Empirical Software Engineering (2022) 27: 75
https://doi.org/10.1007/s10664-021-10101-6

Visualizing the customization endeavor in product-based-evolving software product lines: a case of action design research

Oscar Díaz[1] · Leticia Montalvillo[1] · Raul Medeiros[1] · Maider Azanza[1] · Thomas Fogdal[2]

Accepted: 6 December 2021 / Published online: 18 March 2022

Abstract
Software Product Lines (SPLs) aim at systematically reusing software assets, and deriving products (a.k.a., variants) out of those assets. However, it is not always possible to handle SPL evolution directly through these reusable assets. Time-to-market pressure, expedited bug fixes, or product specifics lead to the evolution to first happen at the product level, and to be later merged back into the SPL platform where the core assets reside. This is referred to as product-based evolution. In this scenario, deciding when and what should go into the next SPL release is far from trivial. Distinct questions arise. How much effort are developers spending on product customization? Which are the most customized core assets? To which extent is the core asset code being reused for a given product? We refer to this endeavor as *Customization Analysis*, i.e., understanding the functional increments in adjusting products from the last SPL platform release. The scale of the SPLs' code-base calls for customization analysis to be conducted through *Visual Analytics* tools. This work addresses the design principles for such tools through a joint effort between academia and industry, specifically, Danfoss Drives, a company division in charge of the P400 SPL. Accordingly, we adopt an *Action Design Research* approach where answers are sought by interacting with the practitioners in the studied situations. We contribute by providing informed goals for customization analysis as well as an intervention in terms of a visual analytics tool. We conclude by discussing to what extent this experience can be generalized to product-based evolving SPL organizations other than Danfoss Drives.

Keywords SPL evolution · Visual analytics · Code diffing · Action design research

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

This article belongs to the Topical Collection: *Open Science*

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

This paper has been awarded the Empirical Software Engineering (EMSE) open science badge

✉ Maider Azanza
maider.azanza@ehu.eus

Extended author information available on the last page of the article

1 Introduction

A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a *prescribed* way (Clements and Northrop 2002). Prescription is given in terms of both developing *for* reuse and developing *with* reuse. This results in the interplay of two distinct activities: (1) *Domain Engineering (DE)*, where the scope and variability of the system is defined and reusable assets are developed to conform the SPL platform; and (2) *Application Engineering (AE)*, where products (a.k.a. variants) are derived by selecting and resolving variability, i.e., by variability configuration (Pohl et al. 2005). To obtain the full benefits, AE should be limited to configuration without further modifying the derived products (Krueger 2006). However, this is not always possible. Time-to-market pressure, expedited bug fixes, or product specifics lead to development to first happen at the product level and to be later merged back into the SPL platform. Indeed, **product-based evolution** (a.k.a. variant-based evolution) is being acknowledged as a major strategy to drive the evolution of a product line (Krüger et al. 2020). Kruger et al. observe that this way of working switches the typical order of DE and AE, i.e., developers first customize core assets to product requirements (AE) to next consolidate the product upgrades into the platform (DE). The bottom line is that AE is no longer limited to mere configuration but it might also involve development. We refer to this practice as 'product customization' as opposed to 'product configuration', where AE is limited to selecting the features to be exhibited by the product with no development involved.

Product-based SPL evolution might result in tensions between the quality and reuse effectiveness required by DE, and the time-to-market and customer pressure faced by AE. SPL managers need to analyze whether efforts invested in product customization pay off in terms of better SPL scoping (i.e., deciding on the products, technical areas, and functionalities that a product line should support). Questions might arise about how much effort is dedicated to product customization; which are the most customized core assets; or to which extent is core-asset code being reused in a given product. We refer to this endeavor as **Customization Analysis**, i.e., understanding the functional increments in adjusting products from the last SPL platform release. Customization analysis can help SPL managers in different ways: (1) identify which product developments should be promoted to the core-asset base, (2) spot overloaded product teams with a heavy customization duty, (3) uncover indecisive or exploratory design where developers' hesitation is reflected in the volatility of the code, or (4), spot eventual merging issues when product developments are merged back into the core-asset base. Being able to quantify the customization activity might help managers take informed decisions about both the SPL's stability and the rearrangement of the SPL task force.

Despite the importance of these decisions, support for customization analysis is rather limited. We abound into this practice by making a case for the use of alluvial diagrams (a.k.a. Sankey diagrams) as a suitable visualization for customization analysis. Specifically, two research questions are tackled:

- *RQ1 (Problem Space):* Which are the information needs for customization analysis? How much time is needed to fulfill these information needs?
- *RQ2 (Solution Space):* Might alluvial diagrams be useful for supporting customization analysis visualization?

We address these questions through a joint effort between academia and industry, specifically, Danfoss Drives, a company division in charge of P400, a product line for the software embedded in frequency converters (Fogdal et al. 2016). On these grounds, we adopt an *Action Design Research (ADR)* approach (Sein et al. 2011). The objective of action research is to solve or at least explain, the problems of an analyzed situation (i.e., product customization) by the researchers interacting with the participants (i.e., P400 engineers) in the studied situations (i.e., customization analysis). Action research becomes action *design* research if the problem is tackled through an artifact (i.e., a new method or a new tool) whose design and evaluation is conducted within the organization (Sein et al. 2011). Accordingly, we develop a *Visual Analytics* tool for customization analysis. Akin to the ADR principles, we distill this experience in some general outcomes. By doing so, we aim at contributing to the two previous RQs:

– *RQ1*. We characterize customization analysis through a Goal-Question-Metric (GQM) model. Based on Danfoss Drives' practices, we introduce a set of analysis questions, and rate the importance and required time to answer such questions (Section 4).
– *RQ2*. We make a case for alluvial diagrams as an effective way to realize the previous GQM model. We flesh out this case through *CustomDIFF*, a visual analytics tool that uses *Git* as the SPL code repository, and *pure::variants* as the variability manager (Section 6).
– *RQ2*. We distill general principles from first empirical evidence through an expert evaluation at Danfoss Drives (Section 8).

This article is an extension of a REVE'17 workshop paper (Montalvillo et al. 2017), which is extended in three major aspects. First, we considerably expand the description about the phenomenon at hand: product customization. Second, it is methodologically sounder since it sticks to the *Design Activity Framework* proposed for visual analytics, and provides empirical evaluation. Third, this paper focuses on annotation-based SPLs using *pure::variants*, whereas the REVE paper tackled component-based SPLs using *FeatureHouse*. By conducting this research in close collaboration with an industrial partner, we hope to facilitate a valuable transfer into practice. We start by introducing the phenomenon under study: product customization.

2 The Phenomenon: Product Customization

Code development during AE (i.e., product customization) has been documented in distinct scenarios: to meet products' deadline and budget (Deelstra et al. 2005; Jensen 2007; Schackmann and Lichter 2006), to expedite bug fixes (Fogdal et al. 2016), to speed up unexpected functional changes in customer needs (Nagamine et al. 2016; Carbon et al. 2008; Iida et al. 2016), to decrease reusable asset complexity for single-product needs (Deelstra et al. 2005; Kircher and Hofman 2012; Bartholdt and Becker 2011), and, finally, in the transition to a fully-configured SPL, product specifics might remain in product teams (Kodama et al. 2014; Takebe et al. 2009). Even fully-configurable SPLs might reach the scale and complexity that make maintenance in short time spans infeasible. Hence, when organizations are faced with urgent customer or market requests, product-specific adjustments are realized first in the product to be later propagated to the SPL platform (Deelstra et al. 2005).

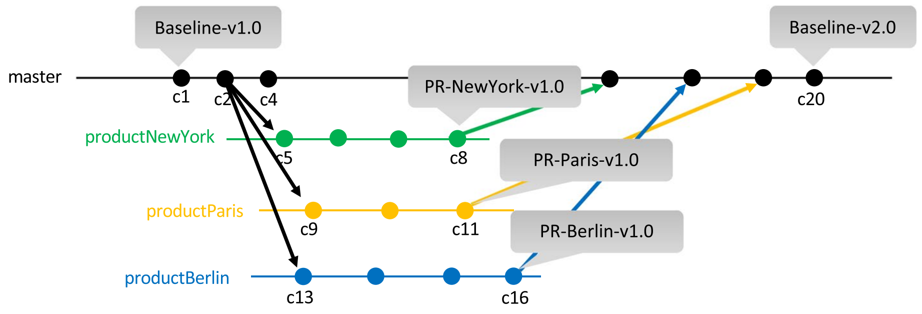


Fig. 1 *WeatherStationSPL* branching model: the *master* branch holds the core assets from where SPL products are branched off

Implementation wise, this is normally realized through a version control system. Here, a *master* branch keeps the core assets, i.e., those assets which are in the SPL baseline. During product customization, it is allowed for this *master* to be branched off to support (urgent) product specifics. This leads to a *grow-and-prune* branching model (Faust and Verhoef 2003). This model states that, during a time-lapse, quick reaction to changes often requires copying and specializing in the branches (grow) to be later cleaned up by refactoring and merged back into the *master* (prune). The notion of time-lapse is important since *product* branches are not intended to live for a long time but are to be periodically merged back into the *master*. In this way, product needs drive the evolution of the SPL.

As an example, consider the *WeatherStationSPL*, an SPL for building web-based applications for weather stations. We borrow this example from the instructional material provided by *pure::variants*.[1] Let us suppose that this SPL holds a baseline release *Baseline-v1.0,* that accounts for seven features, clustered around three *parent* features,[2] namely: *Sensors* that encompasses *AirPressure*, *Temperature*, and *WindSpeed*; *Warnings* that comprises *Gale* and *Heat*; and *Languages* that is the *parent* feature for *English* and *German*. Let us imagine that some urgent customization needs arise that prevent developers from waiting until the next platform release. This causes *Baseline-1.0* to be branched off into three *product* branches: *PR-NewYork*, *PR-Paris*, and *PR-Berlin* (see Fig. 1). This unleashes the grow-and-prune process:

- *Grow-Customization.* AE adjusts core assets to product specifics, potentially evolving the product through different versions (e.g., *PR-Paris-v1.0*).
- *Prune-Consolidation.* Eventually, DE gets integration requests from AE. Missing to reintegrate these product variants back to the *master* risks product-line engineering becoming clone&own development (Krüger and Berger 2020).

It might happen that the customization might be of interest but not yet mature enough to be offered to products other than the product that hosts it. Here, DE developers might integrate the adjustment into the platform using *spurious features*, i.e., transient features that might be exclusively used for the driving product. This allows AE developers to generate again

[1] http://www.pure-systems.com/products/pure-variants-9.html

[2] In *pure::variants* terminology, a *parent* feature serves to aggregate semantically related features

this product using traditional configuration mechanisms.[3] This guarantees that the platform is under control, and application projects may maintain their independence by providing product-specific artifacts as new features. After a modification has been evaluated, a judgment will be made as to whether the change should be applied to additional products and so incorporated into the core assets (Fogdal et al. 2016). For Danfoss Drives, decision is taken by the *Change Control Board* that includes domain experts but also application engineers who were involved in the customization in the first place. The choice is not always easy, as evidenced by the so-called *configuration oscillation* phenomenon (Faust and Verhoef 2003). Here, engineers can be hesitant about which new assets should be promoted to the platform.

The bottom line is that the Change Control Board regulates the tempos of this grow-and-prune process. In this setting, customization analysis helps the Change Control Board identify if the customization endeavor matches the plan by correlating where the customization is occurring with where the customization is planned to occur. A lot of customization happening in unplanned areas may be an indicator of problematic code, that is, a code that continuously requires patching. In general, customization analysis might help foresee distinct problematic scenarios, namely:

– product units that are under heavy customer pressure in terms of specifics that need to be accounted for. This might lead to reinforcing these units or to rescheduling customer petitions,
– stable features that are passed (almost) untouched when deployed in distinct products. This showcases mature features. This scenario might point to needing to consider deviating programming resources to other units,
– unstable features which might require additional refactoring, and testing efforts to cope with emerging scenarios coming from customer petitions. This might lead to reinforcing the domain engineers in charge.

Tracking these scenarios directly from code is time consuming and error prone. Tools are needed that abstract from the code-base. We tackle this challenge as a joint effort between academia and industry. On these premises, we decided to follow Action Design Research as our research methodology (Sein et al. 2011).

3 A Brief on Action Design Research

Sein et al. define Action Design Research (ADR) as a research method for generating prescriptive design knowledge through building and evaluating IT ensemble artifacts in an organizational setting (Sein et al. 2011, p. 40). A key insight is the role played by the organization (i.e., Danfoss) in driving and shaping the design knowledge that ends up being instantiated in the IT artifact (i.e., the *CustomDIFF* visualization tool). Hence, the term *ensemble artifact* denotes the artifact taking its full meaning in conjunction with the context where it displays its utility (i.e., *CustomDIFF* reflects the practice and brings utility to Danfoss). Therefore, ADR conceives artifact design as a result of a

[3] Notice that *spurious features* are transient. They are in trial for a period of time until they are finally integrated (and hence available for other products of the SPL) or discarded.

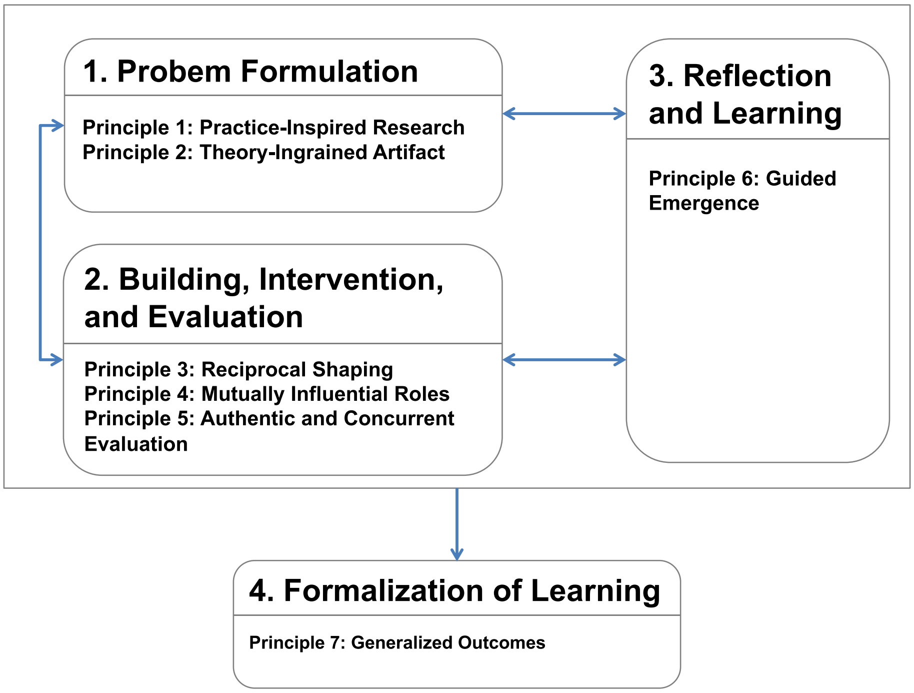

1. Probem Formulation
Principle 1: Practice-Inspired Research
Principle 2: Theory-Ingrained Artifact
2. Building, Intervention, and Evaluation
Principle 3: Reciprocal Shaping
Principle 4: Mutually Influential Roles
Principle 5: Authentic and Concurrent Evaluation
3. Reflection and Learning
Principle 6: Guided Emergence
4. Formalization of Learning
Principle 7: Generalized Outcomes

Fig. 2 The ADR Method: the first three stages conform an iterative cycle where tasks are intermingled till distilled into the final learnings at the end of the project (taken from Sein et al. 2011). Arrows stand for influential flows

researcher-practitioner collaboration within an organization. Figure 2 reproduces ADR stages and principles (Sein et al. 2011).

