The Journal of Systems & Software 205 (2023) 111809

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

SPLLS2: Software product lines extraction driven by language server protocol☆,☆☆

Francesco Bertolotti, Walter Cazzola *, Luca Favalli

Università degli Studi di Milano, Computer Science Department, Milan, Italy

ARTICLE INFO

Article history:
Received 1 March 2023
Received in revised form 14 June 2023
Accepted 21 July 2023
Available online 5 August 2023

Keywords:
Software product lines
Language server protocol
Feature-oriented programming
Aspect-oriented programming
Design patterns

ABSTRACT

Software product lines (SPL) describe highly-variable software systems as a family of similar products that differ in terms of the features they provide. The promise of SPL engineering is to enable massive software reuse by allowing software features to be reused across a variety of different products made for several customers. However, there are some disadvantages in the extraction of SPLs from standard applications. Most notably, approaches to the development of SPLs are not supported by the base language and use a syntax and composition techniques that require a deep understanding of the tools being used. Therefore, the same features cannot be used in a different application and developers must face a steep learning curve when developing SPLs for the first time or when switching from one approach to a different one. Ultimately, this problem is due to a lack of standards in the area of SPL engineering and in the way SPLs are extracted from variability-unaware applications. In this work, we present a framework based on LSP and dubbed SPLLS2 that aims at standardizing such a process by decoupling the refactoring operations made by the user from the effect they have on the source code. This way, the server for a specific SPL development approach can be used across several development environments that provide clients with customized refactoring options. Conversely, the developers can use the same client to refactor SPLs made according to different approaches without needing to learn the syntax of each approach. To showcase the applicability of the approach, we present an evaluation performed by refactoring four SPLs according to two different approaches: the results show that a minimal implementation of the SPLLS2 client and server applications can be used to reduce the effort of extracting an SPL up to the 93% and that it can greatly reduce or even completely hide the implementation details from the developer, depending on the chosen approach.

1. Introduction

Research context. Modern software systems must fulfill the needs of a ever-growing customer base. Due to the diversity of human needs, software should be customizable and reconfigurable. To answer these needs, during the last decades researchers and practitioners gained interest in software product lines (SPL) as an engineering technique for the development of highly-variable systems. SPLs can be implemented in many ways. Most approaches embrace either the *compositional* or the *annotative design philosophies* (Kästner and Apel, 2008). The *annotative* approaches use macros such as the C `#ifdef` to highlight system portions intended to implement a software feature. The *compositional* approaches use variability-aware preprocessors called *composers* to generate a program variant from a set of *features* and a *configuration*. Base languages—*i.e.*, languages with their compilers, without any external tools—rarely provide native support for either philosophy. For instance, C is one of a few alternatives providing native support for annotative approaches through its derivatives. However, compositional approaches are not supported out-of-the-box. Java is very popular among SPL researchers (Batory et al., 2004; Bergel et al., 2005; Koscielny et al., 2014; Figueiredo et al., 2008) and can feed C macros to the C preprocessor to implement variability, however this solution is not very popular whereas developers usually prefer an external composer to implement SPLs. State-of-the-art SPL development environments such as FeatureIDE (Thüm et al., 2014; Meinicke et al., 2017) can cope with all the aspects of SPL development, including construction, software artifacts management, configuration and product derivation, and yet developers must struggle to keep such tools up to the evolution of the

☆ Editor: Laurence Duchien.

☆☆ This work was partly supported by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

* Corresponding author.
E-mail addresses: bertolotti@di.unimi.it (F. Bertolotti), cazzola@di.unimi.it (W. Cazzola), favalli@di.unimi.it (L. Favalli).

https://doi.org/10.1016/j.jss.2023.111809

base language—for instance, Java has a 6-month release cycle since March 2021.[1] This led the researchers to invent new techniques that do not rely on external tools and instead exist within the boundaries of the base language used by the application. Examples of such techniques are the *variability modules* architectural pattern (Setyautami and Hähnle, 2021) and the *devise pattern* (Bertolotti et al., 2022).

Problem statement. Considering all these approaches, the vastness and diversity of the research on SPLs is apparent. Even when considering only compositional approaches, developers must choose one of many composers to translate features into a valid Java application: FeatureHouse (Thüm et al., 2014), AHEAD (Batory et al., 2004), Antenna[2] and AspectJ (Mezini and Ostermann, 2004) among many others. SPL developers must struggle to learn the syntax and composition mechanisms of each tool and the acquired skills are hardly transferable to a different SPL development approach. In fact, there is no general consensus on how the composition mechanism should be performed, thus the source code of the core application and its features are structured differently depending on the composer tool. The diversified range of tools and methods is among the primary hindrances to the proliferation of SPLs. Artifacts developed with one suite are not compatible nor reusable with other ones (Chimalakonda and Hyung, 2016) except for case-by-case *ad hoc* solutions, such as a (work in progress) converter between AHEAD, and FeatureHouse projects.[3] Such a limitation causes a dissonance between the promise of SPLs of enabling massive software reuse for an ever-growing customer base and the actual inconsistency, incompatibility and reusability issues of a diversified range of tools for their development. In other words, SPL engineering (SPLE) fails to respond to the innate volatility of requirements caused by factors such as customer needs, market change, global competition, and government policies (Jayatilleke and Lai, 2018) due to its inability to change approach according to the new requirements. This predicament ultimately motivates the need for the research community to improve the standards in the area of SPLs (Chimalakonda and Hyung, 2016).

Contribution. This problem is similar to a problem recently tackled by the *language server protocol*[4] (LSP): the innate diversity of programming languages makes it hard to allow programming language support to be implemented and distributed independently of any development environment, which the developers may be accustomed to. The LSP quickly became the *de facto* standard for the development of language support because it makes easy to port existing language services to several different development environments and their users. Without the LSP, new programming languages may struggle to be adopted because they offer limited tooling support and users may be skeptical about abandoning the development environment of their choice. Similarly, we want to facilitate the adoption of new SPL development approaches by bringing them to the development environments the users are already accustomed to. This work proposes a framework—dubbed SPLLS[5]—inspired by the LSP and is used to refactor variable software systems into SPLs in a tool-agnostic way. Following this approach, the SPL refactoring process is standardized, so that the same client can be used with several servers and vice versa; by following a unique protocol, users can leverage the acquired expertise across several SPL development tools and approaches.

Evaluation case study. To support this contribution, we present a proof-of-concept plugin that helps the developers performing a refactoring according to the SPLLS framework. Then, we evaluate the associated development effort. This research is validated by answering the following research questions.

- **RQ$_1$** How effective is SPLLS at reducing the effort of refactoring standard applications into SPLs?
- **RQ$_2$** How effective is SPLLS at abstracting the SPL development tools/approaches to developers?

Structure. The remainder of this paper is structured as follows. In Section 2 we contextualize our work with regards to any background information it is based on. In Section 3 we discuss the SPLLS framework, its notation, capabilities and applicability. In Section 4 we introduce the case studies on which we performed a refactoring based on SPLLS and discuss the experiment performed to evaluate SPLLS. In Section 5 we provide an overview of related works in this research area. Finally, in Section 6, we draw our conclusions and outline some future directions of this research.

2. Background

This section introduces the research context and any terminology required to understand the contribution of this work. First, we discuss the topic of SPLE. Then, we overview the two SPL development approaches that will be part of our evaluation case study. Finally, we briefly discuss the LSP as the technology that inspired this contribution.

2.1. Software product line engineering

Variability-rich production is a kind of industrial production that has been dealt with in classical engineering through the creation of product lines. SPLE follows the same idea to handle the complexity of variability-rich software systems and to allow for massive software reuse. This goal is achieved by capturing the similarities among software systems pertaining the same application domain and modeling them as part of a software family. Two members of the same family are distinguished by the *features* they are comprised of—*i.e*, the collection of characteristics and end-user-visible behaviors of the products. Features and their relations can be expressed in a variety of ways; in this work we will focus on feature diagrams based on the *feature model* (FM) formalism. Since their introduction as part of the FODA method (Kang et al., 1990), FMs have become the *de facto* standard for variability modeling (Czarnecki et al., 2012).

Given a FM, a product can be derived from a valid *product configuration* (or just configuration). A configuration is considered *valid* when it conforms to the constraints expressed by the FM. Constraints can be directly expressed by the feature diagram—such as, the parent–child relationship, mandatory features, optional features, or groups and alternative groups. Alternatively, dependencies are declared through by defining *cross-tree constraints—i.e.*, logical formulas based on the most common Boolean operators whose terms are the activation statuses of each feature of the FM. Feature dependencies may cause anomalies that degrade the quality of the FM, such as atomic-sets and dead features. Research towards the detection and refactoring of such anomalies is active and includes structural (Benavides et al., 2010) and behavioral (ter Beek et al., 2021) detection strategies.

SPL development can be faced in three ways, regardless of the tools and approaches chosen for their creation (Krueger, 2001; Kühn and Cazzola, 2016):

[1] https://www.java.com/releases/fullmatrix/

[2] http://antenna.sourceforge.net

[3] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.core.conversion.ahead-featurehouse

[4] https://microsoft.github.io/language-server-protocol/

[5] SPLLS is read SPLPS.

- *proactive* or *top-down*—the SPL is created from scratch;
- *extractive* or *bottom-up*—an existing code base is converted into an SPL;
- *reactive*—the development starts from an initial set of core features and then the SPL is developed incrementally by adding more features to the initial set.

The two SPL development approaches discussed in this paper (AspectJ and the devise pattern) can be used to implement SPLs in either a proactive, extractive or reactive manner. However, our contribution focuses on the extractive approach, for which we propose a standard refactoring process.

2.2. Aspect-oriented programming with AspectJ

SPLE is concerned with organizing code into features. Aspect-oriented programming (AOP) is a technique that builds on top of existing technologies to complement the inability of object-oriented systems to properly modularize the crosscutting concerns of complex systems. AOP is a viable SPL development technique, by optionally weaving aspects into an application based on the activation status of features in the configuration. AspectJ (Kiczales et al., 2001) is a popular implementation of AOP and arguably its most mature implementation to date; it strives to achieve four design goals (Kiczales et al., 2001):

- upward compatibility—all valid Java programs are also valid AspectJ programs;
- platform compatibility—all legal AspectJ programs must run on standard Java virtual machines;
- tool compatibility—it must be possible to extend existing tools and libraries running on Java;
- programmer compatibility—programming with AspectJ must feel like a natural extension of programming with Java.

