Features, Believe It or Not!*

A Design Pattern for First-Class Citizen Features on Stock JVM

Francesco Bertolotti
Computer Science Department
Università degli Studi di Milano
Milan, Italy
bertolotti@di.unimi.it

Walter Cazzola
Computer Science Department
Università degli Studi di Milano
Milan, Italy
cazzola@di.unimi.it

Luca Favalli
Computer Science Department
Università degli Studi di Milano
Milan, Italy
favalli@di.unimi.it

Abstract

Modern software systems must fulfill the needs of an ever-growing customer base. Due to the innate diversity of human needs, software should be highly customizable and reconfigurable. Researchers and practitioners gained interest in software product lines (SPL), mimicking aspects of product lines in industrial production for the engineering of highly-variable systems. There are two main approaches towards the engineering of SPLs. The first uses macros—such as the `#ifdef` macro in C. The second—called feature-oriented programming (FOP)—uses variability-aware preprocessors called *composers* to generate a program variant from a set of *features* and a *configuration*. Both approaches have disadvantages. Most notably, these approaches are usually not supported by the base language; for instance Java is one of the most commonly used FOP languages among researchers, but it does not support macros rather it relies on the C preprocessor or a custom one to translate macros into actual Java code. As a result, developers must struggle to keep up with the evolution of the base language, hindering the general applicability of SPL engineering. Moreover, to effectively evolve a software configuration and its features, their location must be known. The problem of recording and maintaining traceability information is considered expensive and error-prone and it is once again handled externally through dedicated modeling languages and tools. Instead, to properly convey the FOP paradigm, software features should be treated as first-class citizens using concepts that are proper to the host language, so that the variability can be expressed and analyzed with the same tools used to develop any other software in the same language. In this paper, we present a simple and flexible design pattern for JVM-based languages—dubbed *devise* pattern—that can be used to express feature dependencies and behaviors with a light-weight syntax both at domain analysis and at domain implementation level. To showcase the qualities and feasibility of our approach, we present several variability-aware implementations of a MNIST-encoder—including one using the devise pattern—and compare strengths and weaknesses of each approach.

CCS Concepts

• **Software and its engineering** → **Software design engineering**; **Software product lines**.

Keywords

Software product lines, variability modeling, design patterns

ACM Reference Format:
Francesco Bertolotti, Walter Cazzola, and Luca Favalli. 2022. Features, Believe It or Not!: A Design Pattern for First-Class Citizen Features on Stock JVM. In *26th ACM International Systems and Software Product Line Conference - Volume A (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3546932.3546989

1 Introduction

Product lines are a staple in industrial production for the creation of highly-variable systems. Following the same concepts, software product lines (SPLs) [27] are an increasingly popular technology to support feature reuse and system variability. Ideally, software product line engineering (SPLE) should provide variability mechanisms to accommodate the introduction and removal of crosscutting and non-crosscutting features, as well as their transformation without invasive changes and ripple effects. State-of-the-art SPL development environments—such as FeatureIDE [24, 39]—can cope with all the aspects of the development of a SPL, including construction, management of software artifacts, configuration and product derivation. However, such tools and techniques are not natively supported by the base language and thus the developers have to struggle to keep up with the evolution of the base language—for instance, Java has a 6-month release cycle since March 2021. Moreover, 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 chosen composer tool. Composer tools are preprocessors that translate feature-oriented code into Java code with regards to a chosen configuration. Possible composers are FeatureHouse [39], AHEAD [3], Antenna[1] and AspectJ [25]. However, it is usually possible to avoid using preprocessors thanks to the Java Virtual Machine (JVM) abstractions [12]. To change a composer is usually unfeasible as the SPL has to be rewritten. The tool chain may not support the new composer so the developers have to learn new syntax, tools and a specific development environment. A closely related problem is that of feature *traceability*: recording and maintaining the potentially

*The title pays homage to the American franchise Ripley's Believe It or Not! https://www.ripleys.com/. It wants to emphasize the obviousness that in this work we will talk about features.

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

SPLC '22, September 12–16, 2022, Graz, Austria

ACM ISBN 978-1-4503-9443-7/22/09…$15.00
https://doi.org/10.1145/3546932.3546989

scattered locations of features in the software artifacts for evolution and maintenance purposes is tedious and error-prone [1] especially when changes to the specification cause changes to the implementation and vice versa. While several feature location and variability mining strategies have been proposed [10, 11, 30] and evaluated [22] in the literature, they must be complemented by ad-hoc refactoring strategies to evolve software into a variability-aware SPL. These problems may obstacle the adoption of SPLs as a more wide-spread engineering technique [16] and solving them requires dealing with their inherent complexity. SPLE involves aspects of domain analysis and implementation, requirements analysis and product derivation; the possible configurations are exponential in the number of features and SPLs, e.g., the Linux kernel [37] has several thousands of features whereas the Neverlang.JS implementation of Javascript [7] has hundreds of language features.

As Larry Tesler stated in an interview for Dan Saffer [31]'s «Designing for Interaction» book: «Systems have an inherent amount of complexity that cannot be reduced». This is known as *the law of conservation of complexity* and leaves one question open with regards to complexity: if it cannot be reduced or hidden, then who should be exposed to such a complexity? In this paper, we present an approach in which managing the complexity of software variability is a matter of *design* rather than a matter of *tooling*. In this approach, software features are modeled through concepts the software developers are familiar with, such as composition, inheritance and design patterns. Feature development and their recording are the same development activity, so that tracing is done with the same tools used to analyze normal code: Eclipse and JetBrains' IntelliJ IDEA, as well as most other modern Java IDEs support finding usages of classes and methods, and class hierarchy inspection and refactoring—including any external dependencies. Most developers are already familiar with these tools: using the same abstractions to implement both features and normal classes makes their expertise applicable to FOP at no additional cost. Should the development environment be changed, the same code can be reused with no changes. The same approach can be used as a refactoring framework to complement variability mining techniques or to avoid the feature location activity by explicitly declaring the variability points when a SPL is developed from scratch. To show the applicability of this approach, we present a design pattern for FOP—dubbed *devise pattern*—and a variability-aware MNIST-encoder implemented using an implementation of this pattern.

The remainder of this paper is structured as follows. Sect. 2 presents background terminology and concepts. Sect. 3 presents the essence of the devise pattern. Sect. 4 shows how the pattern can be customized with additional semantics and used to implement a variability-aware MNIST-encoder application. Sect. 5 reviews related work. Finally, in Sect. 6 we draw our conclusions.

2 Background

In this section, we overview background information and terminology that is relevant to this work, including basic concepts SPLs and FOP, and neural networks for the MNIST-encoder case study.