Problem Formulation. The first stage is triggered by a problem encountered in practice or predicted by researchers. It serves as a catalyst for developing a research strategy. This stage draws on two principles: *Practice-Inspired Research* and *Theory-Ingrained Artifact* (Sein et al. 2011). The former emphasizes viewing organization problems as knowledge-creation opportunities. The second principle highlights that the intervention (e.g., the IT artifact) is to be informed by theories, existing knowledge that grounds design decisions.

Building, Intervention, and Evaluation. This stage builds upon the problem framing and theoretical premises adopted in stage one. These premises provide a platform for generating the initial design of the IT artifact. From here on, the IT artifact is further shaped by organizational use and subsequent design cycles (Sein et al. 2011). Or using Sein et al's principles: reciprocal shaping (i.e., the IT artifact and the organization feedback each other: prototypes serve to profile the interpretation of the organizational environment that help a better fit in subsequent versions), mutually influential roles (i.e., researchers and practitioners bring complementary insights), and authentic and concurrent evaluation (i.e., authenticity is a more crucial element for ADR than controlled conditions, thus assessment should take place within the company and throughout the research).

Reflection and Learning. ADR involves more than merely solving a problem to an organization. To guarantee that contributions to knowledge are made, conscious reflection on the problem framing, theories adopted, and the emerging IT artifact are critical. The principle is termed as *guided emergence* where 'emergence' captures this notion of unanticipated consequences that arise during the intervention in the organization and to which researchers should be sensitive to (Sein et al. 2011).

Formulation of Learning. At this point, we reach an artifact that brings with it some premises about the problem framing and the organization setting (i.e., an *ensemble artifact*). It represents a solution to a problem. Both can be generalized. Sein et al. suggest three levels for this effort: (1) generalization of the problem instance, (2) generalization of the solution instance, and (3) generalization of mechanisms through design principles. Design principles abstract away from the specific IT implementation into the abstract mechanisms that brought the utility, and underlie the solution.

The rest of the paper is structured along these stages.

4 Problem Formulation

Problem formulation draws on two principles: practice-inspired research and theory-ingrained artifact. The former reflects the premise that IT artifacts are ensembles shaped by the organizational context (Sein et al. 2011). Therefore, it is most important to describe the organization whose practices and characteristics will inform the artifact design. The second principle highlights that ADR does not stop at identifying a problem, but provides an intervention to alleviate the problem. This intervention should be informed by existing theories. This section sets this research's problem along with these two principles.

4.1 Practice-Inspired Research

This research builds upon the fifteen-year experience of Danfoss Drives, a company division in charge of P400, a product line for the software embedded in frequency converters (Fogdal et al. 2016). P400 is a member of the *SPLC's Product Line Hall of Fame*.[4] Table 1 characterizes P400 in terms three main contextual dimensions: the stakeholders, the complexity of the task (i.e., customization analysis), and the setting that frames the problem (i.e., the technical infrastructure and features of the SPL that might be relevant for the problem at hand).

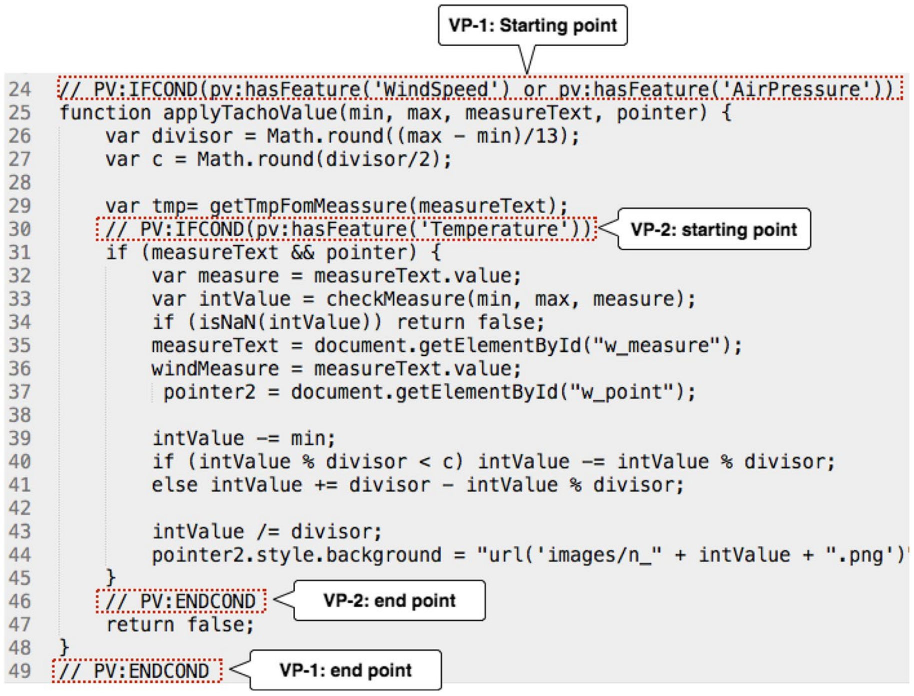
Danfoss follows an *Annotation-Based Approach* to P400 definition. This implies that variations are supported through pre-compilation directives. A directive states when a block code is to be included in the final product based on the presence or absence of a feature selection at configuration time. In *pure::variants*, these directives start with an opening directive *//PV:IFCOND* and end with a closing directive *//PV:ENDCOND*. Figure 3 shows an example. The snippet illustrates two variation points, i.e., VP-1 and VP-2 that correspond to two *ifdef* blocks. In the example, VP-1 comprises lines 24 to 49, whereas VP-2 expands along lines 30 to 46.

Danfoss follows a *Product-Based Approach* to P400 evolution This implies that DE and AE co-exist not only at the onset but throughout the SPL life-cycle. We aim at

[4] https://splc.net/fame.html

Table 1 Contextual characterization

Stakeholder	User	Role	Change Control Board (CCB)
			Developers
		Experience	7-year average SPL experience
Task	Frequency	2–4 weeks	
	Complexity	Feature Tangling (max.)	11
		Feature Scattering (max.)	21
		No. Products (max.)	up to 20 products per platform release
		No. Features (approx.)	10–20 features per platform release
		Code churn (approx.)	2.7K LOC
Setting	Technical environment	Programming language	C++
		Branching strategy	Grow-and-prune
		Variability manager	pure::variants
	SPL Attributes	Lifespan	+ 15 years
		Size (approx.)	800 features & 20 products
		Domain	Embedded systems
		Variability model	Annotation-based



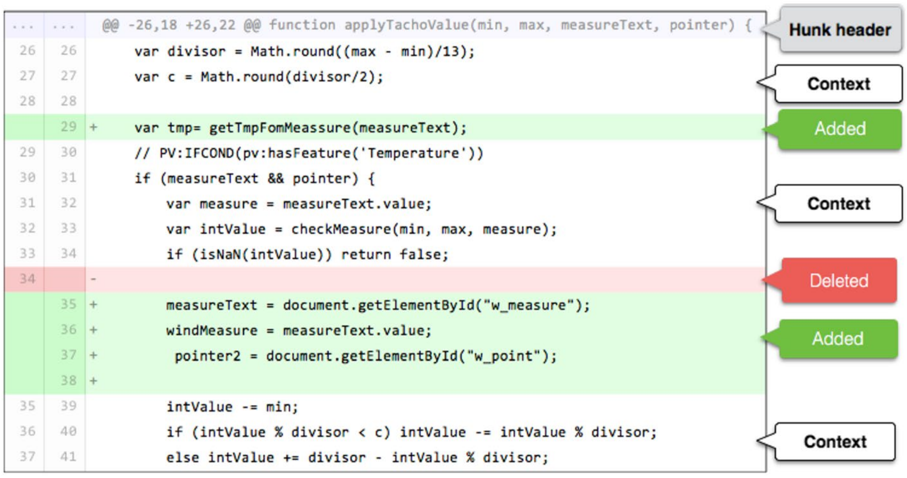
```
// PV:IFCOND(pv:hasFeature('WindSpeed') or pv:hasFeature('AirPressure'))
function applyTachoValue(min, max, measureText, pointer) {
    var divisor = Math.round((max - min)/13);
    var c = Math.round(divisor/2);

    var tmp= getTmpFomMeassure(measureText);
    // PV:IFCOND(pv:hasFeature('Temperature'))
    if (measureText && pointer) {
        var measure = measureText.value;
        var intValue = checkMeasure(min, max, measure);
        if (isNaN(intValue)) return false;
        measureText = document.getElementById("w_measure");
        windMeasure = measureText.value;
         pointer2 = document.getElementById("w_point");

        intValue -= min;
        if (intValue % divisor < c) intValue -= intValue % divisor;
        else intValue += divisor - intValue % divisor;

        intValue /= divisor;
        pointer2.style.background = "url('images/n_" + intValue + ".png')
    }
    // PV:ENDCOND
    return false;
}
// PV:ENDCOND
```

Fig. 3 Variation Points for *Sensors.js* at Baseline-v1.0. VP1 applies when either *WindSpeed* or *AirPressure* are selected. VP2 applies for *Temperature*. Notice how VP2 is scoped within VP1



```
@@ -26,18 +26,22 @@ function applyTachoValue(min, max, measureText, pointer) {
     var divisor = Math.round((max - min)/13);
     var c = Math.round(divisor/2);

+    var tmp= getTmpFomMeassure(measureText);
     // PV:IFCOND(pv:hasFeature('Temperature'))
     if (measureText && pointer) {
         var measure = measureText.value;
         var intValue = checkMeasure(min, max, measure);
         if (isNaN(intValue)) return false;
-
+        measureText = document.getElementById("w_measure");
+        windMeasure = measureText.value;
+         pointer2 = document.getElementById("w_point");
+
         intValue -= min;
         if (intValue % divisor < c) intValue -= intValue % divisor;
         else intValue += divisor - intValue % divisor;
```

Fig. 4 Visualizing code churn: diffing *sensors.js* between the copy at the *master* branch and the customized copy at the *productBerlin* branch

understanding how DE-AE fluctuates throughout this life-cycle. Here, finding out customization endeavors involves looking at the differences between core assets (kept in the *master* branch) and the namesake assets once customized by the product (kept in the *product* branches). Differences between branches are traditionally spotted through file diffing: *DIFF(C0.file, C1.file)*. Back to the *WeatherStationSPL* example, Fig. 4 illustrates the case for *sensors.js*, using the unified format for *diff* display (van van Rossum 2018). This figure illustrates the customization diffing in terms of LOC (Lines Of Code) being changed. For each change hunk, the outcome indicates: the hunk header (i.e., starting and ending line numbers together with the heading of the function the change hunk is part of), the added lines (denoted by a plus sign with a greenish background), the deleted lines (denoted by a minus sign with a reddish background), and the context (i.e., the three nearest unchanged lines that precede and follow the change). The latter is especially important since it provides the context in which the change happened. However, *sensors.js* is just one of the thirty files the *WeatherStationSPL* encompasses. And these thirty files might potentially suffer changes by any of the three products. This implies 30×3 potential DIFFs. Now move to Danfoss Drives.

Danfoss faces scalability issues when conducting Customization analysis for P400. P400 holds over 10,000 core assets and 20 products. Though only a fraction of these assets needs to be upgraded by a limited number of products *in the interim that goes between two SPL releases*, this number is still high enough to be handled through traditional DIFF utilities. The complexity of conducting customization analysis is felt to be proportional to the number of products, features and LOC that are affected in the interim between two SPL releases. In addition, this complexity might also be impacted by the tangling degree and the scattering degree that the SPL exhibits. Table 1 collects some figures about these concerns after the work of Zhang et al. (2013).

For single-off development, traditional DIFF utilities include Microsoft's *Azure DevOps Server*, and *GitHub's* code frequency graph (see Fig. 5). Here, code churn is aggregated in terms of modules like files, package and so on. Yet, these units of aggregation might not fit

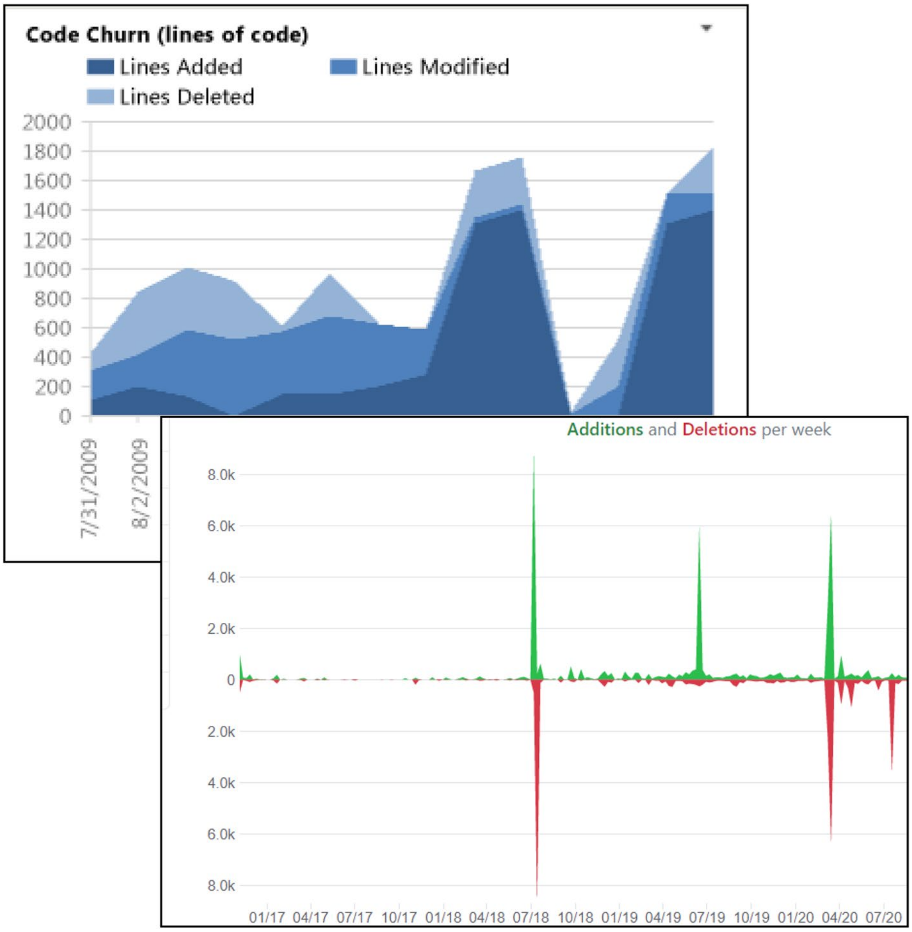


Fig. 5 Code Churn visualization: (Left) Microsoft's Azure opts for a chronological plot where different color shades account for lines added, deleted or modified throughout; (Right) *GitHub* reflects code additions (above) and code deletions (below) along with a common axis in a weekly basis

the abstractions SPL analysts think about. Therefore, in adopting traditional DIFF utilities, customization analysis faces two main issues:

- *Scalability.* Traditional DIFF utilities might not scale to the myriad of artifacts that SPLs exhibit,
- *Abstraction.* Traditional DIFF aggregation is conducted in terms of files while features are crosscuts that need to be abstracted out of a set variation points spread around different files.

