S/W開発現場におけるソフトウェアプロダクトラインの適用と再利用部品整備の試み

An attempt to apply Software Product Line Development and to build reusable components up for software develpment

赤坂 賢洋* 原 周*
Norihiro Akasaka, Hiroshi Hara

ソフトウェアプロダクトライン（Software Product Line, 以降SPLと略記する）は、ソフトウェア部品の再利用性を体系立てる手法として理解されているが、実際にソフトウェア開発現場に適応した事例は多くない。SPLの考え方を具体的にするには、共通部品を、どのような手段で実現するかを決定しなければならず、一般に、この問いに答えを与えるために、開発現場の特性や事業環境を踏まえた難しい判断が要求される。

SPLは、中長期的なロードマップに基づく系列製品の継続的な開発を念頭に置いている。筆者所属部門のソフトウェア開発では、例えば特定の家電製品など、SPLでよく取り上げられるような典型的な系列製品開発はないが、通信・組込み分野の継続的な開発を行っている。そのような現場で、今回我々は「ソフトウェアフレームワーク」に着目し、それにSPLの考え方を適用することで、再利用部品の定義とその開発プロセス、および再利用部品を用いた製品開発のプロセスを構築することを試みた。本稿では、その試みを報告する。

Software Product Line（SPL）is known as a method for making systematic reuse of software components, however, it gives some difficulty to engineers or managers to apply it to their software development site. To realize the concept of SPL they should determine how to implement the reusable components, it generally forces them to make difficult decisions in consideration of their development site characteristics and business environment.

SPL is directed to continuous development of series products based on the medium- or long-term roadmap. In our software development department we are continually developing the communications equipments and embedded instruments though we don't develop the typical series products such as some home appliances taken up well in the context of SPL. In such a site, we focused on "software framework" and tried to define our reusable components and to construct the development process by applying the concept of SPL, in addition we attempted to build the process of product development with those components. In this paper, we report that attempt.

1. まえがき

SPLは、製造業におけるプロダクトライン開発をソフトウェア開発に応用したものである。そのベースとなるプロダクトライン開発の考え方は、企業が大量個別生産を実現する手法として発展した。

自動車メーカを例にとる。メーカの都合を考えれば、単一の車種だけを生産し続ける方が効率が良い。新車種の開発や、それらを生産するライン構築への投資が不要になる。同一車種を生産し続けるラインは改良を重ね、精錬化し時間と共に効率が向上する。しかし、それでは顧客の幅広いニーズに応えることができない。多くの顧客を獲得し、事業規模を拡大するためには、大きさ、デザイン、性能の異なる多数の車種を開発し、生産する「大量個別生産」が必要になる。

この大量個別生産を実現するために、単一車種だけを生産するメリットの裏返しとして存在するデメリットを克服しなければならない。すなわち、少ない投資で、

＊関西事業部　第三技術部

様々な車種を開発し生産する手法を確立しなければならない。その手法の１つがプロダクトライン開発である。

プロダクトライン開発の最も重要な構成要素に、「プラットフォーム」がある。プラットフォームは、広義には多様な製品の共通基盤のみならず、周辺の技術力も含む［1］が、狭義には前者の意味でとらえてよい。つまり、複数の製品に共通したものを「共通基盤」として切り出し、個々の製品は、その共通基盤の上に、製品固有のパーツ・機能を付加して組み上げる。このプロダクトライン開発により、大量個別生産を効率よく実現できるようになる。

ひるがえって、ソフトウェア開発に目を向ける。その黎明期、ソフトウェアと、それを駆動するハードウェアとを組み合わせた製品開発の主体はハードウェアにあった。ソフトウェアはハードウェアの付属品であり、小規模だったため、オーダーメード的に個別に開発することができた。

しかし、製品に要求される機能が増大し続けた結果、機能実現に柔軟性と可換性があるソフトウェアのウエイトが増し、逆にハードウェアは多様な機能の実現をソフトウェアに委ねることによって汎用化していった。結果として、個別かつ大規模なソフトウェアの開発コストは増える一方となった。この状況を解決するため、一般の製造業におけるプロダクトライン開発をソフトウェア開発に応用する、ソフトウェアプロダクトライン開発（SPL）のニーズが高まった。

2. SPLのエッセンス

SPLもプロダクトライン開発の一種であることから、プロダクトライン開発における「プラットフォーム」の考え方を含む。SPLにおけるプラットフォームの例として最も理解しやすいのは、複数のアプリケーションで共通に利用できるソフトウェア部品、すなわちライブラリである。図１に、ライブラリを用いた共通化の概念を示す。図中のソフトウェア製品Ａ、Ｂ、Ｃは、共通ライブラリＬを利用しつつ、製品固有の機能を実現している。

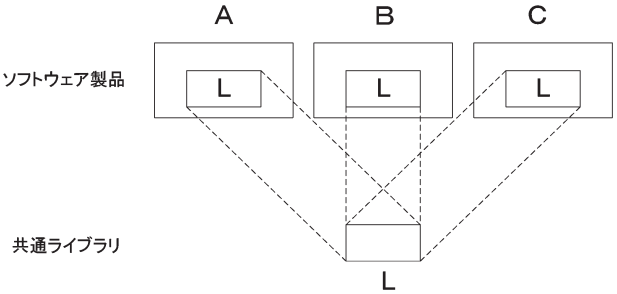

図１　共通ライブラリを用いたソフトウェア製品開発の概念

ここで、共通部品を利用することの効果を最大化するには、Ｌの規模を大きくしなければならない。一般に、共通部分の規模を大きくすると、汎用性が下がり、利用性も下がる。しかしライブラリには、「必要な部分のみを利用し、そうでない部分は利用しない」という、「つまみ食い」ができるメリットがある。つまり一般に、Ｌの規模を大きくした場合の個別生産（ソフトウェア開発においては、大量個別生産における「大量」の概念は重要でないため、本稿では「個別生産」の用語を使う）の形態は、図２のようになる。

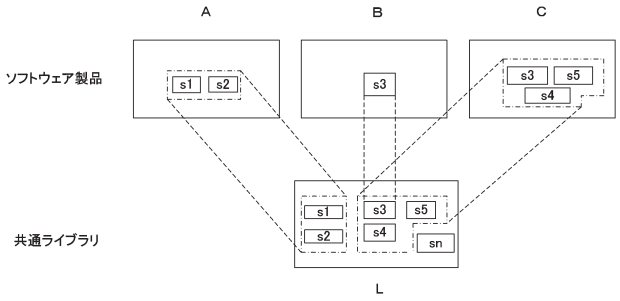

図２　共通ライブラリを用いたソフトウェア製品開発の概念（実際）

つまり、規模の大きなＬの部分s1, s2, s3,…snから、適宜必要なものを利用する形態である。

しかし、これでは所期の「共通部品を利用することの効果を最大化する」という目的を達成していない。自動車開発に例えると、ミラーやワイパーを共通化していることにとどまり、個別生産を効率化したとは言えなくなる。自動車メーカでは、この課題を「車台の共通化」で解決した。つまり小さな単位の部品ではなく、自動車の構造のベースとなる車台のレベルで共通化し、そこに製品個別のパーツを組み上げていくことで、個別生産とコスト削減の両方を実現した。