2.1 Software Product Lines

Following the ideas of product line engineering in industrial production, SPLE strives to ease the development of variable software systems, introducing the concepts of software variants and software families. Following the *feature-oriented programming* (FOP) paradigm [28], software products can be described in terms of the *features* they provide. Similar software products that share commonalities but differ for a subset of their features are called *software variants*. A collection of software variants is usually called *software family*. The goal of SPLE is to ease the deployment of software families through formalisms such as the *Feature Model* (FM) [17]. Given a FM, a software variant is identified through a configuration—*i.e.*, a collection of active features. The validity of a configuration is determined by analyzing the FM with regards to active and inactive features: the FM structure determines feature dependencies by defining mandatory features, optional features, or groups, and alternative features, as well as, the simple parent-child relationship, since a feature can be active only if its parent is also active. Dependencies can also be explicitly defined using cross-tree constraints—i.e., Boolean expressions among features of the FM. If the truth value of any cross-tree constraint is false for a configuration then that configuration is invalid. Feature dependencies may cause anomalies such as atomic-sets—*i.e.*, sets of features that must either be all active or all inactive at the same time—and dead features—that can never be active in any valid configuration. Such anomalies can be detected and refactored to improve the quality of a FM; this is an active research area and includes structural [5] and behavioral [4] approaches. In addition to FOP, other paradigms for SPLE such as aspect-oriented programming (AOP) [15] and delta-oriented programming (DOP) [32] exist.

2.2 MNIST-encoder

Neural Networks (NNs) are computational graphs with trainable parameters designed to solve specific tasks. When trained, these parameters are optimized so that a *loss* function is minimized. By minimizing the loss, the NN can learn specific patterns that are useful to solve the intended task. NNs are highly-customizable software systems: the type of the architecture and of the loss function, as well as the number of trainable parameters, are all factors that can substantially affect the NN performance. NNs learn hidden internal representations of the data on which they are trained. These representations can be explicitly trained to satisfy certain properties. For example, *contrastive learning* [8] is a technique in which a NN is trained so that semantically similar data points have close hidden representations. This can be achieved by means of a well-designed loss function. Contrastive learning can be both supervised and self-supervised. In the first case, semantically close data points are known in advance and the NN is trained so that their hidden representation is also close. For example, data points with the same label are trained to be close to each other, whereas data points with different labels are trained to be far apart. In the latter case, semantically close data points are not known beforehand and are instead generated using augmentation pipelines. These kinds of architectures are usually referred to as *encoders*. In this work, we trained a set of NN encoders using contrastive learning on the popular MNIST dataset [19]. It contains 60,000 gray scale images

of 28×28 pixels. Each image represents a numerical digit from 0 to 9. For this reason, we called this application MNIST-encoder.

3 Devise Pattern

To properly frame this contribution, we went back to Prehofer [28, 29]'s work and the origin of FOP. FOP is a model for object-oriented programming which generalizes inheritance. Instead of using a rigid class structure, features are similar to mixins [6] and implement services that can be used by other objects. Therefore, objects behaviors are implemented by leveraging the aggregation of several features whereas more modern software composition tools such as FeatureHouse [2] give up aggregation in favor of superimposition—*i.e.*, the process of composing software artifacts by merging their substructures. In this work, we remain true to the original vision of FOP while taking a more naïve approach, in which classes are the result of feature aggregation. This process is eased by changes recently introduced in object-oriented programming languages—such as lambdas in Java 8. We propose the *devise pattern*, designed to achieve the following FOP goals:

- *separation of concerns*—the modeling code is separated from the implementation code;
- *light* on the domain analyst—the modeling code of a feature and of the FM is minimal, it contains no semantics and can be automatically generated if the FM is already available;
- *light* on the developers—the implementation code of a feature takes little to no boilerplate code (the same magnitude of a `#ifdef` macro in C);
- *flexibility*—the implementation code of feature behavior can either be embedded in the application or separated from it to support information hiding and reuse;
- *statically-checked*—both the modeling code and the implementation code are checked by the stock language compiler.

The *devise pattern*, its participants and its application are structured following Gamma *et al.* [13]'s work on design patterns.

Purpose and Scope. The devise pattern is a *class behavioral* pattern. It deals with the relationship among classes implementing crosscutting concerns (features) and with how these classes and their instances (feature actions) interact and set responsibility.

Intent. Explicitly express the variability points of an algorithm at source level so that they can later be traced and refactored. Keep the FM and its implementation aligned by means of the compiler. Plan the feature semantics ahead and defer their execution until they are ensured to be active in a valid configuration. Render the main application unaware of the underlying configuration.

Motivation. Consider a variability-aware machine learning application in which two different loss functions can be used: Triplets [33] and InfoNCE [26]. These two specific loss functions are not interchangeable and choosing one over the other in a configuration affects the preparation of the training set and the graph of the model to be trained. In both cases, the code is scattered across the main application. The loss function is then a crosscutting concern and can be modeled as a feature. Different features may have constraints with each other: no loss function is needed if the model is restored from memory. Otherwise, either one of them must be active, but not both in the same configuration. At each point of the execution in which a configuration choice is relevant, the main application must explicitly declare a variability point and any dependencies among features which the variability point is concerned with. To summarize, to solve the problem of variability of loss functions it means to solve four sub-problems: to declare the cross-cutting concerns (features declaration), to declare a variability point in the application (variability points declaration), to declare constraints among features (constraints declaration), and to configure product variants (configuration management). A solution to the *feature declaration* problem is to separate the class hierarchy of the main application from the feature hierarchy, so that class instances (objects) and feature instances (feature actions) can be combined at will through aggregation. With this structure, cross-cutting concerns can be identified simply by inspecting the class hierarchy. In our example, each of the two loss functions will inherit from the same `Feature` abstract class. Any other class that does not inherit from `Feature` will not be identified as a cross-cutting concern. A common solution to the *variability point declaration* problem is the usage of conditional compilation with `#ifdef` macros [20]. While this solution is extremely simple, it is usually considered error-prone due to the low level of abstraction. To maintain the benefits of an `#ifdef` while improving the abstraction, a solution would be to separate the declaration of a variability point from its implementation. For example, both Triplets and InfoNCE are implemented in their own classes and the main method only declares the variability point in which one of the two must be chosen in a configuration. A common problem with *constraints declaration* is that feature constraints are usually declared at FM level, so it is hard to ensure that dependencies expressed in the source code align with those declared in the FM. A solution to this issue would be to declare the feature constraints directly at source level: the alignment between the representation at FM level and the implementation can then be checked automatically. For instance, the alternative nature between Triplets and InfoNCE that we discussed earlier should be expressed both at FM level and at source level. Any inconsistencies can be revealed by analyzing the source code against the FM. To solve the *configuration management* problem, the application needs an activation mechanism that handles the execution of each feature: feature actions must be executed if and only if the corresponding feature is active in the current configuration. In this example, the main application is a client for two possible services provided by the alternative Triplets and InfoNCE features. The service that is actually provided when the application is run is determined by a configuration, whose validity is checked against the FM.

