# Low-level Variability Support for Web-based Software Product Lines

Ivan do Carmo Machado[*]
Federal University of Bahia
Salvador, Brazil
ivanmachado@dcc.ufba.br

Alcemir Rodrigues Santos
Federal University of Bahia
Salvador, Brazil
alcemirsantos@dcc.ufba.br

Yguaratã Cerqueira Cavalcanti
Federal University of Pernambuco
Recife, Brazil
ycc@cin.ufpe.br

Eduardo Gomes Trzan
Recôncavo Institute of Technology
Salvador, Brazil
eduardo.trzan@reconcavo.org.br

Marcio Magalhães de Souza
Recôncavo Institute of Technology
Salvador, Brazil
marcio@reconcavo.org.br

Eduardo Santana de Almeida
Federal University of Bahia and
Fraunhofer Project Center for
Software and Systems Engineering
Salvador, Brazil
esa@dcc.ufba.br

## ABSTRACT

The Web systems domain has faced an increasing number of devices, browsers, and platforms to cope with, driving software systems to be more flexible to accomodate them. Software product line (SPL) engineering can be used as a strategy to implement systems capable of handling such a diversity. To this end, automated tool support is almost indispensable. However, current tool support gives more emphasis to modeling variability in the problem domain, over the support of variability at the solution domain. There is a need for mapping the variability between both abstraction levels, so as to determine what implementation impact a certain variability has. In this paper, we propose the *FeatureJS*, a FeatureIDE extension aiming at `Javascript` and `HTML` support for SPL engineering. The tool combines feature-oriented programming and preprocessors, as a strategy to map variability at source code with the variability modeled at a higher level of abstraction. We carried out a preliminary evaluation with an industrial project, aiming to characterize the capability of the tool to handle SPL engineering in the Web systems domain.

## Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*

## General Terms

Feature Oriented Software Development

[*]Corresponding author.

## Keywords

Software Product Line Engineering, Web Systems Domain, Feature Composition, FeatureIDE, Eclipse plugin.

## 1. INTRODUCTION

The increasing number of devices and platforms connected to the Internet has established a need for the Web systems development to make software flexible enough to accomodate a number of client-side specifications, without increasing development cost and time.

This is a scenario in which SPL engineering can be used as a viable software development strategy. SPL engineering consists of a systematic way to capture the common parts between products of a family of systems, while accomodating the differences. It results in establishing a reusable platform and a set of variable parts. The main benefit is a maintainable and comprehensible software asset base that can be reused, configured and extended, enabling the generation of families of program variants for a domain [13].

Feature-oriented software development (FOSD) provides SPL engineering with support for variability modeling and implementation. In FOSD, potential configurations emerge from the set of available features, organized under precise semantics and clear refinement concepts [19]. A feature is a unit of functionality that satisfies a requirement, and provides a potential product configuration option [1].

FOSD supports the automatic composition of features so as to generate products in an SPL, enabling the achievement of promised SPL benefits, earlier described in this Section.

A multitude of tools are readily available to help automating different methods and languages for FOSD[1]. They allow the modeling of features and their relationships in what is called a feature diagram [11], which represents the common and variable parts of a family of systems explicitly and comprehensively. The tools also enable the association of features and their source code, and provide automatic mechanisms to generate the product configuration.

Several languages and application domains can benefit from those tools. However, when implementing SPLs in not

[1]A list of tools for implementation in FOSD can be found at `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/fosd-tools/`.

so conventional domains, such as Web systems, we notice a gap between theory and pratice. An industry partner proposed the application of FOSD practices to develop an SPL project in the Web systems domain. A large portion of the client-side code must be implemented in `JavaScript`, to ensure the applications would run smoothly in a greater range of devices, browsers, and platforms. As this programming language has its particularities, existing tools could not provide the adequate variability support.

In this scenario, we proposed the *FeatureJS*, an Eclipse plugin aiming to accomodate the development of SPLs implemented in `JavaScript`. It extends the FeatureIDE, an Eclipse-based framework for feature-oriented programming [20]. Based on the observed evidence that, in practice, a feature does not map cleanly to an isolated module of code, but rather it may affect many artifacts of a software system [1], *FeatureJS* combines composition of features with preprocessor-based software development.

This proposal is in accordance with an important challenge for the field: the need of an unified and efficient framework to handle the combined use of composition and annotation-based approaches [1]. The use of annotation-based approaches in the context of FOSD has been subject of both *for* and *against* discussions [12]. As far as we know, just a few studies propose tools to support such a combination, as will be discussed later in Section 5.

