ソフトウェアファクトリ序説
An Introduction to Software Factory Concept

松本　吉弘
財団法人　京都高度技術研究所

１．まえがき

ソフトウェア生産方式には、わが国が世界をリードした 70-80 年代のいわゆる「サイロ型ソフトウェア生産」があるが、90 年代からは、米国がリードする「水平開放基盤型ソフトウェア生産」がこれにとって替わっている。つぎに来るべきソフトウェア生産方式は、「セル」とよぶ小集団、または個人が開発・保守するサービス・コンポーネントを、利用者が低コストで組み立て、互いにネット上で組み合わせてシステムを構築する「グリッド状開放基盤型ソフトウェア生産」であろうと、筆者は推測している。

高品質・商用アプリケーションソフトウェアの開発・保守・改善サービスを、高い顧客要求満足度のもとで、短い開発リードタイム、短いタイム・ツー・マーケット、高生産性のもとで、工業的に形成し、提供する組織体のひとつとしてのソフトウェアファクトリがある。この解説では、「グリッド状開放基盤型ソフトウェア生産」を実現するためのソフトウェアファクトリについて、提言を行っている。

２．ソフトウェアファクトリの定義

2.1　古典的ソフトウェアファクトリ定義

1968 年、ドイツ・ガルミッシュにおいて、第 1 回ソフトウェアエンジニアリング・ワークショップが開かれ、「ソフトウェアエンジニアリング」という語が初めて鋳造されたとされている。そのとき、M.D. McIlroy は、**再利用によってソフトウェア生産性、品質を向上するための組織を、ソフトウェアファクトリとよぶ**ことを提唱した [McIloy69]。

その後まもなく、米国にあっては、Software Development Corporation (SDC)、TRW Inc. などが、ソフトウェアファクトリを試み、IBM は、サンノゼ事業所に、Santa Teresa Laboratory [McCue78]と銘打ち、周到にデザインされたソフトウェアファクトリを発足させた。また、AT&T は、Programmer's Workbench System (PWB)を開発し、全事業所に適用しようとした。これらの試みは、いずれも数年後に終焉を迎えている。その後、ビクタ・バシリらは、Experiment Factory (EF)と称する「経験」の再利用によってソフトウェアの生産性、品質を向上するための組織を開発し、国防省傘下の一部のソフトウェア開発現場に適用して、成果を挙げた [Basili89]。バシリらは、この成果に基づいて、1990 年代初め、Motorola, Inc.で商用 EF を試みた。

筆者は、1980 年代、シーメンス、ABB などにおいてソフトウェアファクトリを構築するための計画にいくつか参加した。その後、ヨーロッパ共同体のなかの委員会は、Eureka

Software Factory (ESF)と称する構想を発表した[Aaen97]。ここでは、ソフトウェアファクトリは、**プログラミング品質および生産性を向上させるために必要な標準的なツール、成果物、およびマネジメントデータを管理するためのデータベースを共有する、ソフトウェア開発・販売・工程管理・原価管理・調達管理・品質管理を統合した組織である**、としている[Aaen97]。

わが国では、1960 年代末から 1970 年代初めにかけて、電機、電子、通信メーカ各社が、それぞれのソフトウェア事業所をソフトウェアファクトリ（またはソフトウェア工場）と位置づけ、独自のマネジメント体制を確立した。当時、米国は、日本の自動車産業に対するバッシング嵐の只中にあり、わが国ソフトウェアファクトリも、バッシング対象候補のひとつに挙げられた。マイケル・クスマノによる調査報告書「Japan's Software Factories」[Cusumano91]は、MIT スローンスクールが、バッシングのための種探しを目的とする米国のソフトウェア企業から受けた調査受託に基づいて作成されたものである。これを受けてたつ我々が、当時もっとも警戒した点は、ソフトウェアファクトリ設立に、日本政府からの資金援助を受けているか、いないかという質問であった。もちろん、受けている事実はなかったから問題は起きなかったが、受けていたとしたならば、後に他の分野のプロジェクトが体験したようなバッシングを受けたことであろうと推測している。

わが国における、当時の主なソフトウェアファクトリを表 1 に掲げる。当時の日本は、ソフトウェア危機と呼ばれる状況にあったにも拘らず、トップの事業経営者はソフトウェア事業のあり方に無知であり、ソフトウェア事業組織の確立や人集めになんらの関心を示さなかった。一方、電力、鉄鋼、公共機関など基幹産業からのソフトウェア需要は急速な増加を示し、そのしわ寄せはすべて現場のソフトウェア技術者に課せられていた。当時のソフトウェア開発現場では、過度な業務負担に耐えず、突然、出社せず、住所にも戻らない失踪者が絶えず、筆者も警察署へ何度か出頭して警官の協力を仰ぐ苦しみを味わっていた。

表 1 に掲げるソフトウェアファクトリは、現場環境を改善し、現場技術者の苦しみを少しでも軽減しようとの願いから、企業トップを説得し、既設ビル借用またはビル新築から技術者集め、組織作りまでを行ってようやく実現した産物であった。まさに、飢餓地獄のなかでの断末魔のあがきとも言えるものであった。

設立年	企業名	施設	製品	従業員数
1969	日立製作所	日立ソフトウェア工場	BS	5000
1976	日本電気・ソフトウェア戦略プロジェクト	府中事業場	BS	2500
		三田事業場	RT	2500
		三田事業場	App	1500
		我孫子事業場	Tel	1500
		玉川事業場	Tel	1500
1977	東芝	府中ソフトウェア工場	RT	2300
1979	富士通	システム本部	App	4000
		蒲田ソフトウェアファクトリ	App	上記中1500
1983		沼津ソフトウェアファクトリ	BS	3000
1985	日立製作所	情報システム工場	App	6000
				内訳: SE:4000
				内訳: PG:2000

表 1　1960－70 年代におけるわが国の主なソフトウェアファクトリ[1]

これらわが国ソフトウェアファクトリの製品は、当時、まだ戦後復興期にあった電力、鉄鋼、自動車、化学、石油化学、交通など産業・公共分野の近代化、工業化、自動化における中心的役割を果たすとともに、銀行第一次オンライン化、第二次オンライン化を支援する役割を果たした。たとえば、電力供給品質（供給安定、電圧・周波数の安定度など)、自動車などに用いる圧延鋼板の品質などに関して、世界トップの地位を獲得することができたのも、これらソフトウェアファクトリの努力の賜物であった。

当時のソフトウェア企業は、わが国だけではなく、アンドリュー・グローブなどが指摘するように、「縦のサイロ」と呼ばれる形態のなかで、互いに競争を展開していた。個々の企業が、サイロのなかに閉じこもり、他社プロダクトとの関係をまったくもたない縦社会を構成していた。これが 1995 年に入って「モジュラー・クラスタ」と呼ばれる形態へ向かって大きく転換することになった。ソフトウェアプロダクトの共通水平構造を主要企業が協働して識別し、ひとつひとつの水平構成要素の内部を隠蔽するとともに、外部からのアクセス手段（インタフェース）を標準化し、公開することによって、互いに水平プロダクトを共通に利用できるようにする方式、すなわち「オープン化」が始まったのである。この変化は、「垂直から水平への移行 (vertical-to-horizontal transition)」として、知られている。

「モジュラー・クラスタ」時代に入ってからは、サイロの高さ競争ではなく、水平構成要素およびそれに対するアクセス・インタフェースに対して先鞭をつけるための競争が始

[1]筆者は、1977 年に府中事業所内に、電力、鉄鋼、公共、交通システム向けの実時間制御システムに対象を絞ったソフトウェア工場（以下、TSF: Toshiba Software Factory と略称する）の発足に参加し、ソフトウェア再利用を徹底して実践した [Matsumoto81, 87, 92a,92b, 93]。http://www5d.biglobe.ne.jp/~y-h-m/EssenceOfToshibaSoftwareFactory.pdf を参照されたい。

