Empirical Software Engineering (2022) 27: 85
https://doi.org/10.1007/s10664-021-10104-3

Open-source software product line extraction processes: the ArgoUML-SPL and Phaser cases

Rodrigo André Ferreira Moreira[1] · **Wesley K. G. Assunção**[2,3] · **Jabier Martinez**[4] · **Eduardo Figueiredo**[1]

Accepted: 9 December 2021 / Published online: 8 April 2022

Abstract
Software Product Lines (SPLs) are rarely developed from scratch. Commonly, they emerge from one product when there is a need to create tailored variants, or from existing variants created in an ad-hoc way once their separated maintenance and evolution become challenging. Despite the vast literature about re-engineering systems into SPLs and related technical approaches, there is a lack of detailed analysis of the process itself and the effort involved. In this paper, we provide and analyze empirical data of the extraction processes of two open-source case studies, namely ArgoUML and Phaser. Both cases emerged from the transition of a monolithic system into an SPL. The analysis relies on information mined from the version control history of their respective source-code repositories and the discussion with developers that took part in the process. Unlike previous works that focused mostly on the structural results of the final SPL, the contribution of this study is an in-depth characterization of the processes. With this work, we aimed at providing a deeper understanding of the strategies for SPL extraction and their implications. Our results indicate that the source code changes can range from almost a fourth to over half of the total lines of code. Developers may or may not use branching strategies for feature extraction. Additionally, the problems faced during the extraction process may be due to lack of tool support, complexity on managing feature dependencies and issues with feature constraints. We made publicly available the datasets and the analysis scripts of both case studies to be used as a baseline for extractive SPL adoption research and practice.

Keywords Software product lines · Re-engineering · Mining software repositories · ArgoUML · Phaser

Communicated by: Philippe Collet, Sarah Nadi, Christoph Seidl, and Leopoldo Motta Teixeira

This article belongs to the Topical Collection: *Software Product Lines and Variability-rich Systems (SPLC)*

✉ Wesley K. G. Assunção
wesleyklewerton@gmail.com

Extended author information available on the last page of the article.

1 Introduction

Software Product Lines (SPL) (Northrop and Clements 2012) are not always developed from scratch but emerge from an existing system or from existing variants (Berger et al. 2013). This kind of adoption is known as extractive adoption (Krueger 2001), which is undertaken by a re-engineering process. Re-engineering processes to extract SPLs have been analyzed from different perspectives. For instance, industrial cases present lessons learned regarding organizational and technical aspects (Martinez et al. 2017), researchers aim to increase the level of automation of the re-engineering steps (Assunĉão et al. 2017), and practitioners apply risk evaluation methods and cost models to measure the return on investment (Ali et al. 2009).

In this work, we focus on the process and the effort required for extractive SPL adoption and we fill the gap on the lack of comparable and reproducible research on the topic given the confidentiality constraints of existing industrial cases reported in the literature (Martinez et al. 2017). We can argue that there is still no evidence about the cost-effectiveness of using automatic approaches for different re-engineering activities, and that ways to estimate the re-engineering effort beforehand could be improved. We consider that the main reason is that we lack baselines for comparison and empirical data providing clues for estimation. As in other software engineering research fields, open-source software systems provide valuable assets and information for extractive SPL adoption research (Martinez et al. 2017; Wolfart et al. 2019).

In a previous work (Martinez et al. 2020), we analyzed empirical data of an existing SPL extraction process: the ArgoUML monolithic architecture transition to ArgoUML-SPL. This paper extends our previous work (Martinez et al. 2020) not only by expanding the analysis and discussion about the ArgoUML to ArgoUML-SPL transition, but also by reporting observations of the extractive SPL adoption in a second case study, namely Phaser. ArgoUML[1] is probably the most representative system that can be comparable to industrial practices (Assunĉão et al. 2017; Martinez et al. 2017). The transition from ArgoUML to ArgoUML-SPL was an academic project, but it was focused on industry-strength practices (Couto et al. 2011). Phaser[2], on the other hand, is an open-source 2D game framework for Web browsers. It has been developed and maintained by a large community of JavaScript developers, receiving frequent updates. The community of Phaser developers extracted an SPL in version 2.3.0 aiming at enabling the game developers to pick and choose which features they would like to use while developing and building their own game.

These systems were selected mainly because of their relevant size and the cross-cutting features they contain. For instance, we selected Phaser because it is an open-source software system maintained by a large community of developers. Moreover, ArgoUML-SPL was extensively used for validating automatic or semi-automatic techniques for feature location in families of systems (AL-Msie'deen et al. 2013; Assunĉão et al. 2017; Eyal-Salman et al. 2013a, b, c, 2014; Linsbauer et al. 2013; Martinez et al., 2016, 2017, 2018; Michelon et al. 2021, 2021; Strüber et al. 2019; Ziadi et al. 2012), extracting SPLs or composing new product variants (Fischer et al. 2014; Klatt et al. 2014; Ziadi and Hillah 2018), and its artifacts were used as source of information for reverse engineering studies (Linsbauer et al. 2014; Martinez et al. 2015). These studies mainly used ArgoUML-SPL variants as a source of

[1]ArgoUML: http://argouml.tigris.org and ArgoUM-SPL: http://argouml-spl.tigris.org. Moved on July 2020 to https://github.com/argouml-tigris-org, https://github.com/argouml-tigris-org/argouml-spl, respectively.

[2]Phaser: http://phaser.io/

information. However, among these studies, there is no translation from the results to actual operationalization of the re-engineering process. For instance, it is unclear what means e.g., fifty percent of precision compared to the effort (e.g., months) of performing manual location because the metrics are not comparable. In the literature, we can find some pieces of work, about other cases, describing the effort related to the duration (Ali et al. 2011; Dhungana et al. 2011; Kolb et al. 2006; Zhang et al. 2011; Yang et al. 2009) and number of developers (Hariri et al. 2013; Olszak and Jørgensen 2012; Otsuka et al. 2011) to conduct the re-engineering process. However, only brief or no discussion is provided.

Current SPL research focuses mostly on the result of the re-engineering, not on the actual process of obtaining an SPL. For instance, in the seminal publication where ArgoUML-SPL was presented (Couto et al. 2011), the focus was on an in-depth characterization of the extracted SPL in terms of size metrics, crosscutting metrics, crosscutting behavior analysis, granularity metrics and analysis of where in the code structure the annotations were inserted with more frequency. Contrary to the characterization of *the structure* of the extracted SPL, in this work we contribute with a missing analysis and characterization on *the process* of creating ArgoUML-SPL from the ArgoUML monolithic architecture. Additionally, we also analyze the process of creating the Phaser-SPL aiming at providing a complementary set of examples. These pieces of information will enable us to provide a deeper understanding of the strategies for SPL extraction and their implications.

The contributions of this work are:

- A characterization of the processes where two software systems were transformed into SPL. This characterization was possible using practices from the mining software repositories field (Hassan 2008). The insights of this characterization can serve as an experience report, describing the steps, artifacts, time taken, tools used, and problems of two specific cases, for companies and practitioners willing to re-engineer an existing monolithic system into an SPL. That is, practitioners can then have examples of how other re-engineering processes were conducted.
- A publicly available dataset and analysis scripts[3] to be used as a baseline for extractive SPL adoption research. This dataset includes a large amount of data for the two illustrative and comprehensible examples of SPLs creations.

This paper is structured as follows. Section 2 presents background information and definitions used in this paper. Section 3 explains how we design the case studies and the relevant differences between the ArgoUML and Phaser systems. Results are presented and analyzed in Section 4, focusing on answering our research questions. Section 5 is devoted to broad discussions on the results. Section 6 presents and analyses threats to the study validity. The existing related literature and other pieces of work describing the re-engineering process is presented in Section 7. Finally, Section 8 concludes and outlines future work directions.

2 Background

This section presents background information on diverse topics needed for a better understanding of this work. Variability management in SPLs can be implemented with dif-

[3]https://zenodo.org/record/5519966.

ferent approaches, where annotative or compositional approaches are the main subdivision (Kästner et al. 2008). Sections 2.1 and 2.2 explain the annotative approach and feature interactions, present in ArgoUML-SPL, while Section 2.3 presents the compositional approach, used in Phaser. Section 2.4 presents a general introduction to the re-engineering process towards SPLs. Finally, Sections 2.5 and 2.6 introduce the ArgoUML and Phaser case studies.

2.1 Annotative SPLs

The annotative approach is a well-known technique for handling software variability (Babar et al. 2010). It has long been used in programming languages like C (commonly known as `ifdef` preprocessor directives), and it can also be used in object-oriented languages, such as C++ (Hu et al. 2000) and Java. For example, the Java Preprocessor[4] functionality is the one used in ArgoUML-SPL. In this approach, annotative directives indicate pieces of code that should be compiled/included or not based on the value of variables. The pieces of code can be marked at the granularity of a single line of code or to a whole file. Feature toggling is also a used annotative approach (Mahdavi-Hezaveh et al. 2021) where there is no need of specific variability management libraries as the annotations are based on the standard `if` clauses of the target programming language.

The code snippet in Listing 1 shows the use of preprocessor directives in ArgoUML-SPL. In this example, the `//#if defined(UseCaseDiagram)` directive in line 5, for instance, indicates the beginning of code belonging to the Use Case Diagram feature. On the other hand, the directive in line 8 specifies the code of the State Diagram feature. The `#endif` directives (e.g., lines 7 and 10) determine the end of code associated with each feature. Each identifier, such as `UseCaseDiagram` in line 5, is associated with a Boolean value defined in a configuration file for each product of ArgoUML-SPL. Its value indicates the presence of the feature in a product and, consequently, the inclusion of the bounded piece of code in the compiled product.

2.2 Feature Interactions

Feature interaction occurs when the behavior of one feature is influenced by the presence of other features (Apel et al. 2013). The interaction may not be easily deduced from the behaviors of the involved features in isolation and this fact hinders modular reasoning. In fact, not all feature interactions are undesired. One feature often communicates and cooperates with other features to accomplish a task in concert. For instance, the ActivityDiagram feature is lexically nested to the StateDiagram feature in several places of the ArgoUML-SPL code (Couto et al. 2011). By inspecting the code, we observed that this nesting is due to the fact that Activity Diagrams are a specialization of State Diagrams, as defined by the UML specification (Booch 2005). Another example is the Cognitive feature in ArgoUML which aim is to support the correct UML modelling with predefined critics for each type of diagram, or the Logging feature which aims to capture events in all features. Feature interactions exist because of this expected cross-cutting behavior.

[4]javapp: http://www.slashdev.ca/javapp

```java
public final class DiagramFactory {
  private DiagramFactory() {
    super();
    diagramClasses.put(DiagramType.Class,
        UMLClassDiagram.class);
    //#if defined(USECASEDIAGRAM)
    diagramClasses.put(DiagramType.UseCase,
        UMLUseCaseDiagram.class);
    //#endif
    //#if defined(STATEDIAGRAM)
    diagramClasses.put(DiagramType.State,
        UMLStateDiagram.class);
    //#endif
    //#if defined(DEPLOYMENTDIAGRAM)
    diagramClasses.put(DiagramType.Deployment,
        UMLDeploymentDiagram.class);
    //#endif
    //#if defined(COLLABORATIONDIAGRAM)
    diagramClasses.put(DiagramType.Collaboration,
        UMLCollaborationDiagram.class);
    //#endif
    //#if defined(ACTIVITYDIAGRAM)
    diagramClasses.put(DiagramType.Activity,
        UMLActivityDiagram.class);
    //#endif
    //#if defined(SEQUENCEDIAGRAM)
    diagramClasses.put(DiagramType.Sequence,
        UMLSequenceDiagram.class);
    //#endif
  }
  ...
}
```

Listing 1 Example of variability management with annotative directives

2.3 Compositional SPLs

Compositional approaches to implement SPLs enable the addition of implementation fragments in specified places of a system (Apel et al. 2009; Batory et al. 2004; Prehofer 2001; Schaefer et al. 2010). With the compositional approach, we defined separated reusable assets that are composed during derivation when features are selected. A relevant example is FeatureHouse which is based on source code superimposition and merge (Apel et al. 2009). Compositional or positive variability mechanisms include generative programming (Czarnecki and Eisenecker 2000), feature-oriented programming (Batory et al. 2004; Prehofer 2001), aspect-oriented programming (Kiczales et al. 1997) or delta-oriented programming (Schaefer et al. 2010). In fact, it is also possible to combine compositional with annotative techniques.