しかしここで、前述の課題を考慮しなければならない。共通化の効果を高めるためには、車を構成する出来るだけ多くの部分を車台に含める方が良い。しかし、例えばエンジンを車台に含めるとどうなるであろうか？同じエンジンを搭載し、外観だけが違う複数の車種を生産することはできるが、エンジンが異なる車種を開発するのにその車台は適さなくなってしまう。どこまでを車台、すなわちプラットフォームとするかは、そのメーカの生産に対する考え方を映す胆の部分となる。

自動車における車台の共通化をソフトウェア開発に当てはめた場合、プラットフォームを「ライブラリ」ではなく、「フレームワーク」に置き換えるとしっくりくる。図３に、フレームワークを用いたソフトウェア開発共通化の概念を示す。

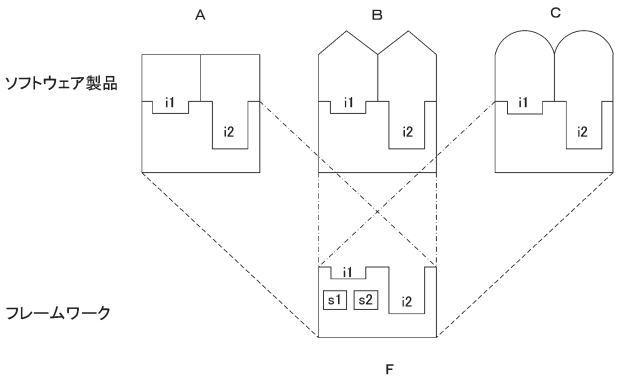

図３　フレームワークを用いたソフトウェア製品開発の概念

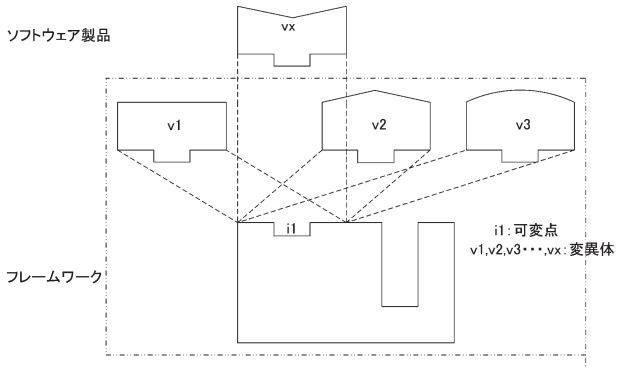

図４　フレームワークにおける可変点と変異体

フレームワークは、車台のように、ソフトウェアの基盤部分に存在する。個々のソフトウェア製品は、この基盤の上に、固有機能を積み上げて開発する。フレームワークには、それだけで機能として完結する部分（s1, s2, s3,…sn）のほか、積み上げる固有機能の形だけを規定するインタフェースの部分（i1, i2, i3,…in）がある。つまりフレームワークには、首尾整った変更不可能な部分と、変更可能な部分が存在する。この「変更可能な部分」は、個別の製品の開発者に、「変更できる部位はどこであり」、「その部位をどのような形で作ればよいか」を示唆している。

ここで、SPLのエッセンスともいえる重要な概念「可変点（variation point）」が登場する。SPLにおける可変点は、より汎用的な定義が付与されている［１］が、ソフトウェア開発の現場に立つ我々としては、このようなフレームワークの変更可能な部分、しばしば「カスタマイズできる部分」のように呼ばれている部位を可変点と理解してよい。可変点に、個別のソフトウェア機能を加えて実際の製品を作り上げる一般的な方法は、可変点の定義するインタフェースに従い機能部位を開発し、フレームワークに積み上げるやり方である。

一方、別のやり方も存在する。それはフレームワークが、可変点に適合するいくつかの機能部位を持ち、提供する方式である。図４に、その概念を示す。

ここで、可変点に適合する機能部位v1, v2, v3は、共通のインタフェースをもつが、実現する機能が異なるソフトウェア部位である。このような、可変点に対する可換なソフトウェア部位により実現されたものを、SPLでは「変異体（variant）」と呼び、可変点と並ぶ重要な概念である。変異体には、図４のようにフレームワークが提供するv1, v2, v3を利用したもののほか、製品Ａの開発者が独自に開発するvxを利用したものも含む。例えば組込み機器で、デバイスに異常が発生した場合にある機器ではLEDの赤色点灯で示すが、別の機器ではブザーを鳴動させることがある。この場合に、フレームワークがLEDの赤色点灯とブザー鳴動の変異体を提供していれば、個別製品の開発者は、そのどちらを利用するかを決めるだけでよい。一方、デバイス異常時にネットワークに特定のメッセージを流す別の機器を開発したい場合は、そのような変異体をその機器の開発者が独自に開発することになる。可変点と変異体は、SPLでは図５のような表記法を用いて表現する。

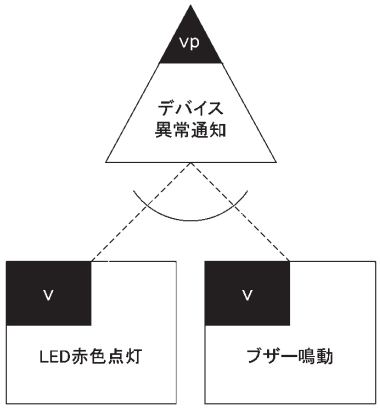

図５　デバイス異常通知の可変性モデル（例）

この表記法により表現されるモデルを可変性モデルと呼び、表現された図を可変性図と称する。ここで、ある製品の仕様が、デバイス異常時にブザー鳴動で知らせるものであれば、図５の「ブザー鳴動」の変異体を選ぶ。このように、変異体を決定することを、SPLでは「可変性の固定」と呼ぶ［１］。

SPLの可変点と変異体の概念は、オブジェクト指向における継承の概念と親和性がある。例えば図５に示す可変性モデルを実装する場合、「デバイス異常通知器」というオブジェクトを考え、その具象クラスとして「LED」「ブザー」を設計すると、そのまま以下のクラス図を描くことができる。

