Flexible Product Line Engineering with a Virtual Platform

Michał Antkiewicz, Wenbin Ji,
Thorsten Berger, Krzysztof Czarnecki
University of Waterloo, Canada

Thomas Schmorleiz, Ralf Lämmel
Universität Koblenz-Landau, Germany

Ştefan Stănciulescu, Andrzej Wąsowski
IT University of Copenhagen, Denmark

Ina Schaefer
Technische Universität Braunschweig, Germany

ABSTRACT

Cloning is widely used for creating new product variants. While it has low adoption costs, it often leads to maintenance problems. Long term reliance on cloning is discouraged in favor of systematic reuse offered by product line engineering (PLE) with a central platform integrating all reusable assets. Unfortunately, adopting an integrated platform requires a risky and costly migration. However, industrial experience shows that some benefits of an integrated platform can be achieved by properly managing a set of cloned variants.

In this paper, we propose an incremental and minimally invasive PLE adoption strategy called *virtual platform*. Virtual platform covers a spectrum of strategies between *ad-hoc clone and own* and PLE with a *fully-integrated platform* divided into six governance levels. Transitioning to a governance level requires some effort and it provides some incremental benefits. We discuss tradeoffs among the levels and illustrate the strategy on an example implementation.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Management

Keywords

product line engineering, clone management, virtual platform

1. INTRODUCTION

Development of multiple variants of products is often needed in order to satisfy conflicting requirements, legal frameworks, or to adapt the products to different geographical regions and usage conditions. In many cases, such product families are created using *clone-and-own*—a new variant is created by copying and customizing assets from an existing variant.

*Supported by ARTEMIS JU grant n° 295397 VARIES

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

Despite having low adoption costs and allowing independence from other developers, cloning easily leads to inconsistencies, redundancies, and lack of control. In the literature, using cloning in the longer term has been considered a harmful practice [8]. It has been traditionally recommended that organizations adopt a more systematic, strategic reuse offered by *product line engineering* (PLE) [10] based on a central platform. Such a platform should integrate the reusable assets and it should be used for deriving new variants of products. Existing incremental PLE adoption strategies [4, 6] discourage relying on cloning due to maintainability issues. However, as shown by industrial practice, eliminating cloning and adopting the integrated platform is not always desirable nor beneficial as it requires high-risk migration processes [2].

In this paper, we present an *incremental* and *minimally invasive* strategy for adoption of product-line engineering called *virtual platform*. Virtual platform allows organizations to achieve many benefits traditionally associated with having a fully-integrated platform but without requiring the high-risk transition processes, while retaining the flexibility and benefits of cloning. Most importantly, it allows organizations to obtain *incremental benefits proportional to incremental efforts* suitable to the frequency of reuse and the required degree of consistency among the variants.

2. VIRTUAL PLATFORM

To describe the spectrum of strategies employed within the virtual platform, we use the following conceptual framework. An *organization* runs many *projects* concurrently. Each project has a *team* and *assets*. The team uses the assets to derive one or more *variants* of products. An *integrated platform* is a special kind of project intended to keep reusable assets that can be used without modification by teams in other projects. The variants can be further characterized by *features*. The customer requests a variant based on the desirable features. The features can be mapped to *fragments* of assets used to specify and implement them.

The main idea of the virtual platform strategy is to apply a clone management approach to make distributed assets reusable instead of physically containing all reusable assets in an integrated platform as typical for PLE. Transitioning from clone-and-own to a fully-integrated platform as advocated in literature is difficult, as it requires transforming assets not intended for reuse into a set of fully reusable assets that features map to. Furthermore, the transition requires introducing new processes, training, and switching development focus from a single variant to the entire family of variants. Such a transition disrupts the organization's

ability to operate and continue development [14].

In reality, there are many practical intermediate points between the clone-and-own and the fully integrated platform. Whether the effort spent by an organization on preparation for reuse (either via clone management or PLE) is *justifiable* depends on the required *frequency of reuse* and the required *degree of consistency* among the reused assets. In the following, we present the governance levels of the virtual platform, discuss their tradeoffs, and illustrate them, where applicable, on the 101companies effort [3].

2.1 L0: Ad-hoc Clone-and-Own

Teams freely copy assets across projects and modify them as needed, without any reuse strategy or process. No preparation for reuse is needed. Entire projects, assets, or fragments of assets are copied. No notion of features is used and therefore no mapping of features to assets exists. A single project containing all assets is used to derive one variant.

Advantages. Cloning is associated with many benefits [2, 8]. It is easy and fast for teams knowledgeable about the project, since no special development tools or processes are needed. Developers of a new variant are also independent from the developers of the original, and free to modify it as needed. Finally, as the original variant may have been tested and used, the new variant may be usable from the beginning.

