii) other attributes could be assessed such as defect density, and (iii) new evaluations can be performed taking into consideration the reuse and stability of both modules’ interfaces and their implementations.

REFERENCES