

XDDP と SPLE の 連携・移行・使い分け ガイドの紹介

2014-10-16

派生開発推進協議会 T14研究会

安倍 信孝、梶本 和博、北崎 敦、
桜庭 恒一郎、林 好一、藤田 将志

派生開発推進協議会

T-14研究会（SPL と XDDP の連携）

派生開発推進協議会

- 略称: AFFORDD
- 今日の開発の殆どを占める派生開発が効果的に行なわれる技術の開発や普及、更にはそれによって得られた「余裕」により、新たな技術の獲得や革新が進む状態を作れるよう後押しすることを目的に設立された非営利団体
- <http://xddp.jp>

T14研究会

- 愛称: SIG-XAS
- ソフトウェアプロダクトライン開発（SPLE または SPL）と XDDP の関連を明らかにし、相互に利用可能な技術・プロセスを見つけるための研究活動を行っている
 - SPLE: 共通資産を基に複数のシステム効率的に開発するための考え方
 - XDDP: 派生開発を効率的に行なうための方法論
- JASPIC プロダクトライン分科会との合同活動もあり

— 目 次 —

1. ガイド作成の背景
 2. ガイドの内容紹介
 3. 移行
 4. 使い分け
 5. 融合
 6. まとめ
- 付録(ガイドの他の部分を一部紹介)

1. ガイド作成の背景

1. ガイド作成の背景

- XDDP (eXtreme Derivative Development Process) は単発の派生開発に有効な手法であり改造時の品質向上が期待できる。
- 一方、ソフトウェアプロダクトライン開発 (SPLE) は、製品系列で長期間に渡って派生を繰り返す場合の効率向上を図る目的で導入する組織が増えてきている。
- T-14研究会では、XDDP と SPLE のそれぞれが適している条件の検討や XDDP から SPLE へ移行する際の留意点などを議論してきた。
- 議論を進める中で、課題や対応策などを広く参考にしてもらうべく「XDDP から SPLE への移行ガイド」を作成することとした。
- 移行ガイドを作成するに当たって、移行の出発点を想定するため、派生開発の状況を調べるアンケートを実施した。
- アンケート結果から、XDDP から SPLE への移行だけでなく、XDDP と SPLE の住み分けや融合も有効そうな場合のあることがわかったため、整理を行なった。

2. ガイドの内容紹介

名称: XDDP と SPLE の連携・移行・使い分けガイド(仮称)

■ 当ガイドの背景

■ 当ガイドの使い方

■ 想定読者

- 技術者、プロセス支援者(SEPG, SQA, etc)、技術のわかるマネジャ
- XDDP はできていること
- SPLE に関する基本的な知識があること

■ XDDP と SPLE

- それぞれの概要
- それらの違い、それぞれの特徴
- 導入/移行のシナリオ

■ XDDP・SPLE の導入/移行/使い分け/融合のシナリオ

■ プラクティス(仮題)

- フィーチャ分析とその成果の扱い
- 既存製品の分析
- リファクタリング

■ 移行、使い分け、融合

- それぞれの違い
 - 移行、融合の際に必須のこととそうでないこと
 - プラクティスの適用
 - 移行
 - 使い分け
 - 融合
- ※プラクティスの組合せパターンか、それが難しければ組合せ例(できれば)

■ 結言

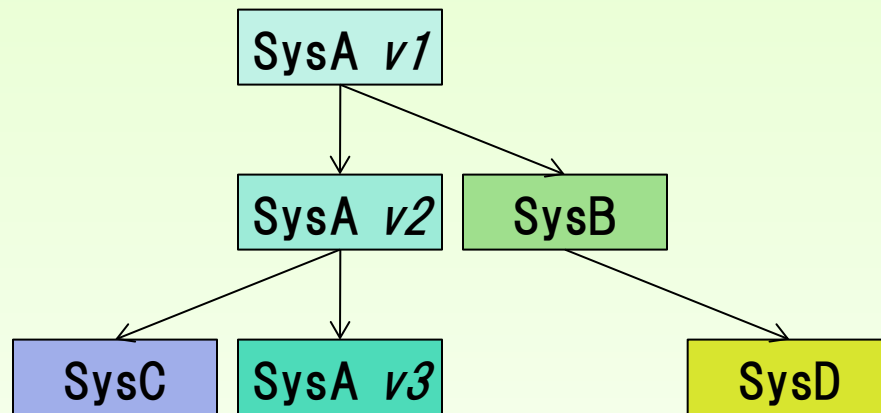
■ 付録

- (アンケート集計結果、関連資料、等)

