59

プロダクトラインエンジニアリングの バリアビリティ（変動性）とバリアント管理の海外事例 [1]

Manage Variants and Variability: Benefiting from Product Line Engineering

・ソースを再利用するつもりだったのに、苦労を再体験してしまった。
・やっとバグをつぶしたと思ったら、コピペ先でバグが突然変異。
・何度も何度も同じ改修を繰り返している。悪い夢を見ているようだ。
・ブランチ × バージョン ＝ 爆発！

これはバージョン管理にバリエーション（製品間の違い）の管理を混同したことに起因する症状

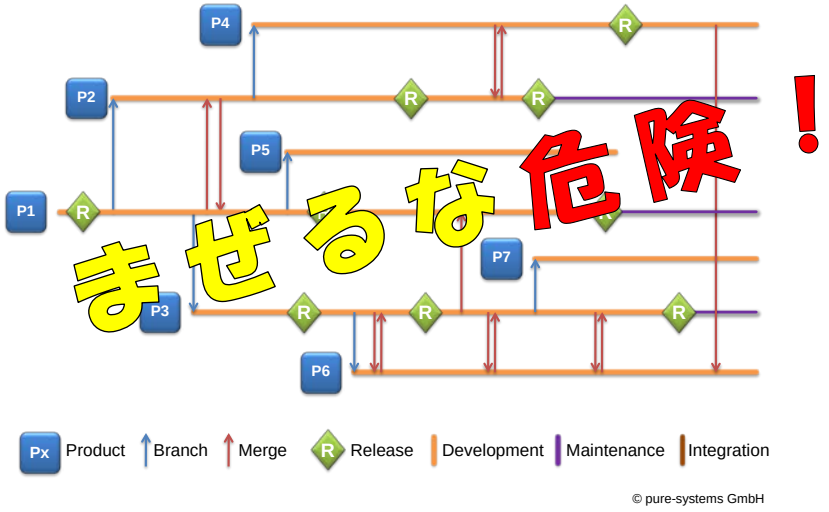


図 59-1　バージョン管理にバリエーションの管理を混同

です。多くの組織では、こうした経験をしたことがあるのではないでしょうか。

ソフトウェアプロダクトライン開発は、製品系列の資産を体系的に再利用することにより、派生製品の開発工数を削減して市場投入時期を早めています。同時に、品質改善や、製品間のトレーサビリティによるメンテナンス性向上などの相乗効果も期待されています。

[1] 事例提供: pure-systems GmbH Dr. Danilo Beuche
翻訳: 富士設備工業株式会社 電子機器事業部 小山 美樹 氏、浅野 義雄 氏

概要

多くの共通性を持つ、複数のソフトウェア集約型製品の開発や保守には、体系的な再利用を支える戦略が要求される。この課題には、プロダクトライン開発の考えに根差したバリアント管理が、実用的な解決策となる。

本編ではプロダクトラインエンジニアリングの原理が応用された、産業界の３つの異なる製品事例（工業オートメーションや車載システム）を基に、プロダクトラインエンジニアリングとバリアビリティ（変動性）に対するモデリングについての基本原理を紹介する。そして、この理解をベースにして３つの事例を紹介して、これらに共通する側面と異なる部分、そしてそれらの効果や得ることのできた教訓について論じる。

1. 導入の背景

似通った複数のソフトウェア集約型製品のコラボレーション開発は、多くの組織にとって課題になっている。なぜなら多くの場合、体系的な再利用戦略は、製品固有のバリエーションを共有資産から迅速に実装することの必要性とはそりが合わず、結果的に共通資産は異なるチームごとで勝手に変更され、メンテナンスや大規模な再利用は非常に困難になる。そしてコピーによる資産の再利用はメンテナンス費用の増加を招く。なぜなら、修正を全ての異なるコピーに反映させなければならないためである。

プロダクトラインエンジニアリングの考えを基にしたバリアント管理を通じた体系的な再利用は、これらの課題の解決を目指している。プロダクトラインエンジニアリングの鍵となる原理は、既存の複数製品と将来の製品を視野にした再利用の全体的な見通しである。バリエーション・バリアビリティは、常に製品を特長付ける側面として捉えることができる。「いつか役に立つかも」と機械的にバリアビリティを増やしていくやり方とは違って、「ちょうどぴったりの」バリアビリティを与えるのがこの原理の考え方である。ただ機械的にやっていたのでは、非常に柔軟かつ複雑すぎるが故に、再利用の困難な部品を生み出してしまうことになる。

市場志向の製品政策や体系的再利用にプロダクトラインエンジニアリングのコンセプトを用いることは、チャンスでもありチャレンジでもある。プロダクトラインエンジニアリングは、新たなユースケースに応じてインクリメントに改変される共有資産に強く依存する。プロダクトラインエンジニアリングでは、ビジネスの可能性と開発の複雑性の間（例えば、プロセス、利害関係者間の意思疎通、技術上の再利用アーキテクチャ、実装などで）や、システムの高い品質と生産性の向上の間で、適切な妥協が必要になる。

他の再利用のコンセプトから、プロダクトラインエンジニアリングを区別する中心的役割は、取組みになくてはならない部分として、バリアビリティと製品インスタンスであるバリアントを系統だって管理することである。これにより、汎用的な資産を修正なしにそのまま使うという通常であ

れば困難な再利用が可能になる。新製品の要件には、多くの場合、以前の製品に対して追加または一部変更がある。事前に、これら新しい要件を知ることや、それに気を配ることは殆ど不可能である。それゆえ全てを満たす単一の標準部品は多くの場合、実行可能な選択肢ではない。

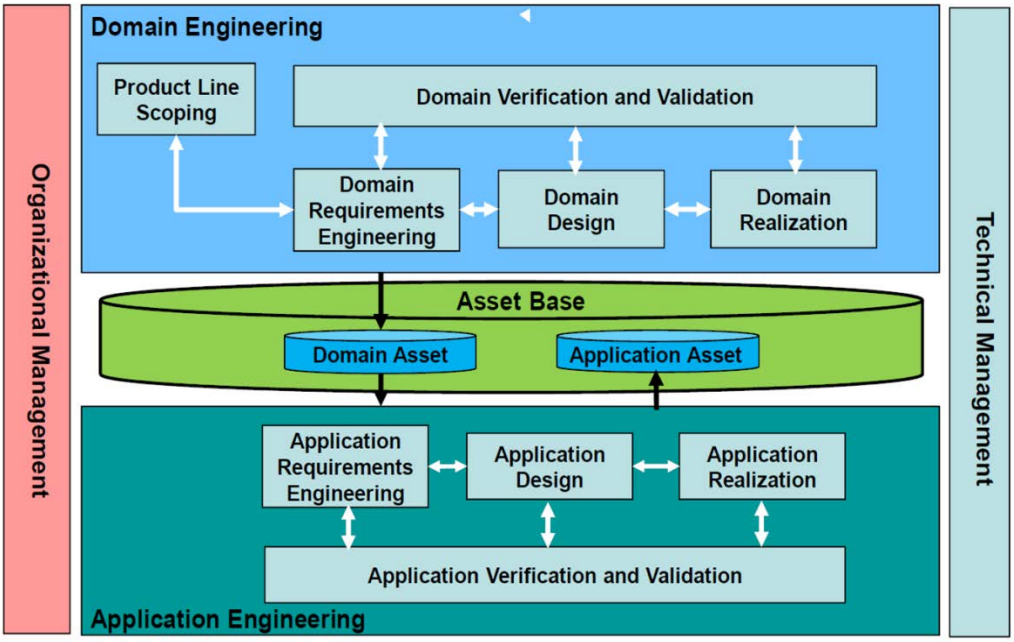


図 59-2　Product Line Engineering Overview (from ISO 26550:2015)