The problem can then be stated in terms of *current DIFF tools not being suitable for accounting for the Change Control Board' information needs*. To provide some empirical evidence about this problem, we conducted a survey among Danfoss' engineers about what would be the time it would take them to answer some customization-analysis questions using traditional DIFF utilities. Figure 6 depicts the results. Though

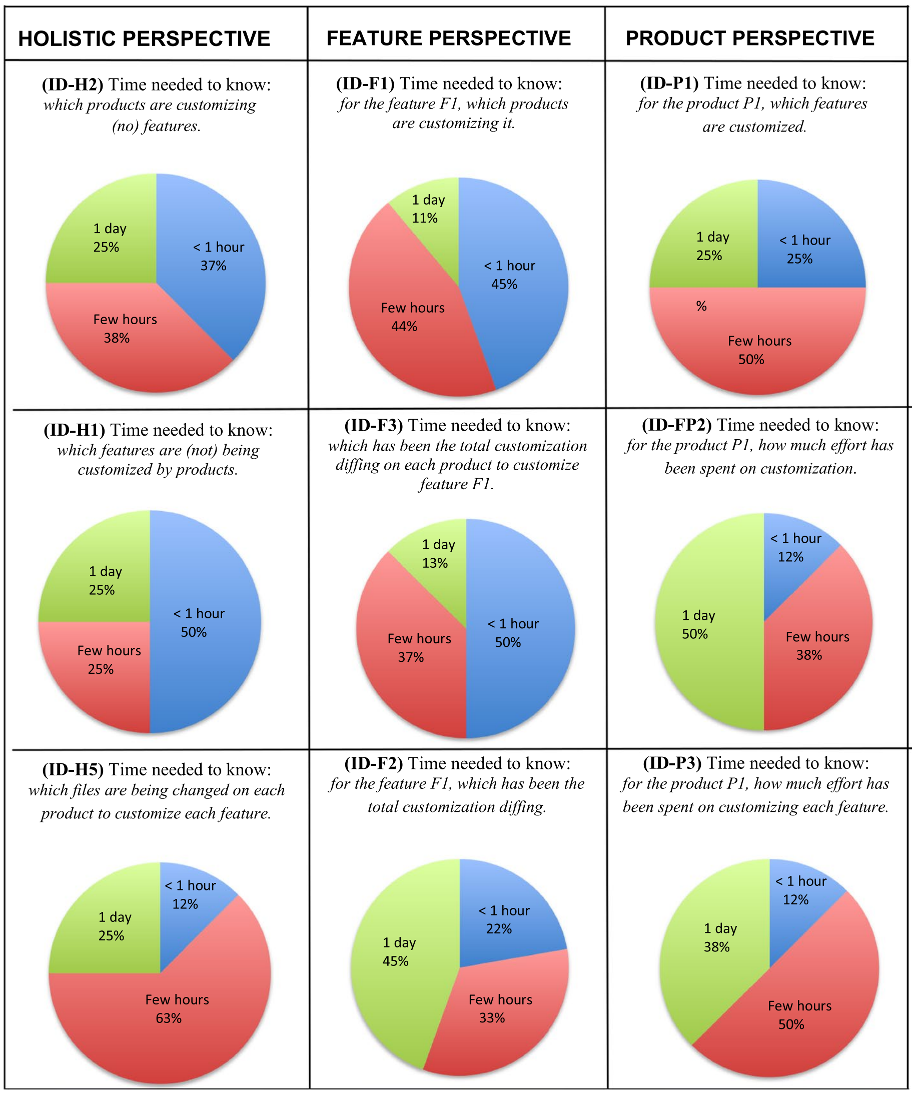


Fig. 6 Time effort in conducting distinct Customization Analysis questions. IDs help link to Table 2

the rationales for the questions are not provided until Section 5, this figure evidences that some questions might require a few hours to be answered. And this is just for one question, let alone a full analysis that might well involve several questions. This is especially so for the holistic perspective and the product perspective (see Fig. 6). Here, answering most of the questions required a few hours. This time effort is what motivates this research in the first place.

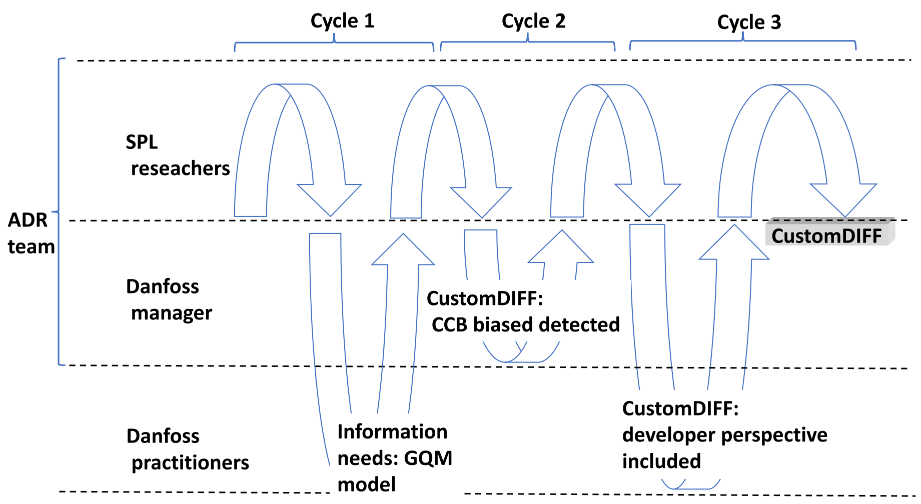

Cycle 1
Cycle 2
Cycle 3
ADR team
SPL reseachers
Danfoss manager
Danfoss practitioners
CustomDIFF
CustomDIFF: CCB biased detected
Information needs: GQM model
CustomDIFF: developer perspective included

Fig. 7 Evolution of the *CustomDIFF* project. Y axis stands for the team members. X axis stands for the evolution in time along with the three main cycles that output distinct artifacts (i.e., the GQM model, the developer perspective, the *CustomDIFF* prototype)

4.2 Theory-Ingrained Artifact

The previous section makes the case for current DIFF tools not scaling up to SPL analysis needs. After all, DIFF tools are not designed for analysis of product-based evolving SPLs but for code peering in single-off development. Therefore, the challenge is not so much about the metric itself (i.e., code churn) but providing the right abstraction at the scale of SPLs. To this end, we resort to theory on Visual Analytics to inform our intervention.

Visual Analytics is defined as the science of analytical reasoning facilitated by interactive visual interfaces (Cook and Thomas 2005). The basic idea is to visually represent the data to allow the human to directly interact with the information, to gain insights, and to ultimately make optimal decisions. Broadly speaking, Visual Analytics helps create a path from data to decision. In this process, visualization plays a twofold role (Reddivari et al. 2014):

- as an abstraction means, by highlighting certain constructs and relationships while ignoring others;
- as an interaction medium, by supporting the workflows for decision making.

By introducing Visual Analytics, decision makers can focus their full cognitive and perceptual attention on visualization-enabled analytical reasoning while taking advantage of automatic data processing techniques. The *Design Activity Framework* is a process model for visualization design (McKenna et al. 2014). Once we are aware of the problem and equipped with the theory, we can now move to the next stage: *Building, Intervention, and Evaluation*. Figure 7 depicts the three main cycles that ended up in CustomDIFF, the Visual Analytics tool. Next, we abound in each of these cycles.

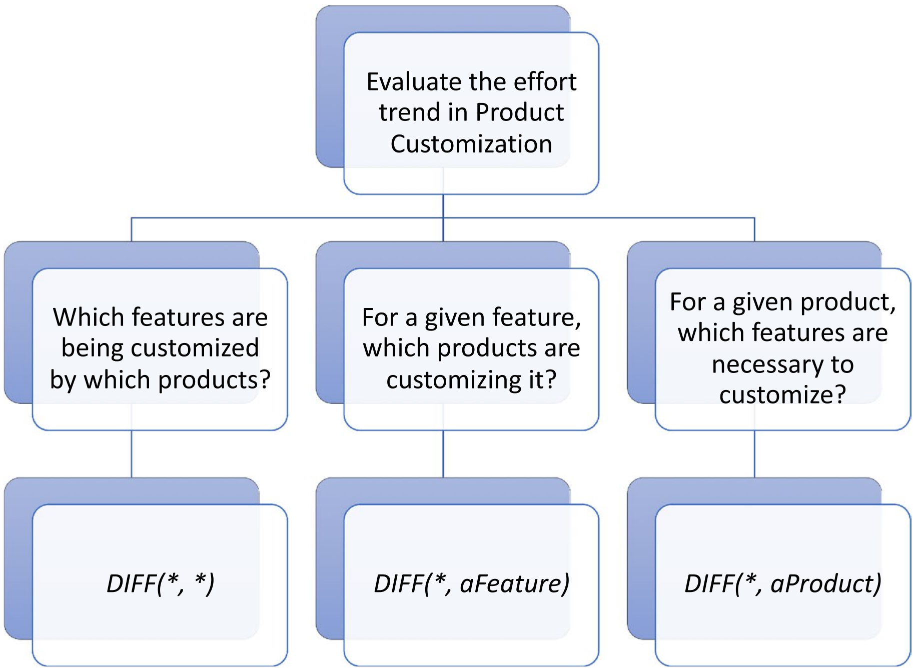

Evaluate the effort trend in Product Customization
Which features are being customized by which products?
For a given feature, which products are customizing it?
For a given product, which features are necessary to customize?
DIFF(*, *)
DIFF(*, aFeature)
DIFF(*, aProduct)

Fig. 8 The GQM model

5 Building, Intervention, and Evaluation: Cycle 1

So far, we have identified information needs whose fulfillment can hardly be satisfied with current tools. Our hypothesis is that dedicated Visual Analytics can improve this situation. Specifically, we adopt the Design Activity Framework framework to inform our building[5] (McKenna et al. 2014).

5.1 Building & Intervention

The first step is *understand* i.e., grasping the problem domain and target users. This activity aims at acquiring knowledge about the phenomenon of interest (i.e., customization analysis), domain-specific questions, and the types of measurements to appreciate this phenomenon (Reddivari et al. 2014). To this end, we resort to a *Goal-Question-Metric (GQM)* model (Basili et al. 1994) (see Fig. 8).

Goal. Following Basili et al.'s recommendations (Basili et al. 1994), we state our goal as follows: evaluate the effort trend (issue) in product customization (object) from

[5] This framework encompasses the following steps: understand (i.e., grasping the problem domain and target users), ideate (i.e., generating ideas for supporting the understand outcomes), make (i.e., concretizing ideas into tangible prototypes) and deploy (i.e., bringing a prototype into effective action in a real-world setting to support the target users' work).

Table 2 Validating the GQM. Rating the importance of information needs using a LIKERT scale from 1 (Not Important) to 5 (Very Important)

Question ID		Likert scale					Avg.
		1	2	3	4	5	
----------	Holistic perspective: I consider important to know ...						
H1	... which features are (not) being customized by products	0	1	3	3	1	3.5
H2	... which products are customizing (no) features	0	1	2	3	2	3.75
H3	... how much effort (i.e., code churn) has been spent on customizing each feature, in total, no matter the product	2	1	4	0	1	2.87
H4	... how much effort (i.e., code churn) each product is spending on customizing each feature	2	2	2	1	1	2.62
H5	...which files are being changed on each product to customize each feature	2	0	2	4	0	3
----------	Feature perspective: for feature F1, I consider important to know ...						
F1	... which products are customizing it	0	0	1	3	3	4
F2	... which has been the total customization diffing (i.e., code-churn aggregate)	2	1	3	1	1	2.75
F3	... which has been the total customization diffing (i.e., code-churn aggregate), broken down by product	2	1	3	1	1	2.75
F4	... which files have been changed	1	2	1	1	3	3.37
----------	Product perspective: for product P1, I consider important to know ...						
P1	... which features are customized	0	1	1	2	4	4.12
P2	... how much effort (i.e. code churn) has been spent on customization, no matter the feature	2	1	2	2	1	2.87
P3	... how much effort (i.e. code churn) has been spent on customizing each feature	2	1	3	1	1	2.75
P4	... which files have been changed	2	1	3	1	1	2.75

the Change Control Board perspective (viewpoint). Our goal is to trace how much effort is being put into product customization.

Question. Questions are posed to characterize the way the assessment/achievement of a specific goal is going to be performed based on some characterized model (Basili et al. 1994). For this model, *feature* and *product* emerge as natural constructs. Indeed, SPL releases, product configuration, programming assignments, or work positions are commonly described in terms of features and products. Using the notions of 'feature' and 'product', we can arrange the questions along with three distinct scopes (refer to Table 2 for the full list of questions):

- *the feature-focused scope,* e.g., for a given feature, which products are customizing it,
- *the product-focused scope,* e.g., for a given product, which features have been customized; if a product has made no change, it is not considered.
- *the holistic scope,* e.g., which is the whole customization effort from the last platform release,

Measurement. Once the questions have been developed, we proceed by associating the questions with appropriate metrics. Factors to be considered for this selection include (Basili et al. 1994): the quantity and quality of the existing data, and the maturity of the measurements. On these grounds, we resort to code churn, a well-established metric to assess evolution in one-off development (Ajila and Dumitrescu 2007; Hall and Munson 2000).

Code churn is a popular measure to inform about the rate at which the code evolves (Khoshgoftaar and Szabo 1994). The churn for a file over a specified period is computed as: $[LinesAdded] + [LinesDeleted] + [LinesModified]$ (Faragó et al. 2015). Traditionally, this is achieved by diffing files (Schulze et al. 2016): $DIFF(aFile_{t1}, aFile_{t0})$ displays the code churn for file *aFile* in the interval *[t0,t1]*. In single-off development, code churn is being extensively used for defect prediction (Nagappan and Ball 2005), assess code erosion (Ohlsson et al. 1999), or detect code volatility (Faragó et al. 2015).

Moving back to SPLs, code churn could also be a valuable metric for customization analysis: $DIFF(aFile_{t1}, aFile_{t0})$ where *t0* stands from the time where the *product* branch is generated out of the *master*'s, and *t1* corresponds to a time before the *product* branch is merged back to the *master*'s. Though the notion of churn might be appropriate, the object of the churn is not. Rather than files, the GQM's questions are posed in terms of 'features' and 'products'. Needed are mechanisms that move from file-based diffing to higher abstraction terms. Specifically, consider DIFF, a function that returns the code churn for *aFile* as it is kept in the *master* branch (i.e., *aFile.core*) *vs.* how it has been customized in a product *branch* (*aFile.aProduct*). Rather than *DIFF(aFile.core, aFile.aProduct*, we long for *DIFF(aFeature.core, aFeature.aProduct)* utilities that abstract out dozens of *DIFF(aFile, aFile)* for those *aFiles* that realize *aFeature* as it is being customized for *aProduct*. That is, *DIFF(aFeature.core, aFeature.aProduct)* outputs the code-churn aggregate of customizing *aFeature* for *aProduct*, no matter the files the *aFeature* is spread over.

On these grounds, we could define three sort of aggregates to measure the customization diffing (see Fig. 8), namely: (1) feature-focused DIFF (i.e., *DIFF(aFeature, *))* where *aFeature* stands for the set of *aFile* realizing this feature, and '*' indicates no matter the *product* branch; (2) product-focused DIFF (i.e., *DIFF(*, aProduct)*) where *aProduct* stands for the set of *aFiles* realizing this product, and '*' indicates no matter the feature; and (3), holistic DIFF (i.e., *DIFF(*, *))* where '*' corresponds for all *aFile* no matter the feature nor the product. These aggregates compute the sum of code churn for the underlying involved files.

