# Software Product Lines

**Klaus Pohl and Andreas Metzger**

## 1 Introduction

Software product line engineering (SPLE) has proven to empower industry to develop a diversity of similar systems at lower cost, in shorter time, and with higher quality when compared with the development of single systems [1, 2]. A software product line (also sometimes called software product family) is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets [artifacts] in a prescribed way" [3].

SPLE exploits the commonalities of the different systems (typically called *applications*) belonging to the product line and systematically handles the variation (i.e., the differences) among those applications. *Commonality* is invariant for (i.e., shared by) all product line applications [4]; for example, all mobile phones allow users to make calls. Product line *variability* defines how the different applications of the product line may vary [5]. Product line applications may differ in terms of features and functional and quality requirements they fulfill; for example, some tablet computers may include mobile broadband connectivity, while others may not.

The SPLE paradigm has a strong track record of success in industry. Success stories can be found in textbooks (such as [1, 3, 6]) or in the product line hall of fame of the leading international software product line conference (http://splc.net/fame.html). Reported benefits of SPLE include improved productivity by as much as a factor of 10, increased quality by as much as a factor of 10, decreased cost by as much as 60%, decreased labor needs by as much as 87%, decreased time to market by as much as 98%, and ability to move into new markets in months, not years.

K. Pohl (✉) · A. Metzger
Paluno (The Ruhr Institute for Software Technology), University of Duisburg-Essen, Essen, Germany
e-mail: andreas.metzger@paluno.uni-due.de; klaus.pohl@paluno.uni-due.de

V. Gruhn, R. Striemer (eds.), *The Essence of Software Engineering*,
https://doi.org/10.1007/978-3-319-73897-0_11

In this chapter, we describe the key differences between software product line engineering and the development of single software systems (Sect. 2). In particular, we provide an overview of the activities and techniques used in the two development processes of SPLE (Sects. 3 and 4) and discuss different ways for modeling the variability of software product lines (Sect. 5). Finally, we provide some examples of using variability modeling techniques in non-SPLE settings (Sect. 6).

## 2 Differences Between SPLE and Single System Development

Figure 1 depicts a well-established SPLE framework defined in the European SPLE research projects ESAPS, CAFÉ, and FAMILIES. The framework was adopted as part of the ISO/IEC standard #26550 (“Software and systems engineering: Reference model for product line engineering and management”). It is described in detail in [1].

The SPLE framework highlights the main differences between the development of single systems and software product line engineering: the two complementary development processes (“domain engineering” and “application engineering”) as well as the explicit modeling and management of product line variability (“domain variability model” and “application variability model”).

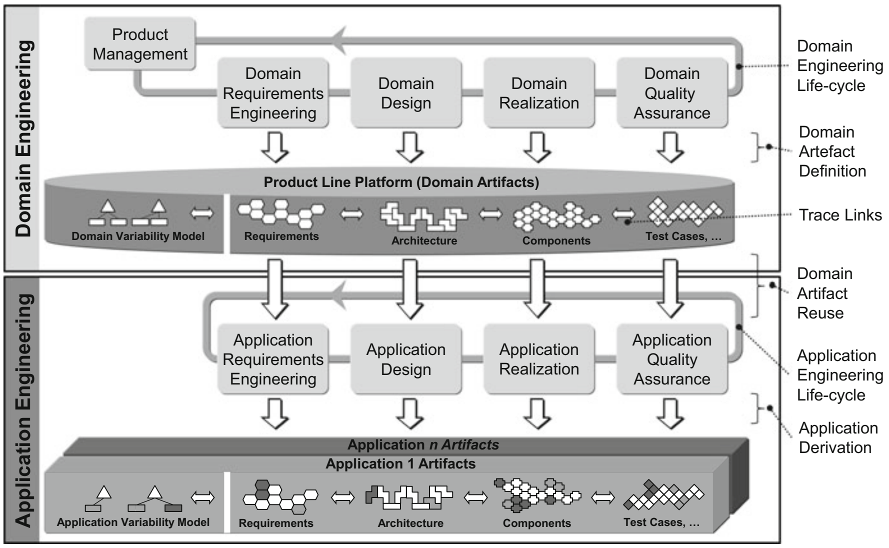

**Fig. 1** SPLE framework (adapted from [1])

## 2.1 Two Development Processes

SPLE differentiates between the following two complementary development processes.