図 59-2 にプロダクトラインエンジニアリングの主な活動の概要を示す。一般にプロダクトラインエンジニアリングにはウォータフォールのようなプロセスは無い。一般には、最初にビジネスゴールとマーケットセグメントが分析されて、アーキテクチャを定義してからコンポーネントとサービスが開発されはするがしかし、現実的には多くの作業が、イテレーティブかつインクリメントに行われる。例えばマーケットの需要は継続的に変化するし、いくつかのアーキテクチャ上のコンセプトも新たな技術導入で変更される。

プロダクトラインエンジニアリングの大切な資質の一つとして挙げられるのが、多くのバリエーションは、あらゆる種類の資産を横断することと、継続的な進化の対象であることを前提にすることである。複数の製品インスタンス（バリアント）を横断するプロダクトライン内の進化を扱うには、エンジニアリングの資産の一部として、バリアビリティとそのコンフィグレーションの、体系的で効果的な表現が必要になる。この情報の表現には、実践活用できる多くの手法がある。その中で最も一般的なアプローチは、フィーチャモデルを資産固有のバリエーションポイントと組合せて使用することである。

2. フィーチャモデルとバリエーションポイントにおける変動性の説明

事例紹介の前に、フィーチャモデルを用いたプロダクトラインエンジニアリングの基本コンセプトと用語について解説する。フィーチャモデルについて理解するために、その小さな例を紹介する。以下の図 59-3 は自動車のある機能のフィーチャモデルである（バリアント管理ツール（pure::variants[2]）のフィーチャモデルの図)。

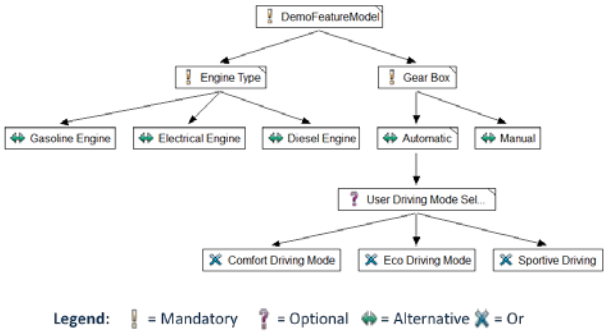


図 59-3　A small model of a Car

フィーチャモデルはプロダクトラインのプロパティを表現するフィーチャ（機能）群で構成される。システムのオプション機能（User Driving Mode Selection）や、そのオプションに対する複数の機能選択肢（Comfort/Eco/Sportive Driving Mode)、製品インスタンスのテクニカルでオールタネイティブな選択肢（Gasoline Engine, Electrical Engine, Diesel Engine からいずれかを選択）などだ。　そしてフィーチャによっては構成のみの情報を提供する（Engine Type, Gear Box のようなmandatory（必須）の機能）である。これらフィーチャは相互にグループ化され、またリンクされて、バリアビリティは識別しやすいツリー構造で表現される。

図 59-3 ツリー構造では、親から子へのラインで階層が表され、各フィーチャ名の前に付くアイコンはバリアビリティのタイプを示す（図 59-3 内の Legend: を参照)。Mandatory なフィーチャは、その親が選択されると必然的に含まれる。Alternative なフィーチャは、その中でどれか一つだけである。Or なフィーチャは一つ以上いくつでも選択可能であり、Option の各フィーチャは単独に選択可能だ。フィーチャモデリングは、フィーチャ間のより複雑な関係性の表現に対する、追加のコンセプトを提供する。これについては、この資料内で後述する。

[2] http://www.fuji-setsu.co.jp/products/purevariants/index.html

2.1. なぜフィーチャモデルのバリエーションの資産管理が必要なのか

同じプロダクトラインからの異なる製品インスタンスを比較する場合、ソースコードの資産や、コンフィグレーションパラメータ、ドキュメント、モデル、一連の資料など、多くのエンジニアリング上の資産に、大小の違いが存在する。

詳細を調査することで、これら違いの一部はプロダクトラインの進化に起因することがわかる。複数の製品インスタンスは異なった時期に開発される（例えば、バージョン管理上の異なったベースライン）。現にベースライン間で、ドキュメントのアップデートや、欠陥の修正が行われる。この手の違いは、プロダクトラインエンジニアリングのコンセプトを導入していない開発においても起こり得る。

しかしながら一般に違いの多くは、再利用資産が、それぞれの製品インスタンスの固有の要件を満たすように、洗練されたポイントでドメインの資産を構成することに起因する。これらのポイントはコンフィグレーションパラメータとして識別できる箇所で、アーキテクチャ上のエレメントとして使用できる一連のコンポーネント群から、一つの実装部品を選択することができる。これら違いが生じるポイント（あるいは場所）をバリエーションポイントと呼ぶ。これは実質的に異なるバリエーションである。単一のバリエーションポイントは、一つ以上のバリエーションを組成する。

プロダクトライン内のバリエーションポイントは、バリエーションの実現に様々な異なるアプローチを取ることができる。それは、システムモデルやソフトウェアキテクチャモデルにアノテ―ションを用いて表現することや、プログラミング言語のプリプロセッサ命令を用いること。さらに、ファイルシステムの構成を使って表現することや、テキストドキュメントなどへマニュアルで修正することへの指示などである。

2.1.1. バリエーションポイントの依存性

通常、現実のプロダクトラインでは、資産となるバリエーションポイントの量は相当多くなる。単に二者択一な選択要件で、それがユーザ視点で単一のバリエーションポイントを表現するのに使用されるだけでも、デザイン、コード、テスト、ドキュメント等にバリエーションが作られることになる。これら異なる資産内のバリエーションポイントは、相互に関連することは明らかである。

これらの関連に加えて、追加の相関も存在する。例えば、特定のサービス品質への要求を満たす場合には、ある固有のコンポーネントのバリエーションが使用できないなどだ。バリエーションポイントと対応するバリエーションの量、加えてそれらの関連により、ソリューション（解決空間）の資産のバリアビリティが定義される。このバリアビリティの複雑性は（多くの製品インスタンスを構成できる）、多かれ少なかれ指数関数的に増加する（新規のオプショナルなエレメントごとに構成可能なインスタンスが倍増する）。そして、全ての構成可能なインスタンスのテストが不可能な状態に陥る。またコントロールすべき依存関係のために、バリエーションポイントの構成が非常