5.2 Evaluation

ADR promotes continuous assessment of the intervention (principle 5: authentic and concurrent evaluation) (Sein et al. 2011). The intervention is not only the IT artifact but also the design principles this artifact realizes. This includes the information needs the IT artifact aims to fulfill. This section provides a first validation of the GQM model.

Participants. Participants were selected who had at least one-year experience on Danfoss Drives. Among the eight participants that took part in this evaluation, three had 10 years of experience while the other five accounted for 9, 7, 6, 3, and 1 year of experience, respectively.

Process. A questionnaire was prepared to assess the relevance of the above mentioned GQM model in terms of the importance given to the different questions raised by it. Face validity was conducted, that is, we checked whether the questionnaire seemed to correspond to the GQM model. A first draft of the questionnaire was prepared by the authors and next, the questionnaire was delivered to this paper's Danfoss Drives' author. Based on his comments, some amendments were added to clarify the purpose and adapt the terminology to that of the practitioners. Next, practitioners were requested to indicate the importance given to each question using a LIKERT scale from 1 (Not Important) to 5 (Very Important).

Results. Table 2 shows the results. Some conclusions can be drawn from them:

- Perspective wise, both feature-focused and product-focused are similarly rated. The highest rated questions are "for the feature F1, which products are customizing it" (avg. 4) and its sibling, i.e., "for the product P1, which are the features being customized" (avg. 4.12). The lowest rated questions correspond to the fine-grained holistic perspective. This might be due to this information being better captured at either the feature perspective or the product perspective.
- Aggregation-level wise, quite an unexpected result: intermediary aggregates were not prioritized. When pondering analysis needs, participants seem to favor either a general overview of the customization diffing or, instead, being able to dive into the specifics.

The bottom line is that all questions rate above 2.5, with four questions going beyond 3.5 (i.e., above "Moderately Important"). This provides first evidence about the interest in customization analysis. Yet, Visual Analytics does not stop at identifying the right data. Making better decisions also depends on the ability to understand and communicate adequately the measurement to the decision-makers. This moves us to the next cycle.

6 Building, Intervention, and Evaluation: Cycle 2

The basic idea is to visually represent the information, allowing humans to interact directly with such information, to gain insight, to draw conclusions, and to ultimately make better decisions. Figure 9 depicts the main ingredients of Visual Analytics tools (Reddivari et al. 2014). For our purposes, the *Data* is kept in a Git repository. This data corresponds to code files whose versions are arranged w.r.t the Git version control system model. This code is

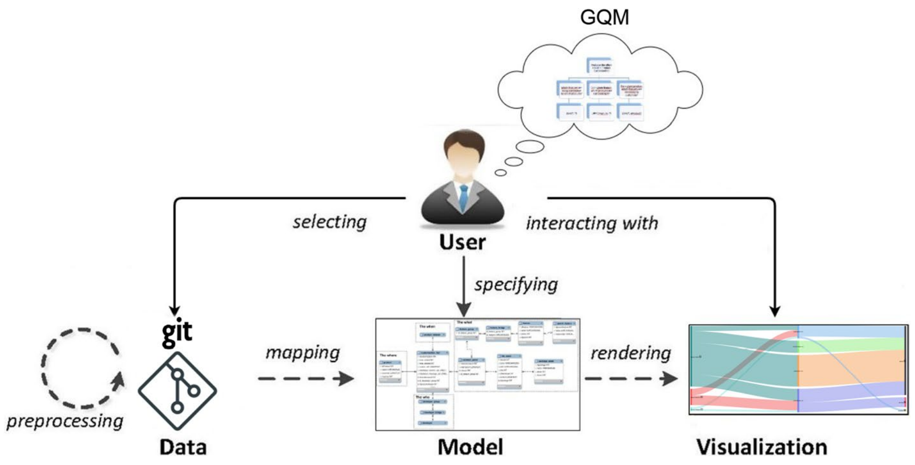


Fig. 9 Visual analytics main interactions (adapted from Reddivari et al. (2014))

pre-processed to distill the metrics to be used to assess the customization effort (i.e., code churn). As a result, code churn is obtained and described along with a *Model* that serves the information needs which were identified in the GQM analysis. This Model ends up being realized through a database that acts as the back-end for a Web application that supports the *Visualization* strategy. Main architectural components include: a mining component that extracts data from Git repositories; a database that holds mined data along with the data Model; and a front-end component that queries the database and display results using appropriate visualization means.

This section instantiates this architecture for *CustomDIFF*, a web-based tool for customization analysis. *CustomDIFF* uses *pure::variants* (Pure-Systems 2018) and *Git* as the variability management tool and version control system, respectively. Three additional resources are made available:

– an interactive online version of *CustomDIFF* which the reader is encouraged to access: http://customdiff.onekin.org/,
– a video describing *CustomDIFF* (6'): https://vimeo.com/577936099,
– a *Zenodo* replication package for the *CustomDIFF* implementation: https://doi.org/10.5281/zenodo.5728000

6.1 Building & Intervention

6.1.1 The Model

Dimensional Modeling is a data structure technique that is specifically designed for data storage when used for decision taking. This implies a sharp distinction between two sort of tables: "fact" and "dimensions" (see Fig. 10). The "fact" table collects the events of the phenomenon under study. For our purposes, the phenomenon under study is "product customization". We consider an event of this phenomenon to occur when it happens the consecutive deletion/addition of code churn for a file. Each code churn gives rise

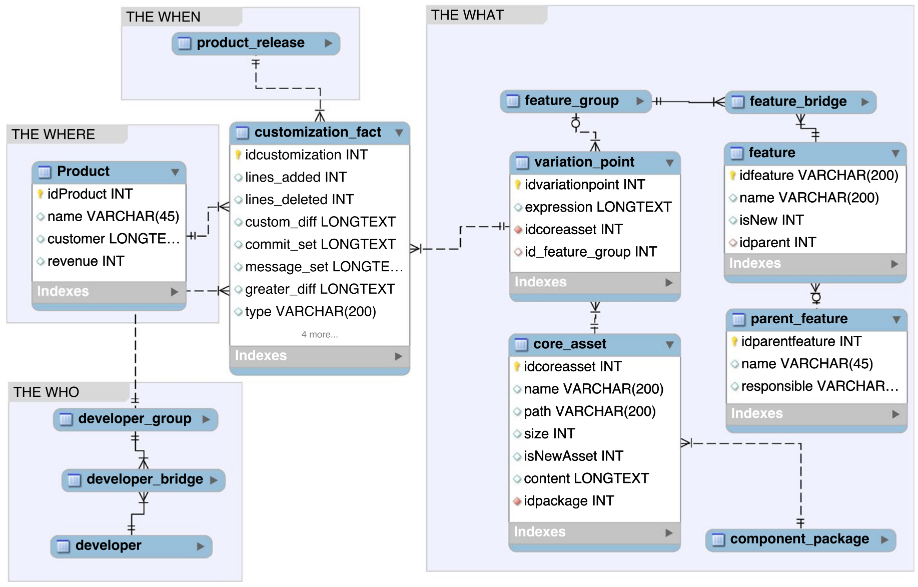


Fig. 10 *CustomDIFF's* data model

to a fact tuple. Fact properties include: the number of lines added, the number of lines deleted or the actual code being changed (*custom_diff*) in a customization episode.

Facts are the finest grain of the customization endeavor. Obtaining a higher perspective of the customization endeavor requires these facts to be aggregated along different *dimensions*: "the what" (i.e., the variation point being affected by the customization), "the where" (i.e., the product in which the customization occurred), "the when" (i.e., the time of the product release), and "the who" (i.e., developers who conducted the customization).

Figure 10 displays the database schema. This schema will be instantiated from the *codebase* of the SPL at hand. The SPL codebase is held in a *Git* repository where the *master* branch contains the core-asset baseline while products branch off the *master*. This moves us to the next sub-section.

6.1.2 The Data

In data warehousing, ETL, which stands for extract, transform and load, is a data integration process whereby data is extracted from data sources (that are not optimized for analytics), and moved to a central host (which is). In our setting, ETL mines the SPL's Git repository, runs the corresponding diffing, and populates the tables. Specifically, facts (i.e., *customization_fact* tuples) are obtained by working out a DIFF between the namesake artifacts of the *master* branch and the *product* branches. For the DIFF depicted in Fig. 4, two facts would be obtained. Fact #1 would stand for the changes introduced in line 29, whereas Fact #2 would correspond to those changes introduced in lines 34–38. Details about the mining algorithm are provided in the Appendix. A related approach is described by Zhang et al. (2013).

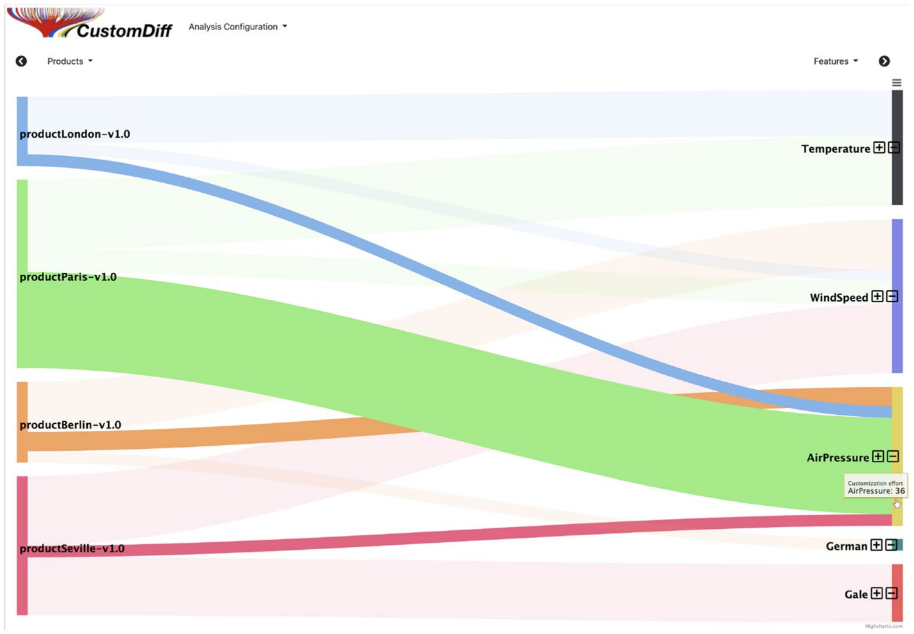


Fig. 11 Customization diffing are visualized as flows from products (as customization emitters) to features (as customization sinks). For example: *ProductLondon-v1.0* requires customization for three features: *WindSpeed*, *AirPressure* & *Temperature*

6.1.3 The Visualization

This section makes a case for the use of alluvial diagrams (a.k.a. Sankey diagrams) as a suitable visualization for customization analysis, measured in terms code churn from diffing. Broadly, we advocate to ideate the notion of customization diffing as an energy flow. The metaphor of energy flow is used in domains familiar with the dynamics of mass flow (e.g., energy, capital, transportation). Examples can be found for resource use (Lupton and Allwood 2017), energy flow (Schmidt 2008; Subramanyam et al. 2015) or material flows (Schmidt 2008), to name a few. For our purposes, the flow stands for the customization energy spent on attending products' specifics. This flow moves from products to features, and from there, it percolates down to the files where these features end up being realized. For flow display, alluvial diagrams are commonly used.

We resort to alluvial diagrams to factor out the customization effort. Figure 11 shows the case for the *WeatherStationSPL*. On the left, flow emitters: the products where AE spends energy to account for their specifics. On the right, flow sink: the features where the energy produced during AE ends up. Arrows connect products with those features being the subject of a customization intervention.

Nodes are the issuers of the flow. The larger the node area, the larger the flow potential. If the node stands for a product, then the node width corresponds to the effort in customizing this product. If the node stands for a feature, then the node width corresponds to the effort in adjusting this feature no matter the product. On the other hand,

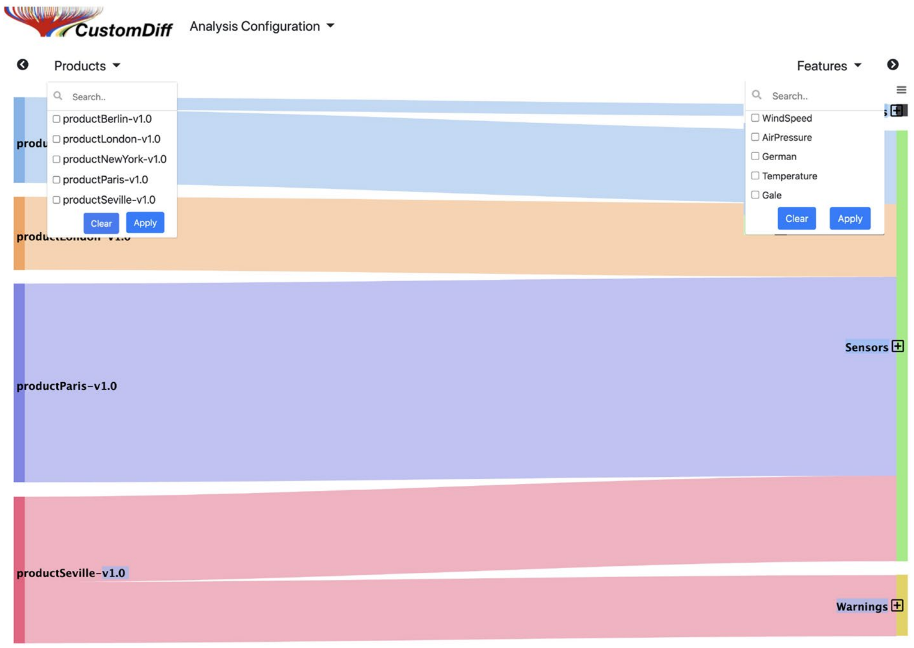


Fig. 12 **Grouping**. The Feature bar aggregates values by parent features. Likewise, the Product bar could also aggregate values by product units

arrows stand for the flow between nodes. If the arrow connects product P to feature F, then the arrow's width captures the effort involved in adjusting F to P demands.

Alluvials pursue a prompt answer to the GQM's questions. Take Fig. 11 as an example. Which features are being customized? *AirPressure*, *Temperature*, *WindSpeed*, *Gale* and *German*. Which products are customizing features? *productBerlin*, *productLondon*, etc. How much effort (i.e., code churn) has been spent on customizing the feature *AirPressure*? This is reflected in the node width; the tool also shows the customization diffing (i.e., 36 LOC) on a mouse over. How much effort has spent *productParis* in customizing *AirPressure*? This is reflected in the arrow width; mouse over to get 31 LOC. This anecdotal evaluation looks promising. Yet, there exists a first stumbling block: scalability.

SPLs can hold hundreds of features that can be combined into thousands of products. Although not every feature/product is involved in each customization cycle, visualization might become cluttered for a larger number of features or products. Filters and grouping are the most common techniques to reduce the number of nodes. Filters limit the flow to either the products or the features that meet the filtering criteria. Grouping permits products/features to be grouped into clusters. Products could be grouped based on product units. A product unit might be in charge of one or several products. Grouping permits this effort to be visualized for the whole unit. Likewise, features might be grouped based on their parent features so that the effort is displayed for the whole set of child features. The grouping of nodes also implies a grouping of the adjacent edges. Figure 12 shows groupings for the case of features. Now the flow concentrates along *WeatherStationSPL*'s parent features, i.e., *Sensors, Languages, and Warnings*.