The rest of this paper is organized as follows. Section 2 provides an overview of SPL engineering at the Web systems domain. Section 3 describes our proposed solution. Section 4 reports a preliminary study on the use of *FeatureJS*, in the context of our collaboration with industry. Section 5 discusses related work, and Section 6 concludes the paper outlining future directions for research.

## 2. SPL AT THE WEB SYSTEMS DOMAIN

The nature of web-based software systems has been impacted by an ever increasing diversity of devices, browsers, and platforms, and the set of different user behaviors and capabilities they should meet.

The dynamic behavior of features in the Web systems domain should be properly handled. A given functionality in a web system can behave differently depending on, e.g., accessibility or security requirements, and the system is expected to work properly. This issue may be even more severe in case a feature `A` should behave differently depending on the selection of either features `B` or `C`.

This situation matches scenarios for introducing SPL engineering. This may provide web-based development with the opportunity to move from a custom software development environment for building web products through systematic reuse of artifacts, by taking advantage of commonalities and predicted variability. Unless the business goals establish a limited audience for the developed web-based systems, SPL engineering can be considered as a suitable strategy to cope with the large amount of system variations.

However, as far as we know, the Web systems domain is not often considered as a potential base for developing SPLs, unlike domains such as embedded systems, or mobile applications. A look into the literature found a few research initiatives handling SPL in the Web systems domain [3][6][9][10][16][18][21]. They usually suggest modeling a SPL by employing conventional modeling techniques (e.g., UML), and tools (e.g. feature modeling support for conventional IDEs, including Visual Studio and Eclipse), or by adapting existing techniques, such as the use of model-driven development strategies in FOSD [21].

Those solutions, along with automated tools to handle SPL in other domains, are mostly concerned with modeling domain variability in a high-level abstraction, as a means to represent the common and variable features of products. While it can facilitate understanding how products can be composed, in terms of features, it is rather important to manage variability at implementation level, given that source code holds important role in establishing variable behavior.

In this effect, for SPL engineering to be effective in the Web systems domain, there is an emerging need of tools to support variability management at diverse abstraction levels, that could accommodate the particularities of different Web-based languages. They should cope with feature interaction, in a way to consider the dynamic behavior of features, as Web systems often provide dynamic adaptive content to support different platforms and devices.

## 3. PROPOSAL

In this section we introduce *FeatureJS*[2], a plugin for FeatureIDE [20], designed to provide automated support to SPL development in `JavaScript` and `HTML`, the most prominent client-side languages for the Web systems domain. Along this section we describe how *FeatureJS* works by illustrating its main functionalities in a motivating example, then we discuss its architecture.

### 3.1 Dealing with Feature Interactions

There has been considerable interest in investigating the complexity of feature interaction [2][5]. Feature interaction handles dependencies between different features in a way that features exclude or require other features. FeatureIDE allows the representation of constraints between features, controlled by the configuration view. In such a view, a configuration either enables or disables the selection of a given feature, according to the constraints associated to it.

For each concrete feature, FeatureIDE automatically creates a directory, called containment hierarchy [4], to store all code belonging to it. When a new product is to be configured, the generator picks all files from the directories associated to all corresponding features, and deploy the product variant in a reliable and efficient manner. In an ideal SPL, where there is a direct, one-to-one mapping between a problem domain variation and a variation point in the solution domain, this strategy would work seamlessly.

However, we should assume that feature interactions can also occur at implementation level, and a single feature can be mapped to multiple code fragments. FeatureIDE partially controls variability at implementation level, as follows. If a given file associated to a feature behave differently depending on the selection of an external feature, the way FeatureIDE handles such a case is to replace the entire file associated to that feature. For languages such as `Java`, *refinement declarations* [1] serve as a strategy to handle changes a feature makes to a program, without changing the core code, e.g., fields and methods can be added to a class, and those will be reached in a program variant only if the feature containing those refinements is selected. However, for languages such as `JavaScript`, that do not enable those declarations,

[2]The plugin can be found at `http://bit.ly/featurejs`

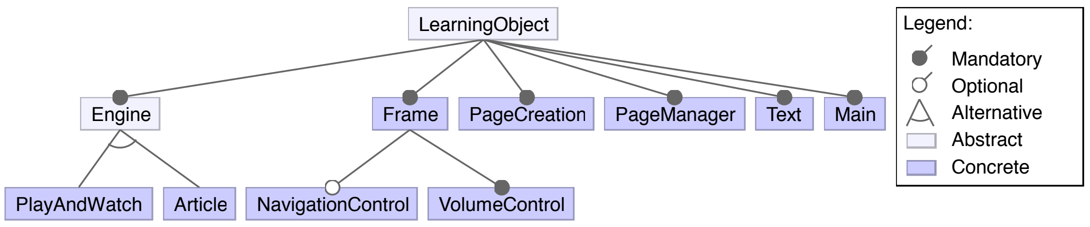

Figure 1: Excerpt from the *Learning Objects SPL* feature model.

applying such a technique to control inner-function variability would lead to a large amount of duplicate code.

