高生産開発言語環境の考察

Consideration of High Production Development Language Environment

石 田 政 海

要 約 システム要求機能は多種多様化・複雑化・肥大化の傾向があり，結果として惹起されるソースコード量の増加は単純にプログラミングコストの増加だけではなく，テスト工数の増加，即応性・品質の低下を引き起こしている．マネジメント技術による標準化や部品化再利用推進策は有効ではあるが，機構的な強制力を伴わないため効果に限界があり何らかの技術転換が期待されている．

本稿では，開発言語環境にユースケースやドメインといった領域概念を取り入れ，その領域で必要となるコンピュータ資源とその関連や状態などの設計情報を前提とすることにより標準化や部品化再利用を強制させ，簡易言語によるプログラム開発を可能とする高生産開発言語環境を考察する．

Abstract Systems requirements tend to be diversified, complicated and enormous, which cause the increase in quantity of source code lines, which may cause not only the increase in programming cost simply, but also the increase in test work load and the decrease in adaptability and quality. Because standardization and reuse of software components by management technology do not accompany systematic powers of compulsion, their effects are limited and the certain technology change is expected.

This paper discusses the high-productivity development language environmment that forces programmers to follow standardization and reusability of software component, and write programs in a simplified programmig language. The development language environment includes the concept of a domain called ‘ use-case ’and‘ domain ’, which decides the computer resources required in that domain.

In this report, I insist on taking in a domain concept called USECASE and DOMAIN in development language environment. The domain decides the computer resources which oneself needs. And a domain can define design information such as relation between computer resources or a state.

By them, development environment gets possible to let a programmer force standardization and reusability of a component.

I consider high production development language environment to make program development by a simplified language possible.

1. はじめに

金融機関向け基幹系システムへの要求機能は多種多様化・複雑化の傾向があり，それらを実現するシステム規模も肥大化している．ソースコードの増加は，単純なプログラミングコストの増加だけではなく，テスト工数の増加，新商品および新サービスへの即応性低下，品質低下などの原因となっている．

ソフトウェア開発における生産性と品質の向上はソフトウェア工学上重要な課題となっており，改善策はプロジェクト管理技術や生産技術として開発され実務に適用されているが，局所的な工程への改善効果はあるものの期待以上の成果はあがってはいない．ソフトウェアの部品化再利用もいろいろな技術が提唱されてはきたが，工業製品における部品化ほどは成功してい

ない．

見かけ上のソースコードを減らすことは簡易言語と同様に生産性向上が期待できる．可読性が向上し，テスト工数が削減できる．開発環境は開発環境を提供する側の判断で仕様が決定されることが多いが，使い手側であるプログラム開発者から見た利便性と効果視点が重要である．部品化再利用を推進して工業製品並みの品質を追求するためには，自由度を機構的に制限できることが重要となる．

本稿では，開発言語や開発環境の本質を明らかにすることによって，課題に対する一つの解法として高性能開発言語環境を考察する．

上位目的は，プログラム開発において可能な限り業務要件だけの記述を可能にし，開発環境や開発言語によって生成可能なものは徹底的に自動生成して，生産性を飛躍的に向上させることにある．

システム要件としては次の 4 点をあげておく．

① 記述するソースコード量を削減する
② 人間に負荷をかけるのではなくコンパイラや開発環境に負荷をかける
③ 部品化再利用を機構的に強制する
④ 実装する際の自由度を機構的に制約する

2. 開発言語や開発環境における課題

手続き型開発言語（以降開発言語と略す）はいろいろな目的や領域分野で使用できるようにするため，固有の業務処理に依存しない汎用的な言語構成要素を備えている．

開発言語の基本構造は歴史的にノイマン型コンピュータをベースとした主記憶空間にプログラムを内蔵する基本構造をベースとしており，データ構造とデータを操作するプログラム制御構造の二つから成り立っている．

計算機の基本構造や制御構造が大前提となっているため，開発言語の構成要素は計算機の構成要素に写像しやすいように設計されている．言い換えると計算機には優しく，それを使う人間には優しくない言語となっている．たとえば IF 文とか WHILE 文などのように，文には予約語が先頭に現れる．以前は代入文にも予約語 SET や LET が必要であった．このように言語はコンパイルするための構文解析が容易に実現できるような文法となっていて，プログラミング開発者である人間の作業負荷を軽減させるという構文規則にはなっていない[1]．

さらに，業務機能要件の理解に加えて，データベースや通信といったコンピュータ資源の利用技術，標準化規約などいろいろな情報を熟知しておく必要があり，プログラミングには特殊な技能が必要であると考えられている．

これらのことが生産性や品質の向上，期間の短縮，トータルコストの削減といったソフトウェア工学上の諸課題に対する原因や解決への阻害要因となっている．主な課題をあげておく．

・柔軟な言語表現力による不均質増加
・標準化や部品化再利用を強制できずコスト増加，品質リスクの増加
・ソースコードの増大による品質劣化とテスト工数増大
・システム基盤や業務部品再利用時などの前提知識を理解するコストの増大
・設計情報とソースコードの乖離