Applicability. The devise pattern should be used to manage the variability of SPLs without preprocessors, as discussed in the motivational example above. In particular, the devise pattern can explicitly declare variability points in an application and untangle code from different concerns by refactoring them into features. The scattered locations of features implemented with the devise pattern can be retrieved automatically with common tools such as an IDE. The devise pattern can also be used to defer the execution of a block of code until the validity of a configuration is checked. Finally, the devise pattern offers a viable solution whenever configuration choices are subject to complex constraints.

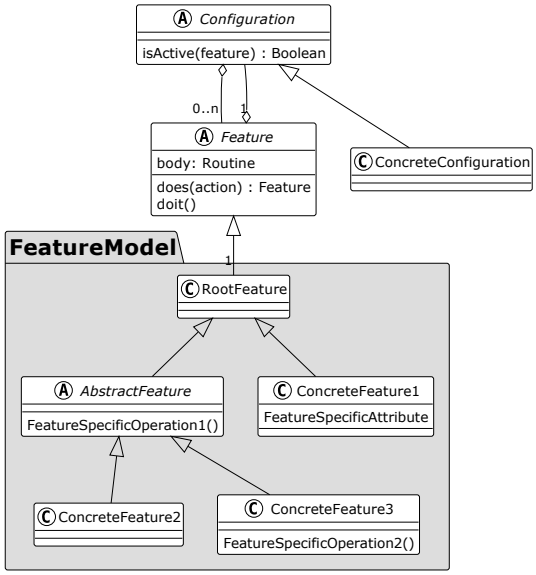


Figure 1: Essential class diagram of the devise pattern.

Structure and Participants. Fig. 1 shows the class diagram of an essential implementation of the devise pattern, representing the hierarchy of the feature classes. Each feature class is used by a client application (not shown in the diagram). There are five main participants to the devise pattern.

- *Feature*—the root of the feature hierarchy. The semantics of a feature action are devised using the `does` method and stored in a `body` field. The `does` method also returns the feature action to allow method call chains. The execution is deferred until the `doit` method is called.
- *Abstract and Concrete Features*—sub-classes of the `Feature` class determine the FM of the SPL. Each direct subclass of `Feature` (`RootFeature` in Fig. 1) is the root of a FM. The complete FM is equivalent to the sub-hierarchy of `RootFeature`, with abstract classes being abstract features and concrete classes being concrete features. Each feature can be enriched with feature-specific attributes (`FeatureSpecificAttribute` in Fig. 1) and operations (such as `FeatureSpecificOperation1` in Fig. 1). Notice that feature attributes are used to support the extended FM formalism.
- *Configuration*—declares an interface to determine if features are active or inactive, *i.e.*, whether their devised action should be executed when its `doit` method is called.
- *ConcreteConfiguration*—implements the `isActive` interface. It stores the activation status of features, checks the validity of a variant and preempts the execution of inactive features.
- *Variant* (Application)—creates feature actions by instantiating features, defines the variability points and the dependencies between feature actions.

Collaborations. Fig. 2 shows the sequence diagram of an exemplary variability-aware application implemented with the devise pattern. The participants are the same as in Fig. 1, with the addition of the `Variant` main application and a `PreMain`. The `main` method stored in the `Variant` is unaware of the current configuration

```
new Hello()
  .does(() -> System.out.println("Hello"))
  .implies(new World().does(() -> System.out.println(" World")))
  .doit();
```

Listing 1: Declaring constraints among feature actions.

which is set by the `PreMain`. In this example, `ConcreteFeature1` is inactive and `ConcreteFeature2` is active. Then, the `PreMain` launches the actual `Variant` `main`. The `Variant` declares two variability points, one for each of the two concrete features. In the case of `ConcreteFeature1`, the execution is devised and deferred to a later time whereas a feature action for `ConcreteFeature2` is devised and executed sequentially by calling the `does` and `doit` methods respectively. When the `doit` method is called, each feature action messages the `Configuration` to check if it is active. In this example, only `ConcreteFeature2` is active and therefore executed, whereas `ConcreteFeature1` is not executed. Notice that the `Variant` only has to devise the semantics of `ConcreteFeature1` and `ConcreteFeature2` whereas their execution or preemption is entirely handled by the `Configuration`.

Consequences. The devise pattern has the following benefits (+) and drawbacks (-).

+ It leads to an inverted control structure referred to as *the Hollywood principle* [38]: the `Configuration` handles the execution of the `Variant` and preempts the execution of inactive features and invalid configurations.
+ It makes the variability points of the application explicit: given a configuration, an active feature action could be replaced with its body without changing the semantics.
+ Feature actions are predictable and their body is a function that cannot cause side effects over out-of-scope variables.
+ Devising feature actions eliminates the need for conditional statements because alternative behaviors are selected based on the configuration; for instance, an alternative group is equivalent to a **`switch`** statement with a **`break`** on each case. For this reason, there is no need for binary flags in source code to control the execution flow.
+ Moves feature-specific attributes and methods from the classes' source code to the features' source code.
- Features increase the number of classes in an application: each feature is an additional class and the body of each feature action is translated into a class by the Java compiler.
- The code of the feature action's body is embedded in the main application, thus the resulting code may be hard to comprehend and analyze. As we will show later in this section, extracting a feature action requires additional abstractions.

Implementation and Sample Code. So far, we provided an overview of the essence of the devise pattern and its participants. However, several implementation-specific improvements can be made to vary the base interface described in Fig. 1. We identified and applied five possible improvements. The resulting interface of the improved implementation is shown in Fig. 3.

Abstractions to express feature dependencies. In our implementation, we chose to enrich the `Feature` class with one method for

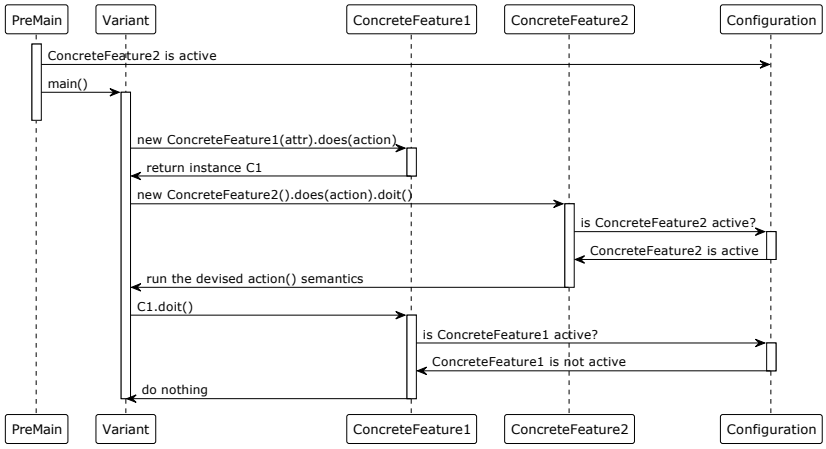


Figure 2: Sequence diagram of an exemplary variability-aware application devising two feature actions: one instance of `ConcreteFeature1` (which is inactive) and one of `ConcreteFeature2` (which is active). In both cases, the execution is deferred until the `doit` method is called. `Configuration` preempts the execution of inactive feature actions (`ConcreteFeature1` in this example).

```
1  void main() {
2    new Hello.does(() ->
3      System.out.println("Hello")
4    ).implies(
5      new WorldAction()
6    ).doit();
7  }
8  @Action
9  public class WorldAction extends World {
10   public WorldAction() {
11     this.does(() -> System.out.println(" World"))
12   }
13 }
```

Listing 2: Embedded and refactored feature actions.