※MS PowerPoint を用い、スライドに主に図を描き、ノートに解説を書く。

どちらも再利用による効率向上が狙い

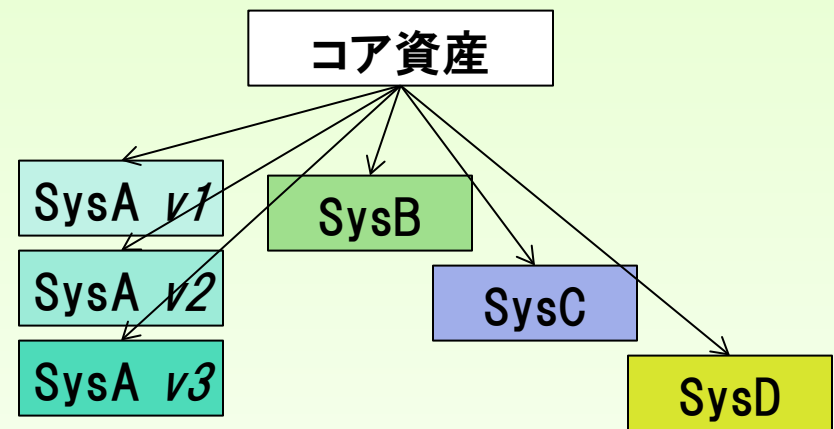
XDDP(派生開発)の典型例



□ 過去のシステムを基に再利用の形を検討する

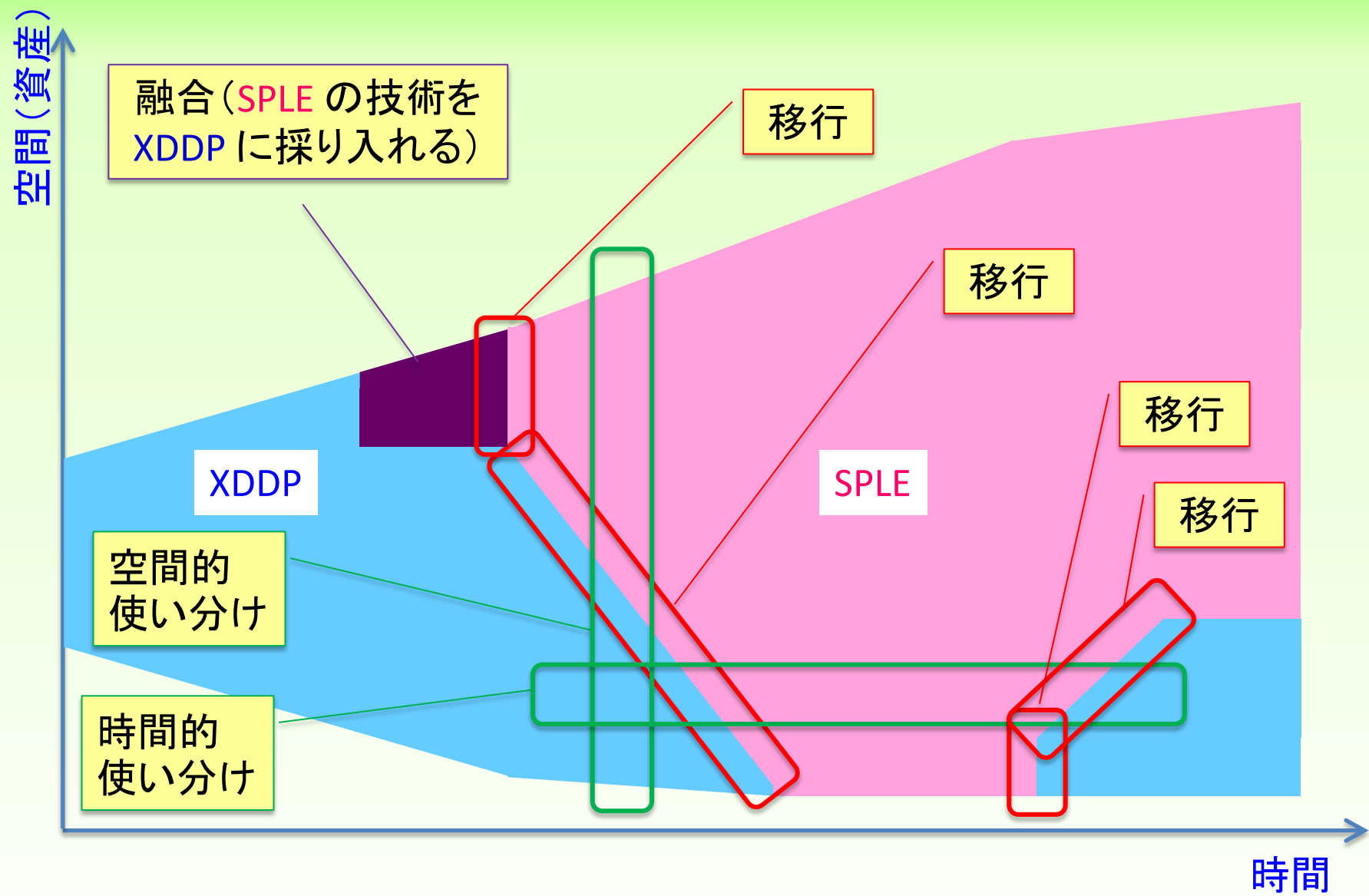
- 作られているものを再利用する

SPLE の基本形



□ 将来のシステムを基に再利用の形を検討する

- 再利用するものをコア資産として用意する



3. 移行

- 同一システム/製品のシリーズに対して、XDDP (SPLE) のプロセスで開発していたのを、SPLE (XDDP) のプロセスでの開発に切り替える

■ XDDP → SPLE または
SPLE → XDDP

- 形態には

- 全面的な切替
- 部分的な切替

があり、後者では次に説明する「使い分け」が生じる

- 移行が向いている場合

XDDP → SPLE:

- 複数システムの並行開発で類似の改造が必要になることが多い
- 共通性の高い複数のシステムを、末端の機能の抜き差しで作る

SPLE → XDDP:

- SPLE による開発費低減分をコア資産保守・更新費が上回る
- ...

■ソフトウェアプロダクトライン(SPLE):成熟期に適した開発方法

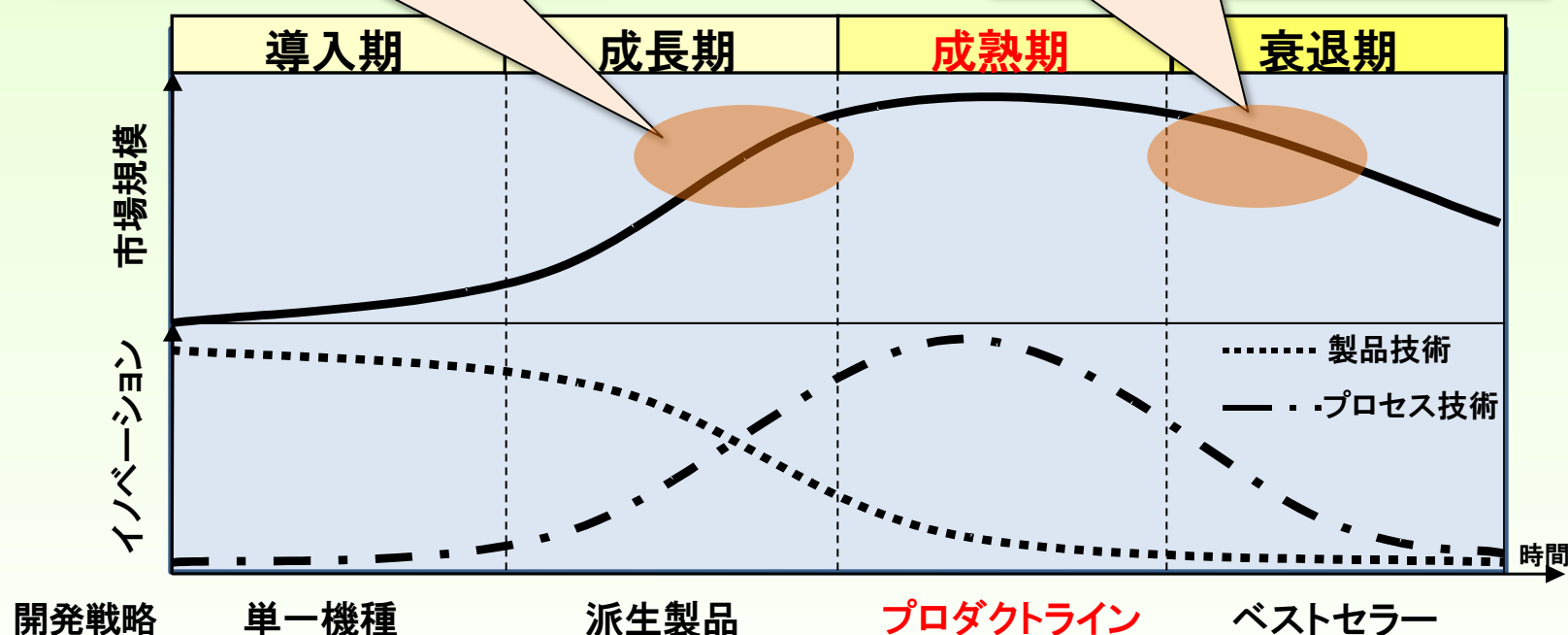
⇒導入期→成長期→成熟期→衰退期と移行する「製品ライフサイクル」に対応した移行シナリオを想定する

■移行シナリオ1:

XDDP で派生開発をしている組織が、SPLE に切替える

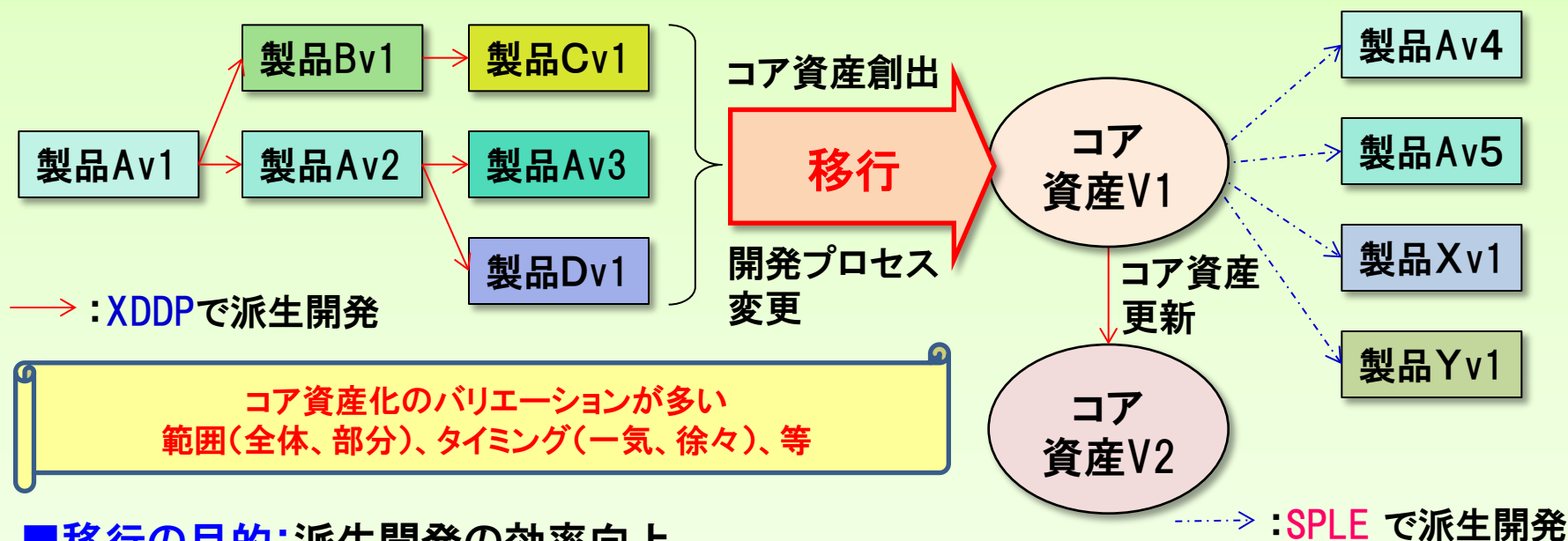
■移行シナリオ2:

SPLE で派生開発をしている組織が、XDDP に切替える



参考文献: 吉村, 菊野, 組込みシステムにおけるソフトウェアプロダクトラインの導入, 情報処理, Vol. 50, No. 4, pp.295-302, 2009.