まった。マイクロソフト、シスコ、グーグル、ヤフー、SAP などが、現在のところこの競争を制している。しかし、これら企業が今後とも覇権を握る保証はない。筆者らは、そのつぎに予想される時代、およびそれへの移行において、先鞭をつけようとしている。

図 S1 は、サイロ時代からモジュラー・クラスタ時代にかけての移行に対して、筆者らが予想している 2010 から 2030 年代へかけての生産方式を対比して示している。きたるべき生産方式を、仮に「ソフトウェア・セル生産」と名づけて、説明する。

ソフトウェア・セル生産の基礎となるのは、CORBA-3 が牽引するコンポーネント指向開発である。ソフトウェア・コンポーネントについては、2.3 で説明する。ソフトウェア・コンポーネント相互の間は、公開された水平、および垂直インタフェースで接続される。したがって、ソフトウェアシステム・プロダクトは、図 S1 で示すようなグリッド状の公開されたインタフェースによって区切られ、コンポーネント間は、接続要素によって連携される。現在、ウェブサービス・コンポーネント相互の間は、SOA (Service-Oriented Architecture)によって互いに連携されているが、同じような疎な連携を、エンタプライズ・アーキテクチャ内部においても実現する可能性が生まれる。

ソフトウェア・コンポーネントは、「セル」と称する専門家小集団、または個人によって開発・保守される。1990-2010 年代に見られた、水平構成要素の供給で強力な覇権を駆使した強力なベンダは、2000 年代のなかでその特性を変化させる。グーグルが主唱する薄いネットワーク・オペレーティングシステムの上で、トランザクション・セキュリティ・永続化などの機能をそれぞれ内蔵するコンポーネントが互いに、開放型インタフェースを介して、互いに連携するようになる。

ソフトウェアファクトリは、ベンダ、またはメーカの資産ではなく、IT ユーザの資産へ向けて、その性質を変えていく。IT ユーザのなかで「セル」を組織し、必要なサービス、またはソフトウェア・コンポーネントをネット上で取り寄せ、ソフトウェアファクトリを用いて組み立て、組み合わせてシステム化する。コンポーネントはアジャイルに差し替えすることができる。

ソフトウェア・コンポーネント内部は、外部との独立性が維持できるため、アジャイル開発を許す可能性を高める。アジャイル開発が許されたソフトウェア・コンポーネントの開発を担当するセルは、アジャイル・セルとしての経験を集積することができる。

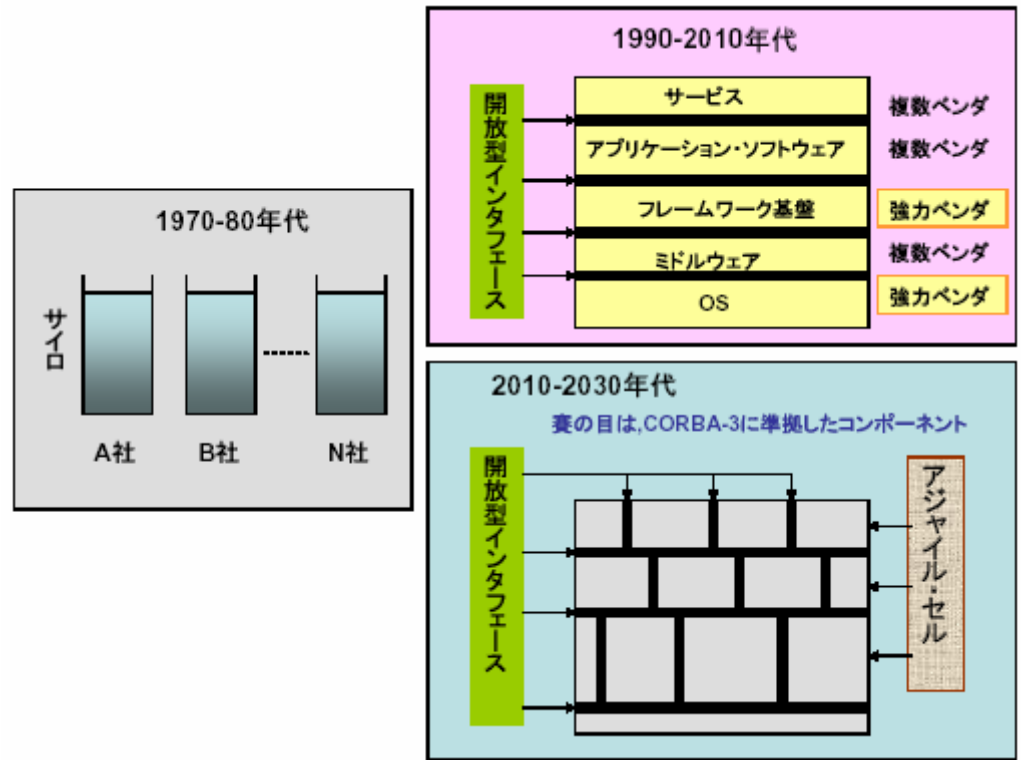

図 S1　ソフトウェア生産方式の推移を説明する図

2.3　最近におけるソフトウェアファクトリ定義

最近の解釈では、ソフトウェアファクトリという語の意味は、GOF [GOF95]のデザインパターンに含まれる「抽象ファクトリ (abstract factory)」に近い。抽象ファクトリの概念を要約して図 1 に示す。図のなかで、「開発者」と称するクラスは、プログラマが作成するアプリケーションプログラムを自動的に生成するための定義記述である。このクラスのなかで、プログラマは、任意のアーキテクチャを選択し、そのなかに使用するコンポーネントをあらかじめコンポーネント・ファクトリで用意されているもののなかから選択し、その組み合わせを記述する。クラス「開発者」をインスタンス化し、実行することによってアプリケーションプログラムが自動生成される。この基礎には、OMG/CORBA3 に主導される、最近のソフトウェアエンジニアリングのコンポーネント指向への移行がある。2002 年に、CORBA3[2]が起案されたが、これは図 2 に示した CORBA Component Model (CCM) を用いる分散アーキテクチャに関する基本仕様である。

[2] http://www.omg.org/technology/corba/corba3releaseinfo.htm

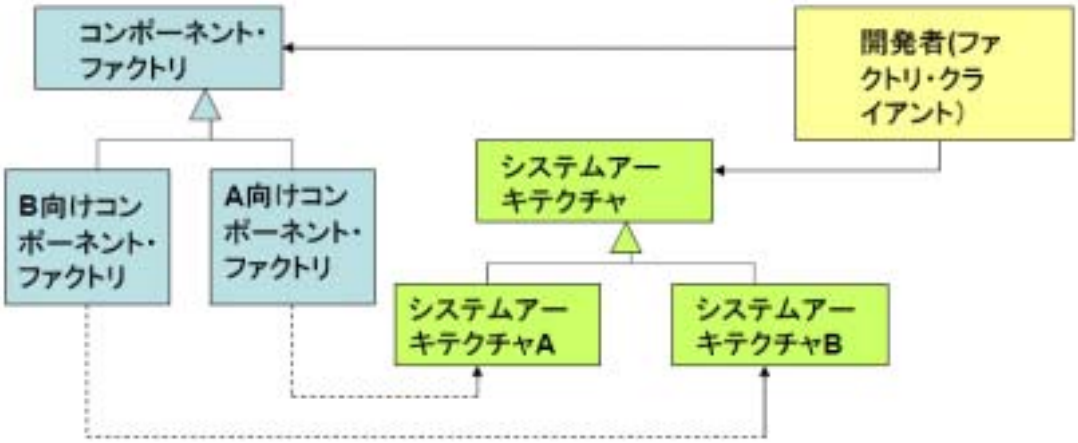

図 1　抽象ファクトリの概念を示す図

CCM のおおよその概念を図 2 に示す。図の左側に配置された公開端部は、外部からの呼びに対して設けられ、右側に配置された公開端部は、このコンポーネントから他のコンポーネントを呼ぶために設けたものである。