In Phaser, a compositional approach is used where features are separated in dedicated JavaScript files, which are then merged during the build process if the features are selected. Researchers have also investigated how to deal with JavaScript SPLs combining both compositional and annotative approaches (Santos et al. 2016). However, Phaser-SPL is based on a dedicated Grunt workflow[5] implemented by the Phaser developers. The code snippet in Listing 2 was extracted from the Gruntfile. This example lists a subset of the available

[5]https://gruntjs.com/

```
var modules = {
    'intro':          { 'description': 'Phaser UMD wrapper',
                      'optional': true, 'stub': false },
    'phaser':         { 'description': 'Phaser Globals',
                      'optional': false, 'stub': false },
    'geom':           { 'description': 'Geometry Classes',
                      'optional': false, 'stub': false },
    'core':           { 'description': 'Phaser Core',
                      'optional': false, 'stub': false },
    'input':          { 'description': 'Input Manager + Mouse
        and Touch
                      Support', 'optional': false, 'stub':
                          false },
    'gamepad':        { 'description': 'Gamepad Input',
                      'optional': true, 'stub': false },
    'keyboard':       { 'description': 'Keyboard Input',
                      'optional': true, 'stub': false },
    'components':     { 'description': 'Game Object
        Components',
                      'optional': false, 'stub': false },
    'gameobjects':    { 'description': 'Core Game Objects',
                      'optional': false, 'stub': false },
    'bitmapdata':     { 'description': 'BitmapData Game
        Object',
                      'optional': true, 'stub': false },
    'graphics':       { 'description': 'Graphics and PIXI
        Mask Support',
                      'optional': true, 'stub': false },
    'rendertexture':{ 'description': 'RenderTexture Game
        Object',
                      'optional': true, 'stub': false },
    'text':           { 'description': 'Text Game Object
        (inc. Web
                      Support)', 'optional': true, 'stub':
                          false },
    'bitmaptext':     { 'description': 'BitmapText Game
        Object',
                      'optional': true, 'stub': false },
    'retrofont':      { 'description': 'Retro Fonts Game
        Object',
                      'optional': true, 'stub': false },
    ...
};
```

Listing 2 Example of variability management with compositional approach

modules followed by a brief description, whether they are optional or not, and if they are replaceable by a stub version. During the build process, the developers are able to define which modules they would like to exclude from their build.

2.4 The Re-engineering Process

The process of extracting an SPL from legacy system variants is discussed in a mapping study (Assunċão et al. 2017). This process covers some specific phases that are designed to deal with commonalities and variabilities among the system variants. More specifically, as results of that systematic mapping, the authors presented a generic process with three phases commonly found in the primary sources. The generic process is presented in Fig. 1.

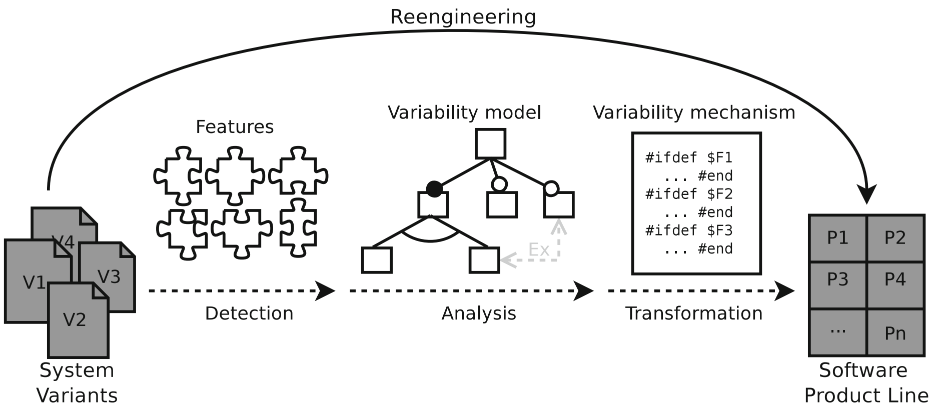


Fig. 1 Generic re-engineering process, extract from Assunĉão et al. (2017)

The three phases are: (i) *detection*, with the goal to identify features (Martinez et al. 2016; Ziadi et al. 2012), which are building blocks of SPLs (Kang et al. 1990), available in each the system variant and locate where such features are implemented (Rubin et al. 2013; Dit et al. 2013); (ii) *analysis*, focus on discovering the relationship between features and representing these relationships, mainly using a variability model, e.g., a feature model; and (iii) *transformation*, based on the information of the previous two phases, implementation artifacts are modified to describe variability, e.g., using annotative approaches or compositional approaches. Figure 1 depicts, on the left side, the system variants developed using opportunistic reuse, e.g., clone-and-own practices (Fischer et al. 2014).

The main motivation of migrating to an SPL is to mitigate engineering problems related to maintenance and evolution. This is a way to remove the technical debt related to deficient variability management (Wolfart et al. 2021). The re-engineering process can take place from a single system to allow systematic generation of a family of system variants (Krueger 2001) and this is the case of the two case studies considered in our study. In such a situation, the motivation is to allow different configurations of products from a common base of artifacts, enabling systematic reuse. Through an extractive adoption of an SPL (Krueger 2001), a company can reduce time-to-market for new products, growing its portfolio of software products, and better fulfilling client demands. In this case, the extraction should be conducted with the support of the available artifacts. Some studies highlight the importance of high-level artifacts, such as feature models and SPL architectures (Assunção et al. 2020; Knodel and Muthig 2005). Other studies describe the process based mainly on source-code (Assunĉão et al. 2017; Laguna and Crespo 2013).

In the generic process described in Fig. 1, it is not straightforward to decide when a specific activity was completed. For instance, an SPL could have been re-engineered (i.e., variants can be derived) but it is difficult to assess its validity. Missing or incorrect implementations using the variability mechanisms, or non-foreseen feature interactions, can lead to incorrect derived variants. SPL testing is a complex field (Engström and Runeson 2011) given the combinatorial explosion of possible variants and the difficulties in test case design and automation. As a simplification, we will consider that the reengineering covers the process until variants can be derived with certain confidence, and it is the responsibility of the SPL maintenance phase to address or try to automate the identification of derivation problems including their fixing.

2.5 ArgoUML in a Nutshell

ArgoUML is an open-source tool that supports the edition of UML 1.4 diagrams and other related functionalities, such as code-generation from UML diagrams or reverse engineering UML diagrams from source code. Its initial release was in 1999 and the latest release was v0.34 in 2011 with 120 KLoC of Java code. Along its history, over 150 developers contributed to its development counting close to 20 thousand commits (from now on, we call them *revisions* following the revision control terminology of SVN[6]). More than six thousand issues were reported, from which around five thousand were closed[7]. Relevant fixes were made around 2014 when the activity on its development was abandoned.

In 2010, an experienced developer and master student, supervised by two software engineering researchers, initiated the re-engineering process of ArgoUML (v0.28.1) to ArgoUML-SPL as part of his master thesis (Couto 2010), which is the basis of one of our studies.

2.6 Phaser in a Nutshell

Phaser is an open-source 2D game framework used as a platform to develop HTML5 games for both desktop and mobile (Photon Storm Ltd 2021). Phaser was first released in April 2013, as version 0.5.0, with the goal of allowing the quick creation of games and removing some complexity imposed by the use of HTML5 purely. The first stable version, namely 1.0.0, was released after six months of massive enhancement and refactoring across the entire codebase. In this work, we rely on the Phaser Community Edition[8] (Phaser CE). The latest version of Phaser CE is 2.16.0 and it was released in June 2020. Phaser CE was originally called Phaser 2 and became community driven since version 2.7.3. According to the lead developer, Phaser 2 was a milestone for the framework, promoting the development of numerous games and popularizing the framework. However, with the advent of Phaser 3, which is a complete restructuring of Phaser using a fully modular structure and combined with a data-oriented approach, Phaser 2 was handed to the community of developers that directly control it nowadays.

In this study we focus on the version 2.3.0[9] of Phaser, in which developers introduced the ability of customizing Phaser with custom builds. According to Phaser documentation[10], "*In previous versions [before 2.3.0] of Phaser we adopted something of a "kitchen sink" approach, whereby lots of features were bundled in with no way to turn them off. This led to some quite large build file sizes, even if you didn't need lots of the capabilities present in there.*" The ability of creating custom Phaser builds enabled game developers to select only those features they will use on their game application. For example, if the game being developed does not have keyboard inputs, the developer can exclude the implementation responsible for handling such a feature. This process of avoiding the inclusion of unused features results in a smaller build file size, which is a benefit for the developers since it

[6]https://subversion.apache.org/

[7]http://argouml.tigris.org/project_bugs.html. Stored (July 2020): https://github.com/argouml-tigris-org/argouml/releases/tag/GITHUB_IMPORT

[8]Available at: https://phaser.io/download/phaserce. Notice that Phaser-CE is not Phaser 3, which has a very different architecture.

[9]https://github.com/photonstorm/phaser/releases/tag/v2.3.0

[10]http://phaser.io/tutorials/creating-custom-phaser-builds

reduces the game's loading time. A secondary benefit is that it reduces the size of the application programming interface of the framework, but size reduction is the main goal of the custom builds.

3 Study design

This section describes the methodology of our study in order to investigate the SPL extraction processes of ArgoUML-SPL and Phaser. Figure 2 describes the steps we followed. Firstly, we defined the goal and research questions of our study (Section 3.1). Secondly, we conducted the extraction of data regarding the re-engineering process (Section 3.2). Then, during the discussion of the results, some questions were raised, leading us to contact the developers in charge of the re-engineering for clarifications (Section 3.3). Finally, we performed the analysis of the data and developer's comments and reported in this paper.

3.1 Study Goal, Questions, and Metrics

The design of our case studies is based on the Goal/Question/Metric (GQM) approach (van Solingen et al. 2002), as described below.

Study goal analyze the SPL extraction processes of two open-source systems for the purpose of observing how the process was conducted with respect to the timeline, number and size of the features, from the viewpoint of developers and maintainers.

Research questions (RQ) Guided by our study goal, we derived the following research questions.

– *RQ1: How has the source code of the systems changed during the re-engineering processes?* We discuss the modifications for turning the monolithic structure of the systems into an architecture that allows dealing with variability and customization.
    – *Metric:* LoC of each feature, percentage of LoC changes in the architecture, features made optional, and interactions among features.
– *RQ2: How was the operationalization of the re-engineering processes?* Since the SVN and GitHub repositories are our source of information, in this question we aim to investigate if the features were extracted all in the same trunk, if there were branches, and the number of revisions in the process. Also, we investigate aspects related to the time, effort, and team involved in the extraction of the case studies. The goal of this RQ is to provide insights for those willing to move from monolithic systems to SPL architectures.

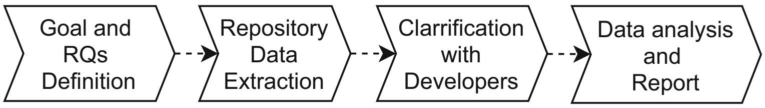


Fig. 2 Study Steps

  - *Metric:* number of branches vs number of features (branching strategy), number of revisions per feature, number of months for each feature extraction, number of people, and initial/final revisions.

- *RQ3: What were the problems faced when conducting re-engineering processes of the systems?* Since the re-engineering process is a complex intellectual activity, it is an error-prone task. Here we investigate which and how bugs were found after the SPL extraction, and barriers found by the collaborators during the process.

  - *Metric:* bugs found, inconsistencies detected, and feedback from developers.

3.2 Repository Data Extraction

Our study relies on data mined from version control systems of both systems. In the ArgoUML-SPL case, we mined data from the SVN repository that was used as the version control system during the re-engineering process. In the Phaser case, we mined data from GitHub, since the system is hosted in this repository. Next we provide more details of the extraction for each system.

3.2.1 ArgoUML-SPL extraction

A set of scripts were implemented to create a dataset and to automate the analysis[11]. From the 156 revisions of the ArgoUML-SPL re-engineering process, we automatically identified the SVN revisions with Java code changes (51 revisions) and we automatically downloaded them. The rest of the revisions were mainly updates of the website content and SVN structure (e.g., empty folders' creation). We then extended the ground-truth extractor of the ArgoUML-SPL feature location benchmark (Martinez et al. 2018) to provide metrics on the LoC of each feature and we ran it for each of the revisions. We also identified, for each LoC, whether it was included as part of a feature or as part of the `Core`, i.e., the common part.

