Promoting Design Knowledge Accumulation Through Systematic Reuse: The Case for Product Line Engineering

Oscar Díaz
University of the Basque Country (UPV/EHU)
oscar.diaz@ehu.eus

Haritz Medina
University of the Basque Country (UPV/EHU)
haritz.medina@ehu.eus

Jeremías P. Contell
University of the Basque Country (UPV/EHU)
jeremias.perez@ehu.eus

Abstract

*DSR raises concerns about Design Knowledge (DK) accumulation across distinct projects. We believe that DK and the artifact(s) that flesh it out, are the two sides of the same coin, to the extent that, for DK accumulation to thrive, artifacts should come along. With these premises, and with a focus on software artifacts, we advocate for complementing the relevance-design-rigor cycles with a fourth step: refactoring. By 'refactoring' is meant the effort that goes in making the design artifact fit to evolve. Specifically, we advocate for artifact development to introduce reuse considerations: development-**by**-reuse permits to start for reusable code, while development-**for**-reuse allows for artifact customization to be merged back to the reuse platform, and hence, making it available to subsequent projects. By intertwining "**for** reuse" and "**by** reuse", a reuse platform gradually emerges that expands beyond a single DSR project, and in so doing, becomes the artifact counterpart of the DK accumulation repository. We operationalize this vision through Product Line Engineering (PLE). This software development methodology advocates for systematic reuse by putting the focus on a family of artifacts rather than on one-off artifacts. This work describes the efforts so far on adopting PLE to explore a design region along with three DSR projects, each with its own artifact, yet similar enough to conform a product family.*

1. Introduction

Design Science Research (DSR) aims to come up with Design Knowledge (DK), i.e., means-end relationships between the problem and solution space [1]. In a recent report, Vom Brocke et al. regret that "most studies focus on a single DSR project, aiming at deriving DK within this project, while knowledge accumulation and evolution across projects are rarely considered as an antecedent or contribution of the project" [2]. We conjecture that this might be partially due to the limited reuse of design artifacts. Unlike commercial artifacts, design artifacts are not an end in themselves but a means to advance DK. Here, artifacts are the carriers of DK's *mechanisms*, i.e., the means that "either lead to or allow users ... to accomplish some aim" [3]. Mechanisms are the way through which DK aims to impact a relevant problem. If DK is essentially evolutionary, so should it be the underlying artifacts as the necessary bearers of DK's mechanisms. Accordingly, if the artifacts are 'rigid', then the underlying DK will be more difficult to be accumulated by other DSR projects. This makes previous authors to distinguish between *fitness-for-use* (i.e., the ability of the design artifact to perform in the current application context with the current set of goals in the problem space) from *fitness-for-evolution* (i.e., the ability of the solution to adapt to changes in the problem space over time). This distinction was enshrined by Gill and Hevner when positing that "the evolutionary fitness of a design artifact is more valuable than its immediate usefulness" [4]. This situation is especially vivid for software artifacts (hereafter just 'artifacts').

When it comes to software development, two common practices might jeopardize *fitness-for-evolution.* First, researchers often make suboptimal development decisions to allow them to get to the evaluation quickly, get feedback, and gain *fitness-for-use*. Speeding up time-to-evaluate might well play the role of time-to-market in the commercial world. Here, suboptimal development decisions lead to the so-called *technical debt*, i.e., the accumulated backlog of software development needed because developers favour a quick solution over a 'fitter solution', usually to reduce the overall implementation time [5]. Second, artifact reuse among DSR projects is frequently achieved via clone&own. Here, a new product starts by cloning an existing one, and next, adapts parts of it to meet the new requirements. Although cost-saving in the short run, clone&own is hardly scalable if the track about the mechanisms existing in several clones is lacking [6]. As a result, clone&own increases

URI: https://hdl.handle.net/10125/71280
978-0-9981331-4-0

'DK entropy': different projects P1, P2,...,Pn might explore nearby design regions adding eventually new stakeholders, goodness criteria or evaluation settings, but conducted upon distinct artifacts: A1, A2,..., An. These artifacts are similar insofar as they might be obtained through cloning, yet their code mechanisms are dispersed as their variations are difficult to trace and compare. In short, technical debt together with *clone&own* practices might lead to *design debt*, i.e., deferring a holistic understanding of the underlying design principles.

If we draw parallels with manuscripts, Systematic Literature Reviews (SLRs) follow protocols to ensure a *systematic* approach to knowledge accumulation. Likewise, if DSR artifacts are regarded as DK holders, then software reuse should also be *systematic* to facilitate DK accumulation. If so, the artifacts A1, A2,..., An are not obtained by cloning but systematically derived. By 'systematic' it is meant that a vision, roles and processes are in place to ensure that reuse is facilitated. This turns A1, A2,..., An from being independent products to become a *product family*. Development wise, this notion of 'family' implies a clear distinction between Domain Engineering (i.e., where the platform is handled through "development for reuse") and Application Engineering (i.e., where specific products are derived from the common platform with a focus on "development by reuse"). In other words, design artifacts are handled as a portfolio of related products using a shared platform and an efficient means of production, i.e., using Product Line Engineering (PLE) [7].

