RiPLE-HC: JavaScript Systems Meets SPL Composition

Alcemir Rodrigues Santos*, Ivan do Carmo Machado, Eduardo Santana de Almeida
Reuse in Software Engineering Laboratory – RiSELabs
Federal University of Bahia – UFBA
Salvador, Brazil
{alcemirsantos, ivanmachado, esa}@dcc.ufba.br

ABSTRACT

Context. Software Product Lines (SPL) engineering is increasingly being applied to handle variability in industrial software systems. *Problem.* The research community has pointed out a series of benefits which modularity brings to software composition, a key aspect in SPL engineering. However, in practice, the reuse in `Javascript`-based systems relies on the use of package managers (*e.g.*, `npm`, `jam`, `bower`, `requireJS`), but these approaches do not allow the management of project features. *Method.* This paper presents the RiPLE-HC, a strategy aimed at blending *compositional* and *annotative* approaches to implement variability in `Javascript`-based systems. *Results.* We applied the approach in an industrial environment and conducted an academic case study with six open-source systems to evaluate its robustness and scalability. Additionally, we carried a controlled experiment to analyze the impact of the RiPLE-HC code organization on the feature location maintenance tasks. *Conclusion.* The empirical evaluations yielded evidence of reduced effort in feature location, and positive benefits when introducing systematic reuse aspects in `Javascript`-based systems.

Categories and Subject Descriptors

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

Keywords

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

*Corresponding author.

SPLC '16, September 16-23, 2016, Beijing, China

DOI: http://dx.doi.org/10.1145/2934466.2934486

1. INTRODUCTION

Since the early stages to establish the SPL engineering, several work proposed means to improve source code modularity [2, 3, 4]. The research community has tried to demonstrate modularity as a prominent strategy to mitigate known preprocessor-based implementation issues, such as the increased occurrence of crosscutting concerns and code obfuscation [12, 22]. Nevertheless, conditional compilation has been the widely accepted strategy to implement variability in software systems, despite its proven drawbacks [15]. However, both the ease of use and flexibility `#ifdef` annotations provide, together with usually available robust tool support, it is possible to develop variable systems sufficiently sheltered from inconsistences, even in large software systems like the Linux kernel [19]. Thus, in order to accommodate the benefits of both compositional and annotative approaches to implement variability, the so-called hybrid approaches have emerged [2, 3]. They usually avoid the introduction of new elements – usually unknown – in the existing programming languages, which may ease its adoption.

At the same time, the ever increasing use of `JavaScript` for the implementation of the large software systems imply to deal with a higher complexity. Such fact raises many kinds of challenges, *e.g.*, regarding modularization. Therefore, in order to cope with such increased complexity, software engineers need to resort of external constructs (apart from language-native ones) to achieve reasonable modularization, such as package managers (*e.g.*, `npm`, `jam`, `bower`), dependencies managers (*e.g.*, `requireJS`).

In order to address the new demands of the development of `JavaScript`-based software systems, we proposed tool support for SPL engineering, which allows the introduction of variability in the implementation and imposes feature-oriented code organization of such systems. The approach was first evaluated in the Web domain in conjunction with an industrial partner [14]. Hence, in this present investigation, we elaborate on our preceding proposal to establish the concept of a hybrid composition strategy for SPL engineering, named RiPLE-HC. To the best of our knowledge, no study – apart from our preliminary study [14] – addressed the systematic reuse from modeling stages to low-level variability implementation. In addition, the initial tool was extended to accommodate annotation scattering visualizations based on gathered evidence of the impact of code organization in feature location tasks, discussed later on in this paper.

The contribution of this work is threefold: *(i)* the detailed description of the RiPLE-HC approach to manage variabi-

lity at low-level variability implementation; *(ii)* evidence of the support of novel RiPLE-HC constructs on the maintainability task of feature location; and *(iii)* case studies[1] to gather empirical evidence regarding the approach feasibility and robustness.

The remainder of this paper is organized as follows. Section 2 describes the RiPLE-HC hybrid composition solution. Section 3 discusses the conducted case studies, both industrial and academic ones. Section 4 reports on the planning, execution and results of the controlled experiment. Section 5 discusses related work. Finally, Section 6 concludes the paper and pinpoint directions to further investigation.

2. RIPLE-HC: HYBRID JAVASCRIPT SPL ENGINEERING

The RiPLE-HC implements a strategy to handle variability at both feature modeling and code level for `JavaScript`-based systems. It encourages the use of the feature-based code organization and allows the use of preprocessing annotations for handling fine-grained variability.

Along this section we introduce the concepts and methods underlying the RiPLE-HC strategy. Next, we describe the main features of the proposed tool, and highlight the role a hybrid approach plays to improve code modularity.

2.1 Motivation

`JavaScript`-based systems can be found in different platforms and such programming language is not only used to implement Web-based systems, *e.g.*, Brackets is a powerful general purpose text editor implemented in `JavaScript`[2]. At the same time, the complexity of `JavaScript`-based software systems is increasing and a significant amount of complexity comes from handling the dynamic behavior of their features, which sometimes depend on the presence or absence of another feature.

This ever increasing complexity scenario satisfies SPL engineering key characteristics, as it may provides `JavaScript`-based systems with the opportunity to move from a custom software development approach to build a set of products and assembling reusable modules, in a systematic and coordinated fashion. Unless the business goals establish a limited audience for the developed systems, SPL engineering can be considered as a suitable strategy to cope with the large amount of system variations and complexity [11].

