オブジェクト指向技術による勘定系システムのリエンジニアリング

Re-engineering the Banking System with the Object-oriented Technology

向　井　　丞

要　約　現在，日本ユニシスでは地域金融機関向け次世代基幹勘定系パッケージ SBI 21（Strategic Banking Integrated-system for 21 century）の開発を行っている．SBI 21 は数々の特長を持っているが，開発言語が COBOL でありながら，分析/設計工程にオブジェクト指向技術を採用しているというのもその特長の一つである．

このシステムをオブジェクト指向技術で構築するに際して，本開発では，先に販売し豊富な導入実績を持つ TRITON と FAST 1100 の二つのパッケージのプログラムソースコードとデータベース項目を入力としてリエンジニアリングするという手順を採っている．

本稿では，オブジェクト指向技術による分析/設計結果の COBOL による実装構造を紹介するとともに，既存システムをオブジェクト指向技術でリエンジニアリングする際の手順/利点/考慮点を記述する．

Abstract　Currently, Nihon Unisys develops the next-generation banking system package for the regional banks, SBI 21 (Strategic Banking Integrated-System for 21 century). SBI 21 has a number of features; one is that the object-oriented technology is adopted in the analysis and design phases and COBOL is used as the programming language.

SBI 21 is re-engineered with the object-oriented technology on the basis of the source programs and data items from two existing package systems, TRITON and FAST 1100, which are installed in many financial institutions in Japan.

This article introduces the application structure implemented by COBOL as a result of object-oriented design and analysis, and also describes procedures, advantage, and considerations of re-engineering the existing system with the object-oriented technology.

1. はじめに

日本ユニシスでは，今までにも地域金融機関向け勘定系システム・パッケージとして，SYSTEM-F，FAST 1100，TRITON というパッケージを開発し提供してきた．そして，今回，日本版ビッグバンに対応すべく，それらの後継パッケージとして，1996年4月より，SBI 21（Strategic Banking Integrated-system for 21 century）の開発に着手した．

開発に際しては，以下の方針で行うこととした．

1）開発工数の削減，及び，機能の網羅性の確保という観点から，新たに一から開発するのではなく，既存のパッケージ資産を活用し，それをリエンジニアリングするという方法を採る．

2）リエンジニアリングに際しては，分析設計工程にオブジェクト指向技術を採用する．

本システムに限らず，現在，種々の要因から再構築の検討を迫られているシステム

があろうかと思う．その際，そのシステム規模が大きい場合は，コスト/リスクとも非常に大きくなるため，その再構築の方針決定を委ねられた立場のシステム・エンジニアにとっては，苦悩の日々の始まりとなる．特に，オープン化に代表される近年の情報技術の多様化に伴う選択肢の急激な拡大が，その苦悩に拍車をかけている．

本稿は，上述した立場のエンジニアの方々に対して，SBI 21 の開発を大規模システムの再構築事例として提示するという観点から記述するものである．従って，SBI 21 というパッケージの紹介を意図するものではない．本稿が既存システムの再構築をお考えの方々の参考になれば幸いである．

2. オブジェクト指向技術の採用に関して

2.1 その採用理由

現在，多くの金融機関の勘定系システムは第三次オンライン・システムと呼ばれたシステム構築時期から 10 年以上を経過し，そのシステムは複雑化/肥大化し保守の限界に達しつつあると言われている．かつ，日本版ビックバンによる大競争時代を迎え，他の金融機関との差別化のために多種多様のサービス/商品の迅速な提供を図る必要のある金融業界において，それを支える勘定系システムは今後とも益々複雑化/肥大化していく傾向にあると予想される．従って，勘定系システムの再構築に際しては，この傾向に歯止めがかけられる構造の実現ということが重要なポイントとなる．そのためには，複雑化したシステムを一旦一つ一つのシンプルな構成要素に分解し，再度それを組み合わせシステムとしてまとめあげるという考え方を採る必要がある．

一方，オブジェクト指向技術では，まず役割と責任という観点から「オブジェクト」を導出するということから始め，次にそのオブジェクト間の関連を定義するという手順を辿る．この手順は，まさしく上記の考え方に合致する．これが分析設計工程にオブジェクト指向技術を採用した理由の一つである．

