Software Factories，現状と将来 ——マイクロソフト社の次世代開発基盤技術

Software Factories, Present and Future
——Next Generation Development Platform of Microsoft Corporation

萩 原 正 義

要 約 Software Factoriesはマイクロソフト社が2004年に発表したソフトウェア開発の生産性を改善する開発基盤技術であり，ソフトウェア開発の工業化を目指す．Software Factoriesは，ソフトウェアプロダクトライン，アーキテクチャフレームワーク，コンテキストに依存した開発環境，モデル駆動型開発，の四つの要素技術から構成され，日本のソフトウェア工場の考え方，ソフトウェアセル生産方式を取り入れた普及が期待される．本論文では，Software Factoriesの現状と今後の方向性を説明する．

Abstract Software Factories was announced as the software development platform in 2004 by Microsoft Corporation to improve the productivity and aims to industrialize software development. Software Factories consists of four technologies; software product lines, architecture frameworks, development environments in context, and model driven development. Software Factories will become prevalent with Japanese Software Factory and software cell development method. This paper introduces the present condition and future directivity of Software Factories.

1. はじめに

これまで多くの開発方法論，設計手法，パラダイム，アーキテクチャ，フレームワーク，プログラミングモデル，プロジェクト管理方法などの要素技術が進化をしているにも拘わらず，ソフトウェア開発の諸問題は依然として解決せず，むしろ，複雑さと混乱を招いている．これを要素技術の進化を有効に活用していない開発基盤の問題ととらえ，米マイクロソフト社は2004年に次世代開発基盤技術Software Factories[*1]を発表した．その技術は徐々に現実化し普及を始めている．Software Factoriesの起源は日本における1970年代の「ソフトウェア工場」にあり，現在の要素技術を前提に進化させた考え方である．本論文では，Software Factoriesの現状と今後の方向性を説明する．

最初に，ソフトウェア開発の条件に応じて適切に技術，手法，プロセスを選択することの困難さを考えてみよう．クリスタルと呼ばれる開発プロセス群では，プロジェクトの規模とプロジェクトの開発の影響度に応じて開発プロセスを適切に選択する指針が示されている．プロジェクトの規模は，開発メンバー数であり，3～6，～20，～40，～80，～100，～200，～500．～1000で分類する．開発の影響度とは，問題が発生した場合の影響度であり，人命に関わる，大きな経済的損失を被る，快適さを失うなど4レベルで分類する．一般には，影響度が小さいプロジェクトでは軽量な開発プロセス，影響度の大きなプロジェクトでは重量級の開発プロセスを採用する．プロジェクトによって異なる適切さの選択は，開発プロセスだけではな

く，アーキテクチャ，設計手法，実装技術，テスト手法，実行環境など，多くの開発基盤技術要素で考慮しなければならない．

Software Factories は，この開発基盤技術要素の適切な選択の問題を考慮の上，特に開発生産性の改善を行う．現状の開発よりも数十パーセント程度の生産性の改善ならば，適切な開発プロセスやプロジェクト管理を行うことで実現可能であるが，数倍から数十倍の生産性を品質とともに確保したいのであれば，開発プロセスの見直しだけでは不十分である．つまり，Software Factories は，プロジェクト管理の改善や実装の容易化などの局所的な解決を図るだけではなく，開発基盤技術の全体を見直す．したがって，Software Factories が対象とする技術領域は広範であり，また，実現が困難な問題は多く，難解であると批判を受けることも多い．ただ，開発基盤技術の改善は必要不可欠な段階にきており，生産性の改善も避けられない状況にある．

Software Factories が対象とする技術領域は四つの要素技術から成る．

① ソフトウェアプロダクトライン（ドメイン工学，体制）
② アーキテクチャフレームワーク（資産，可変性の実現）
③ コンテキストに依存した開発環境（開発プロセス単位，ツール，プラクティスの提供）
④ モデル駆動型開発（DSL，テンプレート）

これらの要素技術を単独で使ったとしても一定の効果は生まれるが，適切に組み合わせることで有力な開発基盤技術となる．以下ではそれぞれの要素技術の Software Factories での位置づけを解説する．

2. ソフトウェアプロダクトライン

本章では，Software Factories の基本となるソフトウェアプロダクトライン[4]の概要を説明し，その中で Software Factories の特長を明らかにする．また，基本概念として重要な特性（feature）とドメイン工学の位置づけを説明する．

2.1 ユースケース，機能，特性

図 1 は Web アプリケーションによりオンラインで商品を購入する例である．簡単にするため，Web 購入者が商品を検索する，商品を注文する，注文状況を見る，また，サイト管理者が売上を分析する，の四つのユースケースが定義されるとする．ユースケースは，この Web アプリケーションと相互作用する Web 購入者やサイト管理者などのアクターに対して，業務的価値を与えるシステムの振る舞いの単位である．この四つのユースケースを実現するためには，Web アプリケーションは，販売商品を登録し，価格設定などを行う商品管理，Web 購入者を認証し注文を実行するための顧客情報を管理する顧客管理，実行した注文の記録，注文状況の追跡，決済のための注文管理，売上などのデータを分析するためのデータ分析の機能を持っていなければならない．これらの機能を使いユースケースは実現されるが，機能とユースケースの関係は一般には多対多の関係にある．たとえば，商品管理の機能は商品検索，商品の注文など複数のユースケースで利用され，また，商品の注文のユースケースは，商品管理，顧客管理，注文管理など複数の機能を利用する．ユースケースや機能を実現することで，Web アプリケーションの開発は可能であるが，類似の Web アプリケーションの開発を将来複数回着手する可能性のある場合は，これだけでは不十分である．すなわち，将来，商品の種別を増や