Thus, we expanded FeatureIDE capabilities to integrate an annotation-based with the native composition-based approach, as a more general approach to enable variability management at implementation level. While the former enable inner-function statements behave differently, depending on the selection of a given feature, the latter handles the inclusion or exclusion of an entire function in a product variant. In this approach, we cope with *functional interactions*, subsuming interactions that can violate functional specifications [2]. We next illustrate how our proposal works.

Figure 1 shows a feature model excerpt, from the *Learning Object* SPL. It has a root feature called `LearningObject`, representing the domain under analysis, a set of mandatory features, including a *concrete feature* called `Frame`, and an *abstract feature* called `Engine`. The latter has two subfeatures, we placed as alternative features: `PlayAndWatch` and `Article`. The object engine can be set in either one of the two ways.

Let feature `Frame` has a containment hierarchy with a (set of) runnable file(s), which behave differently depending on the `Engine`'s subfeature selection. Listing 1 illustrates how *FeatureJS* deal with the use of preprocessor directives (annotation-based approach) to manage variability at implementation level, in this *Learning Object* SPL.

```
(function (window){
                              ⋮
// #ifdef PlayAndWatch
ref.clickForwardPage = function(event){
   if(_mode == _pageTypeEnum.PLAY || _mode ==
 _pageTypeEnum.PLAY_ANDROID || _mode ==
 _pageTypeEnum.PLAY_ANDROID_CHROME) { _page.
     animate();
   }
   else{
      _pageManager.forwardPage();
   }
}
// #elif Article
ref.clickForwardPage = function(event){
   _pageManager.forwardPage();
}
// #endif
                              ⋮
}(window))
```

Listing 1: Excerpt code from the *Frame.js* (feature `Frame`).

The directives in the source code delimit blocks of program that are compiled only if a specified condition is true. It enables modifying the behavior of a function or a statement, depending on the feature selected. Therefore, a product variant is generated by assembling the code fragments that correspond to a configuration. In this example, after binding the variants, the `ref.clickForwardPage` declaration statement will be set differently, depending on the selection of either feature `PlayAndWatch` or `Article`.

In cases where more than one product configuration includes the same *JavaScript* file, but depending on the feature selection a *function* behaves differently, the use of preprocessor directives may be employed to generate different product variants. The main reason is that composition rules for augmenting functions with new properties in *JavaScript* is not always safe [8]. In addition, this strategy may reduce maintenance effort, as business rules from a single function will be self-contained in a single file.

Besides controling and managing variability at both model and implementation levels, this approach might anticipate program-level customizations of core assets for a custom product to early in the development cycle.

## 3.2 Architecture

This section discusses the architecture and components of *FeatureJS*. Figure 2 shows a deployment view of *FeatureJS*, highlighting how the plugin relates to external entities. These are the components `de.ovgu.featureide.core` and `de.ovgu.featureide.fm.core`, from FeatureIDE, and `org.eclipse.vjet.core`, from VJET plugin[3]. This latter provides IDE capabilities to support JavaScript faster development, such as code completion, code templates, wizards, debug support, and native type and syntax checking, to identify errors through semantic validation.

Figure 3 shows how the packages and classes relate to each other, and how they relate to FeatureIDE core entities. *FeatureJS* uses `ComposerExtensionClass`, from package `de.ovgu.featureide.core.composers`, as this is the default composer implementation provided by FeatureIDE.

The *FeatureJS* encompasses the following classes:

**(i) FeatureJsCorePlugin.** This class links FeatureIDE to *FeatureJS* plugin by creating an activator class, allowing this latter to be managed by the FeatureIDE framework.

**(ii) FeatureJSComposer.** This is a **business rule class** that integrates FeatureIDE Wizard's information. It is responsible for retaining variability information for further treatments. This works by screening each file, within the allowed extensions, to treat the preprocessor directives. It also creates the containment hierarchies, and handles the software configuration build performance.

**(iii) FeatureJSModelBuilder.** This class is responsible for traversing each feature and the associated files, displaying them in FeatureIDE's `FSTModel`, which represents the project structure.

[3]`http://eclipse.org/vjet/`

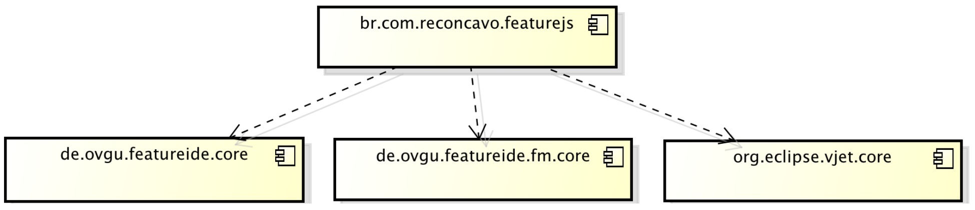

Figure 2: FeatureJS deployment view.