課題の解決を目指してプロジェクト管理や品質管理，生産技術が検討されてはきたが，汎用

的かつ自由な表現が可能な開発言語を前提とした開発環境下でいくら規則や標準化によって制約しようとしても，機構的に強制させることができないため，成功しているとはいえない．

仮りに成果があったとしてもソフトウェアライフサイクル全体ではなく，ある特定の工程に限定した改善にとどまっているというのが現実である．

3. システム化の表現モデル

オブジェクト指向技術の普及によりモデル化技術が着目されるようになっている．初期のオブジェクト指向言語である Smalltalk が，オブジェクトというモデル概念を導出することによって実世界をそのままコンピュータ世界へ写像表現することを可能とし，その後のオブジェクト指向言語に影響を及ぼしている[2][3]．

モデル化技術は，複雑な実世界を抽象化または単純化する技術であり本質的な特徴を理解するのに有効である．また参照モデルとして位置づけることにより異なるモデルを比較することに役立つ．

開発言語の特性に起因する課題の解法を考察するには，開発言語そのものや前提となるコンピュータシステムのモデルを明確にすることが重要である．

3.1 ノイマン型コンピュータ構造モデル

ノイマン型コンピュータとは，プログラムをデータとして記憶装置に格納し，これを順番に読み込んで実行するコンピュータであり，この方式はストアドプログラム方式と呼ばれている．この構造は計算機動作に関する基本構造でありハードウェア概念とソフトウェア概念を分離した．COBOL や FORTRAN などの開発言語だけではなく，現存するほとんどの開発言語はノイマン型コンピュータのデータ構造と動作（以降 ノイマン型コンピュータ構造モデル と称する）に影響を受けている．ノイマン型コンピュータ構造モデルは，「アルゴリズム（データ操作）を逐次実行して，データ領域を操作するという論理的な固まり」から構成される．

言語コンパイラはそれぞれの言語構文規則とセマンティックに従い，ソースプログラムをターゲットとなる機械語～つまりアルゴリズムとデータ領域～に変換する．

3.2 設計と実装の分離

ノイマン型コンピュータ構造モデルでは，データ領域とデータ操作（アルゴリズム）が，そのまま高級言語に写像表現されているため，設計者や実装者はそれらを論理的なプログラム単位としてとらえている．

論理的なプログラム単位は構造化プログラミング技法による機能という視点から分割されるが，分割単位には明白な基準がない．またプログラム単位すなわち 1 本の COBOL プログラムの内部においても，データ領域とデータ操作，外部プログラム呼出などが自由に記述可能であり，部品化再利用の阻害要因となっている．

設計と実装を分離するには，実装時における自由度を設計時において制約することが必要である．設計と実装の役割境界は次のようになる．

① プログラム単位および機能の決定は設計の専権
② プログラム単位内部のデータ領域の宣言または定義は設計の専権
③ 外部プログラム再利用の決定は設計の専権

④　設計によって制約されたデータ領域（含む外部プログラム呼出に関わる操作およびデータ連携）に対するデータ操作のみが実装の専権

ここで①から③の成果物は設計情報であり，データディクショナリやリポジトリに蓄積・管理される．現在の開発環境でもリポジトリは実用化されてはいるが，開発言語のコンパイラからは独立または分離して存在している．つまり設計成果物の蓄積情報としては活用しているが，コンパイラへの入力成果物とはなっていない．

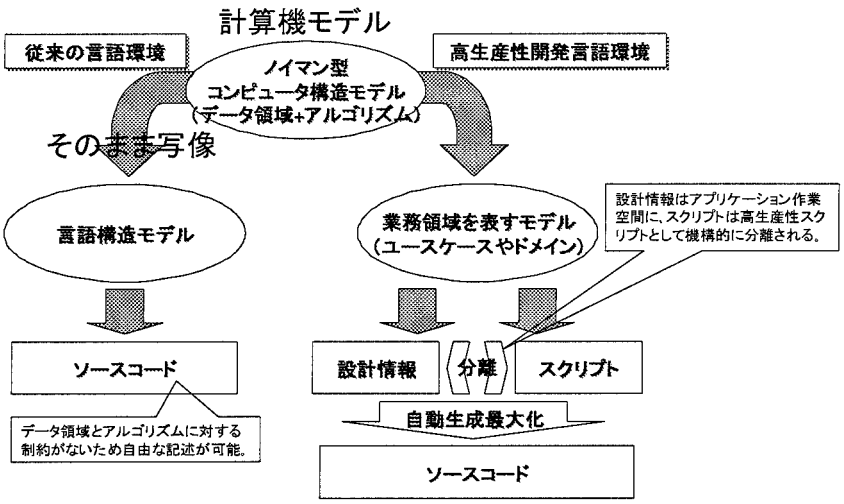

図 1　設計情報とスクリプトの分離

本稿の高生産性開発言語環境（図１の右側）では，ユースケースやドメインと言った業務領域を表す概念を論理的なプログラム分割単位とすることで，設計情報とスクリプトを分離させることが容易となる．ビジネスロジックはスクリプトによって記述し，設計情報とマージすることで最終的なソースコードを生成することできる．