Research effort concerning the introduction of SPL engineering in the Web systems domain can be found elsewhere [6, 16, 24]. However, they are mostly concerned with modeling domain variability in a high-level abstraction, as a means to represent the common and variable features. While it can facilitate the understanding of how products can be composed, in terms of features, it is rather important to manage variability in both, coarse and fine-grained implementation levels, given that source code holds important role in establishing variable behavior. Therefore, we proposed an approach to handle `JavaScript`-based systems variability at code level [14]. The approach aims to promote the modular and systematic reuse of artifacts in a feature-oriented fashion.

[1]Raw data is available at: http://goo.gl/8U75wA

[2]Available at: http://brackets.io/

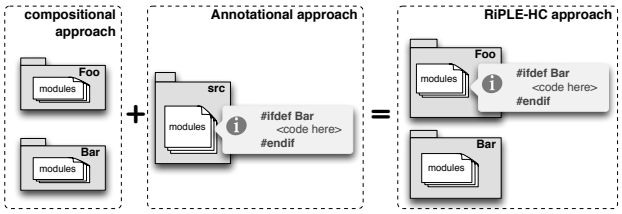


Figure 1: RiPLE-HC code organization: blending feature-based code organization and preprocessing annotations.

2.2 Concept

The **RiPLE-HC** is the **Ri**SE **P**roduct **L**ine **E**ngineering approach for **H**ybrid **C**omposition of `JavaScript`-based systems. As the name suggests, it is a hybrid approach that blends *compositional* and *annotative* approaches of SPL engineering [1]. The **RiPLE-HC** explores the modularization of the compositional approaches and the flexibility to handle feature interactions that annotative approaches enable. Such a blending allows to manage variability at different levels of the development phase. Besides, while the composition handles the inclusion or exclusion of an entire functionality in a product variant (coarse-grained variability), the annotations enables inner-function statements to behave differently (fine-grained variability), depending on the selection of a given feature.

Figure 1 shows how the RiPLE-HC employs the concept of feature-oriented software product lines [1] to organize the source code. Containment hierarchies organize the features [5], in which each directory holds elements, thus including the source code, of a given feature. The containment hierarchy is a way to modularize the code and ease the composition implementation. However, in practice, feature interaction problems – the behavior of a given feature `Foo` being changed due the presence or absence of feature `Bar` (as Fig. 1 illustrates) – make it too hard to have no code scattering, which directly impacts the code organization. The hybrid composition of the RiPLE-HC makes it possible to handle feature interaction limitations of pure composition [9] by allowing the use preprocessing annotations. Thus, there may be eventual preprocessing annotations concerning a given feature (*e.g.*, `Bar`) scattered through different folders (*e.g.*, `Foo`). It is worth to notice that our approach does not assume a `JavaScript` module equal to a single *.js* file.

Thus, while the composition-based approach handles most of the work while composing a new product, the annotative approach adds a preprocessing step preceding the real composition. In fact, although preprocessing annotations can be used anywhere within a module, so that variability management can count solely on annotations, the feature-oriented code organization fosters the inclusion of code mostly belonging to a given feature `Foo` in its particular folder. Conversely, annotations should preferably handle fine-grained variability (*e.g.*, feature interactions handling) to prevent problems with code obfuscation [12].

2.3 Implementation

The RiPLE-HC[3] was implemented as a plugin for FeatureIDE [23], a variability management tool designed to

[3]Available at: https://goo.gl/Ar2cJC

provide automated support to SPL development. Thus, we expanded the FeatureIDE capabilities to integrate preprocessing annotations with the native composition-based approach, as a more general approach to enable variability management at implementation level. While the former enables inner-function statements to 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 could potentially violate functional specifications [1].

We next describe how the approach handles *coarse* and *fine*-grained variability. Detailed information on the first version of the FeatureIDE plugin can be found in our preceding work [14].

2.3.1 Coarse-grained Variability

The RiPLE-HC relies on the FeatureIDE capabilities to automatically create the containment hierarchy (Fig. 1), in which there is a directory to store all the code belonging to each concrete feature. This is a FeatureIDE inner concept. While *abstract* features are dedicated to group *concrete* features and usually are named with more general terms, the concrete features are those which actually provide the functionalities' code. When a new product is to be configured, the automated product generator picks all files from the directories associated to all corresponding features and deploys the product variant in a safe and effective manner.

In the FeatureIDE, the variability is partially controlled at implementation level, *i.e.*, if a given file associated to a feature behaves differently depending on the selection of an external feature, it replaces the entire file associated to that feature. In programming languages such as `Java`, *refinement declarations* [5] 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 programming languages which do not enable those declarations, such as `JavaScript`, applying such a technique to control inner-function variability would lead to a large amount of duplicated code.

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.

2.3.2 Fine-grained Variability

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. The RiPLE-HC rely on such control for the composition and adds its own support to handle such dependencies with low-level annotations.

Let us consider an SPL project called `algorithms.js` (Fig. 2), which has a root feature `Algorithms`, representing the domain under analysis, a set of mandatory features including a *concrete feature* called `Knapsack` and an *abstract feature* called `Queue`. `PriorityQueue` and `SingleQueue` are alternative children of `Queue`. Therefore, one and only one of them can be included, if their parent feature is included in a configuration.

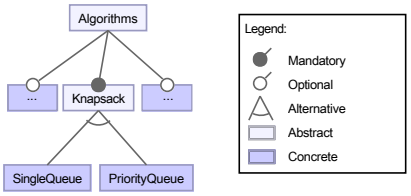


Figure 2: `algorithms.js` SPL sample feature model.