6.2 Evaluation

To assess the suitability of alluvial diagrams, informal demo sessions were conducted. Interestingly enough, during the demo sessions two sort of clusters, similar to those that emerged during the *understand* evaluation phase, started to surface: coarse-grained (preferred by the Change Control Board) and fine-grained (preferred by developers). At the onset, researchers were mainly concerned about an intervention for correlating where the customization was occurring with where the customization was planned to occur. Here, the stakeholders are SPL managers interested in tracing the balance between DE and AE. Yet, as the interaction with practitioners advanced, a new role started to emerge, i.e., developers, who were concerned not only about the high-level customization representation but also about the code behind it. Rationales rest on the grow-and-prune model. Single-off development is more about growing than about pruning. That is, single-off development does not face later consolidation of upgrades into a common platform (i.e. the *master*). Applications evolve at their own pace. By contrast, product developers at SPLs are well aware that sooner or later their upgrades need to be merged back into the SPL platform. The bottom line is that developers' concerns include not only customizing but also consolidating the upgrade. The alluvial diagram so far seems to be appropriate to capture the customization diffing but falls short to assess the consolidation effort. This requires moving down to code, to how features are fleshed out.

7 Building, Intervention, and Evaluation: Cycle 3

This section moves the developer perspective to the forefront. The alluvial diagram so far might be sufficient for managers to evaluate the effort trend in product customization (i.e., the GQM model in Fig. 8). Yet, developers do not stop at the big picture. When it comes to calibrating the consolidation effort, developers might need to go down to how features are both spread along distinct classes (i.e., scattering) and mixed up with other features (i.e., tangling).

7.1 Building & Intervention

CustomDIFF needs to cater for scattering and tangling. Nevertheless, the introduction of additional details might lead to cluttered interfaces. Hence, we should care not only for the visualization as such, but also for the interaction workflows that allow users to smoothly transit between the distinct perspectives. Accordingly, this subsection is structured along these two concerns: fine-grained visualization and workflows.

7.1.1 Extending The Visualization

So far, alluvial diagrams stopped at products and features. This alluvial is now extended to account for scattering, tangling and code peering.

Scattering. We resort to extending the scope of the alluvial flow by incorporating the *package* bar at both ends (see Fig. 13). Details follow:

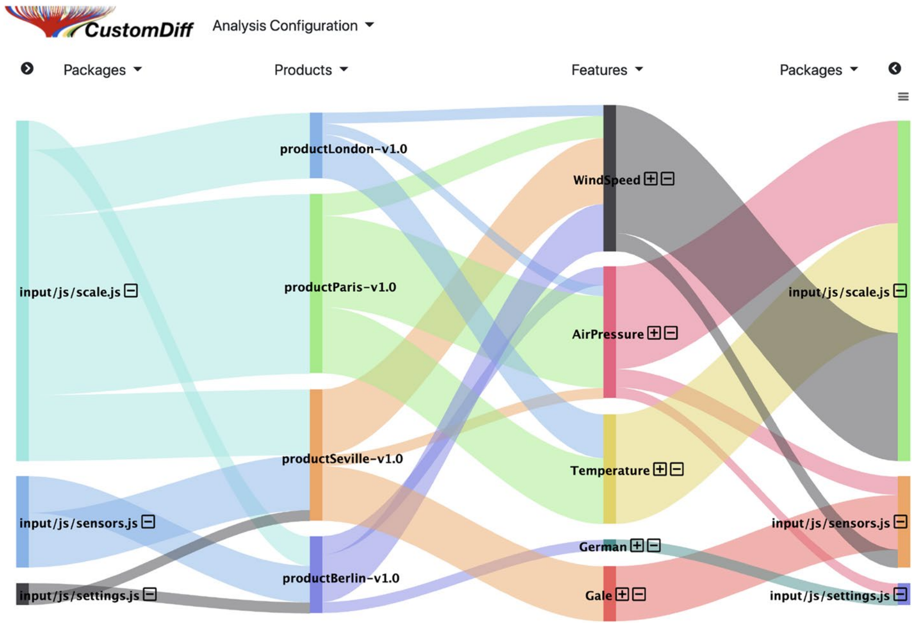


Fig. 13 Scattering visualization. Two additional Package bars are added at each extreme to expand the flow down to files. For example: customization on *AirPressure* (no matter the product) goes across files *scale.js*, *sensors.js* & *settings.js*

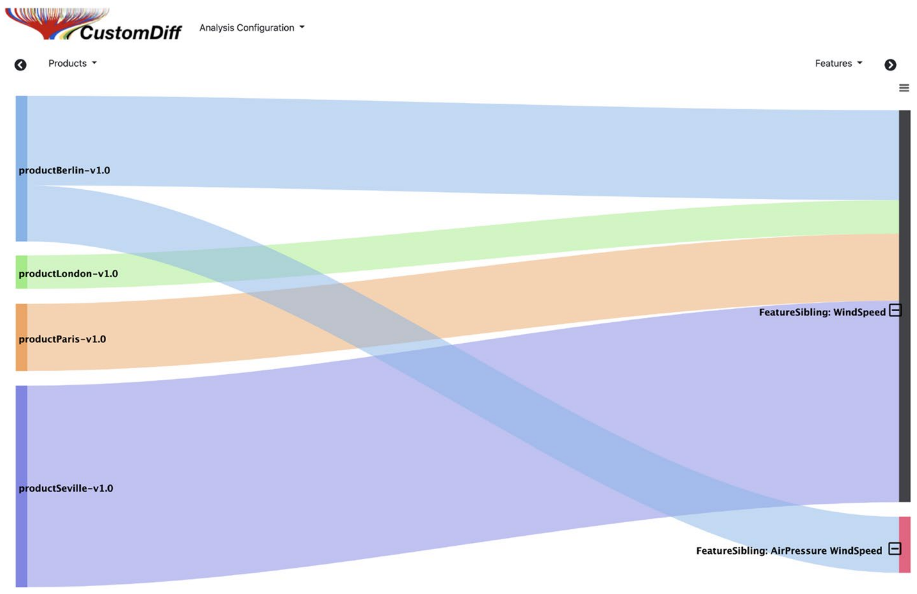


Fig. 14 Tangling visualization for feature *WindSpeed*. Customizations involving *#ifdef* blocks that include *WindSpeed* in their directives are broken down into two FS sets: *'FeatureSibling: WindSpeed'* and *'FeatureSibling: AirPressure-WindSpeed'*

- On the right side, the package bar extends the flow from *feature* to *file* so that DE developers can have a piece-meal perspective on how their core assets are being adjusted. This permits an estimation of the eventual consolidation effort. For instance, the code churn for file *scale.js* is three times larger than the one for *sensors.js*. Accordingly, engineers might now foresee that merging *scale.js* might most likely demand more time than merging *sensors.js*.
- On the left side, the package bar extends the flow from *product* to *file* so that product units can assess what other units are working on the same files. This permits promptly spotting overlapping risks. For instance, developers at *productBerlin-v1.0* and *productSeville-v1.0* might probably arrange a meeting together to prevent redundant efforts. This in turn, might alleviate the burden at consolidation time by smoothing out their differences in advance.

Tangling. We resort to grouping (see Fig. 14). So far, the Feature bar captures the customization diffing along with the features being updated. Nodes might account for parent features that can be next broken down into their child features. We can further break down feature nodes into *Feature Sibling (FS) Sets*, a set of features that appear together in at least one *ifdef* directive, no matter the boolean expression that links them together. Figure 14 shows the case of *WindSpeed*. This feature is broken down into two FS sets. First, the *WindSpeed* set which accounts for modified *ifdef* blocks with *WindSpeed* as the only feature in their pre-compilation directive. Second, the *AirPressure - WindSpeed* set that agglutinates *ifdef* blocks where these two features are referred to in their directives, no matter the boolean operator. The flow from *productBerlin-v1.0* to *AirPressure - WindSpeed* depicts the tangled effort made to evolve the *AirPressure* and *WindSpeed* features. By zooming into FS sets, developers can have a first insight into the extent of tangling along with the customization diffing.

Code Peering. CustomDIFF sticks to the traditional DIFF view for code peering. Yet, some subtle changes are needed. Traditionally, the DIFF context refers to the three nearest unchanged lines that precede and follow the change (see Fig. 4). The context serves as a reference to locate the places of the changed lines. However, this might not be enough for variability-intensive code. Here, the code holds variability points where a pre-compilation directive regulates whether the block code is to be included in the final product. This pre-compilation directive is a main contextual clue to know which features are affected. Yet, these pre-compilation directives are right at the start of the block, potentially away from where the change has occurred, and hence, these directives might not show up in a traditional DIFF context. Figure 4 illustrates this situation. The change is located at line 29. However, the context (i.e., lines 26, 27, 28) does not include the pre-compilation directive. This deprives engineers from promptly knowing which features are affected. Therefore, feature-minded DIFF utilities should include pre-compilation directives as part of the DIFF context. Figure 15(c) mimics the case of Fig. 4, but now information about the pre-compilation directive is included into the hunk headings. The hunk corresponds to the changes in lines 43 and 50 under the scope of VP-1 *(WindSpeed or AirPressure)*. In the case that VP-1 is nested within another variation point (e.g., VP-2), this is reflected in the hunk's heading along the pattern $:< enclosingVP > -- > nestedinto -- >< enclosedVP >$. In short, pre-compilation directives should become a main ingredient of the DIFF context for variability-intensive code.

Fig. 15 The Developer workflow. Moving down to code through hypertext navigation along with three ▶ HTML pages: (a) alluvial diagram; (b) gateway panel; (c) DIFF View; and (d) back to the alluvial diagram for a related feature

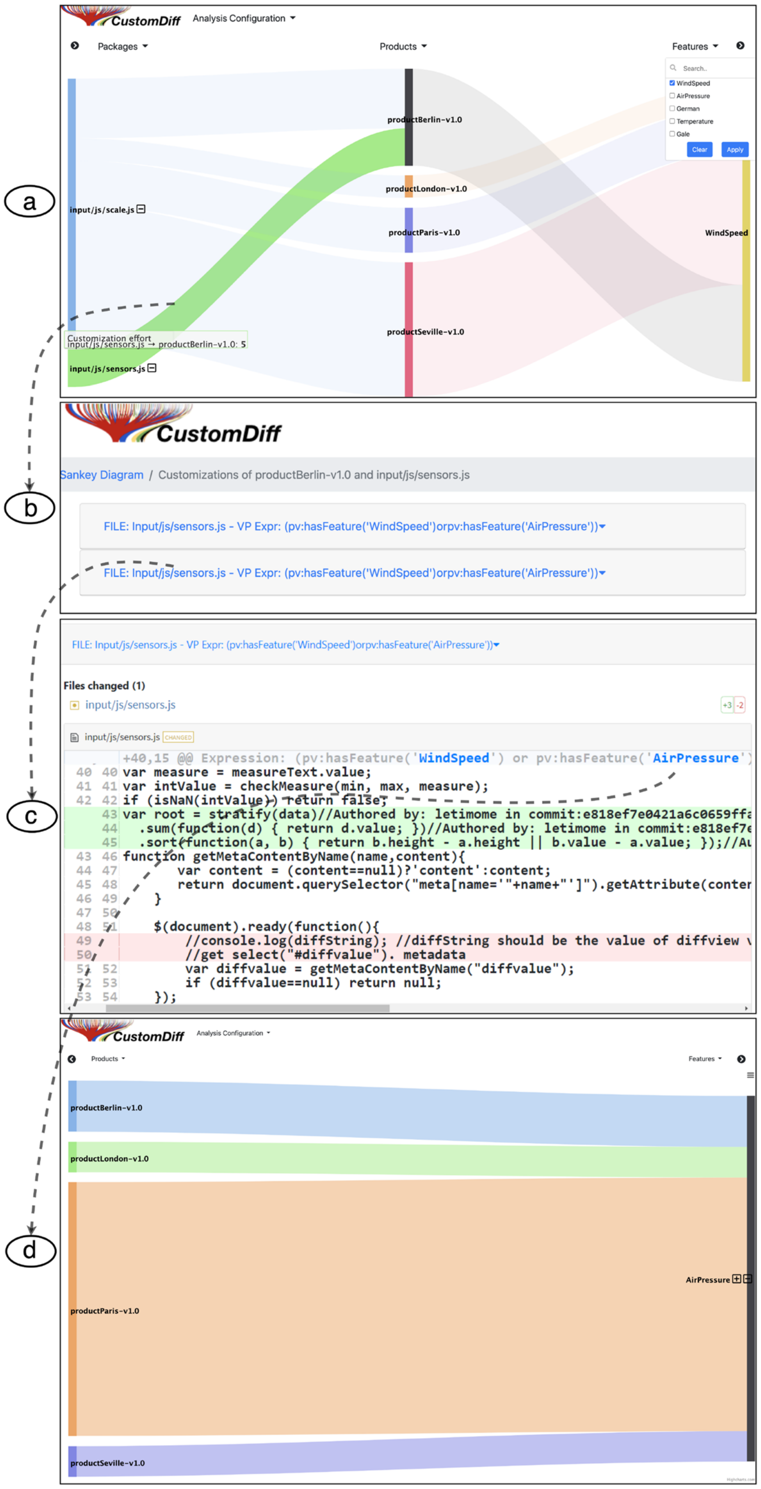
7.1.2 The Workflows

Based on insights gained during the GQM evaluation, *CustomDIFF* considers two main perspectives: coarse-grained (preferred by the Change Control Board) and fine-grained (preferred by developers).

The Change Control Board Workflow. Here, stakeholders are interested in correlating where the customization is occurring with where the customization is planned to occur. They aim at differentiating between mature features (none or minimal customization needed) *vs.* immature features (recurrent customization required). They could start by displaying the alluvial diagram for the selected features at the highest abstraction level (see Fig. 12). Next, they can click on the grouping button to unfold parent features into their child features (see Fig. 11), or even further, moving down to the feature siblings where the feature participates (see Fig. 14). Notice that alluvials are abstracted from the customized code, hence, *only* product, features or variation points that have been customized will appear in the alluvial. Additional details can be obtained by extending the flow to the right/left by introducing the Package bar alongside the Feature/Product bar (see Fig. 13).

One key characteristic of interactive alluvials is the tracing of flows throughout the graph. Users may select a node or edge, and the contributions of all flows are highlighted and moved to the foreground. In this way, interactive alluvials draw the attention of the analyst to the largest flows, the largest consumer, or the main flow deviations or losses.

The Developer Workflow. Here, stakeholders are interested in assessing the complexity of integrating *product* branches back into the *master*. This might require moving down to code, i.e., from alluvials to DIFF views. This is achieved through a 'gateway panel'. As an example, consider we would like to get insights on the effort of merging back customizations on *WindSpeed* to the *master*. We start by filtering upon *WindSpeed* (see Fig. 15a). Four products turn out to be responsible for having customized *WindSpeed*: *productBerlin, productLondon, productParis*, and *productSeville*. By displaying the Package bar, we also note that two files are being affected: *scale.js* and *sensors.js*. The visualization so far might help to get a global view, yet it is insufficient for developers who need to foresee the integration difficulties ahead. To this end, both nodes and arcs of alluvials are turned into hyper-links. Click on an arrow and a new browser tab opens a gateway panel. This panel displays a list of the affected *ifdefs* blocks (Fig. 15b). Click on an *ifdef* block for the corresponding code to appear in the DIFF view (Fig. 15c). In the example, the DIFF view shows that *WindSpeed* is tangled with other features (e.g., *AirPressure*). Hence, consolidating *WindSpeed* might also impact *AirPressure*. The developer might now wonder the extent to which *AirPressure* is being customized, which can be analyzed by moving back to an alluvial view. This workflow might then need to move back and forth between the alluvial view and the DIFF view. To this end, the DIFF view is turned into a hypertext, i.e., feature names in the DIFF context (variation points) are turned into URLs. Click on an URL and a new browser tab shows the alluvial view for the feature at hand (Fig. 15d. In this way, a *CustomDIFF* session is realized as a succession of tabs where alluvial views and DIFF views intermingle as consecutive browser tabs.