Disadvantages. If not carefully managed, cloning has serious drawbacks [2, 8]. It does not scale: with an increasing number of variants, the overhead for synchronizing assets may exceed the benefits of the initial reuse. Cloning also requires governance and discipline among developers. Without specified cloning practices and recorded provenance information, the assets used to create the original and the clone easily become disconnected and inconsistent. This can result in redundant work and can hinder long-term evolution.

Tactics. Traditional small-scale reuse tactics such as component libraries and frameworks can be used to make the assets more reusable. Also, cloning can be better managed by using branching and merging capabilities of a (distributed) version control system, which automatically records some information needed for locating features.

Example. The goal of 101companies [3] is to aggregate a set of contributions from different authors who implement the same set of features of a fictitious human resources management system, while illustrating different implementation languages and techniques. The practice within 101companies can be characterized as ad-hoc clone-and-own, except that the system is described by a feature model. Each contribution is also characterized by a set of features, but without any mapping to the assets. No libraries or frameworks are used to make the project assets reusable.

Recommendation. Ad-hoc clone-and-own is appropriate when the frequency of reuse is very low and maintaining the consistency among the projects is not important.

2.2 L1: Clone-and-Own with Provenance

Teams record provenance information about the original projects and per cloned asset. Teams use the provenance information for impact analysis and change propagation.

Advantages. Provenance information enables propagation of extensions and bug fixes among the cloned assets. During development of an original asset, teams can send notifications about changes to teams working on the clones. Conversely, teams working on a cloned asset can decide whether change

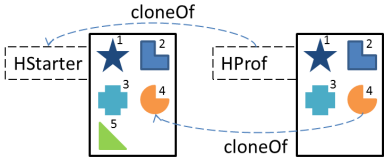

Figure 1: Recording provenance information

progagation is needed upon receiving a notification.

Disadvantages. Since the provenance information is coarse-grained (entire asset), teams need to manually locate the relevant fragments of assets and propagate the change.

Tactics. Develop small, cohesive assets. Use facilities of a version control system, such as branching, to isolate modifications into coherent groups across assets, and exchange changesets. Incorporate metadata using explicit feature annotations in code or in commit messages. Use clone detection tools to recover provenance information.

Example. In 101companies, we extended the metadata of a contribution to contain information about the original contribution it was cloned from. Fig. 1 illustrates two contributions: `HStarter` and `HProf`. Solid line boxes represent the projects. The shapes numbered 1–5 represent fragments of assets contained within the projects. Dashed lines represent the provenance links (`cloneOf`) for project `HProf = cloneOf(HStarter)` and for asset 4 `HProf::4 = cloneOf(HStarter::4)`. We can also see that asset 5 was not cloned. We analyzed the version control history to detect instances of cloning and recover provenance information. We detected that in a commit to `HProf`, a new asset 4 was added which was a clone of an existing asset 4 in project `HStarter`, that is, asset 4 was cloned from `HStarter` to `HProf`.

Recommendation. Clone-and-own with provenance is appropriate when the frequency of reuse is low and maintaining the consistency among the assets is moderately important.

2.3 L2: Clone-and-Own with Features

Features succinctly characterize the functionality of a variant from the customer's point of view. Teams declare features and map them to asset fragments that implement them. Features can be *modular* (implemented in a single asset) or *cross-cutting* (distributed across assets), or *tangled* (a single asset can contain overlapping fragments corresponding to many features). Teams propagate features among projects by cloning the corresponding asset fragments and recording provenance information. Teams leverage notifications about feature-related changes and perform change propagation.

Advantages. Features provide a functional decomposition, and allow reasoning about the co-evolution of projects and their assets in terms of features instead of physical assets. Teams benefit from a better overview of the projects in terms of user-relevant functions. Teams can make better reuse decisions and more easily propagate features across projects, as the relevant fragments of assets can be located easily.

Disadvantages. Features can have complex dependencies and interactions, challenging their reuse. Thus, teams need to rely on intricate domain and implementation knowledge.