Let feature `Knapsack` has a containment hierarchy which holds a number of items, which should behave differently depending on the `Queue`'s sub-feature selection. Listing 1 illustrates how the RiPLE-HC deals with the use of preprocessor directives (annotative approach) to manage variability at implementation level.

The directives in the source code delimit blocks of program that are compiled only if a specified condition is true. They may be employed to generate different product variants by assembling the code fragments in cases where more than one product configuration includes the same *JavaScript* file, but a given *function* behaves differently depending on the feature selection. 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 the maintenance effort, as the business rules from a single function will be self-contained in a single file.

In the `algorithms.js` SPL example, after binding the variants, the variable `queue` declaration statement will be set differently, depending on the selection of either feature `PriorityQueue` or `SingleQueue`. This shows how RiPLE-HC might anticipate program-level customization of core assets for a custom product to an earlier phase in the development cycle. Besides, it controls and manages variability at both model and implementation levels to handle product enhancements.

```
function knapsack(items) {
                              ⋮
  var queue;
//#ifdef PriorityQueue
  queue = new PriorityQueue();
//#elif SingleQueue
  queue = new SingleQueue();
//#endif
                              ⋮
  });
```

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

2.4 Inherited Characteristics

We built the RiPLE-HC upon influences of previous hybrid approaches, namely FeatureC++ [3] and FeatureHouse [2], which unintentionally allow the use of annotations together with composition. In fact, compositional and annotative approaches pushed the state-the-art and practice, respectively, to another level. While the former has grown significantly and as a consequence has gathered much attention by researchers [1], the latter is one of the most used approaches in the implementation of SPL in industry.

Table 1 enumerates benefits and drawbacks of the RiPLE-HC. Each table item has a mark indicating whether the RiPLE-HC inherited the characteristics of those approaches completely (+), partially ($^+/_-$), or ignored them (−). It is not proven that hybrid approaches inherit all valuable characteristics from compositional and annotative approaches. Kästner and Apel [12] advocated that although it does not automatically dissolve all disadvantages of either approach, some benefits from both still holds after the blending.

Additionally, from Table 1, it can be seen that the RiPLE-HC resorts from better modularization (separation of concerns – Figure 3) to provide better handling of *feature interactions* (Listing 1). Some sort of scattering should not be seen as a design flaw when kept under a defined threshold [18]. Thus, although the scattering code traceability and the maintenance of the variability may be affected by the scattering introduced by the conditional compilation, the provided tool support minimizes such effect.

Benefits and limitations of compositional approaches: The compositional approaches implement features in distinct modules (*i.e.*, it aims to eliminate the code tangling). The benefits of using them include: *(i) modularization* – they compose selected modules to bind a product of the line; *(ii) traceability* – it is straightforward the location of the code implementing each feature of the feature model; and *(iii) language support for variability* – the languages are designed in a disciplined and well-defined way being aware of variability. As the drawbacks, they entail *(i) feature interactions handling* – although there are significant gains in terms of modularization, handling feature interactions is still a challenge in compositional approaches; *(ii) coarse granularity* – which is too restrictive for implementing variability, especially in the occurrence of *feature interactions*; and *(iii) difficult adoption*, which is usually for the introduction of new language concepts and raised complexity of the SPL implementation [12].

Benefits and limitations of annotative approaches: The benefits of using an annotative approach include: *(i)* the *simple programming model* – code is annotated and removed; *(ii)* the *fine granularity* – arbitrary code fragments can be marked; *(iii)* the *variability despite the feature interactions* – they are able to handle the interaction between dependent features. On the other hand, as the drawbacks they entail *(i)* the *separation of concerns* – the modularity and traceability are likely the biggest problems with preprocessors; and *(ii)* the code *obfuscation* – the use of preprocessors at a fine granularity with nesting can make difficult to read and follow the control flow of the code [12].

Table 1: RiPLE-HC inherited characteristics from compositional and annotative approaches.

Compositional Approaches
✘ *Drawbacks*
(+) Coarse granularity (−) Poor feature interactions handling ($^+/_-$) Difficult adoption
✔ *Benefits*
($^+/_-$) Modularization ($^+/_-$) Traceability ($^+/_-$) Disciplined variability support
Annotative Approaches
✘ *Drawbacks*
($^+/_-$) Code obfuscation ($^+/_-$) Separation of concerns
✔ *Benefits*
(+) Simple programming model (+) Fine granularity (+) Ease to use (+) Strong feature interactions handling

3. CASE STUDIES

This section discusses the case studies conducted to assess the feasibility of the RiPLE-HC approach, carried out in industry and academic settings. These are discussed in sections 3.1 and 3.2, respectively.

3.1 An Industrial Case Study

The first case study was held in industry. Together with an industry partner[4], we developed the project called `MDC Learning Objects`. It consists of a family of Web-based systems, in the domain of learning objects. *Learning objects* are generally understood to be digital entities deliverable over the Internet [7]. Learning objects aim at stimulating learners' knowledge formation and retention.

The project comprises a set of `42` features. The core features have, together, around `3.7` KLOC, including `11` variation points. For this particular case study we considered three different products, fully functional, generated from the core asset base. In order to maintain the confidentiality of the information, the products will be referred to as APP1, APP2, and APP3. Table 2 shows code metrics extracted from each product[5], such as lines of code (LOC), number of files, number of functions, number of declarative statements (DS) – naming a variable, a constant, a procedure, or specifying a data type –, and executable statements (ES) – initiating actions.

Our partner reported gains in development time, what might result in order of magnitude cost reductions in next products' releases. For instance, we observed a reduction in the development time employed in APP3 – `720` engineer-hours for APP1, whereas `122` for APP3 – although it is larger in size than preceding ones (Table 2). As the core platform had already been well-established, the time demanded was mainly dedicated to build the product-specific parts. Further detailed information about this case study is available in our preceding work [14].