■移行シナリオ1:XDDP で派生開発をしている組織が、開発方法を SPLE に切替える



■移行の目的: 派生開発の効率向上

■背景: 製品系列のバリエーション、派生開発件数の増加に伴う開発コスト増大

■課題: ①既存資産を基に SPLE コア資産の開発方法検討

⇒現状資産の状態や組織等の状況に適した SPLE の開発方法を採用する

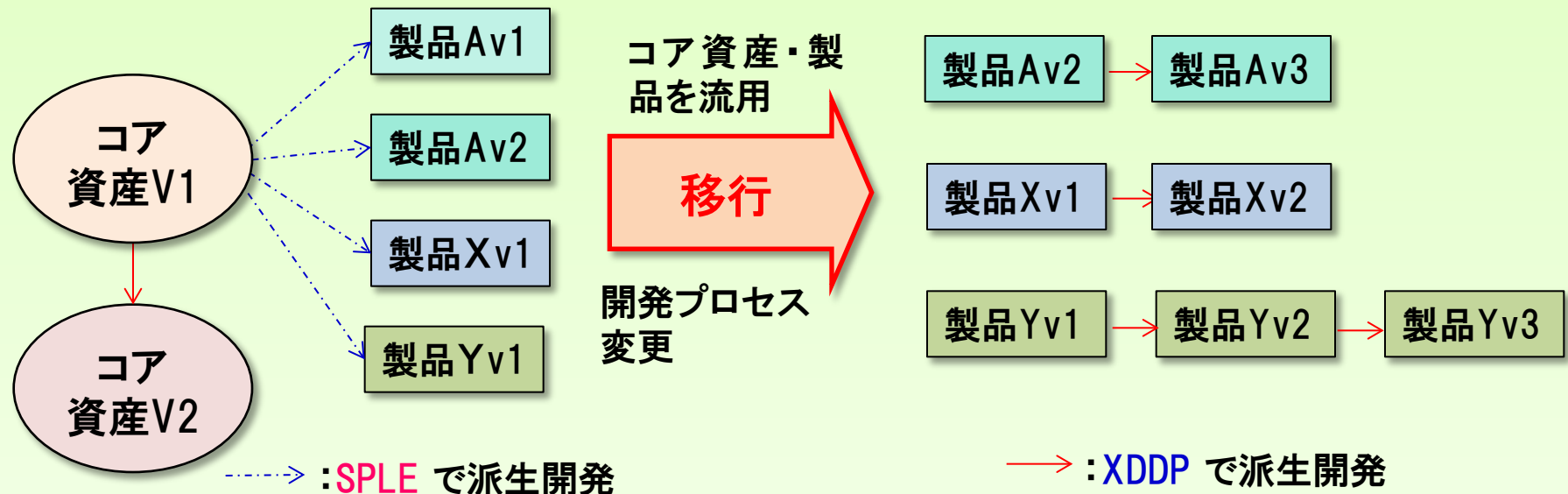
②コア資産の可変性を実現するためのアーキテクチャ検討

③移行の際に XDDP の成果物を活用する方法の確立

⇒過去の XDDP 成果物(変更要求トレーサビリティマトリクス)をコア資産化の際の検討材料として使用する、等

3.4 移行シナリオ2: SPLE から XDDP へ

■移行シナリオ2:SPLE で派生開発をしている組織が、開発方法を XDDP に切替える



■移行の目的:開発効率の維持 ←コア資産維持工数の削減

■背景:製品系列寿命の衰退期への遷移に伴う派生開発件数の減少

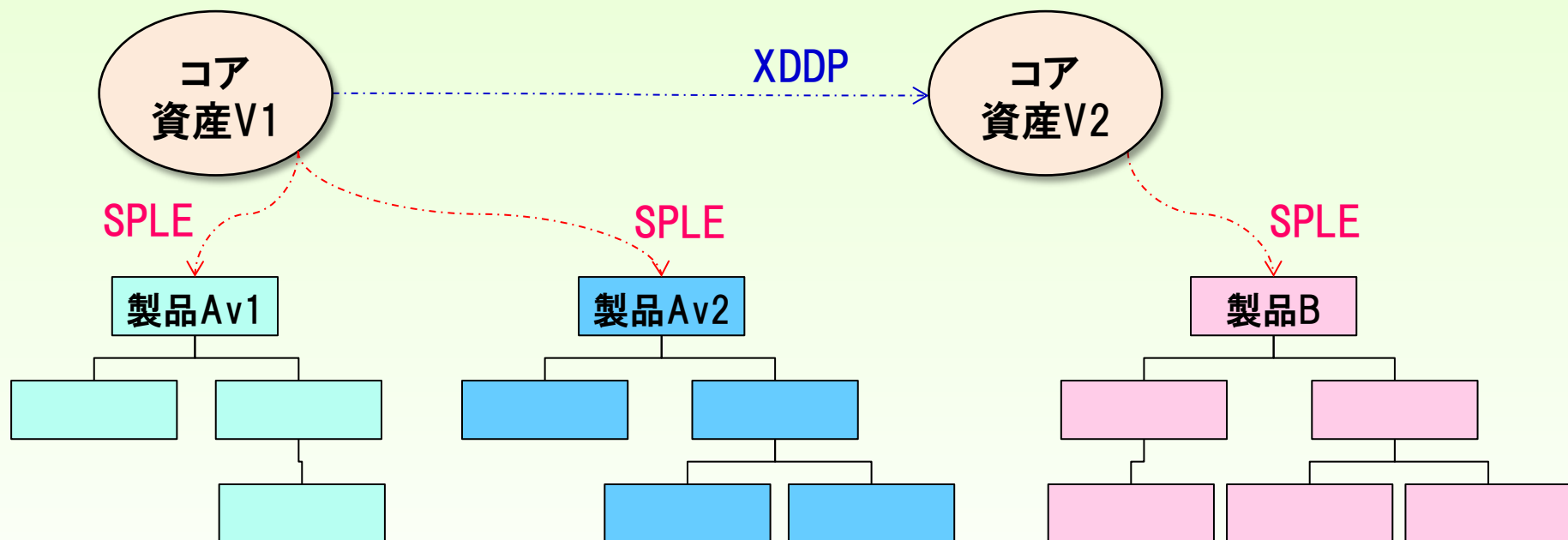
■課題: ①SPLE コア資産の運用方法改訂(例:コア資産管理をコア資産管理チームから製品開発チームに移管してしまう)

②SPLE 開発で納入した製品の保守方法確立(担当組織、改訂プロセス、等)

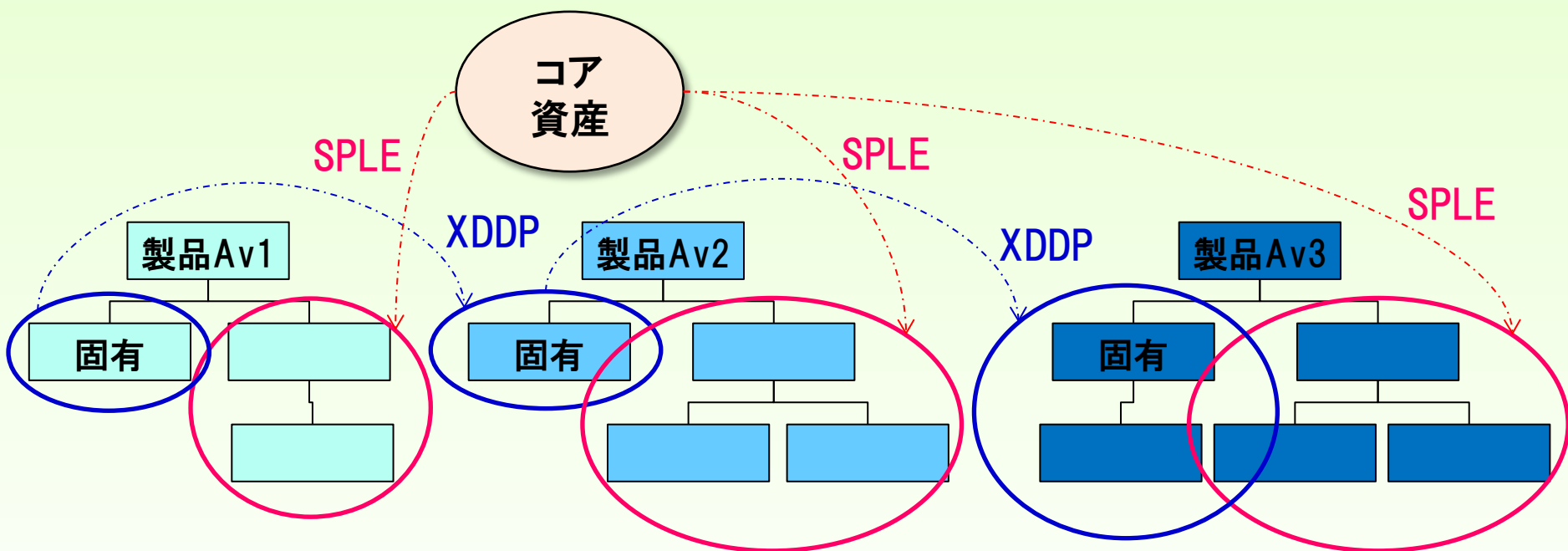
4. 使い分け

- 同じシステム/製品の中で、
XDDP と SPLE を別々の部分
に適用する
 - 「移行」においてこのパターンが
現われることがある
- 極端な例として、どちらか一方
しか適用しない場合もある
 - 途中で条件が変わり、「移行」を
実施することはない
- 使い分けが向いている場合の
例
 - 既に関済した部分の改造には
XDDP を、新規開発/企画部分
には SPLE を適用
 - 特定システム固有の部分があ
り、システム間共通のコア資産
にする意味が薄い
 - 変更が予想されない、あるいは
あっても頻度が低く、SPLE によ
る投資回収が相当先になる部
分には XDDP を、変更の頻度
や同じ機能の変種が多い部分
には SPLE を適用
 - ...

- 各製品の開発には **SPLE** を、コア資産の進化には **XDDP** を適用する
 - この場合の **XDDP** は「フルの」**XDDP** ではない。文書が揃っているのでスペックアウトは行なわないであろうし、単なる追加であれば **XDDP** が特に必要という訳ではないので
 - だが、ならうべき **XDDP** のプラクティスはいくつもある
 - 例: **XDDP** の「ベースの仕様書の更新はリリース後に行なう」は **SPLE** では「今回の製品開発に基づいてコア資産を更新するのは製品リリース後に行なう」



- そのシステム/機種に固有の(他のシステム/機種と共有しない)部分には **XDDP** を適用し、他の部分に **SPLE** を適用する
 - 同一製品のバージョン間
 - 同じ PL の中の、機種固有部が似ている製品間
- 逆に、**XDDP**で作った複数機種の中の似通った部分を **SPLE** で作るべきだとなった場合には、「移行」の一種として対処する



5. 融合

- 基本的には **XDDP** を実施するが、**SPLE** に使う技術を応用して効率・品質のさらなる向上を図る
- **SPLE** に使う技術 = 可変性を扱う技術
 - 可変性モデリング(フィーチャモデリング)
 - 可変性モデルと資産の関連付け
 - 可変性の実現(サブクラス化、設定ファイル、モジュールの入れ換え、等)
- 融合が向いている場合の例
 - 類似システム/サブシステム間の共通性は把握しているが、可変性の詳細は設計後に初めて明らかになる
 - フィーチャモデルで共通性を整理しておき、可変フィーチャを適宜追加していく
 - フィーチャモデルの、結果としての可変フィーチャとそれに対応する資産を関連付けておき、将来の保守・再利用に備える

* フィーチャ: システムの提供するサービス、機能、特性等、そのシステムを表現する際に用いる概念。例: DVDレコーダにおける再生、録画、録画予約、おまかせ録画、起動時間、最大録画可能時間、使用可能メディア等。