The AspectJ development process is based on the *join point* and *advice* concepts. Join points are well-defined points in the program's data flow, such as method calls and field accesses; each can be considered as a node in a call graph. AspectJ provides the *pointcut* construct to capture several join points based on wildcards and combination operators (`&&`, `||`, `!`). Advices are a method-like mechanism that declares the code that should be executed before, after or instead of each of the join points in a pointcut. This process is called *weaving*: AspectJ can be used to implement variability in SPLs by weaving different advices onto the same join point, depending on the product configuration. Although pointcuts are not always granular enough (Cazzola and Vacchi, 2014, 2013) to capture the variability points in the code of an SPL, it is always possible to extend the base application with hook methods—*i.e.*, empty methods placed in the code for later extensions—for the aspect to be woven (Kästner et al., 2007).

2.3. Devise pattern

The devise pattern (Bertolotti et al., 2022) is a design pattern (Gamma et al., 1995) for the development of SPLs based on Prehofer's definition of features (Prehofer, 1997, 2001). According to Prehofer, features are similar to mixins (Bracha and Cook, 1990) and provide services while avoiding a rigid class structure, as objects with individual services can be composed from a set of features instead of using inheritance and method overwrites. As the core functionality is separated from interaction handling, it provides more structure and clarifies dependencies between features. The devise pattern follows the same philosophy but adds an additional layer of abstraction to standardize the way features are declared and composed, without relying on external tools and preprocessors. Instead, the devise pattern exists within the boundaries of the base language. To achieve this result, the devise pattern splits the code into two hierarchies: the *class* hierarchy—*i.e.*, all the inheritors of the `Object` super-class—and the *features* hierarchy—*i.e.*, all the inheritors of the `Feature` super-class. The class hierarchy is the traditional object-oriented hierarchy: objects are instances of a class and contain both code and data in a format that is conform to their class. The feature hierarchy represents instead any domain modeling code: each feature is a (usually empty) class that declares an element of the FM, whereas instances of these classes are called *feature actions* and are used to declare the services composed with the base application. The services implemented within a feature action are declared using a syntax that is similar to `#ifdef` macros; their execution is initially delayed by leveraging a functional interface or anonymous subclasses. Then, each feature action can either be performed or preempted, based on the activation status of the feature according to a configuration and on the validity of the configuration. Please refer to Bertolotti et al. (2022) for a complete overview of the devise pattern and to Section 3.6 for concrete examples on how to implement SPLs using the devise pattern.

2.4. Language server protocol

The relationship between language providers and tooling providers traditionally works in a one-to-many fashion, in which language features such as diagnostics, auto-completions, and code navigation for a given programming language are implemented once for each environment they are destined to. This is due to the fact that each language has its own API and the language features must be implemented in such a way that matches the API provided by the tools they aim to use. The same process must be repeated for each tool and for each language, leading to considerable effort. In order to avoid this problem, Microsoft introduced the *language server protocol* (LSP) to standardize the communication between tools and languages. In the LSP, the language runs a standalone server process and the development tool communicates with the server over JSON-RPC. The LSP provides messages to perform several actions over the documents being edited: change updates, *go to* requests, hover actions, semantic tokens and several others.[6] For instance, refactoring is supported by code action requests,[7] that can be adapted to the refactoring operations presented in this work.

3. SPLIT overview

This section describes a framework for refactoring object-oriented systems into variability-aware SPLs. This framework, dubbed SPLIT, is not based on any specific approach or tool for SPL development, neither annotative nor compositional. Instead, it sets the properties of the resulting SPL and a common framework for its development, regardless of the approach. Moreover, SPLIT must support a minimal set of refactoring operations. Minimality is intended in the sense that the supported operations are expressive enough to implement variability awareness while excluding features supported only by a few specific approaches. Such a limitation may neglect the peculiarities of most complex compositional approaches, but it is necessary to overcome their relative inconsistency and reusability issues. For instance, FeatureHouse features allow to wrap the code of other features using the `original()` primitive. Instead, SPLIT supports the addition of

[6] https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#languageFeatures

[7] https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_codeAction

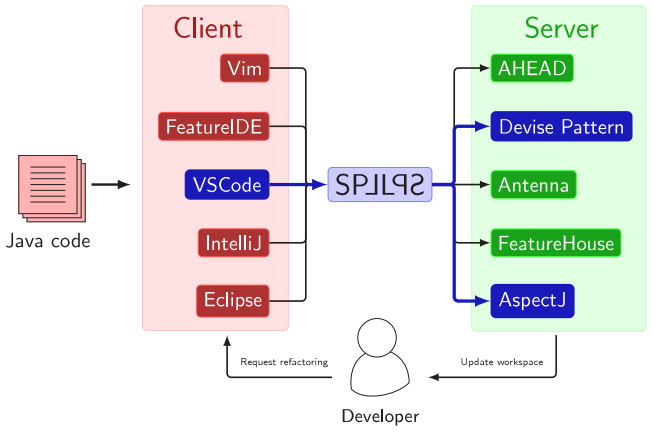

Figure 1. Overview of the SPLLSP protocol.

code before or after other features' code; these operations are enough to simulate wrappers without loss of generality. First, we overview the general idea behind the approach, then we list the supported capabilities and the refactoring process. Finally, we exemplify the implementation of the SPLLSP framework in AspectJ and using the devise pattern.

3.1. How it works

SPLLSP aims to standardize the protocol that must be followed to refactor code into an SPL, so that the same implementations can be reused to extract SPLs from variability-unaware applications. Fig. 1 shows the similarity between SPLLSP and the LSP: in both cases the client application is the plugin for a development environment and the server is an application providing services that are specific to a tool. The main difference is that in the LSP the tool is a programming language implementation whereas in SPLLSP the tool is an SPL development framework. In the example shown in Fig. 1, the client application is the VS Code[8] plugin highlighted in blue. The user can interact with the development environment to explicitly request refactoring operations, that the client translates into messages sent to the server; each refactoring request is used by the server to extract an SPL from a variability-unaware application, according to the capabilities offered by SPLLSP. As we will discuss in Sections 3.2 and 3.4 in detail, the available refactoring operations are feature declarations, insertion point declarations, modify operations and add operations.

Each time the user interacts with the development environment to request a refactoring operation, the client (VS Code in this example) translates it into a JSON-RPC request. While the choice of the client and server applications, as well as their implementation and offered capabilities, is left to the developers, the protocol relies on JSON-RPC to represent the messages to ensure all servers can communicate with all clients and vice versa. The same choice is made by the LSP to implement its capabilities. Each refactoring request is sent to one or more server applications capable of handling SPLLSP JSON-RPC requests. Fig. 1 depicts the protocol in general. In the current prototype only two servers are provided: the AspectJ and devise pattern SPLLSP servers.

The server translates the refactoring request into an action performed over the workspace—*i.e.*, on either the project files or their content. The result is an updated version of the original code according to the chosen SPL development tool or approach. For instance, the AspectJ SPLLSP server may prepare a refactoring in which the refactored code is moved to an aspect and the original code is substituted by a hook method. Finally, the updated version of the workspace is fed back to the client that can either accept or refuse the changes.

The basic idea behind the SPLLSP protocol is straightforward: introducing an indirection layer between the two versions of the application in form of the JSON-RPC requests and responses allows for the client application to be reused across several project migration scenarios. The user does not need to learn the syntax of new SPL development because the interactions with the development environment performed to refactor the original code are the same regardless of the chosen SPL development approach. Moreover, the server application can also be re-used to refactor object-oriented applications into SPLs across any development environment for which a SPLLSP client is available, while providing an abstraction to hide the details of the SPL approach to the developers.

While the LSP and SPLLSP share similar architecture, technological space, and benefits, the main contribution of this work is the difference in their objectives. The LSP is not concerned with how the programming languages are used by the developers and focuses on the tooling perspective, aiming for the same development environments to be used with several language services and vice versa. Instead, SPLLSP shares similar advantages, but focuses on the methodology perspective, by standardizing the way developers face the problem of extracting SPLs from variability-unaware applications.

3.2. Notation & capabilities

To move towards the adoption of a standard in the definition of tools and methods for SPL implementation, the SPL community should define the minimum set of capabilities that are considered valuable for the creation of SPLs, as well as the properties of software by means of SPLE. Our proposal in this regard is based on the *additive universe conjecture* (Batory, 2021):

> *«Every FM that uses subtractive features can be transformed to a new FM that uses only additive features; the two FMs share the same set of products.»*

The main appeal behind the additive universe conjecture is the intuition that *monotonic reasoning* is easier than non-monotonic reasoning (Batory, 2021)—*i.e.*, reasoning is harder if conclusions can be invalidated by adding more facts. There are a few motivations that support sticking only to increasing monotonic SPLs and removing subtractive features altogether. Due to their simpler structure, monotonic SPLs are easier to understand and to analyze and it is usually simpler to implement additive features compared to subtractive features (Schulze et al., 2013). For instance, as we will show in Section 3.5, it is relatively simple to add fields and methods to a Java class using AspectJ. Instead, it is impossible to weave an aspect that removes access to a field unless limiting field access to the usage of getters and setters and then implementing a subtractive feature as a decorator that raises an exception when getter and setter methods are used (Chimalakonda and Hyung, 2016).

Sticking to increasing monotonic SPLs, let us introduce any concept that is relevant to the SPLLSP framework. Gold boxes represent the capabilities provided by SPLLSP, whereas blue boxes contain any notation that is required to express these capabilities. This notation is partially based on existing literature on the topic, mainly that regarding delta-oriented programming (DOP)

[8] https://code.visualstudio.com/

(Schaefer et al., 2010) and aspect-oriented programming (AOP) (Kiczales et al., 1997). First, increasing monotonic are made of a collection *base* and *extension* feature; their definition is inspired by the DOP paradigm and its *core modules* and *delta modules* respectively (Schaefer et al., 2010).

Notation 1 (*base feature*)**.** *The base feature is all the code of an SPL that is shared among all products of the SPL and that can never be removed.*

Notation 2 (*extension features*)**.** *An extension feature is code that implements a functionality of one or more concrete products of the SPLs, by extending either the base feature or another extension feature in an additive manner.*

Henceforth, when we refer to features without specifying whether they are the base feature or extension features, we imply that either are acceptable in the given context for brevity. SPLIS2 supports base features and extended features through the following capability.

Notation 3 (*feature declaration*)**.** *Feature declaration is a capability that can be used to declare a feature and add it to the FM. Performing a feature declaration requires the following information:*

- *name*
- *parent feature*
- *feature modifiers (or group, alternative group, mandatory, abstract)*

Extension features can extend the base feature with new code by performing two kinds of operations: *add* and *modify* operations. In this context, both the code of the base feature and the code of the extension feature can be considered as generic text strings, as long as the final product of their composition—*i.e.*, the text obtained by combining the two original strings, can be compiled and executed.