す，価格設定方法を変更する場合や，銀行振り込み，代引き，クレジットカードなどの複数の決済方法を注文方法で選択する場合への対応を考える必要がある．このように複数のプロジェクトで再利用可能な資産を作っておくことが，ソフトウェアプロダクトラインの考え方である．従来の一度限りの作り切りの受託開発に対して，バージョンアップをしながら保守により資産を継続化する製品開発の考えを採用したとも言える．複数回のプロジェクトで再利用可能な資産を構築するために，機能のうち，プロジェクト毎に要求が変わる部分を，置き替え可能な可変性を持たせた設計としておく．たとえば，商品管理での価格設定方法，注文管理における決済方法などに可変性を持たせておく．可変性を持つ機能をプロジェクト毎に取捨選択できるように，機能の選択の単位とするのが特性である．これがユースケース，機能，特性の関係である．

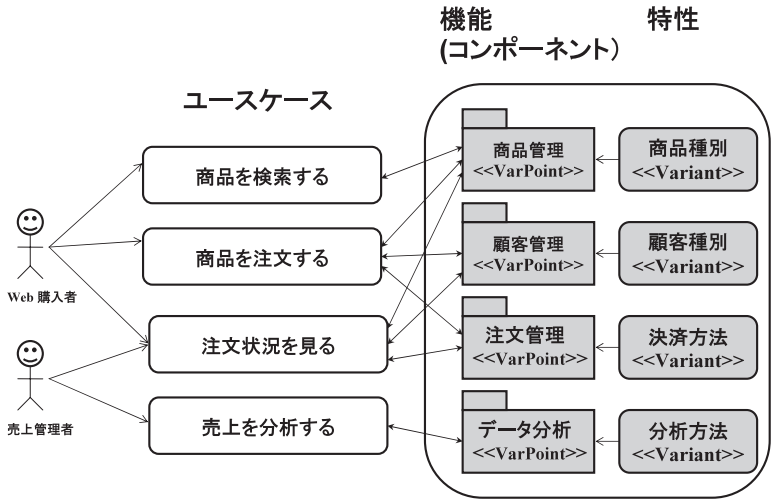

図 1　Web アプリケーションにおけるユースケース，機能，特性

特性は，ユーザに提供するビジネス上の可変性と，ソフトウェア開発上の可変性に分類される．前者は問題領域特性，後者は解決領域特性と呼ぶ．問題領域特性はユーザの選択肢，ユーザにとっての価値を表現する．一方，解決領域特性は，設計の単位，拡張性，テストなどに関して定義する．正確には，特性は機能の要求だけでなく，その機能をよりよい品質で提供する要求を含むので，機能要求だけではなく非機能要求も含んでいる．一般的にいえば，特性は「仕様」に近い．乗用車のラインナップでは，備品（オプション）の選択がユーザに価値を与える問題領域特性であり，機能要求だけでなく，性能に関する非機能要求も含まれる．これらの特性をユーザはカタログから選択し，乗用車の製造メーカは特性の選択に合わせて製造を開始する．乗用車の製造メーカは乗用車をフルカスタムで開発するビジネスは行わず，ニーズの高い仕様の選択肢を用意してラインナップを構成する．そうすることで，100% のユーザ要求は満足できなくても，大半の要求を満足する製品を効率的に製造することが可能となっている．ソフトウェア開発で，この仕様の選択ができるのはパッケージソフトウェアを利用する場合だが，フルカスタムでの開発が主流である受託開発であっても，現在の状況からカスタム化のニーズの高い仕様の選択肢と品質が確保された部品を用意して効率的な開発に移行させることは可能である．

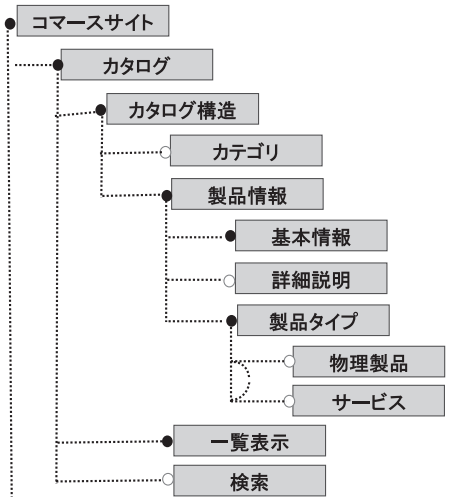

図2　特性モデルツリー

特性は特性名で識別し，可変点と可変点での選択肢をツリー構造で表現できる．図2はコマースサイトにおける特性モデルツリーの例である．特性名は特性の分析段階で，ドメインの用語定義と同様にステークホルダー間で合意を取っておく必要がある．ただし，特性の利用は複数のプロジェクトにまたがるため，プロジェクト毎にステークホルダーが異なる点には注意が必要である．

2.2　開発プロセス

図1で示されるユースケース，機能，特性は，開発プロセスにおける開発作業の分担の単位となる．RUP（Rational Unified Process）などの開発プロセスの特徴であるユースケース駆動の考え方では，ユースケースの要求を単位として開発チームを構成し，各ユースケースの優先度を考慮して開発担当者を割り当てる．ユースケース単位による進捗管理は，ユーザに対して進捗の説明がしやすく，また，優先度の高いユースケースを重視した開発体制や手順を採れる点で優れている．しかし逆にオブジェクト指向開発では，ユースケースは複数クラスに横断的になり，また，1クラスは複数のユースケースで共有されるので，ソースコードの共有管理の課題を伴う．これに対して，ソースコード管理システムなどソフトウェア構成管理やコード規約を利用する．