次章以降では，データ領域に対してはアプリケーション作業空間という概念，アルゴリズムに対しては高生産性スクリプト概念を導入することによる高生産性実現の機構を考察する．

4.　アプリケーション作業空間

ある特定の業務処理をコンピュータ上で動作させる論理的な空間をアプリケーション作業空間と定義する．従来のアプリケーション作業空間は，ノイマン型コンピュータ構造モデルに最適化されている言語構成要素に影響を受けているため，コンパイル単位や実行単位としてのソースコードファイル概念，または論理的なプログラムやモジュール概念にマッピングされている．実行時にはコンピュータメモリ空間として実現されるため，アプリケーション作業空間の状態は初期化状態または不定といった未定義な状態であり，アプリケーションが使用するデータ領域やコンピュータ資源をアルゴリムが明示的に準備する必要がある．

アプリケーション作業空間のマッピング基準をコンピュータの動作モデルに合わせるのではなく業務領域概念に合わせること，またアプリケーション作業空間の準備作業を軽減することが重要である．

4.1 業務モデリング指向

UML[*1] 以前のオブジェクト指向開発方法論では，モデル表現としてのオブジェクトを強調したため，業務処理とオブジェクトとの関係が不明確だった．

例えば OMT（Object Modeling Techinc）ではソフトウェアの構造を静的モデル，動的モデル，機能モデルに 3 分割して表現するが，機能モデルでは表記法として DFD（データフローダイアグラム）を採用している．オブジェクト指向の本質的な成果物としてのクラス定義に展開される静的モデルや動的モデルは必須成果物であるが，機能モデルはクラス導出とメッセージ連鎖による機能実現を証明するための補完的な役割にとどまっている[4]．

ヤコブソンが導出したユースケース概念や UML におけるドメイン概念は業務分析とオブジェクト分析を橋渡しするために有益である．金融ソリューション事例を参考にすれば，開発領域は預金，融資，為替という単位で大きなドメインに分割される．また取引パターンとしての普通預金入金というユースケースに分割され，アプリケーション制御単位やテスト検証単位として利用される．

4.2 ドメインとユースケース

工業製品とそれを構成する部品の関係を，ユースケースとクラスなどの業務部品に対応させることができる．部品を組み合わせることで製品を製造することはできるが部品そのものは中間成果物に過ぎない．最終的な製品が提供するべき機能を検証することが必要となる．

ユースケースでは再利用する業務部品が提供する機能をすべて利用するわけではなく一部分を利用する．ユースケースは業務部品の利用する範囲や目的を明確に定義することが可能である．そのためユースケース内で利用される業務部品やデータベースなどのコンピュータ資源同士の相関も明確となる．

ドメインを基準にしてドメイン内部で利用する業務部品やコンピュータ資源を導出できるが，それぞれの相関はユースケースによって異なる可能性があるため明確には定義できない．そのためアプリケーション作業空間のマッピング基準としてはユースケースが最適となる．ユースケースは業務フロー表現と親和性がよく，コンピュータシステムに詳しくない業務領域担当者にも理解しやすく，導出や評価・検証が可能である[5][6]．

4.3 資源の定義と関連

ドメインやユースケースなどの領域概念をアプリケーション作業空間へのマッピング基準に導入すると，特定領域に特化されたアプリケーション作業空間で利用する業務部品やコンピュータ資源の特定が可能となり，また業務部品や資源の相互関係も設計時に仕様として決定可能となる．従来は資源間の制約条件などをアルゴリズムとして明示的に実装する必要があったが，設計情報にあらかじめ定義しておくことで自動生成が可能となる．

設計と実装での成果物定義の境界は次の二つとなる．

・アプリケーション作業空間の範囲定義
・アプリケーション作業空間で利用する資源の定義

5. 高生産性スクリプト

ソースコード量の削減効果は，初期開発時の生産性向上と，可読性向上による保守性向上で

ある．また，テスト工数を削減することが可能となるし，移植性も向上する．

ソースコード量の削減基準は見かけ上のソースコードでよい．つまり最終的な機械語に翻訳する高水準言語コンパイラではなく，簡易言語やスクリプト言語を入力としたソースコードジェネレータであっても効果は変わらない．GNU[*2]で提供されている多様な言語，例えばオブジェクト指向言語である Eiffel は C/C＋＋ソースコードを生成するフロントプロセッサとして実現されているが，生産性の判断は Eiffel のステップ数を基準にしている[7]．