This paper tunes PLE for DSR as a systematic approach to design artifact reuse. Contributions rest on:

- introducing a design process where "for reuse" and "by reuse" intertwine along a chain of DSR projects to achieve *fitness-for-evolution* and *fitness-for-use*, respectively (Section 3),
- operationalizing this process along PLE (Section 5),
- providing a pilot study (Section 6).

2. Background

Fit artifacts allow to be reused and extended to settings other that those originally considered, and in so doing, increase projectability (e.g., broader applicability scope) and confidence (e.g., sounder evaluation) [2]. Efforts have been made to enhance fitness for distinct sorts of DSR artifacts:

- for processes, Winter introduces a method for Situational Method Engineering to be adapted/extended to allow for the systematic design of certain tasks considering specific contexts and goals in the area of Information System Management [8],
- for conceptual models, Vom Brocke and Buddendick tackle the requirements for reusable conceptual models where the distinction between 'for reuse' vs. 'by reuse' is also highlighted [9],
- for software, frameworks and configuration approaches have been proposed [10].

We argue that neither frameworks nor configurations fully account for DSR needs. First, *frameworks* limit extensions to actuate upon the hot spots foreseen by the framework [11]. This might compromise future requirements that might not be accommodated through the envisioned hot spots. Alternatively, *configuration* captures variations within if-then statements where 'if' conditions are evaluated at run time against a configuration file. Here, a single artifact handles all functionality, no matter whether it is statically known that a configuration option will never be selected in a certain scenario. From a DSR perspective, this option hinders the exploration of the problem space through *separated* artifacts rather than a *single* artifact with all configurations built in. When configuration stands for distinct design criteria, configuration dependencies might need to be checked. This requires additional code that interferes with those supporting the configured functionality itself. Besides hindering maintenance, code tangling challenges the smooth exploration of distinct parameter configuration bundles for evaluation purposes. Alternatively, configuration can be moved from run time to compile time through *Conditional Compilation*. Here, variants are enclosed within *#ifdef* and *#endif* marks, and associated with precompilation directives, i.e., Boolean expressions upon "configuration parameters". The important point is that so-marked code is conditionally removed before compilation. The *cpp* preprocessor is a case in point [12]. Conditional compilation just delivers the code that is needed for the selected features. And, what is also relevant, configuration-dependency checking is outsourced from the application code to dedicated configurators. Yet, reusability is not only a matter of programming effort but of being *systematic*, i.e., identifying, understanding and managing the set of process and roles that interplay in making software reusable. This is when *Product Line Engineering* (PLE) comes into play.

PLE aims to identify commonality and variability among applications *within a domain*, and build reusable assets to benefit future development efforts [7]. In

PLE, the product plays an ancillary role in favour of the notion of 'domain'. A domain is an area of knowledge that is scoped to maximize the satisfaction of the requirements of its stakeholders [7]. The result is a Software Product Line (SPL), i.e., "a set of applications 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" [13]. DSR wise, we rephrase this definition by describing a SPL as *a set of design artifacts sharing a common managed set of DK's mechanisms that satisfy the specific needs of a particular design region and that are developed in a prescribed way*. But, what sort of prescription? This moves us to the next section.

3. Fit design as a Continuous Improvement practice

'Fitness' departs from 'utility' in its prospective, imaginative function [4]. Utility might be evaluated w.r.t. current conditions. By contrast, 'fitness' is (partially) evaluated w.r.t. foreseen condition. The current conditions are frequently difficult to be accurately apprehended, yet alone estimations about future requirements. We recognize this difficulty in coming up with fit artifacts, and hence, put the focus on the *process* that might lead to fit artifacts. We might ignore what a 'fit artifact' is, but we can provide the means and appreciate the practices that increase the chances to come up with fit artifacts. This vindicates a shift from *the result* (i.e., the artifact) to *the process* (i.e., the design). This is when Continuous Improvement comes into play [14].

Continuous Improvement (CI) was born as a management practice for organizations need to be fit, i.e., promptly responding to changing customer needs, market changes and competition threats. This description bears resemblance to DSR challenges: "both problem and solution spaces are subject to constant and increasing change, so that past DK is prone to rapid aging, ... and, hence, DK requires constant updates in the form of revision and further evolutionary development" [2].

Based on this resemblance, we can look at CI in the search for fit artifacts. Specifically, CI sustains that a desired result is achieved more effectively when related resources and activities are managed as a process [15] . This process commonly intermingles two loops: the improvement cycle and the standardizing cycle [14]. The former goes along the "plan-do-check-act" (PDCA) cycle. *Plan* refers to setting a target for improvement. *Do* means implementing the plan. *Check* is the control for effective performance of the plan. Finally, *Act* refers to standardizing the new (improved) process and setting targets for a new improvement cycle. As the resulting work process, following each cycle of improvement, becomes unstable due to the nature of change, a second cycle is, therefore, required to stabilize it: the "standardizing cycle" that goes along the "standardize-do-check-act" (SDCA) cycle. The main purpose of this cycle is "to iron out abnormalities in the resulting work process and bring it back to harmony before moving to a new improving cycle" [15].