**Domain Engineering** The domain engineering process (shown in the upper half of Fig. 1) is responsible for defining the commonality and the variability of the product line, as well as for developing the *domain artifacts*. Domain artifacts "realize" commonality and variability. They include, among others, requirements artifacts (e.g., use case diagrams, requirements models), architectural artifacts (e.g., component models, class diagrams), implementation artifacts (e.g., source code files, libraries), and test artifacts (e.g., test cases, test data). The *product line platform* encompasses the domain artifacts of the product line. Important parts of the product line platform are the domain requirements and the product line architecture. The product line architecture is often called the *reference architecture* of the product line [1, 6]. We elaborate on the development activities executed during domain engineering in Sect. 3.

**Application Engineering** The application engineering process (shown in the lower half of Fig. 1) is responsible for deriving concrete applications from the domain artifacts. During application engineering, the variability of the domain artifacts is exploited and bound (resolved) according to the needs and requirements of the particular application. Thereby, invariant and variant domain artifacts are reused. We elaborate on the activities during application engineering in Sect. 4.

## 2.2 Product Line Variability

Product line variability is the key, crosscutting concern in SPLE [1, 3]. Product line variability defines how applications of a software product line can differ, for example, in terms of properties, features offered, functions, or qualities offered. Whether a given property is invariant (common) or variable for the applications of the software product line is determined by explicit management decisions, typically made by product management [1, 5]. Product line variability is documented in so-called variability models. The SPLE framework in Fig. 1 differentiates between two types of variability models: domain variability models and application variability models [7].

During domain engineering, the variability of the product line is defined and documented in the domain variability model. In application engineering, the variability defined in the domain variability model is bound in order to fulfill the application-specific requirements. The variability bindings for a specific application are documented in the application variability model.

Product line variability is preplanned in order to fulfill different market and stakeholder needs. Still, application engineers may face the problem that individual customer- or market-specific needs cannot be satisfied by reusing common and variable domain artifacts. In this case, customer- or market-specific extensions or adjustments of the common and variable artifacts are required. The adjustment required can be enabled by initiating a product line evolution (e.g., by introducing additional product line variability) or by adapting the application artifacts and document such adaptation in the application variability model [7]. An application variability model thus documents both the variability bound for the specific application and the application-specific adaptations.

## 2.3 Software Variability Versus Product Line Variability

Software variability refers to the ability of software systems or artifacts to be efficiently extended, changed, customized, or configured [8]. Most modeling and programming languages provide mechanism for software variability. Examples include abstract superclasses allowing different specializations, interfaces facilitating different implementations, or conditional compilation (e.g., using #ifdefs) facilitating the inclusion of different code fragments.

In contrast to software variability, product line variability defines how the applications of a product line can differ. Together with the commonalities, product line variability defines the scope of a product line (see Sect. 3.1). Product line variability is preplanned. Defining whether a given feature, functional or quality requirement is product line variability or not requires explicit decisions from product management or other stakeholders.

Software variability can represent both product line variability as well as commonality. As an example for software variability, take the abstract superclass *Communication* with two concrete subclasses, *WiFi* and *MobileBroadband*, documented in a UML class diagram. Clearly, the superclass together with the subclasses documents software variability. In principle, any of the two or even both subclasses could be used in place of the superclass.

This software variability, on the one hand, can represent a commonality if the stakeholders, for instance, had decided that all product line applications must include both subclasses *WiFi* and *MobileBroadband*—in other words, if the stakeholders have decided that the product line applications cannot differ in terms of the communication used. In such cases, software variability documents commonality of the product line and not product line variability. On the other hand, this software variability could also document product line variability. For example, if the stakeholders had decided that for each application of the product line the engineer has to choose at least one of the two communication subclasses, the applications could differ in terms of the subclasses they include for communication.

Consequently, product line variability cannot be automatically derived from software variability. In other words, product line variability cannot be identified by analyzing software variability documented in existing software artifacts or models. Thus, defining and modeling product line variability requires additional modeling concepts (see Sect. 5).

## 3 Domain Engineering

### *3.1 Product Management*

The main task of product management in SPLE is product line scoping [9]. One facet of product line scoping is the definition of the product portfolio, that is, the set of applications offered for a certain market segment by a particular business unit or company. Further facets commonly include the definition of which set of features as well as which set of domain artifacts can be economically reused. If the scope of a software product line is too broad, domain artifacts may become too generic and the effort of realizing them may become too high. In this case, the product line may not be economically viable. On the other hand, if the scope is defined too narrow, required features as well as functional and quality requirements of many customers may not be covered and thus only very few applications might be derived from the product line. Again, the product line may not be economically viable. Therefore, product line scoping techniques need to include cost estimations and benefits as well as business and technical experts.