3.2 Academic Case Studies

Six open-source systems were manually transformed into SPL by using the RiPLE-HC approach. These systems were selected from *qualitas.js* corpus `JavaScript` systems dataset [21]. The transformed projects range from small to large

[4]http://www.reconcavotecnologia.org.br/
[5]Used tool: http://www.scitools.com/download/

Table 2: 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 3: Target systems characterization metrics.

System(v)	LOC	# Modules	# Features(CT)	# Directives	# Files	Build(s)	Domain
`algorithms.js` (0.20)	1,594	29	28 (6)	6	4	11.89	*miscelaneous*
`jasmine` (2.0.0)	2,956	48	4 (-)	14	4	3.52	*testing*
`floraJS` (1.0.0)	3,325	26	18 (-)	16	2	4.27	*simulation*
`video.js` (4.6.1)	7,939	38	13 (-)	29	10	7.03	*video player*
`TimelineJS` (2.25.0)	18,237	89	15 (-)	75	6	9.98	*web library*
`brackets` (0.41)	122,971	403	13 (1)	107	19	42.27	*text editor*

v: version; **CT:** Number of cross-tree constraints; **Directives:** Number of annotated blocks processed; **Files:** Number of files with annotated blocks; **Build:** Average of time to build; **s:** seconds.

systems. Table 3 shows descriptive metrics reproduced from the *qualitas.js* – such as *(i)* lines of code (LOC) and *(ii)* number of modules of each system – and the metrics extracted from the SPL versions of the systems, such as, *(iii)* the number of features and cross-tree constraints, *(iv)* the number of annotated blocks processed; *(v)* the number of files with annotated blocks; *(vi)* the average time of build (measured in seconds). We executed a full product build with all the features selected 10 times to compute the average of time needed. Table 3 also describes the amount of time each iteration took (column Build).

3.2.1 Granularity

The RiPLE-HC enables developers to adjust the granularity of the variability by annotating the corresponding scattered variability. Thus, at least two main levels of granularity could be experienced: modules dedicated to a given feature processed by composition, and the scattered feature code by pre-processing the conditional compilation annotations.

3.2.2 Trade-offs

The RiPLE-HC slightly modified how the `JavaScript` projects should be structured. In comparison with the current *state-of-the-practice*, instead of using an *ad-hoc* organization (*i.e.*, there is no standard followed by all the projects), the RiPLE-HC requires a more systematic way to organize the source code, regarding the features. Regardless of the notable differences in the code organization, there was no additional effort for feature code location in maintenance tasks.

3.2.3 Scalability

In order to investigate scalability issues, we included in the analysis both small and large-sized `JavaScript` projects. The *qualitas.js* dataset was built with the most popular repositories from *GitHub*. We faced difficulties to extract SPL from the `brackets` project with nested annotated blocks, which is the second biggest project in the corpus. However, when the nested blocks were left aside, the build occurred in around 40 seconds. Table 3 shows both measures, *number of features* and *number of existing annotated blocks*, may impact on the time to build as both yield more `I/O` operations. For instance, `algorithms.js` took more time than the remaining systems smaller than `brackets`. Additionally, the build time gets larger as the number of annotated blocks increases. The case studies showed that the RiPLE-HC can provide support to handle most of the `JavaScript` projects, since the case studies successfully accomplished are representative of the corpus, given that about 75% of them are smaller than 4,85 KLOC in size, and 50% of them are smaller than 1,3 KLOC.

3.2.4 Lessons Learned

While extracting the product lines from the target systems, we did not experience any issues apart from those regarding the build of the products from systems with annotated nested blocks. In addition, we realized that some systems lack any systematic source code organization, which means that most modules are placed in the same folder. In fact, as soon as the systems increase in size and/or complexity some folder organization is used, accordingly to the modules' functionalities, which recalls to the feature-oriented way to arrange the code. Although such a characteristic was not statistically checked, the way the code is organized may be an indicator that the demanded effort to migrate a set of single systems to a SPL with RiPLE-HC.

3.2.5 Threats to Validity

It is worth mentioning that these case studies were not designed to draw quantitative conclusions based on descriptive statistics, for instance, regarding the scalability of the tool support. On the other hand, the case studies can show the feasibility and applicability of the method and a proper support. They also served to gather insights about open rooms for improvement in the tool. This could be particularly observed in how the tool could improve nested annotated blocks. Nevertheless, the selection of the case studies systems may pose a threat to the validity of this study. Hence, we included systems of different domains.

4. CONTROLLED EXPERIMENT

We planned and carried out a controlled experiment with Software Engineering students to gather evidence on the maintenance effort demanded by the RiPLE-HC feature-based code organization in comparison to the current *state-of-the-practice* of `JavaScript` code organization. This section presents each phase of the experiment, as well as it discusses the results of this empirical evaluation.

4.1 Methodology

The goal of this empirical study was to compare the impact of two approaches to organize the source code in feature location from the point of view of novice developers, regarding *response time* and *correctness*: the *ad-hoc* approach, *i.e.*, tacit knowledge of the software engineers, hereinafter referred to as *Standard* – with no systematic way to organize the code – and the RiPLE-HC – with a feature-oriented code organization. Therefore, we pursue the answers to the following research questions:

*RQ*1**:** Does the code organization based on the RiPLE-HC approach reduce the *time* required for feature code location in maintenance tasks?

*RQ*2**:** Does the code organization based on the RiPLE-HC approach improve the *correctness* of feature code location in maintenance tasks?