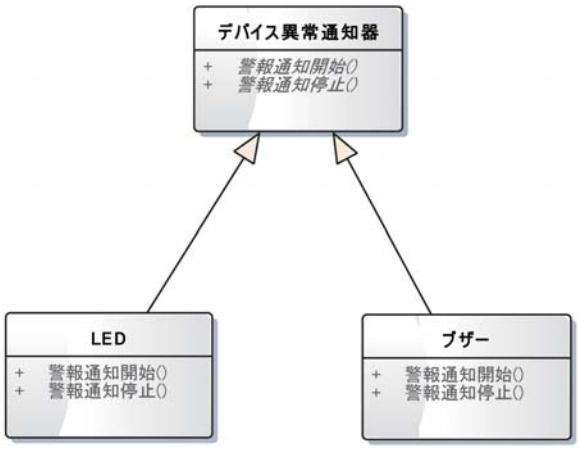

図６　デバイス異常通知器のクラス図表現

デバイス異常通知器クラスは、仮想的な操作「警報通知開始」と「警報通知停止」をもっている。おのおのが、「LED」具象クラスでは赤色点灯と消灯、「ブザー」具象クラスではブザーの鳴動開始と停止として実装される。

このことから、フレームワークをオブジェクト指向フレームワークとして実装することは良いアイデアであるが、実現に際しては、再度3.1節で考察することとする。

ここまでで、SPLのエッセンスとも呼べる重要な概念「可変点」と「変異体」を述べた。SPLは一般に、中長期的なロードマップに基づく系列製品の継続的な開発を念頭に置いていると考えられているが、これまでの議論をまとめると；

「『可変点』と『変異体』を明確に定義したフレームワークを利用することで、一般のソフトウェア開発現場でもSPLの恩恵を享受できる」

ともとらえられる。筆者らはこの考察に基づき、通信・組込み系開発を主とする我々のソフトウェア開発現場にSPLを導入することを試みた。次章より、その試みの具体的な内容を述べる。

なお、余談ながら、プラットフォームを利用したプロダクトライン開発で先行した自動車産業では、近年「脱プラットフォーム化」の流れもある［２］。脱した先は「モジュール化」であり、これはソフトウェア開発における「ライブラリによる共通化」に近い。自動車産業のこの流れをソフトウェア開発に置き換える考察も有意義と思われるが、本稿では言及しないこととする。

3.　SPLプロセスの構築

3.1　プラットフォームの設計

前節までで述べたように、SPLの導入に際して最もコアな要素は、プラットフォームの設計である。その設計に際しては、大きく以下の３つの問いに答えを出さなければならない。

⑴　プラットフォームのアーキテクチャはどのようなものか？

⑵　プラットフォームに含める共通機能は何か？

⑶　プラットフォームはどの言語で実装するか？

次節より、我々の検討の経緯と結論を述べる。

3.1.1　プラットフォームのアーキテクチャ

次節の検討内容にも関連するが、我々の開発現場では、通信・組込み分野という大括りな系列開発である。H/Wは製品ごとに個別で、採用されるOSも多様である。これらは我々の開発現場の課題であるが、開発現場の課題を解決するためにSPLを導入するのであるから、我々の導入するSPLは、これらの課題を解決するものでなくてはならない。そこで繰り返しとなるが、我々は、まず以下の３つの課題をプラットフォームアーキテクチャの課題として識別することからスタートした。

課題１：我々の開発現場に適用できるプラットフォームとはどのようなものか？

課題２：個別のH/Wに対応できるプラットフォームとはどのようなものか？

課題３：個別のOSに対応できるプラットフォームとはどのようなものか？

以上を課題として識別し、それらに対する解を検討した。

⑴　ドメイン依存レイヤと非依存レイヤの分離

例えば冷蔵庫やエアコンなど、特定の系列製品の継続的な開発をターゲットとしたプラットフォームであれは、それらの製品の共通機能を抽出してプラットフォームとすることがSPLのセオリーである。これを、SPLでは「ドメイン要求開発」と呼ぶ。図７に、ドメイン要求開発のイメージを示す。これについては、3.1.2節で詳しく述べる。

しかし、我々の開発現場では、このような典型的な系列製品開発とはやや趣を異にしている。特定のH/Wに依存した組込み開発もあれば、汎用のPCやサーバ上で稼働するアプリケーション開発もある。分野では、情報通信システムの開発が主ではあるが、それらとて、伝達する主信号を扱う通信H/W向けの組込みF/Wを開発することもあれば、それらを制御監視する、ヒューマンインタフェースをもつ監視システムを開発する場合もある。