に複雑になる。しかしながら、バリエーションポイントのコンセプトは追加の複雑性を生むように見えるが、それは真実ではない。バリエーションポイントは、プロダクトライン内に既に存在するバリエーションを単一の形式で明らかにするだけである。もちろんプロダクトラインの資産のほうが再利用可能にするためのバリエーションがあるので、単一製品開発の資産と比較すれば、より複雑となる。

効率的にドメイン資産内のテクニカルなバリエーションポイントの再利用を管理すること、加えて通常大量に上るソリューション内のバリエーションポイントの複雑性が露呈しないように、単純化されたビューが求められる。そのようなビューは、固有資産のバリエーションポイントを How の観点から What の観点で抽象化することで得ることができる。このような観点のシフトによって導かれる問題空間志向のバリエーションポイントは、製品インスタンスの定義に使用する言語内で表現される。多くの場合、このビューはフィーチャモデルによって非常に効果的に提供される。

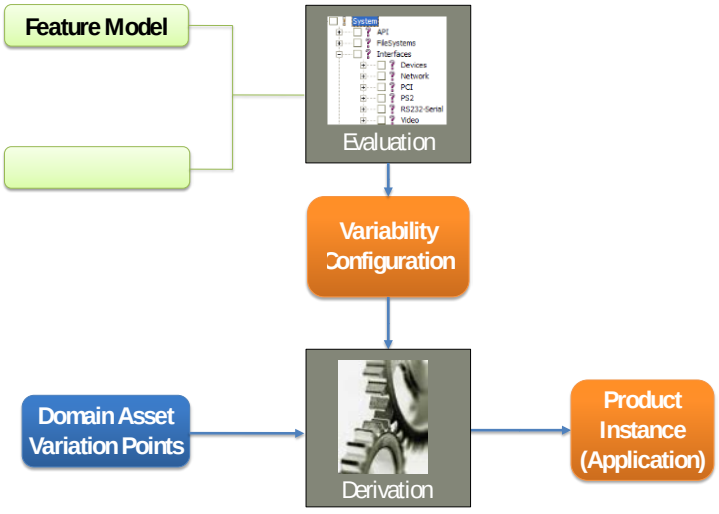


図 59-4　Basic Process of Feature-based Product Instantiation.

2.1.2. フィーチャモデルでの用例

十分に用意されたフィーチャモデルベースのプロダクトラインでは、フィーチャモデル内の各フィーチャの選択によって作られたフィーチャリストによって、製品インスタンスが定義される。その結果であるバリアビリティの構成は、資産内のバリエーションポイントへのマップ情報と結合する（フィーチャやフィーチャの組み合わせと、システムの各要件や設定部品との紐付け）。この段階でバリエーションポイントを構成（依存・排他関係等を加味して）することで（インクリメントに行われる）、最終的に製品インスタンスを生成することができる。このプロセスの概要を、以下図 59-4 に示す。

フィーチャモデルはバリアビリティを記述して管理するだけの手段では無く、コンフィグレーシ

ョンファイル、あるいはドメインスペシフィック言語と、組み合わせて使用される。

異なる観点から要約すると、フィーチャモデルとフィーチャの選択は、エンジニアリングライフサイクルにおいて、利用可能なバリアビリティ（フィーチャモデル）と、その活用（フィーチャの選択やバリアビリティの構成）といった新たな資産になる。そして様々な側面から、要求仕様や設計ドキュメント等と同様に管理されなければならない。変更や進化を構成管理することや、課題があれば変更管理にも紐付けなければならない。フィーチャモデルはプロダクトラインの仕様の一部であり、バリアビリティの構成は製品インスタンスの仕様の一部である。

3. 事例

実例は、課題やコンセプト、成果を説明する上で極めて重要である。包括的であるがゆえに、抽象的な概念の理解に役立つので。総合的な事例を介して、どのようにして各コンセプトが開発の課題の様々な側面に対応し、また相互に作用するかを理解することができる。

この資料では、工業用オートメーション、車載システムのアプリケーションドメインから、３つの実践的な事例を紹介する。これらの複雑な事例を通じて、コンセプトの拡張性や、現実の問題への適応力も理解することができる。これらは、今日の組込みシステムを象徴する以下の特徴を持つ。

- 高度に分散化された機能
- 機能安全性
- リアルタイム要求
- ソフトウェア集約型システム：ソフトウェアが機能実装の主要な要素
- 機械、メカトロニクス、電気、電子などの各要素の組み合わせ
- リアクティブ/コントロール 動作

各事例は複雑であり、全ての詳細説明は適さない。その代わりに、以下に各事例の概要とバリアント管理の応用についての主な側面を紹介する。

3.1. 事例 1: Danfoss’ Frequency Converter Product Line

Danfoss 社は周波数コンバータと関連製品の主要メーカの一つである。周波数コンバータは、高出力モータ用の電子制御装置だ。そのアプリケーションは非常に広範で、ファクトリーオートメーション、空調設備、送水ポンプ、クレーン（起重機）などであり、広範な電力範囲と電圧、また多様なアプリケーションに固有な構成のモータに活用される。Danfoss 社は 10 年以上前に、従来の取組みでは問題が増加の一方であったことから、プロダクトラインエンジニアリングの取組みへの移行を開始した。当初の開発は同じコードと共通の制御ハードウエアをベースにしたものの、異なる製品間でコードはバージョン管理システムの個別のブランチに分岐されてしまい、同じ課題の修正が何回も必要になるといった結果に陥っていた。[1]

3.1.1. プロダクトラインエンジニアリングへの移行

製品開発の中でもソフトウェア開発は、多くのバリエーションを抱えていたので、効果的で質の高い再利用への需要は、当初から明白であった。オリジナルのソフトウェアから分岐したブランチがかなり多く存在していて、かつ製品開発は中断させられない状態であった。そのため、異なる製品間でコードを、プロダクトラインの取組みで共有する再利用コード資産にどうやってマージするかということが課題となった。

全製品の（10 機種以上)、全ての既存コードをマージするのは現実的ではないので（全部で数百万行のコード、製品ごとで少なくとも数十万行)、移行を担当するチームは 60/40 のアプローチを取ることにした。最初に２つの主力製品の約 60%のコードを再利用資産にマージするために、小さなチームに２か月が与えられ、次にマージされたコード資産を全製品が以前と同様に動作するように、全製品のコード基盤に統合した。この 60%については、主にバリエーションが少ない箇所や、バリエーションが許容できそうな箇所などの知見を基に決定された。また検討の上、実現が見込まれる箇所も追加された。この取り組みの詳細は [1] に紹介されている。残りの 40%は、各製品で独自のコードをキープする。この分割方式で、結集した開発を迅速かつ比較的少ない労力で進めることができるといった恩恵を受けることができた。残りのコードをマージするには数年要したが、既にプロダクトラインエンジニアリングの恩恵を得ることができていた。