更に，今後ともその機能範囲が拡大傾向にある勘定系システムを必要都度全面更改するというのでは，コスト/リスク両面からいずれ限界が訪れるであろうことは想像に難くない．これに対応するためには，あらかじめ部分的/段階的に更改できる構造にしておくということがアプリケーション構造決定の際の重要なポイントとなる．それに対してもオブジェクト指向技術でいうところの「カプセル化」という考え方が最適であるということも採用した大きな理由である．

2.2 開 発 言 語

オブジェクト指向技術を採用したことにより，常識的に考えれば開発言語にはオブジェクト指向技術の支援機能を持つ言語を採用することとなる．その代表的な例が C＋＋であろう．しかしながら，勘定系システムが大規模システムであり，かつ，極めて公共性の高いシステムであるということを考えると，開発言語選定に際しては，オブジェクト指向技術を支援する機能を有しているか否か以前に以下のことを考慮する必要がある．

1） 大規模システムであることを考えると，将来の保守性が重要なポイントとなるため，可読性に優れた言語であること．更に，勘定系システムは事務処理中心のシステムであることから，作表機能にも優れていること．

2)　全金融機関がほぼ同時期に対応せざるを得ない案件(例：古くは税制変更対応，最近では2000年対応，デビットカード等）が発生し，一時的にプログラマに対する需要が集中することがあることから，外部要員も含め有知識者の多い言語であること．

3)　激変する昨今の情報処理技術を考えると，今後とも何が主流になっていくか不透明であるため，開発言語は特定ベンダーに依存する固有言語ではなく，国際規格が存在し，全プラットホーム（メインフレーム/UNIX/PC）で提供されている言語であること．

以上のことを考慮し，最終的に開発言語にはCOBOLを採用した．なお，既存勘定系システムがCOBOLで記述されている場合が多いというのも決定に際しての大きな理由である．

COBOLはオブジェクト指向を支援する言語機能は有していない．それに対しては，本システムが提供する開発環境と後述するアプリケーション構造とで補うものとした．なお，開発環境に関しては，本書掲載の「SBI 21におけるオープン環境を利用した開発基盤」を参照されたい．

3.　SBI 21のアプリケーション構造

アプリケーション構造を決定するにあたって，最も重視したのは，オブジェクト指向技術の「カプセル化」という考え方に基づく部分的/段階的に書き替えることができる構造の実現である．また，それ以上に意識したのは，オブジェクト指向技術の採用はあくまで上記目的達成のための手段であって，オブジェクト指向技術で勘定系システムを構築すること自体が目的そのものではないということである．従って，オブジェクト指向技術がそぐわない部分に関してまで無理に適用するということはしていない．

3.1　全　体　構　造

SBI 21のアプリケーション構造は図1に示す通りである．

まず，勘定系システム全体を約100個のサブシステム（図1の業務サブシステムA，B…n）に分割してある．このサブシステムとは論理的なものであり，一つのサブシステムの中には複数のオブジェクトが定義されている．このサブシステムという考え方を導出するに至った経緯は3.2節に記述する．サブシステムの単位としては，例えば顧客，普通預金，定期預金，証書貸付という単位である．これらのサブシステムを制御するために取引ナビゲータと呼ぶ構成要素を導出している．

取引ナビゲータとサブシステムの関係を，定期預金を解約し，その元金/利息額を普通預金に入金するといった処理を例にとって以下に説明する．

①　まず，取引ナビゲータが定期預金サブシステムの解約という処理（SBI 21ではこのようにサブシステムの外部から呼び出される処理をサービスと呼ぶ）を呼び出す．その際，解約というサービスで必要とする情報を入力パラメタにセットして渡す．