Notation 4 (*add operations*)**.** *Add operations denote the effect of an extension feature of causing the addition of brand new classes, interfaces, fields and methods to the code of a feature.*

Notation 5 (*modify operations*)**.** *Modify operations denote the effect of an extension feature of causing the modification of already existing code (such as, the body of a method) of another feature.*

The definitions of add operations and modify operations are based on the *modifies* and *adds* clauses in a delta module (Schaefer et al., 2010). However, our definition differs from the original, because in this context the distinction is tied to the level of granularity at which an operation performs: a modifies clause from DOP may be considered as an add operation in SPLIS2 if the modification causes the creation of brand new fields or methods. In other words, in SPLIS2 add operations equate to a delta module adding brand new classes or interfaces or modifying existing ones by adding new fields and methods, whereas modify operations equate to a delta module modifying the body of an existing a method. In the SPLIS2 protocol, all modify operations are exclusively additive. A modify operation can extend the code by adding new code, but it cannot replace nor remove existing code. Otherwise, the SPL would not be increasing monotonic and would not fit this standard. Finally, let us introduce the following notations.

Notation 6 (*feature action*)**.** *The new code added to a feature by an extension feature through the effect of an add operation or a modify operation is called feature action.*

Notation 7 (*insertion point*)**.** *An insertion point is a point in the code of a feature that can be extended with a feature action by effect of a modify operation.*

Feature actions and insertion points are a generalization of the join points and advices used in AOP (Kiczales et al., 1997). In particular, an insertion point generalizes a join point because it can be placed at any point of any method instead of being tied to data flow of the program—*e.g.*, its method invocations. Each extension feature can perform several add operations and modify operations and therefore its code can be made of several feature actions, possibly scattered across the code of other features.

SPLIS2 supports insertion points, modify operations and add operations through the following capabilities.

Notation 8 (*insertion point declaration*)**.** *Insertion point declaration is a capability that can be used to declare a variability point within the code of a feature. Performing a feature declaration requires the following information:*

- *file*
- *row and column of the insertion point*

Notation 9 (*feature actions declaration*)**.** *Feature action declaration is the capability of performing add operations and modify operations over features. Performing a feature action declaration requires the following information:*

- *feature*
- *insertion point (optional for add operations)*
- *source code of the feature action*

Notice that for add operations, specifying the insertion point is optional because in most object-oriented programming languages *the position* in which new classes, interfaces, fields and methods are added is not semantically relevant.

3.3. Messages

As discussed in the previous section, SPLIS2 standardizes the development of increasing monotonic SPLs through the following capabilities:

- feature declaration;
- insertion point declaration;
- feature actions declaration.

SPLLSP uses a JSON-RPC message to express each of these three capabilities. The messages contain a header part and a content part. The header part consists of the content length and the content type, whereas the content part uses JSON-RPC to describe requests and responses, as shown in the message below, used to send a feature declaration request to the server.

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "workspace/declareFeature",
    "params": {
        "name": "Theme",
        "parent": "Base",
        "modifiers": ["mandatory", "alternative"]
    }
}
```

The structure of each message varies depending on the capability it supports: each capability has a different method and different parameters. Instead, response messages sent from the server to the client are instructions needed to translate a requested operation into a workspace edit, in compliance to the SPL development approach implemented by the server. We do not discuss this kind of messages in detail because SPLLSP responses leverage the workspace editing capabilities provided by the standard LSP protocol.

Feature declaration. The feature declaration request is sent from the client to the server when the user interacts with the development environment to create a new feature of the FM. Feature declaration messages are identified by:

1. the `"workspace/declareFeature"` method;
2. parameters defined according to the following class.

```
class DeclareFeatureParams {
    String name;
    String parent;
    String[] modifiers;
}
```

Insertion point declaration. The insertion point declaration request is sent from the client to the server when the user interacts with the development environment to set an insertion point within the code of the application. Insertion point declaration messages are identified by:

1. the `"textDocument/declareInsertionPoint"` method;
2. parameters defined according to the following class.

```
class DeclareInsertionPointParams {
    URI file;
    Position position;
}
```

According to the LSP, the Position class is comprised of two integers representing a line and a character within the text document.

Feature action declaration. The feature action declaration request is sent from the client to the server when the user flags a portion of code as part of a feature action. Feature action declaration messages are identified by:

1. the `"workspace/declareFeatureAction"` method;
2. parameters defined according to the following class.

```
class DeclareFeatureActionParams {
    String featureName;
    DeclareInsertionPointParams insertionPoint;
    Location location;
}
```

The `Location` class is comprised of an `URI` object to identify the text document within the workspace and a `Range` object. The `Range` object contains two `Position` objects: one for the initial character and one for the final character in the range. Overall, the `Location` object uniquely identifies the portion of code that is part of the feature action within the code of the workspace.

3.4. Extraction process

Let us explain the *refactoring process* with the help of the example shown in Fig. 2. The actors involved in the refactoring process are: the developer, the workspace (or application), the SPLLSP client (or development environment) and the SPLLSP server. The example shown in Fig. 2 summarizes the process of a feature declaration refactoring operation, but the process is essentially the same for all refactoring operations, albeit with different messages sent between the client and the server and with a different result.

First, the developer interacts with the development environment by providing any information that the server needs to perform the refactoring. The pieces of information that the developer must provide depend on the implementation of the SPLLSP client, but must be at least those reported in the gold boxes in Section 3.2, according to the refactoring capability to be performed. For instance, in Fig. 2 the developer provides a representation of the FM in JSON format which is the format accepted by the VS Code client. However, a different client may be based on a different representation—such as the model.xml created by FeatureIDE. When the developer confirms the refactoring request, the SPLLSP client translates the provided information into messages according to the requested capability—*i.e.*, either feature, insertion point or feature action declarations. In Fig. 2, the FM written in JSON triggers several feature declaration messages, one for each feature (*e.g.*, `declare feature: Base`, `declare feature: Theme`, …). According to the message type definitions presented in Section 3.3, each feature declaration message contains the following parameters: the name of the feature, its parent and any modifiers. In a different refactoring scenario, different messages may be triggered and the message structure would be changed accordingly. The SPLLSP server receives the requests and translates any message into an internal representation that depends on the server implementation. For instance, the server may keep an internal representation of the entire FM (as in the third column of Fig. 2) to keep track of any constraints among features. This step can be entirely skipped if the SPL development approach implemented by the server does not need an internal representation to perform its refactoring. Finally, the internal representation is used by the server to plan a sequence of workspace edits. Each workspace edit is fed back to the client using standard LSP messages. Upon receiving the workspace edit messages, the client application applies any suggested changes to the workspace. For instance, the AspectJ SPLLSP server used in this example responds with workspace edit messages that trigger the creation of a hierarchy of the directories mirroring the FM structure. Each directory will contain all aspects relevant to the feature actions performed by that feature. At this stage, the directories are empty because no feature action has been yet declared. Each directory will be populated at a later stage, when the developer will interact with the client to declare feature actions for that feature. A different

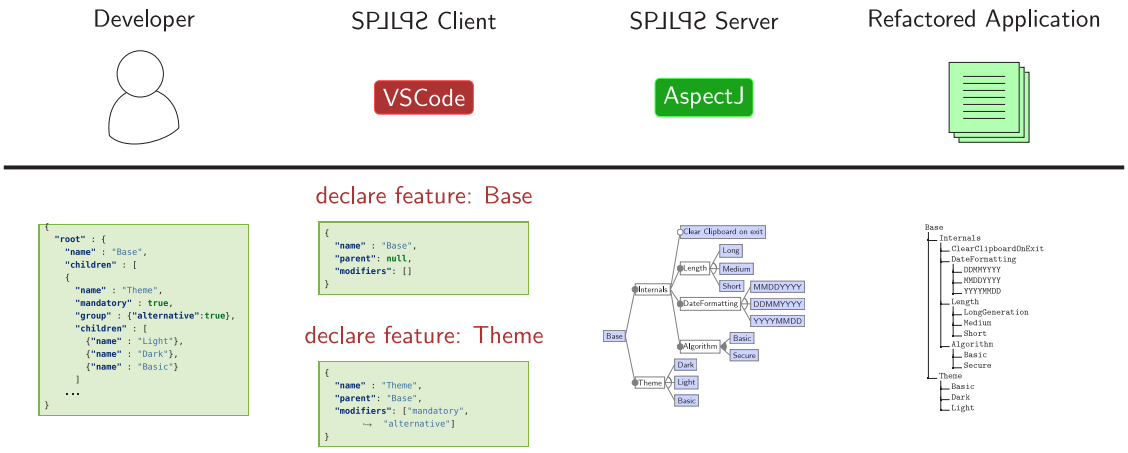

Figure 2. Feature declaration refactoring process in SPLIF2.

```
1  class Foo {
2      public void bar() {
3          /* ... */
4          Foo.__insertion_point__1();
5          /* ... */
6      }
7      public static Optional<?> __insertion_point__1() {
8          return Optional.empty();
9      }
10 }
```

(a) Insertion point declaration by adding hook methods.

