Tracking Feature Usage: Incorporating Implicit Feedback into Software Product Lines

Abstract

Implicit feedback is collecting information about software usage to understand how and when the software is used. This research tackles implicit feedback in Software Product Lines (SPLs). The need for platform-centric feedback makes SPL feedback depart from one-off-application feedback in both the artefact to be tracked (the platform vs the variant) as well as the tracking approach (indirect coding vs direct coding). Traditionally, product feedback is achieved by embedding 'usage trackers' into the software's code. Yet, products are now members of the SPL portfolio, and hence, this approach conflicts with one of the main SPL tenants: reducing, if not eliminating, coding directly into the variant's code. Thus, we advocate for Product Derivation to be subject to a second transformation that precedes the construction of the variant based on the configuration model. This approach is tested through *FEACKER*, an extension to *pure::variants*. We resorted to a TAM evaluation on *pure-systems GmbH* employees(n=8). Observed divergences were next tackled through a focus group (n=3). The results reveal agreement in the interest in conducting feedback analysis at the platform level (perceived usefulness) while regarding FEACKER as a seamless extension to *pure::variants*' gestures (perceived ease of use).

Keywords: Software Product Lines, Implicit Feedback, Continuous Development

1. Introduction

Continuous deployment is considered an extension of Agile practices whereby the user feedback is extended beyond development by frequently releasing and deploying software to customers and continuously learning from their usage (Dakkak et al., 2021a). Feedback can be obtained either directly, e.g., through reviews (i.e., explicit feedback) or through tracking user interactions (i.e., implicit feedback). We focus on implicit feedback, i.e., the automatically collected information about software usage from which user behavior and preferences can be inferred (van Oordt and Guzman, 2021).

In one-off development, implicit feedback (hereafter just 'feedback') benefits developers and analysts alike. Developers can benefit from a more timely awareness of bugs, bad smells, or usability issues (Johanssen et al., 2019) while being

motivated by their software's real value to real users (van Oordt and Guzman, 2021). Regarding the analysts, they resort to feedback for requirement prioritization (Wang et al., 2019; Johanssen et al., 2019) and requirement elicitation (Liang et al., 2015). No wonder feedback is catching on among one-off developers (Fitzgerald and Stol, 2017). Unfortunately, feedback has not received the same attention for variant-rich systems in general and Software Product Lines (SPLs) in particular (Dakkak et al., 2021a).

Software Product Line Engineering (SPLE) aims to support the development of a whole family of software products (aka variants) through systematic reuse of shared assets (aka the SPL platform) (Clements and Northrop, 2002). To this end, SPLE distinguishes between two interrelated processes: Domain Engineering (DE), which builds the SPL platform, and Application Engineering (AE), which derives individual products from the SPL platform. In this setting, experiences with feedback are very rare SPL.

Traditionally, feedback is conducted at the application level (van Oordt and Guzman, 2021). Yet, a hallmark of SPLE is to move development efforts as much as possible from AE to DE. Feedback should be no exception. Thus, we advocate for feedback practices to be part of DE and, as such, to be moved upwards from products to the SPL platform. This leads to platform-based feedback rather than product-based feedback. Unfortunately, the limited support for continuous deployment among variability managers (e.g., *pure::variants* (Beuche, 2019), *GEARS* (Krueger and Clements, 2018)) hinders this issue from being grounded on empirical evidence. On these premises, we tackle two research questions:

- How could 'platform-based feedback' be incorporated into SPLE practices?
- How would variability managers be enhanced to support 'platform-based feedback'?

In addressing these questions, we contribute by

- elaborating on general requirements for leveraging variability managers with feedback services (Section 4);
- providing proof-of-concept of how these requirements can be realized for *pure::variants* as the variability manager and *Google Analytics (GA)* as the usage tracker (Section 5);
- providing proof-of-value through a hybrid evaluation: a Technology Acceptance Model (TAM) evaluation is first conducted among employees of *pure-systems GmbH* (n=8), followed by a focus group for the most divergent TAM questions (n=3) (Section 6).

The research is conducted as an *Active Design Research (ADR)* practice (Sein et al., 2011). ADR aims to solve or at least explain the issues of a problematic situation (i.e., implicit feedback not being conducted at the SPL platform level) by favoring a bottom-up approach where researchers join practitioners

in their local problems/solutions, which are next argumentatively generalized. This bottom-up approach fits our research setting where, to the best of our knowledge, no experiences, let alone theories, have been reported on implicit feedback in SPLs (Dakkak et al., 2021a). We start with a brief about implicit feedback in one-off development.

2. Background: Implicit Feedback

The industry is embracing agile and continuous methods, which inherently includes customers' feedback in the analysis loop of the development process (Schön et al., 2017; Johanssen et al., 2019). Specifically, the literature distinguishes two customer feedback types: explicit and implicit. The first refers to what users directly say or write about a software (e.g., tweets, reviews written in an app store, etc.) (van Oordt and Guzman, 2021; Johanssen et al., 2019). By contrast, implicit feedback automatically collects data about the software usage or execution from which user preferences and trends can be inferred (Maalej et al., 2009; van Oordt and Guzman, 2021). In short, explicit feedback is how users see the software, while implicit feedback depicts how they actually use it.

Uses of implicit feedback in one-off development include:

- *Requirement Prioritization.* Based on actual feature usage, managers prioritize which requirement implementation or maintenance should be performed first (Johanssen et al., 2019; Hoffmann et al., 2020; van Oordt and Guzman, 2021; Johanssen et al., 2018).
- *Requirement Elicitation.* Discovery of new requirements by analyzing the different usage paths of the users (Hoffmann et al., 2020; Oriol et al., 2018)
- *Optimization.* Improvement of different quality indicators (Martínez-Fernández et al., 2019; Johanssen et al., 2019) or optimization of functionalities to be easier and faster for the users to perform them (van Oordt and Guzman, 2021).
- *Bug Awareness.* Early identification and fixing of bugs by capturing them through crash reports and raised errors in the logs (van Oordt and Guzman, 2021; Johanssen et al., 2019; Olsson and Bosch, 2013).
- *User Understanding.* Understanding how users interact with the software (e.g., analyzing how they adapt to new functionalities)(van Oordt and Guzman, 2021; Johanssen et al., 2019).

SPLs certainly need to prioritize requirements, optimize code, or understand users. Yet, the fundamental difference rests on the level at which the decision-making process occurs: the platform level rather than the product level. In particular, in SPLE the prioritization of requirements or the optimization of code is often conducted at the platform level rather than based on individual products. Though application engineers can provide information about specific

products, such as usage or bugs, the ultimate decision of which code to optimize or which requirement to prioritize often lies with the Domain Engineering board. Therefore, we advocate for implicit feedback to be harmonized with SPLE practices and hence, be moved to the platform level.

3. Practical case: Implicit Feedback at WACline

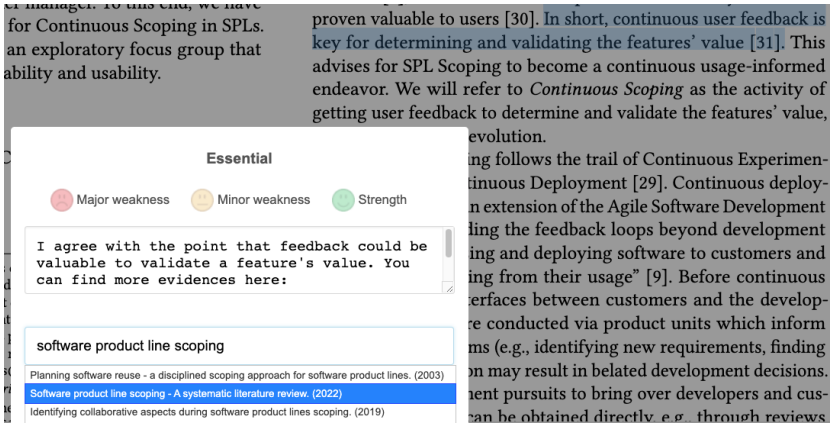

Figure 1: The *Commenting* feature. Besides adding notes, reviewers can include bibliographical references from Google Scholar.

This section introduces the case that triggered this research: *WACline*, an SPL in the Web Annotation domain (Medina et al., 2022). With 85 distinct features, *WACline* supports the creation of custom browser extensions for Web annotation with a current portfolio of eight variants. WACline engineers need to track feature usage. The discomfort of supporting implicit feedback by directly acting upon the variant's code causes need of a more fitting approach.

3.1. The running example

As the running example, Fig. 1 shows the case of the *Commenting* feature. This feature permits adding a comment to a highlighted text on the web. In addition, it allows reviewers to query Google Scholar directly without opening a new browser tab.

Implementation-wise, WACline follows an annotation-based approach to SPLs. This implies that variations are supported through pre-compilation directives that state when a block code is to be included in the final product based on the presence or absence of a feature selection at configuration time. Using *pure::variants* as the event manager, WACline variations are denoted through pre-compilation directives that start with an opening directive *//PVSCL:IFCOND* and end with a closing directive *//PVSCL:ENDCOND*. Fig. 2 (left) provides an example with the *//PVSCL:IFCOND* directives (aka *#ifdef* block) framing part of the codebases of *Commenting* and *Replying* features.

```
openForm (annotation) {
  Form.showForm(annotation, (err, annotation) => {
    // PVSCL:IFCOND(Commenting, LINE)
    if (err) {
      Alerts.errorAlert({ text: 'Unexpected error'})
    } else {
      LanguageUtils.dispatchCustomEvent(
        Events.createComment, {
         annotation: annotation
        })
    }
   // PVSCL:ENDCOND
   // PVSCL:IFCOND(Replying, LINE)
     LanguageUtils.dispatchCustomEvent(
       Events.createReplying, {
         annotation: annotation
       })
  // PVSCL:ENDCOND
  })
}
```

```
openForm (annotation) {
  Form.showForm(annotation, (err, annotation) => {
    if (err) {
      Alerts.errorAlert({ text: 'Unexpected error'})
    } else {
      LanguageUtils.dispatchCustomEvent(
        Events.createComment, {
         annotation: annotation
        })
      ga('send',{
        hitType: 'event',
        eventCategory: 'Commenting',
        eventAction:'commentingEnacted',
        variant:'WACproduct'
      })
    }
  })
}
```

Figure 2: **Platform** code (with pre-compilation directive) vs Feedback-minded **variant** code (i.e., each execution of this code populates the feedback log with an event occurrence described along with the 'eventCategory' (e.g., *Commenting*), the 'eventAction' (e.g., *CommentingEnactment*) and the 'variant' (i.e., the product which raises the event).

WACline delivers Web products. Web applications are one of the earlier adopters of implicit feedback. This is achieved with the help of web analytic services such as GA (W3Techs, 2022). GA tracks and reports website usage. It provides insights into how visitors interact with a website, including the number of visitors, their behavior, and their demographics. Operationally, interaction tracking is achieved by inlaying tracking instructions (known as 'hits' in the GA parlance). Fig. 2 (right) displays an example of *Commenting* once incorporated into a *WACproduct* (notice that the *#ifdef* block is no longer there). *WACproduct* is now 'feedback-minded', i.e., each execution of this code populates the feedback log with an event occurrence. Those hits run in the web browser of each client, collecting data and sending it to the feedback log. Eventually, this feedback log is consulted through appropriate dashboards (Google, 2016).