②　定期預金サブシステムでは解約の処理を実行し，その結果(元金，利息額等）を出力パラメタにセットして取引ナビゲータに返す．

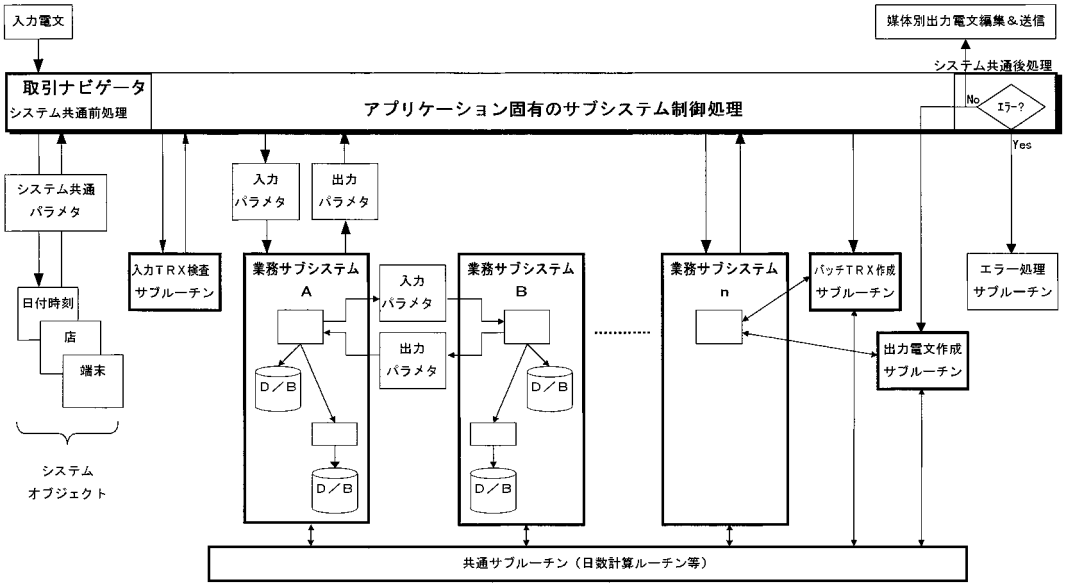

図 1　アプリケーション全体構造概念図

③　次に取引ナビゲータは普通預金サブシステムの入金というサービスを呼び出す．その際，定期預金サブシステムから受け取った元金/利息額を普通預金サブシステムの入力パラメタの入金金額にセットして引き渡す．

④　普通預金サブシステムでは入金処理を実行した後，その結果を出力パラメタにセットして取引ナビゲータに返す．

すなわち，取引ナビゲータと各サブシステムの間，及び，各サブシステム間は，入力パラメタ/出力パラメタだけで情報伝達を行っている．従って，各サブシステムの外部インタフェースである入力パラメタと出力パラメタさえ保証すれば，サブシステムの中はどのようにでも書き替え可能な構造となっており，これによって部分的/段階的書き替えが可能な構造を実現している．

3.2　サブシステム導出の経緯

当初のアプリケーション構造にはサブシステムは存在していなかった．しかしながら，実際にオブジェクト分析を行った結果でプロトタイプを作成してみると，以下のような不具合を生じることがわかった．

1)　COBOL には，オブジェクト指向を支援する言語機能はない．従って，例えばC＋＋のように，オブジェクトの属性のデータ項目単位に，それを公開する(Public 指定)，非公開にする（Private 指定）いうような設定を行い，公開すると指定した属性に関してはオブジェクトの外部から直接参照できるというようなことを行うことはできない．かつ，COBOL の言語特性を考えた場合，属性/入出力パラメタは各々一つの構造体として定義するのが最適である．このような制約下でカプセル化を実現するためには，属性は全て非公開扱いとせざるを得ない．

2)　しかしながら，属性を全て非公開扱いとし，入出力パラメタだけでデータ授受したのでは，各構造体の項目間の転記処理が増加する．特に，オブジェクト間の

関連が強く，自身の属性項目の多くを相手のオブジェクトに伝達する必要がある場合は，その転記負荷が大きい（図 2）.