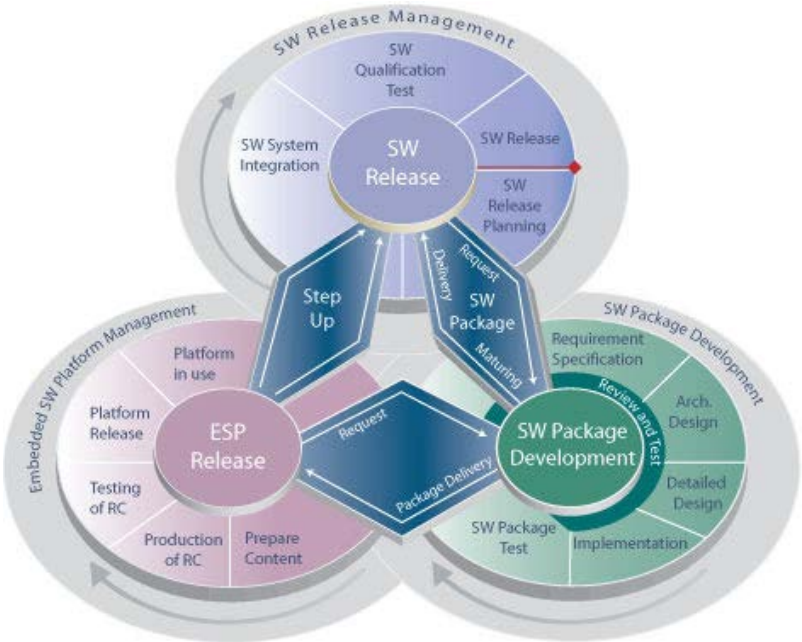


図 59-5　PL Development Process at Danfoss

最終的にバリアビリティの管理にフィーチャモデルを活用することは、当初から予定していた。しかし、このプロダクトラインのフィーチャモデリングは、製品コードの 80%がプロダクトライン資産ベースになるまで開始されることなく、製品間のコードの違いを見出すために必要な、多くのバリエーションが露呈することになった（600 にも上る C++ の#defines）。　それらの多くは単純にソースコードの派生であり、そのまま進めることができるものもあれば、削除することで製品の振舞いを変えてしまいかねない理由から維持すべきものもあった。これらの #defines から、最初のフィーチャモデルを自動生成させて、pure-systems 社のバリアント管理ツールである pure::variants [pv] で管理した。pure::variants は、異なる製品を組上げるソフトウェア資産に必要なコンフィギュレーション（make ファイルやヘッダーファイルなど）を導出して、開発ワークフローの中心的ツールとなっている。

フィーチャモデルの完全利用は、製品の特有な部分（パラメータデータベースと呼ばれる）をプロダクトラインへの取組みに統合した後に実施された。このデータベースは、各製品の一部分で、製品のテクニカルな観点、及び非常に重要なこととしてユーザ視点（使用される用途固有の）で調整するパラメータを持つ。そのため、ユーザの眼に触れるパラメータの選択肢は異なる製品間で多くの違いがあり、それはコード内にも反映する。pure::variants のプラットフォームは、プロダクトラインに則したカスタムパラメータのデータベース管理ツールとして使用された。当然ながらプロダクトラインエンジニアリングの導入で変わるのはツールやコードだけでない。開発組織の変化も求められる。安定していること、よりよくテストされたソフトウェア、新しい機能の採用と製品の修正を迅速化する、といった需要を満たすために、図 59-5 に示す標準的なプロダクトラインのリリースプロセスを採用した。

このプロセスは非常に上手くいった。サイクルタイムは変動するが、通常数週間程度。このプロセスの詳細は[2],[3]に紹介されている。

ソフトウェア資産にプロダクトラインの原理を採用した成果は、要件など他の資産にも拡張された。バリアビリティ情報内の異なるレベルの詳細をカバーするために、要件とコード間のバリアビリティのリンクを介した、階層化されたフィーチャモデルを pure::variants 内で整備した。図 59-6 に、バリアビリティ管理において異なるレベルの成果物間のフィーチャの関係性、およびその効果を図示する。この詳細は、[3] で報告されている。

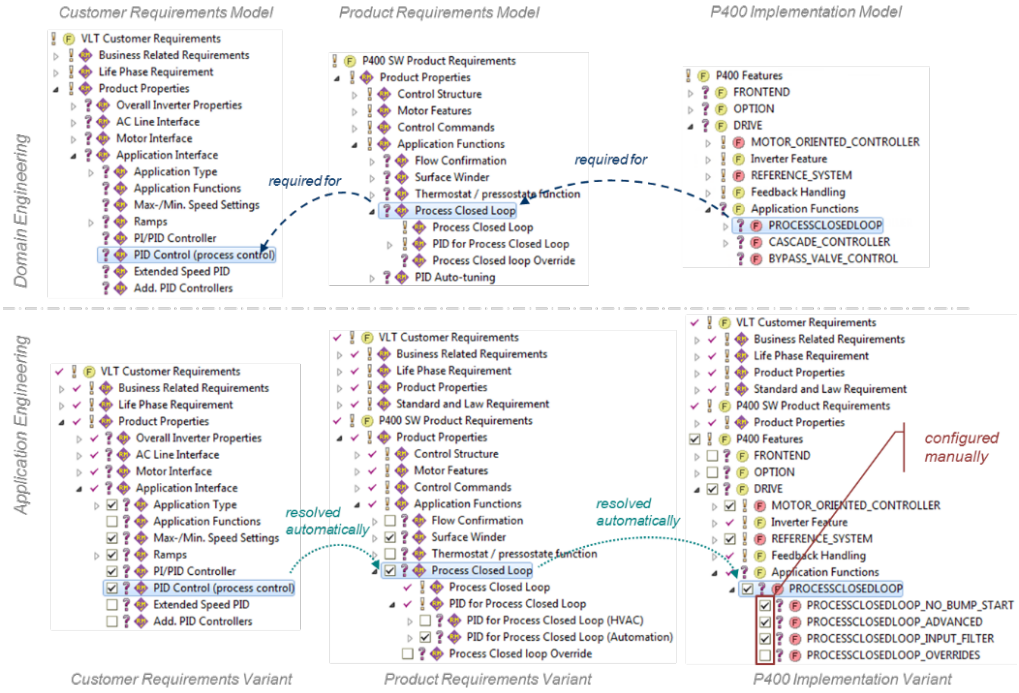


図 59-6　Illustration of product engineering using the reusable assets: Portfolio-Requirements-Implementation