```
1  class Foo implements InsertionPoint {
2      public void bar() {
3          /* ... */
4          this.__insertion_point__1();
5          /* ... */
6      }}
7  public interface InsertionPoint {
8      default Optional<?> __insertion_point__1() {
9          return Optional.empty();
10     }}
```

(b) Insertion point declaration by adding external interfaces.

Listing 1 Insertion point declaration in AspectJ. The code added by the refactoring operation is highlighted in green.

server may suggest a different refactoring of the workspace upon receiving the same messages, depending on the SPL approach it supports: in the devise pattern, each feature corresponds to exactly one Java class, therefore the devise pattern SPLIF2 server suggests the creation of a new class when it receives a feature declaration message.

This process is repeated each time the developers performs a refactoring. For example, if the developer moves a portion of code from the base feature into an extension feature, the request is translated into two messages: the declaration of an insertion point and the declaration of a feature action. In the case of the AspectJ SPLIF2 server, the insertion point declaration substitutes the original code with a call to a hook method and the feature action declaration creates a new aspect that weaves the original code into the base feature in correspondence to the hook method call.

Once the developer can no longer identify any variability point within the original application, the process is considered done and the application has been successfully refactored into an SPL.

3.5. Refactoring Operations in AspectJ

The first SPLIF2 server exemplified in this paper is used to create SPLs based on AspectJ. In this section, we overview how AspectJ capabilities are mapped to SPLIF2 capabilities to support the extraction of SPLs from Java applications. With regards to the feature action declaration capability, the discussion will be split in two paragraphs for better readability: the first concerns feature actions created through a modify operation whereas the second concerns feature actions that are created through an add operation. At server side, the effect a performing add operations and modify operations over the original Java application involves moving a piece of code from a Java class into an AspectJ advice.

Features declaration. AspectJ and AOP in general do not include the feature concept nor it is necessary a good solution to map each aspect into a single feature. Instead, a feature can be made of several aspects (Kästner et al., 2007), each changing the code of the base feature according to the insertion of one or a few feature actions through pointcuts and advices. Therefore, as shown in Fig. 2, the AspectJ server implemented in this work simply uses the concept of feature as a logical information to organize code: the AspectJ SPLIF2 server generates a directory structure that mirrors the FM provided by the user; each directory will contain all the aspects that pertain the corresponding feature. This directory structure is useful upon the generation of a concrete product according to the chosen configuration: upon compiling the system with the AspectJ compiler, the activation of a feature within the product is performed by weaving all the aspects contained in the directory corresponding to that feature.

Insertion points declaration. Although pointcuts allow for fine-grained selection of insertion points within the base feature by using class and method wildcards within their definition, it is not possible to create a pointcut that can cut at any point of the execution in the general sense. However, there is a refactoring that can be performed on any Java class to support the declaration of insertion points in SPLIF2: the creation of hook methods. Hook methods are usually empty methods that provide additional nodes within the data flow graph that can be captured by pointcuts. An example of the declaration of an insertion point using hook methods is shown in Listing 1(a): in this case, the hook method returns an `Optional` to be as generic as possible, but it could also be a `void` method. Creating hook methods arguably makes the code of the base application more cluttered. However, a more sophisticated server could use interfaces to avoid cluttering the body of the base application with several additional methods. Such an example is shown in Listing 1(b), in which the

hook method is isolated within the `InsertionPoint` interface, whereas the `Foo` class only has to implement that interface and was not cluttered by new methods. Since hook methods do not implement any behavior when used alone, it should be noted that the insertion point declaration capability is usually used jointly with the feature action declaration capability. In this case, a block of code is removed from the base feature and moved into an AspectJ advice. The original code in the base feature is replaced by a call to a hook method. More on this in the paragraph about the modify operation.

Modify operation feature actions. Modifying the body of methods using AspectJ can be performed by weaving advices around calls to hook methods that were previously declared as insertion points. This refactoring technique is exemplified in Listing 2 through the `ModifyOperations` aspect.[9] The feature action highlighted in green in Listing 2(b) is obtained from code that was originally part of the base feature—as shown by the red box in Listing 2(a). Performing the exemplified modify operation moves the highlighted code to the **`around`** advice shown at line 6 of Listing 2(b). In a product configuration in which the feature that contains the `ModifyOperations` feature action is active, the generated advice is woven to the `Foo` class; AspectJ can identify the correct insertion point through the pointcut highlighted in blue in Listing 2(b), that matches all calls to the `__insertion_point__1` method. Notice that the insertion point on line 5 is not part of this refactoring and was instead created through a distinct insertion point declaration. As previously mentioned, it may be beneficial to declare an insertion point and a feature action that adds a block of code jointly instead. This can be achieved if the client allows the user to select code and refactor it into a feature action at the same time. The client can then trigger two messages: the first for an insertion point declaration and the second for a feature action declaration hooked to the newly created insertion point. The server refactoring capabilities when declaring an add block feature action can be arbitrarily complex: for instance, the feature action may require additional information depending on the original code. In that case, the user can edit the code manually to provide the necessary information to the insertion point. We will discuss an example in which such an edit is needed in Listing 7 (page 11, marked with a ✻). Otherwise the server can perform a complex refactoring in a similar fashion, so that variables declared outside of the feature action are passed as an argument to the advice.

Add operation feature actions. Defining feature actions related to add operations is straightforward using inter-type declarations. According to the AspectJ programmer guide,[10] inter-type declarations are "declarations that cut across classes and their hierarchies". Inter-type declarations can be used to declare members or change the inheritance relationship between classes. Such an example is shown in Listing 3. The aspect in this example adds three elements to the `Foo` class: respectively, the code highlighted in red adds an interface, the code highlighted in blue adds a new field and the code highlighted in green adds a new method. The corresponding add operation feature action is performed if and only if the `AddOperations` advice is woven to the `Foo` class—*i.e.*, if a feature that contains the `AddOperations` feature action is active in the product configuration.

[9] The name chosen for the aspect is not relevant as long as the aspect is connected to the feature model. In this and all the following examples we chose names with the goal of highlighting the type of operation that each feature action is performing.

[10] https://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html#inter-type-declarations

```
1  class Foo {
2      public void exitFrame() {
3          if (Configuration.getInstance()
4            .is("clear.clipboard.on.exit.enabled", false)) {
5              Foo.__insertion_point__1();
6              EntryHelper.copyEntryField(this, null);
7          }
8          /* ... */
9      }
10 }
```

(a) Code highlighted by user before performing a modify operation.

```
1  public aspect ModifyOperations {
2      pointcut InsertionPoint() :
3       call(public static Optional<?>
4         Foo.__insertion_point__1(..));
5      Optional<Object> around() : InsertionPoint() {
6          EntryHelper.copyEntryField(this, null);
7          return Optional.empty();
8      }
9  }
```

(b) Feature action created by effect of a modify operation.

Listing 2 Code needed to perform a modify operation in AspectJ. The code that has been moved from the base feature to the `ModifyOperations` feature is highlighted in red in Listing 2(a). In Listing 2(b), the pointcut highlighted in blue captures the call to the insertion point. The new feature action obtained from the code of the base feature is highlighted in green.

```
1  public aspect AddOperations {
2      declare parents: Foo implements NewInterface;
4      private NewField Foo.field = new NewField();
6      public NewField Foo.getField(){
7          return this.field;
8      }
9  }
```

Listing 3 Code needed to perform add operations in AspectJ. The code highlighted in red adds an interface to the class declaration. The code highlighted in red blue adds a new field. The code highlighted in green adds a new method.

3.6. Refactoring operations in the devise pattern

The second SPLIcitar server exemplified in this paper is used to create SPLs based on the devise pattern. In this section, we overview how the devise pattern capabilities are mapped to the corresponding SPLIcitar capabilities to support the SPL extraction from Java applications. With regards to the feature action declaration capability, the discussion will be split in two paragraphs for a better readability: the first concerns feature actions created through a modify operation whereas the second concerns feature actions created through an add operation. At server side, the effect of performing add operations and modify operations over the original Java application involves moving a piece of code from a Java class into a feature action class.

Features declaration. According to the devise pattern, any Java class that inherits (directly or indirectly) from the `Feature` superclass is considered a feature. Therefore, the devise pattern SPLIcitar server responds to a feature declaration request by creating an initially empty class that inherits from `Feature`. The class can optionally be abstract and annotated with annotations such as `@OrGroup`, `@AlternativeGroup` and `@Mandatory` according to the type of feature being declared, as described in Section 2.1. In other words, the devise pattern SPLIcitar server populates the workspace with a hierarchy of classes that mirrors the FM declared by the user. The general project structure of an SPL is exemplified in Fig. 3 that represents the class diagram of the `Feature` hierarchy, the

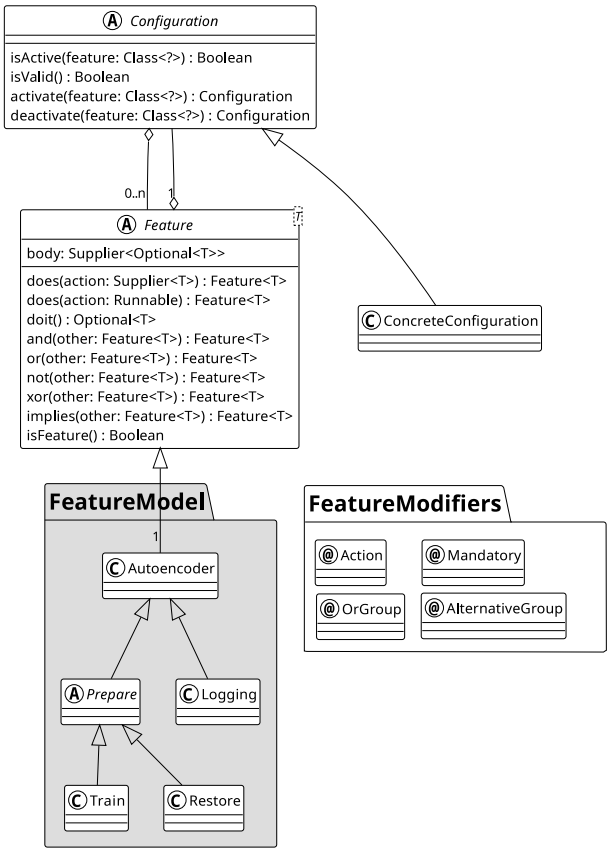

Figure 3. Class diagram of an exemplary FM using the devise pattern.

```
1  class Foo {
2      //class Bar extends Feature
3      private Bar barAction = new BarAction();
5      public void bar() {
6          /* ... */
7          barAction.doit();
8          /* ... */
9      }
10 }
```

Listing 4 Insertion point declaration using the devise pattern. The code added by the refactoring operation is highlighted in green.

```
1  public class Subtractive {
2      public void bar() {
3          /* ... */
4          new Bar()
5              .does(() ->
6                  EntryHelper.copyEntryField(this, null)
7              )
8              .doit();
9          /* ... */
10     }
11 }
```

Listing 5 Subtractive features using the devise pattern. The code in green is removed when the `Bar` feature is inactive. This behavior is not compliant with SPLLIFT.

available modifiers and the application programming interface for the `Feature` and the `Configuration` classes.

Insertion points declaration. In the devise pattern, modify feature actions are declared through the Java functional interface.

```
1  public class Bar extends Feature {}
2  @Action
3  public class BarAction extends Bar {
4      public BarAction() { this
5          .does(() ->
6              EntryHelper.copyEntryField(this, null)
7          );
8      }
9  }
10 public class Additive {
11     public void bar() {
12         /* ... */
13         new BarAction().doit();
14         /* ... */
15     }
16 }
```

Listing 6 Additive features using the devise pattern. The code in green is added to the insertion point highlighted in red when the `Bar` feature is active. This behavior is compliant with SPLLIFT.