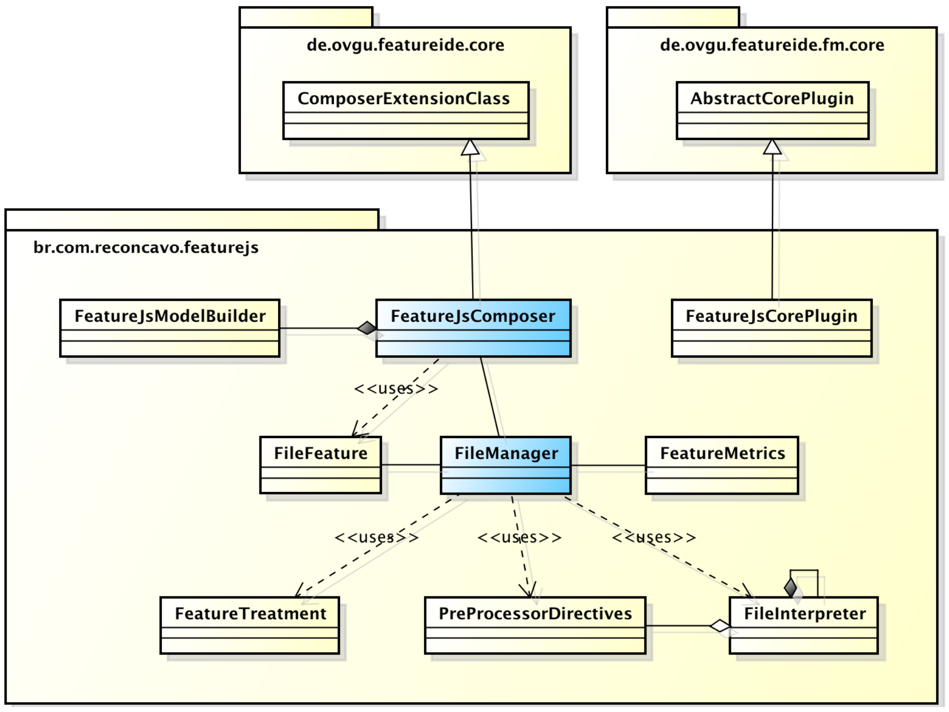

Figure 3: FeatureJS package and class diagram.

**(iv) FileManager.** This class manages files that have been copied to remove the non-selected features, keeping only the selected ones. It is a final class which implements the singleton pattern, hence it cannot be neither inherited nor instantiated. We found it to be a safe approach to avoid likely concurrency problems. As a **business rule class**, it defines the precedence logic for each preprocessor directive, and the replacement policy. Once a directive's block code contains an error, this class handles the error, maintaining the code portion with deviation in a product variant. Meanwhile, an exception is thrown to report the occurrence.

**(v) FileFeature.** This class is responsible for identifying the feature interaction at implementation level, namely the files and features affected by other features. It retrieves all feature interaction occurences.

**(vi) FileInterpreter.** This class is an abstraction of a code fragment framed by a directive. It implements a logical Doubly Linked List structured, as follows:

- The previous `FileInterpreter` is considered as a parent and known by the attribute `parentFileInterpreter`.
- The next `FileInterpreter` is considered as a child and known by the attribute `childFileInterpreter`.
- It has a pattern for the code fragment framed by a directives by the attribute `originalCode`, and its replacement; if it is needed, it is defined by the attribute `innerCode`.

**(vii) FeatureTreatment.** This is a final and non-instantiable class that encapsulates the treatment of each feature, by generating a specific regular expression pattern to recognize a directive. This is a utilitarian class, only accessible by its static methods.

**(viii) PreProcessorDirectives.** This `Enum` represents all available preprocessor directives that are considered during the feature treatment phase in a file. Five directives are available to control conditional compilation, as follows: `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`.

**(ix) FeatureMetrics.** This class gather metrics for measuring the complexity of the SPL. As earlier discussed, *FeatureJS* is concerned with mapping variability from problem domain to solution domain. Thus, this class enables the identification of source files containing preprocessor directives. Besides, it also provides a view to list which files are affected by each feature. It is useful for maintenance purposes, as it is possible to track files that need modification when a feature is included, excluded, or modified.