Each question embraces a couple hypotheses, which this empirical study pursues confirmation. Table 4 describe the *null* (H_0) and *alternative* (H_1) hypotheses. In the former, the observation is that RiPLE-HC (R) code organization approach does not affect the time needed (H_{01}) to locate a feature, i.e., *Standard* (S) yields better results. The same rule applies to f_1*-score* calculations (H_{02}), explained next.

Table 4: Hypotheses tested in the controlled experiment.

	Null hypotheses		**Alternative hypotheses**
H_{01}	$\mu(Time_S) \geq \mu(Time_R)$	H_{11}	$\mu(Time_S) < \mu(Time_R)$
H_{02}	$\mu(F1_S) \geq \mu(F1_R)$	H_{12}	$\mu(F1_S) < \mu(F1_R)$

4.1.1 Metrics

To measure the performance of the subjects, and to test the hypotheses, we leveraged four metrics: *response time*, *precision*, *recall*, and f_1*-score*. The **response time** relates to the effort spent by the subject to accomplish each task, **precision** relates to correctness and it indicates how much the student correctly assigned a piece of code to the feature of a given task. The **recall** also relates to correctness and indicates how much from the source code that belongs to a given feature the student managed to find in a given task. Finally, **f_1-score** is an harmonic mean of precision and recall and it subsumes the results achieved by the subjects with regarding the perspectives of both metrics.

Precision and *recall* were obtained by employing the code *shadowing* technique [10]. The answers were either correct or wrong, based on an oracle built by one of the authors, and reviewed by the other ones. Despite the hard work on manually shadowing the code, the use of such a technique contributes to improve the reliability of the measurement procedure, as it avoids double judgment in similar cases for different subjects. Regarding the f_1*-score*, it depends only on the precision and recall values. The time values were measured in seconds, by using the `PROPHET` tool [20].

4.1.2 Subjects

Nineteen senior undergraduate students enrolled in a Software Engineering course acted as subjects. We designed a form to gather background information regarding their programming experience. Although the target systems were written in JavaScript, we also included questions about programming experience in other languages. The design followed the guidelines from Siegmund *et al.* [20], in which authors observed that programmers holding skills in varying programming languages can yield better results in program comprehension tasks.

4.1.3 Tasks

We considered two open-source `JavaScript` systems: `algorithms.js` and `video.js`. Table 3 characterizes them and they were chosen because they belong to different domains and have different sizes. We designed 21 *static* feature location tasks, *i.e.*, without counting on a running system to perform. Locating feature code for maintenance purposes is a typical task for a developer – which helps developers became aware of the system codebase – and perhaps it is one of the most time-consuming maintenance activities. Although it is not representative of the entire effort a maintenance request demands, it can surely present helpful insights on which direction the RiPLE-HC support should follows, as well as developers who decide for a different approach during their project development.

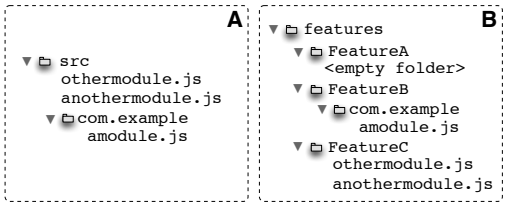


Figure 3: Code organization examples.

In the tasks, the subjects had to find the code of both, *modular* (when the feature is implemented in a single file or in set of files placed together) and *scattered* (when the code of a single feature is spread over several source files) features. Then, the codebase was organized following both approaches. Figure 3 illustrates how each observed approach, namely *Standard* (A) and RiPLE-HC (B). Next, the subjects were asked to find the code implementing a given feature of the target system, and fill in a text field with the names of the files containing the code of the given feature.

Table 5 shows some data about the features used in the experiment. For each target system, there is column that identifies the task, the feature addressed, its type, its size (LOC), and its *scattering degree (SD)*. As *scattering degree*, we consider the number of files containing source code of the feature. Features are then defined as either `modular`, if $SD = 1$, or `scattered`, otherwise.

4.1.4 Experiment Design and Variables

The experiment design consisted of "one factor (code organization) with two treatments (*Standard* and RiPLE-HC)". The experiment comprised two rounds, in which all subjects could use each tool. In **round #1 (R1)**, the students were randomly assigned to control (n=11) and experimental (n=8) groups. The *Group A* addressed the system using *Standard*, and the *Group B* with RiPLE-HC. In **round #2 (R2)**, the groups were then exchanged, so that the control group became the experimental group and the other way round. From the planned 21 tasks, 11 were addressed in **R1** and 10 in **R2**. Each task involved only one feature of the target system. Table 5 shows each of them.

Three groups of variables were considered in this experiment: *independent*, *dependent*, and *confounding variables*. The first one comprised the approaches used in this study, namely ***Standard*** and **RiPLE-HC**. The second group considered the ***time*** – as a measure of effort – and the ***correctness*** – as a measure of effectiveness. The latter encompassed different variables that may affect the task analysis, as follows: ***the level of modularity, each round, the target systems,*** and ***the size of each system***. By *level of modularity* we mean the nature of the feature (modular or scattered); *the rounds* stand for the order in which a participant addressed the system with a given approach; *target system* stands for the familiarity of the participants with them (it might be the case that subjects are familiar with `algorithms.js` but not with `video.js`); and, finally, the *size*

Table 5: Features characterization.