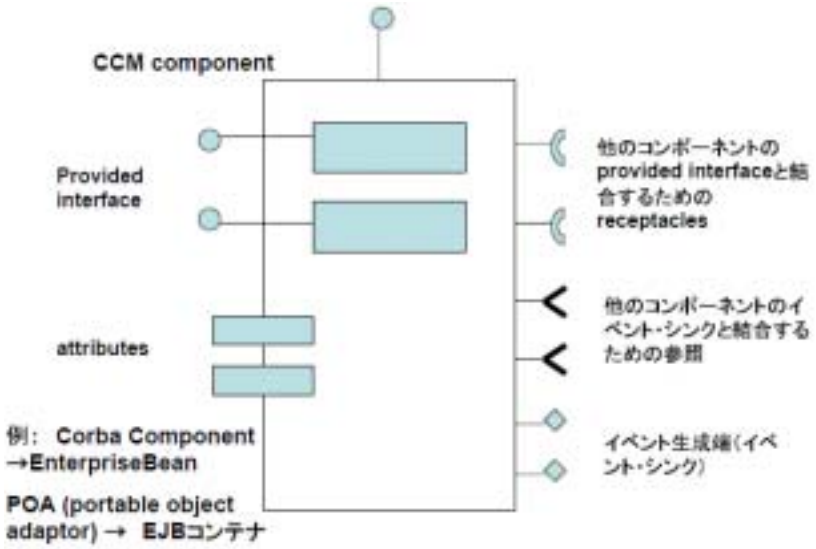

図 2　CCM (CORBA Component Model)の概念を示す図

個々の CCM は、それぞれコンテナに包まれて分散アーキテクチャに組み込まれる。図 3 は、コンテナの概念を示すためのものである。コンポーネントは、図 3 に示すように、インタセプタ、トランザクション制御、セキュリティ制御、永続化制御 (persistency control) などとともに、ひとつのコンテナのなかに隠蔽されて、インタフェース（プロキシ）を介して、分散環境上にあるクライアントからの呼びに応える。

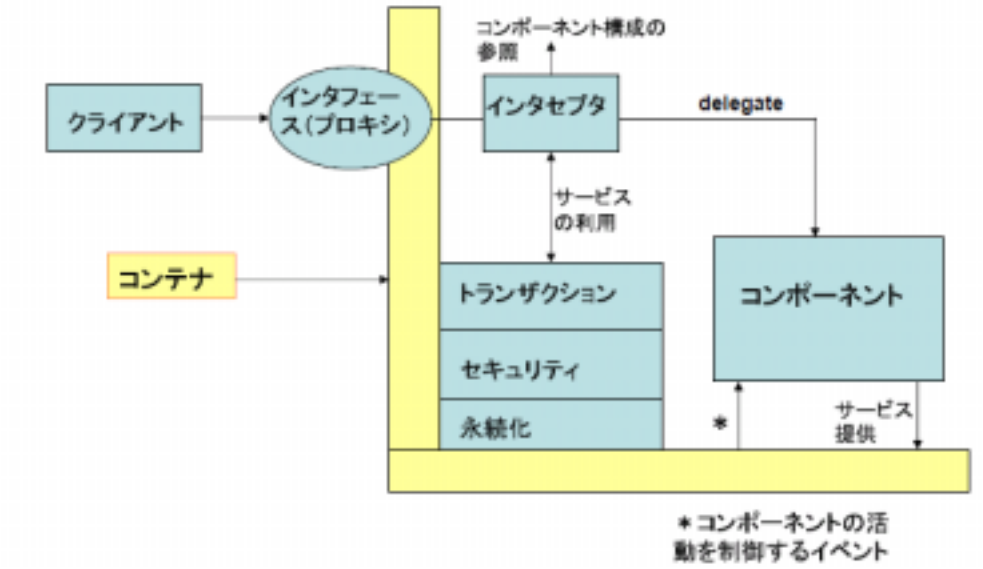

図 3　コンポーネント・コンテナを説明する図

以上に述べた抽象ファクトリというデザインパターンは、プログラム・コードレベルでは実現可能であるが、それよりも上流に位置するエンタプライズ・アーキテクチャ、ソフトウェア要求定義、ソフトウェア設計のレベルでは、そのまま実現することができない。

現在、ソフトウェア業界は、OMG が主導する Model-Driven Architecture (MDA)の渦のなかにあり、モデルを用いれば、上流で描かれるモデルから下流で描かれるモデルまでのシームレスな変換を行うことができ、上流における概念定義からコードに至って、一貫するソフトウェアファクトリの実現が可能である、という夢にとりつかれている。

2000 年代の初めに、Microsoft Corporation は、TSF、EF、ESF、CMMI[3]を参照し、これらからの発展として「ソフトウェア・ファクトリーズ構想 (Software Factories) [Greefield04]」を提案した。この構想は、モデルを描き、それをより具体度の高い別のモデルへ変換するために開発された DSL (design specific language)を用いて、上記の夢を実現しようとするものである。モデル指向開発を主たる軸にしているので、以下、これを Model-Based Software Factory: MBSF とよぶ。MBSF では、ソフトウェアファクトリをつぎのように定義している。**ひとつのソフトウェアファクトリとは、ひとつのプロダクトラインのことである。このプロダクトラインは、ソフトウェアファクトリ・スキーマに基づいたソフトウェアファクトリ・テンプレートに従って、拡張可能なツール、プロセスおよびコンテンツを選定し、フレームワーク部品を適用、組立て、および形成することによって、アーキタイプ・プロダクトの可変部開発および保守を自動化するためのものである。**

この構想には、アプリケーション・ドメインの特化、およびプロダクトライン（製品系列保存庫）という 2 つの前提がある。一意な製品系列を定義するためには、唯一のメタモ

[3] Capability Maturity Model Integration: http:// http://www.sei.cmu.edu/cmmi/

デルのもとに製品系列に属する製品のモデルが統合できる必要がある。このためには、アプリケーション・ドメインをひとつに特定する必要がある。ひとつのプロダクトラインのなかには、特定されたメタモデルを共有する製品のモデル、コード、アーティファクトなどが、資産化されて、記憶・管理される。これらが資産化され、登録されるためには、それぞれの内部で、可変部と固定部が分離されていなければならない。個々の製品は、製品がもつフィーチャ (feature)によって識別される。ソフトウェアファクトリは、プロダクトラインから取り出した資産をカスタマイズして、新しい顧客要求に対応するための仕組みである。したがって、ひとつのソフトウェアファクトリは、上記のように、ひとつのプロダクトラインであると定義されることになる。

2.2 プロセスの工業化について

ソフトウェアファクトリの基本的共通概念のなかに、「工業化」がある。工業化とは何か、を考えるために、ボールドウィン／クラーク [Baldwin00] が提唱する、デザインとそのモジュラ化という思想は、有益な助けになる。ただ、ボールドウィン／クラークが取り上げる対象は、自動車産業、コンピュータ・ハードウェアなどハードウェア生産の世界である。ここでとり上げられている**ハードウェア生産の世界とソフトウェア生産の世界が決定的に異なるひとつの因子に、前者は、調達した部品を加工し、組み合わせる世界であるのに対して、ソフトウェア生産は、問題という原始的な概念を解決するために、前段階で作られたアーティファクト（半製品）を、視点（ビューポイント）を変え、まったく異なる形態をもったつぎのアーティファクトへ創発的、または論理的に変換し、それを組み合わせた後、視点を変えて、再びそれを異なる形態をもったアーティファクトに創発的、または論理的に変換する。このような変換と組み合わせを繰り返しながら、原始概念を解決するための最終製品を形成する。**