The initial revisions mention that the source code of the v0.28.1 release was used to extract the ArgoUML-SPL. However, until revision 41, the repository does not have all the source code, but only Java packages and classes where the annotations were included. Starting from revision 41 the source code is complete (both Core and parts with optional code) to build the software. To conduct our analyses in equal conditions for each revision, for revisions prior 41, we included the missing Java classes from v0.28.1.

3.2.2 Phaser extraction

For the Phaser case study, firstly we evaluated the custom build creation process. It is done via a Gruntfile[12] that lists all the mandatory and optional features with a short description of each one. Then, we investigated the "manifests"[13] folder that contains json files, and each json file lists a set of JavaScript source files (directory of each file). Each json file inside the manifests folder maps a distinct collection of JavaScript source files into different modules, and each of these modules we assumed to be the features of the system listed in the Gruntfile. By extracting the JavaScript files from each of the json files, we are able to map which source files belong to each of the system features.

[11] https://zenodo.org/record/5519966.

[12] More information at: https://gruntjs.com/sample-gruntfile

[13] More information at: https://webpack.js.org/concepts/manifest/

Firstly we needed to verify that our assumption was correct, thus we extracted the list of features from the Gruntfile and the list of names of json files. Then, we matched the features to their respective json files based on their names, followed by a manual validation of the content of each json file and the description of the features provided in the Gruntfile. There were 59 manifest files but 10 of them were stubs, and from the Gruntfile we extracted 43 features. Since the number of manifest files was higher than the number of features, it suggested that for Phaser the relationship between them was not one-to-one.

Most of the features had a json file with exactly the same name as the feature, thus we managed to successfully match all but three of the features from the Gruntfile with a distinct json file. However, there were nine json files that did not have a corresponding feature. The three remaining features were mapped to their corresponding json files by inspecting their content and the feature description with six json files remaining. Therefore, a manual evaluation of the extra json files in order to map to their respective features was required. Once again, we conducted this evaluation by analyzing the description of the features and the content of the extra json files.

We concluded that four out of the six remaining json files belonged to four distinct features that already had corresponding json files. From the remaining two json files, one corresponded to a feature - which was not explicitly included in the Gruntfile list of features - that is automatically added to the build if two other features are enabled. The other json file was a mandatory feature that was also not explicitly listed in the Gruntfile list of features.

Once each feature was mapped to its json files, we evaluated their evolution by measuring the total LoC, number of JavaScript files listed in the json file, whether the feature was listed in the Gruntfile or not, and if the feature was mandatory or optional. This process was done using a set of Python scripts and 17 Phaser releases (from 2.0.0 to 2.16.0) downloaded from the Phaser CE GitHub repository. We also used the software solution Perceval (Dueñas et al. 2018) for gathering information about the commits made during the same time frame.

3.3 Developer

We rely on the information that can be identified and analyzed from the version control system. In addition, we aimed to contact the developers involved in the process to confirm our findings or to request clarifications on certain aspects. For the ArgoUML-SPL case, we contacted all the developers, project managers, and maintainers involved in the re-engineering process. One of them is a co-author and the others were contacted by email or with personal interviews. For the Phaser case, we contacted the lead developer, Richard Davey[14] and “samme”[15], who is a very active developer that also responded to our questions[16].

4 Results and Analysis

This section presents an overview of the processes conducted to re-engineer ArgoUML and Phaser into SPLs, the results collected from the repositories of each system, and the analysis in order to answer the posed research questions.

[14]https://github.com/photonstorm

[15]https://github.com/samme

[16]Through Phaser’s Discord server (An online group-chatting platform https://discord.gg/phaser) and through email

4.1 Overview of the ArgoUML and Phaser re-engineering processes

Figure 3 presents an overview of the re-engineering steps to extract an SPL from ArgoUML. The developers used as input the ArgoUML CookBook[17] and the source-code for conducting four steps, represented by the gray rectangle in the figure. In the first step, the documentation found in the ArgoUML CookBook was studied to identify how the system architecture was designed and organized. This highlights the importance of the software architecture for the re-engineering process. Then, to have a better understanding of the actual system, a step for studying the source-code was also conducted. As a result of these two steps, the developers had information about the packages implementing ArgoUML features. Next, they started locating the features at the level of classes and methods. Finally, the developers annotated the source code.

As a result of the re-engineering process, eight features were made optional by introducing Java annotations (see Section 2.1), which are similar to C `ifdef` preprocessor directives but using a Java library for annotations[4]. Six features were related to diagrams and two were representative cross-cutting features, namely `Logging` and `Cognitive` support. The latter is responsible for providing critics and warnings regarding UML model instances. This re-engineering process was made in 156 commits, with contributions from three other developers. The latest revisions in 2014 consist only of fixes on the Java comments related to the granularity types of the variability annotations (meta-data on the variability annotations). Those revisions were made by a researcher on a study for consolidating ArgoUML variants in an SPL (Klatt 2014).

Figure 4 presents an overview of the process to introduce customization in the Phaser framework. The source of information used by developers for the re-engineering process were: (i) the game objects, that represent features such as sprites, animation, collision, and sound, to cite some; and (ii) the source code of Phaser 2.2.2.

As reported in the release notes[9], the framework suffered with code duplication, “God classes” structure, and “kitchen sink” approach. Then in the first phase of the process, the developers analyzed the features and source code in order to plan the re-engineering. In the second step, the features were modularized as Game Objects, with the purpose of turning the features into first-class citizens, i.e. individual JavaScript files, with regard to their capabilities. Next, stub classes and hooks were created to manage dependencies and essential code minimally required by the applications. Finally, feature toggles (Mahdavi-Hezaveh et al. 2021) were used to enable and disable features, allowing customized builds.

As mentioned in Section 2.4, the re-engineering process is followed by an SPL maintenance phase where the derived variants are more in-depth analyzed regarding their validity. For the two case studies, ArgoUML and Phaser, no automatic nor systematic means were introduced for SPL testing as part of the re-engineering process validation. In the case of ArgoUML-SPL, the existing source code of the tests (JUnit) were not included in the re-engineering process. However, as it will be presented in Section 4.3, the SPL maintenance phase identified and fixed bugs in some variants. It was much later, in 2020, when Fischer et al. (2020) made the effort of trying to re-structure the ArgoUML tests to make them aware and work for ArgoUML-SPL variability. However, they were not integrated in the ArgoUML-SPL repository by adding them to the ArgoUML-SPL derivation pipeline. The

[17]https://argouml-tigris-org.github.io/tigris/wiki-argouml/wiki

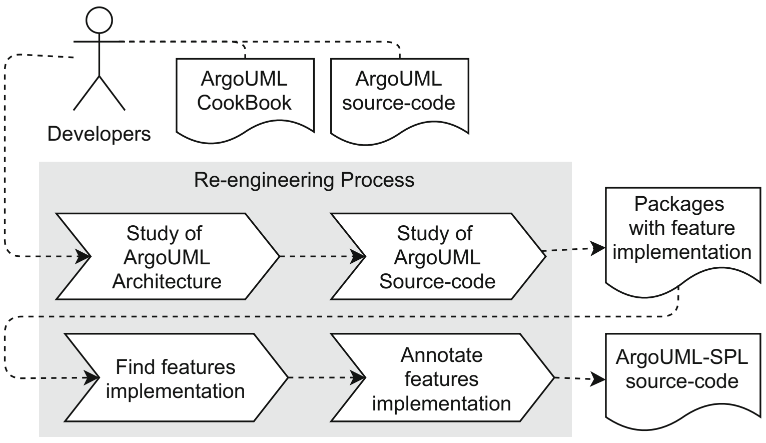

Developers
ArgoUML CookBook
ArgoUML source-code
Re-engineering Process
Study of ArgoUML Architecture
Study of ArgoUML Source-code
Packages with feature implementation
Find features implementation
Annotate features implementation
ArgoUML-SPL source-code

Fig. 3 Re-engineering process to extract ArgoUML-SPL from a monolithic version of ArgoUML, adapted from Couto (2010)

reported success rate of the tests was 100%. Instructions for deriving and executing variants are described in the ArgoUML-SPL feature location benchmark website[18] and so far (including configurations exercised by the authors of the present paper) no further compilation, runtime, or functional errors have been reported. In the case of Phaser, the authors of the present paper, also generated a set of distinct configurations, and tried them, but it is difficult to assess the existence of runtime bugs. Some inconclusive reports of runtime problems exist regarding the custom build configurations[19].

It is not straightforward to evaluate the extent at which deriving products in an SPL fashion requires lower effort than modifying the original single-system. First, modifications per variant (i.e., in a clone-and-own way) can be error-prone and the location of the removed features cannot be directly reused in next variants. In the case of ArgoUML, the effort will be similar to our reported effort for adding the variability annotations per feature. Successful SPL specific products from ArgoUML-SPL have been extensively used for research purposes, as referenced in Section 1. In the case of Phaser, the effort to remove features before the feature modularization process could be significant and error-prone. Successful Phaser custom builds have been used by the Phaser game development community when optimizing games with respect to the loading time. The use of a custom build for a given game can be checked by comparing the loaded phaser.min.js (which include a JavaScript comment on the version used) with the same file of the official archived release of Phaser[20]. As example, the Garden Tales game[21] used a v2.10.0 custom build[22] of 536,496 characters while the complete v2.10.0[23] weighted 824,392 characters.

[18]https://github.com/but4reuse/argouml-spl-benchmark
[19]https://www.html5gamedevs.com/topic/16720-custom-phaser-builds/
[20]https://phaser.io/download/archive
[21]http://phaser.io/news/2021/02/garden-tales
[22]https://games.softgames.com/garden-tales/js/custom-phaser.min.js
[23]https://github.com/photonstorm/phaser-ce/releases/download/v2.10.0/phaser.min.js

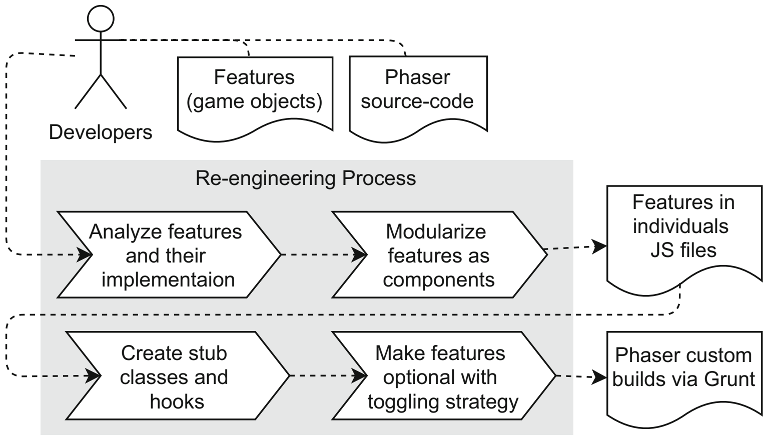

Developers
Features (game objects)
Phaser source-code
Re-engineering Process
Analyze features and their implementaion
Modularize features as components
Features in individuals JS files
Create stub classes and hooks
Make features optional with toggling strategy
Phaser custom builds via Grunt

Fig. 4 Re-engineering process to implement custom build on the Phaser framework

4.2 Making the Architecture Variable

For the ArgoUML, the evolution during the re-engineering process can be mainly analyzed by checking the changes in the repository. Figure 5 shows the activity in the repository from the day of the first revision until the latest. A vertical line, in October 2010, separates two periods that we manually identified analyzing the revisions: the SPL extraction itself with significant activity (135 revisions) and the SPL maintenance with sporadic commits (21 revisions). Since the extraction of ArgoUML was a Master thesis research project, the main developer (Marcus Couto) followed an explicit workflow for the SPL extraction, matching his Master course agenda. That is, Marcus Couto and his supervisors planned to extract the 8 optional features of ArgoUML-SPL in about 12 months. Once Marcus Couto

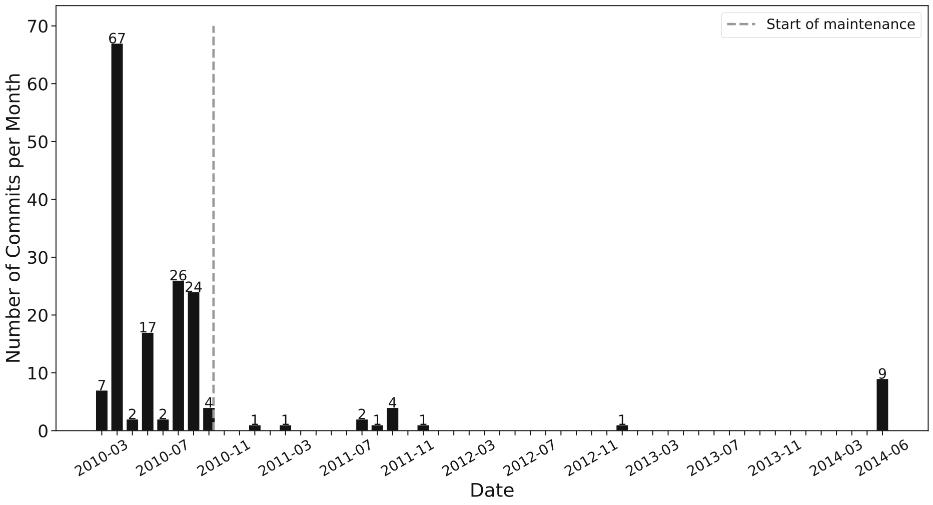


Fig. 5 Commits per month in ArgoUML-SPL extraction SVN

finished the extraction and concluded his Master's, we consider that ArgoUML started the SPL maintenance period.