algorithms.js					video.js				
Task	**Feature**	**Type**	**Size** (LOC)	**SD**	**Task**	**Feature**	**Type**	**Size** (LOC)	**SD**
Task 1	KarpRabin	Modular	57	1	Task 1	AutoSetup	Scattered	35	2
Task 2	BellmanFord	Modular	43	1	Task 2	FullScreen	Scattered	173	7
Task 3	PriorityQueue	Scattered	34	2	Task 3	PlaybackRate	Scattered	88	3
Task 4	Fibonacci	Modular	28	1	Task 4	Mute	Scattered	59	5
Task 5	BinarySearch	Modular	13	1	Task 5	WebKit	Modular	11	1
Task 6	Dijkstra	Modular	33	1	Task 6	OldWebKit	Modular	11	1
Task 7	Heap	Scattered	73	3	Task 7	Mozilla	Scattered	19	2
Task 8	InsertionSort	Modular	16	1	Task 8	Microsoft	Modular	10	1
Task 9	MergeSort	Modular	24	1	Task 9	BigPlayButton	Scattered	13	3
Task 10	Stack	Scattered	13	2	Task 10	LoadingSpinner	Scattered	23	3
Task 11	CountingSort	Modular	35	1	-	-	-	-	-

LOC: Lines of Code; **SD:** Scattering Degree.

of the system stands for the extent to which the difference in the target systems influenced the analysis of the source code.

4.2 Execution

This section describes the subjects characterization and the preparation for the experiment execution.

Subjects Characterization. The answers suggested that ≈ 32% of the students had previous industry experience. All of them had been enrolled in the university for at least three years. Before joining the experiment, they had taken at least five programming courses.

Their programming experience have been evaluated in a 5-point likert scale (1 to 5, in which 1 is the lowest value and 5 is the highest one) with a questionnaire adapted from Siegmund *et al.* [20], and the results are described next. More than 70% of the participants ranked themselves as 4 or higher experience in C programming; regarding Java programming, over 60% of them reported as being experienced programmers; and a small set of about 33% had previous experience in JavaScript programming.

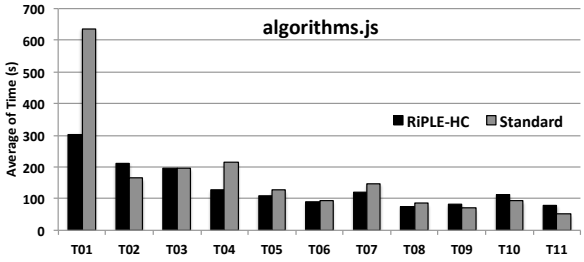
Preparation. In the experiment, the training section took about 60 minutes. It consisted of establishing a common vocabulary, explanations on the environment where they had to report the results, and the forms to fill out. Next, both rounds were performed. *R1* took about 70 minutes and *R2* took about 50 minutes. The confounding variables may explain the observed difference in execution time between the rounds.

4.3 Results and Discussion

In this section, we present and discuss raw data, and the impact of both approaches on the dependent variables *time* and *correctness*. While the former produces evidence on the cost of maintenance tasks, the latter produces evidence on whether developers may or may not benefit from using the RiPLE-HC over *Standard*.

We started the analysis by applying the *Shapiro-Wilk* test to verify the normality of each sample, namely *Time* and F_1*_score* in both, *R1* and *R2*. Results pointed out normality in the samples generated in *R1*, concerning to time values for both treatments (RiPLE-HC and *Standard*). Besides, we carried out a data transformation on the values from *R2*, by applying a logarithmic function to adjust the statistical differences found. Then, we carried out the hypothesis testing, by applying the *independent T-Test* to assess *Time* (*R1*), and the non-parametric *Mann-Whitney U* test was used for *Time* (*R2*) and F_1*_score* (in both *R1* and *R2*).

(a) *Round 1*



(b) *Round 2*

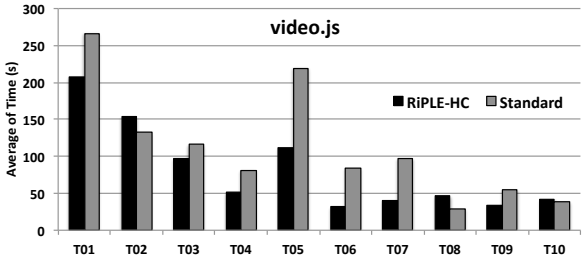


Figure 4: Average time spent in each task.

4.3.1 Execution Time

Figure 4 shows the average time spent in each task, in seconds. We may observe similar results between tasks carried out by the control and experimental groups. In both, the earlier tasks demanded more time to produce the results. The lack of familiarity with the tools may explain those values. As the subjects gained confidence on the source code, the time spent decreased. Indeed, the similarity in time spent refutes the arguments in favor of the harmfulness of the code scattering. To a certain extent, the scattered code produced by using the RiPLE-HC approach did not demand extra effort.

Although both target systems are small, there is a significant size difference between them. However, such a difference does not affect the effort to locate the features. Subjects spent less time analyzing the second system – Figure 4(b), than the preceding one – Figure 4(a). The lower values in *R2* can be a result of the likely maturation effect, given that subjects were already familiar with the activity.

Most subjects spent less time on average to perform fea-

Table 6: Mann-Whitney U Test of hypothesis for *Time* spent.

Approach	Round 1 Mean Rank	Round 1 p-value	Round 2 Mean Rank	Round 2 p-value
RiPLE-HC	137.48	.539*	9.30	.36
Standard	171.05	.542**	11.70	

*: Equal variances assumed; **: Equal variances not assumed.

Table 7: Mann-Whitney U Test results for F_1*_score*.