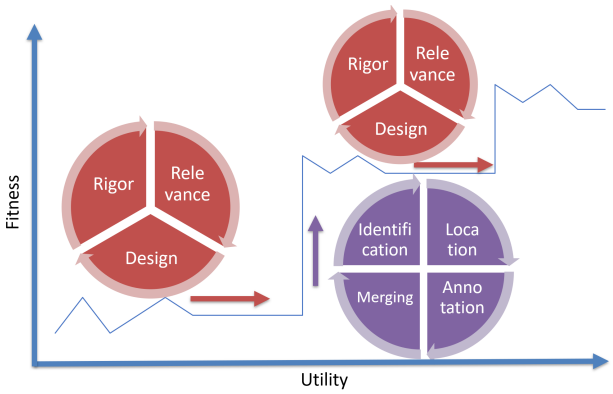

Figure 1. Fit-minded design processes

Accordingly, we advocate for a similar approach to *fit-minded design processes* (see Fig. 1):

- the Utility Cycle (i.e., the counterpart of the PDCA cycle), which stands for the activities associated with relevance-design-rigor [16],
- the Fitness Cycle (i.e., the counterpart of the SDCA cycle) which complements the previous one with 'refactoring' activities, i.e., tuning the artifact to be eventually reused, adapted or appropriated by scenarios/developers other than those originally considered.

Broadly, the Utility Cycle ends with an evaluation about the utility of the artifact and some design principles that are abstracted out of the experience (i.e., *fitness-for-use*). At this point, the development team faces a crossroad. On the way towards DK accumulation, Vom Brocke et al. introduce the metaphor of a journey along a three-dimensional space: *projectability* of the problem in the problem space, *fitness* of the solution in the solution space, and *confidence* in the current evaluation evidence. This journey is marked by the development of different DSR artifacts that explore distinct stakeholders, contexts or related practices (i.e., a design region). Traditionally, these DSR artifacts tend to be kept separated where reuse is frequently conducted through *clone&own*.

We depart from this scenario in two ways. First, we advocate for transiting this three-dimension space through intertwining 'utility cycles' (advancing projectability and confidence) and 'fitness cycles' (advancing fitness). Second, this process is conducted not through *clone&own* but systematic reuse (i.e., PLE). At the onset, a first artifact *A* is engineered for variability, resulting into a fitter *A'*. By intertwining 'utility cycles' and 'fitness cycles', additional artifacts fleshed out distinct DK advances, resulting into a set of artifacts A', A", A"', etc. Rather than keeping each artifact apart, 'a platform' is gradually generated in the fitness cycle. This platform collects commonalities and variabilities in the design space in terms of variation points (aka features). This platform is the Software Product Line (SPL). The main premise is that most developed design artifacts are not brand-new artifacts but rather variants of other artifacts within the same design region. Hence, and except for the very first iteration, DSR artifacts are obtained out of the SPL.

To be effective as an accumulation mechanism, SPLs should be feedbacked from the insights gained in adapting the SPL artifacts to scenarios other than those initially considered by the SPLs. Artifact developers branch off the SPL codebase and adapt the core code to account for unexplored design regions. Once these regions have been explored ('utility cycle'), and the mechanisms have been accordingly adapted/created and evaluated, the 'fitness cycle' cares about merging back these new developments into the main SPL branch. The vision is then for *SPLs to embody DK that goes beyond a design artifact to include a set of artifacts, i.e., a product family, and, in so doing, facilitates DK accumulation for a given design region*. Next, we introduce a pilot study.

4. Pilot study: *Review&Go*

Review&Go tackles the lack of software scaffolds for peer reviewing. This project was conducted using DSR and presented at [17]. Fig. 2 outlines the main constructs using the Inner/Outer Models [18]. The *Inner Model* refers to the Justificatory Theory that introduces the variables to act upon. *Review&Go* is informed by theories on quality feedback. Four main independent variables are introduced: specific (i.e., pointing to paragraphs in the manuscript where the feedback applies), timely (i.e., provided in time along the conference/journal deadline), contextualized (i.e., framed with reference to methodological criteria of ample support within the community), and selective (i.e., commenting in reasonable detail, distinguishing major concerns from minor concerns that can be corrected). On the other hand, the *Outer Model* describes how independent variables can be manipulated through an IT artifact (i.e., *Review&Go*), and how dependent variables are measured (e.g., TAM for reviewers). The resulting DK is tentatively abstracted in terms of Design Principles (see Table 1).

So far, evaluation is based on the notion of utility as usefulness (e.g., effectiveness in performing the review). From this perspective, *Review&Go* might be tentatively evaluated as useful. Yet, Gill et al. introduce an additional utilitarian perspective, i.e., that of evolution [4]. *Review&Go* can be deployed (reproduced) across the reviewers' desktop, but to what extent can *Review&Go* be useful to explore the design landscape of quality feedback beyond peer review? The journey ahead might transit along two main dimensions: the confidence dimension and the projectability dimension [2].

The confidence dimension. One possible follow-on is to ascertain which mechanisms of those already studied have a greater impact on reviewers' acceptance. *Review&Go* was evaluated as a whole. However, Niehaves et al. advise for a more piecemeal approach to ascertain how each mechanism ponders the final result, and whether inter-dependencies of simultaneously implemented mechanisms might exist [18]. In this respect, we might be interested in calibrating whether efficiency or efficacy has the strongest impact on *Review&Go* adoption. Even a single latent variable (e.g., timely) might be impacted by different mechanisms (e.g., head-start template, resumption utility) where their impact on the clutterness of the graphical interface might suggest to adopt a single one, hence, leading to a more discriminatory evaluation. Yet, an alternative path would be to consider another stakeholder, i.e., the manuscript authors. When it comes to feedback effectiveness, reviewers mostly discuss feedback design matters like timing, modalities and connected tasks. In contrast, receivers put the stress on "comments being usable, detailed, considerate of affect and personalised" [19]. This vindicates the interest in evaluating *Review&Go* but now from the perspective of authors.