However, *WACproduct* is not just a standalone product, but a variant derived from the WACline platform. Realizing implicit feedback in the conventional way (i.e., at the variant level) causes the 'gut feeling' that a fitting approach was needed.

3.2. The 'gut feeling'

While providing tracking code at the variant level, WACline engineers were aware of eventual risks in terms of delays, partially, and chances of 'merge hells' in the Git repository. The following paragraphs delve into these issues.

Risk of delays. Continuous deployment brings developers and customers closer, reducing the latency in development decisions often associated with the traditional model where product units serve as the interface between customers and development teams. SPLE introduces an additional layer of indirection

between users and code developers. As code developers, domain engineers might not have direct access to the users through the application engineers. This additional layer can lead to further delay in receiving user feedback, slowing down the decision-making process.

Risk of partiality. Eager to promptly content their customers, application engineers might be tempted to conduct feedback analysis locally (i.e., usage tracking limited to their products), favoring changes to be conducted at the product level, overlooking the implications for the SPL platform as a whole.

Risk of 'merge hell'. In SPLE, changes made to the product branches tend to be merged back into the platform branch. If GA hits are added at the application level, then each AE team will inject them for their purposes. However, this way of conducting feedback analysis increases the likelihood of encountering merge conflicts while merging variant branches back into the platform's Master branch.

Alternatively, WACline engineers attempted to define 'feedback' as a feature, but it did not work either. Usage feedback is not a "prominent or distinctive user-visible aspect of a variant" but a measurement for decision-making that might involve tracking distinct single features or feature combinations (e.g., Is F1 and F2 used together?). Feedback directives can be considered crosscuts on the SPL's Feature Model. This situation highlighted two significant inconsistencies when conducting feedback at the variant level:

- a conceptual mismatch (tracking subject). SPLs reason in terms of *features*, while GA (and other analytic services) consider products the unit of tracking,
- a process mismatch (scope). SPLs set the analysis for a collection of products (i.e., the SPL portfolio), while GA is thought to track a single product.

These observations prompted the endeavor to integrate implicit feedback into existing variability management systems, ensuring a platform-wide coverage of feedback.

4. Requirements for feedback-minded variability managers

Design principles (aka general requirements) provide guidelines for designing effective solutions that can solve real-world problems and meet the needs of the stakeholders (Gregor et al., 2020). Therefore, design principles are very much framed by the problems and stakeholders where the principles were born. This section introduces design principles derived from our experience handling implicit feedback in WACline.

4.1. General Requirements

This subsection identifies feedback tasks to be considered during the SPLE process, and hence, they are amenable to being accounted for by variability managers. Fig. 3 depicts the traditional SPLE process (Apel et al., 2013) enriched with feedback tasks. Specifically:

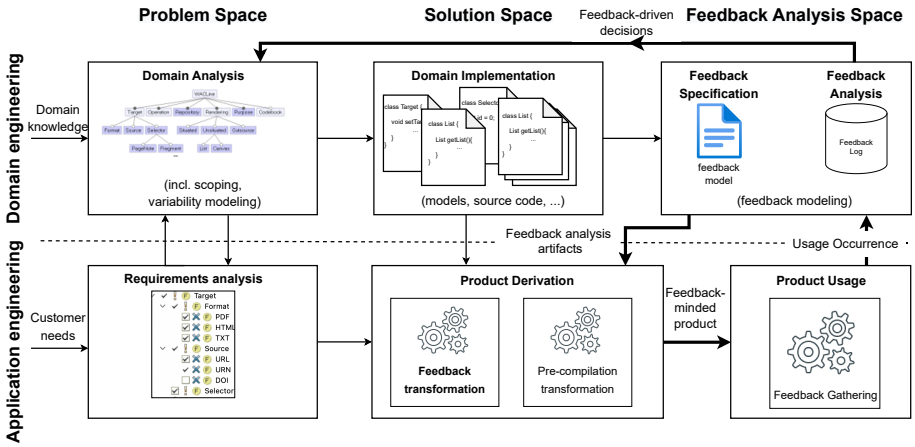

Figure 3: Leveraging SPLE with Feedback Analysis. The bold line denotes the feedback workflow: analysis-specification-coding-dataGathering-analysis.

- Domain Engineering now includes *Feedback Specification* (i.e., the description of the data to be collected in the Feedback log) and *Feedback Analysis* (i.e., the study of the data in the Feedback log),
- Application Engineering now incorporates *Feedback Transformation* (i.e., inlaying feedback tracking instructions during variant derivation) and *Feedback Gathering* (i.e., collecting implicit feedback during variant execution).

The following subsections delve into these tasks.

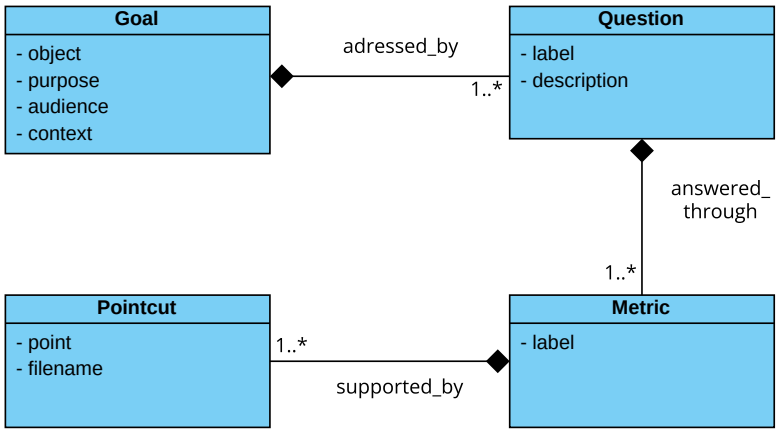

Figure 4: The Feedback Model.

4.1.1. Feedback Specification

Like in one-off development, feature usage can become a main informant for different decision-making issues throughout the lifetime of the SPL platform. The Goal-Question-Metric (GQM) approach is a popular framework for driving goal-oriented measures throughout a software organization (Basili and Rombach, 1988). By defining specific goals and identifying key performance indicators, the GQM approach provides a structured way for decision-making. We frame Feedback Specification as a GQM practice with metrics related to feature usage. Fig. 4 depicts our (meta)Model.

Basili and Rombach (1988) introduce the following template for goal description:

> Analyze *¡object of study¿* with the purpose of *¡purpose¿* with respect to *¡quality focus¿* from the point of view of the *¡perspective¿* in the context of *¡context¿*

In one-off development, the 'object of study' tends to be a single product. However, SPL heavily rests on features as a significant construct throughout the SPLE process. Accordingly, it is natural to introduce features as the 'object of study' of feedback analysis. After all, chances are that agendas, organizational units or release schedules are set regarding features (Hellebrand et al., 2017; Wnuk et al., 2009). Yet, for some analysis scenarios, features might turn too coarse-grained, and analysis might need to be conducted regarding particular feature combinations (e.g., as described in the pre-compilation directives that conform the *#ifdef* blocks). Hence, the codebase to be analyzed may be denoted by a single feature or a combination of features. This will identify 'the slice of the SPL codebase' that stands for the object of study.

Next, this code slice is to be analyzed in the pursuit of distinct purposes (e.g., refactoring, testing, etc.) concerning 'usage' from the point of view of a given audience (e.g., domain engineers, testers, product units, product engineers) in a given context. Finally, a *Goal* is refined in terms of *Questions*. A question should be open-ended and focus on the specific aspect of the goal that needs to be measured. Hence, questions are to be answered with the help of *Metrics*, i.e., measures that indicate progress towards the goal.

For example, consider the *Commenting* feature as the 'object of study' (see Fig. 1). For this object, possible purposes include:

- feature scoping. Engineers might want to assess the extent Google Scholar is queried from within *Commenting*. The rationale goes as if this particular utility is seldom used w.r.t. the usage of *Commenting*, then it might be better to detach it from *Commenting* and offer it as a separate sub-feature.

- feature optimization. *Commenting* is made available in two ways: 'double click' and 'right click'. Engineers wonder whether this double choice may confuse users and, if so, which is the most popular approach for this feature to show up.

If the SPL Control Board (i.e., the committee in charge of the SPL fate) conducts this study in the context of planning the next WACline release, the resulting goal is left as follows:

> Analyze *Commenting* with the purpose of *scoping & optimization* with respect to *usage* from the point of view of the *Control Board* in the context of *the planning for the next release*

The example so far assumes the context to be the SPL platform as a whole. Though the object of the study is limited to the Commenting's codebase (i.e., the *#ifdef* blocks involving Commenting), the context is the entire SPL portfolio, i.e., track Commenting no matter the variant in which this feature is deployed.

In addition, we also came across scenarios where the analyst was interested in the usage of a feature but for a particular set of variants. Back to the running example, *Commenting* adds menus and widgets to the Web interface. If utilized with other GUI-intensive features, such as *Highlighting*, *Commenting* may lead to cluttered interfaces that can potentially discourage users. It should be noted that *Commenting* and *Highlighting* have no feature dependency; hence, they do not appear together in pre-compilation directives. However, the combining effect regarding the cluttered interface requires the tracking of *Commenting* for those variants where Highlighting is also deployed. We underline that 'contextual features' might exhibit no dependency on 'objectual features' as captured in feature models yet exhibit some sort of implications that qualify the context of the tracking (e.g. F1's power consumption might impact the response rate of F2 and F3, yet this implication might not always be reflected in the feature model). In this case, the goal would look as follows:

> Analyze *Commenting* with the purpose of *usability evaluation* with respect to *usage* from the point of view of the *UX designer* in the context of *Highlighting*

Here, the *Highlighting* feature characterizes the context where Commenting is to be tracked, i.e., the subset of the SPL portfolio subject to tracking.

We now move to the questions. Previous goals on Commenting could be addressed through two questions with their respective metric counterparts, namely:

- How is *Commenting* used? Commenting permits querying Google Scholar from within. The Control Board wonders about the extent this feature is coupled with the fact of commenting, and if sparse, decides to offer it as a separate sub-feature. To this end, *GoogleScholarHappened* metric is introduced;
- How is *Commenting* enacted? This feature offers two enactment modes: *DoubleClickEnactment* or *RightClickEnactment*. Assessing the popularity of each of these modes helps to determine if they are alternatives or if any of them is residual, and hence, a candidate to step out in the next release.