生産性を向上させるためには，スクリプトの記述自由度を制約することと自動生成率を向上させる機構の導入により記述量を少なく抑えることが重要である．

5.1 設計情報とスクリプトの分離

データ領域とアルゴリズムの分離と同様，設計情報とビジネスロジックも分離させる．

設計情報の管理要素は大きく二つある．

① データ構造要素 （データベース，ファイル，テーブルなどデータ項目の構造体）

② 関数要素（再利用する業務部品や前提となるミドルウェアが提供する関数群など）

二つの要素を管理することにより，スクリプトからデータ項目を参照した場合に，そのデータ項目がどのデータベースに定義されているか特定できる．また業務部品であれば，関数名，その関数の戻り値，入出力引数を定義しておくことにより，その関数を呼び出すために必要なデータ項目や，呼び出したあと値が返されるデータ項目も特定できる．

スクリプトでの関数呼出時に，指定した関数名により関数が資源要素内で一意に決定する場合にはスクリプトでは引数の並びや戻り値の代入などを省略できる．例えば，「普通預金．入金（）；」の様に実引数を省略記述すれば関数の引数仕様に変更が生じてもスクリプトレベルでは影響を受けないメリットを享受できる．ビジネスロジックを記述するスクリプトと，コンピュータ資源やミドルウェアおよび業務部品のインタフェース定義などを分離することにより抽象的な呼出制御の記述が可能となる．

5.2 資源操作処理（たぐり寄せ）の自動生成

従来の言語では，例えば資源に含まれるデータ項目としてデータベースレコードにのみ存在するデータ項目を参照する場合，前もってデータベースレコードを読み込み，バッファへ転送するという本来のビジネスロジック以外の記述が必要であった．しかしアプリケーション作業空間に使用する資源があらかじめ準備されているという前提にたてば，スクリプトはビジネスロジック記述に専念できる．データベースやファイルといったコンピュータ資源の存在や特性差をスクリプト作成者が意識しなくても良いように，データのたぐり寄せのような資源操作処理を自動生成すれば見かけ上のソースコードは削減できる．

ワーク領域の宣言や，関数を呼び出すための引数領域や戻り値の領域の宣言は設計情報から自動生成できるし，データベースの状態を管理すれば適切な箇所に OPEN や SELECT 指令を自動生成できる．

5.3 業務領域に最適化された開発言語

開発言語は汎用的であるという常識があるが，利用者の立場に立つと汎用的である必要はない．むしろリアルタイム処理やバッチ処理，データベース更新処理，メッセージ処理などさま

ざまな処理形態の特性差を，言語構成要素を駆使して実装するのは非効率的である．開発生産性向上には，処理形態に最適な開発言語を用意するか，または開発言語が適用領域に最適化できる機構が求められる．

6. 高生産性開発環境コンセプト

6.1 システム全体構造

ソフトウェア開発領域を限定することで，データ定義やコンピュータ資源の準備などの宣言および定義と，処理アルゴリズムを分離させる．領域におけるデータ操作に制限を与え，データベースやコンピュータ資源に対する操作を自動生成することで，見かけ上のソースコードを減らすことができる．それによって生産性の向上やテスト工数の削減および品質の向上を実現すること，また人間に優しい開発言語とそれを支援する開発環境を提供することでソフトウェア開発における前提条件の理解や定型的な作業を軽減させ，本質的なビジネスロジック記述に注力できる新たな高生産性開発環境コンセプトを紹介する．