■ **SPLE** に使う可変性モデルと各種成果物の可変点との間の関連付けは、可変点(どこが変わりうるのか)に関する間接的なトレーサビリティを示すことになる

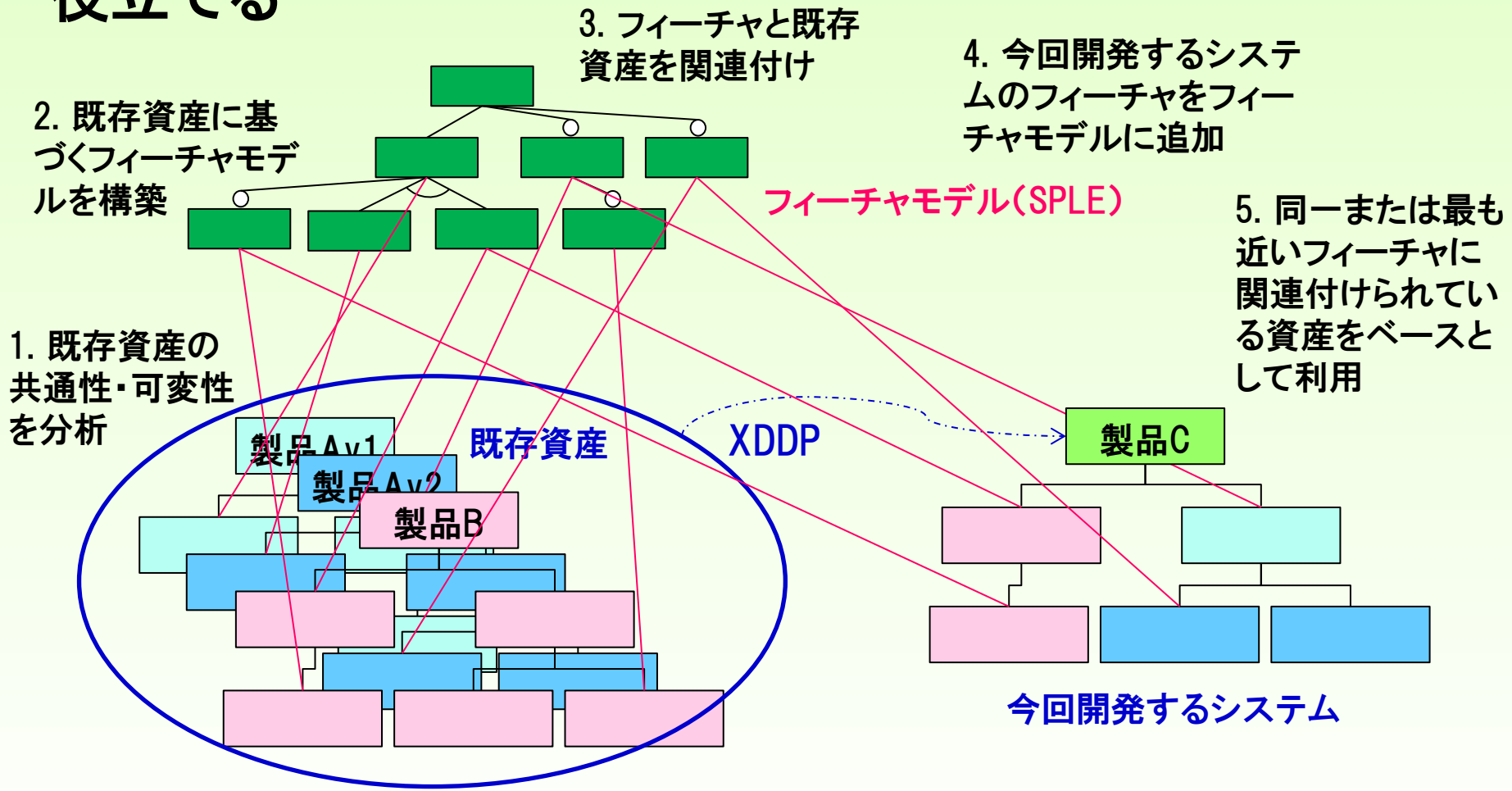
■ 成果物: 要求仕様、アーキクチャ、詳細設計、実装、テスト仕様・設計、開発プロセス、等

■ 可変性モデル: 有向グラフ形式のフィーチャモデルが一般的

※ モデルではないが、システムごとにフィーチャと要求/設計要素/コンポーネントの関係をマトリクスで表現した「星取表」もよく用いられる

■ **XDDP** では可変点は明示的には扱わないが、それぞれの開発で扱ったフィーチャをフィーチャモデルに組み上げ、開発間で共通であった点、異なった点を可変性モデルとして整理することによって、次回以降の開発のベースとして最も適切な過去資産(要求仕様の一部、アーキテクチャの一部、コードの一部、テストケースの一部、他)を見つけるのに役立てることができる