3.1.2. 適用の効果

Danfoss 社でプロダクトラインは大きな成果を得た。その成果の一つに、プロダクトラインの寿命が想定より長かったことが挙げられる。継続的に新機能を統合して市場の需要に順応できるので、このプロダクトラインからの製品は 15 年たっても強い競争力がある。数年先を見越した製品開発と保守ができる高度なコードの共有により、新しいハードウエアプラットフォームへの置換えも実現できるようになった。もちろん新しいハードウエアプラットフォームへの移管には多くの労力を伴うが、その費用等は、同じコードや要件を共有する複数の製品で分割できる。プロダクトライン手法への取り組みにより、新機能を追加することで全製品の用途・応用範囲を進化させることと、サプライチェーンの変化に対応するために変更されるハードウエアの要求に対処することが、同時に行えるようになった。その他興味深い効果として、製品の欠陥が減少したことがある。共有されるコード資産内の欠陥は飛躍的に減少し、殆どの欠陥は、共有化への移管段階にある共有されない資産であった。

3.2.　事例 2: Embedded Security Product Line

２つ目の事例はプロダクトラインエンジニアリングにおける興味深い課題を例証する。プロダクトラインエンジニアリングの導入には様々な変化が求められる。また、組織の体制や、よりテクニカルな部分への、新たな挑戦的課題ももたらす。この事例の課題は、これら両方の組み合わせで、それは組織の体制変更とテクニカルなプロダクトラインのインフラを築くといった比較的シンプルなコンセプトを用いることで難易度を下げることができる。

3.2.1. 適用したプロダクトラインの概要

この事例のプロダクトラインは、組込みシステムのプラットフォームである。各製品は基本的にマイクロコントローラや外部通信機能と、それに対応するソフトウェアパッケージで、よりよく定義された標準化プラットフォームをベースにする。顧客ごとで異なる需要には、様々なマイクロコントローラや、顧客ごとで解釈される特定スタンダード（サブセット）への対応といったプロダクトライン内のバリエーションが必要だ。これにより標準化パーツにも僅かな変化をもたらすことに加えて、顧客ごとで固有のソフトウェアが追加でバンドルされる。プロダクトラインで同時に開発される製品は x 10 種程度で、開発チームに 100 人以上の開発者が関わる。他の似通ったプロダクトラインとの違いからくる課題は、最終製品を開発して出荷するだけでなく、製品開発プロセスに顧客やサードパーティが直接関わることである。一部の顧客は、製品ソースコードを定期的に検査する。システムの一部は外部監査に提供されて、コードの品質や安全性について検査され、スタンダード準拠であることの認証を受けなければならない。定期的な検査は、数週間以内の間隔で行われる。監査と認証は、顧客によるレビューや、現行の反復開発における内部レビューほど多くは実施されない。ただし、失敗すると、深刻な遅延と費用増大を招くことになる。

3.2.2. 事例 2 の課題

一製品を扱う場合は、これらの取組みに必要な資産は比較的容易に生成できていた。基本的にソースコードは、リポジトリからドキュメントとひとまとめにして引き渡された。プロダクトラインの取組みでは、これが困難になった。プロダクトラインのソフトウェア部品は様々な標準プログラミング言語（Java、C、複数のアセンブラ）で構成される。ドキュメントは部分的にソースコードから Doxygen[3]を用いて生成され、一部は Microsoft Office ツールや LaTeX[4]で人手によって記述される。これらいずれの言語やツールも独自にプロダクトラインを意識しない。しかしながら、これらに対してバリエーションポイントのコンセプトを用いてスーパーセットな扱いを実施することができる。以下の 表 59-1 で資産に用いられたバリエーションポイントのコンセプトの概要を示す。

---

[3] ソースコード・ドキュメンテーション・ツール：http://www.doxygen.jp/

[4] テキストベースの組版処理システム/ A document preparation system：https://latex-project.org/

表 59-1　Variation point concepts of used languages

言語	バリエーションポイントのコンセプト
Java	Java annotations
C/C++	#if flogical expressiong #elif flogical expressiong #else #endif
Assembler – a51	#if flogical expressiong #elif flogical expressiong #else #endif $if flogical expressiong $elif flogical expressiong $else $endif

Assembler – arm	IF flogical expressiong
	ELIF flogical expressiong
	ELSE
	ENDIF

製品レベルのバリエーションの静的な最小変位と結合は、性能やメモリ制限などの特定のテクニカルな制約に対する根本的要素であり、最終製品の結合負荷を軽減することを主眼にした仕掛けである。これらのコンセプトは上手く働いて、性能を満たし、メモリ制限も維持されて、再構成も容易で、またこれらツールや言語の標準的なアプローチをベースにしたため、これらの使用によって開発者が困るようなことはなかった。異なるツールや言語内のバリエーションポイントの調整は、pure::variants 内のフィーチャモデルとバリアントモデルに実現されて、メンテナンスされた。

ここまでは教科書どおりのプロダクトライン成功例である。ある意味でプロダクトラインの標準的な成功事例に見える。でも実は、そうでもない。多くの資産が単一製品には使用されないバリエーションも含むので、一製品のソースパッケージのサイズが増大した。更なる進化とバリエーションで、さらに多くの関係ないバリエーションが引き渡されるコードに含まれる。これは必要以上にソース資産が大きくなる上に、製品構成のみに照らして評価しなければならないので、検査や認証作業がより困難になる。しかしこれらは成果物のソースなので、作業を容易にするツールの支援、あるいは単純に時間をかけるかの、いずれかの選択になる（しかし市場のスピードに遅れを取るわけにはいかない)。ただこれは大きな問題では無い。

厄介なのは、バリエーションポイントからどのような機能が実装されているかわかるので、IP(知的財産)として保護されるべきパーツが含まれることになるが、このプロダクトラインは、同

一ビジネスで競合となる複数の顧客に供給されるので、完全なコード全てをさらけ出すということが許されない。

ビジネスの観点として、個々の顧客にプロダクトラインの 150%資産を配布することは認められない。効率の良いコード（性能要件を満たさないとビジネスにならないので）に着目したアプローチ（静的な結合に標準的なコンセプトを使用する）を選択する一方で、100%のソース資産を顧客や監査に提供する必要があることに、問題が生じた。要求されるパッケージの構成や処理を人手によって行われなければならなかった。頻繁に生じる顧客とのやり取りとタイミング要求の組合せによるストレスやコスト増は深刻であった。異なるアプローチに使用するために、150%ソース資産をリファクタリングすることに疑いの余地はなかった。百万行に及ぶコード資産があり、その多くがハンドコードされたもので、リファクタリング対象とされてきた事実を踏まえた。プロダクトラインの取組みを行わなくてもやれてきたが、それはビジネス上の目標を犯すことになっていた（開発コストが高騰していたので議論するまでもなく、また製品ごとの 100%資産の為にもリファクタリングは必要であった)。それゆえ、テクニカルな解決策が求められた。本質的に必要とされたのは、製品の全てのタイプの資産（C、Java、2 種類のアセンブラ）に対して、150%ソースから100%ソースへの変換であった。

3.2.3. 解決策

プロダクトラインのバリアント固有の資産の生成は、pure::variants で行われた。それは資産を選択して、バリアント変換に固有製品バリアント以外の資産を除外することに使用される。これはバリアント固有の構成をバリアントモデルに設定して、150%ソース資産内のバリエーションポイント情報と結合させる。図 59-7 に pure::variants のバリアントデータ生成フローの概要を図示する。