大規模システム開発では，ユースケース数の増大に対応し，また，成果物の共通化，再利用を積極的に行う開発の基盤の存在が重要である．この点では，ユースケースを単位とする開発作業の分担に加えて，機能を単位とした開発作業の分担を考える必要がある．複数アプリケーションに共通の成果物を提供することで，冗長性を減らし保守性と管理可能性を得ることができる．この共通の成果物を複数プロジェクトから再利用する場合は，プロジェクト毎に異なる要求を可変性として管理する，拡張性を持つアーキテクチャを構築することになる．プロジェクトの計画段階で，このアーキテクチャの構築を積極的に行うのがドメイン工学の考え方である．ドメイン工学では，プロジェクトに先立つアーキテクチャなどの共通の成果物を構築する開発段階を，独立した開発プロセスとして切り出し，適切な担当者を割り当てる．結果として，開発プロセスはドメイン工学を実行するプロセス（プロダクトライン開発プロセス）と，プロ

ジェクト毎のアプリケーション開発を実行するプロセス（プロダクト開発プロセス）に分離される．図3はソフトウェアプロダクトライン開発でのこの二つの開発プロセスを示している．

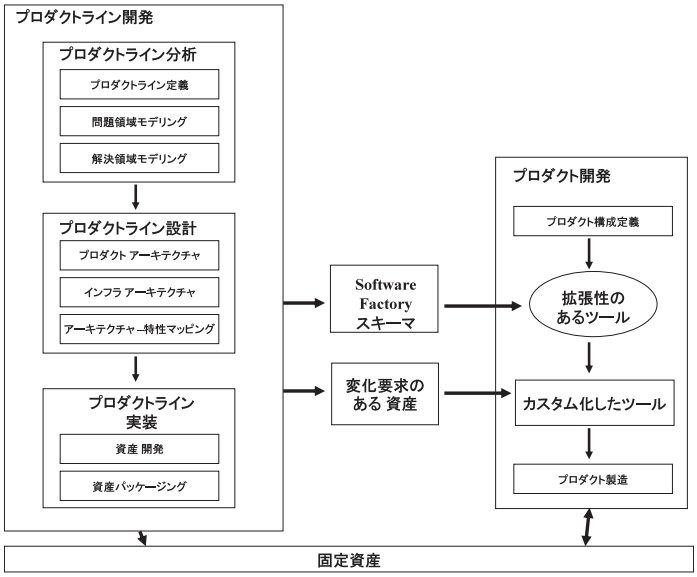

図3　ソフトウェアプロダクトラインの二つの開発プロセス

従来の RUP のように一つの開発プロセスの中で繰り返し型開発を行う場合に比べて，二つの開発プロセスそれぞれで繰り返し型開発を行う点が異なっている．この開発プロセスは，主にパッケージ製品ファミリーの開発に有効と考え，プロダクトライン開発と呼ばれていたが，パッケージ製品だけでなく，一般のアプリケーション開発にも適用するのが Software Factories の考え方である．この結果，プロダクトライン開発が有効となる開発と，プロダクトライン資産の再利用が有効とならない場合のユースケース駆動や機能に基づく開発のいずれの開発方法を選択すべきかの判断が必要となる．

プロダクトライン開発では，二つの開発プロセスの作業分担にバリエーションを持たせることが可能である．⑴システムインテグレータやソフトウェア製品会社がプロダクトライン開発，エンドユーザがプロダクト開発，⑵日本でプロダクトライン開発，海外でプロダクト開発を行うオフショア開発，⑶プロダクトライン開発を行う複数のソフトウェア開発会社がバリューチェインを作り，ユーザがプロダクトラインの成果物を調達する，などである．

プロダクトライン開発では，プロダクトライン開発プロセスとプロダクト開発プロセス間での成果物の引き渡しなどの連携方法についての詳細な規定はない．しかし，実際の開発ではこの連携方法の規定は非常に重要である．Software Factories では，プロダクトライン開発プロセスからプロダクト開発プロセスへ，開発手順，利用する資産と成果物，その他のプラクティスを開発環境のパッケージ化形式で転送する．従来の設計図や指示書のような紙や口頭での曖昧な説明と煩雑な作業を最小化し，開発環境自体のパッケージを転送することで，ソフトウェア部品の再利用だけでなく，開発環境の再利用を推進する．開発環境はソフトウェア開発における工場であるので，生産工場の再利用と言える．すなわち，できあいの部品の再利用に対して，製造設備の再利用に相当する．

3. アーキテクチャフレームワーク

本章では，Software Factories で生産性を改善するための再利用可能な資産の根幹となるアーキテクチャの表現方法と可変部分を分離する方法を説明する．また，資産の標準化の重要性を指摘する．

3.1 コンポーネントによるアーキテクチャ表現

資産をどのような技術で実現するにしても，その技術を選択する意図を説明する表現が必要である．資産の表現には，一般的にアーキテクチャのコンポーネント図が有効である．ここではこのコンポーネント図が，実現技術の選択に対してどれだけ柔軟な表現力を持つかを，アプリケーション処理をワークフローで実行する例で説明する．

この場合，アクティビティを単位とした処理の流れを記述する．代表的には UML アクティビティ図を記法として利用し，アクティビティをコンポーネントで，アクティビティ間の処理の流れをコンポーネント間の関係で表現する．コンポーネント間の関係は，条件により処理の流れを制御する，一つのアクティビティを複数回繰り返す，複数のアクティビティを並行実行する，などの意味を表現する．