For these metrics to be enactable, the feedback specification needs to indicate how is going to be supported: the pointcut. A pointcut is a join point which

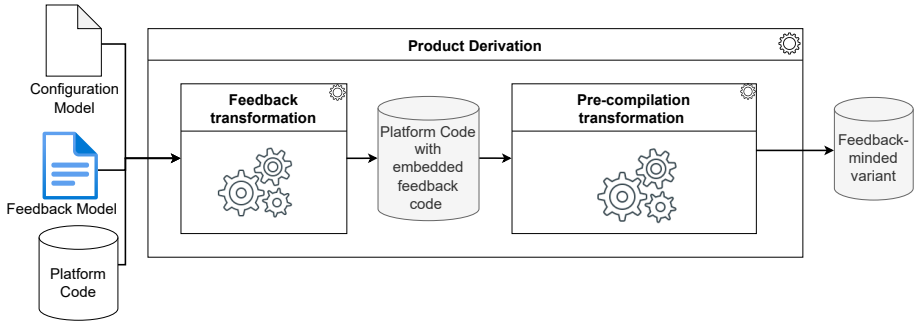

Figure 5: Two-step Product Derivation. First, the Feedback Transformation injects the tracking hit along with the feedback model. Second, the Pre-compilation Transformation filters out those *ifdef* blocks that do not meet the configuration model. The result is a feedback-minded product.

denotes a given line of code within a file. A pointcut denotes the point in 'the slice of the SPL codebase' (i.e., the object) where a tracking hit should be raised and collected in the feedback log. A metric might involve one or more pointcuts. As we will see later, the metric *DoubleClickEnactment* will require two pointcuts, one for each of the points in the code where double-click can be enacted. Section 5.1 will introduce the concrete syntax for this model using YAML.

4.1.2. Feedback Transformation

Feedback transformation refers to the interpretation of the feedback model. In one-off development, the analyst manually introduces the tracking instructions into the product codebase. SPLs aim at reducing, if not removing, coding during AE. The SPL way is to resort to 'transformations' that generate the variant through 'transforming' the SPL platform along with a given configuration model. Likewise, feedback goals might be better supported through 'transforming' the SPL platform rather than directly acting upon the variant code. Rather than polluting the platform code, 'separation of concerns' advises for feedback-specific code to be added on the fly during Product Derivation. Specifically, Feedback Transformation takes the SPL platform, the configuration model and the feedback model as input and injects the corresponding tracking code (see Fig. 5). This code is next subject to the traditional transformation based on pre-compilation directives. The resulting variant will be feedback-minded, i.e., it will populate the feedback log at run-time.

4.1.3. Feedback Gathering

Feedback gathering refers to capturing usage data from users (Johanssen et al., 2019). Traditionally, in the same vein as any other data, usage data should meet specific quality properties (Redman, 2013), namely accuracy (i.e., tracking hits should be error-free and reflect real-world values), completeness (i.e., tracking hits should include at least all the information specified in the

feedback model), consistency (i.e., data gathered should be consistent among the different products) and, timeliness (i.e., data should always be up-to-date and reflect the real use of the platform). Thus, feedback gathering goes beyond a programming issue to also include data governance considerations. The question is what SPLE brings to account for these quality factors:

- accuracy. SPLE-wise, the feedback specification and specifically, the contextual features provide a declarative way to set the scope of the feedback analysis. Rather than naming the products subject to the tracking (extensional approach), SPLs offer features as an implicit means to single out the variants of interests. On the downside, faults in the feedback specification will have a more significant impact as they propagate to a larger set of variants;

- completeness. SPLE-wise, completeness imposes a careful consideration not only of the contextual features to be considered but also of the end-users (or framing installations) where the variants are to be deployed. This might be an issue if domain engineers do not have access to the ultimate users/installations where the variants are run;

- consistency. SPLE-wise, the transformation approach ensures that the feedback code is generated in the same way no matter the variants in which this code ends up being injected;

- timeliness. SPLs evolve quicker than one-off applications if only by the number of variants they consider and the different sizes of the codebase. The larger the number of variants and the codebase, the larger the chances for feedback analysis to evolve, and hence, the more challenging to keep usage data up-to-date. Traditional product-based feedback does not scale to this volatile scenario. Keeping pace with feedback analysis evolution calls for model-driven approaches where analysis is described at a higher-abstraction level, and transformation takes care of implementation details. Model-driven not only accounts for code consistency but also speeds up development, hence, reducing the cost of updating feedback specification which facilitates up-to-date data.

4.1.4. Feedback Analysis

To aid decision-making in dynamic environments, dashboards are deemed essential (López et al., 2021). A dashboard is a graphical user interface that provides an overview of key performance indicators, such as feature usage, relevant to a particular business objective or process (such as feature optimization) (Few, 2006). *Google Analytics* offers dashboards for website owners to analyze trends and patterns in their visitors' engagement with the website (Plaza, 2011). However, SPL scenarios differ in two key aspects. Firstly, the focus of the tracking is not a single application but rather (a subset of) the SPL portfolio. Secondly, features become the central unit of analysis. Thus, dashboards must be designed to accommodate these unique characteristics.

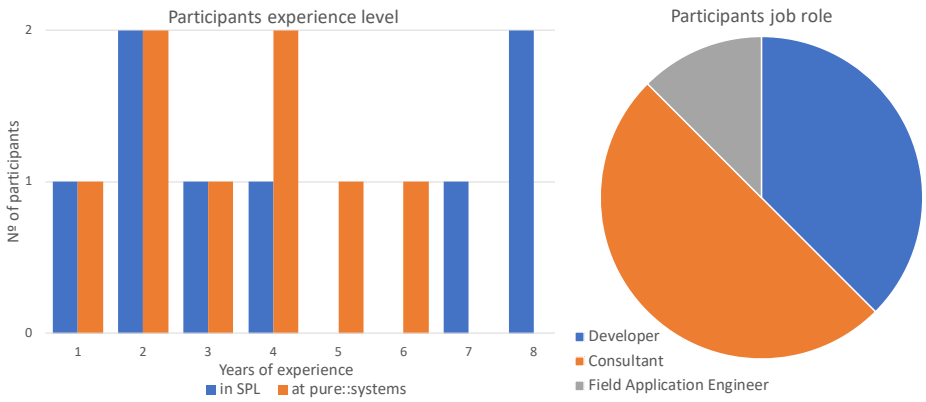

Figure 6: Participants demographics: SPL experience & role played.

4.2. Validating the requirements

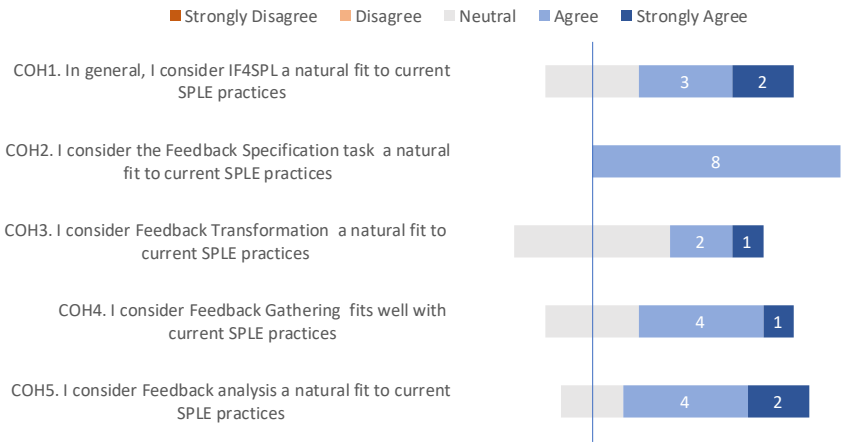

Figure 7: Assessing coherence for IF4SPL.

The previous subsection introduced a way to support implicit feedback in SPLE, named IF4SPLE (i.e., Implicit Feedback for SPLE). This proposal was derived from our experience handling implicit feedback in WACline. To improve external validity, this section conducts a survey on the extent IF4SPLE accounts for *coherence* (i.e., the quality of forming a unified whole with traditional SPLE practices) and *consistency* (i.e., the quality of seamlessly fitting with prior roles in SPLE). The next paragraphs describe the evaluation of IF4SPLE principles.

Goal. The purpose of this study is to *assess* the goodness of the IF4SPL principles for *seamlessly introducing Implicit Feedback into SPLE practices* from the

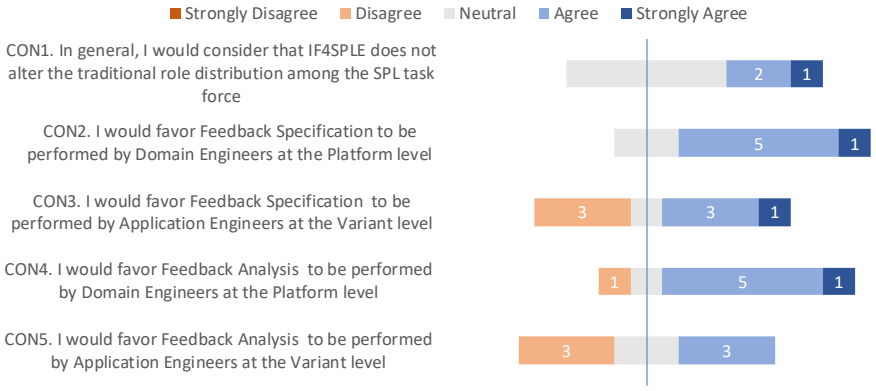

Figure 8: Assessing consistency for IF4SPL.

point of view of *annotated-based SPL practitioners* in the context of *pure-systems GmbH* *pure-systems GmbH*[1] and annotated-based SPLs.

Participants. To look into the generality of the solution, we resorted to *pure-systems GmbH*, which is a leading provider of software product lines and variant management engineering tools, including *pure::variants*. The suitability of this company is twofold. First, *pure-systems GmbH* is well placed to assess the extent *FEACKER* is seamlessly integrated with its own tool (i.e., *pure::variants*). Second, *pure-systems GmbH* acts as a consultant for a broad range of companies, and hence, faces different requirements and domains. Hence, its participation will provide a high variation of settings in which to support generalization.

Design. The experiment was conducted within the monthly seminar regularly kept at *pure-systems GmbH*. The seminar lasted for 90 minutes. The participants were introduced to the importance of Implicit Feedback in single-off development and the interest in bringing these benefits to SPLE. Next, IF4SPLE was introduced. At this time, participants were asked to fill up the questionnaire (see next paragraph). Fifteen employees attended the seminar, yet only eight completed the survey. For these eight employees, demographic data is displayed in Fig. 6.

Instrument. To better profile what is meant by 'a seamless introduction', we characterize this adverb in terms of *coherence* (i.e., the quality of forming a unified whole with traditional SPLE practices) and *consistency* (i.e., the quality of seamlessly fitting with prior roles in SPLE). We resort to an *ad-hoc* questionnaire where items are assessed through a five-point Likert that is refined for each of the tasks introduced by IF4SPLE (i.e., Feedback Specification, Feedback Transformation, and so on). Whereas coherence looks at the process, consistency

[1] `https://www.pure-systems.com/`

regards the roles played by participants in an SPL organization. Specifically, we consider two roles (i.e., DE vs AE). The rationale reads as Implicit Feedback might be conducted as part of DE with the participation (or not) of application developers. And the other way around, it is possible for Implicit Feedback to be defined for a set of products without implying the whole platform. Figs. 7 and 8 show the results of IF4SPLE's coherence and consistency, respectively.