Figure 6 shows the evolution of the source code while transitioning to a variability-rich system. In this case, the horizontal axis represents the different revisions where Java code changes were made to make the architecture variable; e.g., revision 12 (R12) is the first one with Java changes. Thus, Figure 6 shows time progression in terms of the relevant revisions. The vertical axis corresponds to the percentage of the total LoC of ArgoUML and this way we observe how the code base size evolves in terms of the Core feature and other optional features. A vertical line (R135) also separates the two periods and we can observe how the variable parts did not significantly change during the SPL maintenance period.

In the first revision, the code base corresponds almost completely to the core feature (non-optional part of ArgoUML). As the re-engineering process advances, new optional parts start to appear and grow. The final ArgoUML-SPL has eight optional features and 17 feature interactions, that are presented grouped in Fig. 6 as "Interactions". We can observe in Fig. 6 that after the last revision, the re-engineering process had an impact on 22.78% of the source code, which corresponds to the features that became optional. The core/common implementation of the ArgoUML-SPL represents 77.22% of the code. Measures on the LoC of each feature and its interactions are reported later in Table 2.

The transition to the custom build architecture was different for Phaser. The ability to create custom builds was introduced in version 2.3.0, and the subsequent versions either introduced new features or converted existing features into optional ones. In the first version with custom builds, i.e., 2.3.0, there were 21 optional features that grew to 31 until the release of version 2.5.0. The final list of features with their respective version in which they became optional is shown in Table 1.

In version 2.4.0, four optional features were added to the Gruntfile: `Rope`, `Tilesprite`, `Creature` and `Video`, while `Pixi` became a mandatory feature. Only `Creature` and `Video` were new features, `Rope` and `Tilesprite` were already modularized but were part of the `Pixi` feature. On the same commit that `Pixi` became a

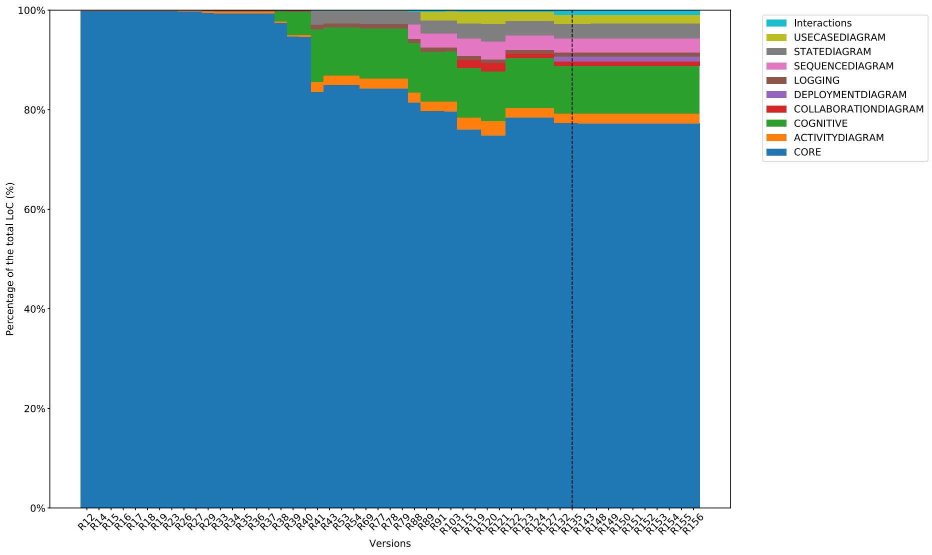


Fig. 6 Evolution of the common Core and variable part during the ArgoUML-SPL extraction

Table 1 Optional features and respective release version in Phaser

Feature	Version	Feature	Version
arcade	2.3.0	text	2.3.0
bitmapdata	2.3.0	tilemapcolision	2.3.0
bitmaptext	2.3.0	tilemaps	2.3.0
debug	2.3.0	tweens	2.3.0
gamepad	2.3.0	creature	2.4.0
graphics	2.3.0	rope	2.4.0
intro	2.3.0	tilesprite	2.4.0
keyboard	2.3.0	video	2.4.0
net	2.3.0	color	2.5.0
ninja	2.3.0	create	2.5.0
outro	2.3.0	dom	2.5.0
p2	2.3.0	flexgrid	2.5.0
particles	2.3.0	pixidefs	2.5.0
rendertexture	2.3.0	scale	2.5.0
retrofont	2.3.0	weapon	2.5.0
sound	2.3.0		

mandatory feature, both `Rope` and `Tilesprite` were extracted from the manifest file of feature `Pixi`, and became optional features on their own. However, both were composed of a game object component and a `Pixi` component. This was only addressed in later stages - in version 2.7.3, where the `Pixi` component was removed and its content merged with the game object component, in order to cut down the number of internal classes and inheritance. In version 2.5.0, seven features were added to the Gruntfile: `Scale`, `Dom`, `Create`, `Flexgrid`, `Color`, `Weapon` and `Pixidefs`. `Weapon` was the only new feature, while the others already existed and were modularized. `Scale`, `Dom`, `Create` and `Color` were modularized as stubs, which means that, although they are optional and in theory can be removed from the custom build, Phaser still requires certain functionalities from these features in order to work properly. Stub features contain just the bare minimum of functions that Phaser needs to work, greatly reducing the file size.

After version 2.5.0, the number of optional features remained constant throughout the versions. The number of mandatory features listed in the Gruntfile did not change after version 2.4.0, staying with 13 mandatory features during the evolution of the system. In version 2.4.0 the feature `Pixi`, which is a 2D WebGL renderer, became mandatory due to, according to Phaser's lead developer, the amount of customization made to the library in order to fix bugs. After that, it was not possible to swap with another `Pixi` version. However, on the next version it was removed from the explicit list of mandatory features in the Gruntfile, but remained a mandatory feature due to being automatically added to the build also via the Gruntfile.

The activity in the repository of Phaser is presented in Fig. 7, from the day when the modularization of the system started. The first vertical line, in March 2015, indicates the release of Phaser 2.3.0 allowing custom builds. The second vertical line highlights the release of Phaser 2.5.0, in June 2016, that is the last version in which new optional features were added. Comparing this figure with the monthly activity in the repository of ArgoUML-SPL (Fig. 5), we can see that Phaser has much more activity. This was expected, since Phaser

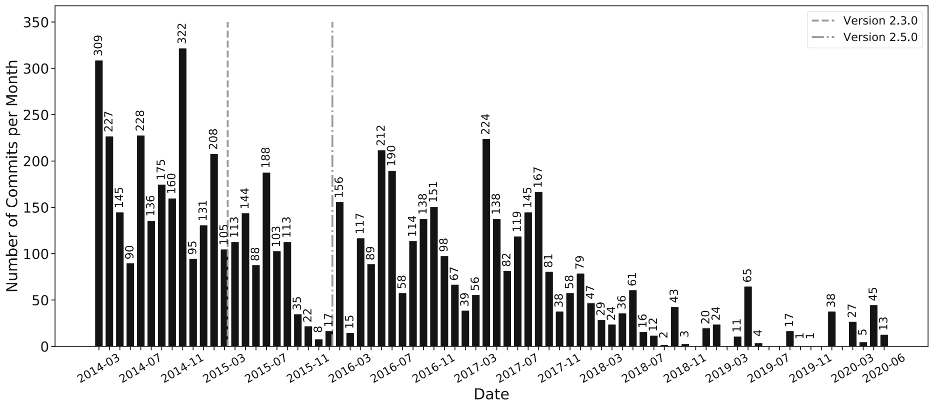


Fig. 7 Commits per month in Phaser CE

has a large community of developers and is widely used in practice. It is important to highlight that Phaser 2 was handed to the community on 9th January 2017, when it became the Phaser Community Edition. Since then, the main developer started working on developing Phaser 3, which was released on 18th February 2018. As he was the main contributor to the project, the amount of commits decreased following the release of Phaser 3.

Figure 8 presents the number of lines of code of each feature over the versions evaluated. The optional features that were already implemented and modularized but were not removable via Gruntfile in a specific version are represented by a bar with increased transparency. Since the first three versions did not possess the ability to create custom builds via Gruntfile, all the features are transparent. It is also worth noting that since the system was

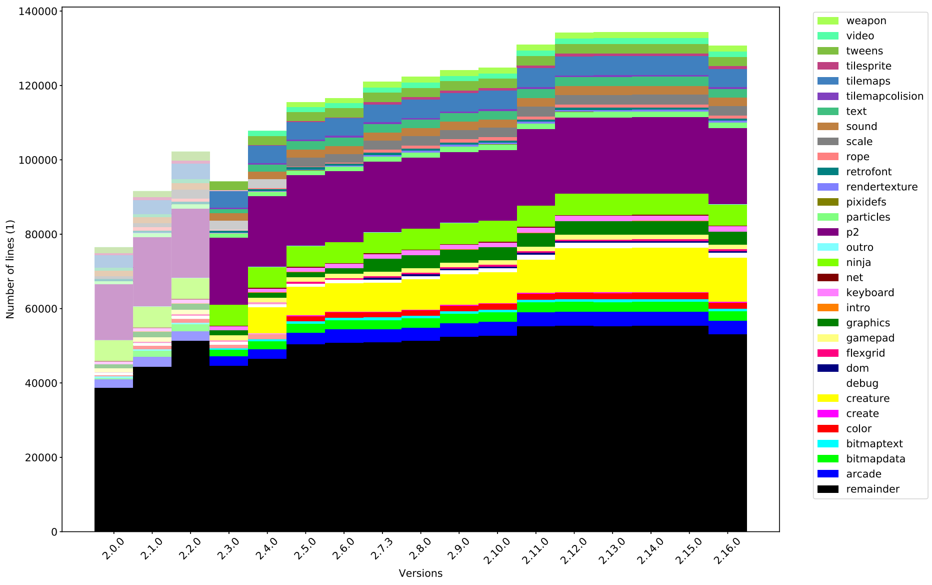


Fig. 8 Evolution of optional features of Phaser CE

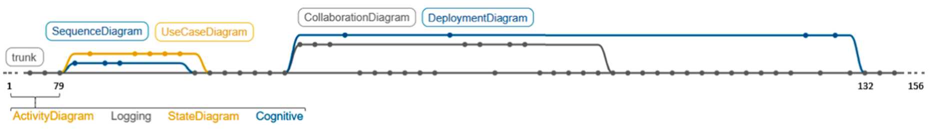


Fig. 9 Revision history graph of ArgoUML-SPL SVN. The eight extracted features were extracted in a separated branch and then merged in the trunk. This figure illustrates four branches from where information is available

also receiving new features over time, the vertical axis shows the actual number of lines of each feature instead of the percentage of the total lines of code like the chart presented for ArgoUML-SPL (Fig. 6).