## 4. PRELIMINARY EVALUATION

In this section we describe the subject of the study, and present the early results about the feasibility of using *FeatureJS* to develop Web-based SPLs.

### 4.1 Domain Problem

The advent of Web-based digital interactive technologies has led teaching and learning methodologies to a next instructional setting level. The goal is to use such technologies to stimulate the learners' knowledge formation and retention. This instructional technology concept is known as "learning object". Learning objects are generally understood to be digital entities deliverable over the Internet, making them accessible and usable by multiple users in parallel [7]. They have a great potential for reusability, generativity, adaptability, and scalability. This principle is based upon the idea that a course or lesson can be built from reusable instructional components which can be built separately but modified to the user's needs [22].

In this scenario, our industry partner, *Recôncavo Institute of Technology*[4], based on Salvador, Brazil, has developed a series of learning objects, intended for K-12 education, as a subcontractor for one of the leading provider of online educational content to K-12 schools in Brazil.

The previous stage to adopting an SPL approach was that applications were usually developed one at a time. Applications were implemented in the `ActionScript` language, until they decided to turn their applications platform-independent. As of this point, new applications would be implemented in `HTML5` - mainly `JavaScript` and `HTML` -, so that their applications could reach a greater number of customers, due to the cross-platform capabilities of these technologies.

As reuse was merely opportunistic in their development cycle, the problem with cost and scale was imminent. Since opportunistic reuse can be more expensive than a systematic reuse of software artifacts, our partnership enabled their software development process to transition to an SPL approach. Hence, the SPL selected for this study is part of this project, in the Web-based learning systems domain.

### 4.2 Feasibility Study

The project, called `MDC Learning Objects`, comprises a set of `42` features. The core features has, together, around `3.7` KLOC. The `MDC` project has `23` boolean configuration variables and can, in theory, be deployed in over `3800` different configurations. However, it is worth saying that such number is not realistic, due to a set of very specialized requirements for each individual learning object, which demands concrete features to be implemented and selected prior to delivering a product configuration. That is, we can generate thousands of different configurations, but for a single product variant to run properly, a series of amendments should be implemented, so as to match product-specific requirements. Those features mainly include the management of metadata, such as the media scripts, particular to every single learning object, and as such must be shared with other objects at all.

Thus, for this particular case study we consider three different products, fully functional, generated from the core asset base. Due to the mutual confidentiality and non-disclosure agreement, we cannot describe the applications' name, and some features' name are also omitted. Thus, we will herein call the product variants as APP1, APP2, and APP3. Tables 1 and 2 show data about our target SPL. The former shows code metrics extracted from each product[5], such as LOC (Lines Of Code), number of files, functions, number of declarative (which name a variable, constant, or procedure, and can also specify a data type) and executable statements (which initiate actions). The latter shows the product configurations, highlighting the variable features selected for each product. There are an additional `19` mandatory features, that we do not list here for the sake of space.

Table 1: Products metrics generated from the SPL.

| | LOC | Files | Functions | DS | ES |
|---|---|---|---|---|---|
| Core | 3,778 | 47 | 421 | 796 | 2,003 |
| APP1 | 5,568 | 62 | 510 | 972 | 3,243 |
| APP2 | 5,188 | 61 | 518 | 964 | 3,039 |
| APP3 | 6,520 | 63 | 514 | 978 | 4,027 |

DS: Declarative Statements, ES: Executable Statements.

Table 2: Variant configuration Matrix.

| # | MDC Features | APP1 | APP2 | APP3 |
|---|---|---|---|---|
| 1 | PageCreation | ✔ | ✔ | ✔ |
| 2 | WatchPage | ✔ | | ✔ |
| 3 | PlayPage | ✔ | | ✔ |
| 4 | ArticlePage | | ✔ | |
| 5 | MatchColsTask | | ✔ | |
| 6 | FillInTask | | ✔ | |
| 7 | SubtitleManager | ✔ | ✔ | ✔ |
| 8 | VideoManager | | | |
| 9 | AnimationManager | ✔ | ✔ | ✔ |
| 10 | AudioManager | ✔ | ✔ | ✔ |
| 11 | NavigationControl | ✔ | ✔ | ✔ |
| 12 | Article | | ✔ | |
| 13 | BP | | ✔ | |
| 14 | PlayAndWatch | ✔ | | ✔ |
| 15 | ACJC | ✔ | | |
| 16 | AGR | | | ✔ |
| 17 | Animations | ✔ | ✔ | ✔ |
| 18 | Background | ✔ | ✔ | ✔ |
| 19 | Buttons | ✔ | ✔ | ✔ |
| 20 | Environment | ✔ | | ✔ |
| 21 | Locutions | ✔ | ✔ | ✔ |
| 22 | Music | ✔ | | ✔ |
| 23 | Effects | ✔ | ✔ | ✔ |

✔: Selected feature.