The projectability dimension. So far, *Review&Go's* insights are very specific to the context of peer reviewing. We can increase projectability by introducing additional stakeholders, e.g., editors. Editors are encouraged to take an active role in the review process by engaging with reviewers and critically evaluating their reports [20]. Yet, and unlike reviewers, editors analyse the manuscript through the lenses of the reviewers. The manuscript is already commented, and the editor duties include assessing, pondering, blending and framing reviewers' comments. These

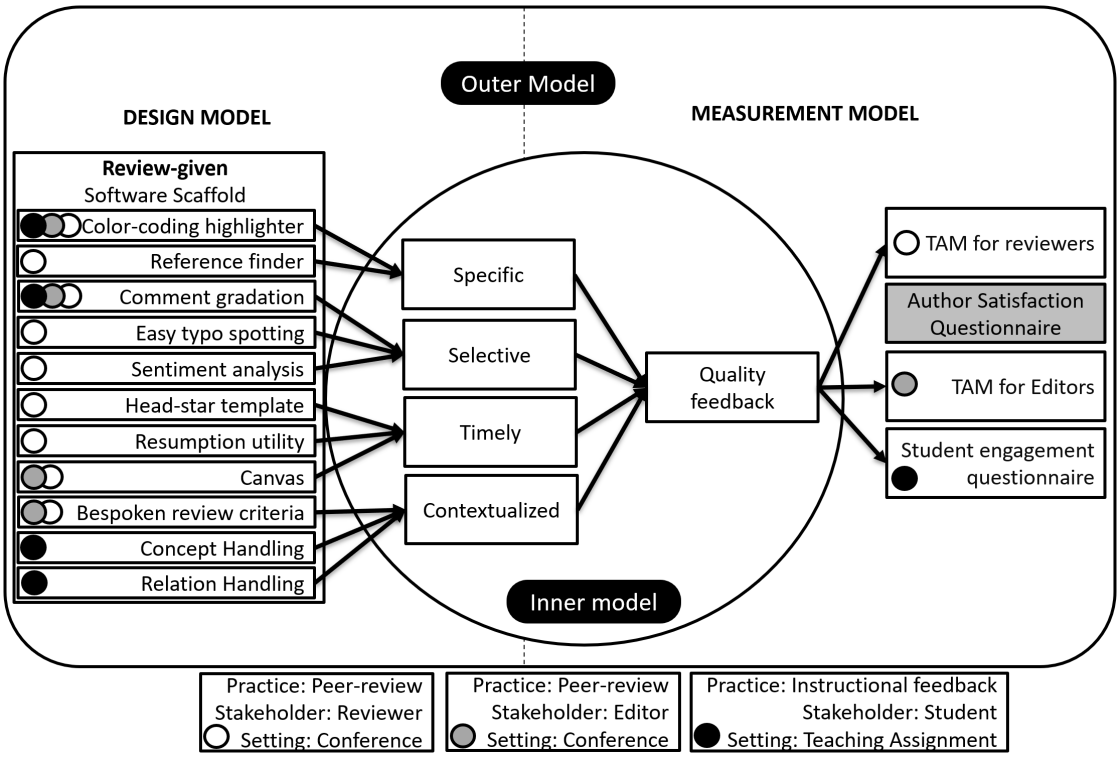

Figure 2. 'Accumulated' Inner/Outer Model: same Inner Model but different Outer Models for three different projects (bottom of the figure)

Table 1. Review&Go. Design Principles along the schema in [3].

Aim, Implementer, User	Context	Mechanism
To allow reviewers (enabler) to tune review frameworks to journal aims (user)	Peer review as a gate-keeping activity	Color-coding highlighter
To allow reviewers (enabler) to provide constructive feedback to authors (user)	Peer review as a manuscript-improvement activity	Comments with gradation & Reference Finder & Sentiment Analysis
To allow reviewers (enabler) to ponder of the manuscript's (de)merits	Peer review as a grading activity	Canvas
To allow reviewers (enabler) to speed up report writing to meet the Journal deadline (user)	Peer review as a time-intensive activity	Text generation based on boiler-plate templates
To allow reviewers (enabler) to provide constructive feedback to authors (user)	Peer review as a fragmented activity	Resumption facility

activities might also be characterized as a (meta)review endeavour, although with some adjustment since now the input is not just a manuscript but a commented manuscript.

Complementarily, we can enhance projectability by considering brand-new contexts where feedback also plays a major role. Student assessment is a case in point. Here, "review" is conducted by students as an instructional means to capture the essence of someone's else writing. This frames this practice as a review activity, hence, rising an opportunity for *Review&Go* reuse. As expected, some specifics emerge: the review is not a critical assessment but a comprehension assessment. To this end, the output is not a textual review but a concept map. The relevance and theoretical underpinning of this approach comes from the area of conceptual mapping [21]. This setting introduces new societal needs (e.g., student accessibility) and technical issues (e.g., transparent integration with popular mapping tools like CMap Tools) that put existing *Review&Go*'s DK at play.