アクティビティ図に加えて，アプリケーション処理は状態遷移でも表現可能である．図 4 は状態遷移による表現の一例を示している．アプリケーションの振る舞いを有限個の状態ととらえ，一つの状態から別の状態への遷移で振る舞いを記述する．遷移はイベント受信と遷移条件を判断して実行される．アプリケーションは何らかのイベント受信で開始され，複数の状態を経由して最終的に終了する．この一連の動作で，どの状態がどの物理レベルのソフトウェア部品で実現されるかを決定することなしに，アプリケーション全体の振る舞いを仕様化できる．状態をコンポーネントで，状態遷移をコンポーネント間の関係で表現する．

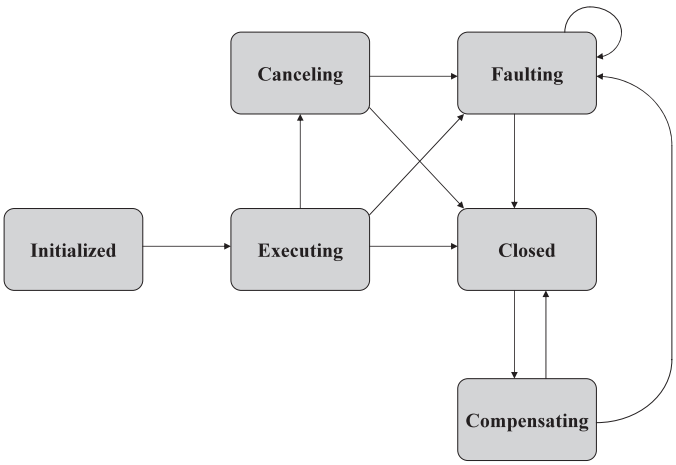

図 4　コンポーネントによる状態遷移図

これらにより，資産をワークフローで実現するとき，アクティビティ図，状態遷移いずれでも，コンポーネント図で表現できることが示される．利用する実現技術により，コンポーネントやコンポーネント間の関係の意味付けが違うだけで，アーキテクチャはコンポーネント図で抽象化できることがわかる．

3. 2　可変性の分離

長期にわたり資産を再利用可能とするためには，コアとなるアーキテクチャに拡張性を持たせ，可変部分を分離することが重要である．これは，アーキテクチャにおける重要な原則，関心の分離の実施の例である．一般に，変化の度合いが同程度の部分を一体としてパッケージ化することが重要である．

アーキテクチャの可変性は，コンポーネントのどの場所が変化するかで表現可能である．さらに，UML にはコンポーネントを束ねるパッケージが存在する．UML パッケージは物理的な配布モジュールではなく，論理レベルで複数コンポーネント間にまたがる変更に同期が必要となる範囲を示している．したがって，UML パッケージをうまく利用することで，可変部分を他の部分から切り出す表現が可能となる．UML パッケージは可変パラメータを指定することで，さらに可変部の定義方法を明確化することも可能である．

可変性を持つアーキテクチャの設計では，必ずしも UML のこうした機能を使う必要はないが，何らかの設計上の意図を明確化する表現法を利用して，資産管理を考えることは重要である．

可変性の分離には，代表的な二つの方法が存在する．一つは，小規模あるいは変化の範囲が小さいシステムにおいて，ビジネスプロセスやユースケースを単位として可変性をとらえる方法である．もう一つは，比較的大規模，あるいは変化が広範囲におよぶシステムにおいて，関心の分離に従い，それぞれの関心を単位として可変性をとらえる方法である．図 5 および図 6 はこの代表的な二つの可変性の分離方法を示している．

このようにして可変性を分離できれば，可変部分をカプセル化し変更の影響を局所化し，あるいは，実行環境の外に定義して，可変点での選択肢を柔軟に取ることが可能となる．たとえば，オブジェクト指向におけるインターフェイスと実装の分離の原則を使った，ストラテジーパターンによる実装や，DI（Dependency Injection）による実行時の実装部分の選択がこの例である．インターフェイス定義に従っていればその実装はカプセル化され，実装方法の変更の局所化が可能となる．可変性の実現方法はこれ以外に多数存在し，その決定はプロダクトライン開発プロセスで資産を作るアーキテクトの判断による．単一の実現方法がすべての条件において有効とはならず，前提となる条件，たとえば，性能を重視する，保守性を重視する，特定のミドルウェアの利用を前提とする，などに基づいて実現方法を決定し，場合によっては，複