### *3.2 Domain Requirements Engineering*

Domain requirements engineering encompasses the typical requirements engineering activities [10], such as elicitation, negotiation, documentation, validation, and management, but in this case for the common and variable requirements for the product portfolio envisioned by product management. To identify all relevant common and variable requirements, product line requirements engineers have to involve a larger number of stakeholders than for single systems and have to consider additional requirements sources and constraints [1]. For example, a product line may address multiple customer groups and thus requirements engineers need to involve representatives of those groups, which requires support for the elicitation and documentation of common and variable requirements [11].

The amount of commonality and variability defined in domain requirements engineering has a huge impact on all other product line engineering activities, both in domain and application engineering. A high percentage of common features and common domain requirements in a product line typically require lower effort for designing and realizing the product line. Moreover, common requirements and domain artifacts are essential to engineering a product line platform that is stable yet flexible enough. On the other hand, the extent of variable requirements determines the potential number of different applications that can be derived from the product line and thus has significant impact on whether all goals and needs of the envisioned customers and/or market segments may be satisfied [1]. If a set of differing but related requirements is identified, two principal ways to treat those requirements exist. Those requirements may be defined as variable in the domain requirements.

Or those requirements may be harmonized or generalized and thereby defined as a common domain requirement. Determining how to treat those requirements is clearly a trade-off decision that has to be made in concert with product management and scoping.

### 3.3 Domain Design

Domain design encompasses all activities for defining the reference architecture of the product line. Numerous SPLE design methods have been advocated and targeted techniques for modeling variability in the architecture are available.

Traditionally, product line architecture approaches have been *component based*. In such a setting, variability is realized as component compositions and/or by introducing variation points into the components themselves. More recently, *aspect-oriented architectures* have been proposed to better address crosscutting features. Crosscutting features are encapsulated into modular units, the aspects, and composed by means of aspect-oriented mechanisms such as advices, join-points, and point-cuts [12]. Most recently*, service-oriented architectures* have been considered as part of SPLE [13]. In contrast to a component, which represents a comprehensive piece of software that is part of the software product line, a service represents functionality with associated quality characteristics (typically defined in a service-level agreement) offered by a service provider via a service interface [14]. The service itself or the service provider can change as long as the functionality and the service-level agreement remain the same.

### 3.4 Domain Realization

Domain realization deals with the detailed design and the implementation of the domain artifacts, for example, as reusable components or services. Variability can be realized using the capabilities of existing programming languages, compilers, and linkers [15]. Approaches include the use of inheritance (e.g., implementing alternative subclasses for an abstract superclass), aspect-oriented programming (e.g., the weaving of alternative code), conditional compilation (e.g., using preprocessor directives such as #ifdef), and binary replacement (e.g., providing the linker with alternative implementations of libraries).

To explicitly handle feature modularity and feature dependencies (or interactions) at the language level, new types of programming languages have been proposed that consider features and variability as first-class concepts. *Feature-oriented programming* supports the flexible and modular composition of systems from individual features. In FOP, “a feature module encapsulates changes that are made to a program in order to add a new capability or functionality” [16]. In *delta-oriented programming*, a compositional programming language, a product line is realized by a core module and a set of delta modules. The core module implements a valid application developed with single system development techniques. Delta modules specify changes to be applied to the core module to implement additional applications. Changes to the core model include the adding of additional code (as in FOP), but also removing and even the modification of code [17].

## *3.5 Domain Quality Assurance*

Quality assurance of domain artifacts is essential for successful product line engineering. A fault in a domain artifact may affect all applications of the product line in which this artifact is reused. Quality assurance techniques from single-system engineering cannot be directly applied to domain artifacts. As an example, a domain requirements specification can define a variable requirement $r$, that is related to variant $v_1$, and a variable requirement $\neg r$ related to variant $v_2$. Performing a consistency check of the domain requirements specification $R = \{r, \neg r\}$ using quality assurance techniques from single system development would identify a contradiction between $r$ and $\neg r$. Yet, if the variants $v_1$ and $v_2$ are defined to be mutually exclusive, the contradicting requirements can never be implemented together in the same application. Thus, the two requirements will never cause an inconsistency. A central challenge for quality assurance techniques in domain engineering is thus the consideration of product line variability [18].