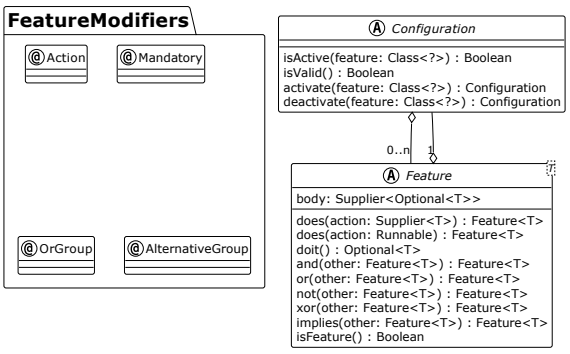


Figure 3: Extended class diagram of the devise pattern.

each of the most common Boolean relations, which are often used to declare cross-tree constraints in FMs: `and`, `or`, `not`, `implies`, `xor`. An example of usage of this API is shown in Listing 1. In this example, if the `Hello` feature is active, then the `World` feature must also be active. This constraint can be expressed by using the `implies` method. This is possible because the execution is deferred until all actions and their dependencies have been devised—*i.e.*, when the `doit` method is called (on line 5).

Refactor embedded code out of the main application. In most cases it is beneficial to decouple the declaration of the variability point and its implementation, otherwise the devise pattern acts identically to *`#ifdef`* macros. In our implementation, we chose to provide an `@Action` annotation and an `isFeature` method that returns **`false`** if the class is annotated, so that annotated classes are not considered as part of the FM and instead their activation status is determined based on their super-class. Consider refactoring Listing 1 so that the implementation of the `World` feature action is decoupled from the variability point declaration. The result of the refactoring is shown in Listing 2, in which the embedded feature action for the `World` feature was moved to the `WorldAction` annotated class. Thanks to this refactoring, the main method is unaware of the `World` feature implementation and the refactored `WorldAction` can also be reused in different parts of the application without code duplication. This was not possible in Listing 1.

Extended features parametrization. To implement extended features and their parameters, consider using configuration methods instead of constructors: non-default constructors must be overridden by child classes, causing unnecessary overhead for the domain analyst writing the model. In Listing 3, `Hello1` and `Hello2` are devised with the same semantics, but the second one does not require sub-classes to override the non-default constructor.

Non-void feature actions. Listing 1, 2 and 3 show **`void`** feature actions, implemented using the `Runnable` interface. A more flexible implementation may allow feature actions to return values. In our implementation, the body of a feature action returns an `Optional` type. Feature actions can be devised by providing either a `Supplier` (with return value) or a `Runnable` (without return value). In the latter case, executing the feature action will return an `Optional.empty()` value.

Related Patterns. A builder [13] can greatly benefit from using the devise pattern to configure the creation of complex object variants. The `doit` method of the `Feature` class is structured as a

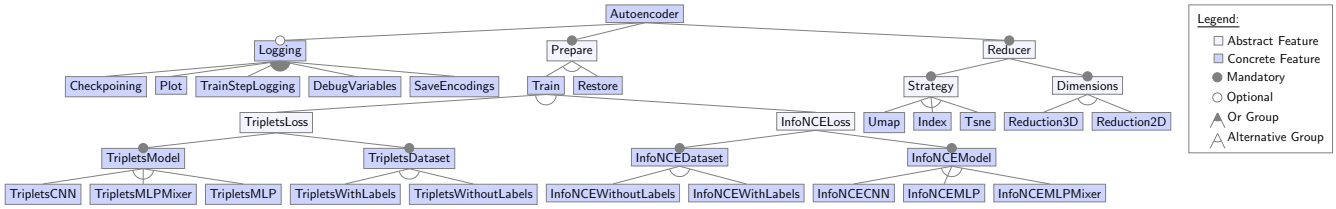


Figure 4: FM of the MNIST-encoder software family.

```
void main() {
  new Hello1(42).doit();
  new Hello2().config(42).doit();
}
public class Hello1 extends Feature {
  private int param;
  public Hello(int param) { //Must be overridden by subclasses!
    this.param = param;
    this.does(() ->
      System.out.println(param + " is the answer")
    );
  }
}
public class Hello2 extends Feature {
  private int param;
  public Hello2 config(int param) { //No need to override
    this.param = param;
    return this.does(() ->
      System.out.println(param + " is the answer")
    );
  }
}
```

Listing 3: Configuring extended features.

template method [13]. The separation between the feature abstractions and their implementation through a `Runnable` or a `Supplier` functional interface is akin to a bridge pattern [13]. The enforcement of the same configuration across all features and the main application can be achieved with a singleton object [13].

4 Case Study: MNIST-encoder

In this section, we will discuss three different implementations of the MNIST-encoder in which the variability is handled at source level without using external preprocessors: using JSON configuration files, the Variability Modules in Java (VMJ) [35] architectural pattern, and the devise pattern.

4.1 Application Overview

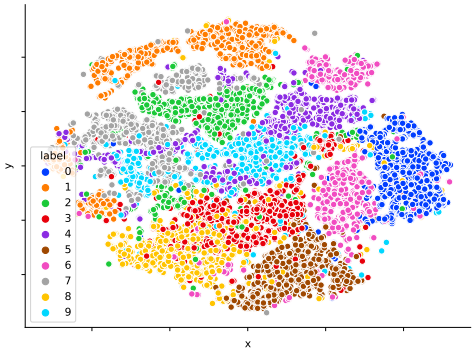
The design of deep learning applications often offers huge challenges in terms of variability. Several aspects of NNs, including architecture, training procedure and dataset, can be modeled differently to achieve different results. SPLE represents a valuable asset to model the variability of this kind of applications and to produce a family of related but different NNs. In this work, we embraced this approach to develop a family of MNIST-encoders (see Sect. 2.2). First, we analyzed the application domain and produced the FM presented in Fig. 4. The FM shows that the MNIST-encoder SPL has to deal with several variability concerns.