The decrease in the number of lines from version 2.2.0 to version 2.3.0 called our attention (see Fig. 8). We could observe that there are two commits where many lines were removed. The first one relates to the heavy customization of the `Pixi` library mentioned by the lead developer. On this specific commit[24] some `Pixi` files that were no longer used in the Phaser build at the time, totaling 8100 deleted lines across 45 files. The second commit[25] refers to the restructuring of the core game objects and the use of the new components "mixins". There were over 4400 deletions across nine files.

4.3 Re-engineering Processes Operationalization

Analyzing the data of ArgoUML-SPL, we were able to analyze that the SPL transition consisted of "one feature at a time" process, using a branching strategy. Figure 9 shows the revision history graph regarding the created branches. For four features (`SequenceDiagram`, `UseCaseDiagram`, `CollaborationDiagram`, and `DeploymentDiagram`), feature-specific branches were used before merging. Also, for those branches, features were extracted in parallel (`SequenceDiagram` in parallel with `UseCaseDiagram`, and `CollaborationDiagram` with `DeploymentDiagram`). The extraction of `ActivityDiagram` started in R12, `Logging` in R17, and `Cognitive` and `StateDiagram` both in R33. According to the feedback from the main developer, the decision of creating branches per feature was taken since the beginning. However, we did not find evidence in the repository about branches for the first four features. Hence, it was probably made without adding the branches in the SVN.

The team involved in ArgoUML-SPL extraction was small. Marcus Couto[26] was in charge of all the re-engineering process, except the `UseCaseDiagram` and `SequenceDiagram` branches which were extracted by Camilo Ribeiro[27]. Eduardo Figueiredo[28] (an author of this paper) and Marco Tulio Valente[29] were two experts supervising the extraction, acting like project managers, but not interacting directly with the repository. Finally, Christian Kästner[30] reported variability-related bug fixes that were

[24]https://git.io/JtNGO
[25]https://git.io/Jz4S2
[26]https://github.com/marcusvnac
[27]https://github.com/camiloribeiro
[28]https://github.com/emagno and https://orcid.org/0000-0002-6004-2718
[29]https://github.com/mtov and https://orcid.org/0000-0002-8180-7548
[30]https://github.com/ckaestne and https://orcid.org/0000-0002-4450-4572

Table 2 Duration between the start of the extraction until the feature and their corresponding feature interactions got stable

Feature	Size F. core LoC Interact. LoC	Revisions (global)–branch–	Dates dd/mm/yyyy (global)	Months (global)
Activity	5,950 208	12 → 132	03/04/2010 – 27/09/2010	5.9
Cognitive	28,380 2,069	33 → 132 (33 → 143)	03/04/2010 – 27/09/2010 (07/10/2011)	5.9 (18.4)
Collaboration	2,676 199	95 → 132 (95 → 148)–95 → 115–	12/08/2010 – 27/09/2010 (08/06/2014)	1.53 (46.53)
Deployment	3,243 1,708	98 → 132 –98 → 132–	12/08/2010 – 27/09/2010	1.53
Logging	2,312 690	17 → 132 (17 → 152)	03/04/2010 – 27/09/2010 (16/06/2014)	5.9 (51.16)
Sequence	8,437 420	80 → 123 (80 → 148)–80 → 88–	21/06/2010 – 25/09/2010 (08/06/2014)	3.2 (48.26)
State	8,801 207	33 → 132	03/04/2010 – 27/09/2010	5.9
UseCase	5,294 60	81 → 132 (81 → 156)–81 → 89–	21/06/2010 – 27/09/2010 (28/06/2014)	3.26 (48.93)
			Average:	4.14

Results are provided for the SPL extraction and, separately, for the whole history including maintenance (global)

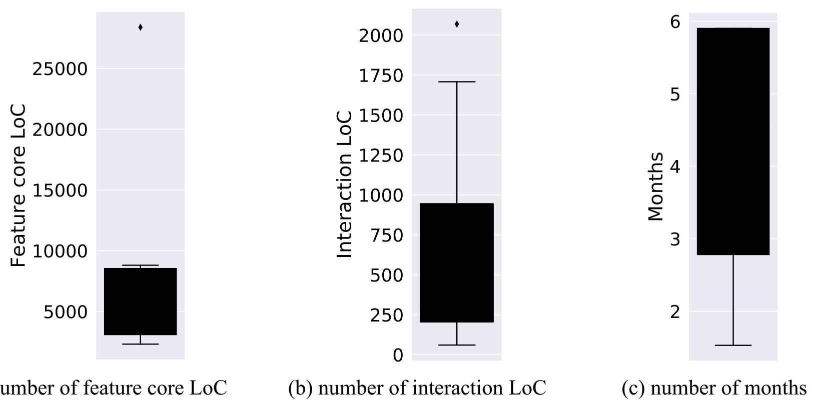


Fig. 10 Box-plots of the size, interactions, and duration of feature implementations

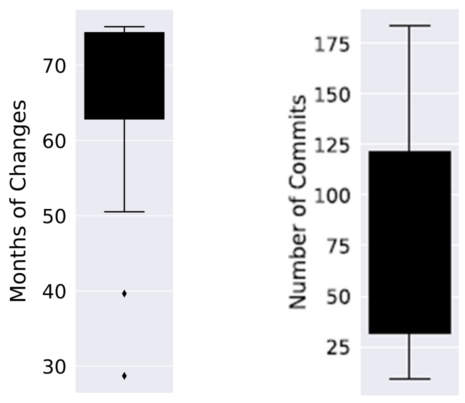
solved in R143, and Benjamin Klatt[31] improved the meta-data of the Java variability-related annotations.

To analyze how time-consuming was the extraction of ArgoUML-SPL we measure time from two dimensions. Firstly, the range of revisions in the repository since a feature started to be extracted until its last change or revision to one of its feature interactions (i.e., until the feature was completely stable). Secondly, the dates from this revision range are used to calculate the number of months. Both time dimensions, together, provide a more complete overview of the time involved in the extraction of each feature. Table 2 summarizes this information, including an analysis of the activity on the trunk for the feature under-study (i.e., LoC changes related to this feature or one of their feature interactions) during the revision range.

We can observe interesting facts in Table 2. For instance, `DeploymentDiagram` was the quickest extracted (only 1.53 months, 34 revisions). All the revisions were made in a dedicated branch and, after its merge into the trunk (see Figure 9), no more changes were made to this feature nor to any of its feature interactions. Contrary to this, `Logging` was the feature that took more time to be globally stable. This cross-cutting feature had a sustained activity in the trunk. The features `CollaborationDiagram`, `SequenceDiagram`, and `UseCaseDiagram` had significant activity after their branches merged, that means that their extraction was not completed between merging the branch and the start of the SPL maintenance period.

Regarding feature interactions, in the last revision we could observe 17 different types of interactions. Among these interactions, nine are related to the cross-cutting feature `Logging` interaction with some diagram type, three involve the cross-cutting feature `Cognitive` and a diagram type, and five are interactions only between diagrams. On average, each feature interaction has 2.64 revisions, commonly spread along the re-engineering process. The size of these interactions varies, ranging from interactions implemented with only one or two LoC, to one interaction with 1,653 LoC. Furthermore, Figs. 10a, 10b and 10c display the box-plots of the number of core LoC, interaction LoC and months.

[31] https://github.com/BenjaminKlatt



(a) number of months (b) number of commits

Fig. 11 Box-plots of the duration and number of feature changes

For the Phaser case, there was no branching for specific features. According to the lead developer, Richard Davey, all new features went into the "dev" branch and eventually were merged to "master" when ready. There were very few branches due to the lead developer's preferences, and none of them were for specific features.

About the developers' contributions, by evaluating the commits during the process of adding new optional features to the Gruntfile (up until version 2.5.0), we noticed a huge discrepancy between the main developer, who made over 2900 commits, while the developer with the second highest number of commits made only 150, nearly 20 times fewer commits. This directly relates to the fact that since most of the commits were made by the lead developer, the strategy of using branches was not used as he preferred using the strategy mentioned before.

Table 3 presents the optional features, the first and last version they were changed, the respective dates of first and last change, the number of months between the interval and the number of commits made. Additionally, Figs. 11a and 11b present the box-plots of the number of months and commits. Once the features were implemented, they were not changed over many releases, which means low volatility of features (i.e., many inclusions and exclusions). Interestingly, for Phaser the number of months to conduct the transition is quite long in comparison to ArgoUML-SPL. However, as we have mentioned, the transition of Phaser to an SPL was together with other common implementation and maintenance activities, differently from ArgoUML-SPL in which the developers focused only on extracting the SPL.

We measured the correlation between the number of months and the number of commits of the features by calculating the Spearman correlation coefficient, since our data does not follow a Gaussian distribution. We calculated the correlation coefficient with the help of the Scipy Python package[32]. With a correlation coefficient of 0.65 and a p-value of 7.4 x 10-5 (0.0074% probability of an uncorrelated system producing a dataset that has a

[32]https://docs.scipy.org/doc/scipy/index.html

Table 3 Version and duration between the start of the extraction until the feature and their corresponding feature interactions got stable

Feature	Versions	Dates (dd/mm/yyyy)	Months	Commits
arcade	2.0.0 → 2.15.0	05-03-2014 – 15-05-2020	74.35	184
bitmapdata	2.0.0 → 2.15.0	02-03-2014 – 15-05-2020	74.45	171
bitmaptext	2.0.0 → 2.15.0	06-03-2014 – 15-05-2020	74.32	71
color	2.0.0 → 2.15.0	25-03-2014 – 15-05-2020	73.69	63
create	2.3.0 → 2.15.0	08-07-2015 – 15-05-2020	58.25	22
creature	2.3.0 → 2.15.0	13-04-2015 – 05-15-2020	61.07	59
debug	2.0.0 → 2.15.0	02-03-2014 – 01-06-2020	75.01	145
dom	2.1.0 → 2.15.0	10-11-2014 – 15-05-2020	66.13	30
flexgrid	2.0.0 → 2.15.0	05-09-2014 – 15-05-2020	68.30	31
gamepad	2.0.0 → 2.15.0	14-03-2014 – 15-05-2020	74.05	40
graphics	2.0.0 → 2.15.0	06-03-2014 – 09-06-2020	75.14	72
intro	2.0.0 → 2.7.3	14-03-2014 – 03-07-2017	39.65	9
keyboard	2.0.0 → 2.15.0	03-03-2014 – 15-05-2020	74.42	74
net	2.0.0 → 2.15.0	25-03-2014 – 15-05-2020	73.69	14
ninja	2.0.0 → 2.10.0	06-03-2014 – 22-05-2018	50.53	33
outro	2.0.0 → 2.15.0	10-03-2014 – 15-05-2020	74.18	23
p2	2.0.0 → 2.11.0	06-03-2014 – 01-10-2018	54.86	124
particles	2.0.0 → 2.15.0	11-03-2014 – 05-15-2020	74.15	119
pixidefs	2.2.0 → 2.15.0	17-02-2015 – 15-05-2020	62.88	23
rendertexture	2.0.0 → 2.15.0	14-03-2014 – 15-05-2020	74.05	28
retrofont	2.0.0 → 2.15.0	11-03-2014 – 15-05-2020	74.15	33
rope	2.0.0 → 2.15.0	14-03-2014 – 15-05-2020	74.05	59
scale	2.0.0 → 2.15.0	10-03-2014 – 15-05-2020	74.19	155
sound	2.0.0 → 2.15.0	02-03-2014 – 29-05-2020	74.91	156
text	2.0.0 → 2.15.0	06-03-2014 – 15-05-2020	74.32	165
tilemapcolision	2.2.0 → 2.15.0	17-02-2015 – 15-05-2020	62.88	41
tilemaps	2.0.0 → 2.15.0	02-03-2014 – 29-05-2020	74.90	163
tilesprite	2.0.0 → 2.15.0	07-03-2014 – 15-05-2020	74.28	85
tweens	2.0.0 → 2.15.0	03-03-2014 – 15-05-2020	74.42	89
video	2.2.0 → 2.15.0	05-03-2015 – 15-05-2020	60.42	64
weapon	2.4.0 → 2.11.0	03-06-2016 – 25-10-2018	28.71	48
		Average:	67.76	77.19

Spearman correlation at least as extreme as the one computed from ours), indicating that the relationship between number of commits and months has a relatively high correlation.