そこでまず、プラットフォームを「ドメイン依存レイヤ」と「ドメイン非依存レイヤ」に分離することを考え

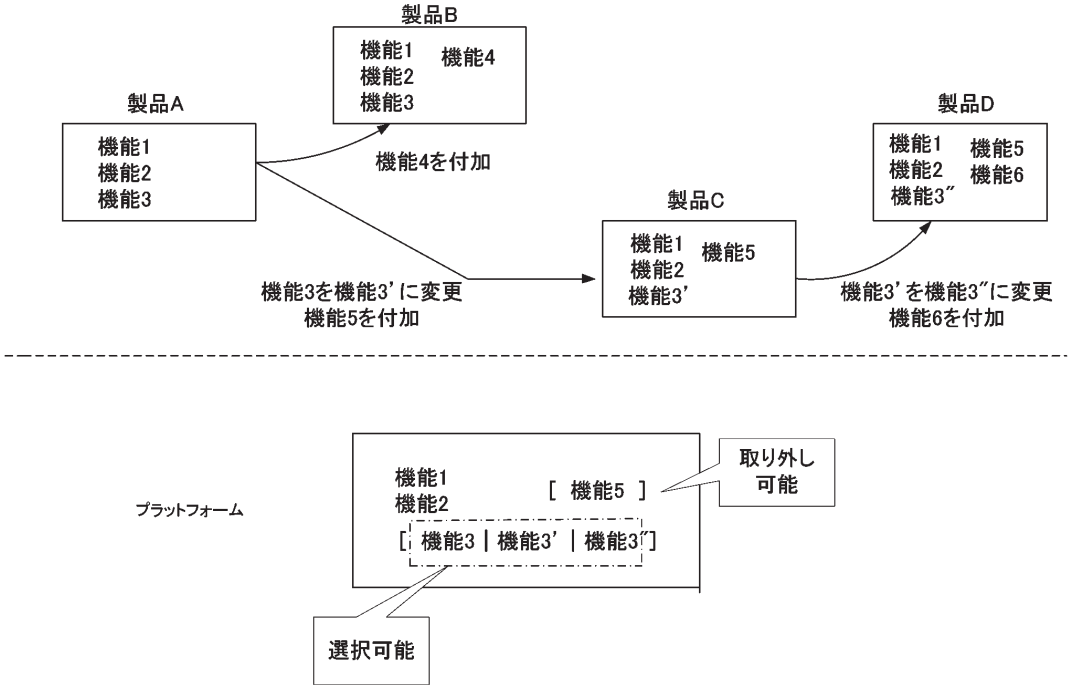

図７　ドメイン要求開発のイメージ

た。図７のイメージで規定された典型的なSPLプラットフォームでは、対象とする系列製品が明確である分、共通部分により多くの機能を含めることができ、SPLの利点である「共通化による開発コスト削減」の効果は大きい。しかし、事業環境の変遷等により対象とする系列製品が変わった場合、別のプラットフォームをゼロから組み上げなおさなければならない。これを避けるために、プラットフォームは「ドメイン依存レイヤ」と「ドメイン非依存レイヤ」に分離することが必須と考えた。我々のプラットフォームの「ドメイン依存レイヤ」にどのような機能を含めるかはこの時点では検討に至っていなかったが、「ドメイン非依存レイヤ」に含める機能はこれまでの開発経験からイメージできる。そこで、まずこのようにプラットフォームのレイヤを二分し、「ドメイン非依存レイヤ」から検討を着手した。

なお、「ドメイン依存レイヤ」「ドメイン非依存レイヤ」おのおのに含める機能の考察は、3.1.2節で述べる。

⑵　個別H/Wへの対応

プラットフォームは、個別で互いに関連のないH/Wに対応しなければならない。これは、H/Wを制御するS/Wでは普遍的な課題で、一般には、H/Wを抽象化するレイヤを設け、そこでH/Wごとの差異を吸収する戦術をとる。このレイヤは、しばしばHAL（Hardware Abstraction Layer）と呼ばれ、多くのOSやミドルウェアが実装している。

我々は、このレイヤを導入する前に、まずS/Wの実装面からH/Wの違いを考察した。我々の開発現場では、S/WがH/Wを制御する場合、大きく以下の３つの方法をとる。

⑻　H/Wのもつレジスタが特定のメモリにマッピングされており、S/Wはそのメモリに対するread/writeでH/Wを制御する。

⑼　UNIX系OS等で、H/Wを制御するデバイスドライバが提供される場合、S/Wは、そのドライバに対するioctlもしくはデバイスファイルへのread/writeでH/Wを制御する。デバイスドライバが提供されない場合は、そのようなデバイスドライバを開発する。

⑽　デバイスドライバの上位に、さらにOSS（Open Source Software）やミドルウェアでデバイス制御の方式やインタフェースが規定されている場合、S/Wはそのインタフェースに該当するAPIやコマンドを実行することによりH/Wを制御する。無線LANデバイスにおけるhostapdやwpa_supplicant、Bluetoothデバイスにおけるbluezがその例である。

このような状況のため、H/Wに対する「物理的」操作は、プラットフォームの機能として提供する必要はないと判断した。

一方、より上位の「論理的」な操作、例えば「LEDを赤色点灯させる」等は検討する余地がある。これについては、3.1.2節で述べることとする。

⑶　OS差異の吸収

我々の開発現場では、種々のOSが採用される。表１に、我々の開発現場で用いられるOSをまとめた。

表１　開発現場で用いられるOS

リアルタイム性	OS	例
―	OS レス	OS レス開発、SDK 開発
リアルタイム OS	TRON 系	NORTi、メーカ独自 OS
	その他	VxWorks
準リアルタイム OS	Linux 系	OpenWRT、WindRiver Linux
	Windows 系	WindowsCE
非リアルタイム OS	Linux 系	Android、各種 Linux ディストリビューション
	Windows 系	クライアント Windows、Windows Server

OSの差異は、開発生産性向上の妨げになる要因の１つであるため、プラットフォームでその差異を吸収することが望ましい。そこで、OSの直上に、OS差異を吸収する「OS依存レイヤ」を設け、OS非依存レイヤと分離した。OS依存レイヤはドメインには依存させないため、そのものはドメイン非依存レイヤに属する。

以上をまとめ、本節冒頭に掲げた課題１、２、３に対応する我々のプラットフォームのレイヤ構造は、図８に示す構造とした。

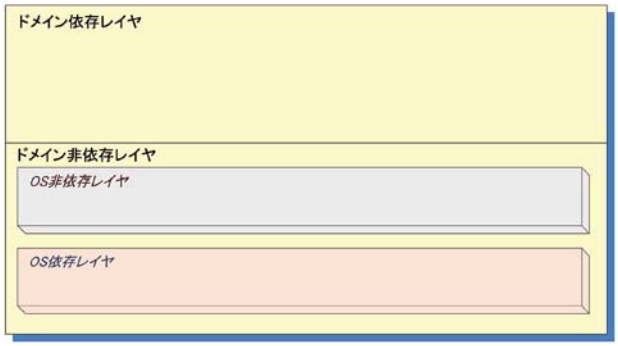

図８　プラットフォームのレイヤ構造

次節で、おのおののレイヤに持たせるべき機能を考察する。本節の課題２についてもレイヤ構造のみでは解決策を提示できていないため、合わせて検討する。

3.1.2　プラットフォームの提供する機能

プラットフォームが提供する機能の検討は、SPLではドメイン要求開発の中で行う。ドメイン要求開発の典型的な成果物の１つは、フィーチャーツリーである。フィーチャーとは、システムの末端利用者から可視な特徴と定義される［1］。フィーチャーツリーは、これをトップダウンなツリー形式で表現したものである。筆者らは、プラットフォームのフィーチャーツリー作成にあたり、過去に実施した、もしくは現在実施しているプロダクトのフィーチャーツリーと、未来に実施する、すなわち部門の経営方針に基づき受注を狙うプロダクトのフィーチャーツリーを個別に作り、それらから、以下のフィーチャー、サブフィーチャーを抽出してプラットフォームのフィーチャーツリーを作成した。

✓　複数のプロダクトで完全に共通なフィーチャー
✓　複数のプロダクトで完全に共通ではないが、可変点を定義することにより共通化できるフィーチャー

作成したフィーチャーツリーの一部を図９に示す。

図９　フィーチャーツリー（一部）

さらに、各フィーチャーが特定のドメインに依存するか、依存しないか、またフィーチャーを細分化して実装まで立ち入った場合に、特定のOSに依存するか、依存しないかを分別し、3.1.1節で考察したレイヤ構造に当てはめた。これらにより、まずドメイン非依存レイヤについて、そこに実装するフィーチャーを設計した結果を図10に示す。

図10　プラットフォームのレイヤとフィーチャー構成

最下層に位置するOS依存レイヤには、各プロダクトで共通に利用されるものの、実装がOSのアーキテクチャに依存するものを配した。下表に、OS依存レイヤでの実装フィーチャーを示す。