When it comes to assessing the internal consistency, or reliability, of questionnaires, Cronbach's alpha coefficient measure is used. It helps determine whether a collection of questions consistently measures the same characteristic. Cronbach's alpha quantifies the level of agreement on a standardized 0 to 1 scale. Higher values indicate higher agreement between items. Thus, high Cronbach's alpha values indicate that response values for each participant across a set of questions are consistent. On these grounds, we calculate the Cronbach's alpha values of the questionnaire items for coherence and consistency, leading to a result of 0.63, and 0.8 alpha values, respectively. In the case of consistency, we reversed the results of questions CON3 and CON5 as they favor AE to conduct implicit feedback affairs. An acceptable degree of α reliability is defined as 0.6 to 0.7, and a very good level is defined as 0.8 or greater (Ursachi et al., 2015). Therefore, we can consider the questionnaire reliable enough.

Results. **Coherence.** Participants generally believe IF4SPLE is a natural fit for current SPLE practices (question COH1). Feedback Specification (question COH2) and Feedback Analysis (question COH5) were the tasks that unanimous agreement reached about being conducted at the level of DE, hence moving Implicit Feedback from being a product concern to being a platform concern. Surprisingly, while positive, feedback transformation (i.e., injecting feedback concerns when the product is generated mimicking traditional techniques based on pre-compilation directives) did not meet our expectations, with most participants ranking it as 'neutral'.

Consistency. Participants generally considered that IF4SPLE does not alter the traditional role distribution among the SPL task force (question CON1). That said, in line with IF4SPLE, *pure::variant*'s employees seem to favor domain engineers before application engineers when conducting feedback specification and analysis (questions CON2 and CON4).

The next section looks at the feasibility of this approach, using *pure::variants* as the variability manager.

5. Proof-of-concept: implicit feedback in *pure::variants*

This section introduces *FEACKER* (a portmanteau for FEAture and traCKER), an extension for *pure::variants* along the IF4SPLE principles. *FEACKER* is publicly available at `https://github.com/onekin/FEACKER`.

The deployment diagram of the *FEACKER* system is presented in Fig. 9. The specification and transformation components are responsible for incorporating the tracking code, utilizing GA as the tracking framework. At run-time, variant execution on the *Client PC* generates data in the GA log, which is

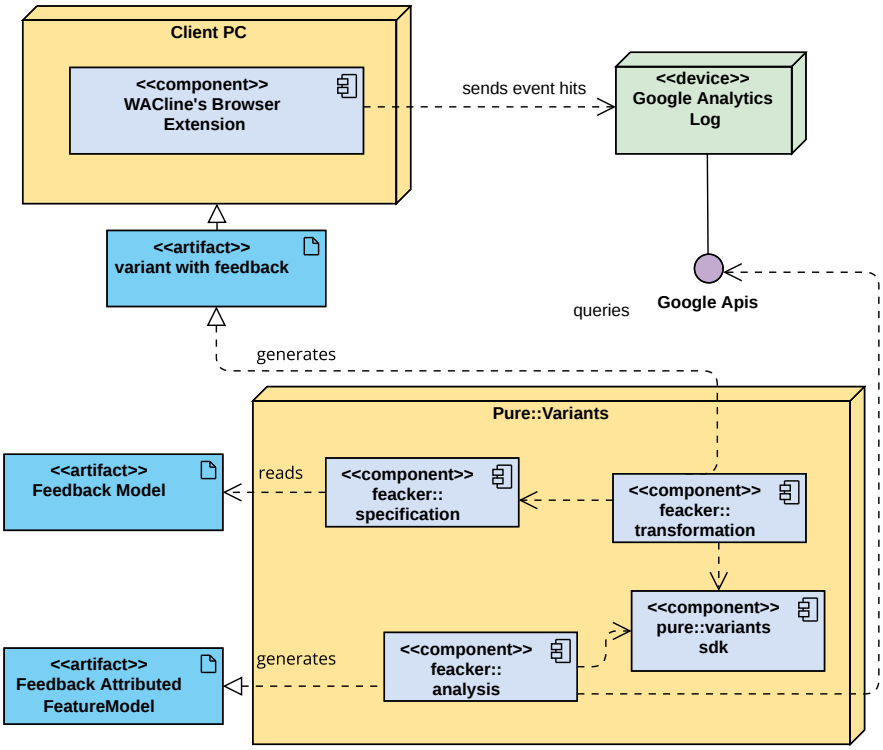

Figure 9: *FEACKER*'s deployment diagram. *Pure::variants* is extended with three new components: *feacker::specification, feacker::analysis* and *feacker::transformation*

.

later analyzed by the analysis component of *FEACKER*. This component processes the feedback data and presents it through an attributed feature model, as described in Section 5.4[2]. Details follow.

5.1. Feedback Specification

FEACKER resorts to the YAML language for the concrete syntax of the Feedback Model introduced in Section 4.1.1. Fig. 10 provides a *myFirst-feedback.yaml* snippet for the running example. Clauses on lines 2-5 capture the Goal. Two clauses have a documentation purpose (*audience* & *purpose*), whereas *target* and *context* have execution implications. Both hold a feature-based predicate similar to the one in pre-compilation directives (so far, only

[2]The implementation of the solution relies on the *pure::variants* SDK, utilizing three extension points: specification, transformation, and analysis. It should be noted that *pure::variants* uses pre-compilation directives for variability support and introduces the concept of a Family Model, which separates the SPL platform implementation through logical names referred to as 'model elements'.

```yaml
goals:
  - audience: "Control Board"
    purpose: ["Scoping","Optimization"]
    target: Commenting
    context: Highligting or Emailing
    questions:
      - label: "Q1"
        description: "How is Commenting used?"
        metrics:
          - label: "queryGoogleScholarHappened"
            pointCuts:
              - fileName: "CommentingForm.js"
                path: "app/scripts/annotationManagement/purposes/"
                anchor: |
      - label: "Q2"
        description: "How is Commenting enacted?"
        metrics:
        - label: "doubleClickEnactment"
          pointCuts:
            - fileName: "ReadAnnotation.js"
              path: "app/scripts/annotationManagement/read/"
              anchor: |
              highlight.addEventListener('dblclick', () => {
                this.openCommentingForm(annotation)
                [*GA_INJECT*]
              })
            - fileName: "ReadAnnotation.js"
              path: "app/scripts/annotationManagement/read/"
              anchor: |
               {
                 this.openCommentingForm(annotation)
                 [*GA_INJECT*]
               }
        - label: "rightClickEnactment"
          pointCuts:
            - fileName: "ReadComment.js"
              path: "app/scripts/annotationManagement/read/"
              anchor: |
                ...
```

Figure 10: The feedback model for the goal: Analyze *Commenting* with the purpose of *scoping & optimization* with respect to *usage* from the point of view of the *Control Board* in the context of *variants that exhibit either Highlighting or Emailing*.

AND/OR are supported). *Context* holds 'a contextual predicate' to delimit the variants whose configuration model should match this predicate. Target holds 'an objectual predicate' to characterize the *#ifdef* blocks whose pre-compilation directives should satisfy this predicate. Thus, the slice of the platform codebase to be tracked is intensively defined by the *#ifdef* blocks whose pre-compilation directives satisfy the objectual predicate when deployed in variants matching the contextual predicate subject to the tracking (i.e., the slice of the platform codebase to be tracked). Both predicates are consulted during feedback transformation to filter out the *#ifdef* blocks subject to tracking as a result of matching

the variability of interest (object of study) in the appropriate variants (the context). Back to the example, the tracking scope is that of *#ifdef* blocks whose pre-compilation directives hold *Commenting* when in variants that exhibit either *Highlight* or *Emailing*.

As for questions, they do not have any execution implications apart from structuring the metrics. The sample snippet introduced two questions: Q1 and Q2. The former is grounded on *QueryGoogleScholarHappened* metric. As for Q2, it is based on the counting of *DoubleClickEnactment* and *RightClickEnactment*.

Finally, pointcuts are captured through three clauses. *FileName* and *Path* single out the code file in *pure::variants*' Family Model. As for *the anchor*, it holds the line of code where the GA_INJECT directive is to be injected at transformation time.

5.2. Feedback Transformation

In *pure::variants*, Product Derivation has two inputs: the configuration model (a .vdm file) and the SPL platform. The output of this process is a variant product, where *#ifdef* blocks that do not align with the configuration model are removed. This filtering of non-conforming blocks is achieved in *pure::variants* by selecting the 'File Processing' option. This is depicted on the right side of Fig. 11 (b).

In *FEACKER*, Product Derivation extends *pure::variants'* through an additional step: Feedback Transformation. The output is also a variant, but now the variant code includes GA hints in accordance with the feedback model. Fig. 11 (c) illustrates the progression of a code snippet. The process departs from the platform codebase. The Feedback Transformation injects the GA hits along with the feedback model. GA hits are composed of an interaction (*eventAction*) that happens during the usage of the feature (*eventCategory*) while using a given product (*variant*). Finally, the pre-compilation directives are removed from the code. The output is a feedback-minded variant. Refer to Appendix Appendix A for the algorithm.

5.3. Feedback Gathering

FEACKER resorts to GA for feedback gathering. Event data is collected into the GA feedback log. The utilization of GA presents both advantages and disadvantages. The benefits include GA's ability to comprehensively collect event data regarding visitors, traffic sources, content, and conversions. Additionally, GA provides advanced data visualization tools and it is highly scalable. On the downside, keeping event data at Google might jeopardize confidentiality.

5.4. Feedback Analysis

GA provides data visualization tools to track single apps. By contrast, we pursue feature-centric dashboards that track an SPL portfolio. This is a topic in its own right. For completeness' sake, this subsection explores the use of attributed feature models for feedback visualization.