**Quality Assurance of Domain Artifacts** Quality assurance of domain artifacts calls for quality assurance techniques that work in the presence of variability, including formal verification and testing. For the *formal verification* of product line artifacts, prominent verification techniques from single systems engineering have been adapted to the software product line setting, including type checking, model checking, and theorem proving. To handle variability during verification, various strategies have been followed, such as checking representative applications, checking features in isolation, or aiming to check all potential applications of the product line [18].

As in the development of single systems, *testing* in SPLE aims to execute the software to uncover the evidence of defects. One class of domain testing techniques includes techniques for developing reusable test cases in domain engineering and reusing and executing these test cases in application engineering [19]. In addition, domain testing aims to uncover evidence of defects in domain artifacts before these artifacts are reused in application engineering. Due to the variability defined in the domain artifacts, testing all potential product line applications (i.e., all potential combination of the common and variable artifacts) during domain engineering is impossible [20]. Typical domain testing strategies thus reduce the number of artifact combinations by using pairwise or $t$-wise testing strategies, or by focusing on important features and feature combinations.

**Variability Analysis** The consistency of the variability model is often a prerequisite for the analysis of domain artifacts. Variability analysis techniques help to ensure this consistency. Variability analysis aims to check and ensure whether certain properties for a given variability model hold. Examples for properties checked are satisfiability (i.e., whether at least one application can be derived from the variability model), membership (i.e., whether a given configuration is consistent with the variability model and thus represents a valid application of the product line), commonality (i.e., the set of "features" that appear in all applications), and "dead" features (i.e., features that cannot be selected for any application).

Manual analysis of variability models is error prone and infeasible when facing large-scale variability models. A broad spectrum of automated variability analysis techniques has been proposed which can be categorized in three main classes [21]: propositional-logics-based (using SAT or BDD solvers), constraint-programming-based (using CSP solvers), and description-logics-based (using DL reasoners). In general, variability model analyses exhibit an exponential worst-case execution time. Yet, research results indicate that in most cases variability model analysis can be mastered quite successfully using powerful solvers [22].

## 4 Application Engineering

### *4.1 Application Requirements Engineering*

During application requirements engineering, the requirements for a specific application are defined. In general, the application-specific requirements should be satisfied by exploiting the variability and using the commonality defined for the software product line. The application-specific binding of the variability is defined in the application variability model [1, 7].

In SPLE research, many publications convey the impression that an application can be derived from the domain artifacts by binding the defined variability of the product line and thus application derivation is seen purely as a feature selection process. For example, decision models define the decisions to be taken to derive an application of the product line [23]. To guide users to make those decisions, specific tools have been suggested. In the extreme, fully automated approaches have been devised that aim at optimal feature selection, for example, using search-based techniques.

In practice, customer- or market-specific applications often cannot be fully realized by reusing domain artifacts alone [7]. Often, there are application-specific requirements that cannot be satisfied by reusing domain requirements and thus require application-specific extensions to satisfy them. In order to handle such application-specific deviations from product line requirements, these application-specific extensions should be modeled as application-specific variation and documented, in addition to the variability bindings, in the application variability model [7].

### *4.2 Application Design*

Based on the application requirements, the application-specific architecture is derived from the domain architecture. The application architecture is typically a specialization of the reference architecture of the product line [1].

During application design, the design alternatives documented as variability in the domain architecture are assessed. The alternatives that fit the application requirements best are selected. Yet, in the case of application-specific deviations (see above), additional design decisions may have to be taken in order to derive an architecture that satisfies the application-specific requirements. Or even the architecture might have to be extended or adjusted, or the evolution of the product line architecture might be triggered.

### *4.3 Application Realization*

During application realization, code artifacts developed during domain engineering are derived and adjusted based on the application architecture and the application-specific requirements. For example, by parameterizing code modules using software configuration techniques, code modules can be adapted to fit a particular application. Application realization techniques facilitate such adaptations. An alternative approach to software configuration is code generation. Code generation techniques for product line applications have mainly adapted techniques from model-driven development and domain-specific languages.

Generative software product lines, a subclass of software product lines, support the derivation of individual applications without programming glue code or modifying the domain components. Yet, such an ideal approach is often not possible in practice (see above). In other words, application-specific coding and adjustments are usually required.

### *4.4 Application Quality Assurance*

Due to the variability of the reusable artifacts, it is impossible—except for trivial product lines—to comprehensively test all potential product line applications during domain engineering. Moreover, if based on concrete application requirements, specific variants are developed or application-specific extensions are made (e.g., see the discussion in Sect. 4.1), such variants and extensions can only be tested during application engineering.