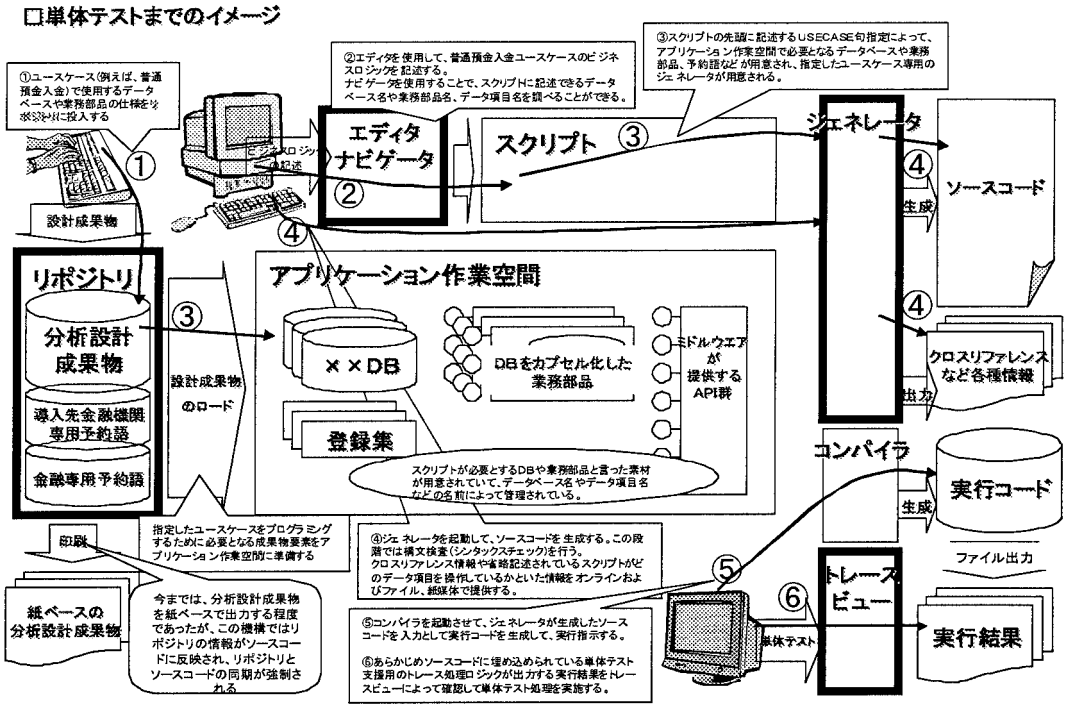

図 2　高生産性開発環境イメージ図

図 2 では単体テストまでのイメージを表している．プログラムで必要なデータ領域はアプリケーション作業空間にあらかじめ用意される．スクリプトはアプリケーション作業空間に用意されたデータ領域を前提にして可能な限りビジネスロジック記述に専念することができ，アプリケーション作業空間とスクリプトからソースコードが生成できる．

アプリケーション作業空間が広い業務領域にマッピングされていると，必要となる資源が多くなり，相互関係も複雑化してしまい，スクリプトで関連や制約条件などを記述しなくてはな

らなくなる．領域概念とくにユースケース概念を導入することでアプリケーション作業空間の大きさを限定することが可能となり，関連や制約条件も設計情報としてリポジトリでの管理が可能となり，自動生成率を上げることができる．

スクリプトではビジネスロジックに専念したいため，データ項目がどのコンピュータ資源に保存されていて，どのようにたぐり寄せればいいのかには本来関心がない．そこでアプリケーション作業空間ではデータ項目を識別名（名前）で管理することでスクリプトとの結びつけを可能とする．ユースケースと言った閉じた領域であればあらかじめ名前が衝突したり曖昧にならないような用語の標準化は可能である．

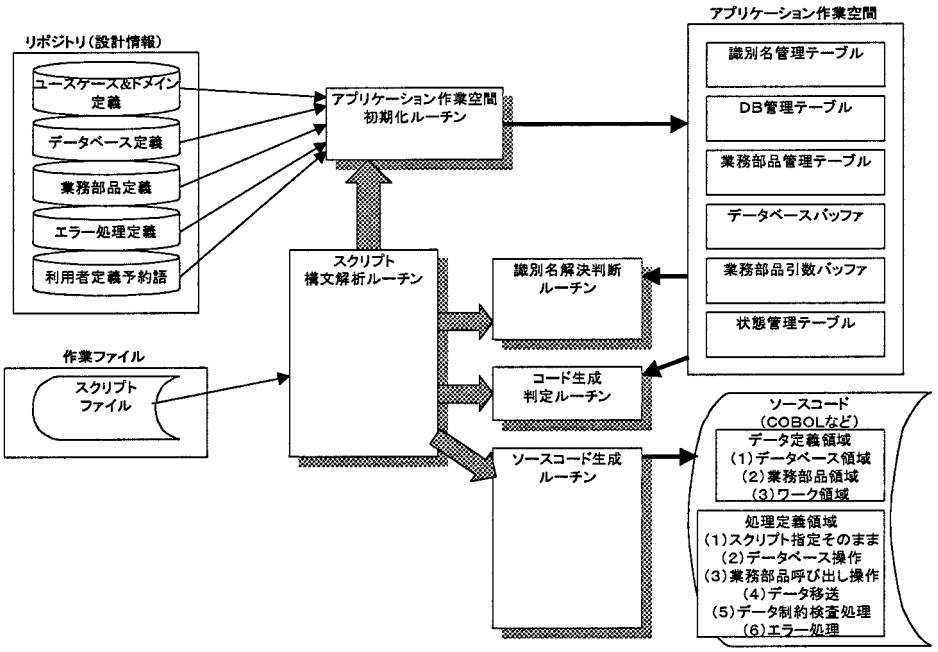

図 3　制御構造とテーブル構造

次に高生産性開発環境を具体化させるため制御構造とテーブル構造を図 3 に示し，処理の概要を説明する．

なお事例としては「コンピュータのコンソールまたは端末から，普通預金の口座番号を入力することで口座情報と顧客情報を表示する」を考える．

6.2　スクリプト

スクリプトとしては次の様な簡単な記述から実行可能なソースコードを生成することが可能である（図 4）．

```
Domain 普通預金;
Accept  口座番号;
Print 氏名、口座番号、顧客番号、残高;
```

図 4　簡単なスクリプト例

通常のコンパイラではデータ領域とアルゴリズムがスクリプトとして記述されていなければコンパイルできない．ここで記述されているのは分離されている設計情報を除いたビジネスロジック部分だけとなっている．四角で囲まれた部分がアプリケーション作業空間に相当しており，この空間で必要になるコンピュータ資源は暗に用意されているという前提でスクリプトが記述できる．ここの例では1行目の Domain 指定によって設計情報が特定可能となる．3行目のデータ項目をどのデータベースから読み込まなければならないかというたぐり寄せの記述も省略可能である．