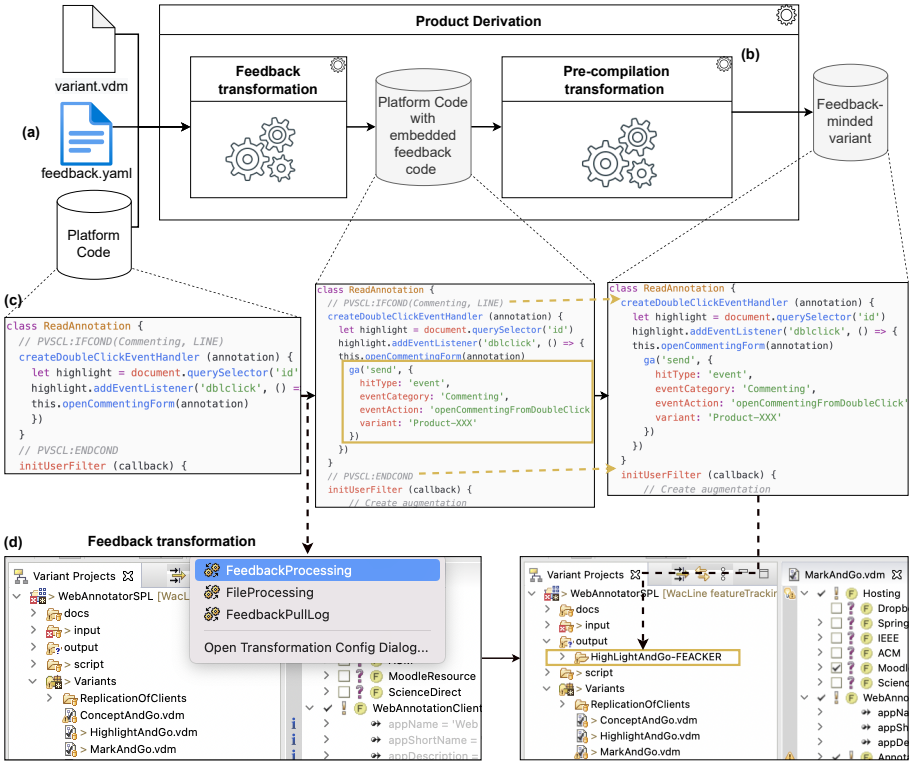

Figure 11: FEACKER's realization of Fig. 5 with the feedback model supported as a *yaml* file (a). The Feedback Transformation takes the platform code as input and injects the GA hits to the corresponding *#ifdefs* (c). The resulting code is next subjected to the Pre-compilation transformation that filters out *#ifdefs* that do not meet the configuration model. GUI-wise, *pure::variants* is extended with a second user interaction: *FeedbackProcessing* (d). On clicking, FEACKER conducts the Feedback Transformation, and next invokes *pure::variants*' *FileProcessing* to launch the Pre-compilation Transformation. The result is a feedback-minded variant, i.e., its usage will populate the feedback log.

In an attributed feature model, extra-functional information is captured as attributes of features (Benavides et al., 2010). Examples include non-functional properties (e.g., cost, memory consumption, price) or rationales for feature inclusion. This permits leveraging the automated analysis of feature models by incorporating these usage parameters (Galindo et al., 2019). Likewise, we add 'usage' as a feature attribute. However, unlike other attributes such as cost or memory consumption, 'usage' is highly volatile and needs frequent updates. In this setting, databases resort to derived attributes, i.e., attributes that do not exist in the physical database but are derived at query time from the database. Regarding derived attributes, two steps are distinguished: definition (when the query is defined) and enactment (when the query is triggered).

FEACKER supports this vision, namely:

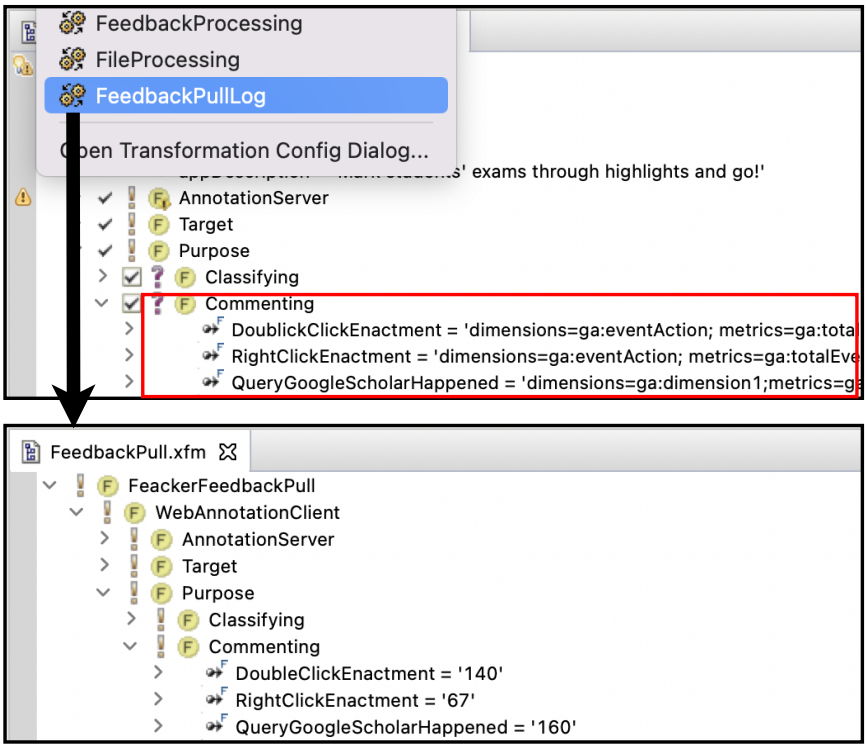

Figure 12: Attributed Feature models: *Commenting* holds four attributes for GA API calling parameters (top side). Select *FeedbackPull* from the drop-down menu for these API calls to be enacted. Results are stored in *FeedbackPull.xfm*, a copy of the input feature model where feature-usage attributes will show the metric values as returned by the API call (bottom side).

- definition time. Analysts might enlarge the feature model with usage attributes. These attributes hold a GA query to obtain the aggregate for this metric using the GA's API. Fig. 12 (top) illustrates this vision for our running example. *Commenting* is the object of study for three metrics: *DoubleClickEnactment, RightClickEnactment*, and *QueryGoogleScholarHappened*,

- enactment time. It is up to the analysts to decide when to enact these queries. To this end, *FEACKER* adds a 'FeedbackLogPull' option to the *pure::variants* processing menu. When selected, *FEACKER* (1) retrieves the queries held as values of the attributes, (2) executes the queries, and (3) creates a clone of the feature model, where attributes now hold the GA responses (Fig. 12 (bottom)). Refer to Appendix Appendix A for the algorithm.

At present, query results are integrated into the model without further processing. Subsequent executions of 'FeedbackPull' will replace the existing clone

with a new one that mirrors the current feedback log. All in all, using feature models falls short. First, it is not evident how to visualize metrics that involve multiple features (e.g., usage of F1 but not F2). Second, feature models offer a static view of the current variability, whereas feedback analysis necessitates a representation of how usage evolves. This calls for *pure::variants* to be enlarged with full-fledged dashboards. Alternatively, extending GA dashboards with feature-centric visualizations. This is still a work in progress.

6. Proof-of-value: Evaluation

6.1. Design of the Evaluation

Table 1: In the pursuit of a realistic evaluation.

	Realistic ...			
	participants	**tasks**	**environment**	**Instrument**
Pure::Systems	✓	×	✓	TAM
WACline	✓	✓	×	Focus Group

ADR emphasizes that authenticity is a more important ingredient than controlled settings (Sein et al., 2011). For an authentic evaluation, Sjøberg et al. (2002) introduce three factors: (1) realistic participants, (2) realistic tasks, and (3) a realistic environment. Sjøberg et al. recognize the difficulties of meeting all factors simultaneously, given the developing nature of the interventions in research.

FEACKER is not an exception; in fact, quite the opposite. However, the participation of practitioners from WACline and *pure-systems GmbH* provides a unique opportunity to assess the utility of *FEACKER*'s outcomes. The practitioners from WACline meet the first criteria of realistic participants, as WACline is web-based. They also used *FEACKER* in a real setting. However, WACline's limited size may limit the results' generalizability. On the other hand, practitioners from *pure-systems GmbH* are well placed to assess the seamless integration of *FEACKER* with *pure::variants*. Furthermore, the company is a consultant for a broad range of customers, providing insights on the extent the approach could be generalized to settings other than Web-based variants. However, they have not used *FEACKER* in a real task. As a result, we opt for a hybrid evaluation that combines both quantitative and qualitative evaluation. Specifically (see Table 1),

- for quantitative evaluation, we resort to the Technology Acceptance Model (TAM) as the instrument and *pure-systems GmbH* as the participant. TAM is founded upon the hypothesis that technology acceptance and use can be explained by a user's internal beliefs, attitudes, and intentions (Turner et al., 2010),
- for qualitative evaluation, we draw on Focus Groups as the instrument and WACline practitioners as the participants. Focus groups are suggested

in Software Engineering as an empirical method for getting practitioner input and qualitative insights, particularly in the early phases of research for identifying issues (Kontio et al., 2008). In this case, the evaluation resorts to a realistic task (i.e., the Web-based *Commenting* feature), and the focus groups are structured along the dissents expressed in the TAM evaluation.

This hybrid approach rests upon the assumption that both populations (i.e., *pure-systems GmbH* and WACline) share a common mindset that sustains that *the intention of use* of the former serves to guide *the real use* of the latter. This is undoubtedly a threat to 'hybrid-ness' validity, yet this likeness between both populations is (partially) supported by both using *pure::variants* as the variability manager.

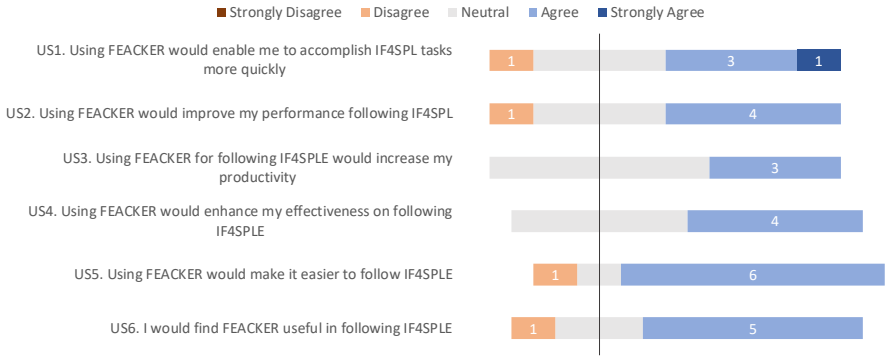

Figure 13: *FEACKER*'s perceived usefulness.

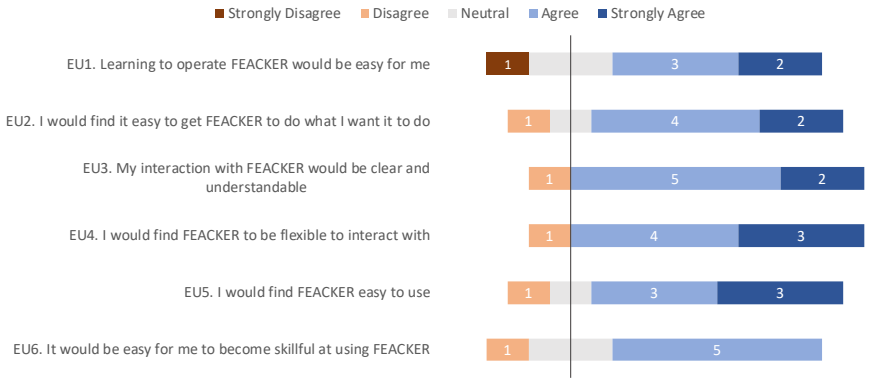

Figure 14: *FEACKER*'s perceived ease of use.

6.2. TAM evaluation

Goal. The purpose of this study is to *assess* the perceived ease of use and usefulness of *FEACKER* with respect to *introducing IF4SPLE practices into pure::variants* from the point of view of *annotated-based SPL practitioners* in the context of *pure-systems GmbH*.