Application testing techniques support the derivation of application-specific test cases from reusable domain test artifacts [19, 24]. Some application testing techniques aim to minimize the retesting of application parts already been tested for another application of the product line, thereby representing a special case of regression testing [25].

## 5 Modeling Product Line Variability

As explained in Sect. 2.3, product line variability differs significantly from software variability. Product line variability needs to be explicitly defined to empower and support the communication, discussion, management, and analysis of product line variability. Here, we introduce key constructs and two different approaches for modeling product line variability.

### *5.1 Key Modeling Constructs*

There are few, simple modeling constructs required for modeling product line variability:

- A *variation point* documents a variable item and thus defines "what can vary" (without saying how it can vary). As an example, the color of a car may vary.
- A *variant* documents a concrete variation and is related to a variation point. A variant thus defines "how something can vary." As an example, colors for a car may include black, red, and white.
- A *variability constraint* defines restrictions about the variability, for example, to define permissible combinations of variants in an application or to define that the selection of one variant requires or excludes the selection of another variant. As an example, only one single color may be chosen for any concrete car.

### *5.2 Integrated Versus Orthogonal Modeling of Variability*

There are two principal ways in SPLE research and practice to explicitly document product line variability: integrated and orthogonal documentation.

**Integrated Variability Modeling** To support the integrated documentation of product line variability, dedicated or specialized modeling and documentation concepts are introduced into existing modeling languages or document templates. An example for the integrated documentation of product line variability is depicted in Fig. 2a. The figure shows a UML class diagram extended by two stereotypes, "VariationPoint" and "Variant." The stereotypes are used to explicitly document the product line variability. This example models a product line, in which communication is defined as product line variability (documented by *Communication* being a variation point and *WiFi* and *MobileBroadband* being variants).

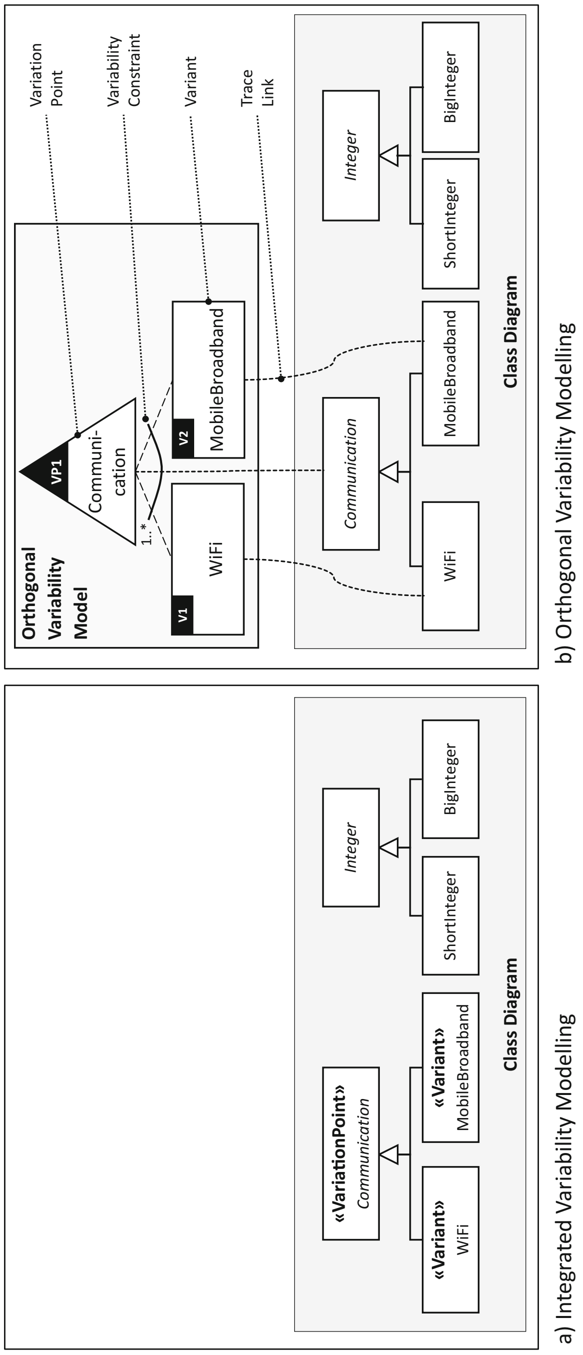

**Fig. 2** Illustration of integrated vs. orthogonal variability modeling

Feature models are a commonly used form of integrated variability modeling (e.g., see [15]). A feature model is a tree or a directed acyclic graph of features. A feature model is organized hierarchically. A feature can be decomposed into sub-features. A mandatory feature has to be selected if its parent feature is mandatory or if its parent feature is optional and has been selected. Mandatory features define commonalities. Mandatory features have to be selected for all applications of the product line. Optional, alternative, and "or" features define variability in feature models. As a result, a feature model is a compact representation of all mandatory and optional features of a software product line. Each valid combination of features represents a potential product line application.