Approach	Round 1 Mean Rank	Round 1 p-value	Round 2 Mean Rank	Round 2 p-value
RiPLE-HC	13.27	.20	10.80	.82
Standard	9.73		10.20	

ture location when the target system was organized with the RiPLE-HC. RQ1 is primarily interested in analyzing whether the RiPLE-HC approach reduces the time needed to locate features. The hypothesis test was performed in both rounds by considering the average time spent by the subjects in each task. Table 6 shows that the subjects that used the RiPLE-HC spent less time to perform their tasks. The significant difference on the mean values is due to the mentioned data transformation. However, with a p-value higher than .05, it is impossible to refute the null hypothesis (H_{01}) in any rounds.

4.3.2 Correctness

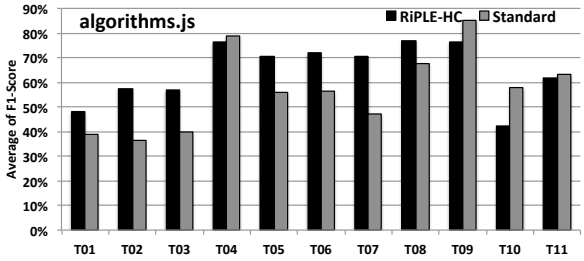
The results indicate that both approaches produced similar impact on feature location for modular and scattered features. In most cases in *R1*, the F_1*_score* of both approaches were higher than 50%. (Figure 5(a)). However, both approaches had worse results in *R2* (Figure 5(b)). Although the RiPLE-HC does not excel `Standard` results in tasks T01, T03, and T04, the results were good in all the other tasks. Subjects inspecting source code organized with the RiPLE-HC yielded slightly better results when compared to the *Standard* approach. In fact, in *R1*, while the median of RiPLE-HC was around 0.8, in the *Standard* approach was around 0.6. In *R2*, the difference was around 20%. We believe that subjects might have been affected by the novelty on how the code is organized in the RiPLE-HC prior to the training section of the experiment. Such an impact might explain the perceived reductions in gains concerning to the source code organization.

Regarding the analysis of correctness, as RQ2 stands out, the hypothesis testing considered the average of the F_1*_score* of the subjects in every task. Table 7 shows the observed results. The subjects who used the RiPLE-HC approach got better results in both rounds. However, we see that p-value is greater that .05 in both rounds, thus, we cannot refute null hypothesis (H_{02}) in any of them.

4.4 Threats to Validity

In this section, we discuss potential threats to the validity of this empirical study. We believe that presenting such detailed information may contribute to further research and replications of this study [25], which may be built upon the results presented herein. Next, we detail the main threats according to *external*, *internal*, *construct*, and *conclusion* validity.

(a) *Round 1*



(b) *Round 2*

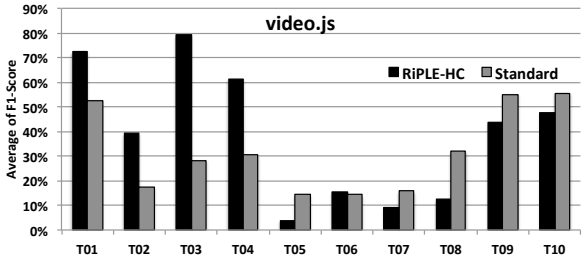


Figure 5: Average f_1-score in each task.

External validity: We identified some threats that may limit the ability to generalize the results. For example, the study was carried out in an *in-vitro* setting, which means a sample selected pseudo-randomly. In addition, most the subjects were characterized as inexperienced with industry projects, which poses a threat to the study. We attempted to mitigate such a threat by characterizing and reporting the environmental settings of the experiment, since it is unfeasible to reproduce a realistic environment.

Internal validity: There are possible threats that may happen without the researcher's knowledge affecting individuals from different perspectives, such as *(i)* the maturation and learning effects, *(ii)* the testing repetition since several tasks were carried out, and *(iii)* the experiment instrumentation. These threats were mitigated by choosing different features for each task, as well as by randomizing the sequence of task's execution to omit possible relationships. Finally, the only artifacts used were the source code which subjects were already familiar with, and the feature models and the `PROPHET` tool, explained in the training session.

Construct Validity: Confounding constructs may affect the findings. For instance, the presence or the absence of knowledge about a particular programming language may not explain the causes of failures in the feature location tasks. In fact, the differences may depend on the subjects' experience, which was controlled with the characterization form, to ensure that subjects had substantial experience to accomplish the tasks.

Conclusion Validity: We observed from the results a likely *low statistical power*, which concerns to the power of used tests to reveal a true pattern in the data. Employing well-known measures mitigated such a threat. Another observed threat is the *fishing* for a specific result, which we mitigated by relying the analysis only on the gathered data. There is a threat on the *reliability of treatment implementation*, when subjects are treated differently, which was minimized

by avoiding communication with the subjects and leaving time for discussion of the experiment between the training and the experiment sessions. Finally, the *random heterogeneity of subjects*, which was measured by the characterization form and presented, but no additional actions were taken to control it since the experiment took place in the context of an academic course.

5. RELATED WORK

There is a number of tools available to foster modularity in `JavaScript`-based systems, namely, package managers (*e.g.*, `npm`, `jam`, `bower`, etc.), dependencies managers (*e.g.*, `requireJS`), among others. However, these approaches do not allow the project features management based on feature model or product composition. In fact, our approach does not exclude or intends to substitute such tools, but to improve the reuse in such systems. There are some investigations we deem as related to ours. They are discussed next.