Regardless of the objective pursued (i.e., confidence,

projectability), *Review&Go* is taken as the starting point. This implies not only to capitalize upon *Review&Go*'s DK but also to tap into *Review&Go*'s codebase. This is in accordance with our vision that for DK accumulation to thrive, DSR artifacts should come along. This calls for artifacts to be fitted for the journey ahead. Unfortunately, this was not the case for *Review&Go*.

Conceived as a monolithic application, *Review&Go*'s mechanisms were difficult to isolate. This was needed to either conduct a discriminatory evaluation or to adapt *Review&Go* to distinct settings. Our first attempt was to *clone&own Review&Go*. This meant a different clone for each possible journey: a *Review&Go* for exploring the editor needs, a *Review&Go* for looking at the student instructional needs, and so on. Yet, this resulted in divergent projects, missing opportunities to share code and insights about the phenomenon at hand, i.e., quality feedback. Some first attempts were made to try to componentize *Review&Go*, yet we utterly fail since most of the mechanisms could not be isolated as single components. Rather, the realization of DK mechanisms frequently crosscuts different functional units (e.g., files, classes, methods), being tangled and scattered throughout the codebase.

In short, in the search for quick time to evaluate, *Review&Go* postponed (un)intentionally refactoring efforts. Yet, this *technical debt* might never be paid if mechanisms do not need to be reused. In other words, reusability efforts do not need to be conducted for no matter the mechanism but just opportunistically for those mechanisms that might eventually pay off. Hence, the Fitness Cycle includes exploring fertile design regions, and making informed decisions on what reusability efforts should be incurred or paid off, and when. The next section builds a case for operationalizing the Fitness Cycle as a PLE endeavour.

5. The Fitness Cycle

The Utility Cycle ends up with a design artifact that fleshes out DK's mechanisms reckoned to help users. Yet, this is not the end of the story. We advocate for designers to go one step further and analyze the extent existing mechanisms can be reused in pursuit for exploring nearby design regions. To this end, we follow PLE and its distinction between Domain Engineering and Application Engineering.

5.1. Domain Engineering in DSR

Domain Engineering is the process of analyzing the domain and developing the reuse platform. Here, we distinguish between the problem space and the solution space. The former takes the perspective of stakeholders and their problems, and views of the entire domain [22]. In a DSR setting, the domain stands for a practice where a practical problem arises whose solution is to be mediated through design artifacts. The results of domain analysis are documented in a *Feature Model*. In the PLE literature, a 'feature' stands for "a characteristic or end-user-visible behavior of a software system" [22]. For our purposes, however, we are not interested in all "end-user-visible behavior" but just those aspects that might have an impact on 'utility', i.e., the DK's mechanisms. Therefore, we conceive *a feature as a neat description of a DK's mechanism.*

However, not all mechanisms necessarily become features, or at least, not right away. We previously observed that the Fitness Cycle includes exploring fertile design regions, and making informed decisions on what reusability efforts should be incurred or paid off, and when. Turning a mechanism into a feature might involve a costly refactoring process that should be balanced against opportunities for this cost to pay off. Considerations to be pondered about include:

- current utility, i.e., how the mechanism ranked during the last evaluation of the artifact,
- foreseen utility, i.e., how the mechanism might serve to explore "the design fitness landscape",
- resource availability, including both development (technical skills) and evaluation (subjects to tap into).

Assessing each of these concerns is a research item in its own right. For instance, foreseen utility might be ranged along with four possible values: *re-usable* (i.e., the mechanism can be used as it is), *adaptable* (i.e., the mechanism might need some tuning), *appropriation* (i.e., the mechanism might be used in a way the authors did not intent) or *detrimental* (i.e., the mechanism might be harmful in the new setting). Only mechanisms ranked as 're-usable' are moved unchanged during the refactoring process. They conform "the commonality" of the product line. By contrast, the rest of the options call for the mechanism's code counterpart to be customized or be removed altogether to prevent feature creeping in the new scenario. Variability wise, the mechanism's code counterpart is a candidate to become a feature, i.e., amenable to be identified, annotated and tested as a configurable option at precompilation time. This moves us to the solution space.

The solution space represents the developer's perspective. It covers the design, implementation, validation and verification of feature realization,

```
48 // PVSCL:IFCOND(Canvas, LINE)
49 // Set create canvas image and event
50 let overviewImageURL = chrome.extension.getURL('/images/overview.png')
51 this.overviewImage = this.container.querySelector('#overviewButton')
52 this.overviewImage.src = overviewImageURL
53 this.overviewImage.addEventListener('click', () => {
54   this.generateCanvas()
55 })
56 // PVSCL:ENDCOND
57 // PVSCL:IFCOND(ResumptionFacility, LINE)
58 // Set resume image and event
59 let resumeImageURL = chrome.extension.getURL('/images/resume.png')
60 this.resumeImage = this.container.querySelector('#resumeButton')
61 this.resumeImage.src = resumeImageURL
62 this.resumeImage.addEventListener('click', () => {
63   this.resume()
64 })
65 // PVSCL:ENDCOND
```

Figure 3. Pre-processor directives to annotate variant code for features Canvas and ResumptionFacility.

and their combination in suitable ways to facilitate systematic reuse [22]. Main steps include:

- feature location, which refers to deciding which source code supports a given feature. The difficulties are twofold. First, features tend to be rather domain-specific entities and orthogonal to typical structures found in programs, such as components, classes or methods [22]. They are crosscut, scattered and tangled along the codebase. Second, features are rarely documented, developers' knowledge about the features fades quickly, and developers leave projects. This sustains feature location to be conducted within the same DSR project where the artifact originates, if possible.