**Orthogonal Variability Modeling** To support the orthogonal documentation of product line variability, product line variability is documented in a dedicated model. In other words, the documentation of product line variability is separated from the documentation of the software development artifacts. Thereby, the variability of the product line is treated as a first-class product line artifact. By relating the product line variability defined in the orthogonal variability model with the software artifacts defined in the artifact models, the realization of product line variability within the software artifacts is documented. Figure 2b sketches an example of an orthogonal documentation of product line variability and its relation to software development artifacts. As depicted in the figure, the documentation of product line variability is clearly separated from the documentation of other software development aspects. Note that the orthogonal variability model only defines product line variability. It does not define product line commonalities.

**Integrated Versus Orthogonal Variability Modeling** Integrated variability modeling increases the complexity of the software artifact models and documentations due to the additional documentation of product line variability within those artifacts. Moreover, product line variability is redundantly defined in different development artifacts such as requirements models, component diagrams, code, or test cases. As a result, understanding and tracing product line variability between different artifact models becomes difficult. First, different modeling constructs are used to represent the variability in the different models. As a consequence, product line variability is represented differently in the various models. Second, dependencies between the variability defined in the different artifact models are typically not documented explicitly. Third, it is difficult, if not impossible, to keep the variability defined in the different models consistent.

Orthogonal variability modeling avoids those three significant drawbacks of integrated variability modeling. In an orthogonal variability model, *only* the variability of a product line is defined. Commonalities of the product line are only documented in the base models—a key difference from "traditional" feature models, which define both commonalities and variability. The explicit differentiation between variation point and variant marks a second key difference from feature models, which do not provide explicit modeling concepts for variation points. As a third key difference, the variability definition in an orthogonal variability model is free from realization concerns. Therefore, orthogonal variability models provide a clear separation between product line variability (documented in an orthogonal variability model) and software variability (specified in the base models). When using feature models, the separation between product line variability and software variability often gets blurred [5]. Defining the variability in a dedicated, orthogonal variability model avoids this problem.

Product line variability defined in variability models must be interrelated with the software development artifacts defined in the base models. Establishing and maintaining trace links between variability models and the base models is not trivial. A solution for the interrelation is to parameterize the base models to indicate which base model elements link to which feature. However, this solution violates the key principle of keeping product line variability separate from base models. More recent solutions argue for dedicated mapping specifications, which define mappings from variation points and variants to MOF-compliant base models.

## 6 Variability Modeling in Non-Product-Line Settings

Explicit variability modeling and management can also support the development of software-intensive systems in non-SPLE settings. In this section, we briefly describe some examples.

**Clone-and-Own Development** There are cases in which a strategic and planned definition of a product line is not economically viable or not even possible. Beyond the investment in technical design and development of domain artifacts, the introduction of SPLE usually requires a change of processes and organization structure and thus requires significant investments. Therefore, and for many other reasons, instead of following an SPLE approach, software systems are quite often created by "cloning" existing ones (e.g., by copying and modifying requirements, architecture, and code of preceding systems). We strongly believe that the use of the "copy-and-modify" (aka "clone-and-own") approach will increase. Reasons for this increase are, among others, the need to adapt the applications to new technology and service offerings at run time and the rapid changes of the system context and the system requirements. Increasing change demands make a prediction of the scope of a potential product line much harder if not impossible.

Still, even if the SPLE approach is not followed to its full extent in these settings, principles and techniques from variability management facilitate addressing key challenges faced.

As an example, software configuration management tools may be extended with explicit variability management facilities (e.g., see [26]). Thereby, variability is identified (e.g., by deriving variability information based on "copy-and-modify" activities executed by the engineers) and managed in a non-product-line setting.

As another example, the German BMBF projects SPES 2020 and SPES XT (http://spes2020.informatik.tu-muenchen.de/) incorporated variability management into the engineering process of embedded systems [27]. Here, variability modeling principles and techniques facilitate the management of variability of related, single applications.