Tactics. Use a framework for managing cloned product variants, such as Rubin et al. [13, 11, 12], which treats features as the prime reuse units. It relies on metadata about the features of variants, their location in code, their dependencies, and their origins if cloned from other projects. Thus, relevant fragments can be located. Operators and metadata as speci-

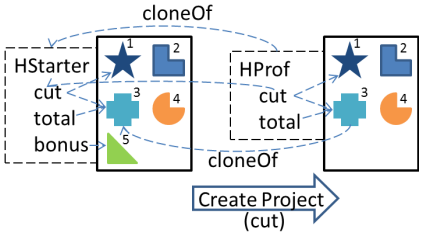

Figure 2: The scenario Create Project

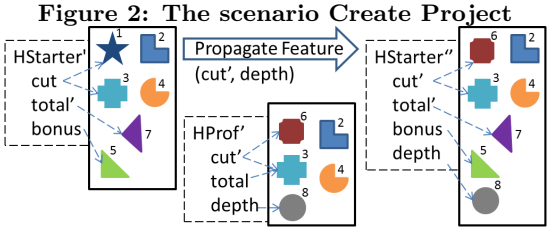

Figure 3: The scenario Propagate Feature

fied by Rubin et al. support the following scenarios in this level: refactoring to introduce features, variability and commonality analysis, propagating and sharing features among variants, retiring features, and establishing new variants [12].

Further, use white-box reuse approaches, such as those by Holmes and Walker [5]. Such approaches identify other parts of the implementation that features depend on using various techniques, including static code analysis.

Example. We applied a variation of Rubin et al.'s clone management framework to 101companies, and implemented tool support for feature location and two scenarios as follows.

The scenario *Create Project* is shown in Fig. 2 (block arrow). We represent the metadata using dashed lines: a box for the features of a project and arrows for the mapping of the features to asset fragments. For example, the project `HStarter` implements three features, `cut`, `total`, and `bonus`, which are mapped to fragments 1, 3, and 5. The other fragments, 2 and 4, are integral to the project. The mapping between features and asset fragments is computed automatically by a simple feature location algorithm. We use the links `cloneOf` to record the provenance of features.

Developers create a new project from an existing project by deselecting undesired features. In Fig. 2, a developer selected the project `HStarter` and deselected the features `total` and `bonus`. Since `cut` requires the feature `total`, both must be cloned. The developer deselected the feature `bonus` and therefore the corresponding asset was not cloned. In our approach, instead of physically removing fragments of the features that are not cloned, we comment these fragments out to compensate for the imprecision of the feature location. Also, features which cannot be located are always cloned since they cannot be commented out. Asset fragments 2 and 4 were also cloned as they are integral parts of the project. Provenance information (`cloneOf`) was also recorded.

Thereafter, developers manually inspect the assets, build the project, and uncomment the code that is still needed. For instance, in one case, parts of the implementation of one unselected feature were used in implementation of a cloned feature—these needed to be uncommented for the project to build. Finally, the developer confirmed the successful creation of a new project, while our tool recorded the set of features and their provenance information.

The scenario *Propagate Feature* is shown in Fig. 3 (block arrow). Developers first identify dependencies of a feature they want to propagate. Next, they retrieve fragments of assets related to the given feature and its dependencies. Finally, they clone the needed fragments to their project, using and recording provenance information. In Fig. 3, the team of `HStarter'` propagated a new feature `depth` and the extended feature `cut'` from `HProf'`. In the resulting project `HStarter"`, fragment 1 was replaced by the new fragment `HProf'::6`, and the fragment `HProf'::8` was added.

Recommendation. Clone-and-own with features is appropriate when the entire features are reused and the frequency of reuse is medium as well as reasoning about the features and maintaining the consistency among them is important.

2.4 L3: Clone-and-Own with Configuration

For individual projects, teams add the capability to disable features and to derive variants by selecting subsets of features. They add feature constraints to exclude invalid combinations.

Advantages. The ability to derive multiple variants from a single project reduces cloning and increases reuse potential.

Disadvantages. Focus of a developer is shifted from a single variant to a set of variants, which complicates development.

Tactics. Use a feature model [7] per project, to define features and constraints; use a configurator. Use traditional variability mechanisms, such as configuration parameters, preprocessors, generators, or component frameworks.

Recommendation. Clone-and-own with configuration is appropriate when frequent derivations of similar variants containing subsets of features are needed and when maintaining the consistency among the projects is important.

2.5 L4: Clone-and-Own with a Feature Model

An organization creates a central feature model that covers all projects and all implemented features. Teams create new projects by taking an existing project as a basis and then propagating the needed features from all other projects as allowed by feature model constraints. Teams extend the central feature model.

Advantages. The assets distributed across the projects are reusable as if they were integrated into a platform. The central feature model constrains the valid combinations of features that can be reused together.

Disadvantages. The assets are still distributed and their consistency still needs to be managed. Multiple versions of the same feature exist. Manual integration of the cloned assets is needed as they are not prepared to work with each other as there is no product line architecture.

Tactics. Merge feature models of projects into the central feature model.

Example. We envision the scenario *Create Project* as follows. Fig. 4 shows a feature model containing features from a number of contributions. The team creates a new project `HSimple` by selecting three features. The tool creates the project by cloning the fragments implementing these features.