- feature annotation, which accounts for documenting the connection between a feature and its implementation. A common technique is the use of pre-processor directives (aka *#ifdefs*). Pre-processor directives are expressed in terms of Boolean expressions upon feature constants. Fig. 3 shows an annotated *#ifdef* from *Review&Go*. This allows switching features on and off when deriving individual variants from the product line.

5.2. Application Engineering

During Application Engineering, the needs of a specific DSR project are expressed in terms of existing DK mechanisms ready for re-use, i.e., features. Characterizing a DK's mechanism as a feature implies that artifacts can be derived in terms of these features (aka product configuration). Based on conditional compilation, *#ifdefs* directives can be used to filter out optional code. Automation tools (e.g., Ant) are used to generate different artifacts based on the feature selection.

However, in a DSR setting, existing features rarely fully satisfy the demands of the new DSR project. Features might need to be tuned while brand-new features might need to be introduced. That is, 'developing with reuse' provides a head-start, but it does not remove the need for artifact customization. Therefore, to be effective as an accumulation mechanism, SPLs should be feedbacked from this customization, i.e., extend the initial 'domain' with scenarios other than those initially considered by the SPL. This feedback makes SPLs depart from clone&own insofar as customization branches do not persist dangling but end up being merged back to the reuse platform. It is this very merging effort that makes the SPL become the container for mechanisms that expand along a design region, i.e., the SPL domain. It is from this perspective that we regard SPLs as main enablers of DK accumulation.

6. Pilot study: from *Review&Go* to *Feedback&Go*

At the onset, no SPL existed but an one-off artifact, i.e., *Review&Go*. We started by identifying and extracting features from this artifact.

6.1. Domain Engineering

We first analyze which *Review&Go* mechanisms are worth being turned into features in terms of current utility, available resources and foreseen utility (see Table 2).

- current utility. *Review&Go* resorts to TAM's relative advantage as a proxy for 'utility' (refer to [17]). A Likert scale was used where a metric is calculated as $(\#Agree + \#StronglyAgree)/\#Subjects$. The values 'High', 'Medium' and 'Low' are assigned if the result is above 0.6, between 0.6 and 0.4, or below 0.4, respectively,

- available resources. Besides the difficulty of finding relevant scholars that might want to participate as subjects, we focus on the availability of technical expertise. This might be an issue in the academia where artifact development mainly rests on the back of PhD students. In our case, just two of the three authors of *Review&Go* were available, making

Table 2. Looking for feature candidates

Mechanism	Current Utility	Resource Availability	Foreseen utility: the 'editor' stakeholder	Foreseen utility: instructional feedback
Highlighter	High	High	Re-usable	Re-usable
Comment Gradations	High	High	Adaptable	Appropriation
Reference finder	Medium	Medium	Re-usable	Detrimental
Sentiment Analysis	Low	Low	Re-usable	Detrimental
Typo	High	High	Re-usable	Detrimental
Canvas	Medium	Medium	Re-usable	Re-usable
Boiler-plate templates	High	Medium	Detrimental	Detrimental
Resumption facility	Medium	Medium	Detrimental	Re-usable

risky additional developments upon Sentiment Analysis.

- foreseen utility. Each mechanism is pondered for its potential utility. As reflected in Table 2, the Editor scenario is close to *Review&Go*, where the editorial practice is regarded as a sort of meta-review. Hence, most *Review&Go* mechanisms can be reused as such. By contrast, the Instructional scenario focuses on a design region far distant from the *Review&Go's*, leading to an eventual refactoring of six mechanisms. Certainly, the farther the region, the larger the refactoring effort.

At this time, the designer can consider one or more of the foreseen scenarios. We decided to consider both. This implied to refactor all mechanisms as features except 'Highlighter' and 'Canvas'. That is, *ifdef* blocks were annotated that permit to filter mechanisms out at compile time. This resulted in *Review&Go':* same functionality, yet *Review&Go'* is *fitter* than *Review&Go*. That is, *Review&Go'* can evolve into artifacts that might exhibit or not certain features. If all features are selected, then *Review&Go* is assembled black. Yet, some features might be deliberately left out, giving rise to simpler artifacts to e.g., ascertain the impact on adoption. In short, *Review&Go'* accounts for a set of products. To highlight this fact, we rename *Review&Go'* as *Feedback&Go* to denote a nascent platform for exploring the feedback-giving landscape.

6.2. Application Engineering

Even at this very early stage, *Feedback&Go* accounts for varied artifacts. Different feature combinations result in distinct artifacts. Rather than clone&own or start from scratch, developers can now cherry-pick those features whose code is to be included in the onset codebase, branching off from *Feedback&Go*. From then on, developers can customize the codebase to account for their scenarios' specifics along the Utility cycle. Table 3 outlines the case for the *Concept&Go* project, indicating features reused-as-is, features adapted, and new features [23].