**Cloud Computing** Cloud computing aims to provide seamless reconfiguration of the infrastructure in real time based on measuring infrastructure usage and system execution parameters in real time. When combined with the Internet of Things [28], system execution data is enriched with data about the system context obtained by thousands of sensors. Big data analytics facilitates turning all this data into potential actionable insights with very low latency.

Together, these emerging technologies empower software developers and operators (DevOps) to continuously adjust the system based on instantaneous feedback obtained from system execution and the system context [29]. As a consequence, the tension between upfront investment and planning of a software product line and the increased agility fostered by instantaneous feedback and continuous deployment must be reconciled.

As an example, the EU FP7 project CloudWave (http://cloudwave-fp7.eu/) addressed this challenge by employing variability models to structure feedback about the dynamic reconfiguration of the cloud and in turn drive future reconfigurations [30]. An interesting opportunity is inferring the changes of product line variability and commonalities from analyzing operational and contextual data from cloud operations.

**Adaptive Systems** Driven by the Internet of Services, the Internet of Things, and the emergence of new highly distributed systems, such as cyber-physical systems and ultra-large-scale systems, the need for software to live in an *open* and highly dynamic world is becoming mandatory. Traditionally, software development rests on a closed world assumption. The closed world assumption roughly means that the boundary between the system and its context is known during system development and that the boundary does not change during system execution [31]. In contrast, open world systems cannot be specified completely during design time due to incomplete knowledge about, for instance, services and their actual quality provided during run time, sensors available during system operation to obtain environment information, the availability of other systems to interact and cooperate with, or the quality of data obtained. Such systems must frequently adapt to the dynamic changes faced during run time [14, 32].

As an example, variability models have been used to define the configuration space of a system (i.e., the set of all valid system configurations), thereby describing possible and permissible run-time adaptations of the system [33]. Variability models and mechanisms are well suited to deal with run-time adaptations by considering features as the unit of adaptation.

Oftentimes, foreseeing future context conditions and defining appropriate adaptation options during design time is not possible, and thus defining a variability model completely during design time is not feasible. A possible solution is to apply learning and reasoning techniques to variability models, thereby dealing with unknown situations [34]. For example, the DFG Priority Programme projects iObserve and iObserve 2 (https://www.iobserve-devops.net/) used such principles to update variability models to unknown situations during run time. In the iObserve approach, reinforcement learning is employed to improve the self-adaptive systems adaptation knowledge expressed in terms of variability models [35].

## 7 Summary

Software product line engineering has proven to facilitate the development of a diversity of similar software-intensive systems at lower cost, in shorter time, and with higher quality when compared with the development of single systems. We have described the main principles and techniques of software product line engineering. Moreover, we sketched how product line engineering principles can facilitate managing variability in non-product-line settings.

## References

1. Pohl, K., Böckle, G., van der Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer, Berlin (2005)
2. Metzger, A., Pohl, K.: Software product line engineering and variability management: achievements and challenges. In: International Conference on Software Engineering (ICSE) – Future of Software Engineering Track (FOSE 2014), Hyderabad, India (2014)
3. Clements, P., Northrop, L.: Software product lines: practices and patterns, reading. Addison-Wesley, Upper Saddle River, NJ (2001)
4. Coplien, J., Hoffmann, D., Weiss, D.: Commonality and variability in software engineering. IEEE Soft. **15**(6), 37–45 (1998)
5. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.-Y., Saval, G.: Disambiguating the documentation of variability in software product lines: a separation of concerns, formalization and automated analysis. In: 15th Int'l Requirements Engineering Conference (RE 2007), New Delhi, India (2007)
6. van der Linden, F., Schmid, K., Rommes, E.: Software product lines in action. Springer, Berlin (2007)
7. Halmans, G., Pohl, K., Sikora, E.: Documenting application-specific adaptations in software product line engineering. In: 20th Int'l Conference on Advanced Information Systems Engineering (CAiSE 2008), Montpellier, France (2008)
8. Galster, M., Weyns, D., Tofan, D., Michalek, B. and Avgeriou, P.: Variability in software systems: a systematic literature review. In: IEEE Transactions on Software Engineering. available online (2013)
9. Helferich, A., Schmid, K., Herzwurm, G.: Product management for software product lines: an unsolved problem? Commun. ACM. **49**(12), 66–67 (2006)
10. van Ommering, R., Bosch, J.: Widening the scope of software product lines: from variation to composition. In: 2nd Int'l Software Product Line Conference (SPLC), San Diego, USA (2002)
11. Bühne, S., Lauenroth, K., Pohl, K., Weber, M.: Modelling features for multi-criteria product-lines in the automotive industry. In: ICSE Workshop on Software Engineering for Automotive Systems (SEAS 2004), Edinburgh, UK (2004)