Participants. We sought the assistance of employees at *pure-systems GmbH*. As they were involved in commercialising and developing *pure::variants*, we believed they were suitably positioned to assess its 'ease of use'. Furthermore, half of the participants were consultants, exposing them to a wide range of domains, thereby providing a wider perspective on the FEACKER's 'usefulness'.

Design. FEACKER was evaluated as part of the monthly seminar in which the IF4SPLE was previously evaluated (as described in Section 4.2). The seminar was structured as follows. An initial 90-minute session provided an overview of implicit feedback, including an in-depth examination of the *FEACKER's* components. A live demonstration of the tool followed this. Participants were encouraged to ask questions throughout the seminar. Finally, participants were allowed to interact with the tool and complete a TAM questionnaire.

Instrument. The study evaluates two internal variables of TAM: perceived usefulness and perceived ease of use. Rationales follow. As consultants of *pure::variants*, the participants were in a good position to provide an informed opinion on how much they believe using *FEACKER* would improve their job performance (perceived usefulness). As developers of *pure::variants*, the participants could provide an informed opinion on the degree of ease in using *FEACKER* for existing *pure::variants* users (perceived ease of use).

Results. Fig. 13 displays the chart for the variable 'perceived usefulness'. Notice that 'usefulness' is measured regarding the support given to the IF4SPLE process. The chart reveals a prevailing agreement among participants regarding the usefulness of *FEACKER*. However, the chart also indicates there is no full consensus on this matter (this will later serve to inform the Focus Group). As for 'perceived ease of use' (see Fig. 14), results tend to be more conclusive on the seamlessness to which *FEACKER* is embedded within *pure::variants*. The results suggest that using *FEACKER* does not interfere with the existing GUI gestures of *pure::variants*.

Threats to validity.

- **Construct Validity** refers to the level of precision by which the variables specified in research assess the underlying constructs of concern. Here, the constructs are 'ease of use' and 'usefulness'. To mitigate its potential influence, the researchers utilized the TAM questionnaire. To ensure internal consistency, Cronbach's alpha is calculated for the questionnaire, which resulted in an α value of 0.62 and 0.97 for usefulness and ease of use, respectively. Cronbach alpha values of 0.7 or higher indicate acceptable

internal consistency, with values above 0.9 perceived as redundancy of some questions.

- **Internal Validity** refers to how much the independent variable(s) or intervention was the actual cause of the changes observed in the dependent variable. Here, the intervention is *FEACKER*. Yet, other factors besides *FEACKER* might influence the results. First, the background of the participants. In this regard, the researchers consulted employees of *pure-systems Gmbh*. Except for one, all participants have at least two-year experience in both SPL and *pure::variants* (see demographics at Fig. 6). Second, the clarity and understandability of the questionnaire. To mitigate its potential influence, we resort to the TAM questionnaire, which has been widely employed and validated in assessing software engineering artefacts (Wallace and Sheetz, 2014).

- **External Validity** assesses to what extent the results obtained in this study can be generalized to other scopes different to the one approached in the study. We defer the discussion of this issue to Section 7.

6.3. Focus Group evaluation

TAM measures *intention to use*, but it is not based on *the real use*, hence, contradicting the second principle of an authentic evaluation (Sjøberg et al., 2002). To mitigate this drawback, we resort to a second evaluation, now based on Focus Groups.

Goal. The purpose of this study is to *delve into* the divergent items of a previous TAM evaluation with respect to *introducing IF4SPLE practices into pure::variants* from the point of view of *annotated-based SPL practitioners* in the context of the WACline SPL.

Participants. The group was formed by three engineers with at least two-year experience in using *pure::variants*[3]. All of them were part of WACline's core-development team.

Design. Participants were asked to create a feedback model collaboratively. The model should approach analysis opportunities they were interested in. They came up with 18 different hits related to 11 features[4]. Feedback was gathered during a week about WACline products running in a sandbox framework. So collected feedback was presented through the usage-based attributed feature model shown in Fig. 12. Once the usage feedback was collected, the focus groups started.

[3]Focus groups are typically made up of 3-12 people (Parker and Tritter, 2006)

[4]The resultant feedback model can be found in a branch of WACline's repository: `https://github.com/onekin/WacLine/tree/featureTracking/WACline`

Instrument. The focus group was structured along those issues that rose the most significant divergence in the TAM evaluation (see Fig. 13 and Fig. 14). Each issue was turned into a question for the focus group, namely:

- **Is the feedback model able to capture your feedback needs effectively?** This question accounts for the TAM's US1 statement: *Using FEACKER would enable me to accomplish IF4SPL tasks more quickly.* Fig. 13 depicts the divergences in the answers: disagree (1), neutral (3), agree (3), and strongly disagree (1). Among the tasks introduced in the IF4SPL, the more labour-intensive one is feedback specification since it requires knowledge about the feature model, the SPL product portfolio, and the code of the feature(s) at hand. Hence, the focus question delves into the extent FEACKER's feedback model can directly capture the needs of the analysts.

- **How seamlessly were IF4SPLE's new tasks integrated with your SPLE practices?** This question accounts for the TAM's US2 statement: *Using FEACKER would improve my performance following IF4SPL.* Fig. 13 depicts the divergences in the answers: disagree (1), neutral (3), and agree (4). IF4SPLE proposes the need for an analysis space and the consequent tasks in SPLE. The way that these tasks are integrated into the traditional workflow may have a direct influence on the performance of the team. Hence, the focus question seeks to explore the impact of IF4SPLE tasks on the performance of domain engineers.

- **How seamless was *FEACKER* integrated with *pure::variants*' gestures?** This question accounts for the TAM's EU1 statement: *Learning to operate FEACKER would be easy for me.* Fig. 13 depicts the divergences in the answers: strongly disagree (1), neutral (2), agree (3), and strongly agree (2). Being *FEACKER* an extension can disrupt the way of interacting with *pure::variants*. Thus, this question tries to determine how much *FEACKER* disturbs existing *pure::variants*' gestures.

Results.

- **Is the feedback model able to capture your feedback needs effectively?** Participants indicated that most of their feedback expressions refer to a single feature. No case popped up with AND/OR expressions. Two participants missed the ability to record values of variables at execution time. This would allow capturing the number of items a user has at the same time to operate within the GUI or capture values that could make the system fail.

- **How seamlessly were IF4SPLE's new tasks integrated with your SPLE practices?**Participants agreed on feedback functionality being subject to the same reuse principles as domain functionality and hence, being moved to DE. As one participant put it: "If testing is mainly a DE

activity, why should feedback analysis differ? After all, usage feedback informs feature evolution, sort of perfective maintenance". This said, they expressed two worries. First, the feedback model is fragile upon changes in the *#ifdef* blocks. This introduces a dependency between code developers and feedback analysts. How severe is this limitation? Participants observe that feedback analysis tends to be conducted for mature features where customer usage drives feature evolution rather than feature 'upbringing'. For mature features, the expectation is that code would be relatively stable and hence, with a low risk of interference between feedback analysis and feature developers. Second and most important, upgrades on the feedback model would only take effect by deriving the products again. This might be a severe concern for some installations, but, after all, this is what continuous deployment is supposed to be. Finally, some exciting issues arose with no clear answer: How is the evolution of feedback models conducted? How would different feedback models co-exist? Should variability be engineered within the feedback model itself? What strategy would be to face large feature models with intensive feedback needs? Should techniques similar to those for SPL testing be used?

- **How seamless was *FEACKER* integrated with *pure::variants*' gestures?** Participants mostly appreciate conducting feedback analysis without leaving *pure::variants*. This seems to suggest that variability managers should not overlook this concern.

Nevertheless, some issues emerged:

- Feedback Model. All participants found it cumbersome to specify. Difficulties came from Feedback Models' dependency on the *#ifdef* blocks, specifically, the anchor specification. This limitation might be eventually mitigated through dedicated editors,
- Feedback Transformation. Participants found the approach intuitive and akin to the *modus operandi* of *pure::variants*,
- Attributed Feature Models. Participants foresaw the benefits of introducing feature usage for more sophisticated feature analysis. However, feature models are very poor as dashboards. Participants indicate the need to enhance variability managers with this functionality[5] or, instead, tap into existing dashboard applications through dedicated connectors.

6.4. Threats to validity

We follow here the threats to validity for Qualitative Research as described by Maxwell (1992)

[5]It rests to be seen whether *pure::variants*' tabular view of features which includes attributes, might provide a better fit.

- **Descriptive Validity** refers to the accuracy and completeness of the data. To mitigate this threat, we took notes about participants' reactions as well as recording the session's audio to facilitate further analysis. Moreover, we asked for clarifications from the participants during the discussion. Another limitation is the small number of participants. This is partially due to the pursuit of realism in selecting the participants, who were required to be knowledgeable about both *pure::variants* and GA. Favouring realism versus quantity is supported by the literature on focus groups, which recommends purposive sampling as the method for participant recruiting (Morgan, 1997).

- **Interpretive Validity** refers to how well researchers capture participants' meaning of events or behaviours without imposing their perspective. It should be noted that the problem was pointed out by the practitioners themselves when attempting to accord *pure::variants* and GA. To improve interpretative validity, we began the session with a brief presentation on the study's objectives and the proposed intervention. This presentation was aimed at establishing common terminology and avoiding misunderstandings.

- **Reproducibility**. *FEACKER* and WACline are readily accessible to the public through download. The additional infrastructure utilized in this study, including *pure::variants* and GA, is widely used among practitioners, facilitating the dissemination of this work's results.

7. Generalization

Following ADR, the situated learning from the research project should be further developed into general solution concepts for a class of field problems (Sein et al., 2011). Sein et al. suggest three levels for this conceptual move:

- generalization of the problem instance, i.e., to what extent is *implicit feedback* a *problem* for SPL other than WACline?

- generalization of the solution instance, i.e., to what extent is *FEACKER* a *solution* to implicit feedback in SPLs?

- derivation of design principles, i.e., what sort of design knowledge can be distilled from the *FEACKER* experience that might inform variability managers other than *pure::variants*.

The rest of this section tackles these questions.

7.1. Generalization of the Problem Instance

Generalizing from a local experience starts by identifying the contextual parameters that might have influenced the intervention and its utility. Table 2 gathers what we believe are the main contextual parameters that frame our

Table 2: Contextual characterization of the local experience in WACline.

Technical environment	Programming language	*JavaScript*
	Branching strategy	Grow-and-prune
	Variability manager	*pure::variants*
	Tracker Manager	*Google Analytics*
SPL Attributes	Lifespan	5 years
	Size (approx.)	85 features & 7 products
	Domain	Web
	Variability model	Annotation-based
Implicit Feedback	Purpose	Scoping

case study. The question arises about whether our setting is somehow unique or rather other SPL installations can share it.