Table 3. Concept&Go as a product of the Feedback&Go SPL: features reused-as-is, features adapted, and brand-new features.

Type of features	Feature	LOC	Percentage of reuse
Reused features	Highlighter	9,476	100%
Modified features	Comment Gradation, Resumption facility	1,263	78.23%
New features	Concept handling, Relation handling	2,904	0%
Total LOC		**13,444**	**76.35%**

Once the Utility Cycle for the *Concept&Go* project is over, a new Fitness Cycle starts: Concept&Go is merged back to the *Feedback&Go* platform. This moves us back to Domain Engineering (round two). Besides brand-new features as in round one, Concept&Go contributes by customizing features already present in the SPL. This involved important refactoring. In this way, *Feedback&Go*'s Feature Model grows in width (distant scenarios, new features) and depth (nearby scenarios, adapted features) as new design territories are explored. This turns *Feedback&Go* into the artifact counterpart of the Inner/Outer Accumulative Model in Fig. 2 that collects insights from three different DSR projects.

7. Discussion

By drawing a parallel with CI, we advocate for Hevner's Utility Cycle to be intertwined with the Fitness Cycle with a focus on refactoring and systematic reuse (see Fig. 4). Development wise, and along PLE, this is realized by the interplay of development-by-reuse with

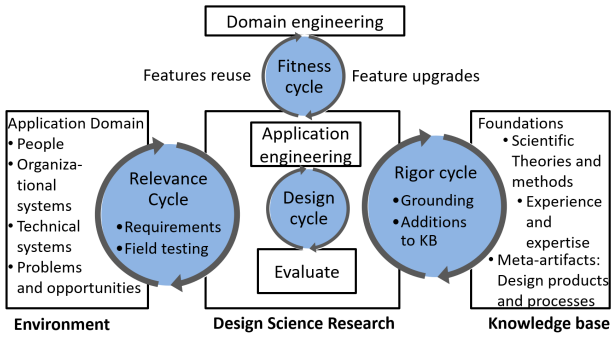

Figure 4. Hevner's relevance-design-rigor cycles complemented with the fitness cycle

development-for-reuse. In this way, a reuse platform (i.e., an SPL) gradually emerges that expands beyond a single DSR project. From this perspective, PLE might facilitate:

- DK accumulation, by describing distinct artifacts along with a common Feature Model,
- DK confidence, by the fact that artifacts using the same mechanism/feature are underpinned by the very same codebase,
- DK evolution, by speeding it up through systematic reuse.

While systematic reuse is efficient, it could prevent brand-new ways of approaching and solving problems. PLE reduces the entry barrier to explore nearby regions by providing a head-start. Yet, if the region to be explored is distant, PLE might result in a technology/software 'lock-in'. Here, clone&own might represent a more libertarian option out of the PLE directives. Yet, this frame is precisely what permits distinct mechanisms to be compared, leading to knowledge accumulation. Hence, even if clone-and-own is used, knowledge accumulation will still require comparing distinct artifacts, no matter how these artifacts were developed.

Finally, although this paper puts the focus on coding, systematic reuse is also a managerial effort. Three stakeholders can be identified: the management, which initiates the reuse initiative and monitors the costs and benefits; the development for reuse team (aka domain engineering), which is responsible for producing, classifying, and maintaining reusable assets; and the development with reuse team (aka application engineering), which is responsible for producing applications using reusable assets [22]. In an academic setting, PhD supervisors can play the role of management, as they agent who pose the long-run vision and domain knowledge to lead the process. As for students, they start developing *with reuse* by capitalizing on their predecessor's code. Yet, they should be systematic while coding also *for reuse* to contribute to the common core, mainly during the last stages of their PhDs. In this respect, the *Review&Go* experience had to face some worth-noticing issues from PhD students, namely:

- questioning developing *for reuse*. For DSR, artifacts are a must, but not 'fit artifacts'. Fit artifacts cause additional costs (i.e., refactoring). Benefits, however, are not always obtained from those who suffer the costs. Research is frequently measured in terms of publications. Doing any work beyond the basic artifact, although beneficial for the research group as a whole, might take time from another publication. This requires adequate policies where artifact reuse is recognized in terms of manuscript authorship or acknowledgement.
- questioning developing *with reuse*. Easy access to existing software does not necessarily increase software reuse. Sometimes, modifying and adapting software can be more expensive than programming the needed functionality from scratch. For instance, researchers might be reluctant to reuse someone else's code if quality documentation is not in place.

Here, supervisors play a pivotal role as 'the conveyor belt' among DSR projects, putting in place the development practices (and incentives) for building up trustworthiness and compensation policies. For the *Review&Go* project, developing *for reuse* was promoted by awarding authorship to the authors of artifacts which were able to be customized for another DSR project. Notice that the compensation is not for mere usage but customization. A DSR artifact might be open source (promoting usage), yet not engineered for variability. However, if this effort was made, and this facilitates the design journey to another PhD student, the first student is rewarded. This also highlights the importance of rightly ascertaining which would be the most likely next design research region to explore. Making the effort of developing for reuse does not ensure the artifact will be actually (re)used. This very much depends on fittingly choosing which DSR mechanisms are turned into features. This introduces an incentive for final-year PhD students to carefully think about the next follow-ons. PhD students are well placed to indicate the most promising paths for improving projectability and confidence for their artifacts. Rather than an after-thought, paving these paths as features might pay

off, should following PhD students effectively transit these paths.