```
1  public class Foo {
2      private AddOperations added = new AddOperations();
3      public void bar() {
4          /* ... */
5          new LogField(added).doit();          *
6          /* ... */
7      }
8  }
9  public class Bar extends Feature {}
10 @Action
11 public class AddOperations extends Bar
12     implements NewInterface {
14     public NewField field = new NewField();
16     public NewField getField(){
17         return this.field;
18     }
19 }
20 @Action
21 public class LogField extends Bar {
22     public LogField(AddOperations added) {    *
23         this.does(() ->
24             System.out.println(added.getField())
25         );
26     }
27 }
```

Listing 7 Code needed to perform add operations using the devise pattern. The code highlighted in red adds an interface to the class declaration. The code highlighted in red blue adds a new field. The code highlighted in green adds a new method.

Therefore, the insertion points are the points of the application in which the feature action is actually executed. In terms of the devise pattern, this corresponds to calling the `doit` method on a instance of a class inheriting from `Feature`. This is exemplified in Listing 4. In this example, the `Foo` class holds a reference to all feature actions that must be inserted within its code. The code of the `barAction` feature action is inserted on line 7 when the `doit` method is called, but only if the `Bar` feature is active in the current product configuration.

Modify operations feature actions. Modify operations over the body of methods using the devise pattern is done by *devising* feature actions, while delaying their execution until the activation status of the feature and its validity can be determined based on a configuration. Feature actions can be devised by passing a function to the `does` method of an object that inherits from the `Feature` class. The `does` method could be invoked at any point of the execution, however this would not be compliant with the monotonic reasoning requirement of SPLLIFT because feature actions devised inside the base feature represent subtractive features when the corresponding feature is inactive, as shown in Listing 5. Therefore, the SPLLIFT server for the devise pattern does

not perform refactoring operations such as in Listing 5. Instead, the SPLIFT server implements feature actions in external classes annotated with the `@Action` annotation, as shown in Listing 6. In this example, the insertion point at line 13 is devised with the semantics from the `BarAction` class, lines 5–7. This modify operation is strictly additive since activating the `Bar` feature only adds code to the base feature whereas no code is added when the `Bar` feature is inactive.

Add operations feature actions. Since the devise pattern is designed to work on vanilla Java without support of any external tools, it is impossible to add fields, methods and interfaces to an existing class in the traditional sense. Therefore, the devise pattern performs add operations by moving fields, methods and interfaces from the code of the base feature into the code of feature action classes. Such an example is shown in Listing 7. The feature action class in this example (`AddOperations`, on lines 11–19) adds three elements to the `Foo` class: the code highlighted in red adds an interface, the code highlighted in blue adds a new field and the code highlighted in green adds a new method, respectively. Then, the added elements are accessed by passing an instance of the `AddOperations` class to any relevant feature action, as done when passing `added` on line 5 of Listing 7. Notice how this usage of the devise pattern, marked with ✻, matches the previously mentioned manual refactoring in which additional information is passed to a feature action by the user. In this case, the `LogField` feature action needs pieces of information stored within an `AddOperations` object, that must therefore be passed as an argument when invoking the feature action.

3.7. Generalization

The refactoring approach presented in this section can be generalized beyond the two examples we provided as long as the chosen SPL development tool can support the SPLIFT capabilities. In summary, the contribution of this section is a general framework for the standardization of the refactoring process of applications into SPLs. To implement this framework, developers must implement a server that accepts the following types of requests:

- feature declaration;
- insertion point declaration;
- feature actions resulting from modify operations;
- feature actions resulting from add operations.

Each of these requests must be translated into a refactoring operation over the workspace, by mapping them to the capabilities of the chosen SPL development tool.

3.8. Limitations

When performing the refactoring of a variability unaware application into a SPL according to SPLIFT, the result is an increasing monotonic SPL—*i.e*, an SPL that contains only additive features. This limitation allows the framework to be less restrictive with regards to the capabilities of the SPL development tool, since no complex capabilities are needed to be compliant with SPLIFT. This limitation to the refactoring operations causes no loss of generality according to the additive universe conjecture. Therefore it should be possible to implement any SPL while being limited to the basic capabilities offered by SPLIFT. However, transforming a non-monotonic SPL into an increasing monotonic SPL may not be trivial or convenient, depending on the application. Such a transformation may also reduce the comprehensibility of the SPL, despite monotonic reasoning being arguably easier in general (Batory, 2021). Therefore, the SPLIFT framework may be extended in a future work by including remove operations among the feature action capabilities, as well as deletion points—*i.e.*, portions of the code of a feature that can be removed by effect of a remove operation. Similarly, existing modify operations could also be extended to support both insertions and deletions at the same time, by effect of a unique feature action in correspondence of modification points. This change would negate the applicability of SPLIFT to some specific SPL development approaches in favor of the creation of more comprehensible SPLs.

4. Evaluation

In this section, we assess the ability of SPLIFT of abstracting the implementation details of SPL approaches to the developer when refactoring variability-unaware projects into SPLs according to the proposed protocol. As a result of this abstraction, SPLIFT should achieve the following goals:

1. developers should write less lines of code, as most of the boilerplate code is generated and handled by the server;
2. developers require less familiarity with the SPL approach, as most of their complexity is handled by the server.

This evaluation aims to answer the research questions introduced in Section 1, to determine whether these hypotheses are correct. To answer **RQ_1**, we will measure the effectiveness of the SPLIFT by comparing the amount of code written manually by developers against the amount of code written automatically by SPLIFT. To answer **RQ_2** we will measure the amount of code that is written manually by the developers and that is specific to the chosen SPL approach, such as any call to the devise pattern API or the creation of any pointcuts and advices when using AspectJ. Take Listings 3(a), 3(b) and 3(c) as an example. Listing 3(a) represents the code before any refactoring is performed. Listing 3(b) is the result of the SPLIFT performing four insertion points declarations and four feature action declarations, one for each case of the switch statement. Upon receiving each request, the server automatically generates the code of each feature action; for instance lines 11–18 show a feature action for the `DarkTheme` feature. At the same time, the code of the original application is replaced by a call to the feature action. These changes are the result of SPLIFT performing an insertion point declaration and a feature action declaration. So far, the code was written automatically by SPLIFT: the developer only selected the portions of code to be refactored and the destination feature for each action. Instead, Listing 3(c) shows the same application after the user performed additional refactoring over Listing 3(b). In this case the refactoring opportunity was spotted and then manually performed by the user, to leverage the peculiarities of the devise pattern. Given this example, we measure the code written by the SPLIFT server as the difference between Listing 3(a) and 3(b) and we measure the code written by the developer as the difference between Listing 3(b) and 3(c). These changes count towards measuring the results for answering **RQ_1**. Additionally, any code written manually by the user that requires a knowledge of the devise pattern API (such as, the call to the `xor()` method) counts towards measuring the results for answering **RQ_2**.

4.1. Case studies

The systems under testing will be two variability-unaware open source projects from GitHub which we refactored into SPLs using SPLIFT: Pixelitor.[11] and JPass[12] Both projects are written entirely in Java by developers unrelated to the authors of this

[11] https://github.com/lbalazscs/Pixelitor
[12] https://github.com/gaborbata/jpass

```
switch (theme) {
  case "Dark":
    UIManager.setLookAndFeel(
            "com.formdev.flatlaf.FlatDarculaLaf");
    break;
  case "Light":
    UIManager.setLookAndFeel(
            "com.formdev.flatlaf.FlatIntelliJLaf");
    break;
  case "Nibus":
    UIManager.setLookAndFeel(
            "javax.swing.plaf.nimbus.NimbusLookAndFeel");
    break;
  case "System":
  default:
    UIManager.setLookAndFeel(
            UIManager.getSystemLookAndFeelClassName());
    break;
}
```

(a) Original application.

```
switch theme {
  case "Dark": new DarkTheme.Action1().doit(); break;
  case "Light": new Lightheme.Action1().doit(); break;
  case "Nimbus": new NimbusTheme.Action1().doit(); break;
  case "System":
  default: new SystemTheme.Action1().doit(); break;
}

public class DarkTheme<T> extends Theme<T> {
  @Action
  public static class Action1<T> extends DarkTheme {
    public Action1() {
      this.does(() -> {
        UIManager.setLookAndFeel(
          "com.formdev.flatlaf.FlatDarculaLaf");
      });
    }
  }
}
/* Code of the other feature actions... */
```

(b) Refactoring of application 3(a) by the devise pattern SPLIS2 server.

```
new DarkTheme.Action1().xor(
  new NimbusTheme.Action1(),
  new LightTheme.Action1(),
  new SystemTheme.Action1()
).doit();