■既存資産の共通性・可変性をフィーチャモデルを用いて整理し、次回以降の XDDP 開発のベースを決める際に役立てる

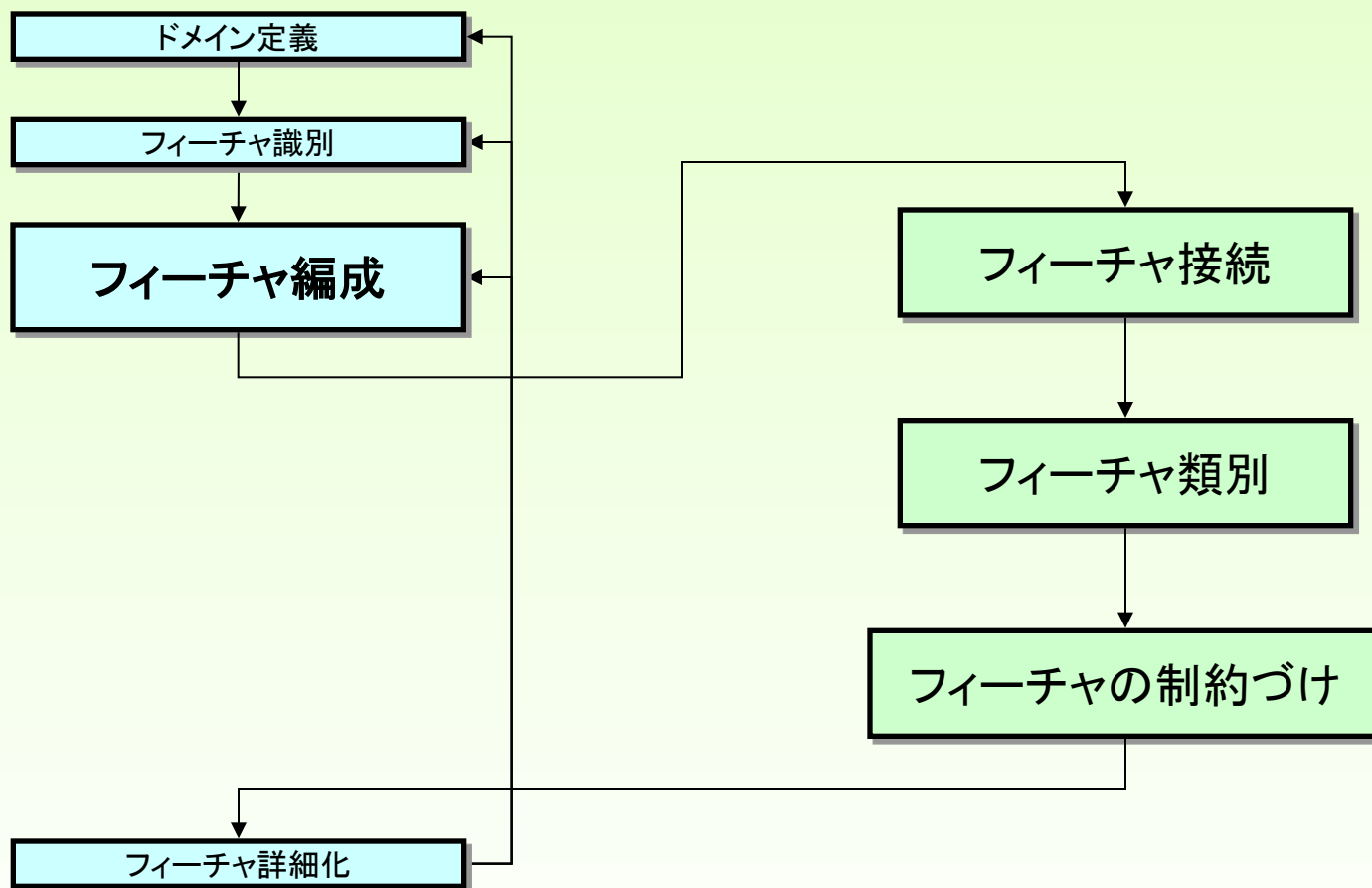


6. まとめ

- XDDP と SPLE は排他的な存在ではなく、適材適所の考え方で適用するもの
 - 時間的な「使い分け」と空間的な「使い分け」(同時並行)がある
 - 時間的な使い分けには「移行」が伴う
 - 双方のいいとこどりをした「融合」型の適用もある
- ガイドでは XDDP と SPLE に用いるプラクティスも解説する
- ガイドは11月をめどに公開予定
 - www.xddp.jp からたどれるところに掲載

付録

(ガイドの他の部分を一部紹介)



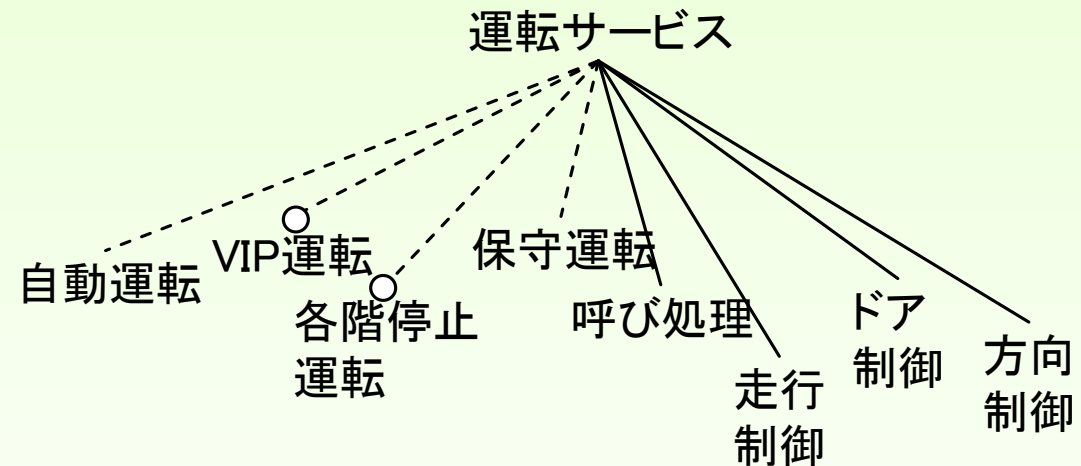
■ フィーチャ接続では、似たようなフィーチャ同士に注目し、それらを一つで概念で表わしてみる

■ フィーチャ編成では、まず同じ概念を指す複数のフィーチャをまとめることから始める

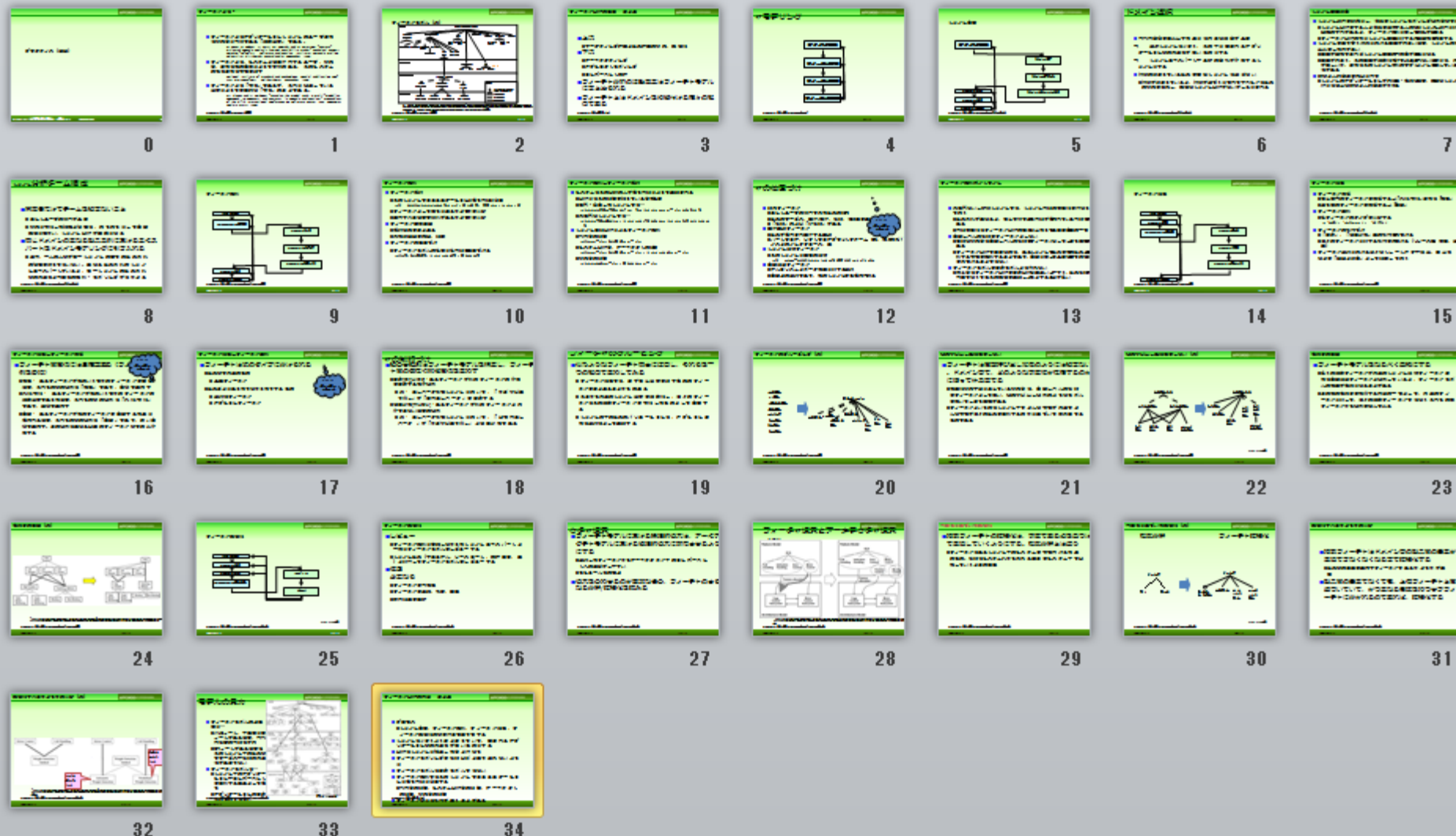
■ そこから共通のドメイン概念を抽象化し、個々のフィーチャはその抽象フィーチャを特化したものとして定義する