表２　OS依存レイヤの実装フィーチャー

フィーチャー	説明
マルチスレッド	マルチスレッドは、多くの処理系向けに提供される pthread（POSIX Thread）が提供するスレッド、ミューテックス、条件変数をベースに実装したが、TRON 系 OS では、おのおのを iTRON 準拠のタスク、ミューテックス、イベントフラグで実装した。
プロセス間通信	プロセス間通信の方式として、パイプおよび共有メモリを提供する。
ネットワーク	頻繁に利用される TCP/UDP のソケットサーバ・クライアント等ネットワーク機能を提供する。
ファイル／ログ	ファイル操作、ロギング機能を提供する。
その他	汎用機能で実装が処理系依存であることが多い時刻、スリープ、コンテキストスイッチ関連等の API を提供する。

SPLの概念を当てはめると、本レイヤに実装されるフィーチャーは可変点であり、各OS向けの実装が変異体に相当する。図11に例を示す。表記は、２章で述べた可変性モデルのための図式表記法を用いている。

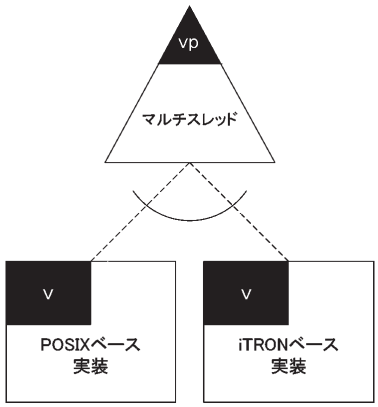

図11　OS依存レイヤの可変性モデル（例）

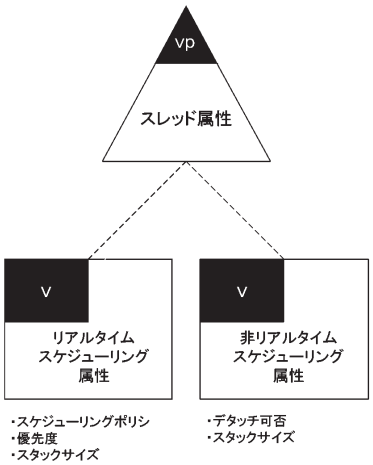

図12　OS依存レイヤの可変性モデル（スレッド属性）

また利用者に提供する機能の観点でも可変点を定義できる。図12に、スレッド属性の可変性モデルを示す。

OS非依存レイヤには、各プロダクトで共通に利用されるフィーチャーのうち、本質的に、もしくはOS依存レイヤの助けを借りることにより、OS非依存に実装できるフィーチャーをそろえた。下表に、OS非依存レイヤの実装フィーチャーを示す。

表３　OS非依存レイヤの実装フィーチャー

フィーチャー	説明
マルチスレッド	OS 依存レイヤの実装を利用することにより、より上位のスレッド間通信、スレッドプール等を実装する。
コンテナ	キューや動的配列、連想配列等を提供する。これらは C++言語では STL（Standard Template Library）や boost 等 OSS のライブラリが存在するが、組込み S/W ではメモリフラグメンテーション防止の観点で利用できないケースがある。そのためプラットフォームに提供させた。データ構造や操作アルゴリズムの実装は、特に組込み開発では開発量の増加や品質低下の原因になりがちなため、プラットフォームが機能提供する意義は大きい。
メモリ管理	上記、メモリフラグメンテーションを防止するメモリ管理機能を提供する。
データ管理	組込み開発では、データ管理をグローバル変数で実装する例が多く、それが S/W の可読性や品質を低下させる原因になる。データのカプセル化と排他制御、トランザクション制御の機能をプラットフォームが提供することにより、それらを改善する。
通信フレームワーク機能	通信 S/W が汎用的に行う外部装置とのメッセージ送受信、および外部からのイベントのハンドリングと振り分け、状態遷移設計に基づくイベント処理の形を規定するフレームワーク機能を提供する。

OS非依存レイヤが提供するフィーチャーの１つである「通信フレームワーク機能」は、本プラットフォームがライブラリではなくフレームワークである所以である。図13に、通信フレームワーク機能の概念を示す。

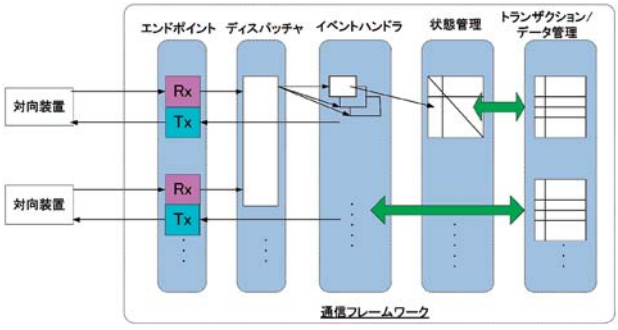

図13　通信フレームワークの概念

通信フレームワーク機能は、イベントドリブンな通信S/Wのアーキテクチャを規定する。本機能を用いた通信S/Wは、他装置との通信端点となるエンドポイント、受信したメッセージを振り分けるディスパッチャ、ディスパッチャが振り分けたイベントを処理するイベントハンドラ、振り分けられたイベントを状態遷移に基づき処理する状態管理で構成される。おのおのについて、プラットフォームが実装を補助する。これにより、プラット

フォームの利用者は、S/Wアーキテクチャを新たに検討する必要がなく、また異なるプロダクトで同じアーキテクチャを採用することにより、S/Wの見通しが良くなり、開発生産性も向上する。

3.1.3　ドメイン依存レイヤの提供フィーチャー

前節では、プラットフォームが提供する機能のうち、ドメイン非依存レイヤに実装する機能を説明した。本節では、その上位に位置するドメイン依存レイヤの提供フィーチャーを述べる。

フィーチャーツリーの分析から、ドメイン固有のフィーチャーは、「組込み機器の制御・監視」と「ネットワーク機能」に大別できることがわかった。前者は組込みS/Wの本質とも言えるが、プロダクト間の差が大きく、共通部位を見出すことは困難に思われた。各プロダクトの機器が搭載しているデバイスはさまざまであり、要求される制御も個別であるためである。