The `MDC` project employed a reactive SPL approach [15], in which a single product is subsumed into an SPL. In this approach, not all possible variations are implemented beforehand, but instead only those variations needed in current products are implemented, in an incremental fashion. Figure 4 illustrates the process, which is explained next.

The *APP1* was the first application to be implemented. Based on this first application, *APP2* was implemented.

[4]`http://www.reconcavotecnologia.org.br/`

[5]Code metrics gathered with the *Understand* tool, available at `http://www.scitools.com/download/`

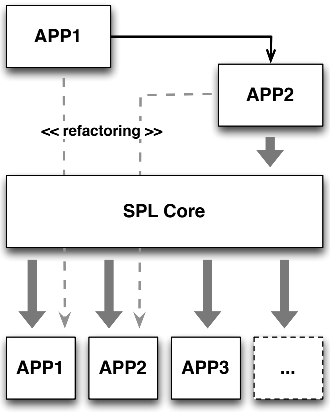

Figure 4: Reactive SPL process adopted.

Table 3: Development time.

| Application | Dev. Time |
|---|---|
| APP1 | 720 engineer-hours |
| APP2 + SPL Core | 448 engineer-hours |
| APP1 Refactoring | 160 engineer-hours |
| APP3 | 122 engineer-hours |

Next, by analyzing existing assets, and defining the SPL commonalities and variabilities, it was possible to define the *SPL Core*. It was necessary to refactor part of *APP1* into the core architecture. Then, the following application could be developed, by systematically reusing the core, and integrating the product-specific parts.

At the end of this evaluation, the project MDC deployed three different products, sharing parts of the implementation. Table 3 shows the time spent in the MDC SPL implementation, comprising the implementation of both core and products-specific parts.

The initial development of the *APP1* took 45 working days of two software engineers, working around *8* hours a day each on this project, for a total of around 720 engineer-hours. The work in this first application comprised tasks such as domain analysis, design, and implementation of the application, identifying opportunities for reuse, and customer validation of the implemented features.

Next, developers took 13 working days, for a total of 208 engineer-hours, to build the *APP2*. In order to identify reuse opportunities in both applications, and systematize what could be leveraged as both common and variable parts, it took an another 15 working days. A total of 240 engineer-hours was employed to build the *SPL Core*, turning the project into a reusable platform.

After establishing the *SPL Core*, it took an additional *ten* working days to refactor the *APP1* for performance increases, and to acommodate changes in code so as to make the applications run smoothly on the Android platform, a requirement that was identified during *APP2* implementation. This task took a total of 160 working hours.

*APP3* was developed within a 7-day period, and took developers about 122 engineer-hours.

*FeatureJS Metrics* provides a textual output, comprising the metrics associated to the feature interaction. It describes how features interact with each other, and shows the files affected by preprocessors directives. Listing 2 shows a sample output. Figure 5 shows a graphical representation to better explain the concept. As mentioned in Section 3.2, as a single feature can be mapped to multiple code fragments, *FeatureJS Metrics* identifies files with preprocessor directives, as a means to analyze the impact in case of modifications, and easily locate the concrete features (containment hierarchy) each file comes from.

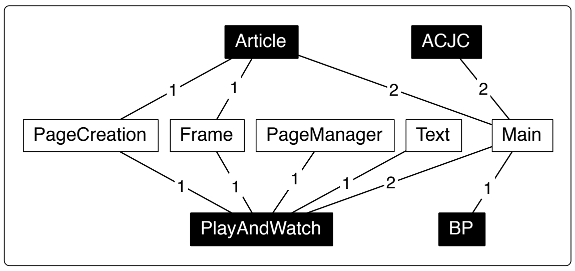

Figure 5: Feature interaction at the *Learning Objects* SPL.

```
Article
- PageCreation: 1 (PageFactory.js),
- Main: 2 (Main.js, index.html),
- Frame: 1 (Frame.js),
- Text: 1 (FabricaTextoLegenda.js);

BP
- Main: 1 (index.html).

ACJC
- Main: 2 (Main.js, index.html);

PlayAndWatch
- PageCreation: 1 (PageFactory.js),
- PageManager: 1 (PageManager.js),
- Frame: 1 (Frame),
- Main: 2 (Main.js, index.html),
- Text: 1 (SubtitleFactory.js).
```

Listing 2: *FeatureJS* feature interaction view.