12. Pohl, K.: Requirements engineering: fundamentals, principles, and techniques. Springer, Heidelberg (2010)
13. Niu, N., Easterbrook, S.: Extracting and modeling product line functional requirements. In: 16th Int'l Requirements Engineering Conference (RE 2008), Barcelona, Spain (2008)
14. Figueiredo, E., Cacho, N., Sant'Anna, C., et al.: Evolving software product lines with aspects: an empirical study on design stability. In 30th Int'l Conference on Software Engineering (ICSE 2008), Leipzig, Germany (2008)
15. Mohabbati, B., Asadi, M., Gasevic, D., Hatala, M., Müller, H.: Combining service-orientation and software product line engineering: a systematic mapping study. Inf. Soft. Technol. **55**(11), 1845–1859 (2013)
16. Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M.P., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Autom. Softw. Eng. **15**(3–4), 313–341 (2008)
17. Capilla, R., Bosch, J., Kang, K.-C.: Systems and software variability management. Springer, Heidelberg (2013)
18. Batory, D., Höfner, P., Kim, J.: Feature interactions, products, and composition. In: 10th Int'l Conference on Generative Programming and Component Engineering (GPCE 2011), Portland, USA (2011)
19. Haber, A., Hölldobler, K., Kolassa, C., Look, M., Rumpe, B., Müller, K., Schaefer, I.: Engineering delta modeling languages. In 17th Int'l Software Product Line Conference (SPLC 2013), Tokyo, Japan (2013)
20. Lauenroth, K., Metzger, A., Pohl, K.: Quality assurance in the presence of variability. In: Intentional perspectives on information systems engineering, pp. 319–334. Springer, Heidelberg (2010)
21. Lee, J., Kang, S., Lee, D.: A survey on software product line testing. In 16th Int'l Software Product Line Conference (SPLC 2012), Salvador, Brazil (2012)
22. Pohl, K., Metzger, A.: Software product line testing. Commun. ACM. **49**(12), 78–81 (2006)
23. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: a literature review. Inform. Sys. **35**(6), 615–636 (2010)
24. Pohl, R., Stricker, V., Pohl, K.: Measuring the structural complexity of feature models. In 28th Int'l Conference on Automated Software Engineering (ASE 2013), Palo Alto, USA (2013)
25. Dhungana, D., Grünbacher, P., Rabiser, R.: The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. Autom. Softw. Eng. **18**(1), 77–114 (2011)
26. Engström, E., Runeson, P.: Software product line testing: a systematic mapping study. Inf. Softw. Technol. **53**(1), 2–13 (2011)
27. Stricker, V., Metzger, A., Pohl, K.: Avoiding redundant testing in application engineering. In: 14th Int'l Software Product Line Conference (SPLC 2010), Jeju Island, South Korea (2010).
28. Berger, T., Rublack, R., Nair, D., Atlee, J., Becker, M., Czarnecki, K., Wasowski, A.: A survey of variability modeling in industrial practice. In 7th Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2013), Pisa, Italy (2013)
29. Rubin, J., Kirshin, A., Botterweck, G., Chechik, M.: Managing forked product variants. In: 16th Int'l Software Product Line Conference (SPLC 2012), Salvador, Brazil (2012)
30. Pohl, K., Broy, M., Daembkes, H., Hönninger, H.: Advanced model-based engineering of embedded systems. Springer, Cham (2016)
31. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010)
32. Bosch, J.: Building products as innovation experiment systems. In: 3rd Int'l Conference on Software Business (ICSOB 2012), Cambridge, USA (2012)
33. Cooper, K., Franch, X.: Editorial. J. Syst. Softw. **81**(6), 841–842 (2008)

34. Díaz, J., Pérez, J., Alarcón, P.P., Garbajosa, J.: Agile product line engineering: a systematic literature review. Softw. Pract. Exp. **41**(8), 921–941 (2011)
35. Metzger, A., Bayer, A., Doyle, D., Molzam Sharifloo, A., Pohl, K., Wessling, F.: Coordinated run-time adaptation of variability-intensive systems: an application in cloud computing. In ICSE 2016 1st Int'l Workshop on Variability and Complexity in Software Design (VACE), Austin, Texas (2016)