ボールドウィン／クラークの思想で注目すべきは、ハードウェア生産の世界においても、創発的な変換が要求されるプロセス（わが国では、「すり合わせ」という用語によって知られている）があり、このような変換を「デザイン」という概念で表象し、デザイン間のモジュラ化を進めることによって、工業化が推進できるとしている点である。さらに、注目すべきは、デザイン間のモジュラ化を進めるために役立つ「モジュラ・オペレータ」と称する 6 つの方法を提言し、モジュラ化によって形成される価値 (ここでは option value と呼ばれている)の評価によって、最善なモジュラ・オペレータの選択を行う方法を示したことにある。

ソフトウェア生産の世界での問題点は、抽象レベルの異なる複数の空間に亘って、ソフトウェア・プロセスが展開されることである。ビジネス・アプリケーションにおけるエンタプライズ・アーキテクチャ・フレームワークと称する方法では、エンタプライズ・アーキテクチャ(EA)、アプリケーション・アーキテクチャ(AA)、データ・アーキテクチャ(DA)、テクニカル・アーキテクチャ(TA)という空間に亘ってソフトウェア・プロセスが展開され、

ひとつの空間内における「抽象レベル内デザイン」と、複数の空間に跨った「抽象レベル間デザイン」が存在する。したがって、モジュラ化には、「抽象レベル内モジュラ化（水平モジュラ化）」と「抽象レベル間モジュラ化（垂直モジュラ化）」が存在する。

水平モジュラ化において、モジュラ･オペレータを適用することはできる。しかし、抽象レベルの異なる垂直モジュラ化に対して、ボールドウィン／クラークのモジュラ・オペレータが適用できるかどうか、については、今後の研究課題である。

2.3 モデル変換

最近中心的な話題となっている設計手法のひとつに、モデル指向手法がある。モデルという語彙を正しく理解するためには、古典的なソフトウェア意味論で提唱された、領域(domain)、論理システム (logical system)、モデルの関係を明らかにしておく必要がある。領域は、k 個の元 (element)をもった空でない集合 (set)U で表される。U^k は、U に含まれる k 個の元の直積集合を表す。k 個の項をもつ述語記号 (predicate symbol)P が存在し、その項に用いられる U^k の部分集合を[[P]]とする。また、k 個の項をもつ関数記号 (function symbol) f が存在し、その項に用いられる U^k の部分集合を[[f]]とする。いま、ある閉じた論理式 ϕ が与えられ、この式を、U, [[P]], および[[f]]を用いて評価し（ϕ に U, [[P]], [[f]]の値を選んでマッピングし、実行すること）、その結果、ϕ の値が真になるとき、マッピングに用いた U, [[P]], [[f]]の値の集まり S をモデル[4]と称し、$S \models \phi$ と表示する。MDA においては、U, [[P]], [[f]]の論理システムへの写像を、論理システムの上に重畳して表示するために、UML (Unified Modeling Language)が用いられる。

たとえば、エンタプライズ･アーキテクチャ手法によれば、ビジネス･アーキテクチャ(BA)、アプリケーション・アーキテクチャ (AA)、データ・アーキテクチャ (AA)、およびテクニカル・アーキテクチャ (TA)と称するそれぞれの抽象レベルにおいて、(i)ひとつ、または複数のモデル、(ii)それらモデル間の依存関係、(iii)論理システムおよび、(iv)モデルから論理システムへのマッピングが記入された論理システムが、UML で記述される。ビジネス・レベルで記述された UML 文書は、アプリケーション･アーキテクチャで作成される UML 文書に変換される。さらに、それはデータ・アーキテクチャ、テクニカル・アーキテクチャに逐次変換される。

この変換に対する理解を助けるために、図 4 を掲げる。

[4] たとえば、U を自然数の集合とし、[[plus]]=m+n という論理式があったとき、この論理式は、U のなかの元の値を選んで m、n にマッピングするとき、ϕ の値が真になり得るので、「自然数集合は、論理式[[plus]]=m+n のモデルである」、ということができます。

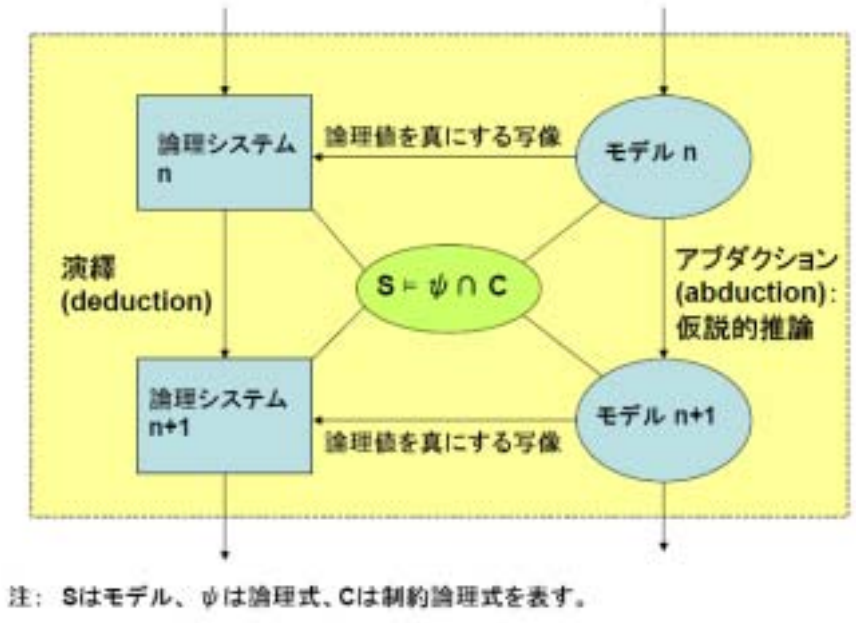

図 4　上流から下流へ向かう設計プロセス上でのモデル変換を説明する図

図 4 は、上流（より抽象度が高い）にあるレベル n から、より下流（より具体度が高い）にあるレベル n+1 へ向かう設計変換プロセスを説明するために作成されている。この図を理解するうえで注意しなければならないことは、ここで記述されている「モデル」は、ソフトウェア意味論でいう伝統的な「モデル」であるということである。MDA で使用されている「モデル」という語は、ソフトウェア意味論で言う「モデル」でなく、同じ抽象レベルにおける論理システムとモデルの組み合わせ、すなわちその抽象レベルにおいてモデルが写像された論理システムの記述を意味している。以後、MDA でいうモデルを、MDA/model と呼び、ソフトウェア意味論で言う「モデル」と区別する。UML は、MDA/model を記述するために企画されたものである。

OMG (Object Management Group)は、抽象度の高い MDA/model から、それより具体度の高い MDA/model への変換を、より形式的に進めるために、QVT (Query/Views/Transformations)仕様を作成し（2002 年 4 月および 2005 年 11 月改訂）、この仕様を満たす変換システムに関する RFP (request for proposal)を発行した。この RFP に対して、およそ 16 件の提案が提出された。それらの内容はきわめて複雑で、まだ現場での実用に耐えるものではない。

2.4　人の設計能力との相互作用

まったく人の介入を必要としないでソフトウェアが生産できるようなソフトウェアファクトリを構築することは、不可能であることを認めない人はいない。どこかに、ソフトウェアファクトリ開発投資と、ソフトウェアファクトリを利用してソフトウェアを生産する