6.3 リポジトリに登録される設計情報

アプリケーション作業空間として，ユースケースまたはドメインを定義する．ドメインで使用するコンピュータ資源（たとえばデータベース資源や業務部品資源など）と，それらが複数ある場合には関連情報，またワーク領域が必要であれば登録する（表1，表2）．

表 1　ユースケース＆ドメイン定義

ドメイン名	普通預金		
使用する資源名	顧客データベース 普通預金データベース		
関連名	口座保有	顧客データベース	普通預金データベース
		多重度　1	多重度　0．．N
ワーク領域	全資源の和集合を自動生成 データ項目の重複を排除 自動的に付加するプリフィックス指定は WRK		

表 2　データベース定義

データベース論理名	顧客データベース	
データベース物理名	customer.db	
データ項目名	データ型	桁数
顧客番号	文字列型	10桁
氏名	文字列型	20桁
住所	文字列型	60桁
生年月日	日付型	
電話番号	文字列型	12桁
データベース論理名	普通預金データベース	
データベース物理名	ftm.db	
データ項目名	データ型	桁数
口座番号	文字列型	7桁
顧客番号	文字列型	10桁
残高	整数型	15桁

6.4 識別名管理テーブル

アプリケーション作業空間初期化ルーチンは，スクリプトに記述されたドメイン名“普通預金”をキーとしてリポジトリからソースコード生成に必要な設計情報を抽出して，メモリ空間上にアプリケーション作業空間領域を用意する．

特に重要なのは識別別名管理テーブルであり，スクリプトで参照可能なデータ項目名に関する情報とソースコード生成タイミングに応じたデータ項目の状態を管理している．コード生成判定ルーチンは識別名管理テーブルを参照することでコード生成の可否判断やデータベースなどの外部資源に固有な操作，例えばデータベースの OPEN や SELECT 文，データベースバッファからプログラム領域へのレコード転送処理を自動生成する．

表 3　識別名管理テーブル

データ項目名	定義されている資源名	状態	値
顧客番号	顧客データベース	未初期化	
	普通預金データベース	DBからロード済	
氏名	顧客データベース	未初期化	
住所	顧客データベース	未初期化	
生年月日	顧客データベース	未初期化	
電話番号	顧客データベース	未初期化	
口座番号	普通預金データベース	外部入力済	
残高	普通預金データベース	DBからロード済	

識別名管理テーブルのデータ項目状態が，アプリケーションから見て意味を持たない値を表す“ 未初期化 ”の場合は，値を参照する右辺値操作のソースコード生成はできない（表 3）.

「Accept　口座番号；」はデータ項目状態にかかわらず値を代入する左辺値操作であるためソースコード生成は可能である．ソースコードを生成するタイミングで状態は“ 未初期化 ”から“ 外部入力済 ”に書き換えることで右辺値操作のソースコード生成が可能なデータ項目となる.

データベースに定義されているデータ項目に参照が発生した場合には，未初期化かどうかをチェックして, 未初期化であればデータベースレコードの読み込みが可能かどうかを判定する．データベースが OPEN 状態でなければ先だって OPEN するソースコードを自動生成する．データベースからレコードを読み込むソースコードを生成したタイミングで，そのレコードに定義されているデータ項目状態を“ DB からロード済 ”に書き換えることで右辺値操作のソースコード生成可能なデータ項目となる.

スクリプトの 3 行目では，“ 氏名 ”, “ 口座番号 ”, “ 顧客番号 ”, “ 残高 ”の状態がいずれも“ 未初期化 ”ではなくなるため右辺値操作が可能となり，ソースコード生成が可能となる.

6.5　データベース直接操作から業務部品再利用への変換

データベース定義ではデータ項目情報を定義しているが，これをデータベース操作がカプセル化された業務部品として再利用する場合には関数定義に置き換わる．関数定義では，関数名，関数の戻り値，引数情報となる．引数では，データ型，桁数情報に加えて入出力指定が加わる（表 4）.

スクリプトに関しては，どの関数を呼び出すのかを明示的に記述する必要がある．但し，関数名だけで関数が一意に決定する場合，スクリプトで引数の並びを明示的に記述する必要はない（図 5）.

「業務部品資源を使用した場合のスクリプト」では，「普通預金照会」と「顧客照会」を順次呼び出せばよい.「普通預金照会」に関しては業務部資源の定義から，戻り値，口座番号，顧客番号，残高，という引数が必要であることがわかる．この事例では識別名が一意に区別でき，自動生成する事が可能であるため省略している．省略した場合には，関数における引数の並びが変わったとしてもスクリプトを修正する必要がない.