Kästner and Apel [12] presented an initial discussion on the hybrid approaches. They managed to show an eventual path to combine advantages, increase flexibility for the developer, and ease the adoption of a hybrid approach. Later on, Apel *et al.* [2] introduced a language independent approach based on superimposition, called FeatureHouse, which unintentionally allowed hybrid composition of `C/C++` systems. They built a number of systems in different languages and conducted their discussion regarding the challenges addressed in the constructions of their approach. The RiPLE-HC was developed after discarding such initiative as viable to an extension aiming to our partner's system domain.

Prehofer [17] treated the feature interaction issue with *lifters*. A *lifter* defines a modular means to implement the feature interaction. Liu *et al.* [13] proposes refactoring in legacy applications 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. All the following studies [6, 16, 24] proposed strategies to handle Web-based SPLs to a certain extent. Nevertheless, they also focus rather on modeling aspects. By contrast, in this present investigation we considered a lower level of abstraction, while proposing a strategy to cope with variability at the implementation level. None of these studies deal with feature-based composition nor presented empirical evidence of such. Therefore, to the best of our knowledge, there is a lack of empirical evidence on the impact of hybrid composition software development and on the maintenance tasks in `JavaScript`-based systems.

6. CONCLUDING REMARKS

This paper presented the RiPLE-HC, a hybrid approach for SPL composition that introduces systematic reuse to `JavaScript`-based systems. In order to evaluate the proposed approach, we carried out two case studies, which considered both academy and industry standpoints and an empirical investigation comprised of a controlled experiment. The academic case studies serve to reinforce prior evidence from the Web-based case study in an industrial setting, whereas the controlled experiment produced new evidence on the benefits of systematic code organization.

Case studies observations. The case studies showed that the RiPLE-HC can handle real-world systems from small to large-sized projects, as well as systems from different domains. As expected from the literature, the time needed for building a new variant seems to be associated with the number of features defined and the number of existing annotated blocks. Scalability problems were faced with nested blocks, as such, they are not recommended in the current stage of the prototype implementation. Additionally, we observe that even with no systematic way to structure the code, as soon as the systems increase in size, the project structure tends to assume characteristics of feature-oriented organization, which may indicate that larger projects might benefit from this novel approach. Moreover, the industrial case study showed the possibility to reduce costs of development as early as in the third product, which required only 17% of the engineer-hours of the first product.

Experiment results. In all the four scenarios defined for the experiment (Rounds 1 and 2 regarding both hypotheses: *time* to complete the task – 2 scenarios – and *correctness* of the answers – 2 scenarios), the mean of the results of the subjects indicated slightly better result in favor of our approach. However, the data points did not allow us to statistically refute the null hypotheses. Therefore, it is not possible to generalize that developers addressing feature location tasks in the current *state-of-the-practice* of code organization take longer, neither that they make more errors than those addressing the code structured with the proposed approach. In addition, the feedback of the participants suggested that *(i)* the RiPLE-HC provides better code organization regarding the systems functionalities; *(ii)* the composition can ease the product development for different platforms; and *(iii)* that their unfamiliarity with the approach may have hindered better results, which should be further investigated.

Future Work. We plan to implement a set of improvements in the RiPLE-HC tool support, such as the filtering the scattering graphs. We also plan to improve the tool support to better handle nested annotated blocks. Additional case studies to better investigate the synergy of the hybrid composition with the dynamic nature of `JavaScript`, such as global scope, function redefinition, weakly typing, as well as the usage of different current available frameworks like angular and react are possible directions to further investigation. Furthermore, we plan to replicate the controlled experiment, using more experienced subjects and different types of tasks, so as to compare the results and identify opportunities to improve the proposed RiPLE-HC approach.

Acknowledgments

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[6]), funded by CNPq and FACEPE, and FAPESB.

References

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Berlin Heidelberg, 2013.

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

[2] S. Apel, C. Kästner, and C. Lengauer. Language-independent and automated software composition: The featurehouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the 4th Generative Programming and Component Engineering*, pages 125–140, 2005.

[4] D. S. Batory. Feature-oriented programming and the ahead tool suite. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 702–703, 2004.

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

[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] G. C. S. Ferreira, F. N. Gaia, E. Figueiredo, and M. A. Maia. On the use of feature-oriented programming for evolving software product lines — a comparative study. *Science of Computer Programming*, 93, Part A(0):65 – 85, 2014. Special Issue with Selected Papers from the Brazilian Symposium on Programming Languages (SBLP 2011).

[10] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *30th Int'l. Conf. on Software Engineering (ICSE)*, pages 261–270, 2008.

[11] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems: A systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2014.

[12] C. Kästner and S. Apel. Integrating compositional and annotative approaches for product line engineering. In *Proceedings of the Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 1 – 6, 2008.

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

[14] I. C. Machado, A. R. Santos, Y. C. Cavalcanti, E. Trzan, M. de Souza, and E. S. Almeida. Low-level variability support for web-based software product lines. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, pages 15:1–15:8. ACM, 2014.

[15] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 495–518, 2015.

[16] 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.

[17] 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.

[18] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki. The shape of feature code: An analysis of twenty C-preprocessor-based systems. *Journal on Software and Systems Modeling*, page 1–29, To appear 2015.

[19] A. R. Santos and E. S. Almeida. Do #ifdef-based variation points realize feature model constraints? *SIGSOFT Softw. Eng. Notes*, 40(6):1–5, 2015.

[20] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[21] L. Silva, M. Ramos, M. T. Valente, A. Bergel, and N. Anquetil. Does javascript software embrace classes? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 73–82, 2015.

[22] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with c news. In *Proceedings of the USENIX Summer 1992 Technical Conference*, pages 185–197, 1992.

[23] 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.

[24] 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*, pages 44–53. IEEE Computer Society, 2007.

[25] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.