Customization effort
input/js/sensors.js → productBerlin-v1.0: 5

Sankey Diagram / Customizations of productBerlin-v1.0 and input/js/sensors.js

FILE: Input/js/sensors.js - VP Expr: (pv:hasFeature('WindSpeed')orpv:hasFeature('AirPressure'))

FILE: Input/js/sensors.js - VP Expr: (pv:hasFeature('WindSpeed')orpv:hasFeature('AirPressure'))

FILE: Input/js/sensors.js - VP Expr: (pv:hasFeature('WindSpeed')orpv:hasFeature('AirPressure'))

Files changed (1)

input/js/sensors.js +3 -2

input/js/sensors.js CHANGED

```
+40,15 @@ Expression: (pv:hasFeature('WindSpeed') or pv:hasFeature('AirPressure')
var measure = measureText.value;
var intValue = checkMeasure(min, max, measure);
if (isNaN(intValue)) return false;
var root = stratify(data)//Authored by: letimome in commit:e818ef7e0421a6c0659ffa
  .sum(function(d) { return d.value; })//Authored by: letimome in commit:e818ef7e
  .sort(function(a, b) { return b.height - a.height || b.value - a.value; });//Au
function getMetaContentByName(name,content){
       var content = (content==null)?'content':content;
       return document.querySelector("meta[name='"+name+"']").getAttribute(conter
    }

    $(document).ready(function(){
        //console.log(diffString); //diffString should be the value of diffview v
        //get select("#diffvalue"). metadata
        var diffvalue = getMetaContentByName("diffvalue");
        if (diffvalue==null) return null;
    });
```

7.2 Evaluation

According to the GQM in Fig. 8, we aim at evaluating the effort trend in product customization. This is broken down into three perspectives: holistic, feature-centric and product-centric. These are the information needs. Yet, the added value of visual analytics goes beyond the information needs to care also for the way this information is delivered. Therefore, this section evaluates *CustomDIFF* from the analytical side (i.e., usefulness) but also the visual side (i.e., ease of use). Perceived usefulness is the degree to which a person believes that using a particular system would enhance his or her job performance. On the other hand, perceived ease of use is the degree to which a person believes that using a particular system would be free of effort (Davis 1989). Accordingly, we can state our evaluation goal as:

> Assess the ease of use and usefulness of *CustomDIFF* with respect to conducting customization analysis from the point of view of SPL practitioners in the context of Danfoss Drives.

The question remains how to conduct the evaluation. Sein et al. emphasize that authenticity is a more important ingredient for ADR 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 to meet all factors simultaneously, given the incipient nature of the interventions being checked out. SPLs are no exception, quite the contrary. In our case, it was not possible to use real tasks from Danfoss Drives, but we could tap into engineers from Danfoss Drives to check out complex-enough tasks upon the *WeatherStationSPL*. This subsection reports this *expert evaluation*.

7.2.1 Participants

Our evaluation derives its value from the expertise of participants in handling product-based SPL evolution. As customization analysis allows for different perspectives (feature vs. product) that might depend on the participants' role, the selection was balanced between DE and AE. To attain this goal, we had the invaluable collaboration of a person with managerial responsibilities in Danfoss Drives, who encouraged participation through an open call. This call stated that participation was voluntary and that results would be gathered anonymously. No rewarding procedure was set except a coffee after the evaluation. Six people were able to participate: 1 product release manager, 3 software developers with a hybrid role in both domain and application engineering tasks, and 2 code reviewers in charge of branch integration. Participants' average expertise in Danfoss Drives was 7 years.

7.2.2 Measurement Tool

Perceived Ease of Use We resorted to a questionnaire based on the *Technology Acceptance Model (TAM)* (Davis 1989) (see Table 4). Questions are arranged along a LIKERT scale from 1 (strongly disagree) to 7 (strongly agree).

Perceived Usefulness We could have resorted again to the TAM questionnaire. However, TAM's perceived usefulness questionnaire has been criticized for being too general

(Hornbæk and Hertzum 2017). Hence, we decided to create our own questionnaire (see Table 5). This questionnaire was checked for reliability. The reliability of any given measurement refers to the extent to which it is a consistent measure of a concept, i.e., the questionnaire's items all reflect the same concern. Cronbach's alpha is one way of measuring the strength of that consistency. The Cronbach's alpha for the questionnaire in Table 5 is 0.93. A general accepted rule is that α of 0.6-0.7 indicates an acceptable level of reliability, and 0.8 or greater a very good level (Ursachi et al. 2015).

7.2.3 Procedure

The procedure was designed with two goals in mind. On one hand, the tasks presented to participants needed to be meaningful enough to allow them to understand *CustomDIFF* and its application for visual analytics. On the other hand, given that participants have a busy agenda, it had to be limited in time with the aim of not disrupting their jobs more than necessary. To this end, the *WeatherStationSPL* was used as the running example. This SPL is included in the *pure::variants* experimental material, and hence, participants were already familiar with it. Consequently, participants could focus on customization issues rather than spending time understanding the SPL domain itself.

For the evaluation, three product variants were created: *productParis*, *productBerlin*, and *productNewYork*, each with a set of customizations. Customizations were designed to mimic the complexity that could be found in Danfoss in terms of affected features and products.

Upon this setting, participants were requested to conduct distinct tasks to assess the three perspectives identified in Section 5: holistic perspective, feature perspective, and product perspective (see Table 3). These tasks are felt representative of the information needs identified in Table 2.

The evaluation was conducted in two sessions. The first session presented *CustomDIFF* (1h 45') to participants. *CustomDIFF's* rationale and operations were introduced with the help of the *WeatherStationSPL*. The second session was a hands-on experience, where participants explored *CustomDIFF* on their own (1h 30'). The aforementioned sample cases were introduced. Next, participants were asked to fill two on-line questionnaires to assess *CustomDIFF's* usefulness and ease of use. Due to agenda constraints, participants were divided into two groups, with 2 and 4 participants each. During the sessions, a researcher was observing participants' interactions with the tool. Participants raised questions, doubts, and comments that were noted by the researcher.

7.2.4 Results

Ease of Use (see Table 4) Participants rated *CustomDIFF* with an average of 5.44. Although results seem to suggest that *CustomDIFF* is affordable enough, the lowest ranked question was the second one, which seems to suggest that some analysis workflows might not be as direct as expected. In addition, the researcher observing the participants noted a caveat for the interaction with *CustomDIFF*. Specifically, the difference between node clicking and arrow clicking was not apparent, and some participants expressed confusion. For instance, when looking at the customization effort undertaken in *productParis* to adjust the *Sensors* feature, some participants first clicked on the *productParis*. Although participants were not expected to write anything concrete, just to explore customization, direct

Table 3 Tasks for the expert evaluation

Perspective	Task	Outlier identification task
Holistic	Which parent features are not customized by the products?	Analyze how *productParis* is changing the implementation of the *Sensors* parent feature
Feature-centric	Which products are customizing the *AirPressure* child-feature?	Analyze how the code that realizes *AirPressure* has been changed by the product portfolio
Product-centric	Which parent-features is *productBerlin* customizing?	Analyze how the implementation of *productBerlin* has evolved

Table 4 *CustomDIFF's* perceived ease of use

ID	Item:	P1	P2	P3	P4	P5	P6	Avg.	St. Dev.
E1	Learning to operate *CustomDIFF* would be easy for me	6	4	7	6	6	6	5.83	0.98
E2	It would be easy for me to become skillful at using *CustomDIFF*	7	4	6	6	6	5	5.67	1.03
E3	My interaction with *CustomDIFF* would be clear and understandable	6	4	6	6	6	5	5.5	0.84
E4	I would find *CustomDIFF* easy to use	6	4	6	6	6	5	5.5	0.87
E5	I would find *CustomDIFF* to be flexible to interact with	5	4	6	5	6	5	5.17	0.75
E6	I would find it easy to get *CustomDIFF* to do what I want it to do	5	4	6	6	5	4	5	0.89

The six practitioners (P1, P2...) were asked the extent of their agreement for each item along a LIKERT scale that ranges from 1 (strongly disagree) to 7 (strongly agree)

observation of the participants showed that not all of them selected the *Sensors-product-Paris* arrow. Some participants first clicked on *productParis* and, in a second interaction, determined the adjustment for *Sensors*.

Usefulness (see Table 5) The questionnaire aims at capturing usefulness for conducting the tasks at hand. As long as these tasks have first been validated as relevant for customization analysis (the GQM model), they might conform to a sort of benchmark against which analysis tools can be evaluated. On these grounds, *CustomDIFF* yields good results from 6 (U1) to 5.33 (U6). That said, a difference can be appreciated between those questions aimed for the Change Control Board (coarse-grained: U1, U2, U3, U4) vs. those questions thought for application engineers (fine-grained: U5, U6). The latter outputs worse average and standard deviation outcomes. This might be due to Change Control Board members appreciating the novelty of alluvial diagrams, while application engineers did not see major differences w.r.t. the traditional DIFF view.

These results seem to suggest that participants found *CustomDIFF* easy to use (items ranked above 5 out of 7) and useful (items ranked above 5.33) for fulfilling the information needs in Table 3.

7.2.5 Threats to Validity

Construct Validity refers to the degree of accuracy with which the variables defined in a study measure the constructs of interest. Here, the constructs are ease of use and usefulness. As for the former, we resort to Davis' questionnaire whose validity and reliability have been previously endorsed (Mathieson et al. 2001; Agarwal and Prasad 1998). Riskier is the use of our own questionnaire to assess usefulness. To ensure internal consistency, Cronbach's alpha is calculated for the questionnaire which resulted in an α value of 0.93, showing a high reliability, though the low number of participants needs to be considered.

Internal Validity refers to the extent to which the intervention or independent variable(s) were actually responsible for the effects seen in the dependent variable. Here, the intervention is *CustomDIFF*. Yet, other factors besides *CustomDIFF* might influence the results. First, the participants' background. In this respect, we were especially careful to focus on SPL engineers who had at least one-year experience. Second, the questionnaire's

Table 5 *CustomDIFF's* perceived usefulness. LIKERT scale ranges from 1 (strongly disagree) to 7 (strongly agree)

ID	Item:	P1	P2	P3	P4	P5	P6	Avg.	St. Dev.
U1	*CustomDIFF* was useful to determine which parent-features are not customized by the products	7	5	6	6	6	6	6	0.63
U2	*CustomDIFF* was useful to determine which parent-features is *productBerlin* customizing	7	4	6	6	6	6	5.83	0.98
U3	*CustomDIFF* was useful to determine how *productParis* is changing the implementation of the *Sensors* parent-feature	6	5	6	6	7	5	5.83	0.75
U4	*CustomDIFF* was useful to determine which products are customizing the *AirPressure* child-feature	6	4	6	6	6	6	5.67	0.82
U5	*CustomDIFF* was useful to determine how is each product customizing the code that realizes the *AirPressure* feature	6	4	6	6	7	4	5.5	1.22
U6	*CustomDIFF* was useful to determine how the implementation of *productBerlin* has evolved	6	4	6	6	6	4	5.33	1.03

understandability. To reduce its influence, we carefully designed a running example that aimed at helping participants contextualize the different questions and ensuring a common understanding.

External Validity tackles the representativeness of the study and the ability to generalize the conclusions beyond the scope of the study itself. Representativeness can be challenged by the participants or the evaluation tasks. As for the former, we resorted to Danfoss engineers. To account for different perspectives, we aimed at involving practitioners with experience in both DE and AE. As for the tasks, the *WeatherStationSPL* is a rather small SPL compared to industrial SPLs. It is explicitly designed to help new users understand the concepts of *pure::variants* and, thus, it is expected to be cleanly engineered (which is, quite safe to say, not the case for industrial SPLs). Yet, for an incipient tool such as *CustomDIFF* to be put to the test using industrial code is difficult. Moving to variability-intensive open applications could have been an option. This would have brought more realistic tasks, yet at the expense of reducing the realism of participants and environment. The latter are however pivotal in a decision-taking scenario such as customization analysis.

8 Formalization of Learning

In accordance with ADR, the situated learning from the 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 customization analysis a *problem* for organizations other than Danfoss Drives;
- generalization of the solution instance, i.e., to what extent is *CustomDIFF* a *solution* to customization analysis; and
- derivation of design principles, i.e., what sort of design knowledge can be distilled from the *CustomDIFF* experience that might inform other tool builders.

The rest of this section tackles these questions.

8.1 Generalization of the Problem Instance

We tackle product customization at Danfoss Drives. This section elaborates on generalizing this experience to product-based evolving SPLs. Product customization challenges the traditional vision whereby new requirements are transmitted to domain engineering, features are built on the platform, and the product is then finally created. This conventional vision is increasingly called into question. Indeed, Krueger et al. report that an increasing number of companies and open-source projects add new variations or platform features using *feature forks* through version control systems (e.g., Git) (Krüger et al. 2020). Here, the platform, kept in the *master* branch, evolves through re-integrating these forks. If the *master* branch holds the SPL core assets, and the forks account for transient product enhancements, then we are talking about product-based evolving SPLs.

Provided this way of SPL evolution, the question remains whether customization analysis is also a problem in these organizations. It could be argued that SPLs in an earlier stage of their life-cycle might be the ones subject to a larger customization endeavor while more mature SPLs can obtain most of their products out of the core assets with minimal customization. That said, Danfoss Drives can be considered a mature SPL, and yet product customization is still very relevant. As usual in SPLs, the rationales might rest on scalability. SPLs handle a large number of products. Hence, perfective maintenance needs to scale up to the petitions of not one product's stakeholders but a myriad of products. Timeliness might require the products' perfective maintenance to develop in parallel. If this is your organization's case, chances are you need to track how customization is conducted, i.e., customization analysis.

8.2 Generalization of the Solution Instance

This project develops *CustomDIFF* as an intervention for tackling customization analysis. We resort to visual analytics, using *Git* repositories as the data mines. This is not new. Distinct authors tackle both mining code repositories (we claim no contribution in this area) and software visualization for SPLs (our main contribution). The question arises about how alluvial diagrams compare w.r.t. other visualizations proposed in an SPL setting. Lopez-Herrejon et al. (2018) conduct a systematic survey where interventions are categorized based on the SPL stage to which the visualization technique applies. We enlarge this study for the stage *maintenance and evolution* along three dimensions (Novais et al. 2013): (1) the point of view, i.e., *who* is the user group that will use the visualization, (2) the object of study, i.e., *what* is being analyzed, and (3) the purpose and the focus, i.e., *why* is the analysis being done. Table 6 shows the results. Specifically, *CustomDIFF* can be pigeonholed as helping the Change Control Board (*who*) conduct customization analysis (*what*) for assessing SPL scoping (*why*). We use the visualization means as a way to structure the rest of the paragraphs.