4.4 Error-proneness of the Process

In the context of re-engineering ArgoUML and Phaser into an SPL, we could identify some problems during the conduction of the process. These problems are described per system in the following.

4.4.1 ArgoUML-SPL

Incomplete extraction. Around one year after finishing the ArgoUML-SPL re-engineering process, a revision was made with a message describing variability-related bugs pointed out by Christian Kästner. The bug fix consisted of seven Java classes where missing variability annotations for the Cognitive feature were added.[33] Two of the Java classes were completely specific for the Cognitive feature, and in the other classes, the granularity was at method and statement level. This feature is particularly difficult to locate, giving its cross-cutting behavior. Failing to annotate all the source code related to a feature can lead to compile- or runtime-errors, unexpected or incorrect behaviors, or the presence of dead code. These bugs were found by a tool called LEADT (Location, Expansion, And Documentation Tool), proposed by `?leadt,kastnerleadt` (). ArgoUML-SPL was used as a subject system to evaluate this tool, finding the bugs that were reported and fixed. This is a lesson learned about how, as in other software engineering projects, unnoticed bugs might appear. As mentioned in Section 4.3, some issues have been treated as well after merging a feature branch in the trunk during the extraction period.

Lack of tool support for variability consistency checks. Feature extraction is an error-prone task. We can see how a typo `SEQUENCEIAGRAM` (missing `D`), introduced inconsistencies in the system in R88 and R89. In addition, we observe evidence of feature name refactorings, e.g., `UMLSTATEDIAGRAM` to `STATEDIAGRAM` in R119, and how it affects also its feature interactions. More advanced tool support for checking the consistency of variability information and for variability-aware refactoring will be desired. In this sense, tool capabilities could have helped in solving these issues. As mentioned in Section 2.1, the used tool in ArgoUML-SPL is Java Prepocessor (Javapp) which is not part of Java itself but an external tool that parses Java source code identifying and handling comments with specific keywords for variability management. This parsing and handling are only considered at derivation-time. Thus, being standard Java comments, no keywords highlighting, auto-completion, nor consistency checks for the feature names is performed so the feature naming error was unnoticed during a couple of revisions. Javapp is a very basic variability management tool and its major advantage is that no external dependencies need to be added to the Java source code itself for the derivation of variants. However, generally speaking, it is acknowledged that SPL engineering lack of mature tool support (Horcas et al. 2019). For instance, annotative approaches in Java are supported in FeatureIDE (Meinicke et al. 2017) (v3.8.0) through its integration with Antenna and Munge. Using FeatureIDE with Antenna could have identified the consistency check issue of the feature name through a warning in the source code, but FeatureIDE with Munge does not have current support for it. Regarding name refactoring, both Antenna and Munge allow renaming features with automatic synchronization in the variability annotations.

4.4.2 Phaser

Managing feature dependencies As mentioned by Richard Davey, the biggest challenge to make features optional was resolving dependency issues between modules. In version 2.5.0, seven new optional features were added to the Gruntfile. For illustration, we highlight features `Scale` and `Flexgrid`. `Scale` is a manager that handles the scaling, resizing and

[33] https://github.com/argouml-tigris-org/argouml-spl/commit/a6014294bcd26376ce34089590e6e69267cd01bc

alignment of the game size and `Flexgrid` is a grid manager that works in conjunction with `Scale`. Before being added as an optional feature, `Scale` created a `Flexgrid` object during `ScaleManager` initialization routine. This relationship was changed in version 2.5.0 in order that `ScaleManager` no longer creates a `Flexgrid` object if the class is not available – excluded via custom build for example. In order to implement the custom builds with different components, the developers have used feature toggles in some situations. For example, the feature `Scale` has a feature toggle checking regarding the existence of the feature `Flexgrid`. Other examples are in the implementation of the mandatory features `Input` and `Core`. The former checks if features `Keyboard` and `Gamepad` are available and the latter checks if the feature `Graphics` is available. This indicates that in order to make a feature truly optional, it is pivotal to deal with features that depend on it or interact with it, either by implementing cross-tree constraints or removing these dependencies by making use of feature toggles. Otherwise, the derived product would not function properly, since the removal of a certain feature might break another feature that depends on it.

Lack of feature constraints mechanism It is good practice to create cross-tree constraints in an SPL. If an optional feature requires another optional feature to work properly, it should not be possible to instantiate a custom build with an optional feature without its dependencies. Although there is not any sort of list of dependencies between features with easy access for the users of Phaser, there is a dependency resolver in the Gruntfile for a subset of features. The Gruntfile script excludes certain features that the user has not excluded that depend on other features that the user actually excluded. For example, if the user excluded the feature `Arcade Physics` but did not exclude `Particles` or `Weapon`, the script automatically removes them and warns the user through the console. Other than the ones mentioned before, there is a third one that removes `RetroFonts` if the user excluded `RenderTexture` as, similarly, `RetroFonts` depends on `RenderTexture`.

The constraints mentioned only remove features that the developer most likely "forgot" to remove, since according to one developer, to make custom builds the users should be familiar with the system and should know what they can remove and what they can not. However, there is a special case where if the user did not exclude `Arcade Physics` nor `Tilemaps`, the feature `Timelap Collision` is added to the build. This feature is not listed in the Gruntfile to be excluded and is a compositional feature since it was part of the arcade physics feature and was extracted from it into its own feature.

Lastly, it is possible to use stubs of certain features to greatly reduce the size of the files. In order to make certain features optional, if the user excludes them when making a custom build, instead of completely removing the feature, the stub version of the feature is used instead. The stubs contain only the bare minimum of functions that Phaser needs to work. However, there are some constraints for using stubs that were only specified in the documentation of the commit where they were introduced. For example, developers should not remove the feature DOM if they are using the full `ScaleManager`, which is the Scale feature.

These untreated constraints rely on the hypothesis that since making a custom build is an advanced activity, the user must know what he/she is doing. In order to remove features from the build the user knows why he/she is removing that specific feature and the implications of removing it. However, this is not a good practice and it is highly error-prone.

5 Discussion

Next, the RQs presented in Section 3 are answered.

RQ1: How has the source code of the systems changed during the re-engineering processes?

Answer: There is a great difference between the re-engineering process of ArgoUML-SPL and Phaser to make the architecture variable. The source code of ArgoUML was incrementally changed in 135 revisions to achieve an initial SPL architecture, which was then maintained until revision 156. ArgoUML-SPL has only eight optional features, corresponding to 22.78% of code. On the other hand, for Phaser, the source code modifications were directed towards handling dependencies between modules in order to make them optional, since the feature modules were already modularized. This modularization was observed in the size of the Phaser core (<40%), as the number of lines does not change drastically, unlike in the ArgoUML-SPL case due to the extraction of features from the core. Phaser was composed of 31 optional features introduced along three different releases. Differently from ArgoUML-SPL that had the type of the features established beforehand, for Phaser a feature was optional and became mandatory during the transition. Also, there were some feature refinements, in which features were extracted/decoupled from other features.

RQ2: How was the operationalization of the re-engineering processes?

Answer: For ArgoUML-SPL, the developers adopted a big bang re-engineering process in which developers relied on a stable version of the system and focused exclusively on performing the transition to an SPL. Phaser followed a “strangler pattern” in which the architecture was incrementally modified together with other common implementation and maintenance activities of the systems’ life cycle. The developers in charge of ArgoUML-SPL transition started using feature-specific branches to perform modifications and then merging them into the trunk. In the case of Phaser, there were no feature branches, only a “dev” branch that was used for all modifications and then merged into the master branch. Regarding the time for the transitions, each feature of ArgoUML-SPL took 4.14 months on average, while for Phaser it took 67.76. For both systems, few developers were in charge of the re-engineering process, despite the large community of Phaser.

RQ3: What were the problems faced when conducting re-engineering processes of the systems?

Answer: The problems found during the re-engineering process of ArgoUML-SPL were related to *incomplete extraction*, resulting of the manual process of re-engineering a system into an SPL. Also, we identified a problem related to *variability consistency*, in which the name of a feature was misspelled and required a refactoring operation. For Phaser, developers pointed out the complexity for *managing feature dependencies*, what is a common problem for variability management with compositional approach (see Section 2.3). Another problem is related to *feature constraints*, as the Phaser developers have not used a variability model to manage feature constraints. Based on this, the constraints were managed mainly based on the expertise of developers.

6 Threats to Validity

One of the main concerns when performing case studies like ours is the validity of results and their applicability to other contexts. We discuss some threats to the study validity based on well-known guidelines and four categories of validity threats (Wohlin et al. 2012).

Construct Validity The construct validity reflects to what extent the operational measures that are studied represent what the researchers have in mind and what is investigated according to the research questions (Wohlin et al. 2012). In our study, the first and foremost threat to construct validity concerns the choice of the analyzed projects. We opt to select two very different projects to favor external validity, as discussed below. We also rely on project activity metrics, such as number of commits, branches, and bugs, as proxies for the extraction processes. Despite not being designed for SPLs, they are fine-grained metrics that allowed us to have a deep understanding of the processes. Also, to mitigate this threat, we complemented and confirmed our analysis with feedback from the developers. Additionally, these metrics have been used for evaluating characteristics of features in highly-configurable systems (Michelon et al. 2021). However, it is acknowledged that there is a lack of metrics for the evolution of SPLs (Marques et al. 2019). In summary, we believe that this threat is minimal and that our metrics capture, to some extent, the extraction process.

Internal Validity The internal validity is related to uncontrolled aspects that may affect the study results (Wohlin et al. 2012). The execution of the study steps is a threat to the internal validity, since a poor execution may result in the collection of incorrect data. For instance, in the Phaser case study, we extracted the list of features from the Gruntfile and manifest files. However, the Phaser developers may have documented configuration knowledge in other documents which we are not aware of. Furthermore, compared to the ArgoUML development, Phaser SPL extraction occurred while it received new features. This may have polluted the data we collected. To mitigate these threats, we have contacted developers and researchers involved in the development and extraction of both systems to verify the best way of collecting correct data. As described in Section 3, we clarify that we designed our study based on evidences of the re-engineering process, mainly the source code repository where the re-engineering process took place. We used the developers mainly for clarifications or confirmations of our findings and analyses. We did this to mitigate the possible subjectivity of relying only on the developers, in particular when both SPL re-engineering processes took place several years ago, and they might have forgotten many details about it.

External Validity The external validity concerns the ability to generalize the results to other environments, such as to industry practices (Wohlin et al. 2012). The diversity of the projects, and their organizational factors, represent a relevant threat to external validity. Regarding this validity threat, we focused the analysis on two open-source projects, namely ArgoUML and Phaser. They are large and popular projects, including one that is highly investigated in several research papers (ArgoUML, as it will be presented in next Section 7 on related work) and one that is maintained by an active community of software developers (Phaser). Based on that, the study of these two systems helps to understand the phenomenon we are investigating. The results can be generalized to projects with similar characteristics.

Conclusion Validity The conclusion validity concerns with issues that affect the ability to draw the correct conclusions from the study (Wohlin et al. 2012). These results reflect our perceptions and interpretations of the metrics collected from the ArgoUML and Phaser projects. For instance, we concluded that the re-engineering process of Phaser required a higher effort than the re-engineering process of ArgoUML-SPL based on source code changes to make modular features and turn them into optional, aggravated with the identified fact that the re-engineering process was in parallel with maintenance and evolution activities. However, this conclusion could be different if other metrics were analyzed. To mitigate the bias of relying on the interpretations of a single person, all authors participated in the data analysis process and discussions on the main findings. Nonetheless, there may be several other important aspects in the data collected, not yet discovered or reported by us.

7 Related Work

For a further investigation of pieces of work on the topic of extracting SPLs from monolithic systems, we rely on information available at ESPLA[34] catalog (Martinez et al. 2017). ESPLA is a catalog of case studies on extractive software product line adoption that is collaboratively maintained by the SPL community.