そこで筆者らは、一旦フィーチャーリストから離れ、組込みS/Wの構成要素をオブジェクト指向的に抽象化する検討を行った。その概念を図14に示す。

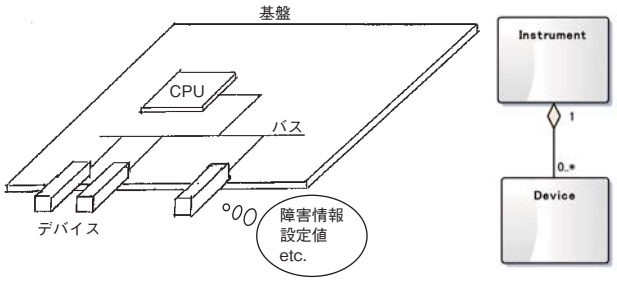

図14　組込みS/Wの構成要素（概念）

組込み機器は、一般に基板上にCPUや各種デバイスを配し、それらをバスで接続して構成する。バスの種別や、CPUとデバイスとの間の通信方式は種々存在するが、それらは3.1.1⑵項で述べたように、デバイスドライバやミドルウェアが隠蔽する（もしくは、そのような隠蔽層がなければ、それらを隠蔽するデバイスドライバやミドルウェアを自作する）。CPUはプラットフォーム自身を駆動させるものであるから除外すると、図14右のように、Instrument（機器）クラスが、Device（デバイス）クラスを任意の数分集約しているとモデル化できる。

次にデバイスクラスの属性と操作を考える。デバイスは、名前や製品ID、F/Wバージョンのほか、各種設定値（properties）をもつ。組込みS/Wの重要な責務のひとつに、このようなデバイスの属性を取得・設定することがある。また利用を開始するために、給電したりリセット解除したりする場合がある。これらが操作となる。設定値は個別に識別できる必要があるが、その数も、型もプロダクト固有である。そのため、デバイスクラスの属性は、ドメイン非依存レイヤが提供する連想配列（Map）型で表現することが望ましい。デバイスに設定（Set）するタイミングと、実際にH/Wに反映（Apply）するタイミングは異なるかもしれないし、反映する方法も設定値ごとに異なるはずである。

デバイスは、各種情報（データ）を入力したり出力したりするために存在する。そのためデバイスへのデータの入出力操作も必要である。しかし、これに関してはデバイスごとに固有のため、まずはプラットフォームのスコープ外とした。一方、デバイスの障害検知は組込みS/Wに普遍的な機能である。情報と同様に、デバイスは何らかの障害情報を保持する。これもデバイスの属性である。ただし、それを外部に伝える方法は２パタンある。１つは外部から取得するパタン、もう１つはデバイスが外部に通知するパタンである。ただし設定値同様、障害の種類や型はデバイスにより異なる。これらを考慮した、プラットフォームがサポートするクラス構造を、図15に示す。

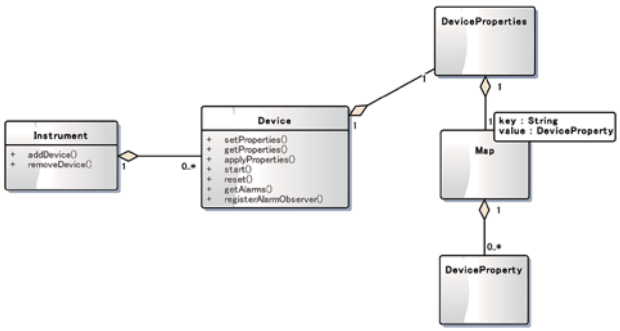

図15　ドメイン依存レイヤのクラス図
（デバイス制御・一部）

利用者は、これら各クラスと操作を可変点としてプラットフォームを利用することになる。

ドメイン固有のフィーチャーのうち、もうひとつの「ネットワーク機能」については、個別にオブジェクトとその属性・操作を定義していくことができる。表４に、筆者らが抽出したネットワーク機能に関連するフィーチャーの一部を一覧する。

表４　ネットワーク機能のフィーチャー（一部）

ネットワークインタフェース	MAC アドレス
IP アドレス	サブネットマスク
ゲートウェイ	NTP クライアント
HTTP クライアント	SNMP エージェント
統計	ping 送信

なお、ドメイン依存レイヤについては今年度以降の開発スコープとしており、現時点では上記のような概念設計にとどまっている。本稿執筆以降、具体化していく計画である。またドメイン依存レイヤには、将来的には「組込み機器の制御・監視」、「ネットワーク機能」に並ぶ第三、第四のブロックが導入されていくものと予想する。これらを総合して、プラットフォームの構成を図16にまとめる。

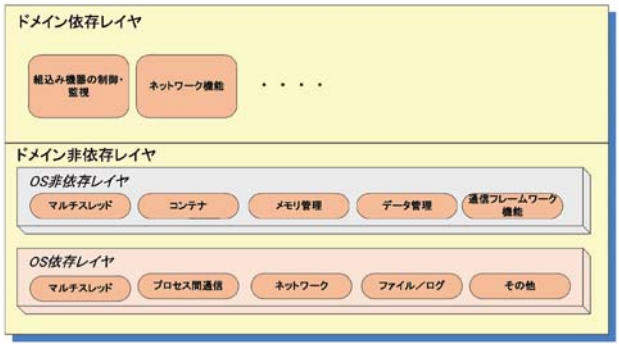

図16　プラットフォームのレイヤとフィーチャー構成（完成版）

3.1.4　プラットフォームの実装言語

3.1.1(1)で述べたように、プラットフォームをオブジェクト指向フレームワークとして実現するのは良いアイデアである。筆者らは、その実装言語としてC++言語を選択した。本プラットフォームは通信分野のほか、組込み分野も指向している。JAVA、C#、Rubyと言ったオブジェクト指向言語を用いた組込み開発の事例も一般にはあるが、我々の現場では、長年C言語もしくはC++言語（そして一部アセンブリ言語、シェルスクリプト）が用いられてきた。今後もこのトレンドが急激に転換することはないと判断した。

ただし、C++言語はパフォーマンス、実行モジュールのサイズ、組込み向けコンパイラの対応の面で、C言語に劣る面がある。C言語でもオブジェクト指向プログラミングは可能であるが、近年では組込み向け開発環境でも、（言語仕様に制約がかかる場合はあるが）C++言語を選択できることが一般的になっている。C++言語を採用するデメリットとメリット（開発生産性、保守性）とのトレードオフを勘案し、C++言語を採用した。

3.2　SPLプロセス

前節で、通信・組込みS/Wを開発する筆者らの現場に適用可能なプラットフォームの設計を述べた。本節では、SPLのもうひとつの重要な側面である、開発プロセスについて述べる。

3.2.1　プラットフォームの開発プロセスと成果物

プラットフォームを用いたSPLの導入には、QCD（品質／コスト／納期）の改善が期待される。プラットフォームの再利用性が高まるほど、同じ機能の製品を開発するための開発規模が減るため、作り込む不具合の削減（Q）、開発工数・工期の削減（CD）への効果は高まる。しかしそのためには、プラットフォームの品質が担保されているという前提が必要になる。そこで筆者らは、プラットフォームの品質を担保するために必要な成果物と開発プロセスを検討した。