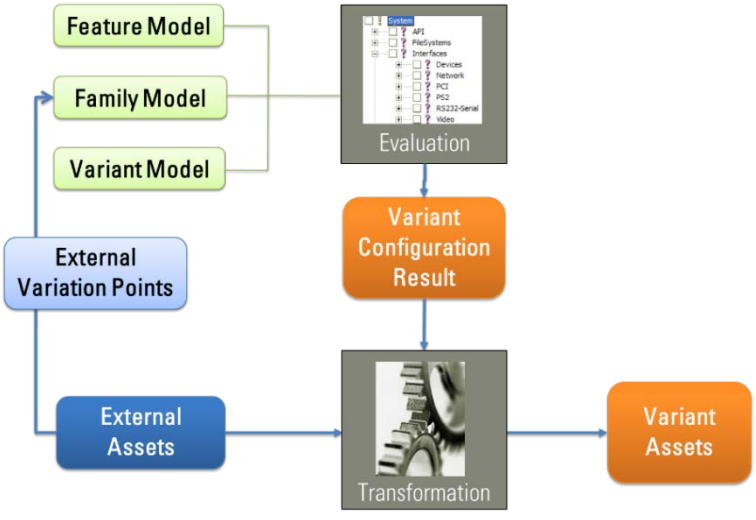


図 59-7　Variant Generation Dataflow with pure::variants

解決策の一つは、pure::variants で選択した製品バリアント固有に関連するファイルのみを使用すること。これにより、製品データの規模と、製品に寄与しないファイルを飛躍的に削減した。これは選択されるファイル内に存在するバリエーションについては放置する。仮のソースコード例（選択されるファイル内に存在する）で、この課題を説明する。編集されなければ、コード片は存在するし目に触れる。ファイル内のこれらパーツは、必ずしもオプションとは限らないと解釈できる。必要なことは、オリジナルのファイルをツールが入力として取り込んで、２つのバリエーションポイントの構成（VP_SEC_PROTO13 と VP_CUSTOMER_X）によって、バリエーションを解決するソースファイルを生成することである。また一方 #ifdef SYS_WINDOWS は、製品のバリエーションポイントではないので残る。

```
…
#ifdef VP_SEC_PROTO13
#ifdef VP_CUSTOMER_X
/* customer specific protocol implementation, protected IP */
…
#else
/* standard specific protocol implementation */
#ifdef SYS_WINDOWS
#endif
#endif
#endif
```

図 59-8　仮のソースコード例：顧客ごとで異なるコード片が存在する

標準の C プリプロセッサは、全ての#defines を解決する出力の生成に使用できる。しかし、ソースコードが結果として読めないので、100%ソースコード生成の選択肢にはならない。ただし、C ソースコードを読んで解析することができる、良い C 言語の構文解析ツールがある。そのような解析ツールを用いて、バリエーションポイントに関連する#ifdef 等のみを解決することで、100%ソースコードを生成できる。このアイデアを基にして、最終的なソリューションが pure::variants への拡張として、多くの既存ブロックを踏まえて開発された。図 59-9 でデータフローを図示する。一つのソースファイル（いずれのタイプでも）が、Variant Result Model (VRM)（= バリアント構成をモデル化したもの） の情報を基にするセレクタに渡される。そして pure::variants の他の変換ステップによる処理のために単純にファイルをいつも通りの結果（150%から 100%変換への対象で無いデータ）として受け渡すか、あるいは、ブラインド化と称するプロセスに渡される。

ブラインド化はバリエーションポイントの抽出（Analyzer）からなり、その出力はバリエーションポイントを汎用バリアビリティ交換言語で記載する。そしてソルバーがどのようにバリエーションポイントを有効にするかを決定して、ブラインダーに結果を伝達する。ブラインダーは基になるソースファイルから、関連しない全てのバリエーションポイントを取り除く。２つの出力方式があ

る。ひとつは使用されないバリエーションをブラインドテキストで置き換える（それゆえブラインダーと称される）、もう一つはそれらをファイルから削除する。ブラインド化は行番号を維持できることなどから全製品バリアントで同期する利点があり、また内部の監査には十分と考えられている。ブラインドテキストの存在そのものが余計な情報源になるなら、削除の手段を講じることができる。

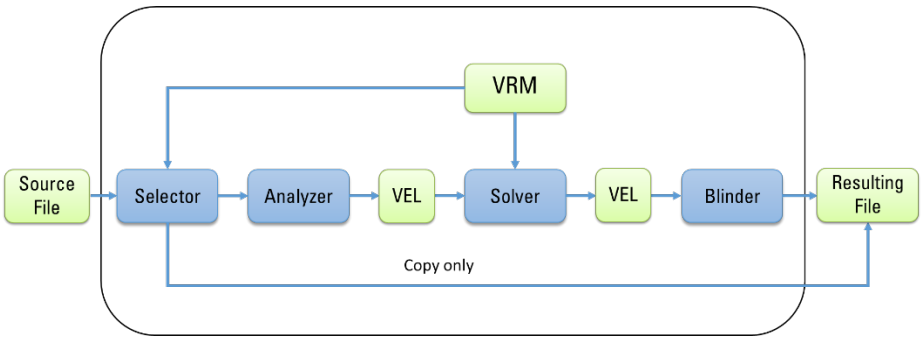


図 59-9　150% Source to 100% Source data flow in the pure::variants Source Transformation

上述のサンプルコードの場合、ブラインド化と削除方式の出力結果の違いは、以下のようになる。(Configuration: VP_SEC_PROTO13 == true and VP_CUSTOMER_X == false)

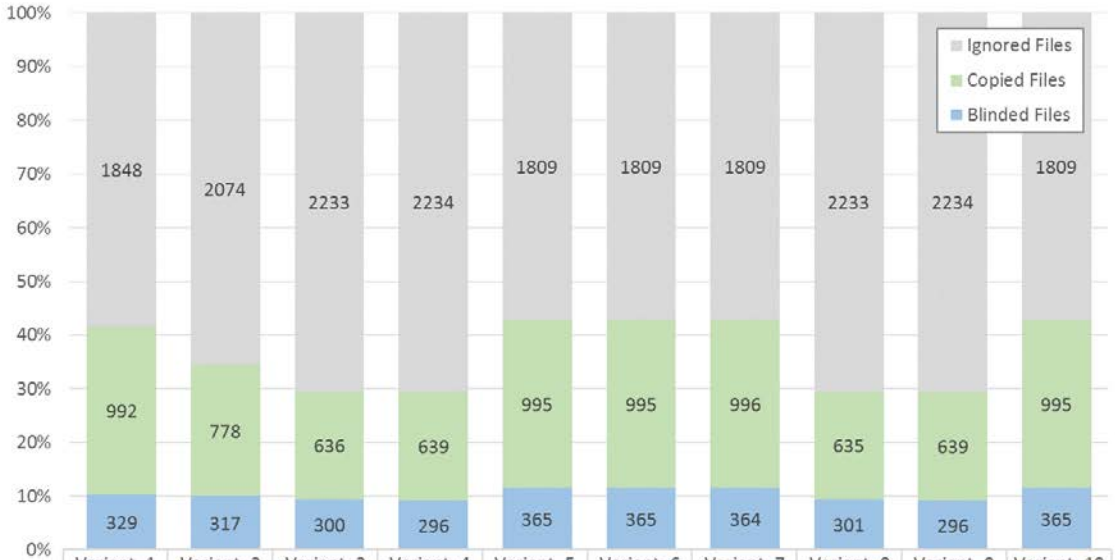
… /* *　　Blinded */ /* standard specific protocol implementation */ #ifdef SYS_WINDOWS #endif	… /* standard specific protocol implementation */ #ifdef SYS_WINDOWS #endif