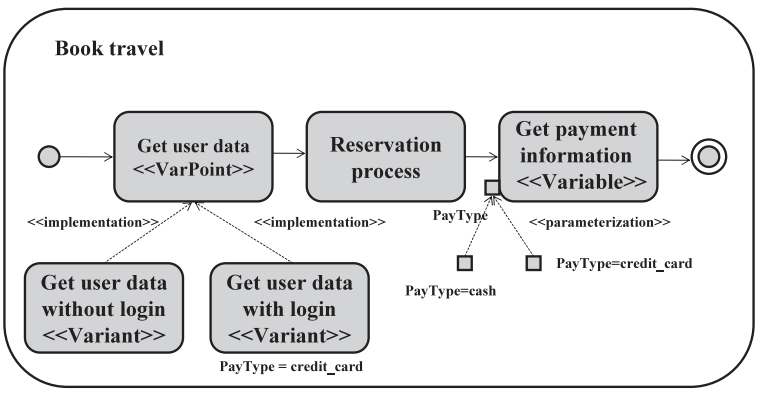

図 5　可変性の分離方法．ユースケース単位

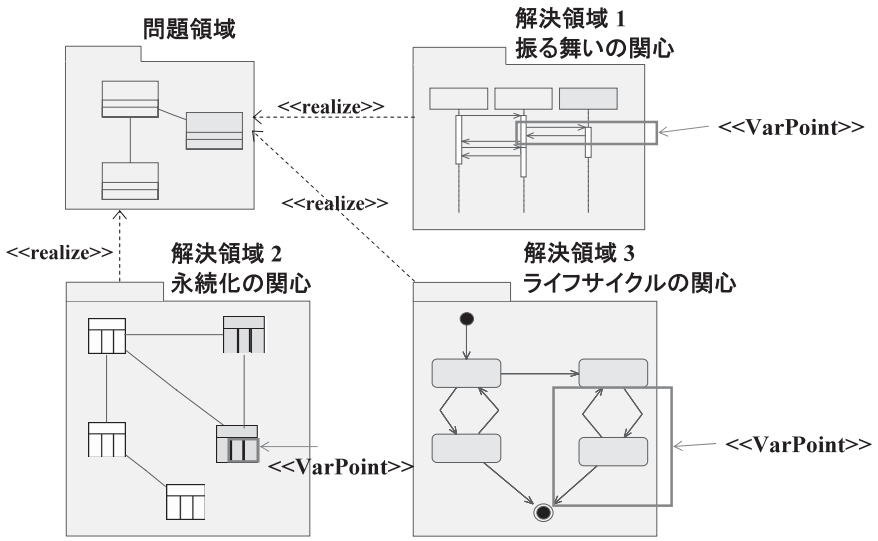

図 6　可変性の分離方法．関心の分離単位

数の実現方法を組み合わせることが必要となる．

3. 3　資産の標準化の重要性

要求に対する網羅性，矛盾の発見はボトムアップ分析によらなければならない．パターンを使いトップダウンで開発を進める方法は，一見効率的であるが多くの場合品質に問題が生じやすく，また，開発者のスキルに依存しやすい．たとえば，データ中心アプローチは，概念モデルを画面帳票のデータ項目に基づき作成する原則を持っている．概念モデルが表現し，構築されるデータアーキテクチャは，個別のアプリケーション要求に依存しない，要求に先行して存在するアーキテクチャ先行の原則を持っている．が，アーキテクチャの拠り所はあくまでボトムアップ分析による業務の現場の個別のデータ項目に基づくのである．オブジェクト指向開発でも，クラス定義は技術だけの基準で決定できないので，何らかのビジネス要求を根拠に決めなければならない．データ中心アプローチとオブジェクト指向を組み合わせれば，クラス定義は業務で使われる画面帳票を入出力とするエンティティで決定できるし，SOA のサービス要求や応答のメッセージを単位とした業務から決定できる．

ただし，これらのボトムアップ分析での部署毎やアプリケーション毎の個別の画面帳票，メッセージ，ユースケースは，システム全体の断片的な表現になっている．したがって，それらすべてを実現するシステム全体では，表現間に矛盾が生じるのが一般的である．この矛盾を解決するのは，ステークホルダー間の合意であり，また，一歩進めて標準化である．たとえば，ドメインの用語定義，エンティティタイプ，サービス，論理レベルのコンポーネント，特性を標準化する．自動車や家電製品が，設計過程で標準部品の策定を進めた結果，システム全体としての品質の確保，保守性，再利用による生産性の向上ができたのを手本とすべきである．従来の開発方法論では，一部を除きこのプロセス上の標準化の重要性が見落とされてきた．Software Factories では，繰り返し開発した実績の結果，蓄積された成果物をソフトウェアプロダクトラインの視点で再構築し，成果物を標準部品として資産とし，プロジェクトの要求を特性に代表させてサービスメニュー化する．この標準化は図 3 のプロダクトライン分析より前段階

の準備期間で行う．標準部品の資産の構築は1回のプロジェクトでは通常不可能であり，複数回のプロジェクトの実績と検証を積み上げることが必要である．

4. コンテキストに依存した開発環境

本章では，Software Factories のプロダクト開発プロセスを説明する．また，チーム開発での開発環境の現状を紹介する．

4.1 開発プロセスを決めるロール，タスク，資産

開発プロセスをプロジェクトの要求に合わせて構成する場合，開発プロセスがロール，タスク，資産の3要素で捉えられることに注目する．すなわち，開発プロセスは誰（ロール）が，何（資産）を，いつ（タスク）開発するかで記述可能である．Software Factories ではプロダクトライン開発において再利用資産が構築されている前提で，プロジェクト毎の要求や制約に従ってアプリケーションをプロダクト開発プロセスで開発する．Software Factories は，プロダクト開発プロセスに対する開発方法論とプロジェクト管理方法に中立であり，特定の開発方法論やプロジェクト管理方法を規定しない．しかし，プロダクト開発プロセスを効率的かつ属人性をできるだけ排除して実行するために，ビューポイントと呼ぶ概念と，開発方法論を支援する Software Factory スキーマの機構を導入している．

4.2 ビューポイント定義

ビューポイントとは，IEEE1471[5]のアーキテクチャ定義に含まれる概念である．Software Factories は，この概念を拡張し，開発プロセスの単位と位置付けている．たとえば，プロジェクトで画面設計は必ずといっていいほど必要となる工程である．画面設計では，プロジェクトによって通常のウィンドウフォーム UI，Web ブラウザ UI，モバイル UI などの画面種別の違いがあり，それぞれで，開発に利用する開発手順，資産と成果物，ツール，プラクティスも異なる．そこで，画面設計に関係する開発手順，資産と成果物，ツール，プラクティスをひとまとめにし，画面設計ビューポイントを定義する．ビューポイントには画面設計以外にも，アーキテクチャ設計，データベース実装など，多くのプロジェクトで再利用性の高い開発プロセスの単位の定義が考えられる．従来，資産と成果物などソフトウェア部品やドキュメントなどの再利用は多かったが，Software Factories では，開発プロセスを再利用し，それ自体を資産として保守していく考え方をとる．IEEE1471 では，ここまで踏み込んだ考え方を示していない点で概念の拡張と呼べる．