その第一歩として、まずプラットフォームを利用する側、すなわちプラットフォームを用いた製品開発のプロセスと成果物がどうあるべきかを考えた。図17に、その概念を示す。

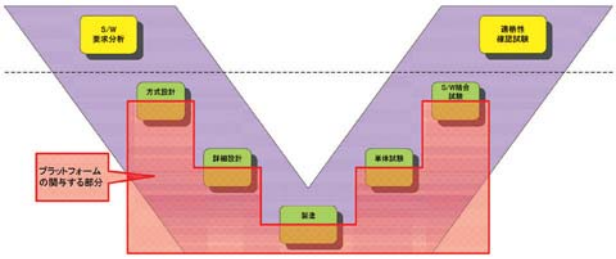

図17　プラットフォームを用いた製品開発のＶ字モデル

S/W要求分析では、製品の利用者が受けることのできるサービス、すなわちユースケースを定義する。このフェーズでは、製品がプラットフォームを用いて実現されるか、そうでないかを意識しない。利用者（アクタ）にとって、利用する製品の実現方法は興味の対象外だからである。そこで、まずS/W要求分析フェーズでは、プラットフォームは関与しないこととした。一方、方式設計は実現方法の検討となるため、プラットフォームを用いるかどうかは最大の関心事となる。そこでこのフェーズから、対応する試験フェーズであるS/W結合試験までをプラットフォームの関与するフェーズとした。

SPLでは、プラットフォームの開発プロセスは「製品管理」「ドメイン要求分析」「ドメイン設計」「ドメイン実現」「ドメイン試験」のサブプロセスからなる。このうち「製品管理」は、系列開発する製品のロードマップを示すサブプロセスである。本稿では、当サブプロセスは通常のS/W開発におけるシステム設計や製品企画に相当するものと考え、Ｖ字モデルの外に存在するものとした。そのため以降の検討では除外した。

以上を踏まえ、検討したプラットフォームの開発プロセスと成果物を、表５に示す。

表５　プラットフォームの開発プロセスと成果物

ドメイン開発サブプロセス	S/W開発フェーズ	成果物
ドメイン要求分析	プラットフォーム方式設計	プラットフォーム方式設計書 プラットフォーム結合試験仕様書
ドメイン設計		
ドメイン実現	プラットフォーム詳細設計	プラットフォーム詳細設計書 プラットフォーム単体試験仕様書 （テストコード）
	プラットフォーム製造	プラットフォームソースコード
	プラットフォーム単体試験	プラットフォーム単体試験成績書 （自動テスト環境）
ドメイン試験	プラットフォーム結合試験	プラットフォーム結合試験成績書

ドメイン要求分析では、共通フィーチャーを抽出し、可変点と（あれば）変異体を定義する。これは、3.1.2節で示したフィーチャーツリーや可変点の決定が該当する。ドメイン設計では、これを実現する方式とインタフェースを規定する。我々のプラットフォームは、3.1.4節で述べたようにC++言語を用いたオブジェクト指向フレームワークとして実現することから、その中にどのようなクラスが含まれ、どこが可変点で、どのように変異体を作成するのか（クラス継承なのか、パラメータによる制御なのか、等）をクラスの外部インタフェース仕様として定義すれば、この両サブプロセスの目的を達成する。それらを定義した「プラットフォーム方式設計書」を成果物とし、これを作成する工程を、S/W開発における方式設計フェーズに当てはめた。フィーチャーツリーの作成と共通フィーチャーの分析も本フェーズに含めたが、次工程への直接のインプットとはならないため、プラットフォーム方式設計書の付録と位置付けた。

ドメイン実現には、S/W開発フェーズにおける詳細設計、製造、単体試験を当てはめた。ここでは、方式設計にて規定したクラスの内部仕様を設計し、ソースコードやテストコードとして実現し、自動試験できる環境を成果物とした。ここは、一般のS/W開発と変わる点はない。

ドメイン試験も、同様に一般のS/W開発におけるS/W結合試験と同じである。我々のプラットフォームでは、クラス組合せ試験と定義した。

3.2.2　プラットフォームを用いた製品開発プロセスと成果物

前節ではプラットフォームの開発プロセスと成果物を述べたが、本節では、プラットフォームを用いた製品開発のプロセスと成果物について言及する。これは、前節の図17で、赤くハッチングした部分の考察である。表６に、製品開発の方式設計～S/W結合試験までの各フェーズの成果物と、それらに対するプラットフォームの関与を示す。

ポイントは、方式設計書である。方式設計書では、製品がプラットフォームのどの機能をどう利用し、プラットフォームに定義された可変点をどう固定するか、すなわち変異体をどう実現するかを規定する。変異体の実現方法は、２章で述べたようにいくつかの実現方法があり、可変点により異なる。例えばクラス継承であったり、コンパイルスイッチ、外部からのパラメータであったりする。これらに対応して、製品の方式設計では、継承するサブクラスやコンパイルスイッチの選択、プラットフォームに与えるパラメータの値を規定する。

表６　製品開発プロセスの成果物とプラットフォームの関与（方式設計～S/W結合試験）

S/W開発フェーズ	成果物	プラットフォームの関与
方式設計	方式設計書	・利用するプラットフォーム機能の選択 ・可変点の固定（変異体）
	S/W結合試験仕様書	・変異体を含めた結合動作の妥当性を確認する試験仕様
詳細設計	詳細設計書	・可変点の固定に関する詳細実現
	単体試験仕様書	・変異体の動作の妥当性を確認する試験仕様
製造	ソースコード	・固定した変異体
単体試験	単体試験成績書	・変異体の動作の妥当性確認
S/W結合試験	S/W結合試験成績書	・変異体を含めた結合動作の妥当性確認

その他の成果物は、全て方式設計の成果物から派生する。詳細設計書では、方式設計で規定した可変点の固定方法とトレースをとり、例えばサブクラスの詳細仕様を規定する。製造ではそれをソースコードとして実現し、単体試験で動作を確認する。トレーサビリティは製品品質を確保する上で重要なファクターであり、それはプラットフォームを用いた製品開発でも、一般の製品開発と変わらない。ただし、表現されるものは、製品固有の部分を除けば、あくまで変異体部分のみである。プラットフォームが実現する部分は3.2.1節で述べたプラットフォームそのものの開発プロセスで品質担保されているため、この部分は完全流用として扱ってよい。これにより製品開発の規模が低減され、ひいては作り込む不具合の削減、開発工数・工期の削減というQCDへの効果として現れる。