Recommendation. Clone-and-own with a feature model is appropriate when the frequency of reuse is high, a global overview of all features and constraints among them is needed, features need to be reused from many projects, and maintaining the consistency among the cloned features is important.

2.6 L5: PLE with an Integrated Platform and Clone-and-Own

An organization creates a platform project and a platform

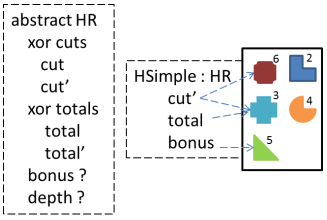

Figure 4: The scenario Create Project, cf. Fig. 3

team. This team adds configurability to the platform so that project teams can derive variants by configuring the platform. However, this team also merges existing projects into the platform and harvests features from existing projects.
Advantages. Increased scale, improved change propagation, reduced redundancy, and configuration over implementation. Although an integrated platform is created, other projects can still be kept and developers can still work on projects as cloning is still allowed. Project teams are not restricted by the platform, which supports innovation.
Disadvantages. Same as level 4, although reduced in severity as the amount of cloning is reduced and consistency is managed through the platform. Projects receive updates when adopting a new version of the platform.
Tactics. Use a clone management framework, such as Rubin et al. [13, 11, 12], to perform development of common architecture and common assets, merging initial set of cloned variants, bringing additional variants into the platform.

Use traditional annotative and compositional PLE techniques [9], which rely on a configuration mechanism (e.g., build system and preprocessor) or a suitable software architecture leveraging programming-language-level mechanisms.
Recommendation. PLE with an integrated platform and cloning is appropriate for frequent reuse, a global overview of all features and constraints among them is needed, and maintaining consistency among projects is important.

2.7 L6: PLE with a Fully-Integrated Platform

Teams only use shared assets contained in the *integrated platform* to derive variants. The platform is completely specified by a feature model: given a set of desired features, a new variant can be completely—often automatically—derived from the platform. No development happens within projects.
Advantages. Makes the sharing of assets explicit, which allows developers to reduce redundant implementation and to propagate new features, extensions, and bug fixes. The full platform integration minimizes custom code for new variants.
Disadvantages. Poses high risks. PLE adoption requires disruptive organizational changes, including a new platform team and new processes for product teams [4]. Relying on the platform for new products also hinders innovation, because the platform restricts developers' freedom, while cloning may still not be entirely prevented [2]. Beginning with a fully integrated platform approach is often not practical, as organizations cannot anticipate all future variants and features. Cost can be also very large [10]. In fact, our survey shows that only a minority of industrial product lines was adopted pro-actively [1]. Most were evolved from one variant or re-engineered from a set of cloned variants.
Tactics. Use annotative and compositional PLE approaches.
Recommendation. A fully integrated platform is hard to achieve, since new features, extensions, and bug fixes are continuously and concurrently developed within projects. Development within a platform, without a project context, is possible but difficult, since projects provide motivation, requirements, and a testing environment. Only changes worth of propagation and sharing are harvested into the platform. Thus, a platform rarely covers 100% of variants.

3. CONCLUSION

We presented an incremental and minimally invasive strategy for adoption of PLE called *virtual platform*. It combines the flexibility of *clone-and-own* with the scalability and consistency of traditional *platform-based PLE* using common variability mechanisms. We presented six governance levels as a roadmap for seamless and gradual adoption of PLE, thus eliminating costly, disruptive, and high-risk transition processes. Adopting each level provides incremental benefit.
Acknowledgements. We thank Julia Rubin for discussions about the relationship between the virtual platform and her clone management framework.

4. REFERENCES

[1] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *VaMoS*, 2013.
[2] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *CSMR*, 2013.
[3] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. 101companies: a community project on software technologies and software languages. In *TOOLS*, 2012.
[4] W. A. Hetrick, C. W. Krueger, and J. G. Moore. Incremental return on incremental investment: Engenio's transition to software product line practice. In *OOPSLA*, 2006.
[5] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.*, 21(4):20:1–20:44, Feb. 2013.
[6] H. P. Jepsen, J. G. Dall, and D. Beuche. Minimally invasive migration to software product lines. In *SPLC*, 2007.
[7] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep., SEI, CMU, 1990.
[8] C. Kapser and M. Godfrey. "cloning considered harmful" considered harmful. In *WCRE*, 2006.
[9] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.
[10] L. N. P. Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
[11] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *ICSE*, 2013.
[12] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *SPLC*, 2013.
[13] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing forked product variants. In *SPLC*, 2012.
[14] F. Stallinger, R. Neumann, R. Schossleitner, and S. Kriener. Migrating towards evolving software product lines: Challenges of an SME in a core customer-driven industrial systems engineering context. In *PLEASE*, 2011.