ビューポイントはどのような開発方法論を前提としても定義が可能であり，また，それらを組み立てた全体の開発プロセスを実行する場合のプロジェクト管理方法も自由に選択可能である．Software Factories はビューポイントの定義や実行方法に何ら技術的制約を持っていない．だだし，Software Factories に関連したマイクロソフト社が提供する Visual Studio などの利用ツールにより，ビューポイントの定義やプロダクト開発プロセスの実行方法に制約を持つ場合はある．

4.3 ビューポイントの定義例

ビューポイントの例としてテストビューポイントを考える．テストをビューポイントで独立

した定義とする理由は，テストを実行する開発プロセス上の工程に自由度を持たせるためである．テストに必要となるテストケース，ユニットテストのテストツールなどを，適切なテスト方法論を前提にテストビューポイントに含める．そして，実装とテストの間を行き来しながら開発をするのであれば，実装ビューポイントとテストビューポイントの間を行き来する開発プロセスの定義を実行すると考える．テスト駆動のように仕様に近い段階でテストを行う場合は，テストビューポイントは設計ビューポイントと関係を持つことになる．このようなテスト工程を開発プロセス上のどの位置に位置付けるかをプロジェクトに応じて変更可能とするためには，テストを独立したビューポイントで定義しておくのが有利である．同様に，再利用性の高い工程，開発プロセスの単位，作業項目は独立したビューポイントで定義しておく．たとえば，セキュリティ設計，ユースケースシナリオによる概念設計の検証などをビューポイントで定義する．また，開発作業の範囲（ビューポイント定義の粒度）の決定は，ビューポイントの再利用性，有効性を加味して考える．テストビューポイントは負荷テストを細分化して別定義とすべきかどうか，あるいは，設計ビューポイントではアーキテクチャ設計を分離し定義すべきかどうか，などを考える．その結果，粒度の大きいビューポイントで粒度の小さいビューポイントを使い，階層的なビューポイント定義が開発プロセス全体を構成する．

4. 4　Software Factory **スキーマ**

再利用可能なビューポイントを資産として用意し，プロダクト開発プロセスでそれらのビューポイントを組み立てて全体の開発プロセスを構成する．ビューポイントを再利用し，実際のプロジェクトの制約条件を満足して開発を行うためには，制約条件に合わせたビューポイントの構成を組み立てなければならない．たとえば，短納期の制約条件を持つプロジェクトでは，アジャイル開発に近いプロセスを意識したビューポイント構成を取り，また，1 ビューポイントの開発作業も複数人で行う場合もある．

プロジェクトの制約条件を満たすために構成された開発プロセスの複数のビューポイントは，ビューポイント間で一貫性条件を持つ．たとえば，概念データ定義を行うビューポイントでの概念データの属性は，画面の実装やデータベーステーブルを実装する実装ビューポイントでの画面のデータ項目やテーブルカラム値と関係づけられる．この関係づけは開発方法論に依存する．すなわち，開発方法論が，開発の基準となる特定のビューポイント，特定の開発資産，特定のモデル要素や属性を決定し，その開発手順に従ってそれらの間の関係の一貫性条件が決定されるからである．この開発方法論が決めるビューポイント間の一貫性条件を Software Factory スキーマと呼ぶ．たとえば，データ中心アプローチの開発方法論では概念データの属性が開発の基準となり，そこからデータベースの設計と実装，画面の設計と実装，アプリケーション処理の実装，テストが推進され，それらの間での一貫性が考慮される．この一貫性条件を Software Factory スキーマで形式化する．Software Factory スキーマはビューポイントとならびソフトウェア開発環境の基盤となる Software Factories の重要な要素技術である．

4. 5　**ビューポイントで構成する開発プロセス**

ビューポイント構成を定義するプラクティスはソフトウェアセル生産方式[6]から得られる．ソフトウェアセル生産方式は，マイクロソフト社が発表している Software Factories には含まれてはいないが，考え方は併用が可能である．ソフトウェアセル生産方式では，プロジェクト

で一人の担当者が行う開発作業範囲をセルという．セルには複数のビューポイントが含まれると考える．図7でこのビューポイントとセルの関係を示す．たとえば，アプリケーション開発のセルを，実装ビューポイントとテストビューポイントを使った作業で定義する．作業範囲をセルで定義し，その作業を実行する担当者は後で割り当てる．セルは，作業に必要な中間成果物など，そのセルを開始するための入力条件と，セルでの作業を実行して生成される中間あるいは最終成果物など，セルを終了するための出力条件とで定義する．セルは事前条件で生成され，事後条件で消滅する．つまり，事前条件と事後条件を定義したセルは，その条件の満足する間は複数のセルが並行に動作し，並行開発を積極的に行うと考える．セルを注意深く構成すれば，並行度が上がり，開発生産性が向上する．セルの開発作業は，ビューポイントの開発手順に従い，指定のツール，資産を活用して行われる．だだし，セルの開発作業にはビューポイントの開発手順だけでは記述しきれない，自動化が困難な作業が存在する．ビューポイント定義だけで開発プロセスを構成するSoftware Factoriesは，自動化が困難な作業は少ないとして自動化を主体と考えている点でソフトウェアセル生産方式との違いがある．