ArgoUML extensive use ArgoUML is one of the most common case studies used in research of extractive adoption of SPLs (Assunção et al. 2017; Martinez et al. 2017). Following the original paper of Couto et al. (2011) from 2011, which describes the extraction of ArgoUML-SPL, many other pieces of work also deal with the same system. Regarding the tasks and artifacts involving ArgoUML, the great majority of papers present feature location techniques that were evaluated using the system source code. Overlap analysis among ArgoUML variants source-code is presented in several papers (Eyal-Salman et al. 2013c; Linsbauer et al. 2013; Ziadi et al. 2012). For instance, Eyal-Salman et al. have studied feature location in ArgoUML based on formal concept analysis, information retrieval, and hierarchical clustering (AL-Msie'deen et al. 2013; Eyal-Salman et al. 2013a, b, 2014). A more specific work presented by Martinez et al. (2016) has focused on name suggestions during feature location. Klatt et al. (2014) used ArgoUML as case study to deal with dependency analysis for consolidating customized product copies. Still using source-code, Fischer et al. (2014) proposed an approach for statically analyzing ArgoUML variants allowing the composition of new products. In another recent work, Michelon et al. (2021) proposed a hybrid strategy based on static and dynamic analysis for feature location in the source code of ArgoUML, that was later compared to feature location techniques traditionally used for fault location (Michelon et al. 2021).

Few studies are not based on source-code. For instance, Linsbauer et al. (2014) relied on requirements, mainly related to configuration of features in variants of ArgoUML, as a basis for reverse engineering feature models. Martinez et al. (2015) reverse engineered UML class models from ArgoUML variants for conducting automatic extraction of model-based SPLs. ArgoUML was used to evaluate the scalability of the approach given its size. Schultheiß et al. also used the UML class models to evaluate n-way model matching techniques that are meant to support variability mining (Schultheiß et al. 2021). The previously mentioned works were interested in technical aspects, thus, they did not investigate the original re-engineering process, nor they compared the effort required for the original extraction. In this work, we are providing such a baseline for comparison. Feature location is a key activity for later integrating the variability mechanism, and given that, in the original extraction, no refactorings were needed to add the variability source code annotations, the effort reported in this paper on the extraction process can be used for both feature location and extraction process comparisons.

[34] https://but4reuse.github.io/espla_catalog/

Extraction process analysis Other pieces of work related to this present study are based on the analysis of the duration and number of people for conducting SPL extraction processes. Table 4 presents the studies found on ESPLA catalog that describe these pieces of information.

Kolb et al. (2006) describe the duration and number of people for improving the design and implementation of an industrial Image Memory Handler (IMH) used in office appliances, such as copier machines, printers, and multi-functional peripherals. Their goal was to reuse IMH in an SPL. The process, mainly based on feature identification and refactorings, took four months and was conducted by five people. The process was not very long because the C source code was already annotated with `ifdefs`. Three open-source forum systems were used by Yang et al. to manually identify features (Yang et al. 2009). During two weeks two graduate students manually performed feature location on Java code and SQL statements. The goal was to synthesize a variability model.

We found seven SPL extraction case studies in four different works published in 2011. For instance, Ali et al. (2011) dealt with Java code of two open-source systems to be decomposed in features to be used in mobile devices. The authors call it miniaturization of a given system. Basically, they performed feature identification, feature constraints discovery, and extraction of reusable assets. The process took approximately 171 hours to recover the 830 feature traces for SIP and 135 hours to recover the 318 feature traces for Pooka. In the paper of Dhungana et al. (2011), three industrial case studies were the basis for manual analysis. These authors also performed feature identification, feature constraints discovery, and extraction of reusable assets. These case studies took longer to conduct the process, namely four years for Siemens VAI Steel Plant Automation Software, 2.5 years for IEC 61499 Industrial Automation System, and 1.5 year for BMD .NET-based Business Software.

Alcatel-Lucent IXM-PF, an industrial telecom software product family that had been developed and maintained for more than ten years, was studied by Zhang et al. (2011). This case study was used to incrementally extract an SPL from their variants. This process took at least 1.5 years to achieve an initial success. The three studies aforementioned do not describe the number of people involved in the re-engineering process. The industrial case study of Fujitsu Kyushu Network Technologies described by Otsuka et al. (2011) also had a long duration, namely more than one year and 100 engineers on average, about 300 engineers at a maximum, in development sites in four geographically distant locations. Similar to previous studies, here the authors conducted manual work, with face to face meetings, to conduct feature identification, feature location, and construction of reusable assets. Olszak and Jørgensen (2012) dealt with BlueJ, an open-source project. The authors performed feature identification in two hours, by inferring almost all program features from the available user documentation. Then, they automatically create a class-preserving decomposition, while leaving the decision about any further manual separation up to the developers. Another study to identify features, and additionally feature recommendation, was conducted by Hariri et al. (2013). The process relies on expert knowledge and manual effort, requiring 30 hours. There is no mention related to people involved in the process.

Related works summary As a summary, based on related work presented in this section, we can see that ArgoUML is a case study extensively used for different purposes in the context of extractive adoption of SPLs. Open-source software is commonly used as a subject system for related studies in the topic of re-engineering application into SPLs. However,

Table 4 Studies that mention the effort for extracting SPLs from legacy systems

Ref.	Year	Case Study	Duration	People
Kolb et al. (2006)	2006	Image Memory Handler (IMH)	4 months	2 researchers, 2 students, 1 domain expert
Yang et al. (2009)	2009	JForum, JGossip and MVNForum	2 weeks for analysis of all the three systems	2 graduate students
Ali et al. (2011)	2011	SIP Communicator	171 hours to recover 830 feature traces	n/a
Ali et al. (2011)	2011	Pooka Email Client	135 hours to recover 318 feature traces	n/a
Dhungana et al. (2011)	2011	Siemens VAI Steel Plant Automation Software	4 years of case study	n/a
Dhungana et al. (2011)	2011	IEC 61499 Industrial Automation System	2.5 years of case study	n/a
Dhungana et al. (2011)	2011	BMD .NET-based Business Software	1.5 years of case study	n/a
Zhang et al. (2011)	2011	Alcatel-Lucent IXM-PF	1.5 years for an initial success	n/a
Otsuka et al. (2011)	2011	Fujitsu Kyushu Network Technologies	more than 1 year	100 engineers on an average
Olszak and Jørgensen (2012)	2012	BlueJ	2 hours for feature identification	n/a
Hariri et al. (2013)	2013	Collaborative Software Suite (CoSS)	30 hours to identify approximately 120 coarse-grained features	n/a

to the best of our knowledge, our study is the first one to investigate Phaser, a real-world case to make a system variable. We can find in the literature some other pieces of work describing the effort related to the duration and number of developers to conduct the re-engineering process. The industrial case studies are those that had a longer duration. Few works describe the people that participated in the process. Despite the existence of many studies in the topic of re-engineering legacy systems into SPLs, the most recent ones do not provide these pieces of information about the effort for the extraction (Assunčão et al. 2017; Martinez et al. 2017).

8 Conclusions

This paper revisited the transition processes from monolithic architecture into SPL of two open-source software systems, namely ArgoUML and Phaser. We aim at providing insights on the process for further researchers and practitioners on the effort required and the common problems in this kind of re-engineering process. Based on data mined from SVN and Git repositories used during the re-engineering, we analyzed details on how these processes were conducted and the SPL extraction impact on source code.

The results show that there is a great difference between the re-engineering process of ArgoUML-SPL and Phaser. For instance, almost one fourth of the ArgoUML source code was changed to tune features to optional, while for Phaser it was more than half of the system. ArgoUML-SPL used a branching strategy and Phaser used only a development branch and different releases. The transition of ArgoUML-SPL was "one feature at a time" in a big bang strategy, on the other hand, the transition of Phaser was together with other engineering activities and the type (mandatory or optional) changed for one feature. However, in both cases, only a few developers were in charge of the re-engineering process. Problems were found in the SPL extraction of both cases, related to lack of tools that led to incomplete and inconsistent feature extractions, complexity on managing feature dependencies when using compositional approach, and issues of not having a variability model to deal with feature constraints.

We are aware that two case studies are not enough to obtain representative effort estimations that can be used in general cases. Therefore, as further work, we aim to compare the results with other SPLs extracted from monolithic architectures with the goal of aggregating them towards more accurate effort and cost estimation means. Apart from that, we reported two extraction processes and the decisions that the involved persons took with its advantages and issues. This was reflected in the paper as it corresponds to our research questions. However, more case studies are also needed for providing comprehensive and concrete guidelines to support the decision-making process when facing SPL re-engineering projects.

Acknowledgements We thank Open-Source committers M. Couto, M. T. Valente, B. Klatt and C. Kästner for their feedback in ArgoUML-SPL, and R. Davey for his feedback in Phaser. This research was partially funded by CNPq, grant no. 408356/2018-9; FAPPR, grant no. 51435; and FAPERJ PDR-10 Fellowship 202073/2020.

Funding Open access funding provided by Johannes Kepler University Linz.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

AL-Msie'deen R, Seriai A, Huchard M, Urtado C, Vauttier S, Salman HE (2013) Feature location in a collection of software product variants using formal concept analysis. In: Safe and secure software reuse. Springer, Berlin, pp 302–307

Ali MS, Babar MA, Schmid K (2009) A comparative survey of economic models for software product lines. In: 2009 35Th euromicro conference on software engineering and advanced applications, pp 275–278

Ali N, Wu W, Antoniol G, Di Penta M, Guéhéneuc YG, Hayes JH (2011) Moms: Multi-objective miniaturization of software. In: 27Th IEEE int. Conf. on software maintenance (ICSM), pp 153–162

Apel S, Kastner C, Lengauer C (2009) FEATUREHOUSE: Language-independent, automated software composition. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09. IEEE Computer Society, Washington, pp 221–231

Apel S, Kolesnikov S, Siegmund N, Kästner C, Garvin B (2013) Exploring feature interactions in the wild: the new feature-interaction challenge. In: Proceedings of the 5th International Workshop on Feature-Oriented Software Development, pp 1–8

Assunčão WKG, Lopez-Herrejon RE, Linsbauer L, Vergilio SR, Egyed A (2017) Reengineering legacy applications into software product lines: a systematic mapping. Empir Softw Eng 22(6):2972–3016

Assunção WK, Vergilio SR, Lopez-Herrejon RE (2020) Automatic extraction of product line architecture and feature models from uml class diagram variants. Inf Softw Technol 117:106198

Babar MA, Chen L, Shull F (2010) Managing variability in software product lines. IEEE Softw 27:89–91

Batory DS, Sarvela JN, Rauschmayer A (2004) Scaling step-wise refinement. IEEE Trans Softw Eng 30(6):355–371

Berger T, Rublack R, Nair D, Atlee JM, Becker M, Czarnecki K, Wasowski A (2013) A survey of variability modeling in industrial practice. In: The 7th int. Workshop on variability modelling of software-intensive systems, vamos '13. ACM, Pisa, pp 7:1–7:8

Booch G (2005) The unified modeling language user guide. Pearson Education India

Couto M (2010) Extracting software product lines: A case study using conditional compilation. Master's thesis, Pontifical Catholic University of Minas Gerais. https://homepages.dcc.ufmg.br/~mtov/diss/2010_marcus.pdf

Couto MV, Valente MT, Figueiredo E (2011) Extracting software product lines: A case study using conditional compilation. In: 15Th european conference on software maintenance and reengineering, CSMR 2011. IEEE Computer Society, Oldenburg, pp 191–200

Czarnecki K, Eisenecker UW (2000) Generative programming - methods, tools and applications. Addison-Wesley. http://www.addison-wesley.de/main/main.asp?page=englisch/bookdetails&productid=99258

Dhungana D, Grünbacher P, Rabiser R (2011) The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. Autom Softw Eng 18(1):77–114

Dit B, Revelle M, Gethers M, Poshyvanyk D (2013) Feature location in source code: a taxonomy and survey. J Softw Evol Process 25(1):53–95

Dueñas S, Cosentino V, Robles G, Gonzalez-Barahona JM (2018) Perceval: software project data at your will. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ACM, pp 1–4

Engström E., Runeson P (2011) Software product line testing – a systematic mapping study. Inf Softw Technol 53(1):2–13

Eyal-Salman H, Seriai AD, Dony C (2013a) Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. In: IEEE 14Th international conference on information reuse integration (IRI), pp 209–216

Eyal-Salman H, Seriai AD, Dony C (2013b) Feature-to-code traceability in legacy software variants. In: 39Th euromicro conference on software engineering and advanced applications, pp 57–61

Eyal-Salman H, Seriai AD, Dony C, Al-msie'deen RA (2013c) Identifying Traceability Links between Product Variants and Their Features. In: REVE: Reverse Variability engineering, Genova, pp 17–22