6.6　スクリプト修正を必要としない業務部品の交換

業務部品は本質的に機能改変により進化するものである．ビジネスロジックを記述する「スクリプト」と生成される「ソースコード」の間に領域における「資源の定義情報」を位置づけ

表 4　データベース操作から業務部品へ変換する場合の定義情報

■データベース資源の定義

データベース論理名	顧客データベース	
データベース物理名	customer.db	
データ項目名	データ型	桁数
顧客番号	文字列型	10桁
氏名	文字列型	20桁
住所	文字列型	60桁
生年月日	日付型	
電話番号	文字列型	12桁

データベース論理名	普通預金データベース	
データベース物理名	ftm.db	
データ項目名	データ型	桁数
口座番号	文字列型	7桁
顧客番号	文字列型	10桁
残高	整数型	15桁

■業務部品資源の定義

関数名	顧客照会		
引数名	データ型	桁数	入出力
戻り値	整数型	5桁	OUT
顧客番号	文字列型	10桁	IN
氏名	文字列型	20桁	OUT
住所	文字列型	60桁	OUT
生年月日	日付型		OUT
電話番号	文字列型	12桁	OUT

関数名	普通預金照会		
引数名	データ型	桁数	入出力
戻り値	整数型	5桁	OUT
口座番号	文字列型	7桁	IN
顧客番号	文字列型	10桁	OUT
残高	整数型	15桁	OUT

■データベース資源を使用した場合のスクリプト

```
Usecase 普通預金入金;
Accept 口座番号;
Print 氏名、口座番号、顧客番号、残高 ;
```

■業務部品資源を使用した場合のスクリプト

```
Usecase 普通預金入金 ;
Accept 口座番号 ;
普通預金照会();
顧客照会();
Print 氏名、口座番号、顧客番号、残高 ;
```

図 5　スクリプト修正イメージ

る.「資源定義」では資源の論理名と物理名との関係付けを定義しており,「スクリプト」は資源の論理名を使って記述する. 資源の物理名はソースコード生成時に展開されるため,「スクリプト」を修正することなしにリポジトリに格納されている資源の定義情報だけを変更することで, 資源定義の物理変更を吸収することができる（図 6).

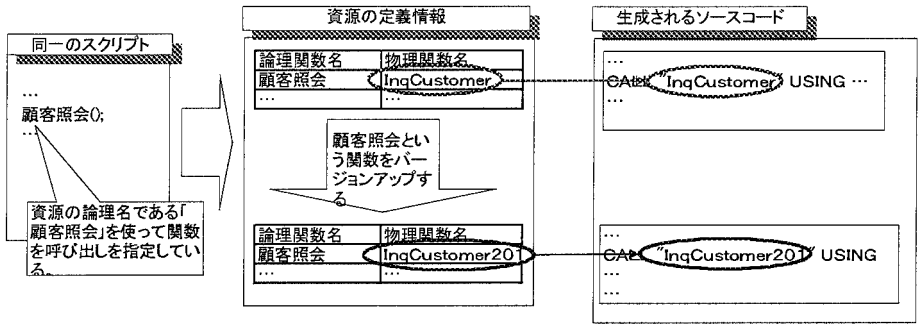

図 6　業務部品の交換例

この機能により次の二つの効果が期待できる.

① 関数やモジュールを別のものに取り替える場合

関数やモジュールなどの統廃合や単体テストまたはプロトタイプ用で暫定的に使用していたものから本番用のものに切り替える時に効果的である.論理関数名を変更せずに,物理関数名を変更することで対応可能である.

② 同じ関数やモジュールを改変する際に引数のシグニチャが変更される場合（図7）

引数が追加されて機能拡張する場合に効果的である．この機能を活用すると進化型プロトタイプを効率よく実現できる．

スクリプトでは，論理名の「普通預金照会();」として引数の並びは指定せずに記述しておく．この状態でソースコードを生成指示すると，リポジトリに定義されている論理名「普通預金照会」で定義されている引数の並び，すなわち"口座番号，顧客番号，残高"が引数として生成される(0.1 ベータバージョン).

```
CALL "普通預金照会 0.1 ベータ" USING 口座番号 顧客番号 残高.
```

このあと，普通預金照会の関数をより詳細に定義してリポジトリに同じ論理名で登録する(1.0A バージョン)．スクリプトは無修正の状態でソースコード生成指示すると，リポジトリに定義されている論理名「普通預金照会」で定義されている引数の並び，すなわち"店番，科目，口座番号，…，非課税限度金額"が引数として生成される．

```
CALL "普通預金照会 1.0A" USING 店番 科目コード 口座番号 顧客番号 担保区分
                               口座開設年月日 優遇金利 通帳証書区分 口座残高
                               通帳残高 非課税限度金額 ….
```

図 7 引数シグニチャを変更した場合の例

7. ま と め

現在の開発モデルは汎用的な開発言語が前提とするモデル，すなわちノイマン型コンピュータ構造モデルに影響を受けており，データ領域に対する表現の自由度が高い.

「業務領域概念に合わせたアプリケーション作業空間」と,「設計情報部分と実装部分を分離・マージするソースコード生成機構」を実現すればソフトウェア工学上の諸課題に対する劇的な改善効果が期待できる．主な改善効果を以下にまとめておく.