分担作業をチーム開発用の開発サーバで実行するには，⑴プロダクト開発プロセスのプロジェクトリーダは，プロジェクトの要求を満足する特性（サービスメニュー）を選択し，特性を実現するビューポイントを開発プロジェクトの制約条件に合わせて選択してプロダクト開発プロセスを構成する，⑵プロダクト開発プロセスのプロジェクトリーダは，特性の実現作業をタスクに分割し，各タスクにセルあるいはビューポイント定義とその担当者の情報を付与して開発サーバに格納して，各セルあるいは担当者にタスクを配布する，⑶指定の各担当者がタスクをビューポイントに従って実行する，という手順を踏む．

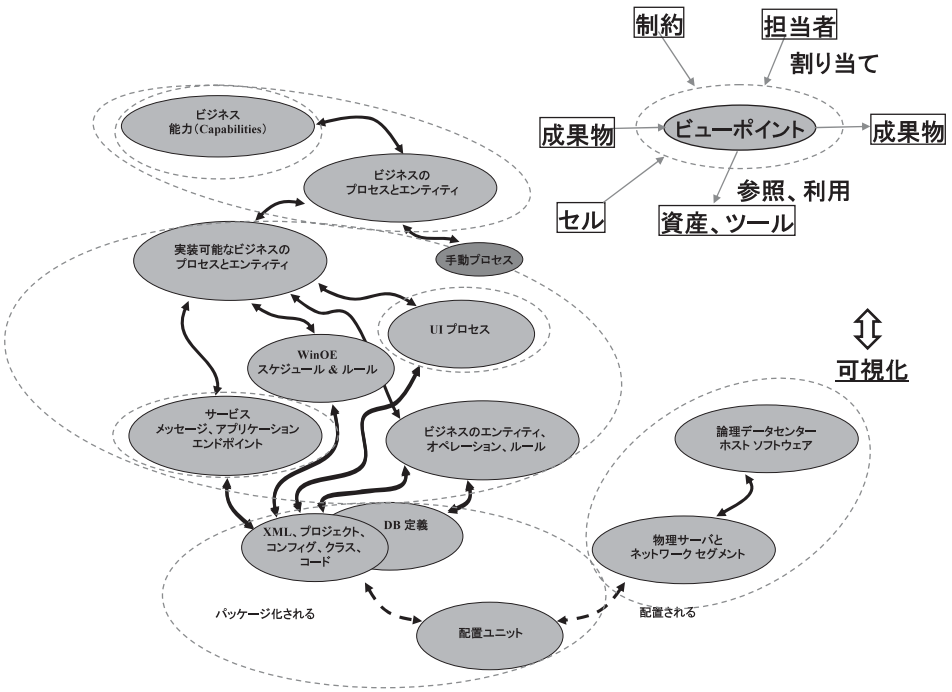

図7　ビューポイントとセルの関係

セルあるいは担当者へのタスクの配布では，タスク内で指定されたビューポイント定義で利

用する資産と成果物，ツールなど，Software Factories の開発環境も合わせてセルへ配布しなければならない．そこで Software Factories 開発環境の配布パッケージを作成するツールが必要である．マイクロソフト社の Visual Studio の場合，Guidance Automation Toolkit[7]が配布パッケージを作成する．

5. モデル駆動型開発

モデル駆動型開発とは，ソースコードによる実装をすることなく概念，設計，あるいは物理レベルのモデルを使い，ソフトウェアを開発する方法である．モデルからソースコードを自動生成して実行する方法と，モデルそのものを実行する方法が存在するが，いずれにしても要求の実現をシームレスに行うことで，要求との追跡性，設計の意図の明確化，保守性や品質の確保，迅速な実装を目指す目的を持つ．しかし，要求には柔軟な変更を求め，保守の対象となる成果物には保守性のために安定していることを求めるため，両者は相反する特徴を持つ．この相反する特徴の結果，要求と設計の間には大きなギャップが存在する．このギャップの存在で，モデル駆動型開発は特定のアプリケーション分野，実行環境に限定して適用する場合がほとんどである．モデル駆動型開発を限定なしに適用可能とするためには，ソースプログラミング言語と同程度の表現を持つモデルが必要であり，モデルが複雑化して，ソースプログラミング言語での記述と大差がなくなりモデルの有効性がなくなる．そこで，適用領域を限定してモデルを定義する方法として DSL（Domain Specific Language）が注目されてきた．DSL は特定のアプリケーション分野，実行環境に限定して適用する点では従来のアプローチと同じであるが，汎用モデルを使わずに限定した適用領域に絞った詳細な表現が可能となる専用モデル定義を作成する点で異なるアプローチであり，開発全体はカバーしない．要求を満たす設計には無数の選択肢が存在し，プロジェクトの制約，品質条件，利用可能な技術選択肢によって決定しなければならない．この決定ではツール支援機能は有効であっても，モデル駆動型開発が目指す完全な自動化はこれまでの実績からかなり困難という状況が見えてきた．