- *Logging*: tweak the output information that is provided to the user during training, including the value of debug variables, the loss and the model checkpoints; logging can also optionally plot the resulting encodings.
- *NN architecture*: we consider three kinds of architectures. The multi-layer perceptron (MLP) [14], the convolutional neural network (CNN) [14] and the MLP mixer [40]. All these NNs can be trained according to different loss functions. We consider only the Triplets [33] and the InfoNCE [26] loss functions.
- *Supervised* or *self-supervised* learning: depending on the approach, a different dataset has to be generated.
- *Dimensionality reduction techniques*: usually, hidden NN representations are high dimensional vectors. To visualize these vectors in 2D scatter plots, it is necessary to project them into a low-dimensional space. This can be achieved with various techniques such as t-SNE [21] and UMAP [23]. Instead, when the hidden representation is already low dimensional, it can directly be plotted without projection.

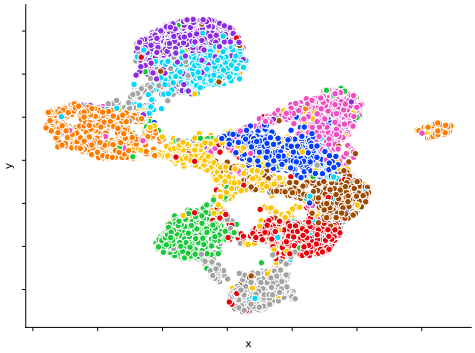
Overall, the FM contains 34 features, 15 of which are extended features that can be parameterized, allowing for additional customization options. In this work, we considered and evaluated 8 out of the total 55,296 valid configurations. Each variant was trained and used to produce the encodings of 10,000 data points from the MNIST dataset. We chose eight specific variants for a better comparison: we kept a shared base configuration and only changed a few features to better showcase the effect that each feature has on the results. The results are shown in Fig. 5. Each color represents a data point labeled with a different digit (from 0 to 9). All variants learned a meaningful representation: data points with the same label are generally clustered together. The first two rows are NN variants using the CNN architecture; the last two rows are NN variants using the MLP architecture. Odd rows use a dataset for supervised learning whereas even rows use a dataset for self-supervised learning. The left column shows NN variants in which dimensionality reduction is performed using t-SNE. On the right column those in which dimensionality reduction is performed using UMAP. Notice that the CNN variants show better clustering on average.

4.2 Variability-aware Encoders

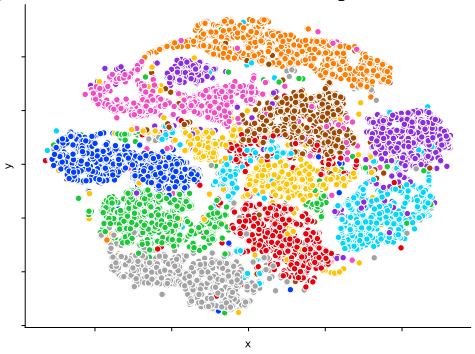
In this section, we overview the three approaches that we chose to turn the base MNIST-encoder implementation into a SPL that is aware of the variability concerns discussed earlier. Since we are focusing on approaches that do not require external tools, the configuration is performed manually by the developer in all these implementations. However, consider that the generation of configuration files can be automated with additional tooling.



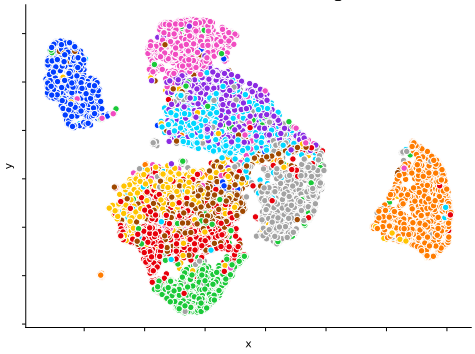
(a) Convolutional neural network, supervised, t-SNE.



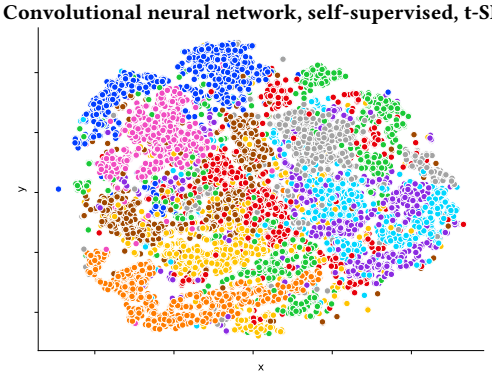
(b) Convolutional neural network, supervised, UMAP.



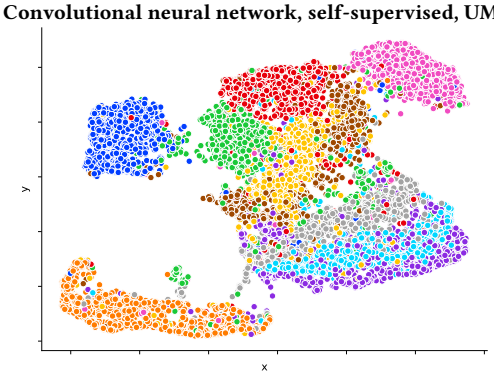
(c) Convolutional neural network, self-supervised, t-SNE.



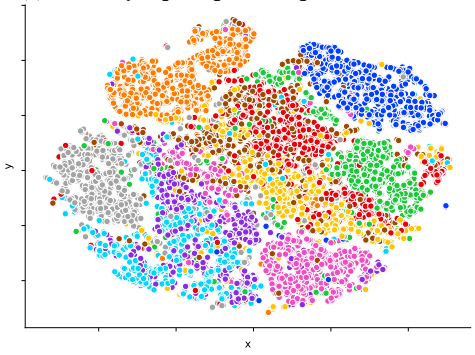
(d) Convolutional neural network, self-supervised, UMAP.



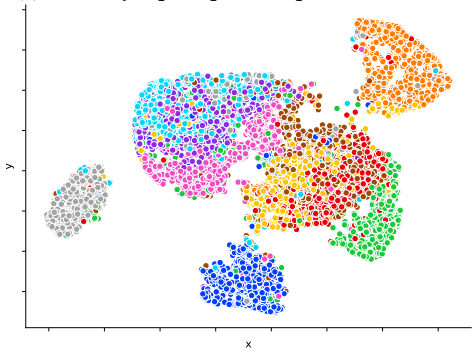
(e) Multi layer perceptron, supervised, t-SNE.



(f) Multi layer perceptron, supervised, UMAP.



(g) Multi layer perceptron, self-supervised, t-SNE.



(h) Multi layer perceptron, self-supervised, UMAP.

Figure 5: Embeddings obtained by eight variants of the MNIST-encoder using the InfoNCE loss. The legend reported in Fig. 5a maps each color to the label of the 10,000 data points.

```
SameDiff engine =
    new Restore<>().does(     // #ifdef RESTORE
        Model::load
    ).xor(new Train<>().does( // #elif defined(TRAIN)
        SameDiff::create
    )).doit();                // #endif
```

Listing 4: `Restore` and `Train` are alternative features.