Trees. Its intuitiveness and the large support of graphical libraries make trees the most popular intervention. Trees naturally convey the notion of hierarchy as well as setting node clusters in terms of family dependencies (e.g., ancestors, siblings, descendants, etc). These affordabilities are put into play for different purposes. Kanda et al. aim at helping engineers in migrating a set of products, created through clone&own, to an SPL (Kanda et al. 2013). For the sake of better identifying commonalities and variability, engineers require to understand how products are derived from each one. This can be challenging if no tracing records exist. In these instances, Kanda et al. introduce the *Product Evolution Tree* visualization whereby derivation relationships among products are displayed in a way similar to VCS branching (Kanda et al. 2013). If the aim is not migration but testing, De Oliveira et al. introduce the *Product Genealogy Tree* with the aim of identifying products to be re-tested when a bug is found in a product (de Oliveira et al. 2012). This tree captures three traces: (1) which products were derived from which core assets, (2) which products are created from already derived products, and (3) which products have propagated changes to which products. When a bug is detected in a product, this visualization can help testers identify which other related products also need to be tested.

Tree-Map. This approach arranges data in a hierarchical, tree-structured diagram where the size of the rectangles is organized from the largest to the smallest. Main benefits include showing the ratio of each part to the whole. Tenev et al. resort to tree-maps for helping domain engineers identify the reuse potential of a number of similar software variants created through *clone&own* (Tenev et al. 2017). They compute the similarities of the source code of multiple software systems, and visualizes the commonalities and variabilities by means of multiple visualization means, such as, bar diagrams, tree-maps and phylogenetic diagrams. These diagrams provide domain engineers information about code

Table 6 Related work on *SPL visualization*, along facets:*who*, *what*, *why* and *visualization means*

Reference	Who	What	Why	How
Kanda et al. (2013)	Domain engineers	Product-to-product relationships	SPL migration	Product evolution tree **FROM** product code
de Oliveira et al. (2012)	SPL testers	Co-product testing	Product testing	Testing tree **FROM** product code
Tenev et al. (2017)	Domain engineers	Code similarity	Spotting reuse opportunities for SPL adoption	Treemaps **FROM** product code
Wnuk et al. (2009)	Change control board	Feature survivability	SPL scoping	Feature survival tree **FROM** requirement documents
Hinterreiter et al. (2020)	Change control board	Customization diffing	SPL scoping	Feature evolution plot **FROM** platform code
CustomDIFF	**Change control board**	**Customization diffing**	**SPL scoping**	**Alluvial diagrams FROM product code**

similarity across a group of cloned software systems at different abstraction levels, i.e., from a single code line through files, folders and subsystems up to the whole system, which can then identify reuse potential and schedule an SPL migration plan.

Charts. Unlike trees, charts are more convenient when a longitudinal analysis is required. Wnuk et al. introduce the *Feature Survival Chart* for the visualization of scoping change dynamics: the X-axis stands for time while the Y-axis holds the features (Wnuk et al. 2009). In this way, the complete life-cycle of a single feature can be followed by looking at the same Y-axis position over time. This work does not tackle code but documents that formulate features for an upcoming platform project. A feature in this case is a concept of grouping requirements that constitute a new functional enhancement to the platform. At this stage, the features usually contain a description, their market values, and effort estimates. Features are refined to requirements which are specified, reviewed, and approved. The requirements are written in domain-specific natural language. The final scope is decided and agreed with the development resources. Wnuk et al. tap into these requirement documents to build up the *Feature Survival Chart* where the introduction/deletion of features are monitored. In this way, the visualization shows the decision process of including or excluding features that are candidates for the next SPL release. The aim is to prevent two sorts of problems: (1) setting too large scope compared to available resources and (2), setting a limited scope early, missing market opportunities. More recently, Hinterreiter et al. (2020) propose an IDE tool for developing and evolving a clone-and-own SPL. Their approach rests on an *ad-hoc* variability-aware VCS. Their tool supports a code-diff view that allows engineers to view how a given feature evolved for a given product (i.e., the product perspective). Specifically, Hinterreiter et al. (2020) propose two metrics to reflect how the code evolves: the relative change of the realizing artifact size (a sort of code churn), and the feature's scattering change. In contrast, *CustomDIFF* is based on *Git* and *pure::variants* (i.e., annotation-based SPLs), reflecting a more industrial setting. In addition, visualization aims to account not only for the product perspective, but also the holistic perspective and the feature perspective.

Both Hinterreiter et al. and Wnuk et al. track SPL evolution based on requirement documents and the codebase, respectively (Hinterreiter et al. 2020; Wnuk et al. 2009). By contrast, *CustomDIFF does not* tackle the visualization of the evolution of the SPL. Rather, we focus on how to inform the follow-on SPL release. Alluvial diagrams depict the customization endeavor *in the interim* between SPL releases.

8.3 Derivation of Design Principles

So far, we have looked at *CustomDIFF* as a whole. Now, we disentangle the distinct mechanisms that on balance are responsible for the usefulness and perceived ease of use as detailed in Section 7.2. Table 7 outlines the main 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 effect (i.e., Change Control Board activity made possible), the cause (affordability brought about by *CustomDIFF*), and the context where this cause can be expected to yield the effect for the target audience (i.e., the Change Control Board and the application engineers).

Table 7 outlines the four principles we consider more relevant. Principles *flow-ness* and *grouping* collect the benefits brought about by alluvials. On the other hand, principles *zoom-ness* and *filtering* are common principles in Visual Analytics. At this point, we also consider it significant to collect stakeholders' opinions about each mechanism in isolation.

Table 7 Design principles from the *CustomDIFF* experience

Provide visual analytics tool with ...	in order for the Change Control Board to ...	*CustomDIFF* realization
Flow diagrams	Track customization efforts along SPL-relevant concerns	Alluvial diagrams
Grouping facilities	Obtain aggregates along with SPL-relevant concerns	Parent-feature grouping
Expansion/contraction utilities	Handle distinct data-granularity levels	Package bar
Filtering and tracing of flows throughout the alluvial diagram	Not feel lost in the tangle of flow branches	Feature-product filtering

Table 8 Questionnaire on *CustomDIFF*'s mechanism

Mechanism:	Item:	P1	P2	P3	P4	P5	P6	Avg.	St. Dev.
Alluvial diagrams	I would find alluvial diagrams useful for grasping the customization diffing	6	5	6	6	6	6	5.84	0.41
Feature-product filtering	I would find feature-based filtering utility useful to easy focus	7	4	6	6	6	6	5.83	0.98
Parent-feature grouping	I would find the parent-feature grouping was useful to abstract away from individual features	6	5	5	5	6	6	5.5	0.58
Package bar	I would find the "Package" bar was useful to further refine the flow	5	5	6	5	6	5	5.33	0.52
n.a.	I would find the VP-enriched context DIFF useful to easy locate change placement into the code	6	5	7	6	5	4	5.5	1.05

Statement agreement is arranged along with a LIKERT scale from 1 ("Strongly disagree") to 7 ("Strongly agree") for the six participants

Participants were the same throughout the research. We limited ourselves to one question for each mechanism. Table 8 collects the results. Mechanisms are in general appreciated with an average above 5 out of 7. Dispersion wise, the VP-enriched context is the mechanism with the larger dispersion. Rationales might rest on this mechanism being of interest only for application engineers while the Change Control Board might not need to delve into the code.

9 Conclusions

This research started with a phenomenon perceived in practice, i.e., product customization at Danfoss Drives. In this setting, we raised two questions: (1) which are the information needs for customization analysis? and (2), might alluvial diagrams be useful for supporting customization analysis visualization? As for the former, we provided some estimates on the cost of answering some questions, and developed a GQM model. We proposed the adjustment of the popular metric of code churn for SPL constructs: *feature* and *product*. However, the potentially high number of *feature churns* and *product churns* require appropriate visualization means. This moved us to the second question. Here, we made the case of alluvial diagrams. Proof-of-concept was provided through *CustomDIFF*, a publicly-available Web application using *pure::variants* as the variability manager, and *Git* as the code repository. Proof-of-value was conducted for ease of use and usefulness using Danfoss practitioners as the subjects. Results seem to suggest that alluvial diagrams facilitate a natural way to describe the flow dispersion at different levels of detail. On one hand, the Change Control Board gets the big picture along with products and features. On the other hand, developers can expand the flow to packages and files, down to the raw code. In this way, *CustomDIFF* advances traditional DIFF utilities for SPL specifics. We ended by generalizing this experience to SPLs other than Danfoss Drives. In short, we contribute to the existing literature by identifying information needs and advocating for alluvial diagrams as a feasible and effective way to reflect product customization in SPLs.

Additional empirical studies are needed to assess the value of customization analysis for actionable interventions. Our first evaluations indicate that customization diffing is a factor, but it is certainly not the only one that intervenes during decision making. In this respect, we plan to supplement *Git* data with data about products, customers, and developers, and to see what other kinds of analysis these additional sources would permit. In the same vein, metrics other than code churn might be of use. Besides tangling and scattering, more traditional metrics such as cyclomatic complexity and Halstead's, might also be useful. Another interesting follow-on would be integrating *CustomDIFF* into a DevOps framework. Here, customization endeavors can be continuously tracked, so that actions can be attached to some customization thresholds being surpassed. Other scenarios include the use of *CustomDIFF* by application engineers to gaze at what other colleagues are customizing. For instance, a feature enhancement introduced in a given product might be promptly and directly incorporated into other products, without waiting for this enhancement to be promoted as a core asset. This opens up new scenarios for SPL evolution. Here, *longitudinal evolution* (between core assets and products) might coexist with *traversal evolution* where products sharing the same features might decide to incorporate enhancements from other products, and later on, be jointly consolidated. The aim is to find ways to alleviate the tension between the quality and reuse effectiveness required by domain engineering, and the time-to-market and customer pressure faced by application engineering.

Appendix: Mining at *CustomDIFF*

CustomDIFF supports the grow-and-prune model (Faust and Verhoef 2003). Here, *product* branches are derived off the *master* branch (see Fig. 1). This appendix outlines how *CustomDIFF* derives customization facts from so-organized *Git* repositories. More details can be found at the *CustomDIFF* repository itself: https://github.com/onekin/customdiff

Algorithm 1 Mining product customizations.

```
List<CustomizationFacts> MINE_CUSTOMIZATIONS (GitRepository gitRepo,
     String baseline_tag)

List<CustomizationFact> customization_facts = new List<
    CustomizationFact> ();
Commit baselineCommit = gitRepo.getCommitByTagName(baseline_tag);

List <Tag> all _tags = gitRepo.getAllTags()
for each (tag in all_tags){
    Commit baseline;
    if(tag.name.matches(PR_PATTERN)) {
        baseline = getBaselineForRelease(tag);
        if(baselineCommit === baseline) {
            List<Diff> diffs = DIFF(baselineCommit, tag.getCommit())
            for each (diff in diffs){
                    List<CustomizationFact> custom_facts =
    extractCustomizationFacts (diff, tag);
                    customizations.add(custom_facts);
                }
        }
    }
}
return customizations;
```

The process starts with the main function *Mine_Customizations*:[6]

"List <Customization_Facts> ***Mine_Customizations****(GitRepository gitRepo, String baseline_tag)"*

This function takes a *GitRepository* as input, and returns the set of *customization_facts* that have been performed to a given baseline by all the products derived from it.[7] In addition, baseline_tag stands for the name of the git tag that identifies the baseline for which the customization facts will be computed. A running example, take the content of Fig. 1 as the *GitRepo*; *"Baseline-v1.0"* as the value for *baseline_tag*, *"PR-"* as the value for *pr_pattern*,

[6] This algorithm was implemented in Java, using the *JGit library* http://www.eclipse.org/jgit/

[7] For automated processing, the following parameters need to be configured beforehand: (1) pr_pattern, i.e the pattern that product release tags should match (e.g., "PR-*"); (2) baseline_pattern, i.e., the pattern that baseline release tags should match (e.g., "Baseline-*"), (3) vp_init_clause, i.e. the pattern that variation point opening clauses should match (e.g., "PV:IFCOND*"), and (4) vp_end_clause, i.e. the pattern that product release tag should match (e.g., "ENDCOND*").

"Baseline-" as the value for *baseline_pattern*, and *"PV:INFOND"* and *"PV:ENDCOND"* as the values for *vp_init_clause* and *vp_end_clause*, respectively.

Algorithm 1 provides the details:

1. Identify which is the *baselineCommit* to analyze (line 4). The function *getCommitByTagName* returns the commit to which the *baseline_tag* points to. For our running example, *baselineCommit* holds the commit *c5*.
2. Identify the product releases that were derived from the *baselineCommit* (lines 6–11). This implies to:
   - For all the tags in *gitRepo*, identify those that are product releases (lines 6–9). First, collect all the existing tags in the repository (line 6). For our running example, the variable *all_tags* holds now: *London-v1.0, NewYork-v1.0, Paris-v1.0, Berlin-v4.0, Baseline-v0.5* and *Baseline-v1.0*. Second, filter out those tags that are not product releases. i.e., those that do not match the *pr_pattern* (line 7–9). For our running example tags *Baseline-v0.5*, and *Baseline-v1.0* are filtered out.
   - Filter out the product releases that were not actually derived from the *baselineCommit* (lines 10-11). This is achieved in two steps. First, we identify the baseline commit from which each product release was derived. This is calculated by *getBaselineForRelease* (line 10). This method takes a product release tag (e.g., *Berlin-v4.0*), traverses the git history (e.g backwards from *c17*) until it finds a commit tagged with a label that matches the pattern *baseline_pattern* (e.g., *Baseline-v1.0*), and finally, returns the commit it points to (e.g. *c5*). Second, we filter out those product releases whose baseline is not equal to *baselineCommit* (line 11). For our running example, the product release *London-v1.0* would be filtered out, as the baseline it was derived from is *c3* instead of *c5*.
3. Finally, compute the customization facts for each product release that was indeed derived from *baselineCommit* (lines 12–16). This implies for each product release to:
   - Perform a DIFF operation between the *baselineCommit* and the commit to which the product release tag is pointing to (line 12). For instance, the DIFF operation for the product release tag *Berlin-v4.0* would be as follows: *diff(c5, c17)*. The result of the operation, i.e., diffs, is the list of diff-outputs (a.k.a patches), one per file that the product has changed from the baseline. For instance, if the product release *Berlin-v4.0* changes five files from the baseline, then diffs would contain five diff-output files, each per file changed (see Listing 1 as an example of a diff-output).
   - For each diff-output, extract the customization facts by calling the method *extractCustomizationFacts* (line 14). This method, parses the diff-output, identifies the set of consecutive changes performed to the same variation point, and returns the corresponding customization facts.
   - Finally, add the extracted customization facts to the global container customizations (line 15). Return this container when all product releases are mined (line 20).

```
diff --git a/input/js/sensors.js b/input/js/sensors.js
index 2575e5c..de6ef5f 100644
--- a/input/js/sensors.js
+++ b/input/js/sensors.js
@@ -24,12 +24,16 @@ function activateMainSensors() {

 // PV:IFCOND(pv:hasFeature('WindSpeed') and pv:hasFeature('
    Temperature')and pv:hasFeature('AirPressure') )
 var windMeasure = 0;
-function applyWindSpeed() {
+function applySensors() {
    var measureText = document.getElementById("w_measure");
    windMeasure = measureText.value;
+    measureText = document.getElementById("t_measure");
+   tempMeasure = measureText.value;
+    measureText = document.getElementById("a_measure");
+   airMeasure = measureText.value;
    var pointer = document.getElementById("w_point");
-
-   applyTachoValue(minWind, maxWind, measureText, pointer);
+
+   applyTachoValue(minWind, maxWind, measureText, airMeasure,
    tempMeasure, pointer);
    setWarnings();
    return false;
 }
```

Acknowledgements Thanks are due to the developers of Danfoss Drives. This project has benefited from a grantfrom pure::systems's GoSPLC initiative.