Software Factories の DSL によるモデル駆動型開発は，これらの状況からツールによる自動化が有効となる特定の煩雑な繰り返し作業を主に想定し，要求定義や設計など，創造力やすり合わせが必要となる部分を無理に自動化の対象とはしない．従来の要求や設計の UML モデルを実行のためのモデルに発展させるのではなく，仕様化に限定して従来どおり使おうと考える．つまり，モデルの位置づけとして，実装に関係する部分をできるだけ DSL にまかせ，仕様に関するモデルと分離して使い分ける考え方である．たとえば，データ設計では，従来どおり ER（Entity Relationship）図を使って概念，論理レベルのデータ定義を仕様化する．この部分の開発は，従来どおりのスキルと手順で行うことが可能である．このデータ定義には，当然，実装に関する物理設計やデータアクセス周りのソースコードに関連する情報は含めない．次に，ER 図で物理レベルのテーブル定義を作ることは可能だが，むしろ，ER 図では不十分な実装機能の記述は，物理データ定義と関連コードを生成する DSL で行う方が有利と考える．レポーティングやデータ変更時の通知などを含めて DSL での表現が可能であるなら，汎用性の高い実装を DSL で開発可能であり，物理レベルの ER 図では不可能だからである．この DSL に代えて，物理レベルの ER 図からテーブルを生成し，テーブルにデータバインドするオブジェクト指向言語のクラス定義や関連するストアドプロシージャの自動生成を行う手順でも，同様に実装可能である．DSL はこの開発上の一連の作業を自動化して生産性を高めてい

ると考えられる．DSL の利用法には，ER 図のような既存の仕様に関するモデルを再利用し，作成するアプリケーションに合わせて，DSL でモデリングする方法がある．たとえば，論理レベル ER 図から，プロジェクトのアプリケーション開発で必要とする部分のエンティティや属性を選択して DSL モデルを作成する．そして，この DSL から生成されるソースコードのプロジェクトに対して，追加のコード生成や検証機能を実行するコンテキスト依存メニューを DSL ツール[8]が提供することで，DSL モデリングを支援するなどの例である．

6. おわりに

Software Factories は現在の進化した要素技術の活用に関して多くの野心的な考え方を持っている．たとえば，アスペクト指向を使い可変性の単位をモジュール化し，ソフトウェアで求められる特性に合わせてモジュールを組み立てるという考え方である．しかし，まだ現状の開発環境の制約で実現されていない．このように現実的な制約があるにしても，実績のある日本のソフトウェア工場の考え方に基づいているので，いずれ Software Factories の理念は実現されると期待している．だだし，リリース済みのプロダクトに対する障害追跡機能など，日本のソフトウェア工場の実績に Software Factories がおよばない部分は存在する．これには日本からのフィードバックが有効であろう．

Software Factories に関連したソフトウェア工学の重要な課題に，ソフトウェアの再利用性と生産性の評価が存在する．Software Factories，あるいは，ソフトウェアプロダクトラインは資産を先行して構築する．この資産の構築が正当化できるかどうかは開発着手の段階で評価しなければならない．しかし，現在のソフトウェア工学では，資産の再利用や開発の生産性を正当に評価する有効な方法は見つかっていない．Software Factories では，資産は可変性を定義する特性を単位としてプロジェクト毎に選択され再利用されるので，特性を再利用性の評価に利用できるかもしれない．なぜなら，特性はコンポーネントより抽象レベルが高い要求を表現し，特定のプロジェクトの開発とは独立したモデルだからである．ただし，既存の資産は可変性を意識した特性による体系化がなされていないので，この評価法の適用は不可能であり，評価においてドメイン分析と同様な資産の分析と整理を必要とすることに制限がある．一方，開発生産性は，ビューポイントあるいはセル単位で測定可能であろう．ビューポイントあるいはセルはその事前，事後条件が定義されるので，終了までの時間を測定可能である．ビューポイントあるいはセル毎の生産性が測定できれば，全体の開発プロセスは組み立てられたビューポイントから測定可能となるはずである．Software Factories が本格的に普及し，多くの実績が蓄積されれば，より正確な評価方法を得る可能性が高まると考えられる．

＊ 1　マイクロソフト社が提供する Software Factories と 1970 年代からの日本のソフトウェア工場を英語と日本語で区別する．

参考文献　［1］ Krzysztof Czarnecki, Ulrich Eisenecker, “Generative Programming: Methods, Tools, and Applications”, Addison-Wesley Pub, 2000.
［2］ Jack Greenfield, Keith Short, Steve Cook, Stuart Kent, “Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools” Wiley, 2004.
（邦訳）ジャック グリーンフィールド，スティーブ クック，キース ショート，ス

チュアート ケント,「ソフトウェアファクトリー―パターン, モデル, フレームワーク, ツールによるアプリケーションの組み立て」，日経 BP ソフトプレス，2005.
［3］ 次世代開発基盤技術“ Software Factories ”詳解 http://www.atmarkit.co.jp/fdotnet/softfactory/index/index.html
［4］ Software Product Lines http://www.sei.cmu.edu/productlines/
［5］ IEEE1471 IEEE Recommended Practice for Architecture Description of Software-Intensive Systems, IEEE Recommended practice for architectural description of software-intensive systems
［6］ 松本吉弘,「最近のソフトウェアファクトリ論」，SEC journal No.7，pp.40-49，情報処理推進機構，2006.
［7］ Guidance Automation http://msdn2.microsoft.com/en-us/library/Aa546407.aspx
［8］ DSL Tools http://msdn.microsoft.com/vstudio/DSLTools

執筆者紹介 萩 原 正 義（Masayoshi Hagiwara）

1993 年マイクロソフト入社．稚内北星大学客員教授，過去に北海道大，早稲田大で非常勤．.NET 開発，アーキテクチャの調査研究と技術啓蒙に従事．開発方法論，データ中心アプローチとオブジェクト指向分析/設計との融合，モデル駆動型アーキテクチャ，サービス指向アーキテクチャ，アスペクト指向，フレームワーク技術などが現在の興味対象．Software Factories および関連技術の研究，普及を推進中．