人の雇用投資との最適配分点が存在するはずである。

2005 年、マイクロソフト株式会社（日本法人)、筆者およびベンチャ企業数社は、「ソフトウェアファクトリーズ構想　ホワイトペーパー　1.0」をまとめた [MSJ06] 。そこでは、ボールドウィン／クラークの「デザイン」に対して、それを遂行するために必要な知識、スキル、経験、能力、および境界条件（デザインを実行するために必要な外界とのコラボレーション・インタフェース）を付加した、「ソフトウェア生産セル、またはソフトウェア・セル」という概念を提言している。

この構想を策定するに至った主な原因と動機は、つぎのことがらである。

(1)　モデル変換を QVT 仕様に従って形式化することは、現場にとってあまりにも負担が大きいこと。

(2)　図 4 で示したひとつのモデル変換過程を、筆者は、「単位設計変換セッション」と称している。ひとつの単位設計変換セッションは、ボールドウィン／クラークの提唱するひとつの「デザイン」に相当するものと捉え、問題レベルからコードレベルに至るすべての単位設計セッションに関して、単位設計変換セッションの選定とそれらのモジュラ化、およびモジュール価値評価のために、ボールドウィン／クラークのモジュラ化手法 [Baldwin00] を適用することができる。

(3)　ソフトウェアファクトリの究極の目的は、ソフトウェア生産の工業化、自動化であるが、人の知力 (intelligence)をソフトウェアファクトリのなかに組み込むことは、決してソフトウェアファクトリの目的に反するものではない。「デザイン」、すなわち「単位設計変換セッション」のなかでは、形式化できない暗黙知識、経験などによって駆動されるプロセスとプロダクトが複雑に組み合わされ、これをモジュラ化することは非常に難しいが、「単位設計変換セッション」をカプセル化すれば、カプセル相互の関係に関しては、モジュラ化およびモジュール価値評価手法を適用できる、と考えた。

(4)　欧州を中心にして、トヨタ自動車の生産管理方式を参照する、システムエンジニアリングに関する業界標準団体 INCOSE (International Council on Systems Engineering)が公開している INCOSE Systems Engineering Handbook において、プロジェクトを 3 種類の小集団 (PDT: product development team; PIT: product integration team; SEIT: systems engineering integration team) から構成し、これらを互いに対話させながら並行に活動させる方式が提唱されたこと。

(5)　わが国では、1970 年代以降、TQC (total quality management) などにおいて小集団活動が活発に機能し、多くの実績を挙げており、TSF においては、後にハードウェア生産で採用されたセル生産における「ソフトウェア・セル」の原始的な形態とも言える、S-Mol、またはユニット・ワークロードと称する小集団が、ソフトウェアプロジェクトを構成していた [Matsumoto94]。

筆者らは、上記「ソフトウェアファクトリーズ構想 1.0」に「ソフトウェア・セル」の概

念を導入した。以下、この手法によるソフトウェアファクトリを Cell-Based Software Factory（CBSF と略称する）とよぶ。CBSF では、ソフトウェアファクトリをつぎのように定義している。**セル生産方式ソフトウェアファクトリは、事業として開発・保守するソフトウェアの適用目的領域を特定し、そのなかで蓄積される知識、スキル、経験、固定ソフトウェア資産、可変ソフトウェア資産、およびツールを利用し、ソフトウェア・セルのモジュラな連携によって形成されるプロジェクトによって、受注したソフトウェアシステムを工業的、または半工業的に開発および保守するための事業組織である** [Matsumoto06]。

ここで、**ひとつのソフトウェア・セルとは、ソフトウェアプロジェクトに与えられた問題に対する、ひとつの視点から見た解 (solution)を、その視点に必要な適切な知識・スキルおよび経験をもった、ひとりまたは複数のメンバから構成されるチームが、その視点に対してあらかじめ設定されたセル役割仕様書、記述形式、ツール、手順書、部品、資産、関連文書に適合するとともに、他のソフトウェア・セルと知的に連携しながら、その視点解に相当する成果物を開発、または保守するように設定された、並行して実行可能なソフトウェア生産ライン上の区切り**[5]**のことである**[6]。

2.5 モデル駆動開発

モデル駆動開発 (Model-Driven Development: MDD)には、大きく分けて 2 つの流れ、すなわち Model-Driven Architecture: MDA、および Model-Integrated Computing: MIC [Sztupanovits97] がある。前者は OMG (Object management Group) によって推進され、十分周知されているので、説明を省略する。後者は、同一抽象レベル上であらかじめ準備されたコンポーネント群を管理するミドルウェアを利用し、コンポーネントを構成要素として記述された視点の異なる複数のモデルを構造的に組み合わせ、意味的に統合し、その結果から実装コードを自動生成するための環境と方法論を指している。たとえば、適用対象が主として物理法則によって支配される組込みシステム（たとえば、自動車搭載用電子システム）のようなケースでは、モデルが記述されている抽象レベルが実装レベルとほぼ一致させることが可能であるため、モデルを構成する要素となっているコンポーネントから、直接に実装コードが生成できる。自動車搭載ソフトウェアでは、OSEK[7] コンポーネントおよび実装基盤を利用して、ECSL (Embedded Control Systems Language) [Krishnakumar06] で記述されたモデルから実装コードへの変換開発が進められている。

つぎに MIC におけるソフトウェア開発の手順を簡略に説明する。MIC では、ひとつの抽象レベルにおける、モデル・モデルから論理システムへの写像・論理システム（図 4 で説

---

[5] 特定の個人を結びつけた実体ではない。

[6] ハードウェア生産セル： 一人から数人が、部品取り付け・組み立て加工・検査など、ひとつの製品を完成するまでの全工程、または比較的大きな部分工程に必要な作業を遂行する生産ライン上の区切りである。コンベヤー生産方式に代わる、少人数完結型生産方式である。

[7] Offene Systeme und deren schnittstellen fur die Elektronik im K raftfahrzeug

明した）が、DSML (Domain-Specific Modeling Language)を用いて記述された表現を、GME (Generic Modeling Environment) が解釈・評価する（コンパイルする）操作において実現される。このことは、GME には、DSML 記述の解釈・評価のために、つぎの 5 つ組が組み込まれているためである。

(1) ドメイン構成要素を表現するために必要な特定の表記（テキスト形式、または図形式）を定義する具体的な構成規則 (concrete syntax)

(2) 概念、関係および一貫性制約 (integrity constraints)を記述するための構文記述のために必要となる抽象文法 (abstract syntax)

(3) モデルのもつ意味を、特定されたドメインへ形式的に写像する際に準拠する意味領域 (semantic domain)

(4) 構成規則に従って記述された（テキスト形式、または図形式の）構成要素から、抽象文法への構成的写像 (syntactic mapping)

(5) 文法に従って定義された概念を意味領域へ関係付ける意味的写像 (semantic mapping)

この 5 つ組は、MetaGME が提供するメタプログラムの記述によって、変更することができる。すなわち、GME は、MetaGME を用いて、特定されたアプリケーションに適した DSML へ向けて、metainterpretation と称する操作において、カスタマイズすることができる。これに加えて、GME は、DSML 記述を解釈・評価するに当たって、動的意味を付加する。

３．ソフトウェアファクトリで踏襲されてきた共通概念

1070-80 年代に生まれた日本のソフトウェアファクトリは、現在もそれぞれの形態で発展を続けているが、それらに共通した特徴的な概念を振り返って整理し結果を、図 5 に示す。基本的な特徴は 3 つに集約される。