■ ドメインの中の製品のバリエーションは、オプションまたは選択肢として表現する

自動運転	呼び処理
VIP運転	走行制御
各階停止運転	ドア制御
保守運転	方向制御



-----: 汎化関係
——: 構成関係



■ 既存製品群の資産を活用してコア資産を開発する際の分析を行う

＜既存製品群資産＞

既存製品仕様

＜活用法＞

フィーチャマトリクス作成時の
フィーチャ、製品レパートリを導出

フィーチャマトリクスを作成し
コア資産候補を抽出

開発技術
ノウハウ

フィーチャと対応するソース
プログラムとの対応表を作成

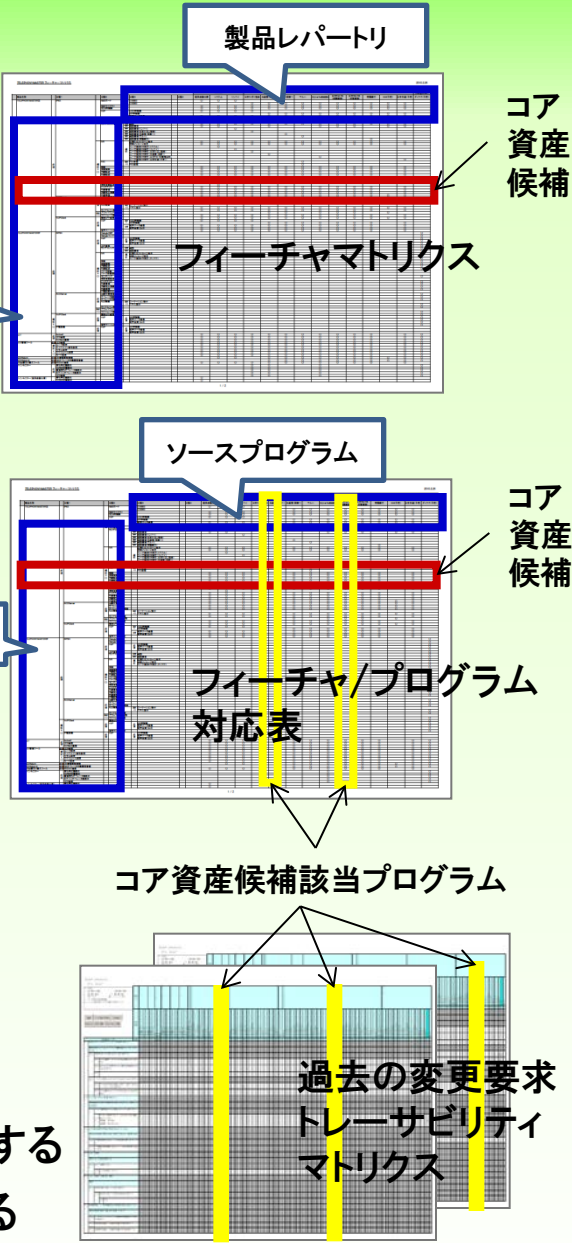
コア資産候補のフィーチャの
過去変更履歴確認時に参照

過去のXDDP変更
要求トレーサビリティ
マトリクス

コア資産候補のフィーチャに
該当するソースプログラムの
過去における変更頻度を調査

＜変更頻度が多い場合＞

- ⇒① **変更**に強い構造にしてコア資産化する
- ② 選択可能なコア資産として開発する

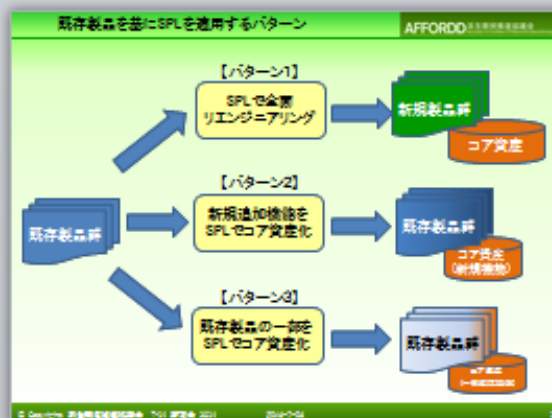


既存製品の分析

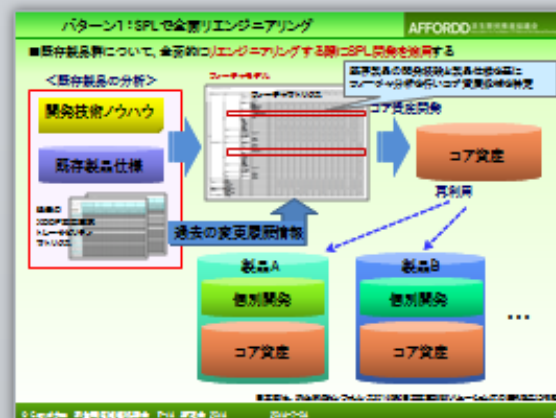
2014-09-17

© Copyrights 派生開発推進協議会 T-14 研究会 2014

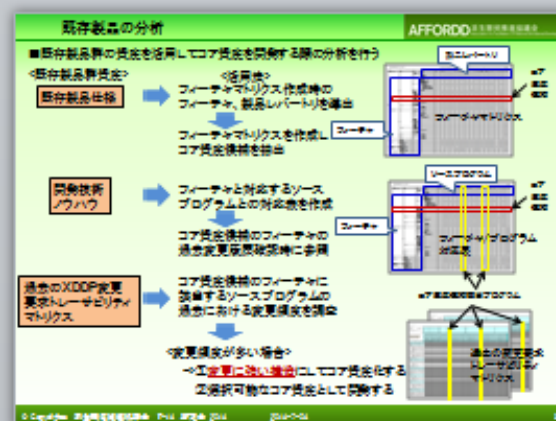
0



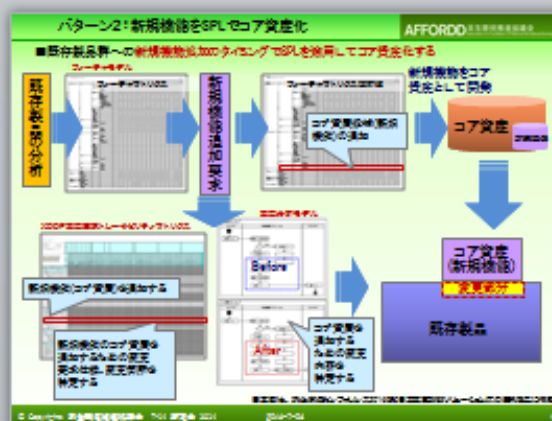
1



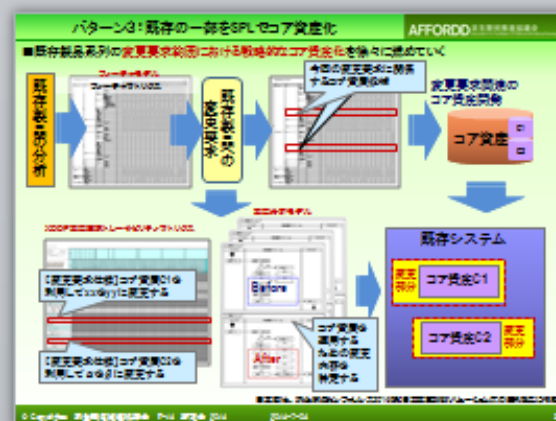
2



3



4



5

解決する課題	既存のコードをコア資産にしたいが、変更/追加が繰り返されて複雑になり、読みにくくなっている。潜在バグがあるかもしれない。 ＝コードの品質が低下している。
目的	既存のコードに対して ・ソフトウェア設計を向上させる ・ソフトウェア設計の劣化を防ぐ ・ソフトウェアを理解しやすくする ・潜在バグを見つけ出す
前提	正しく動いているプログラムをリファクタリングする。
概要	リファクタリングの概要、指針、手順、詳細作業に分けて説明する。
留意事項	・リファクタリングのノウハウの多くはC++やJavaを前提に説明されている。それらをC言語にも適用できるように説明する。 ・コードの改善を主体としてリファクタリングの方法を説明する。⇒アーキテクチャの構造が変更される場合(サブシステムやモジュールの追加/結合/分割/削除など)は、適宜設計ドキュメントを修正する必要が発生する。

(3)リファクタリングの必要のあるコード[コードの不吉な匂い]

#	匂い	説明
1	重複したコード	同じようなコードが2箇所以上で見られたら、1箇所にまとめることを考える。
2	長すぎるメソッド(関数)	長い関数は理解するのが困難である。小さい関数に分けるとコードを追いかけるのが困難となるが、適切な関数名を与えることで、内部の実装を見なくても読み進めることが可能になる。関数の長さを切り詰めることが重要なのではなく、関数名と実装の乖離を埋めることが目的である。
3	巨大なクラス(モジュール)	1つのモジュールが多くの仕事をしていると、変数を持ちすぎている可能性がある。
4	多すぎる引数	引数があまりにも多いと、1つ1つが何を意味しているか理解しづらくなる。
5	変更の発散	1つのモジュールが互いに独立した理由で同じように変更され、その手順も異なる。つまり、1つのモジュールがさまざまな変更要求を被る。
6	変更の分散	1つの変更要求に対して複数のモジュールが影響を被る。変更を行うたびにあちらこちらのモジュールが少しずつ書き換わることになるので、変更とモジュールが1対1になるように修正する。
7	属性、操作の横恋慕	他のモジュールのデータ処理を行っている。
8	データの群れ	数個のデータがグループとなってアクセスされる。
9	基本データ型への執着	基本のデータ型を用いて、それ以上の意味をもつものを表わしている。
10	スイッチ文	コードのあちこちに同じようなスイッチ文が見られる。新たな分岐を追加したときに、すべてのスイッチ文を探して似たような変更をしなければいけない。
11	パラレル継承	新たなサブクラスを定義するたび、別の継承木にもサブクラスを追加しなければいけない