public class DarkTheme<T> extends Theme<T> {
  @Action
  public static class Action1<T> extends DarkTheme {
    public Action1() {
      this.does(() -> {
        UIManager.setLookAndFeel(
          "com.formdev.flatlaf.FlatDarculaLaf");
      });
    }
  }
}
/* Code of the other feature actions... */
```

(c) Further refactoring of application 3(b) by the developer.

Listing 3 Refactoring phases of the theme selection feature.

work and offer several variability points that could be turned into features. Moreover, we chose projects with a graphical user interface so that any changes to the product configuration are immediately apparent during testing. In order to refactor the application into an SPL, we firstly identified the variability points of the applications. Then, we identified the feature that are available for each variability point and organized them into a FM. Finally, we converted the FM into a JSON ready to be fed to the SPLIS2 client.

Pixelitor. Pixelitor is an open source graphic image editor that presents image editing functionalities similar to those offered by

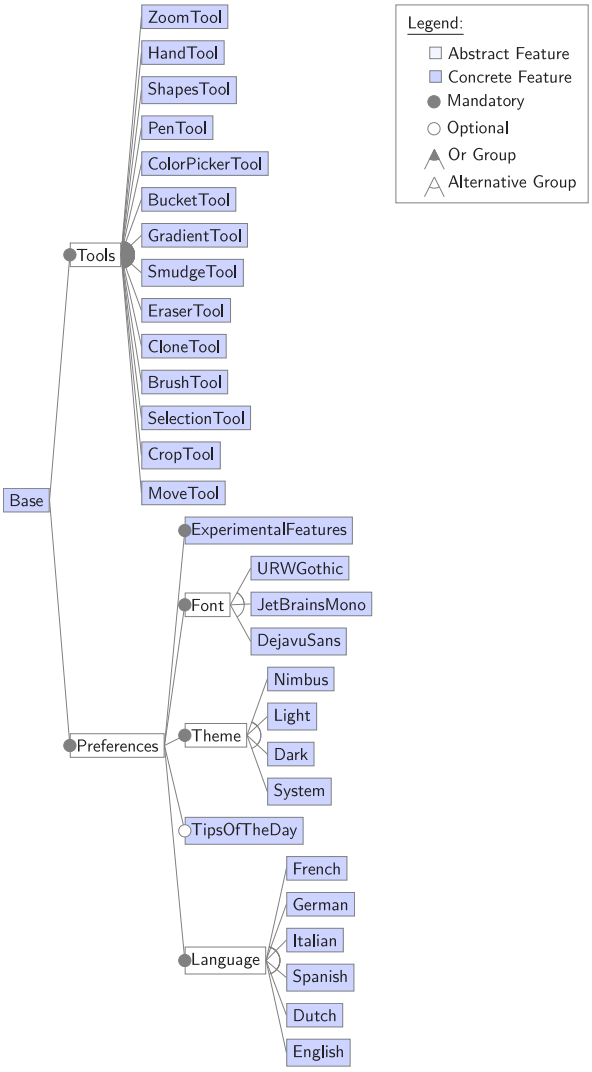

Figure 4. FM of the Pixelitor software family.

other commercial tools such as Adobe Photoshop. We identified 6 variability points in Pixelitor: the available tools, the language, the application font, the optional presence of the "tip of the day" on startup and the usage of any "experimental" features. The domain analysis phase resulted in the FM displayed in Fig. 4. Overall, the Pixelitor software family FM contains 35 features, including 30 concrete features and 5 abstract features. All abstract features are mandatory and are used for structural purposes: `Font`, `Theme` and `Language` represent alternative groups, since only one option can be chosen in each configuration; `Tools` is an optional group instead, therefore any configuration containing at least one tool is valid.

Jpass. JPass is an open source password manager that can be used to organize usernames, passwords, URLs and notes about logins in a safe way. We identified 5 variability points in JPass: the graphical theme of the user interface, the random number generation algorithm, password length, date formatting, and whether the clipboard should be cleared upon exiting the program. The domain analysis phase resulted in the FM displayed in Fig. 5. Overall, the JPass software family FM contains 18 features, including 13 concrete features and 5 abstract features. All

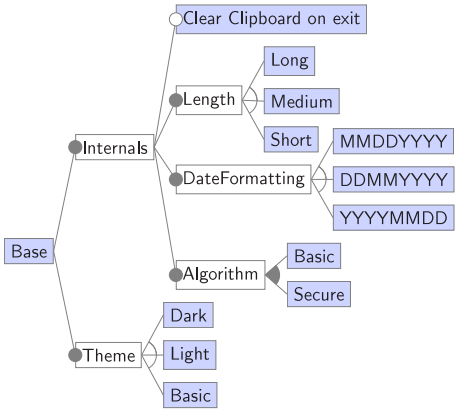

Figure 5. FM of the JPass software family (legend is shared with Fig. 4).

abstract features are mandatory and are used for structural purposes: `Length`, `DateFormatting` and `Theme` represent alternative groups, since only one option can be chosen in each configuration; `Algorithm` is an or group instead, therefore the application can use either one of the algorithms to encrypt passwords or both.

4.2. Implementation

For the purpose of this experiment, we implemented the SPLLSP specification using the two SPL approaches presented in Section 3. The servers for both approaches are implemented using the LSP4J[13] LSP framework for Java. Both servers communicate with a Visual Studio Code[14] (VS Code) SPLLSP client written in Typescript. In particular, the VS Code editor is extended to support the following capabilities:

1. the ability to reify a JSON FM into code and/or project structure;
2. the ability to refactor code from the original application into feature actions pertaining a specific feature from the FM—*i.e.* to request insertion point declarations and modify operations to the server. This refactoring process is initiated within the client by selecting the code portion to be refactored (*e.g.*, with the mouse), performing the refactoring operation from the menu and then choosing the target feature.

Please consider that we did not include any additional capabilities, nor used other refactoring capabilities offered by VS Code to avoid affecting the results of this evaluation. Therefore, the same refactoring performed in a more sophisticated development environment with more complex capabilities could further reduce the development effort. Next, we discuss how each of the two available servers implements the capabilities offered by the client.

Aspectj SPLLSP server. The reification process of the FM starts when the developer selects the feature declaration refactoring operation from the command menu and provides a JSON representation of the FM to the VS Code client. The VS Code client translates the JSON representation of the FM into feature declaration requests sent to the server. Upon receiving these requests, the AspectJ SPLLSP server creates a directory for each feature of the FM unless it already exists. Additionally, the directories are organized following the same tree structure of the FM. An example of such reification process was discussed in Section 3.4 and exemplified in Fig. 2 ("Refactored Application" column).

Modify operations are performed as follows:

At client side:

- The developer selects a code snippet that must be refactored into a feature action.
- The developer requests from the list of available commands a refactoring to the client application.
- The command opens a drop-down menu from which the developer can select the target feature for the new feature action.
- The client translates the refactoring requests into messages to be sent to the SPLLSP server—*i.e.*, an insertion point declaration and a modify operation messages.

At server side:

- The SPLLSP server receives the two messages.
- The SPLLSP server adds a method (that will be used as a hook) to the current class; by default, hook methods are created with the signature `public static Optional<?> __pointcut__n()`, where `n` is the number of hook methods in the class, starting from 0. The default return value of the hook methods is an `Optional.empty()` value. Both can later be changed by the developer if needed.
- The SPLLSP server replaces the selected code with a call to the hook method (insertion point declaration);
- The SPLLSP server generates a file for a new aspect; by default the aspect is called `Aspectn`, where `n` is the number of feature actions in that feature—*i.e.*, the number of files in that directory—starting from 0. It can later be changed by the developer if needed.
- The SPLLSP server generates an around advice within the new aspect and moves the selected code into this advice (modify operation, as in Listing 2).
- The SPLLSP server adds a pointcut to the new aspect that selects the join point represented by the call to the hook method. By default, the pointcut is called `Action` and takes no arguments. This can later be changed by the developer if needed.

These actions are repeated for each feature action that the developer spots within the source code of the original application, each time refining the feature actions that pertain a variability point. Once the refactoring process is complete—*i.e.*, when the developer does not find any more variability point within the original application—products are derived from the list of active features by weaving the respective aspects into the Java code using the ajc compiler. This implementation has some limitations: constraints are not enforced, therefore invalid configurations could be compiled and potentially executed. These problems can be solved through additional tooling support, such as by using the configuration editor provided by FeatureIDE (Pereira et al., 2016) to derive a valid configuration. In fact, notice that product validity is not a code-related problem, but rather a property that must be enforced by the model. Such a support could be added, for instance, by developing a SPLLSP client for FeatureIDE or another variability management tool.

Devise pattern SPLLSP server. The reification process of the FM starts when the developer selects the feature declaration refactoring operation from the command menu and provides a JSON representation of the FM to the VS Code client. The VS Code client translates the JSON representation of the FM into feature declaration requests to be sent to the server. Upon receiving these requests, the devise pattern SPLLSP server creates as many

13 https://projects.eclipse.org/projects/technology.lsp4j
14 https://code.visualstudio.com/

Java files as the number of features contained in the FM, unless they already exist. In this case, the server does not create a complex directory structure mirroring the FM tree structure to avoid having to deal with several different imports; instead, all features are generated within the `feature` package and all classes that need to declare a variability point and to perform a feature action will just `import features.*`. The SPLIS server does not support automatic imports. Instead, this capability is provided by most commonly-used development environments by default.

Modify operations are performed as follows:

At client side:

- The developer selects a code snippet that must be refactored into a feature action.
- The developer requests from the list of available commands a refactoring to the client application.
- The command opens a drop-down menu from which the developer can select the target feature for the new feature action.
- The client translates the refactoring requests into messages to be sent to the SPLIS server—*i.e.*, an insertion point declaration and a modify operation messages.

At server side:

- the SPLIS server receives the two messages
- the SPLIS server adds a new feature action to the feature class specified in the modify operation request (modify operation);
- the SPLIS server replaces the selected code with a call to the `doit()` method of the new feature action (insertion point declaration).

Notice how the client side operations are the same for both the implementations of SPLIS servers. These actions are repeated for each feature action that the developer spots within the source code of the original application, each time refining the feature actions that pertain a variability point. Once the refactoring process is complete—*i.e.*, when the developer does not find any more variability points within the original application—products are derived from the list of active features by creating a Java class that extends the `Configuration` abstract class from the devise pattern, as shown in Fig. 3. Compared to AspectJ, the devise pattern can express and enforce constraints from the FM directly at source level to prevent invalid configurations to ever be executed without the need of any external tools. However, we did not implement a SPLIS server that automatically enforces the constraints during the translation to keep the two server implementations aligned in terms of capabilities; this should provide a fair comparison between the two approaches.

4.3. Evaluation process

For each of the two case study projects, we used the SPLIS VS Code client and the two SPLIS server implementations to reify the JSON representations of the FM into code. Then, we identified the variability points of each application and performed the required refactoring requests to declare insertion points and feature actions. Variability points were spotted manually by the developer, but with some tooling support. For instance, we prioritized the inspection of portions of code that used switch and if statements to determine if the condition was the result of a static configuration (and therefore suitable to be refactored into a variability point) or resolved at runtime (and therefore not applicable). Similarly, we refactored any code using textual configuration files (such as, application preferences) so that the configuration is determined by the FM. Finally, we adapted the code generated by the SPLIS server to best fit the situation—such as, adding any call to the devise pattern API to express cross-tree constraints.

This process was undergone for both SPLIS server implementations and for refactoring both case studies into SPLs. Overall, this resulted in a total four experiments:

1. the Pixelitor SPL using AspectJ;
2. the Pixelitor SPL using the devise pattern;
3. the JPass SPL using AspectJ;
4. the JPass SPL using the devise pattern.

Upon completion of the four applications, we measured the amount of code written in each case by the SPLIS server and by the developer respectively. We also measured the amount of code that must be written by the developer and that requires knowledge of the underlying SPL development approach to be written—such as, using the API that are specific to one of the approaches. Finally, we collected the results.[15]

Table 1
JPass and Pixelitor changes made by the server and by the user.

Project Refactoring	Metric	Server	User	Both
JPass AspectJ	file changed	17	16	33
	line inserted	283	38	321
	line deleted	86	46	132
	char inserted	5292	973	6265
	char deleted	847	943	1790
JPass devise	file changed	26	20	46
	line inserted	344	61	405
	line deleted	0	48	48
	char inserted	7171	1850	9021
	char deleted	0	1358	1358
Pixelitor AspectJ	file changed	41	37	78
	line inserted	684	79	763
	line deleted	60	107	167
	char inserted	13025	1218	14243
	char deleted	883	1628	2511
Pixelitor devise	file changed	49	24	73
	line inserted	764	57	821
	line deleted	5	75	80
	char inserted	17121	1706	18827
	char deleted	0	1471	1471

4.4. Evaluation results