JSON. The Javascript object notation (JSON) is commonly used for serialization and deserialization of objects; it is also used as a format for configuration files. In this version of the MNIST-encoder, the JSON configuration files are deserialized into factories [13]. The MNIST-encoder can be customized by editing one or more of the configuration files: a different JSON configuration will instantiate a different factory and eventually a different variant.

VMJ Pattern. VMJ [35] is an architectural pattern for the generation of SPLs. VMJ is based on the DOP paradigm in which features are expressed as deltas over a core module. Each delta is implemented as a decorator [13]. In VMJ a product is expressed using factories that instantiate a core module and all the required deltas depending on the configuration. Feature selection happens in a module declaration that lists all its requirements. Configurations are expressed as different main methods in which the core modules are configured by applying all the necessary deltas. Please refer to [35] for a complete overview.

Devise Pattern. The MNIST-encoder implementation based on the devise pattern follows the framework discussed in Sect. 3. Each feature in the FM from Fig. 4 is a Java class that directly or indirectly inherits from `Feature`. The effects that the activation of each feature has on a variant are expressed as feature actions—*i.e.*, instances of feature classes—whose semantics are devised by passing executable code to the `does` method. For instance, Listing 4 shows two features: `Restore` and `Train`, which are part of an alternative group—only one of them can be active at the same time. If both are active or both are inactive, the configuration is considered invalid. This is expressed at source level using the `xor` method. When `Train` is active the differentiation engine is created as a clear instance. Instead, when `Restore` is active the differentiation engine is instantiated by loading a previously saved model. In this case, the semantics are devised using Java method references. Using the devise pattern, creating a new configuration should be as effortless and reusable as possible. The MNIST-encoder uses a `BaseConfiguration` class as a template for all eight aforementioned NN variants and leverages inheritance to minimize the required changes. For instance, Listing 5 shows a `DerivedConfiguration` which is obtained by activating seven additional features over the `BaseConfiguration`. The effort of creating a new configuration is minimized by sticking to a declarative approach in which active and inactive features are simply listed with no mention of the control flow of the application. This implementation is based on the extended version of the devise pattern interface shown in Fig. 3, supporting all the modeling techniques provided by mainstream feature modeling tools:

- cross-tree constraints are expressed using Boolean operators over feature actions (`and`, `or`, `not`, `xor` and `implies` methods);

```
public class DerivedConfiguration extends BaseConfiguration {
    public DerivedConfiguration() {
        super();
        this.activate(
            InfoNCELoss.class,
            InfoNCEDataset.class,
            InfoNCEModel.class,
            Reduction2D.class,
            Tsne.class,
            InfoNCEMLP.class,
            InfoNCESupervised.class);
    }
}
```

Listing 5: Creating a `DerivedConfiguration` is eased by extending the `BaseConfiguration` class.

- alternative (`xor`) groups are expressed by an `@AlternativeGroup` annotation as a feature class modifier;
- or groups are expressed by an `@OrGroup` annotation as a feature class modifier;
- mandatory features are expressed by a `@Mandatory` annotation as a feature class modifier.

The `BaseConfiguration` class collects all this information with regards to each feature class in the feature hierarchy and evaluates its validity before running the main application. The execution is preempted if the configuration is invalid with respect to the FM. The full implementation of the devise pattern and its application is available at Zenodo:

`https://doi.org/10.5281/zenodo.6624848`

Summary. In JSON the variability is handled using configuration files and factories. In VMJ the variability is handled by applying different decorators over the base component class. In the devise pattern the variability is handled by declaring variability points and devising feature actions and managed by a configuration class.

4.3 Comparison: Non-Functional Properties

The semantics of each of the eight considered variants of the MNIST-encoder do not change depending on the mechanism used to express the variability: JSON, VMJ or devise pattern. However, the three approaches are substantially different with regards to their non-functional properties. In this evaluation, we identified 13 non-functional properties supported by at least one of the approaches. Then, we classified each non-functional property into one of four categories. The *feature dependencies* category collects all properties dealing with the expressiveness with regards to the base FM formalism: alternative groups, or groups, mandatory features and cross-tree constraints. The *implementation extension* category collects the properties dealing with the capability of changing the behavior of an existing class [35]: adding and removing fields and methods. The *Other FM formalisms* category collects properties dealing with the expressiveness with regards to variants of the base FM formalism: the extended FM formalism and the multi-dimensional FM. The *Quality of life* category collects all other properties that can improve the usability of the approach by providing support to the verification and maintenance of SPLs: static checking capabilities, configuration inheritance, traceability of feature location, separation between modeling code and implementation code, and

	Property	JSON	VMJ	Devise
Feature dependencies	Alternative groups	○	○	●
	Or groups	○	○	●
	Mandatory features	○	●	●
	Cross-tree constraints	○	○	◑
Implementation extension	Add fields & methods	○	●	◑
	Remove fields & methods	○	●	○
FM formalisms	Extended features	●	●	●
	Multi-SPL	◑	●	◑
Quality of life	Statically checked	○	◑	●
	Configuration inheritance	○	◑	●
	Traceability support	○	●	●
	Model and implementation independence	○	○	●
	Intercompatibility	●	●	●

Table 1: Support to VM modeling in different approaches. ○: not supported, ◑: partially supported, ●: fully supported.

compatibility with other approaches. This section discusses each of the 13 properties. Table 1 summarizes this discussion.

Feature Dependencies. In most common variability modeling frameworks, such as FeatureIDE, the FM formalism can be properly expressed by enriching features with additional information—*i.e.*, if features are either *optional* or *mandatory* and if siblings are part of an *alternative group* or an *or group*. JSON cannot express any of these feature dependencies. VMJ can properly support *mandatory* features by combining module requirements and well-designed factories. However, to the best of our knowledge, it is not possible to declare different deltas as part of an *or group* or an *alternative group*, because each delta is modeled as a decorator over the same base component class. In our implementation of the devise pattern, *mandatory* features, *alternative groups* and *or groups* are expressed as simple annotations and checked by the configuration abstraction. Instead, the devise pattern supports cross-tree constraints only partially, because feature dependencies are expressed at source level and evaluated when the feature action is instantiated; therefore any invalidity with regards to cross-tree constraints is captured, but only at runtime. Statically detecting cross-tree constraints would require an external control flow analysis tool.

Extension of a Base Implementation. The VMJ approach natively supports addition and removal of both fields and methods. Addition is simply performed by decorators. Removal of methods is done by throwing a runtime exception in the overridden method. Similarly, removal of fields is done by overriding their getters and setters. Notice that this is possible only if the removed fields are private and never accessed through reflection. The devise pattern can only emulate the addition of fields and methods through aggregation and does not support fields and methods removal. JSON does not support any of the above.