図 59-10　ブラインド化（左）と削除方式（右）の違い

４つの異なる言語（C、Java、２種類のアセンブラ）に対するこの方式の実装は、pure::variants のフレームワークとオープンソースの構文解析ツールの十分な品質のお陰で、多くの作業は必要なかった。またソースコードには余分なものがなく製品内容を正確に反映している。そのため、この変換を使用可能とすることで、API ドキュメント生成ツールの Doxygen もまた一切の修正なしに使うことができる。

3.2.4. 結果と効果

労力に見合った成果を得たか。人手によるパッケージングは、数百に上るファイルのマニュアル検査に何日も必要であり、また一部の IP が見過ごされてファイル内に残ることなど保証も取れない。自動変換方式なら、一貫した品質のパッケージの情報生成が数分で済む。ファイルの 2/3 がブラインド化の前に除外され（図 59-11 参照）、インクルードされるファイルの 10～15% 部分（製品ごとでバリアント構成が異なる）がブラインド化される。この自動化で顧客からの要求に対して、ソースを提供できるまでの時間を飛躍的に改善することができた。またバリアントの進化もより良くサポートされるようになった。ソース資産内の製品に関わらない全ての変更はブラインド化できるので、全製品に対して、変更による影響から再検査や再認証が必要かどうか、ソースレベル上での判断が容易になった。



	Variant_1	Variant_2	Variant_3	Variant_4	Variant_5	Variant_6	Variant_7	Variant_8	Variant_9	Variant_10
Ignored LOC	721504	758644	791639	793961	654944	654944	654944	787967	790289	651272
Copied LOC	514034	473640	448000	443249	582340	582340	582895	451781	447033	586124
Blinded LOC	29394	32648	25293	27722	27648	27648	27093	25184	27610	27536

図 59-11　Selected/Blinded Lines of Code per Variant Comparison

3.3. 事例 3（自動車システムの例）: Exterior Lighting and Speed Control

３つ目は、自動車システムの研究プロジェクト SPES-XT（独）の事例を紹介する。対象の車載システムは外装部品のアダプティブ・ライティングシステムとスピードコントロールシステムの２つからなる。

ライティングシステムの主な機能は以下の通り。

- 方向指示灯（方向指示とハザードライト点滅）
- ヘッドライト：ロービーム（昼間走行灯とコーナーライト）

- ヘッドライト：ハイビーム（対向車が無い場合の自動動作機能）

スピードコントロールシステムの主な機能は以下の通り。

- クルーズコントロールとアダプティブクルーズコントロール
- 車間警告
- ブレーキアシストとアシストブレーキ
- 速度標識検出とスピード制限機能

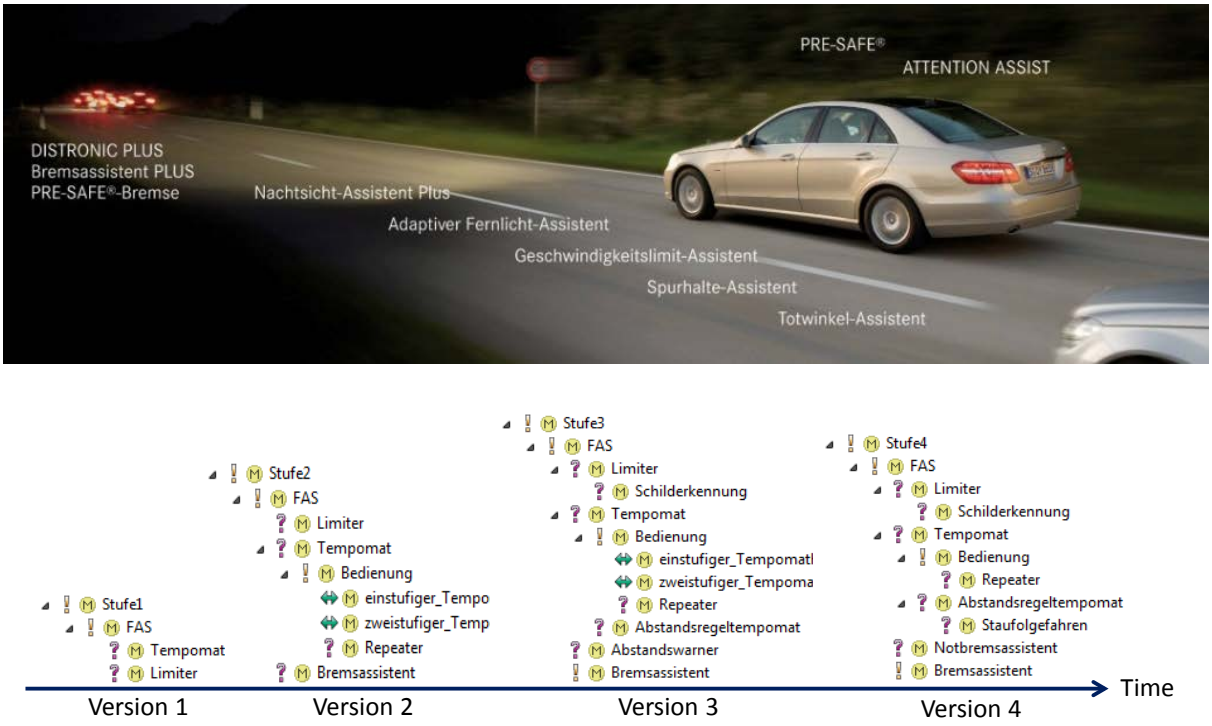


図 59-12　Evolution steps of the automotive system cluster

２つのシステムを用いた主な動機は、機能間の何らかの相互作用を具体化することであるアダプティブクルーズコントロールによって決定される速度情報の影響を受ける、自動ハイビーム機能。相互作用の課題に加えて、事例では主に進化についてフォーカスした。一般的に新しいプロジェクト（モデルチェンジや新モデル）はスクラッチ開発では無く、既存製品をベースにするので。従って新しい機能は追加されるし、既存機能は変更されるか削除される。