In Figure 5, the black boxes represent the features that are also used as directives in the source code, while the white boxes represent the concrete features containing files with those directives. The numbers in the edges represent the amount of files affected by the directives, between any two features. Notice that not only variable features (*PageCreation*, #1 in Table 2), but also mandatory features (*Frame, PageManager, Text*, and *Main*) can have their source code partly modified by an external behavior demand. The feature *Frame* interacts with both *Article* and *PlayAndWatch* features in a single file. Listing 1, earlier presented in Section 3, shows an excerpt code from *Frame.js*, a file from the feature *Frame*. The code excerpt demonstrates how this file can behave differently, depending on which feature is selected.

## 4.3 Discussion

This preliminary evaluation has suggested that the *FeatureJS* plugin may positively impact both the development cost and time, and the maintainability of the overall SPL, in the Web systems domain.

Our industry partner reported gains in development time, what might result in order of magnitude cost reductions in next products' releases. By looking at Table 3, we may observe a reduction in the development time employed in the third product, although it is larger in size than preceding ones, as listed in Table 1. As the core platform was well-established, the time demanded was mainly dedicated to build the product-specific parts.

We cannot neglect the time dedicated for transitioning from a single-systems development to SPL engineering. However, as discussions on this topic are out of scope in this paper, we assume the time spent in the project comprises only the actual SPL implementation, ensuring that the involved engineers had a good theoretical and practical knowledge on SPL engineering, as a set of SPL training and practice classes was previously held.

The capability of the plugin to manage variability at implementation level, by the blend of annotation and feature composition, is way more significant than simply handling variability at modeling level. Especially in the development with `JavaScript` and `HTML`, in which composition rules are not robust enough, counting solely with *mixin* operations to augment functions properties is not either safe nor cost-effective. The use of preprocessor directives also enables inner-function statements to behave differently depending on the feature selected for a given product configuration. Thus, the feature-oriented approach can provide the adequate support for changing an entire function to a final product.

The use of *FeatureJS* may serve as a useful resource to foster SPL development in the Web systems domain. The combined approach *FeatureJS* encompasses enables the systematic reuse of code between products. Besides, by mapping features and preprocessor directives, the plugin unveils the set of files affected by them, which in turn might improve maintenance tasks.

As a counterpoint to the large-scale use of this approach, the plugin still needs further performance improvements. When the *FeatureJS Metrics* class was added to the plugin, the time to generate the configurations was substantially higher. Using a 2.5GHz Intel Core i5 processor, with 4GB 1600 MHz DDR3 memory module, it took `178.28s` to analyze the metrics associated to the MDC SPL Project. Without the analyses, the plugin took only around `18s`.

As the main threat to the validity of this study, the limited and constrained evidence we have at our disposal prevent us from drawing general conclusions on the results. The main gathered data refers to the development time as a function of the product variant size. It is an important area of concern. Given that no previous data, to serve as baseline values, was available, the reduction in development time for the n-ary variant releases might be caused by a maturation effect.

Further studies could contribute to gather more data on the feasibility of our approach, in order to analyze whether *FeatureJS* may also prove effective in larger scenarios.

## 5. RELATED WORK

We need to be aware that the use of preprocessors deviates from the core purpose of feature-oriented programming. Nevertheless, previous work also propose strategies handling such a combination of approches, as it is proven to be a simple but effective means to implement variability [14]. We next discuss each study we found as related to our proposal.

Prehofer [19] treated the feature interaction issue with *lifters*. A *lifter* defines a modular means to implement the feature interaction. Liu et al. [17] proposes refactoring in legacy application through *derivatives*, extending the *lifter* notion, to produce a feature-oriented refactoring of object-oriented systems.

We also found some studies dealing with the composition of Web systems, by using strategies such as XML-based, feature-oriented programming, and a mix of FOSD and model-driven development. They are discussed next.

Pettersson and Jarzabek [18] used an XML-based Variant Configuration Language to turn a Web portal into a more flexible architecture to reap the benefits of new business opportunities that required rapid development and further maintenance. However, developers had to manage multiple languages (XVCL, ASP, HTML) without specific tool support, which may impact productivity and maintenability. Additionally, while runtime debugging the webportals this approach forces developers to remember the several mappings used, increasing complexity.

Trujillo *et.al.* [21] blended FOSD and model-driven development (FOMDD) to optimize the synthesis of portlets in Web portals. A benefit of FOMDD is that it is mathematically based, and this makes connections with category theory easier to recognize. On the other side, FOMDD requires additional effort to manage model and perform all transformations needed, which without the proper tools can imply in high costs and be error-prone.

Capilla and Dueñas [6] presented a light-weight SPL architecture to control the evolution and maintenance of new web products and facilitates the maintenance operations on web sites. Authors advocate that such an approach reduces the development costs, and the benefits of the SPL engineering can be noticed earlier. The research counted on data from an initial analysis of two web sites.