Eyal-Salman H, Seriai AD, Dony C (2014) Feature location in a collection of product variants: Combining information retrieval and hierarchical clustering. In: SEKE: Software Engineering and knowledge engineering, Vancouver, pp 426–430

Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (2014) Enhancing clone-and-own with systematic reuse for developing software variants. In: IEEE International conference on software maintenance and evolution, pp 391–400

Fischer S, Michelon GK, Ramler R, Linsbauer L, Egyed A (2020) Automated test reuse for highly configurable software. Empir Softw Eng 25(6):5295–5332

Hariri N, Castro-Herrera C, Mirakhorli M, Cleland-Huang J, Mobasher B (2013) Supporting domain analysis through mining and recommending features from online product listings. IEEE Trans Softw Eng 39(12):1736–1752

Hassan AE (2008) The road ahead for mining software repositories. In: 2008 Frontiers of software maintenance. IEEE, pp 48–57

Horcas JM, Pinto M, Fuentes L (2019) Software product line engineering: a practical experience. In: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A, SPLC '19, pp 164–176

Hu Y, Merlo E, Dagenais M, Lague B (2000) C/c++ conditional compilation analysis using symbolic execution. In: 30Th int. Conference on software maintenance, ICSM '00. ACM, New York

Kang K, Cohen S, Hess J, Novak W (1990) Peterson, a.: Feature-Oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, SEI CMU

Kästner C, Apel S, Kuhlemann M (2008) Granularity in software product lines. In: 30Th int. Conference on software engineering, ICSE '08. ACM, New York, pp 311–320

Kästner C, Dreiling A, Ostermann K (2011) Variability Mining with LEADT. Tech. Rep. 01/2011, Department of Mathematics and Computer Science, Philipps University Marburg, Marburg. http://www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk

Kȧstner C, Dreiling A, Ostermann K (2014) Variability mining: Consistent semi-automatic detection of product-line features. IEEE Trans Softw Eng 40(1):67–82

Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier J, Irwin J (1997) Aspect-Oriented programming. In: ECOOP, pp 220–242

Klatt B (2014) Consolidation of customized product copies into software product lines. Ph.D. thesis, Karlsruhe Institute of Technology, Germany. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000043687

Klatt B, Krogmann K, Seidl C (2014) Program dependency analysis for consolidating customized product copies. In: IEEE International conference on software maintenance and evolution, pp 496–500

Knodel J, Muthig D (2005) Analyzing the product line adequacy of existing components. In: 1St int. Workshop on reengineering towards product lines. r2PL, pp 21–25

Kolb R, Muthig D, Patzke T, Yamauchi K (2006) Refactoring a legacy component for reuse in a software product line: a case study. J Softw Maintenance Evol Res Practice 18(2):109–132

Krueger CW (2001) Easing the transition to software mass customization. In: Software product-family engineering, 4th int. workshop, PFE 2001, Bilbao. Revised Papers, Lecture Notes in Computer Science, vol 2290, pp 282–293. Springer

Laguna MA, Crespo Y (2013) A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. Sci Comput Programm 78(8), 1010–1034. Special section on software evolution, adaptability, and maintenance - Special section on the Brazilian Symposium on Programming Languages

Linsbauer L, Lopez-Herrejon ER, Egyed A (2013) Recovering traceability between features and code in product variants. In: 17Th international software product line conference, SPLC '13. ACM, New York, pp 131–140

Linsbauer L, Lopez-Herrejon RE, Egyed A (2014) Feature model synthesis with genetic programming. In: Search-based software engineering. Springer International Publishing, Cham, pp 153–167

Mahdavi-Hezaveh R, Dremann J, Williams L (2021) Software development with feature toggles: practices used by practitioners. Empir Softw Eng 26(1)

Marques M, Simmonds J, Rossel PO, Bastarrica MC (2019) Software product line evolution: A systematic literature review. Inf Softw Technol 105:190–208. https://doi.org/10.1016/j.infsof.2018.08.014. https://www.sciencedirect.com/science/article/pii/S0950584918301848

Martinez J, Assunċȧo WKG, Ziadi T (2017) ESPLA: A catalog of extractive SPL adoption case studies. In: Proceedings of the 21st Int. Systems and Software Product Line Conference, SPLC 2017, vol B. ACM, Sevilla, pp 38–41

Martinez J, Ordoñez N, Tërnava X, Ziadi T, Aponte J, Figueiredo E, Valente MT (2018) Feature location benchmark with argouml SPL. In: Proceeedings of the 22nd int. Systems and software product line conference - volume 1, SPLC 2018. ACM, Gothenburg, pp 257–263

Martinez J, Wolfart D, Assunċào WKG, Figueiredo E (2020) Insights on software product line extraction processes: argoUML to argoUML-SPL revisited. In: SPLC '20: 24Th ACM international systems and software product line conference, vol A. ACM, Montreal, pp 6:1–6:6

Martinez J, Ziadi T, Bissyandé TF, Klein J, le Traon Y (2015) Automating the extraction of model-based software product lines from model variants (t). In: 30Th IEEE/ACM international conference on automated software engineering, pp 396–406

Martinez J, Ziadi T, Bissyandé TF, Klein J, Traon YL (2016) Name suggestions during feature identification: The variclouds approach. In: 20Th international systems and software product line conference, SPLC '16. ACM, New York, pp 119–123

Meinicke J, Thüm T, Schröter R, Benduhn F, Leich T, Saake G (2017) Mastering Software Variability with featureIDE. Springer

Michelon GK, Assunção WKG, Obermann D, Linsbauer L, Grünbacher P, Egyed A (2021) The life cycle of features in highly-configurable software systems evolving in space and time. In: ACM SIGPLAN International conference on generative programming: Concepts and experiences, GPCE 2021. Association for Computing Machinery, New York, pp 2–15. https://doi.org/10.1145/3486609.3487195

Michelon GK, Linsbauer L, Assunção WK, Fischer S, Egyed A (2021) A hybrid feature location technique for re-engineering single systems into software product lines. In: 15Th international working conference on variability modelling of software-intensive systems, vamos'21

Michelon GK, Sotto-mayor B, Martinez J, Arrieta A, Abreu R, Assunċào WKG (2021) Spectrum-based feature localization: a case study using argoUML. In: SPLC (A). ACM, pp 126–130

Northrop LM, Clements PC (2012) A framework for software product line practice version 5.0. Software Engineering Institute, Carnegie Mellon University. http://www.sei.cmu.edu/plp/framework.html

Olszak A, Jørgensen BN (2012) Remodularizing java programs for improved locality of feature implementations in source code. Sci Comput Programm 77(3):131–151. Feature-Oriented Software Development (FOSD 2009)

Otsuka J, Kawarabata K, Iwasaki T, Uchiba M, Nakanishi T, Hisazumi K (2011) Small inexpensive core asset construction for large gainful product line development: Developing a communication system firmware product line. In: 15Th int. Software product line conference, SPLC '11, vol 2. ACM, New York, pp 20:1–20:5

Photon Storm Ltd (2021) Phaser: desktop and mobile HTML5 game framework. http://phaser.io/

Prehofer C (2001) Feature-oriented programming: A new way of object composition. Concurr Comput Pract Exper 13(6):465–501

Rubin J, Chechik M, Bettin J (2013) A survey of feature location techniques. In: Reinhartz-berger I, Sturm A, Clark T, Cohen S (eds) Domain Engineering. Springer, Berlin, pp 29–58

Santos AR, do Carmo Machado I, de Almeida ES (2016) Riple-hc: javascript systems meets spl composition. In: 20Th international systems and software product line conference. ACM, pp 154–163. https://doi.org/10.1145/2934466.2934486

Schaefer I, Bettini L, Bono V, Damiani F, Tanzarella N (2010) Delta-oriented programming of software product lines. In: Bosch J, Lee J (eds) Proceedings, Lecture Notes in Computer Science, vol 6287. Springer, Jeju Island, pp 77–91

Schultheiß A, Bittner PM, Grunske L, Thu̇m T, Kehrer T (2021) Scalable n-way model matching using multidimensional search trees. In: 24Th international conference on model driven engineering languages and systems, MODELS 2021. IEEE, Fukuoka, pp 1–12

van Solingen R, Basili V, Caldiera G, Rombach HD (2002) Goal question metric (GQM) approach. Wiley

Stru̇ber D, Mukelabai M, Kru̇ger J, Fischer S, Linsbauer L, Martinez J, Berger T (2019) Facing the truth: benchmarking the techniques for the evolution of variant-rich systems. In: Proceedings of the 23rd Int. Systems and Software Product Line Conference, SPLC 2019, vol A. ACM, Paris, pp 26:1–26:12

Wohlin C, Runeson P, Höst M, Ohlsson M, Regnell B, Wesslén A (2012) Experimentation in Software Engineering, 1st edn. Springer Science & Business Media

Wolfart D, Assunçao WKG, Martinez J (2019) Open source software on the research of extractive adoption of software product lines. In: Proceeedings of latin. science latinoware

Wolfart D, Assunċào WKG, Martinez J (2021) Variability debt: characterization, causes and consequences. In: SBQS 2021: 20Th brazilian software quality symposium

Yang Y, Peng X, Zhao W (2009) Domain feature model recovery from multiple applications using data access semantics and formal concept analysis. In: 16Th working conference on reverse engineering, pp 215–224

Zhang G, Shen L, Peng X, Xing Z, Zhao W (2011) Incremental and iterative reengineering towards software product line: an industrial case study. In: 27Th IEEE int. Conf. on software maintenance (ICSM), pp 418–427

Ziadi T, Frias L, da Silva MAA, Ziane M (2012) Feature identification from the source code of product variants. In: 16Th european conference on software maintenance and reengineering, pp 417–422

Ziadi T, Hillah LM (2018) Software product line extraction from bytecode based applications. In: 23Rd international conference on engineering of complex computer systems (ICECCS), pp 221–225

Rodrigo André Ferreira Moreira is a Master's student at the Federal University of Minas Gerais, Brazil. His current research interests include software product lines and design pattern detection from source code analysis.

Wesley K. G. Assunção is currently a University Assistant at Johannes Kepler University Linz (JKU) - Austria, Post-Doctoral researcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio) - Brazil, and visiting professor at the Graduate Program in Computer Science at Western Paraná State University (Unioeste) - Brazil. Wesley received his M.Sc. in Informatics (2012) and Ph.D. in Computer Science (2017) both from Federal University of Paraná (UFPR) - Brazil. His areas of interest are Software Modernization, Variability Management, Collaborative Engineering of Complex Systems, Software Testing, and Search Based Software Engineering. He published research papers, in collaboration with international researchers, in conferences like ICSME, SANER, MSR, EASE, SPLC, SSBSE, GECCO, to cite some, as well as in journals such as EMSE, IST, and JSS. Wesley has also been serving as reviewers for many conferences and journal, and as organizer of conference, symposiums, workshops, competitions, and meetings. Website: https://wesleyklewerton.github.io/

Jabier Martinez is a research engineer in the Digital Trust Technologies (TRUSTECH) area of Tecnalia since 2018. His background is on providing methods and tools for systems modelling and variability management. After several years of industrial experience, he received his PhD from the Luxembourg and Sorbonne Universities with a thesis about product line adoption and analysis. He co-organizes the Reverse Variability Engineering (REVE) series of workshops.

Eduardo Figueiredo is an associate professor and head of the Software Engineering Laboratory (LabSoft) at the Federal University of Minas Gerais (UFMG) since 2010. He received his PhD degree in Software Engineering from Lancaster University (UK) in 2009 and was a visiting researcher at Carnegie Mellon University (CMU) in 2017. His research interests include configurable software systems, empirical software engineering, and source code analysis. Website: http://www.dcc.ufmg.br/∼figueiredo.

Affiliations

Rodrigo André Ferreira Moreira[1] · **Wesley K. G. Assunção**[2,3] · **Jabier Martinez**[4] · **Eduardo Figueiredo**[1]

Rodrigo André Ferreira Moreira
radro.rs@gmail.com

Jabier Martinez
jabier.martinez@tecnalia.com

Eduardo Figueiredo
figueiredo@dcc.ufmg.br

1 Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil

2 Institute for Software Systems Engineering, Johannes Kepler University Linz (JKU), Linz, Austria

3 Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil

4 Tecnalia, Basque Research and Technology Alliance (BRTA), Derio, Spain