The results of this evaluation are summarized in Table 1. The amount of code written by the server and by the user is measured in terms of files changed, lines inserted and deleted and characters inserted an deleted. Lines are very commonly used by version control systems to measure the difference between subsequent revisions of the same software. However, lines of code are affected by several factors such as code formatting, possibly unrelated to the actual complexity of the changes. To compensate, notice that in Table 1 newline characters do not count towards the amount of inserted and deleted characters. Most notably, most of the code is automatically generated in all four experiments whereas only a small portion is written by the developer. In particular, we refrained from automatically generating any code that can be generated by other refactoring capabilities offered by the LSP and by development environments in general, such as any automatic import. In any scenario in which a feature action requires parameters passed by the caller (as in Listing 6, line 22), the server does not add them automatically, because such parameters can easily be added as method arguments using other refactoring capabilities of the development environment.

[15] The source code used in this evaluation is publicly available at https://zenodo.org/record/7677061.

Similarly, as mentioned in the previous section, the devise pattern SPLIF server does not automatically enforce feature constraints using `and()`, `or()`, `not()`, `xor()` and `implies()` method calls, which are instead added manually by the user if needed. This choice was made to avoid tampering the results by counting the code generated by third party plugins towards the results obtained by SPLIF. As a result, our evaluation is pessimistic: our results are the lower bound obtained when SPLIF is used without any additional tooling support, but can easily be improved upon by using more sophisticated development environments.

For example, using AspectJ the SPLIF server writes 88% of the total lines of code needed to refactor JPass into an SPL based on the messages received by the SPLIF client, with no additional input provided by the user. Going into more detail, 84% of the total characters are written by the server. Instead, many deletions are performed manually by the user (35%), since developers can identify optimizations to save on the amount of boilerplate code written by the SPLIF server. The results are similar when using the devise pattern SPLIF server: performing the same refactoring on JPass saves 85% of all the lines of code and 79% of the written characters. Table 1 shows that the results are mostly consistent across all four experiments, with some outliers such as the devise pattern SPLIF server barely perform any deletions (5 lines in total) on any of the two case studies.

It should be noted that, using the devise pattern, it generally requires more effort with regards to the AspectJ versions in terms of lines of code being written by the user. This is reflected by an overall increase of the amount of code needed to express SPLs using the devise pattern: on average, the SPLIF server for the devise pattern writes 16% more lines of code compared to the SPLIF server for AspectJ; also, on average the developers using the SPLIF server for the devise pattern write 16% more lines of code compared to the SPLIF server for AspectJ. This is due to the devise pattern accounting for the expression of cross-tree constraints and feature dependencies directly at source code level, which usually requires more boilerplate code. Conversely, the AspectJ does not account for the FM structure nor its constraints which saves the amount of code being written but may allow the execution of invalid configurations.

With regards to the scale of the two projects, as already shown in Figs. 4 and 5, the FM for Pixelitor contains 94% more features compared to the FM for JPass. This of course leads to an increase in the amount of code needed to refactor the application into an SPL.

Table 1 does not report the amount of code needed to perform each refactoring and that requires a specific knowledge of the SPL development approach being used for brevity. Instead, the results are directly listed here. Using the devise pattern, the user has to write a total of 10 lines of code (or 117 characters) that need a specific knowledge of the devise pattern API to refactor JPass into an SPL. For Pixelitor, the same refactoring requires 21 lines (or 433 characters) that use the devise pattern API. In both cases, this code is always associated to a call to the API needed to express cross-tree constraints, such as the `xor()` method. A more complex SPLIF server could inspect the feature model to automatically generate these calls, but we refrained from doing so in this work to keep aligned the capabilities provided by the two SPLIF servers. In fact, interestingly, refactoring a project into an SPL using AspectJ never required the user to write code using the AspectJ syntax in any of our experiments. Instead, the SPLIF server could generate all the AspectJ code automatically whereas the user only made changes to the Java code.

4.5. Discussion

RQ$_1$ *How effective is SPLIF at reducing the effort of refactoring standard applications into SPLs?* For all four case studies analyzed in this work, we noticed a substantial save in the amount of code needed to be written by the developers to refactor a Java project into an SPL using SPLIF. More precisely, we achieve the following results:

- AspectJ-JPass—88% of the lines of code;
- AspectJ-Pixelitor—90% of the lines of code;
- devise pattern-JPass—85% of the lines of code;
- devise pattern-Pixelitor—93% of the lines of code.

Given these results, we conclude that an approach based on the LSP such as SPLIF is effective at reducing the refactoring effort needed to convert Java applications into SPLs.

RQ$_2$ *How effective is SPLIF at abstracting the spl development tools/approaches to developers?* For all four case studies analyzed in this work, we noticed a substantial save in the amount of code that requires approach-specific knowledge needed to be written by the developers, such as code using the AspectJ syntax or the devise pattern API. More precisely, we achieve the following results:

- AspectJ-JPass—0% of the lines of code;
- AspectJ-Pixelitor—0% of the lines of code;
- devise pattern-JPass—16% of the lines of code;
- devise pattern-Pixelitor—37% of the lines of code.

Given these results, we conclude that SPLIF is effective at avoiding the need for the developers to know AspectJ. Moreover, knowledge of the devise pattern API is limited to the methods used to express cross-tree constraints. In both cases, an approach based on the LSP was capable of abstracting the majority of the implementation details of the underlying SPL approach.

For both **RQ$_1$** and **RQ$_2$**, we measured the results using only the SPLIF client–server implementations, without any external tools. Additional tooling support could further improve the results by addressing other insertions and deletions, such as imports and method call parameters.

4.6. Threats to validity

The validity of this evaluation could be affected by internal and external threats (Wohlin et al., 2003) that could affect the validity of this evaluation. With regards to internal validity, one of the two approaches used as a case study for this evaluation, share the authors with this work. This may affect the validity of the results due to the expertise that we gained with the tool. However, this issue is limited by the fact that the devise pattern was designed earlier and was unrelated to SPLIF. To further stem this validity issue we did not leverage any peculiarities of the devise pattern in the development of the SPLIF server: in fact, between the two approaches, the AspectJ implementation shows better results overall. Similarly, the evaluation may be affected by the fact that the refactoring was performed by one of the authors of this work. To avoid this threat, the development concerns were separated so that the refactoring capabilities offered by SPLIF and the SPLs were developed by different authors; moreover, we ensured that the framework was as general as possible and was not changed or adapted to ease the refactoring process of these specific SPLs and/or SPL development approaches. One additional issue is that development environment may affect the refactoring process and thus the results: for example, the environment used could support automatic imports for only one of the two approaches, thus affecting the number of lines written by the

user. To avoid this issue we used a plain installation of VS Code running on Docker,[16] without any plugin other than the SPJLS2 client installed.

With regards to external validity, despite the SPJLS2 being general, our evaluation focuses solely on two specific approaches, therefore its applicability may be limited when considering different approaches. To avoid this issue we kept the SPJLS2 specification as general and as limited as possible with regards to expressive power so that a larger variety of SPL development approaches can meet its requirements. Moreover, the chosen approaches for the evaluation are very different, one being annotative and running on stock Java (devise) and the other being compositional and using an external tool (AspectJ).

The evaluation and the SPJLS2 framework in general are based on increasing monotonic SPLs, thus the results do not apply to non-monotonic and decreasing monotonic SPLs. However, it is widely accepted that any non-monotonic SPL can be converted into a monotonic SPL (either increasing or decreasing) that has the same products thanks to the additive universe conjecture (Batory, 2021; Damiani and Lienhardt, 2016).

5. Related work

The LSP is an increasingly popular protocol to face the challenging task of developing modern IDEs. The LSP has mainly been used to implement portable IDE support for over 121 programming languages (Barros et al., 2022), quickly becoming the *de facto* standard to realize editing support for languages. Researchers investigated ways to use the LSP to ease the development activity. For instance, Bünder and Kuchen (2019) discuss how textual domain-specific languages can benefit from the LSP by providing different views over the same program to different developers, depending on their level of expertise. Such an approach differs from Jetbrain's Meta-Programming System (Völter and Pech, 2012), in which several editors can be used to edit the same program, by being able to be used across different development environments, possibly implemented as web applications. The LSP was also used as an infrastructure to simplify the development of graphical modeling tools (Rodriguez-Echeverría et al., 2018) and to migrate Eclipse-based graphical modeling editors to the web (Rani et al., 2020).

In this work, we exemplified the relation between SPJLS2 and two approaches for the development of SPLs, namely AspectJ and the devise pattern. However, a variety of approaches to support the definition of SPLs have been proposed by researchers. Most are based on preprocessors and composers that work outside of the capabilities of the base language, such as AHEAD (Batory et al., 2004), FeatureHouse (Apel et al., 2009) and FeatureC++ (Apel et al., 2005), each bringing forth their own flavor of feature composition. On the other hand, approaches that exist within the boundaries of the base language are more scarce. The variability modules in Java architectural pattern (Setyautami and Hähnle, 2021) is based on variability modules and delta-oriented programming (Schaefer et al., 2010): each feature is implemented using the Java modules and decorators (Gamma et al., 1995) applied over the base feature. Seidl et al. (Seidl et al., 2017) presented a generative SPL development method using variability-aware versions of the observer, strategy, template method and composite (Gamma et al., 1995) patterns and introduced the Family Role Model as a notation to capture constraints on the variable application. In Apel et al.'s book «Feature-Oriented Software Product Lines» (Apel et al., 2013) an entire chapter is dedicated to «Classic, Language-Based Variability Mechanisms», ranging from traditional `if` statements to more sophisticated and flexible programming patterns to support variability. Compared to the variability modules in Java and the devise pattern, the solutions are proposed in a less standardized manner and must be tailored to fit specific problems.

The need for standardization in SPL development is still an open problem that we tried to face in this work. The problem is partially addressed by tools and environments, in which the definition and handling of FMs is standardized by means of the tooling support built on top of them. Among most well-known contributions in this direction some examples are FeatureIDE (Thüm et al., 2014; Meinicke et al., 2016), SPLOT (Mendonca et al., 2009), FAMA (Benavides et al., 2013) and pure::variants (Beuche, 2016). FeatureIDE is an SPL development environment that supports the FM construction, the management of software artifacts, the configuration and product derivation. Moreover, it supports several different composers within the same environment. SPLOT is a web-based collection of tools to reason about FMs and for the interactive configuration of product variants. FAMA is an extensible framework for the automated analysis of FMs. pure::variants is a tool for variability management that supports the entire lifecycle of product lines by connecting the various SPL development activities through proper tooling.