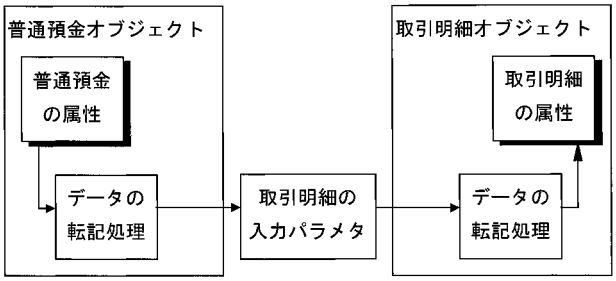

図 2　データ項目の転記処理負荷

この転記処理負荷を軽減するために，サブシステムという考え方を導出し，強固な関連を持つオブジェクト（相互にデータを参照する頻度の高いオブジェクト）に関しては，同一のサブシステム内に定義することとし，同一サブシステム内のオブジェクト間では属性を直接受け渡しできる構造とした（図 3）. ただし，渡された側は，相手側の属性に関してはあくまで参照できるだけであり，その属性に対する操作は一切できない規約となっている.

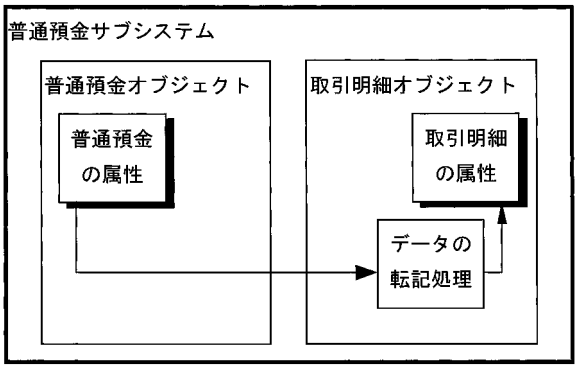

図 3　同一サブシステム内でのデータ項目の転記処理

従って，SBI 21 の場合のカプセル化の単位は，正確にはサブシステム単位と言うことになる.

3.3　カプセル化の例外事項

勘定系システムの場合，処理結果を端末画面/伝票/通帳/レシート等に表示/印字する必要があり，そのためには必要な情報を出力電文にセットして勘定系端末等に送信しなければならない. また，それらの情報の大半はオブジェクトの属性項目（例：残高）である場合が多い.

従って，単純に考えれば，出力電文を作成するための処理は，取引ナビゲータから呼び出されたオブジェクトが，その出力パラメタに出力電文で必要とする属性項目をセットして返し，取引ナビゲータがそれらの項目を出力電文にセットして勘定系端末

に送信するという処理構造となる．しかしながら，そのような処理構造を採った場合，伝票/通帳に印字するデータ項目が追加になる都度，オブジェクトの出力パラメタの構造体にそのデータ項目を追加し，かつ，オブジェクトの内部処理に対しても出力パラメタに追加されたデータ項目に対する転記処理を追加するという変更を加えなければならない．

一方，通帳はまだしも，端末画面/伝票/レシートへの出力の場合は，必ずしもビジネスルールだけで出力すべきデータ項目が決定されるわけではないという側面を持っている．例えば，普通預金の入金処理を行った際，その顧客がカードローンの契約をしているか否かをついでに伝票に出力し，カードローン成約促進のための店頭セールスの一助とするということもあり得るわけである．しかも，このような店頭セールス支援情報の類は固定的なものではなく，その金融機関の営業政策の変化によって絶えず変化するという性格のものである．

これらのことを考えると，出力電文で必要とするデータ項目をオブジェクトが出力パラメタにセットして返すという処理構造は，あまり保守性がいい構想とは言い難い．むしろ，伝票等への出力は，一つの View であるという考え方を採り，オブジェクトの役割からは切り離して考えた方がよい．

従って，本システムでは，出力電文を作成する処理に関しては，出力電文作成サブルーチンという独立したプログラムで取り扱うものとした．このプログラムに対しては，オブジェクトはその属性をそのまま引き渡す（公開する）という構造とし，その属性の中のどのデータ項目を出力電文にセットするかは，出力電文作成サブルーチン側で行うこととした．この考え方は，カプセル化という考え方には反しているが，伝票等に印字するデータ項目が追加になったとしても，オブジェクト自体は何ら影響を受けず，出力電文作成サブルーチンだけを変更すればよいというメリットが得られる．