The first consideration is the domain: the Web. Certainly, the Web is a pioneering domain in applying continuous deployment using implicit feedback: mobile-app development (Xiao et al., 2021); online training platform (Astegher et al., 2021); or mobile and web contexts (Oriol et al., 2018). However, we argue that the interest in implicit feedback is not limited to the Web. Clements's popular definition of an SPL as addressing 'a particular market segment' entails that the evolution of 'this market segment' should, naturally, go hand in hand with the evolution of the SPL (Clements and Northrop, 2002). Hence, SPL scoping (i.e., deciding on the features to be covered by the SPL) is necessarily 'continuous' to keep pace with this market segment. If SPL scoping is continuous, then implicit feedback will become a main informant of how this market segment evolves. And if so, chances are the problem of implicit feedback found in WACline is shared by other installations.

Notice that WACline's incentive for implicit feedback is in informing scoping. However, usage data is also useful for troubleshooting, fault prediction, supporting network operations or development activities (Dakkak et al., 2021b, 2022b), and enabling continuous experimentation (Mattos et al., 2018). Specifically, collected data provides valuable insights into the real-world behaviour and performance of embedded systems, allowing for the identification of potential issues and the development of solutions to improve performance. Additionally, models and simulations used in the evolution of embedded systems can be validated and refined, providing a more robust and accurate understanding of the system behaviour (Dakkak et al., 2021b, 2022b). We can then conjecture that implicit feedback would also be of interest to cyber-physical systems.

7.2. Generalization of the Solution Instance

FEACKER is an intervention for *pure::variants* as the event manager, GA as the event tracker, Web-based product portfolios, and integrated product deployment. The following paragraphs discuss the limitations of such characterization in the generalizability of *FEACKER*.

Technological limitations. The focus on *pure::variants* limits our solution to annotation-based SPLs. Other approaches using feature-oriented programming or component-based SPLs would need to resort to other means as our approach is

heavily based on pre-compilation directives. As for GA, *FEACKER*'s Feedback Model might be biased by hit specification in GA. Other trackers like Grafana , Matomo or Zoho might have different expressiveness, which would involve changes in the Feedback Model.

Organization limitations. *FEACKER* is conceived for organizations with access to SPL variants once deployed. Yet, to collect usage data, a data pipeline must be established that connects to the deployed variants within the customer's network. These variants might reside in a protected intranet with multiple security gates and checkpoints to prevent unauthorized access. The access setup for each customer will be unique and require customized configurations to connect the variants to the data collectors and into the pipeline. This, in turn, raises the issue of data completeness. Suppose access to variant usage is limited, and only a subset of variants can be sampled. In that case, data completeness refers to how well the collected data represents the full range of products used by customers. Collecting data from functioning nodes is necessary for accurate results in some situations. However, this may not be possible in SPLs with hundreds of deployed variants due to technical or resource constraints. The challenge in these cases is to gather data from a representative sample of variants that accurately reflects the SPL platform.

Domain limitations. *FEACKER* is an intervention for Web-based product portfolios. Using *FEACKER* in embedded and cyber-physical domains introduces additional challenges:

- *Performance impact.* The limited resources of embedded and cyber-physical systems make it crucial to optimize data collection, as these activities can consume internal resources. One solution is collecting data during low-traffic hours. The casuistic can be exacerbated in SPLs, where each variant configuration may suffer different performance impacts, calling for a bespoken solution to minimize the performance impact on each deployed variant (Dakkak et al., 2022a; Mattos et al., 2018).

- *Data dependability.* Embedded systems produce various data types, such as sensor readings, network status, performance metrics, and behaviour patterns. Evaluating just one type of data does not give a full understanding of its quality. While specific quality measures, such as integrity, can be applied to individual data types, a complete understanding of data quality requires analyzing the interrelationships and correlations between multiple data sources. The complexity arises when each product configuration has different sensors and data sources, requiring the feedback collection process to be aware of the unique settings and requirements of each configuration (Dakkak et al., 2021b).

7.3. Derivation of design principles

Design principles reflect knowledge of both IT and human behavior (Gregor et al., 2020). Accordingly, a design principle should provide cues about the

Table 3: Design Principles for Implicit Feedback in Variability Managers.

ID	**Provide *variability managers* with...**	**for Domain Engineers to...**	**Realization in *pure::variants***
DP1	... a goal-centric feedback specification model	... structure feedback directives using GQM	a *yaml* realization of GQM constructs
DP2	... a feature-based feedback model	... align feedback with other SPLE tasks	yaml's *target* clause in terms of features
DP3	... feedback transformations	... account for 'separation of concerns' to avoid polluting the platform code	A two-step Product Derivation
DP4	... feature-based feedback dashboards	... track SPL-wide metrics	Feature models with derived attributes

effect (e.g.., allowing for implicit feedback at the platform level), the cause (e.g., feedback transformation), and the context where this cause can be expected to yield the effect for the target audience (e.g., domain engineers). Table 3 outlines the main design principles. We consider as main design decisions the following: characterizing the feedback model in terms of features; structuring the feature model along the GQM model; and supporting the feedback model as a full-fledged model and hence executable by transformation.

8. Conclusion

Implicit feedback is widely recognized as a key enabler of continuous deployment for one-off products. This work examined how feedback practices can be extended to the level of the SPL platform, affecting both SPLE processes and variability manager tools. We advocate for placing features at the centre of the analysis and realizing feature tracking as a transformation approach. We developed *FEACKER* as a proof-of-concept, which is available for public inspection and use. The evaluation included both real practitioners in a TAM evaluation (n=8) and a focus group (n=3). The results suggest that the approach is seamless with respect to current practices but raises several issues regarding its generalizability.

This work sought a global perspective in introducing feedback analysis in SPLs. Each of the feedback tasks (i.e., specification, transformation, gathering and analysis) is a subject in its own right, raising diverse questions:

- How could feature usage leverage existing feature-analysis techniques? Could dependency between two features be weighted based on the (co)usage of these two features? Could this co-usage between features be used by configuration assistants to recommend co-features?
- How could SPL evolution become 'continuous' by tracking feature usage? Which sort of metrics can be most helpful in tracking SPL evolution?
- How can existing experiences on analysis dashboards be tuned for platform-based feedback?
- To what extent does implicit feedback account for the same benefits as in one-off development?

Ultimately, we hope to promote debate among researchers and practitioners about the challenges of introducing implicit feedback in variability-rich systems. At play, bringing continuous deployment into SPLE.

Appendix A. Algorithms

Feedback Transformation (Algorithm 1): The process starts by creating a copy of the platform code. Then, it filters the feedback directives specified in the feedback model (*yaml* file) based on the expressions that match the configuration model. The corresponding event specifications are processed for the directives that pass this filter. The event pointcuts are determined using the file names and anchors defined in the feedback model, with the file name corresponding to an artifact of the Family Model and the anchor serving to single out the code fragment where the GA hit is to be injected. Finally, *FEACKER* generates the GA hits and inserts them into the appropriate code lines.

Feedback Analysis (Algorithm 2): The algorithm gets as input a feature model. First, the analysis component searches for GA query parameters in the given feature model. The value of these attributes (i.e., the GA API query) is then used as the input to call the GA API. Finally, a copy of the feature model is created and the raw results obtained from GA API replace the query parameter value of the attributes in the cloned feature model.

Algorithm 1: Feedback Transformation takes a feedback model, the platform code and a variant configuration model as an input and returns the platform code with embedded feedback hits.

```
1  function PlatformCode feedbackTransformation( FeedbackModel feedbackModel,
       PlatformCode platformCode, VariantConfigurationModel
       variantConfigurationModel){
2      //Extract selected and deselected features from the variant configuration
3      Map<Feature,Boolean> selectedFeatures = extractFeatures(
           variantConfigurationModel)
4      //For each goal check whether the input variant is part of the goal context
           and if it satisfies the target expression
5      foreach(goal in feedbackModel.goals){
6          if(isExpressionTrue(feedbackModel.context, selectedFeatures) and
               isExpressionTrue(feedbackModel.target, selectedFeatures)){
7              //Create the directory for the platform code with the embedded GA code
8              Platform feedbackPlatformCode = copyDirectory(platformCode)
9                  //For each question in the goal, get its metrics
10                 foreach (question in goal.questions){
11                     foreach (metric in question.metrics){
12                         //For each pointcut find where the GA code should be
                               injected
13                         foreach (pointcut in metric.pointCuts){
14                             File file = findFile(feedbackPlatformCode, pointcut.
                                   fileName)
15                             foreach (ifdefBlock in file){
16                                 //Check if the #ifdef block satisfies the target
17                                 if(doesIfdefMatchExpression(feedbackModel.target,
                                       ifdefBlock)){
18                                     //Generate the google analytics hit code and
                                           add it to the given file
19                                     Integer anchorLine = findAnchor(pointcut.
                                           anchor, ifdefBlock, file)
20                                     addGoogleAnalyticsHit(anchorLine, metric,
                                           file)
21                                 }
22                             }
23                         }
24                     }
25                 }
26             }
27         }
28     }
29     return feedbackPlatformCode
30 }
```

Algorithm 2: Feedback Analysis, takes a feature model with GA query parameters on it and returns a cloned feature model with the raw results of those GA query.

```
1  function FeatureModel feedbackPull(FeatureModel implicitFeatureModel){
2      //Search for Google Analytics usage attributes in the implicit feature model.
3      Map<String, String> usageAttributeMap = findFeedbackQueryParameters(
           implicitFeatureModel)
4      Map<String, String> feedackResultsAttributeMap = new Map()
5      //For each attribute, get its value, call GA API and store the result
6      for each (usageAttribute in usageAttributeMap){
7          String googleAnalyticsQuery = usageAttribute.value
8          String feedbackData = googleAnalyticsApi.getFeedbackData(
               googleAnalyticsQuery)
9          //Store GA API results in an auxiliary map
10         feedackResultsAttributeMap[usageAttribute.name]=feedbackData
11     }
12     //Create the cloned feature model
13     FeatureModel clonedFeatureModel=copyFeatureModel(implicitFeatureModel)
14     //For each attribute replace the value with the result obtained from GA API
15     for each (feedbackResultAttribute in feedackResultsAttributeMap){
16         featureModelAttribute = findFeedbackAttributeInFeatureModel(
               feedbackResultAttribute.name, usageAttributedFeatureModel)
17         featureModelAttribute.setValue(feedbackResultAttribute.value)
18     }
19     //Save the changes of the cloned feature model
20     usageAttributedFeatureModel.saveChanges()
21     return clonedFeatureModel
22 }
```

References

Apel, S., Batory, D.S., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines - Concepts and Implementation. Springer. doi:10.1007/978-3-642-37521-7.

Astegher, M., Busetta, P., Perini, A., Susi, A., 2021. Specifying requirements for data collection and analysis in data-driven RE. A research preview, in: Dalpiaz, F., Spoletini, P. (Eds.), Requirements Engineering: Foundation for Software Quality - 27th International Working Conference, REFSQ 2021, Essen, Germany, April 12-15, 2021, Proceedings, Springer. pp. 182–188. doi:10.1007/978-3-030-73128-1_13.