Other FM Formalisms. All three approaches fully or partially support the variants of the base FM formalism: extended feature model and multi-product lines. The parameters of extended features are implemented as fields but are handled differently depending on the approach: in JSON, the fields are simply added to the factories and are deserialized when the configuration is loaded; in VMJ, the parameters are added as fields of the decorator classes and then set by the factory methods called by the main; in the devise pattern, parameters are static fields of the classes from the feature hierarchy. Multi-product lines can be partially achieved in JSON by nesting configurations and in the devise pattern by using multiple configuration classes in the same product which are combined through aggregation; only VMJ directly addresses the problem of multi-product lines and it is designed to fully support the formalism.

Modeling effort	JSON	VMJ	Devise
Variability	597	1528	1066
Configuration (1 configuration)	43	112	109
Configuration (8 configurations)	344	896	293

Table 2: Modeling effort in terms of LoC required to turn a core implementation into a variability-aware one using different approaches. Also the effort to create the first configuration and all the eight configurations from Fig. 5.

Quality of Life. Finally, we consider the aspect of quality of life for variability modeling. This includes the capability of statically checking SPLs and their configurations, the support to the extension of existing configurations, the traceability of feature implementations and the compatibility with other approaches. In the devise pattern all elements of variability are statically checked: *mandatory* features, *or* and *alternative groups*; all product variants coexist and are checked by the compiler, including any return types of the feature actions. VMJ can only check the validity of *mandatory* features using modules. However, decorators over the base module are applied using reflection that can fail at runtime if types mismatch. Similarly, access to removed fields and methods raise runtime exceptions that can cause failures if not properly handled. Both the devise pattern and VMJ support the extension of existing configurations. However, this can be achieved natively with the devise pattern using the `Configuration` class, whereas changing a configuration in VMJ requires refactoring of the existing product. Most notably, removing a delta may not be feasible depending on the ability to remove decorators from a component. Finally, in both VMJ and the devise pattern the feature traceability issue is trivialized by mapping features and their implementations to language-specific abstractions which usages can be easily traced by any commonly used IDE. However, only the devise pattern can properly separate the model from its implementation, whereas in VMJ the deltas are expressed in the decorator classes together with their implementation. None of the above quality of life improvements are directly supported in JSON. However, it should be noted that the three approaches to variability modeling are not mutually incompatible and can be combined at will depending on the scenario to stem and complement their issues. We argue that this point represents the main advantage of using an in-language approach to variability modeling over external preprocessors: software artifacts that can be handled by a specific preprocessor are usually incompatible other preprocessors, therefore migration between different preprocessors can be hard or unfeasible.

4.4 Comparison: Modeling Effort

For each of the three implementations of the MNIST-encoder, we analyzed the effort—in terms of lines of code (LoC)—required to

model the variability and generate new products. All implementations depend on a variability-unaware core application of 2236 LoC. Table 2 reports the results of our analysis.

- Modeling the variability in JSON required the factory classes discussed earlier, for a total of 597 LoC; then each configuration can be written in JSON for a total of 43 LoC for each configuration. Deploying the eight configurations from Fig. 5 costs 344 LoC.
- Modeling the variability in VMJ required adding the component and decorator classes, as well as the same factories used by JSON, for a total of 1528 LoC; then each configuration can be written in a Java main class for a total of 112 LoC for each configuration. Deploying the eight configurations shown in Fig. 5 costs 896 LoC.
- Modeling the variability using the devise pattern required adding one class for each feature of the FM and a variability-aware main class, for a total of 1066 LoC; writing the first configuration requires creating the `BaseConfiguration` abstract class (86 LoC) and a concrete configuration subclass (23 LoC), for a total of 109 LoC. Deploying the remaining 7 configurations shown in Fig. 5 costs 23 additional LoC for each configuration: the total configuration effort is 293 LoC.

The modeling effort among the three approaches is fairly similar: JSON has a slight advantage in terms of both variability modeling and configuration modeling, but it should be noted that the JSON approach does not express the FM formalism and its constraints. Conversely, VMJ and the devise pattern face an initial overhead to introduce the variability with the advantage of expressing feature constraints. Between the two, each feature written in the devise pattern takes slightly less LoC than the respective feature in VMJ, for a total of 1066 LoC vs 1528 LoC. Finally, configuration inheritance in the devise pattern introduces an initial overhead on the first configuration but then the first configuration can be reused to model subsequent configurations, reducing the configuration effort substantially in the long run.

4.5 Threats to Validity

The validity of our results may be threatened by our lack of expertise with the VMJ architectural pattern; our implementation was not reviewed by the original authors [35] and we may have applied the pattern incorrectly, which may lead to different results in Table 2. To stem this issue we applied the same black-box approach to the usage of the devise pattern: the implementation of the devise pattern library and its usage for the development of the MNIST-encoder were performed separately by different contributors of this work. To the best of our knowledge the three variability-aware implementations of the MNIST-encoder should be semantically equivalent, but this is hard to properly verify due to the random nature of the learning process in DeepLearning4J[2]—different runs may result in different NNs. In this regard, we separated the core library that implements most of the functionalities used by each implementation, so that each approach is only concerned with the variability modeling aspect. Our implementation of the devise pattern is written in Java and may not be applicable to other languages. However, the pattern should at least be applicable to any object-oriented language, as we tested by developing minimal implementations in Scala and Kotlin[3]. We do not compare against composers and preprocessors from the literature—they may provide better abstractions for variability modeling. As discussed in Sect. 1, we argue that a direct comparison is not applicable because the two approaches to variability modeling tackle different problems.

5 Related Work

SPLs are very popular and a variety of approaches to support their definition have been proposed by researchers [3, 18, 25]. However, all these approaches are based on preprocessors and, to the best of our knowledge, there has been little research on design patterns in the context of modeling and implementation of SPLs besides our work. The most similar work is the VMJ architectural pattern based on variability modules and DOP [35] that we discussed in Sect. 4 and in which each feature is implemented through decorators [13] over the base implementation. Seidl *et al.* [34] presented a generative SPL development method using variability-aware versions of the observer, strategy, template method and composite [13] patterns and introduced the Family Role Model as a notation to capture constraints on the variable application. Shatnawi and Cunningham [36] addressed the difficulty of specifying and maintaining feature models due to the SPLE tools requiring specific knowledge and skills and they proposed to encode FMs using JSON. Their contribution share with our the choice of using mainstream technologies to develop SPLs. On a similar note, Chimalakonda and Lee [9] 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, since artifacts developed with one suite are not compatible nor reusable with other ones.

6 Conclusions

In this paper, we introduced the devise pattern as a novel technique for modeling and implementing SPLs. It can express all aspects of the extended FM formalism using tools that are familiar to software developers and with a syntax similar to the `#ifdef` macros in C. Feature actions can also be refactored using dedicated abstractions to avoid the `#ifdef` hell problem and code duplication. The devise pattern prevents the need for feature location techniques because variability points are explicitly declared in the application and can be used to complement variability mining techniques. Our contribution is a description of the pattern following Gamma *et al*'s template. We demonstrated its applicability on the development of a variability-aware MNIST-encoder application. Finally, we compared this application to other two variability-aware alternatives implemented using JSON configurations and VMJ decorators respectively. Our contribution resulted more expressive and can model all aspects of the FM formalism; the validity of the configurations can be checked statically and deploying multiple configurations is eased. We believe that it can ease the adoption of SPLs in the industry and research by reducing the barrier to enter associated with the complexity of dedicated tools and environments.