12	怠け者クラス	ほとんど何もしていないモジュール。十分な仕事をせず、理解や保守のためのコストに見合わない。
13	疑わしき一般化	現在は必要とされていない機能のコードがある。
14	一時的属性	インスタンス変数の値が、特定の状況でしかセットされない。ある時は設定されているが、それ以外のときはヌル。
15	メッセージの連鎖	受け取った値を他のモジュールに送り、それに対して受け取った値をさらに他のモジュールに送る。
16	仲介人	値を仲介するだけの関数が多すぎる。
17	不適切な関係	モジュール同士が不必要に密接に結びついている。結びついているモジュールが増えると、変更があったときにそれを伝えることが大変となるため、修正も大変になる。
18	クラスのインタフェースの不一致	同じ処理を実行している2つのクラスが、異なるメソッド名になっている。
19	未熟なクラスライブラリ	モジュールが要求と合致していない。
20	データクラス	データとsetter/getter以外を持たないモジュールは単なるデータ保持用であるため、他のモジュールからのアクセスを過剰に受けがちとなる。
21	相続拒否	サブクラスは親の属性と操作を継承するのが普通だが、ほんの一部しか利用されない。
22	コメント	コメントの多いコードは、分かりにくいコードを補うために書かれているかもしれない。

分類	作業#	作業	詳細	含まれる作業#
作成する	1	サブシステムを作成する		
	2	モジュールを作成する		
	3	変動部を作成する		
	4	関数を作成する		
削除する	5	サブシステムを削除する		
	6	モジュールを削除する		
	7	変動部を削除する		
	8	関数を削除する		
抜き出す	9	サブシステムを抜き出す	一つのサブシステムを二つに分割する	1, 22
	10	モジュールを抜き出す	モジュールを分離して、それを使う	2, 23, 24
	11	変動部を抜き出す	構造体を複数の変数へ分離する	3, 7
	12	関数を抜き出す	関数を分離して、それ呼び出す	4, 25
統合する	13	サブシステムを統合する	二つのサブシステムを一つに統合する	22, 5
	14	モジュールを統合する	二つのモジュールを一つに統合する	23, 24, 6
	15	変動を統合する	複数の変数を構造体へ統合する	3, 7
	16	関数を統合する	二つの関数を一つに統合する	25, 8

分類	作業#	作業	詳細	含まれる作業#
名称変更する	17	サブシステムを名称変更する		1, 5
	18	モジュールを名称変更する		2, 6
	19	変動部を名称変更する		3, 7
	20	関数を名称変更する		4, 25, 8
移動する	21	サブシステムを移動する		1, 22, 5
	22	モジュールを移動する		2, 23, 24, 6
	23	変動を移動する		27, 3, 7
	24	関数を移動する		4, 25, 26, 8
	25	アルゴリズムをコピーする		
	26	アルゴリズムを置き換える		
	27	型定義を移動する		
アクセスを変更する	28	ラッパモジュールを追加する		2, 33
	29	変数をカプセル化する	セットとゲットのペアを追加する	4, 31
	30	モジュールのスコープを狭める	ヘッダを削除する	
	31	変数のスコープを狭める	ヘッダから変数を削除する	
	32	関数のスコープを狭める	ヘッダから関数を削除する	
	33	委譲関数を追加する		4, 25, 26
	34	セット関数を削除する	直接変数にアクセスする	8
	35	ゲット関数を削除する	直接変数にアクセスする	8

作業#28ラッパモジュールを追加する

1. ラッパ用のモジュールを、「[作業#2モジュールを作成する](#)」を適用して作成する。
2. アクセスする関数を特定する。
3. ラッパ用モジュールに、特定の関数をまとめる「[作業#33委譲関数を追加する](#)」を適用して追加する。
4. ラッパモジュールにまとめた関数のそれまでコールしていたモジュール内部のアクセスをラッパモジュールの関数をコールするように調整する。
5. 各関数を調整後、コンパイルしてテストする。
6. ラッパモジュールをアクセスすることで、必要なくなった関数を取り除く。
7. コンパイルしてテストする。