Basili, V.R., Rombach, H.D., 1988. The tame project: Towards improvement-oriented software environments. IEEE Transactions on software engineering 14, 758–773.

Benavides, D., Segura, S., Cortés, A.R., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35, 615–636. doi:10.1016/j.is.2010.01.001.

Beuche, D., 2019. Industrial variant management with pure::variants, in: ACM International Conference Proceeding Series, Association for Computing Machinery, New York, New York, USA. pp. 1–3. doi:10.1145/3307630.3342391.

Clements, P., Northrop, L.M., 2002. Software product lines - practices and patterns. SEI series in software engineering, Addison-Wesley.

Dakkak, A., Bosch, J., Olsson, H.H., 2022a. Controlled continuous deployment: A case study from the telecommunications domain, in: ICSSP/ICGSE 2022: 16th International Conference on Software and System Processes and 17th International Conference on Global Software Engineering, Virtual Event / Pittsburgh, PA, USA, May 19 - 20, 2022, ACM. pp. 24–33. doi:10.1145/3529320.3529323.

Dakkak, A., Mattos, D.I., Bosch, J., 2021a. Perceived benefits of continuous deployment in software-intensive embedded systems, in: IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021, Madrid, Spain, July 12-16, 2021, IEEE. pp. 934–941. doi:10.1109/COMPSAC51774.2021.00126.

Dakkak, A., Munappy, A.R., Bosch, J., Olsson, H.H., 2022b. Customer support in the era of continuous deployment: A software-intensive embedded systems case study, in: 46th IEEE Annual Computers, Software, and Applications Conferenc, COMPSAC 2022, Los Alamitos, CA, USA, June 27 - July 1, 2022, IEEE. pp. 914–923. doi:10.1109/COMPSAC54236.2022.00143.

Dakkak, A., Zhang, H., Mattos, D.I., Bosch, J., Olsson, H.H., 2021b. Towards continuous data collection from in-service products: Exploring the relation between data dimensions and collection challenges, in: 28th Asia-Pacific Software Engineering Conference, APSEC 2021, Taipei, Taiwan, December 6-9, 2021, IEEE. pp. 243–252. doi:10.1109/APSEC53868.2021.00032.

Few, S., 2006. Information dashboard design: The effective visual communication of data. volume 2. O'reilly Sebastopol, CA.

Fitzgerald, B., Stol, K., 2017. Continuous software engineering: A roadmap and agenda. J. Syst. Softw. 123, 176–189. doi:10.1016/j.jss.2015.06.063.

Galindo, J.A., Benavides, D., Trinidad, P., Gutiérrez-Fernández, A.M., Ruiz-Cortés, A., 2019. Automated analysis of feature models: Quo vadis? Computing 101, 387–433.

Google, 2016. Google analytics — hit. URL: https://support.google.com/analytics/answer/6086082?hl=en.

Gregor, S., Kruse, L.C., Seidel, S., 2020. Research perspectives: The anatomy of a design principle. J. Assoc. Inf. Syst. 21, 2. URL: https://aisel.aisnet.org/jais/vol21/iss6/2.

Hellebrand, R., Schulze, M., Ryssel, U., 2017. Reverse engineering challenges of the feedback scenario in co-evolving product lines, in: Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume B, Sevilla, Spain, September 25-29, 2017, ACM. pp. 53–56. doi:10.1145/3109729.3109735.

Hoffmann, P., Spohrer, K., Heinzl, A., 2020. Analyzing Usage Data in Enterprise Cloud Software: An Action Design Research Approach. Springer International Publishing, Cham. pp. 257–274. doi:10.1007/978-3-030-45819-5_11.

Johanssen, J.O., Kleebaum, A., Bruegge, B., Paech, B., 2018. Feature crumbs: Adapting usage monitoring to continuous software engineering, in: Product-Focused Software Process Improvement - 19th International Conference, PROFES 2018, Wolfsburg, Germany, November 28-30, 2018, Proceedings, Springer. pp. 263–271. doi:10.1007/978-3-030-03673-7_19.

Johanssen, J.O., Kleebaum, A., Bruegge, B., Paech, B., 2019. How do practitioners capture and utilize user feedback during continuous software engineering?, in: 27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019, IEEE. pp. 153–164. doi:10.1109/RE.2019.00026.

Kontio, J., Bragge, J., Lehtola, L., 2008. The Focus Group Method as an Empirical Tool in Software Engineering. Springer. pp. 93–116. doi:https://doi.org/10.1007/978-1-84800-044-5_4.

Krueger, C.W., Clements, P., 2018. Feature-based systems and software product line engineering with gears from biglever, in: Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 2, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018, ACM. pp. 1–4. doi:10.1145/3236405.3236409.

Liang, W., Qian, W., Wu, Y., Peng, X., Zhao, W., 2015. Mining context-aware user requirements from crowd contributed mobile data, in: Proceedings of the 7th Asia-Pacific Symposium on Internetware, Internetware 2015, Wuhan, China, November 6, 2015, ACM. pp. 132–140. doi:10.1145/2875913.2875933.

López, L., Manzano, M., Gómez, C., Oriol, M., Farré, C., Franch, X., Martínez-Fernández, S., Vollmer, A.M., 2021. Qasd: A quality-aware strategic dashboard for supporting decision makers in agile software development. Sci. Comput. Program. 202, 102568. doi:10.1016/j.scico.2020.102568.

Maalej, W., Happel, H., Rashid, A., 2009. When users become collaborators: towards continuous and context-aware user input, in: Arora, S., Leavens, G.T. (Eds.), Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA, ACM. pp. 981–990. doi:10.1145/1639950.1640068.

Martínez-Fernández, S., Vollmer, A.M., Jedlitschka, A., Franch, X., López, L., Ram, P., Rodríguez, P., Aaramaa, S., Bagnato, A., Choras, M., Partanen, J., 2019. Continuously assessing and improving software quality with software analytics tools: A case study. IEEE Access 7, 68219–68239. doi:10.1109/ACCESS.2019.2917403.

Mattos, D.I., Bosch, J., Olsson, H.H., 2018. Challenges and strategies for undertaking continuous experimentation to embedded systems: Industry and research perspectives, in: Agile Processes in Software Engineering and Extreme Programming - 19th International Conference, XP 2018, Porto, Portugal, May 21-25, 2018, Proceedings, Springer. pp. 277–292. doi:10.1007/978-3-319-91602-6_20.

Maxwell, J., 1992. Understanding and validity in qualitative research. Harvard educational review 62, 279–301.

Medina, H., Díaz, O., Garmendia, X., 2022. Wacline: A software product line to harness heterogeneity in web annotation. SoftwareX 18, 101090. doi:https://doi.org/10.1016/j.softx.2022.101090.

Morgan, D.L., 1997. Focus Groups as Qualitative Research. Qualitative Research Methods, SAGE Publications, Thousand Oaks, CA. doi:10.4135/9781412984287.

Olsson, H.H., Bosch, J., 2013. Towards data-driven product development: A multiple case study on post-deployment data usage in software-intensive embedded systems, in: Fitzgerald, B., Conboy, K., Power, K., Valerdi, R., Morgan, L., Stol, K. (Eds.), Lean Enterprise Software and Systems - 4th International Conference, LESS 2013, Galway, Ireland, December 1-4, 2013, Proceedings, Springer. pp. 152–164. doi:10.1007/978-3-642-44930-7_10.

van Oordt, S., Guzman, E., 2021. On the role of user feedback in software evolution: a practitioners' perspective, in: 29th IEEE International Requirements Engineering Conference, RE 2021, Notre Dame, IN, USA, September 20-24, 2021, IEEE. pp. 221–232. doi:10.1109/RE51729.2021.00027.

Oriol, M., Stade, M.J.C., Fotrousi, F., Nadal, S., Varga, J., Seyff, N., Abelló, A., Franch, X., Marco, J., Schmidt, O., 2018. FAME: supporting continuous requirements elicitation by combining user feedback and monitoring, in: 26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018, IEEE Computer Society. pp. 217–227. doi:10.1109/RE.2018.00030.

Parker, A., Tritter, J., 2006. Focus group method and methodology: current practice and recent debate. International Journal of Research & Method in Education 29, 23–37. doi:10.1080/01406720500537304.

Plaza, B., 2011. Google analytics for measuring website performance. Tourism Management 32, 477–481.

Redman, T.C., 2013. Data quality management past, present, and future: Towards a management system for data, in: Sadiq, S.W. (Ed.), Handbook of Data Quality, Research and Practice. Springer, pp. 15–40. doi:10.1007/978-3-642-36257-6_2.

Schön, E., Thomaschewski, J., Escalona, M.J., 2017. Agile requirements engineering: A systematic literature review. Comput. Stand. Interfaces 49, 79–91. doi:10.1016/j.csi.2016.08.011.

Sein, M.K., Henfridsson, O., Purao, S., Rossi, M., Lindgren, R., 2011. Action design research. MIS Q. 35, 37–56. URL: http://misq.org/action-design-research.html.

Sjøberg, D.I.K., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M., Karahasanovic, A., Koren, E.F., Vokác, M., 2002. Conducting realistic experiments in software engineering, in: 2002 International Symposium on Empirical Software Engineering (ISESE 2002), 3-4 October 2002, Nara, Japan, IEEE Computer Society. pp. 17–26. doi:10.1109/ISESE.2002.1166921.

Turner, M., Kitchenham, B., Brereton, P., Charters, S., Budgen, D., 2010. Does the technology acceptance model predict actual use? a systematic literature review. Information and software technology 52, 463–479.

Ursachi, G., Horodnic, I.A., Zait, A., 2015. How reliable are measurement scales? external factors with indirect influence on reliability estimators. Procedia Economics and Finance 20, 679–686. doi:https://doi.org/10.1016/S2212-5671(15)00123-9. globalization and Higher Education in Economics and Business Administration - GEBA 2013.

W3Techs, 2022. Usage statistics of traffic analysis tools for websites. URL: https://w3techs.com/technologies/overview/traffic_analysis.

Wallace, L.G., Sheetz, S.D., 2014. The adoption of software measures: A technology acceptance model (tam) perspective. Information & Management 51, 249–259. doi:https://doi.org/10.1016/j.im.2013.12.003.

Wang, C., Daneva, M., van Sinderen, M., Liang, P., 2019. A systematic mapping study on crowdsourced requirements engineering using user feedback. J. Softw. Evol. Process. 31. doi:10.1002/smr.2199.

Wnuk, K., Regnell, B., Karlsson, L., 2009. What happened to our features? visualization and understanding of scope change dynamics in a large-scale industrial setting, in: RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009, IEEE Computer Society. pp. 89–98. doi:10.1109/RE.2009.32.

Xiao, J., Zeng, J., Yao, S., Cao, Y., Jiang, Y., Wang, W., 2021. Listening to the crowd for the change file localization of mobile apps, in: ACM ICEA '21: 2021 ACM International Conference on Intelligent Computing and its Emerging Applications, Jinan, China, December 28 - 29, 2022, ACM. pp. 71–76. doi:10.1145/3491396.3506530.