自動車の事例は、連続する４バージョンの仕様について考察することで、システム進化の様相を示している。顧客視点の各フィーチャに基づいて、これら異なるバージョンを図 59-12 に示す。この事例、およびそのような進化をサポートするための挑戦的課題は、一貫したバリアント管理と再利用であり、これは全ての成果物（要求仕様、モデル、コード、テスト、安全性ケース、各種ドキ

ュメントなど）が正しい順序で、正しい手順で扱われることで一貫性が維持されることである。課題に取組むために詳細なプロセス [5][6]を定義して、進化の要求から変更が持ち上がった際に、一貫した結果を得るために、誰によって、いつ遂行されるべきかを明確に記載した。このプロセスの定義には複数の開発の役割担当者を巻き込む。なぜなら開発される製品の複雑さから、その責任範囲が分割されてしまうからである。このプロセス定義の抜粋（図 59-13）に、３つの異なる役割が示されている。

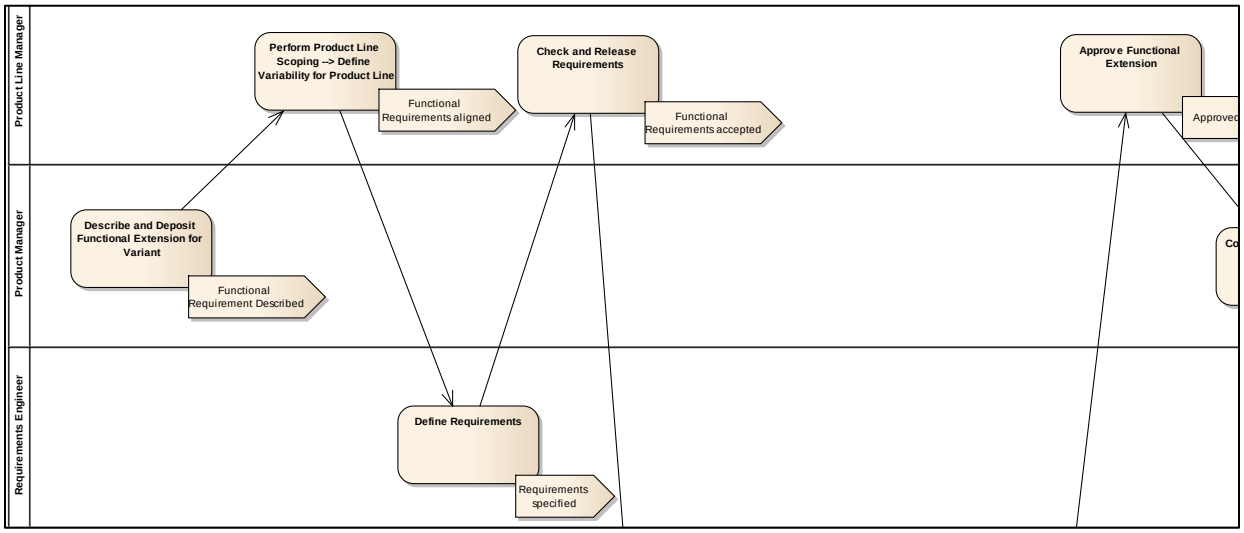


図 59-13　Consistent Variant Management throughout the development process

一般に変更要求が開始点になる（例えばブレーキアシスタント機能の追加：図 59-12 内の Bremsassistent)。要求は、その関連性を吟味して、固有のバリアントのみかプロダクトラインに実現させるのかを決定するために、プロダクトラインの責任者に伝達される。そして決定内容に応じて、プロダクトラインのバリアビリティモデル（フィーチャモデル）を、それに順応させるか、あるいは変更しない。その後、要求は要求仕様の担当者に通達され、要求仕様に追加、修正、削除等を行い、またそれに属するバリアビリティモデル（ファミリーモデル）を更新する。そして、その他の開発段階（モデリング、実装、テスト、アプリケーションなど）も、バリアビリティモデルを必須の作業として取り入れて実施する。

プロセスの定義に加えて、pure::variants、Doors、Matlab/Simulink、Jira を用いたビルド環境で、その有効性を評価した。この中で、Jira は変更管理ツールとして使用され、課題のワークフローとしてプロセスを実施するのに役立った。他のツールは Jira と連携するように拡張されて、タス

[5] プロセスの詳細は、参考文献の[4][5][6]を参照してください。

[6] Due to space limitations, the process is not fully described in detail and just a brief overview is given here. To get a closer look and a deeper understanding the reader is referred to [4],[5] and [6].

クをフェッチして変更要求のステートを更新して、ワークフロー定義内の次のステップへの始動のきっかけとなる。この設備構成で、上述した進化のシナリオで、実エンジニアによって評価が開始された。全てのプロセス上の役割や、業界で採用される全てのツールを使用したわけではないが、成果を確認できた。プロセスや、それをサポートしたツールに対して、大いに役立ったと、エンジニアからポジティブな反響を得ることができた。例えば、プロセスの定義やツールによって支援されるワークフローを起因にした成果物上の間違いは無く、独断的に開発された資産とバリアビリティモデルの一貫性は担保される。

4. 考察

事例から、プロダクトラインエンジニアリングによってビジネスと技術面の両方に効果が得られるという結果が得られている。共通するのは、高品質になってレポートの課題が削減されたことによるメンテナンス作業の軽減、新しい製品バリアント開発期間の飛躍的な加速、コード量や製品開発のための資産を減少できたこと、市場により多くの製品を投入できるようになったことなど。事例ごとで改善の核となる部分に違いはあるが、多くの共通点が相互に見受けられる。

参考文献

[1] Jepsen, H.P. and Dall, J.G. and Beuche, D. 2007. Minimally Invasive Migration to Software Product Lines. In Proceedings of the 11th International Software Product Line Conference (SPLC '07). IEEE Computer Society, Washington, DC, USA, 203-211.

[2] Jepsen, H.P. and Dall, J.G. and Beuche, D. 2009. Running a software product line: standing still is going backwards. In Proceedings of the 13th International Software Product Line Conference (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, 101-110.

[3] Krzysztof Sierszecki, Michaela Steffens, Helene H. Hojrup, Juha Savolainen, Danilo Beuche: Extending variability management to the next level. SPLC 2014: 320-329

[4] Methodik für ein durchgängiges Variantenmanagement und Wiederverwendung im Engineering von Embedded Systems, André Heuer, Tobias Kaufmann, Thorsten Weyer, Michael Käßmeier, Peter M. Kruse, Michael Himsolt, Christian Manz, Martin Große-Rhode, Sebastian Schröck, Michael Schulze, SPES-XT project result, 2015

[5] Advanced Model-Based Engineering of Embedded Systems, Klaus Pohl, Heinrich Daembkes, Harald Hönninger, Manfred Broy, Springer, 2016

[6] SPES_XT Abschlussveranstaltung /EC5: Durchgängiges Variantenmanagement und Wiederverwendung
http://www.fuji-setsu.co.jp/files/SPES_XT_Projektabschluss_EC5(annotated).pdf 英語訳付き

独立行政法人情報処理推進機構 技術本部 ソフトウェア高信頼化センター（IPA/SEC）