However, all these tools share the same limitations. First, they standardize the interaction with the FMs and their configuration, but the developers still have to face a steep barrier to entry to interact with actual software artifacts and when using different composers. Second, these tools exist within a specific development environment whereas an approach based on the LSP should be easier to port to a different context (by developing the client application for a new IDE). In fact, Shatnawi and Cunningham (Shatnawi and Cunningham, 2021) mentioned the difficulty of specifying and maintaining FMs due to the SPL development tools requiring specific knowledge and skills. Their contribution shares with ours the need for these tools to be easily accessible to any developer with limited training, possibly by using mainstream technologies. On a similar note, Chimalakonda and Lee (Chimalakonda and Hyung, 2016) discussed the inconsistency and incompatibility of tools and methods in SPLs and the need for the introduction of standards in their development. They argue that the diversified range of tools and methods is one of the primary hindrances to the adoption of SPLs in the industry. They call for an improvement of the standards in the area of software and systems product lines and invite the SPL community to propose frameworks, processes and activities that could potentially become standard practices in the development of SPLs.

6. Conclusions & future works

SPLs are a powerful engineering approach towards expressing and implementing highly-variable software systems by allowing for massive reuse of already implemented software features across several software variants. However, it is often hard to reuse existing features in a different SPL because each approach to the development of SPLs follows a different refactoring process and tools with a different syntax. For the same reasons, developers cannot use the knowledge they acquired using an SPL development tool in a different context. In this work we presented SPJLS2 as a framework that aims at standardizing the refactoring process of variability-unaware applications into SPLs by expressing the refactoring operations as messages sent across a client–server process, effectively decoupling the refactoring made by the developer from the actual implementation details. Our evaluation showed that this decoupling can greatly reduce the amount of code that developers need to write in order to extract an SPL; moreover, the developers mostly interact with

16 https://www.docker.com/

an identical client application, thus the implementation details of the chosen SPL approach are abstracted: if the SPL development approach is changed by the stakeholders, the developers can still perform the refactoring with the same IDE and following the same process, with limited training with regards to the new tools and composition techniques.

However, a similar approach could also be used in the future to support the SPL migration from one approach to the others, to increase the reusability of the same features across different SPL approaches and tools. For instance, the LSP could be used to provide different views of the same SPL to several developers working concurrently. The protocol would keep an internal representation of the SPL and its features, provide each developer with the chosen view and translate any changes made to a view into the underlying intermediate representation. In such an approach, support for an additional SPL development approach could be achieved simply by implementing a client and a server for that approach instead of needing an *ad hoc* translation for each pair of approaches. Moreover, future works will investigate the applicability of a variant of the SPJLQ2 framework that also supports remove operations and deletion points, as well as more complex modify operations and modification points.

CRediT authorship contribution statement

Francesco Bertolotti: Conceptualization, Methodology, Software, Investigation, Writing – original draft, Writing – review & editing. **Walter Cazzola:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing, Supervision, Project administration, Funding acquisition. **Luca Favalli:** Conceptualization, Methodology, Software, Investigation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

We wish to confirm that there are not known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Data availability

Data will be made available on request.

References

Apel, S., Batory, D., Kästner, C., Saale, G., 2013. Feature-Oriented Software Product Lines. Springer.

Apel, S., Kästner, C., Lengauer, C., 2009. Language-independent. In: Automated Software Composition. In: ICSE'09, IEEE, Canada, pp. 221–231.

Apel, S., Leich, T., Rosenmüller, M., Saake, G., 2005. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: GPCE'05, Springer, Tallin, Estonia, pp. 125–140.

Barros, D., Peldszus, S., Assunção, W.K.G., Berger, T., 2022. Editing Support for Software Languages: Implementation Practices in Language Server Protocols. In: MoDELS'22, ACM, Montréal, Canada, pp. 232–243.

Batory, D., 2021. Automated Software Design, Vol. 1, second ed..

Batory, D., Sarvela, J.N., Rauschmayer, A., 2004. Scaling step-wise refinement. IEEE Trans. Softw. Eng. 30, 355–371.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years Later: A literature review. Inform. Syst. 35, 615–636.

Benavides, D., Trinidad, P., Ruiz-Cortés, A., Segura, S., 2013. FaMa. In: Systems and Software Variability Management: Concepts, Tools and Experiences. Springer, pp. 163–171.

Bergel, A., Ducasse, S., Nierstrasz, O., 2005. Classbox/J: Controlling the scope of change in Java. In: OOPSLA 2005.

Bertolotti, F., Cazzola, W., Favalli, L., 2022. Features. In: Believe It or Not! A Design Pattern for First-Class Citizen Features on Stock JVM. In: SPLC'22, ACM, Graz, Austria, pp. 32–42.

Beuche, D., 2016. Using Pure: Variants across the Product Line Lifecycle. In: SPLC'16, ACM, Beijing, China, pp. 333–336.

Bracha, G., Cook, W., 1990. Mixin-Based Inheritance. In: OOPSLA/ECOOP'90, ACM, Ottawa, Canada, pp. 303–311.

Bünder, H., Kuchen, H., 2019. Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol. In: MODELSWARD'19, Springer, Prague, Czech Republic, pp. 225–245.

Cazzola, W., Vacchi, E., 2013. Fine-Grained Annotations for Pointcuts with a Finer Granularity. In: SAC'13, Coimbra, Portugal, pp. 1709–1714.

Cazzola, W., Vacchi, E., 2014. @Java: Bringing a richer annotation model to Java. Comput. Lang. Syst. Struct. 40, 2–18.

Chimalakonda, S., Hyung, L.D., 2016. On the evolution of software and systems product line standards. Softw. Eng. Notes 41, 27–30.

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wąsowski, A., 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In: VaMoS'12, Leipzig, Germany, pp. 173–182.

Damiani, F., Lienhardt, M., 2016. Refactoring Delta-Oriented Product Lines to Achieve Monotonicity. In: FMSPLE'16, Eindhoven, pp. 2–16.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., Dantas, F., 2008. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: ICSE'08, ACM, Leipzig, Germany.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Jayatilleke, S., Lai, R., 2018. A systematic review of requirements change management. Inform. Sofw. Technol. 93, 163–185.

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. TR CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, USA.

Kästner, C., Apel, S., 2008. Integrating Compositional and Annotative Approaches for Product Line Engineering. In: McGPLE'08, pp. 35–80.

Kästner, C., Apel, S., Batory, D., 2007. A Case Study Implementing Features Using AspectJ. In: SPLC'07, IEEE, Kyoto, Japan, pp. 223–232.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, B., 2001. An Overview of AspectJ. In: ECOOP'01, Budapest, pp. 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videir. Lopes, C., Loingtier, J.M., Irwin, J., 1997. Aspect-Oriented Programming. In: ECOOP'97, Springer, Helsinki, Finland, pp. 220–242.

Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Ferruccio, D., 2014. DeltaJ 1.5: Delta-Oriented Programming for Java 1.5. In: PPPJ'14, ACM, Cracow, Poland, pp. 63–74.

Krueger, C.W., 2001. Easing the Transition to Software Mass Customization. In: PFE'01, Springer, Bilbao, Spain, pp. 282–293.

Kühn, T., Cazzola, W., 2016. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In: SPLC'16, ACM, Beijing, China, pp. 50–59.

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

Meinicke, J., Thüm, T., Schröter, R., Krieter, S., Benduhn, F., Saake, G., Leich, T., 2016. FeatureIDE: Taming the Preprocessor Wilderness. In: ICSE'16-Companion, IEEE, Austin, TX, USA, pp. 629–632.

Mendonca, M., Branco, M., Cowan, D., 2009. S.P.L.O.T.—Software Product Lines Online Tools. In: Companion OOPSLA'09, pp. 761–762.

Mezini, M., Ostermann, K., 2004. Variability Management with Feature-Oriented Programming and Aspects. In: FSE'04, ACM, pp. 127–136.

Pereira, J.A., Krieter, S., Meinicke, J., Schröter, R., Saake, G., Leich, T., 2016. FeatureIDE: Scalable Product Configuration of Variable Systems. In: ICSR'16, Springer, Limassol, Cyprus, pp. 397–401.

Prehofer, C., 1997. Feature-Oriented Programming: A Fresh Look at Objects. In: ECOOP'97, Springer, Helsinki, Finland, pp. 419–443.

Prehofer, C., 2001. Feature-oriented programming: A new way of object composition. Concur. Comput.: Pract. Exp. 13, 465–501.

Rani, F., Diez, P., Chavarriaga, E., Guerra, E., d. Lara, J., 2020. Automated Migration of EuGENia Graphical Editors to the Web. In: MoDELS'20, ACM, Virtual Event Canada, pp. 71:1–71:7.

Rodriguez-Echeverría, R., Cánova Izquierdo, J.L., Wimmer, M., Cabot, J., 2018. An LSP Infrastructure to Build EMF Language Servers for Web-Deployable Model Editors. In: MDE-Tools'18, Copenhage, pp. 1–10.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-Oriented Programming of Software Product Lines. In: SPLC'10, Springer, Jeju Island, South Korea, pp. 77–91.

Schulze, S., Richers, O., Schaefer, I., 2013. Refactoring Delta-Oriented Software Product Lines. In: AOSD'13, ACM, Fukuoka, Japan, pp. 73–84.

Seidl, C., Schuster, S., Schaefer, I., 2017. Generative software product line development using variability-aware design patterns. Comput. Lang. Syst. Struct. 48, 89–111.

Setyautami, M.R.A., Hähnle, R., 2021. An Architectural Pattern to Realize Multi Software Product Lines in Java. In: VaMoS'21, pp. 9:1–9:9.
Shatnawi, H., Cunningham, H.C., 2021. Encoding Feature Models Using Mainstream JSON Technologies. In: ACM-SE'21, ACM, USA, pp. 146–153.
ter Beek, M.H., Damiani, F., Lienhardt, M., Mazzanti, F., Paolini, L., 2021. Efficient static analysis and verification of featured transition systems. Empir. Softw. Eng. 27.
Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. FeatureIDE: An extensible framework for feature-oriented software development. Sci. Comput. Programm. 79, 70–85.
Völter, M., Pech, V., 2012. Language Modularity with the MPS Language Workbench. In: ICSE'12, IEEE, Zürich, Switzerland, pp. 1449–1450.
Wohlin, C., Höst, M., Henningsson, K., 2003. Empirical Research Methods in Software Engineering. In: Conradi, R., Wang, A.I. (Eds.), Empirical Methods and Studies in Software Enginering: Experiences from ESERNET. In: LNCS 2765, Springer, pp. 7–23.

Francesco Bertolotti is currently a Computer Science Ph.D. student at Università degli Studi di Milano and a member of the ADAPT Laboratory. Previously, he was an assistant researcher at the same University where he also got his master degree in Computer Science. His research interests are programming languages, software quality, machine/deep learning techniques and their reciprocal cross-fertilization. He can be contacted at francesco.bertolotti@unimi.it for any question.

Walter Cazzola is currently an Associate Professor in the Department of Computer Science of the Università degli Studi di Milano, Italy and the Chair of the ADAPT laboratory. Dr. Cazzola designed the mChaRM framework, @Java, [a]C#, Blueprint programming languages and he is currently involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution and comprehension, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is associate editor for the Journal of Computer Languages published by Elsevier. More information about Dr. Cazzola and all his publications are available at http://cazzola.di.unimi.it and he can be contacted at cazzola@di.unimi.it for any question.

Luca Favalli is currently a Computer Science Postdoctoral Researcher at Università degli Studi di Milano. He got his Ph.D. in computer science from the Università degli Studi di Milano. He is involved in the research activity of the ADAPT Lab and in the development of the Neverlang language workbench and of JavAdaptor. His main research interests are software design, software (and language) product lines and dynamic software updating with a focus on how they can be used to ease the learning of programming languages. He can be contacted at favalli@di.unimi.it for any question.