3.1 アプリケーション・ドメインの特定、およびそのスコープの限定

ある年数を経て事業を継続している企業であれば、必然的に、自分の事業にとって有利な顧客得意先が形成され、その注文対象であるドメインに関するビジネス（システム）・ノーハウや成果物の集積が生まれるはずである。このようなドメインを得意ドメインと名づけるとすれば、複数の得意ドメインを確立し、それらを中心にして利益を生み出すような経営を行うことが、賢明といえよう。有名レストランが、懐石料理からフランス料理まであらゆるメニューを提供することは考えられない。

得意ドメインを選定する意思決定は、経営者の責任である。最近は、この責任を回避する経営者が多いことが悩みの種である。得意ドメインを決定したならば、そのカバーするスコープ（たとえば、発電というドメインであれば、原子力、化石燃料発電から風力発電、太陽光／熱発電まで、さまざまなスコープがある）を限定する。意思決定によって選択されたドメインおよびスコープを、そのソフトウェアファクトリ・スコープと名づける。

大きな事業組織内には、さまざまな得意ドメインが存在するため、多数のソフトウェア

ファクトリ・スコープが定義され、定義されたスコープの数だけソフトウェアファクトリが組織化される。

特定されたソフトウェアファクトリ・スコープに対して過去に蓄積された知識、経験、プロダクトおよびプロセス成果物は、プロジェクト・データベースに、きちんと記憶・管理されていることが、ソフトウェアファクトリ・スコープを定義可能とする前提となる。事業責任者は、イノベーション・チームを編成し、ソフトウェアファクトリ・スコープに対する過去のプロダクトおよびプロセスを、再利用可能な資産に変換することを命じる。ソフトウェアファクトリ・スコープには、ある程度の可変範囲を設けなければない。この資産を、予定された可変範囲に亘って、将来、カスタマイズできるようにしなければならないからである。必然的に、資産は、固定資産と可変資産に分けて作成される。通常、可変資産は XML を利用したテンプレート言語、または DSL (domain-specific language) によって記述される。

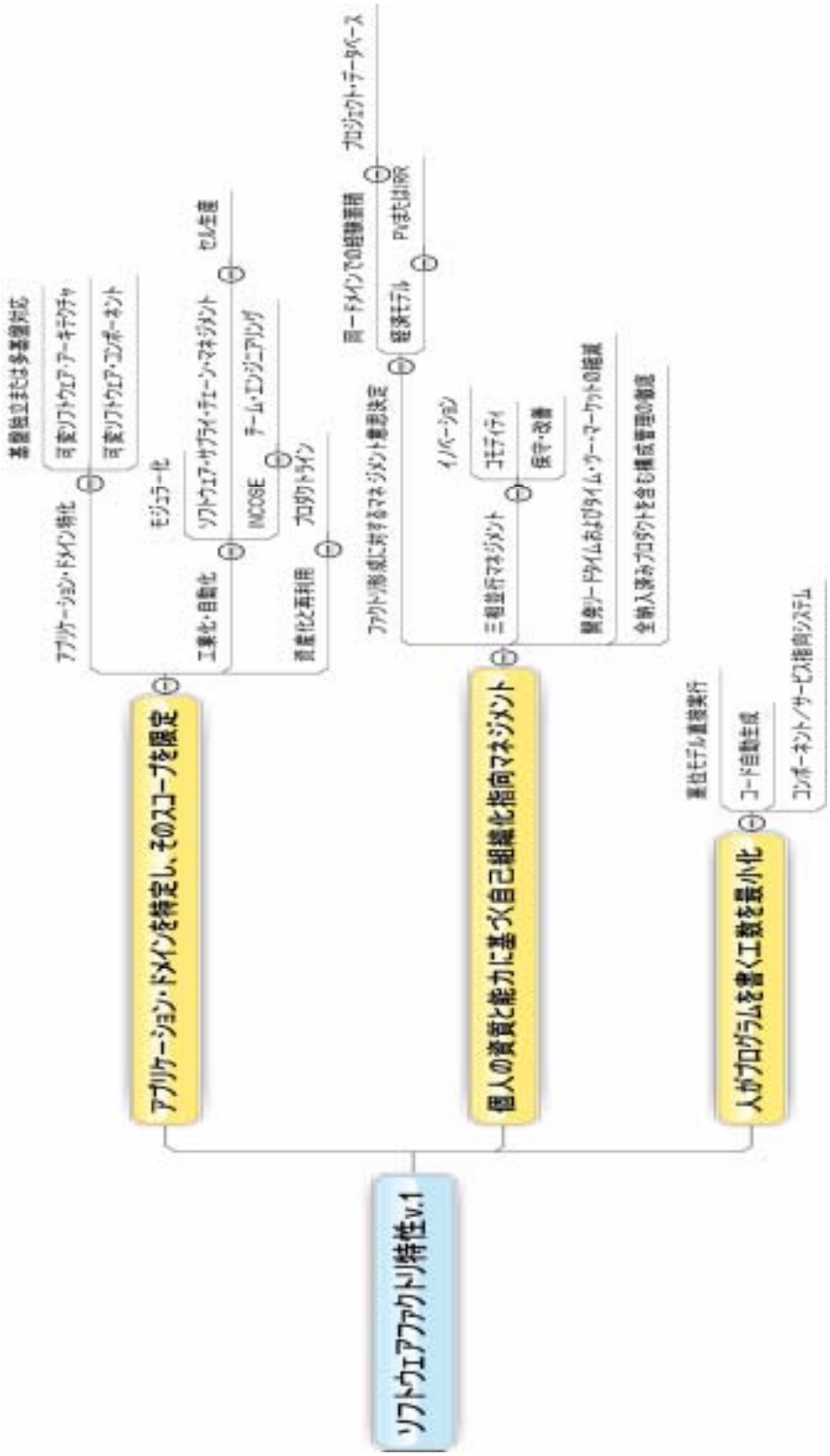

図 5　これまでのソフトウェアファクトリに共通した概念の特徴とそれらの関係を示す図

事業責任者には、自分に直属するスタッフとして、ソフトウェア資産管理課と称する常設組織を設営することが義務付けられる。これは、事業全体が、ITIL (IT Infrastructure Library) に適合するためにも必要なことである。イノベーション・チームによって作成された資産は、ソフトウェア資産管理課に渡され、ソフトウェア資産管理課が責任をもつプロダクトラインに格納され、管理されねばならない。

新しい注文は、事業責任者の責任で、できるだけ得意ドメインだけに絞って受注する。新しい注文をもらったらプロジェクトを編成する。このプロジェクトを、コモディティ・チームとよぶ。コモディティ・チームは、ソフトウェア資産管理課から、利用可能な資産の提供を受けて、それをカスタマイズしながら、顧客の要求を満たす。資産の可変範囲が不十分な場合には、ソフトウェア資産管理課を通してイノベーション・チームにフィードバックし、支援を受けながら新しいプログラムを作成する。イノベーション・チームは、このフィードバックを受けて必要な修正を資産に加えて、プロダクトラインの更新をソフトウェア資産管理課に申請する。

ソフトウェアファクトリの基本となる思想は、なんといっても工業化、自動化である。このためには、ボールドウィンらが説くように [Baldwin00]、生産組織がモジュラ化されていなければならない。彼女らは、生産モジュールを最適化するための単位を「デザイン」と称しているが、TSF ではこれに相当する組織単位を「セル」と呼んでいた。筆者は、ソフトウェアファクトリ・マネジメントを、サプライ・チェーン・マネジメントという概念のなかで捉えている。ある業種のサプライ・チェーン・マネジメントでは、生産組織の最小単位は「セル」と呼ばれることがあるが、TSF はそれに先んじてセルの概念を導入していた。