8. Conclusion

We make the case for Product Line Engineering (PLE) to be incorporated during artifact development. Unlike clone&own, PLE involves an additional refactoring and managerial effort. Yet, we conjecture benefits might go beyond those of software development (e.g., time-to-market, increase product quality, etc.) to include DK accumulation and evolution. Along these lines, we contribute by introducing and illustrating a fit-minded process where Utility Cycles intertwine with Fitness Cycles to produce an artifact: a Software Product Line (SPL). A SPL accounts for a *set* of design artifacts that explore distinct features across a given design landscape (i.e., the SPL's domain). The main take-away then rests on looking at SPLs as the artifact counterparts of DK accumulation, making **systematic** reuse a main enabler of DK accumulation. This sustains the interest in further studying the processes, stakeholders and obstacles of systematic software reuse in DSR projects.

Acknowledgements. This work is supported by Spanish Ministry of Science, Innovation and Universities grant number RTI2018-099818-B-I00 and MCIU-AEI TIN2017-90644-REDT (TASOVA). Haritz Medina enjoys a grant from the University of the Basque Country - PIF17/15.

References

[1] J. Venable, "A framework for design science research activities," in *Emerging Trends and Challenges in Information Technology Management: Proceedings of the 2006 Information Resource Management Association Conference*, pp. 184–187, Idea Group Publishing, 2006.

[2] J. Vom Brocke, R. Winter, A. Hevner, and A. Maedche, "Accumulation and Evolution of Design Knowledge in Design Science Research-A Journey Through Time and Space," *Article in Journal of the Association for Information Systems*, vol. 21, pp. 520–544, 2020.

[3] S. Gregor, L. C. Kruse, and S. Seidel, "The anatomy of a design principle," *Journal of the Association for Information Systems*, 2020.

[4] T. G. Gill and A. R. Hevner, "A fitness-utility model for design science research," *ACM Transactions on Management Information Systems*, vol. 4, no. 2, 2013.

[5] G. Sierra, E. Shihab, and Y. Kamei, "A survey of self-admitted technical debt," *Journal of Systems and Software*, vol. 152, pp. 70–82, 2019.

[6] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake, "Variant-preserving refactorings for migrating cloned products to a product line," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 316–326, IEEE, 2017.

[7] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[8] R. Winter, "Construction of situational information systems management methods," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 3, no. 4, pp. 67–85, 2012.

[9] J. Vom Brocke and C. Buddendick, "Reusable conceptual models–requirements based on the design science research paradigm," in *Proceedings of the First International Conference on Design Science Research in Information Systems and Technology (DESRIST)*, pp. 576–604, Citeseer, 2006.

[10] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[11] A. Mili, R. Mili, and R. T. Mittermeir, "A survey of software reuse libraries," *Annals of Software Engineering*, vol. 5, no. 1, pp. 349–414, 1998.

[12] ANSI, "ISO/IEC 9899:2018 - Programming languages — C," 2018.

[13] P. Clements and L. Northrop, *Software product lines*. Addison-Wesley Boston, 2002.

[14] I. Masaaki, *Kaizen: The key to Japan's competitive success*. McGraw-Hill Education, 1986.

[15] J. Singh and H. Singh, "Continuous improvement philosophy – literature review and directions," *Benchmarking: An International Journal*, vol. 22, pp. 75–119, feb 2015.

[16] A. R. Hevner, "A three cycle view of design science research," *Scandinavian journal of information systems*, vol. 19, no. 2, p. 4, 2007.

[17] O. Díaz, J. P. Contell, and H. Medina, "Performant peer review for design science manuscripts: A pilot study on dedicated highlighters," in *International Conference on Design Science Research in Information Systems and Technology (DESRIST)*, pp. 61–75, Springer, 2019.

[18] B. Niehaves and K. Ortbach, "The inner and the outer model in explanatory design theory: the case of designing electronic feedback systems," *European Journal of Information Systems*, vol. 25, no. 4, pp. 303–316, 2016.

[19] P. Dawson, M. Henderson, P. Mahoney, M. Phillips, T. Ryan, D. Boud, and E. Molloy, "What makes for effective feedback: Staff and student perspectives," *Assessment & Evaluation in Higher Education*, vol. 44, no. 1, pp. 25–36, 2019.

[20] D. P. Newton, "Quality and peer review of research: An adjudicating role for editors," *Accountability in Research*, vol. 17, no. 3, pp. 130–145, 2010.

[21] J. D. Novak and A. J. Cañas, "The theory underlying concept maps and how to construct and use them," tech. rep., Florida Institute for Human & Machine Cognition, 2008.

[22] S. Apel, D. Batory, C. Kästner, G. Saake, S. Apel, D. Batory, C. Kästner, and G. Saake, "Software Product Lines," in *Feature-Oriented Software Product Lines*, pp. 3–15, Springer Berlin Heidelberg, 2013.

[23] X. Garmendia, "Feature-based software development: a case for web annotation-based tools," Master's thesis, Facultad de Informática, Universidad de Murcia, 2020.