All these studies propose strategies to handle Web-based SPLs to a certain extent. Nevertheless, they also focus rather on modeling aspects. By contrast, we considered a lower level of abstraction, while proposing a strategy to cope with variability at implementation level.

## 6. CONCLUDING REMARKS

This paper presents *FeatureJS*, an Eclipse plugin designed to asssist in the development of SPLs in the Web systems domain. It supports coding in `JavaScript` and `HTML`, the most prominent language used for Web systems client-side coding.

*FeatureJS* extends FeatureIDE, an open source platform for feature-oriented software development, by adding the use of an annotation-based approach in conjunction with the native composition-based approach. It enables the mapping between problem domain features and features on the implementation level.

For assessing the feasibility of this approach, we performed a preliminary evaluation within an SPL project from an industry partner in the Web systems domain. A reactive SPL approach was employed in this project, and three fully functional products were generated. The combined annotation and composition-based strategy allows feature composition in a range of combinations. This is rather important in this domain, as products are expected to work in diverse devices, browsers, and platforms smoothly, and it is usually necessary to modify the internal behavior of a feature so as to meet such diversity.

We are not aware of other studies dealing with SPL development in the Web systems domain, that cope with the mix of feature orientation with preprocessors. The preliminary study showed potential for *FeatureJS* to cope with this issue. We believe this plugin can lead to reduce development costs, either by promoting effective reuse and/or reducing the development time. However, we understand that further empirical studies are needed, in order to assess its potential for large-scale use.

## 7. ACKNOWLEDGMENTS

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[6]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08 and CNPq grants 305968/2010-6, 559997/2010-8, 474766/2010-1 and FAPESB.

## References

[1] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, jul 2009.

[2] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring feature interactions in the wild: The new feature-interaction challenge. In *Proceedings of 5th International Workshop on Feature-Oriented Software Development*, FOSD'13, pages 1–8, 2013.

[3] L. Balzerani, D. D. Ruscio, A. Pierantonio, and G. De Angelis. A product line architecture for web applications. In *Proc. of the ACM Symposium on Applied Computing*, SAC, pages 1689–1693. ACM, 2005.

[4] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[5] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, jan 2003.

[6] R. Capilla and J. Duenas. Light-weight product-lines for evolution and maintenance of web sites. In *Proc. of the 7th European Conference on Software Maintenance and Reengineering*, CSMR, pages 53–62, 2003.

[7] L. T. S. Committee. IEEE standard for learning object metadata. Technical report, IEEE, 2002.

[8] T. V. Cutsem and M. S. Miller. Robust trait composition for javascript. *Science of Computer Programming*, 2012. In Press, Corrected Proof.

[9] M. C. Felice, J. B. F. Filho, M. Acher, A. Blouin, and O. Barais. Interactive visualisation of products in online configurators: A case study for variability modelling technologies. In *Proc. of the MAPLE/SCALE Joint Workshop*, 2013.

[10] H. Gomaa and M. Gianturco. Domain modeling for world wide web based software product lines with uml. In *7th International Conference on Software Reuse*, ICSR, pages 78–92. Springer-Verlag, 2002.

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, nov 1990.

[12] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE, pages 311–320. ACM, 2008.

[13] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14, 2012.

[14] C. Kästner, P. G. Giarrusso, and K. Ostermann. Partial preprocessing c code for variability analysis. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS, pages 127–136. ACM, 2011.

[15] C. W. Krueger. Easing the transition to software mass customization. In *4th International Workshop on Software Product-Family Engineering*, PFE, pages 282–293. Springer-Verlag, 2001.

[16] M. Laguna, B. González-Baixauli, and C. Hernández. Product line development of web systems with conventional tools. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Web Engineering*, volume 5648 of *Lecture Notes in Computer Science*, pages 205–212. Springer Berlin Heidelberg, 2009.

[17] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proceedings of the 28th international conference on Software engineering*, ICSE, pages 112–121. ACM, 2006.

[18] U. Pettersson and S. Jarzabek. Industrial experience with building a web portal product line using a lightweight, reactive approach. *SIGSOFT Software Engineering Notes*, 30(5):326–335, sep 2005.

[19] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, ECOOP, pages 419–443. Springer Berlin Heidelberg, 1997.

[20] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[21] S. Trujillo, D. Batory, and O. Diaz. Feature oriented model driven development: A case study for portlets. In *Proc. of the 29th International Conference on Software Engineering*, ICSE, pages 44–53. IEEE Computer Society, 2007.

[22] D. A. Wiley. Connecting learning objects to instructional design theory: A definition, a metaphor, and a taxonomy. In *The Instructional Use of Learning Objects*. Online Version, 2000.

[6]INES - http://www.ines.org.br