[2] `https://deeplearning4j.konduit.ai/`

[3] The Scala and Kotlin implementations are not discussed due to space constraints.

Acknowledgments

This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

References

[1] Hadil Abukwaik, Andreas Burger, Berima Kweku Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, Foutse Khomh and David Lo (Eds.). IEEE, Madrid, Spain, 529–533.

[2] Sven Apel, Christian Kästner, and Christian Lengauer. 2009. Language-Independent, Automated Software Composition. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE, Vancouver, BC, Canada, 221–231.

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 6 (June 2004), 355–371.

[4] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2021. Efficient Static Analysis and Verification of Featured Transition Systems. *Empirical Software Engineering* 27, 10 (Oct. 2021).

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (Sept. 2010), 615–636.

[6] Gilad Bracha and William Cook. 1990. Mixin-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP'90)*, Akinori Yonezawa (Ed.). ACM, Ottawa, Canada, 303–311.

[7] Walter Cazzola and Luca Favalli. 2022. Towards a Recipe for Language Decomposition: Quality Assessment of Language Product Lines. *Empirical Software Engineering* 27 (April 2022). https://doi.org/10.1145/3514232

[8] Ting Chen, Simon Kornblith, and Mohammed Norouzi. 2020. A Simple Framework for Contrastive Learning of Visual Representations. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*, Emtiyaz Daumé III and Po-ling Loh (Eds.). PMLR, Vienna, Austria, 1597–1607.

[9] Sridhar Chimalakonda and Lee Dan Hyung. 2016. On the Evolution of Software and Systems Product Line Standards. *ACM SIGSOFT Software Engineering Notes* 41, 3 (May 2016), 27–30.

[10] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques Using ArgoUML-SPL. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'19)*, Guilles Perrouin and Danny Weyns (Eds.). ACM, Leuven, Belgium, 16:1–16:10.

[11] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denis Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process* 25, 1 (Jan. 2013), 53–95.

[12] David Flanagan. 2005. *Java in a Nutshell*. O'Reilly Media.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Ma, USA.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[15] Iris Groher and Markus Völter. 2009. Aspect-Oriented Model-Driven Software Product Line Engineering. *Transactions on Aspect-Oriented Software Development* 4 (2009), 111–152.

[16] José Miguel Horcas Aguilera, Mónica Pinto, and Lidia Fuentes. 2019. Software Product Line Engineering: A Practical Experience. In *Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19)*, Laurence Duchien and Thomas Thüm (Eds.). ACM, Paris, France, 164–176.

[17] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[18] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Damiani Ferruccio. 2014. DeltaJ 1.5: Delta-Oriented Programming for Java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools (PPJ'14)*, Bruce Childers (Ed.). ACM, Cracow, Poland, 63–74.

[19] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE* 86, 11 (Nov. 1998), 2278–2324.

[20] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, Jeff Kramer, Judith Bishop, Prem Devanbu, and Sebastian Uchitel (Eds.). IEEE, Cape Town, South Africa, 105–114.

[21] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data Using t-SNE. *Journal of Machine Learning Research* 9, 86 (Nov. 2008), 2579–2605.

[22] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC'18)*, Thorsten Berger and Paulo Borba (Eds.). ACM, Gothenburg, Sweden, 257–263.

[23] Leland McInnes, John Healy, and James Melville. 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv e-prints* arXiv:1802.03426 (Feb. 2018).

[24] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[25] Mira Mezini and Klaus Ostermann. 2004. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the 12th international Symposium on Foundations of Software Engineering (FSE'04)*, Matthew B. Dwyer (Ed.). ACM, New Port Beach, CA, USA, 127–136.

[26] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation Learning with Contrastive Predictive Coding. *arXiv e-prints* arXiv:1807.03748 (July 2018), 1–13.

[27] Klaus Pohl, Klaus Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[28] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97) (Lecture Notes in Computer Science 1241)*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer, Helsinki, Finland, 419–443.

[29] Christian Prehofer. 2001. Feature-Oriented Programming: A New Way of Object Composition. *Concurency and Computation: Practice and Experience* 13, 6 (May 2001), 465–501.

[30] Abdul Razzaq, Asanka Wasala, Chris Exton, and Jim Buckley. 2019. The State of Empirical Evaluation in Static Feature Location. *Transaction on Software Engineering and Methodology* 28, 1 (Jan. 2019), 1–58.

[31] Dan Saffer. 2010. *Designing for Interaction: Creating Innovative Applications and Devices*. New Riders.

[32] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Proceedings of the 14th International Software Product Line Conference (SPLC'10) (Lecture Notes on Computer Science 6287)*, Jan Bosch and Jaejoon Lee (Eds.). Springer, Jeju Island, South Korea, 77–91.

[33] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A Unified Embedding for Face Recognition and Clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*, David Forsyth, Ivan Laptev, Deva Ramanan, and Aude Oliva (Eds.). IEEE, Salt Lake City, UT, USA, 815–823.

[34] Christoph Seidl, Sven Schuster, and Ina Schaefer. 2017. Generative Software Product Line Development Using Variability-Aware Design Patterns. *Computer Languages, Systems and Structures* 48 (June 2017), 89–111.

[35] Maya Retno Ayu Setyautami and Reiner Hähnle. 2021. An Architectural Pattern to Realize Multi Software Product Lines in Java. In *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21)*, Paul Grünbacher, Christoph Seidl, Deepak Dhungana, and Helena Łovasz-Bukvova (Eds.). ACM, Krems, Austria, 9:1–9:9.

[36] Hazim Shatnawi and H. Conrad Cunningham. 2021. Encoding Feature Models Using Mainstream JSON Technologies. In *Proceedings of the 2021 ACM Southeast Conferance (ACM-SE'21)*, Eric Gamess (Ed.). ACM, USA, 146–153.

[37] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability Model of the Linux Kernel. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, David Benavides, Don S. Batory, and Paul Grünbacher (Eds.). ACM, Linz, Austria, 45–51.

[38] Richard E. Sweet. 1985. The Mesa Programming Environment. *ACM Sigplan Notice* 20, 7 (July 1985), 216–229.

[39] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79, 1 (Jan. 2014), 70–85.

[40] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Dosovitskiy. 2021. MLP-Mixer: An all-MLP Architecture for Vision. In *Processing of the 35th Conference on Neural Information Processing Systems (NeurIPS'21)*, Marc'Aurelio Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Lian, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., Virtual, 24261–24272.