欧州で実用されているソフトウェアエンジアリングのための業界標準 INCOSE（現在、国際標準機構 ISO への提案が準備されている）[INCOSE04]では、プロジェクトを構成する最小単位を「チーム」とよび、その標準的な活動について規定している。

3.2　個人の資質と能力に基づく、自己組織化指向マネジメント

日本の古いソフトウェアファクトリに対しては、とくに外国から、技術者の人権を無視した作業が強要されていたのではいか、という疑いをかけられることがあった。当時は、(1)企業トップがソフトウェア事業の経営に関してまったく無知であったため、企業として必要なソフトウェアに対するマネジメント能力を欠いていた、(2)ソフトウェア技術教育は、大学はおろか一般の企業では提供することができず、米国で教育を受けた少数の技術者に委ねられていたため、プログラマが極端に不足していた、(3)コンパイルは、都内でも数少ない計算センタが提供する汎用計算機でしか行うことができなかったので、コンパイル作業のために、夜になると、都内の計算センタの空いているマシンに殺到して徹夜する日々が続いていた、(4)納入した計算機の故障が頻繁に起きるため、故障の原因がハードウェアかソフトウェアかを見分けるため、故障の都度、深夜でも地方に出張することを余儀なく

された。このような現象が、前記の疑いを生む結果になったことは否めない。

ひとつのソフトウェアファクトリは、特定されたひとつのアプリケーション・ドメイン、またはアプリケーション・スコープに対して、立ち上げる。ソフトウェアファクトリの立ち上げにとってもっとも必要、かつ重要なことがらは、事業責任者が責任をもって、必要な資源の使用に対して意思決定し、コミット(約定)することである。そのためには、得意ドメインにおける蓄積が必要であり、それとともに、イノベーションに費消するためにどれだけの費用をかけられるか、を論理的に算定する必要がある。この算定方法については、4.1 で詳しく説明する。

ソフトウェアファクトリを中心におくソフトウェア事業のマネジメントは、3 つの基本要素、すなわちイノベーション・チームのマネジメント、コモディティ・チームのマネジメント、ソフトウェア資産のマネジメントから構成される。ソフトウェア資産のマネジメントは、先述のソフトウェア資産管理課を中心にして執行するが、ここで管理する資産には、すでに顧客に対して出荷し、運用中のソフトウェア資産を含む必要がある。このマネジメントの基本になるのは、構成管理手法および ITIL である。

3.3 人がプログラムを書く工数を最小化

1970 年代のソフトウェアファクトリでは、プログラマの不在が緊急の課題であったため、アプリケーション・ジェネレータと称するプログラム自動生成システムが、鋭意開発され、実用された。空白埋め込み言語、ロジック・ダイアグラム、ラダーチャート (ladder chart)、ディシジョン・チャートなどからの自動プログラム生成は、日常茶飯時であった。

当時のアプリケーション・ジェネレータは、データフロー図、E-R 図、状態推移図、論理図などで表現されたプログラミング・レベルの構造に基づいていた。図 6 にそれらに共通したメタ構造を示す。入力は、入力インタフェース、入力変換を経て段階的に抽象化され、中央変換において記憶・管理されているデータ集合を用いて抽象的な出力に変換される。抽象的な出力は、出力変換および出力インタフェースを経て、段階的に具体化され、出力となる。アプリケーション・ジェネレータは、一般に、入力変換、中央変換、出力変換およびデータ集合という 4 部分に分けて開発され、それぞれが可変部と固定部から構成されている。可変値を顧客、または設計者が指示された空白箇所に記入し、アプリケーション・ジェネレータは、その値を受けて可変部と固定部をカスタマイズし、コードを生成し、あらかじめ設定されたソフトウェア・アーキテクチャにこのコードを結合し、ターゲット計算機に実装していた。

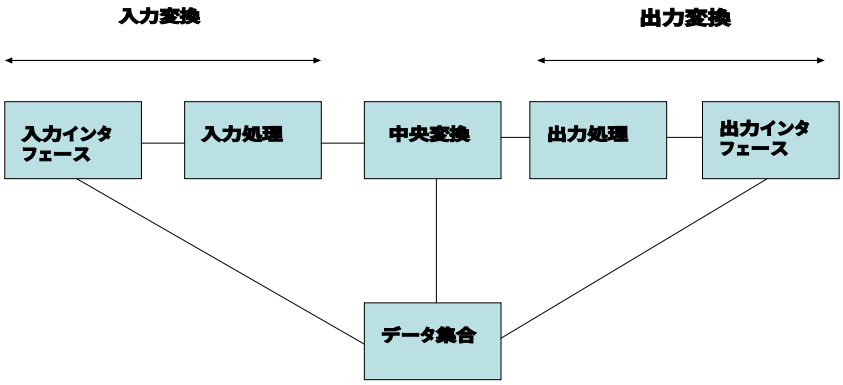

図 6　構造指向設計の基本モデル

CASE (computer-aided software engineering)を含めて、これらツールが、20 世紀末になって使われなくなったのは、俗に言うサイロ問題にぶつかったためであるとされる。サイロのなかに入ってしまうと、外界が見えず、外界との共通性を失った自己中心の閉鎖社会を形成する。20 世紀末から始まった開放型共通基盤によって、前世紀の CASE やツールが、見放された存在となってしまったことを指している。

現在は、特定ドメインのアプリケーション･モデルのために定義された DSL 記述から、目的システムに搭載された開放型基盤上に実装可能なコードへの変換は容易であり、すでに実用されている。しかし、抽象度の高いモデルから、特定された開放型基盤上に実装可能なコードへ変換するためには、途中で、何回かのモデル変換を行って抽象度を下げる必要がある。しかし、このモデル変換は容易ではない。OMG の QVT (Query/Views/Transformations)仕様などによる RFP にも拘わらず、その実現にはまだ多くの問題が残されている。これの対極では、SOA (service-oriented architecture)上で、ウェブサービスを利用したアプリケーションが記述、かつ実装可能となれば、プログラミング工数の縮減にはきわめて役立つであろうことが期待されている。

４．ソフトウェアファクトリの経済モデル

ソフトウェア事業組織が、ソフトウェアファクトリを事業として採用することの意思決定は、当該組織トップマネジメントの責任で行わねばならない。ソフトウェア事業組織は、狙っている顧客の問題領域に関して、十分なソフトウェア製品開発・納入・運用・保守の実績を保有することが必要である。この意思決定を行うためには、対象とする製品系列、または部分製品系列ソフトウェアファクトリのイノベーション段階への投資額 I が、コモディティ段階を通して得られるキャッシュフローの現在価値 (PV)を越えてはならない。つぎ

に試算例を示す。用語については、サイドバーを参照されたい。

- 試算例
  - あるソフトウェア製品系列に関して、資産寿命 5 年のソフトウェアファクトリを利用して、毎年平均 10 システムを受注して、開発・出荷・据付け・運用・保守し、ソフトウェアファクトリのイノベーションおよび保守・改善を勘案し、毎年のキャッシュフロー値が 3,000 万円になるとする。
  - 資本コストが、各種リスクを勘案して 5%と見積もる。
  - これを現在価値 (PV)に換算すると、12,988 万円になる。
  - したがって、ソフトウェアファクトリ開発投資額は、12,988 万円以下であるとき（または IRR が資本コスト（利子率+リスクプレミアム）を上回るので)、当該ソフトウェア製品系列のためのソフトウェアファクトリ開発を決定してよい。