Author Contributions Conceptualization: Oscar Díaz; Methodology: Oscar Díaz; Formal analysis and investigation: Oscar Díaz, Leticia Montalvillo, Thomas Fogdal; Development: Leticia Montalvillo, Raul Medeiros; Evaluation: Leticia Montalvillo, Thomas Fogdal and Maider Azanza; Writing - original draft preparation: Oscar Díaz, Leticia Montalvillo; Writing - review and editing: Raul Medeiros and Maider Azanza; Funding acquisition: Oscar Díaz.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work is supported by the Spanish Ministry of Science, Innovation and Universities grant number RTI2018-099818-B-I00 and MCIU-AEI TIN2017-90644-REDT (TASOVA). ONEKIN enjoys support from the program 'Grupos de Investigación del Sistema Univesitario Vasco 2019–2021' under contract IT1235-19. Raul Medeiros enjoys a doctoral grant from the Spanish Ministry of Science and Innovation.

Data, Materials and Code Availability An interactive online version of CustomDIFF: http://customdiff.onekin.org/; a video describing CustomDIFF (6'): https://vimeo.com/577936099 and a GitHub project for the CustomDIFF implementation: https://github.com/onekin/customdiff. https://github.com/onekin/customdiff.

Declarations

Conflict of interest The authors have no competing interests to declare that are relevant to the content of this article.

References

Agarwal R, Prasad J (1998) A conceptual and operational definition of personal innovativeness in the domain of information technology. Information Systems Research 9(2):204–215. https://doi.org/10.1287/isre.9.2.204

Ajila S, Dumitrescu RT (2007) Experimental use of code delta, code churn, and rate of change to understand software product line evolution. Journal of Systems and Software 80(1):74–91. https://doi.org/10.1016/j.jss.2006.05.034

Bartholdt J, Becker D (2011) Re-engineering of a hierarchical product line. In: de Almeida ES, Kishi T, Schwanninger C, John I, Schmid K (eds) Software product lines - 15th international conference, SPLC 2011, Munich, Germany, August 22–26, 2011. IEEE Computer Society, pp 232–240. https://doi.org/10.1109/SPLC.2011.16

Basili VR, Caldiera G, Rombach HD (1994) The goal question metric approach. Encyclopedia of Software Engineering 2:528–532, http://www.csri.utoronto.ca/~sme/CSC444F/handouts/GQM-paper.pdf

Carbon R, Knodel J, Muthig D, Meier G (2008) Providing feedback from application to family engineering - the product line planning game at the testo AG. In: 12th International conference on software product lines, SPLC 2008, Limerick, Ireland, September 8–12, 2008, Proceedings. IEEE Computer Society, pp 180–189. https://doi.org/10.1109/SPLC.2008.21

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

Cook KA, Thomas JJ (2005) Illuminating the path: The research and development agenda for visual analytics. Tech. rep., Pacific Northwest National Lab.(PNNL), Richland, WA (United States)

Davis FD (1989) Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Quarterly 13(3):319–340. https://doi.org/10.2307/249008

de Oliveira THB, Becker M, Nakagawa EY (2012) Supporting the analysis of bug prevalence in software product lines with product genealogy. In: 16th International software product line conference, SPLC '12, Salvador, Brazil - September 2–7, 2012, vol 1. ACM, pp 181–185. https://doi.org/10.1145/2362536.2362561

Deelstra S, Sinnema M, Bosch J (2005) Product derivation in software product families: a case study. Journal of Systems and Software 74(2):173–194. https://doi.org/10.1016/j.jss.2003.11.012

Faragó C, Hegedüs P, Ferenc R (2015) Cumulative code churn: impact on maintainability. In: Godfrey MW, Lo D, Khomh F (eds) 15th IEEE international working conference on source code analysis and manipulation, SCAM 2015, Bremen, Germany, September 27–28, 2015. IEEE Computer Society, pp 141–150. https://doi.org/10.1109/SCAM.2015.7335410

Faust D, Verhoef C (2003) Software product line migration and deployment. Software: Practice and Experience 33(10):933–955. https://doi.org/10.1002/spe.530

Fogdal T, Scherrebeck H, Kuusela J, Becker M, Zhang B (2016) Ten years of product line engineering at danfoss: lessons learned and way ahead. In: Mei H (ed) Proceedings of the 20th international systems and software product line conference, SPLC 2016, Beijing, China, September 16–23, 2016. ACM, pp 252–261. https://doi.org/10.1145/2934466.2934491

Gregor S, Kruse LC, Seidel S (2020) Research perspectives: the anatomy of a design principle. Journal of the Association for Information Systems 21(6):2. https://aisel.aisnet.org/jais/vol21/iss6/2

Hall GA, Munson JC (2000) Software evolution: code delta and code churn. Journal of Systems and Software 54(2):111–118. https://doi.org/10.1016/S0164-1212(00)00031-5

Hinterreiter D, Grünbacher P, Prähofer H (2020) Visualizing feature-level evolution in product lines: a research preview. In: Madhavji NH, Pasquale L, Ferrari A, Gnesi S (eds) Requirements engineering: foundation for software quality - 26th international working conference, REFSQ 2020, Pisa, Italy, March 24–27, 2020, Proceedings [REFSQ 2020 was postponed], Springer, Lecture Notes in Computer Science, vol 12045, pp 300–306. https://doi.org/10.1007/978-3-030-44429-7_21

Hornbæk K, Hertzum M (2017) Technology acceptance and user experience: a review of the experiential component in HCI. ACM Transactions on Computer-Human Interaction (TOCHI) 24(5):33:1–33:30. https://doi.org/10.1145/3127358

Iida T, Matsubara M, Yoshimura K, Kojima H, Nishino K (2016) PLE for automotive braking system with management of impacts from equipment interactions. In: Mei H (ed) Proceedings of the 20th international systems and software product line conference, SPLC 2016, Beijing, China, September 16–23, 2016. ACM, pp 232–241. https://doi.org/10.1145/2934466.2934490

Jensen P (2007) Experiences with product line development of multi-discipline analysis software at overwatch textron systems. In: 11th international conference on software product lines, SPLC 2007, Kyoto, Japan, September 10–14, 2007, Proceedings. IEEE Computer Society, pp 35–43. https://doi.org/10.1109/SPLINE.2007.25

Kanda T, Ishio T, Inoue K (2013) Extraction of product evolution tree from source code of product variants. In: Kishi T, Jarzabek S, Gnesi S (eds) 17th International software product line conference, SPLC 2013, Tokyo, Japan - August 26–30, 2013. ACM, pp 141–150. https://doi.org/10.1145/2491627.2491637

Khoshgoftaar TM, Szabo RM (1994) Improving code churn predictions during the system test and maintenance phases. In: Müller HA, Georges M (eds) Proceedings of the international conference on software maintenance, ICSM 1994, Victoria, BC, Canada, September 1994. IEEE Computer Society, pp 58–67. https://doi.org/10.1109/ICSM.1994.336789

Kircher M, Hofman P (2012) Combining systematic reuse with agile development: experience report. In: de Almeida ES, Schwanninger C, Benavides D (eds) 16th International software product line conference, SPLC '12, Salvador, Brazil - September 2–7, 2012, vol 1. ACM, pp 215–219. https://doi.org/10.1145/2362536.2362566

Kodama R, Shimabukuro J, Takagi Y, Koizumi S, Tano S (2014) Experiences with commonality control procedures to develop clinical instrument system. In: Gnesi S, Fantechi A, Heymans P, Rubin J, Czarnecki K, Dhungana D (eds) 18th International software product line conference, SPLC '14, Florence, Italy, September 15–19, 2014. ACM, pp 254–263. https://doi.org/10.1145/2648511.2648540

Krueger CW (2006) New methods in software product line practice. Communications of the ACM 49(12):37–40. https://doi.org/10.1145/1183236.1183262

Krüger J, Berger T (2020) An empirical analysis of the costs of clone- and platform-oriented software reuse. In: Devanbu P, Cohen MB, Zimmermann T (eds) ESEC/FSE '20: 28th ACM joint European software engineering conference and symposium on the foundations of software engineering, virtual event, USA, November 8–13, 2020. ACM, pp 432–444. https://doi.org/10.1145/3368089.3409684

Krüger J, Mahmood W, Berger T (2020) Promote-pl: a round-trip engineering process model for adopting and evolving product lines. In: Lopez-Herrejon RE (ed) SPLC '20: 24th ACM international systems and software product line conference, Montreal, Quebec, Canada, October 19–23, 2020, vol A. ACM, pp 2:1–2:12. https://doi.org/10.1145/3382025.3414970

Lopez-Herrejon RE, Illescas S, Egyed A (2018) A systematic mapping study of information visualization for software product line engineering. Journal of Software: Evolution and Process 30(2). https://doi.org/10.1002/smr.1912

Lupton R, Allwood J (2017) Hybrid sankey diagrams: Visual analysis of multidimensional data for understanding resource use. Resources, Conservation and Recycling 124:141–151. https://doi.org/10.1016/j.resconrec.2017.05.002

Mathieson K, Peacock E, Chin WW (2001) Extending the technology acceptance model: the influence of perceived user resources. Data Base 32(3):86–112. https://doi.org/10.1145/506724.506730

McKenna S, Mazur D, Agutter J, Meyer MD (2014) Design activity framework for visualization design. IEEE Transactions on Visualization and Computer Graphics 20(12):2191–2200. https://doi.org/10.1109/TVCG.2014.2346331

Montalvillo L, Díaz O, Azanza M (2017) Visualizing product customization efforts for spotting SPL reuse opportunities. In: Proceedings of the 21st international systems and software product line conference, SPLC 2017, Volume B, Sevilla, Spain, September 25–29, 2017, pp 73–80. https://doi.org/10.1145/3109729.3109737

Nagamine M, Nakajima T, Kuno N (2016) A case study of applying software product line engineering to the air conditioner domain. In: Mei H (ed) Proceedings of the 20th international systems and software product line conference, SPLC 2016, Beijing, China, September 16–23, 2016. ACM, pp 220–226. https://doi.org/10.1145/2934466.2934489

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Roman G, Griswold WG, Nuseibeh B (eds) 27th International conference on software engineering (ICSE 2005), 15–21 May 2005, St. Louis, Missouri, USA. ACM, pp 284–292. https://doi.org/10.1145/1062455.1062514

Novais RL, Torres A, Mendes TS, Mendonça MG, Zazworka N (2013) Software evolution visualization: a systematic mapping study. Information and Software Technology 55(11):1860–1883. https://doi.org/10.1016/j.infsof.2013.05.008

Ohlsson M, von Mayrhauser A, McGuire B, Wohlin C (1999) Code decay analysis of legacy software through successive releases. In: 1999 IEEE aerospace conference. Proceedings (Cat. No.99TH8403), vol 5, pp 69–81. https://doi.org/10.1109/AERO.1999.790190

Pohl K, Böckle G, van der Linden F (2005) Software product line engineering - foundations, principles, and techniques. Springer. https://doi.org/10.1007/3-540-28901-1

Reddivari S, Rad S, Bhowmik T, Cain N, Niu N (2014) Visual requirements analytics: a framework and case study. Requirements Engineering 19(3):257–279. https://doi.org/10.1007/s00766-013-0194-3

Reddivari S, Rad S, Bhowmik T, Cain N, Niu N (2014) Visual requirements analytics: a framework and case study. Requirements Engineering 19(3):257–279. https://doi.org/10.1007/s00766-013-0194-3

Schackmann H, Lichter H (2006) A cost-based approach to software product line management. In: International workshop on software product management, IWSPM '06, Minneapolis/St.Paul, Minnesota, USA, September 12, 2006. IEEE Computer Society, pp 13–18. https://doi.org/10.1109/IWSPM.2006.1

Schmidt M (2008) The sankey diagram in energy and material flow management. Journal of Industrial Ecology 12(2):173–185. https://doi.org/10.1111/j.1530-9290.2008.00015.x

Schulze S, Schulze M, Ryssel U, Seidl C (2016) Aligning coevolving artifacts between software product lines and products. In: Proceedings of the tenth international workshop on variability modelling of software-intensive systems, Salvador, Brazil, January 27–29, 2016, pp 9–16. https://doi.org/10.1145/2866614.2866616

Sein MK, Henfridsson O, Purao S, Rossi M, Lindgren R (2011) Action design research. MIS Quarterly 35(1):37–56. http://misq.org/action-design-research.html

Sjøberg DIK, Anda B, Arisholm E, Dybå T, Jørgensen M, Karahasanovic A, Koren EF, 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. https://doi.org/10.1109/ISESE.2002.1166921

Subramanyam V, Paramshivan D, Kumar A, Mondal MAH (2015) Using sankey diagrams to map energy flow from primary fuel to end use. Energy Conversion and Management 91:342–352. https://doi.org/10.1016/j.enconman.2014.12.024

Takebe Y, Fukaya N, Chikahisa M, Hanawa T, Shirai O (2009) Experiences with software product line engineering in product development oriented organization. In: Muthig D, McGregor JD (eds) 13th International conference on software product lines, SPLC 2009, San Francisco, California, USA, August 24–28, 2009, Proceedings, ACM, ACM International Conference Proceeding Series, vol 446, pp 275–283. https://dl.acm.org/citation.cfm?id=1753273

Tenev VL, Duszynski S, Becker M (2017) Variant analysis: Set-based similarity visualization for cloned software systems. In: Proceedings of the 21st international systems and software product line conference, SPLC 2017, Volume B, Sevilla, Spain, September 25–29, 2017. ACM, pp 22–27. https://doi.org/10.1145/3109729.3109753

Ursachi G, Horodnic IA, Zait A (2015) How reliable are measurement scales? External factors with indirect influence on reliability estimators. Procedia Economics and Finance 20:679–686. https://doi.org/10.1016/S2212-5671(15)00123-9

van van Rossum G (2018) Unified diff format. http://www.artima.com/weblogs/viewpost.jsp?thread=164293. Accessed 15 Jan 2018

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. https://doi.org/10.1109/RE.2009.32

Zhang B, Becker M, Patzke T, Sierszecki K, Savolainen JE (2013) Variability evolution and erosion in industrial product lines: a case study. In: Kishi T, Jarzabek S, Gnesi S (eds) 17th International software product line conference, SPLC 2013, Tokyo, Japan - August 26–30, 2013. ACM, pp 168–177. https://doi.org/10.1145/2491627.2491645

Authors and Affiliations

Oscar Díaz[1] · Leticia Montalvillo[1] · Raul Medeiros[1] · Maider Azanza[1] · Thomas Fogdal[2]

Oscar Díaz
oscar.diaz@ehu.eus

Leticia Montalvillo
lmontalvillo@ikerlan.es

Raul Medeiros
raul.medeiros@ehu.eus

Thomas Fogdal
tfogdal@danfoss.com

[1] University of the Basque Country (UPV/EHU), San Sebastián, Spain

[2] Danfoss, Gråsten, Denmark