7.1 ソースコード量の削減および表現方法の均質化

高生産性スクリプトにより自動生成可能なものは徹底的に自動生成することで実現.

- 実装担当者はデータ項目名と関数名だけの記述で済むため，見かけのソースコード量が削減され可読性が高まり，テスト工数が削減でき，生産性が向上する.
- データ項目の値を得るためのデータベース参照処理等が自動化されるため，生産性が向上する.
- 指定したユースケースやドメインに必要なコンピュータ資源しか利用できなくなるため独自判断が減る．これにより表現方法の均質化が実現できる.

7.2　部品化再利用や標準化の強制

スクリプトから分離されたアプリケーション作業空間が設計情報の利用を強制することで実現.

- データ項目名など識別名が設計時に定義されるため，用語の標準化が機構的に実現できる.
- 上流設計においてデータベースや再利用する部品が定義され，実装担当者がその定義を回避する手段を持たないため部品化再利用が強制される．この部品化再利用の継続により部品品質が向上する.
- データベースや関数呼出後のデータ項目チェックなどのエラー処理も自動生成対象とすることによって統一的なエラー処理と対応漏れ防止を実現できる.

7.3　機能拡張など保守性の向上

抽象的な記述が可能なスクリプトと設計情報とをマージすることによってソースコードを生成する機構で実現.

- 実行環境基盤やミドルウェアなど業務処理以外の技術要素も自動生成対象とすることで，スクリプトからは透過（意識しなくても良い）となり，生産性を向上させることができる.
- データベースの保守処理やバッファ間のデータ転記処理が自動生成されるため，データ項目追加やデータベース仕様の再編成があってもスクリプトへの影響や修正作業を最小限に抑えることができる.
- 業務処理部品のインタフェースや引数の並びをリポジトリで登録しておくため，呼出処理が自動生成できる．この結果，引数の追加などのような仕様変更があっても，スクリプトへの影響や修正作業を最小限に抑えることができる.

7.4　分析および設計からみた要求管理の向上

設計情報がアプリケーション作業空間を経由してソースコードへ強制反映される機構により実現.

- 設計と実装との成果物分離と役割分担が強制される．これにより上流工程における下流工程の自由度を制約することで要求管理精度が向上する.
- ソースコードがスクリプトとリポジトリの設計情報から生成されるため，設計情報とソースコードの内容が完全に同期する．これによってソースコードではなく設計仕様やスクリプトだけで機能改善を行うことが可能となり，生産性や保守性が向上する.

8.　お　わ　り　に

今回紹介した高生産性開発環境のコンセプトは既存の情報技術を組み合わせることで実現可能である.

SBI 21（地域金融機関向け次世代勘定系システムパッケージ）ではオブジェクト指向分析技術を導入して開発しており，ユースケースのシナリオ記述やシーケンス図を活用している．実現可能性と現実的な効果を高めるためには，例えば勘定系アプリケーションのソースコードを入力素材として，ソースコードの自動生成率を向上させる処理パターンの洗い出しや処理の正

規化，標準化を試行してジェネレータの仕様として組み入れることが重要であり，さらなる研究やプロトタイプの試作による検証作業が必要である．

また狭い意味で開発環境の改善だけに注目するのではなく，開発・運用・保守といったライフサイクルを意識したトータルコストの削減が目的であることを自戒としたい．

最後に，今回紹介したアイデアやシステム構造は既に特許出願済みであることを付け加えておく．

＊1　UML【Unified Modeling Language】
オブジェクト指向のソフトウェア開発における，プログラム設計図の統一表記法．

＊2　GNU【GNU is Not Unix】
UNIX 互換ソフトウェア群の開発プロジェクトの総称．

参考文献

［1］ 中田育男, コンパイラ, 産業図書, 1981.
［2］ 加藤木和夫, Smalltalk/V によるオブジェクト指向プログラミング, 日刊工業新聞社, 1990.
［3］ Patric Henry Winston, ウィンストンの Smalltalk, アジソンウェスレイパブリッシャーズジャパン, 1999.
［4］ James Rumbaugh, 羽生田栄一　監訳, オブジェクト指向方法論 OMT, 1992 年 7 月.
［5］ Martin Fowler, UML モデリングのエッセンス−標準オブジェクトモデリング言語入門第 2 版, 翔泳社, 2000.
［6］ Ivar Jacobson & James Rumbaugh & Grady Booch, UML による統一ソフトウェア開発プロセス−オブジェクト指向開発方法論，翔泳社，2000.
［7］ Mike Loukides & Andy Oram, GNU ソフトウェアプログラミング−オープンソース開発の原点, オライリー・ジャパン, 1999.

執筆者紹介

石　田　政　海（Masami Ishida）
1985 年立命館大学法学部卒業．同年日本ユニシス(株)入社．統合 OA システム開発，SWIFT CBT，勘定系システム制御系開発，オブジェクト指向技術適用の企画推進，SBI 21 PC 開発環境の開発に従事．現在，金融第一事業部金融企画推進部企画二室に所属．