- ソフトウェア開発コストへの投資額を、I としたとき、NPV および IRR はつぎの値を表す。
- NPV (Net Present Value)
  - NPV = PV − I　　(PV は、Present Value, 投資の結果、N 年間に得られる入力キャッシュフロー総額の現在価値、ただし N は、ソフトウェアを運用する年数)
  - NPV > 0　のとき、投資してもよい。
- IRR (Internal Rate of Return)
  - 内部収益率のこと。投資を行なう場合、将来予想されるキャッシュフローの正味値を現在価値に換算した値が、現在の投資額と等しくなるような利率のこと。NPV が 0 になるように PV を求めたときに用いた IRR が、「資本コスト」を上回っていたとき、投資してもよいと判断する。ただし、資本コストとは、資金を調達するに当たっては、資金提供者に対して与え得る何らかの利率のこと。リスク調整後の見返り、たとえば借入に対する利息、株式に対する配当、およびマイナス要因としての保守・改善に必要な費用などを考慮して求める。
- N 年間に得られる入力キャッシュフロー総額の現在価値を計算する式：
  - PV = CF1/(1+IRR)**1 + CF2/(1+IRR)**2 + …… + CFn/(1+IRR)**N
  - ただし、CFx は、x 年目の入力キャッシュフロー
- ■ 税引き後決算期利益から配当金と役員賞与を引き、減価償却を足したものをキャッシュフローと称し、企業の自己資金収支のこと。上の式では、該当するソフトウェアだけに相当するキャッシュフローを考える。

サイドバー　正味現在価値および内部収益率

５．21 世紀に入ってからのソフトウェアファクトリ

アプリケーションの対象となるドメインのかなり部分を自然法則が支配する工業用制御システムの分野では、TSF の見られるように、1970 年代からソフトウェアファクトリが事業化されていた。最近でも、組込みシステム向けソフトウェアでは、イノベーションの結果をコモディティ化し、コモディティを国際標準化することによって市場占有率を高めようとする動きが盛んである。自動車搭載用電子システムなど、いくつかの領域では、ポートフォリオ（運用資産）を開発し、コモディティ化し、それを武器にして市場占有率を高めようとする企業がいくつか存在する。欧州における AUTOSAR (Automotive Open System Architecture)[8]と称する自動車業界の団体では、その活動の一部で、自動車搭載用ソフトウェア・ポートフォリオ標準をまとめようとしている。

組込みシステム以外の分野でも、わが国ソフトウェア企業には、今世紀に入って、新たにソフトウェアファクトリを設立し、事業化している企業が複数存在する [NIKKEI06]。苦しむ事態にならないよう、十分な政策と戦略を立てて実施すべきである。

また、ソフトウェア・アウトソーシングのための手段のひとつとして、ソフトウェアファクトリを日本国内の地方や海外に分散して設立する方法も検討されている。

６．むすび

ソフトウェア事業が対象とすべき問題領域は、組込み系やエンタプライズ系だけではない。INCOSE がカバーしようとしている広い領域（宇宙、地球環境、防衛、福祉、教育、行政など）を視野に入れ、市場を開拓することによって、さらに大きな利益創出を目指すようにしなければならない。広い視野から見た、ソフトウェアファクトリの計画を望みたい。

参考文献

[Aaen97] Aaen, I, et al., The Software Factory: Contributions and Illusions, in Proceedings of the Twentieth Information Systems Research Seminar in Scandinavia, Oslo (1997)

[Baldwin00] Baldwin, C.Y. and K.B. Clark, Design Rules, The MIT Press (2000)

[Basili89] Basili, V. R., The Experience Factory: Packaging Software Experience, Proceedings of the 14th Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt MD. (1989)

[Clements02] Clements, P., et al., Software Product Lines, Addison-Wesley (2002)

[Cusumano91] Cusumano, M.A., Japan's Software Factories, Oxford University Press

[8] これに対応して、日本では JASPER (Japan Automotive Software Platform Architecture)が組織されている。

(1991)

[Czarnecki04] Czamecki, K., et al., Staged Configuration Using Feature Models, Proc. Software Product Line Conf. (2004)

[Eppinger91] Eppinger, S.D., Model-based Approaches to Managing Concurrent Engineering, Journal of Engineering Design 2(4) (1991)

[GOF95] Gamma, E., R. Helm, R. Johonson and J. Vlissides, Design Patterns, Addison-Wesley (1995)

[Greefield04] Greenfield, J. et al, Software Factories, Wiley Publishing (2004)

[Habermas84] Habermas, J., The Theory of Communicative Action: Reason and Rationalization of Society, Vol.1, Boston Press (1984)

[INCOSE04] INCOSE/Systems Engineering Handbook, International Council on Systems Engineering, INCOSE-TP-2003-016-02, Version 2a, 1 June 2004

[Kitano02] Kitano, H., System Biology: A Brief Overview, Science, 295:1662-1664 (2002)

[北野 01] 北野宏明・編、システムバイオロジの展開、シュプリンがー・フェラーク東京 (2001)

[Krishnakumar06] Krishnakumar, B., et al., Developing Applications Using Model-Driven Design Environments, IEEE Computer, 39(2), pp.33-40 (2006)

[Matsumoto81] Matsumoto, Y., SWB System: A Software Factory, in H. Hunke (Ed.): Software-Engineering Environments, North-Holland, Amsterdam (1981)

[Matsumoto87] Matsumoto, Y., A software factory: An overall approach to software production, in "Software Reusability" ed. by P. Freeman, pp.155- 178, IEEE Computer Society (March 1987)

[Matsumoto92a] Matsumoto, Y., Toshiba Software Factory, in "Modern Software Engineering, P.A. Ng and R.T. Yeh (eds.), pp.479-501, Van Nostrand Reinhold (1990)

[Matsumoto92b] Matsumoto, Y., Japanese Software Factory, Advances in Software Science and Technology, Vol.4, Japan Society for Software Science and Technology, pp.21-42, Iwanami Shoten, Tokyo (1992)

[Matsumoto94] Matsumoto, Y., Japanese Software Factory, in "The Encyclopedia of Software Engineering", J.J.Marciniak (ed.), 1st Edition, pp.593-605, John Wiley & Sons, New York (1994)

[Matsumoto06] 松本吉弘、ソフトウェアファクトリにおける LAP の適用:「分かる化」へ向けて、情報処理学会・ソフトウェアエンジアリングシンポジュウム 2006 予稿集 (2006)

[McCue78] McCue G.M., IBM's Santa Teresa Laboratory – Architectural Design for Program Development, IBM System J., Vol.7, No.1, pp. 4-25 (1978)

[McIloy69] McIlroy, M. D., Mass-Produced Software Components, in "Software Engineering Reports" on a Conference Sponsored by NATO Science Committee,

Brussels (1969)
[MSJ06] マイクロソフト株式会社、ソフトウェアファクトリーズ構想ホワイトペーパー、Draft 1.0 (2006)
[NIKKEI06] 日経 SYSTEMS 編集部、日経 SYSTEMS、2006 年 9 月号
[NTTdata06] 見える分かるシステム開発ープロセス透明化のススメ、株式会社 NTT データ (2006)
[Parnas72] Parnas, D.L., A Technique for Software Module Specification with Examples, Comm. ACM, 15(5), pp.4.30-4.36 (1972)
[SEC06] 長岡良蔵、プロジェクト見える化とは、SEC journal, Vol.6, pp.28-29 (2006)
[Searls69] Searle, J. R., Speech Acts – An Essay in the Philosophy of Language, Cambridge University Press (1969)
[Stelling06] Stelling, J., et al., Complexity and Robustness of Cellular Systems, in Szallasi, Z., J. Stelling and V. Periwal, Eds., System Modeling in Cellular Biology, The MIT Press (2006)
[Sztupanovits97] Sztupanovits, J., et al., Model-Integrated Computing, IEEE Computer, 30(4), pp.110-112 (1997)
[Winograd97] Winograd, T., The Design of Interaction, P. Denning and B. Metcalfe, Eds., Beyond Calculation, Springer-Verlag (1997)