3.2.3　プラットフォームとプロダクトの構成管理

本節ではSPLにおける構成管理について述べる。SPLにおける構成管理で用いる技法は単一製品開発のものと同一である［３］。ただし、SPLではプロダクト毎に変異体が発生するため、この点を考慮したうえで、構成管理を整備する。

一般的な開発での構成管理パタン［４］では、開発ラインとなる主ライン（Main Line）と、リリース済みラインを管理するためのリリースブランチ（Release Branch, RB）を用意する。これは、開発の成果は主ラインに取り込み、リリースした製品に発生した改修はリリースブランチに取り込むというものである。SPL で

もこの構成管理手法がベースとなるが、課題となるのはリリースブランチの粒度である。

プラットフォームは、変異体として固定されることでプロダクトに取り込まれる。そこで、この変異体の固定をリリース単位としたブランチ構成が考えられる。ただし、この構成では、変異体毎にコードラインがあるため、コードライン間のメンテンスに問題がある。図18に例を挙げて説明する。

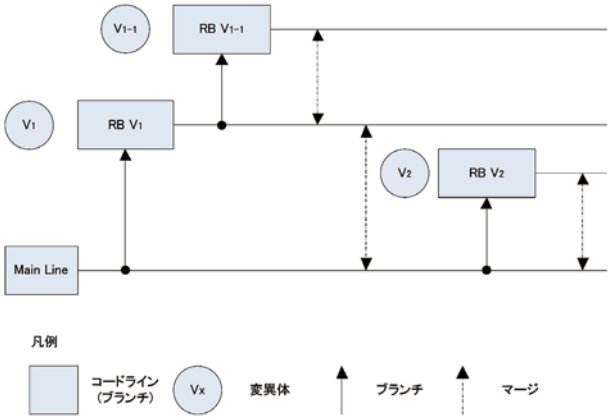

図18　変異体をリリースブランチとした場合

変異体（V_1, V_{1-1}, V_2）は製品毎に発生するため、それに応じてリリースブランチ（RB V_1, RB V_{1-1}, RB V_2）の数が増える。リリースブランチは、元は主ライン（Main Line）からコピーされたものである。このため、変異体の固定箇所以外の殆どの部分はコピーされたものとなる。ひとたびリリースブランチ上で、可変点の固定以外の変更が発生すれば、単一製品開発同様、同じ変更を複数のブランチに対して反映する作業（マージ）が発生する。マージは単一製品であっても困難な作業であるが、SPL の場合は、複数の製品にまたがることになるので、一層困難なものになる。例えば、RB V_1 の系列とは別の変異体V_2 のブランチ RB V_2にて変更が発生したとする。この変更をMain Lineに取り込めたとしても、RB V_1 およびRB V_{1-1} に取り込む必要があるかを検討しなければならない。

リリースブランチのもう一つのアプローチとして、プラットフォームのリリースの単位をリリースブランチとする方法がある。この方法では、プラットフォームの構成管理と、プロダクトの構成管理とを分けて管理する。プラットフォームの構成管理からプラットフォームのリリースブランチを提供し、これをプロダクト間で共有する。プロダクトの構成管理では、プロダクト毎に、可変点を固定することで発生する変異体を管理する。この構成管理の全体像を、図19に示す。

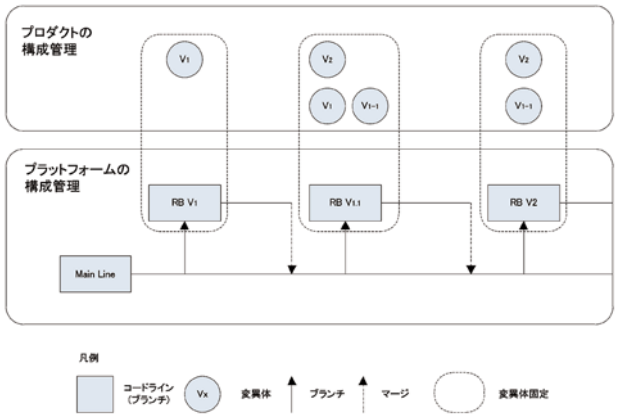

図19　変異体をリリースブランチとした場合

プラットフォームの構成管理は、単一開発同様である。リリースブランチは、プラットフォームの開発計画に基づいて、リリース時生成する。各プロダクトは、このリリースブランチを自身のリポジトリに取り込み、このリリースブランチに含まれる可変点の固定のみを行う。不具合や仕様変更の緊急対応が必要な場合は、改修をリリースブランチ上で行い、プロダクト側の構成管理上では行わない。

プラットフォームのリリースをリリースブランチとした場合は、各プロダクトでは変異体のみが発生し、可変点はプロダクト間で共有されるため、SPLに即した管理であると考える。但し、プラットフォームの開発計画と各プロダクトの開発計画の足並みをそろえ、タイムリーにプラットフォームをリリースする必要がある。

4.　むすび

本稿では、筆者らのソフトウェア開発現場にSPLを導入するために、「ソフトウェアフレームワーク」に着目し、それにSPLの考え方を適用することで、必要なプラットフォームの設計、およびそれを利用した開発プロセスへの考察と実践内容を述べた。プラットフォームの設計では、ドメイン依存レイヤと非依存レイヤに分離することにより、SPLを多様な系列製品開発へ適合させるとともに、レイヤを意識したフィーチャーモデル分析で、筆者らの部門に有効なドメイン要求分析の結果を示した。SPL開発プロセスの考察では、プラットフォームを完全流用部として扱うための開発プロセスの考え方を示し、かつアプリケーション開発を円滑に進める構成管理の手法を述べた。

プラットフォームに対しては、今後はドメイン依存レイヤの実装および拡充を行う。開発プロセスに対しては、成果物のより詳細な定義、実プロジェクトにおける構成管理の実践と効果の測定を行う計画である。

参考文献

［1］K.ポール, G.ベックレ, F.リンデン（2009）：ソフトウェアプロダクトラインエンジニアリング, 株式会社エスアイビー・アクセス

［2］井上 久男（2012）：日産の設計革命、脱プラットフォーム共有化戦略, 日経テクノロジーonline, 2012/6/25

［3］Danilo Beuche（2010）：What's the difference? A Closer Look at Configuration Management for Product Lines（http://productlines.wordpress.com/）, 2010/3/13

［4］スディーブ P バーチャック, ブラッド・アップルトン（2006）：パターンによるソフトウェア構成管理, 株式会社 翔泳社

執筆者紹介

赤坂　賢洋

1995年入社。研究所システム、防衛・通信・医療／バイオ関連システムを担当後、2007年からは情報通信システム、組込みシステムの開発に携わる。

原　周

2007 年入社。入社以来、情報通信システム、組込みシステムの開発に携わる。