この出力電文作成サブルーチンに類するものとしてもう一つ，本システムにはバッチ TRX 作成サブルーチンというプログラムが存在する．勘定系システムにはオンライン取引で発生した種々のトランザクションとその処理結果を一括して処理するためのバッチ処理プログラム群（例：各種日報の作成）が存在する．バッチ TRX 作成サブルーチンとは，これらのバッチプログラムに渡すデータをトランザクション発生都度出力するプログラムである．このバッチプログラムで作成する各種帳表も前述した伝票等と同様，それで必要とするデータ項目がビジネスルールだけで決定されるものではない．従って，これに対しても出力電文作成サブルーチンと同様，オブジェクト側は属性をそのまま引き渡し，バッチ TRX 作成サブルーチン側で実際に出力するデータ項目を選択するという構造を採った．

4. リエンジニアリングの手順

アプリケーション構造は決定したが，実際の開発に際しては以下のような問題が発生する．

1） 当然のことながら，設計要員には金融に関する業務知識が要求される．しかも，勘定系システムは大規模システムであり，必要とされる業務知識範囲は非常に広い．従って，業務知識を有する設計要員を相当数確保する必要があるが，常にタ

イミングよく全業務範囲を網羅できるほどの要員数を確保できるとは限らない.

2) 今回はオブジェクト指向技術という新しい考え方を採用した．従って，確保した全設計要員に対してオブジェクト指向技術の教育を実施し，オブジェクト指向技術で設計できるレベルまでスキルを身につけさせる必要がある.しかしながら，短期間に全要員をそのレベルまで持ち上げることができる保証がない.

現実は，1）の業務知識を有する設計要員の確保だけでも相当難しい．それに加えて，短期間でオブジェクト指向技術で設計できるスキルを身につけさせるということは，いかに教育を行うにしても限界がある．従って，通常のオブジェクト指向技術による開発のように，まず勘定系システムで関係すると思われるものをオブジェクトとして導出していくということから始める手順では，要員スキル面からいって無理がある.

この問題に対応するために，本開発では既存システム（具体的には FAST 1100, TRITON）をオブジェクト指向技術を用いてリエンジニアリングするという手順で開発を行うこととした.

4.1 オブジェクトの導出と属性の定義

オブジェクトの導出に関しては，以下の手順で行った.

① まず，顧客，口座，口座の取引履歴といったわかりやすいオブジェクトを導出し，その役割を定義する.

② 次に，既存システムがデータベースに保有しているデータ項目と突合し，上記①の役割に照らし合わせてそのデータ項目がどのオブジェクト属性に保有されるのがふさわしいかの検討を行う.

③ 上記②の作業において，既存システムの保有データ項目が①で導出したいずれのオブジェクトの属性にも当てはまらない場合が発生する．それは大別すると以下の 3 種類に分類される.

ａ．明らかに別の役割を持つオブジェクトを新たに導出し，そのオブジェクトの属性として定義すべきと判断されたデータ項目

例：キャッシュカード/通帳等の媒体情報，ANSER 加入情報等のように顧客とか口座に付随した契約の契約内容　等

ｂ．基本的な属性項目の加工後の結果（すなわち，他の属性項目から算出可能な項目）ではあるが，プログラム実行時の処理効率上の理由等によりデータベース項目として保有していたもの（以降，これらの属性項目を派生属性と呼ぶ）

例：通帳に記帳されていない取引履歴の総数

顧客単位の定期預金の総残高　等

ｃ．既存のシステム構造（アプリケーション構造，及び，データベース構造）上，必要なため保有していた項目

例：スレーブ数，レコード内のアイテム数　等

④ 上記のうち，a．に該当する項目に関して検討を行い，その項目を属性として保有するにふさわしい新たなオブジェクトの導出を行う.

この②〜④の手順を繰り返すことによって既存システムのデータベース項目は，いず

れかのオブジェクトの属性として定義されることになる．従って，オブジェクトの導出漏れの有無の検証（すなわち網羅性の検証）も同時に行われたことになる．なお，上記③の b.，c. に属するデータ項目に関しては，この段階では無視することとした．

4.2 オブジェクトのサービスの導出

オブジェクトのサービスの導出に関しては，図 4 に示すように既存システムのプログラムソースコードから導出するという方法を採った．なお，その際既存システムには存在しない新機能に関しては，予め定められた書式で定義しておく．（SBI 21 では，これを新機能要件定義書と呼ぶ）

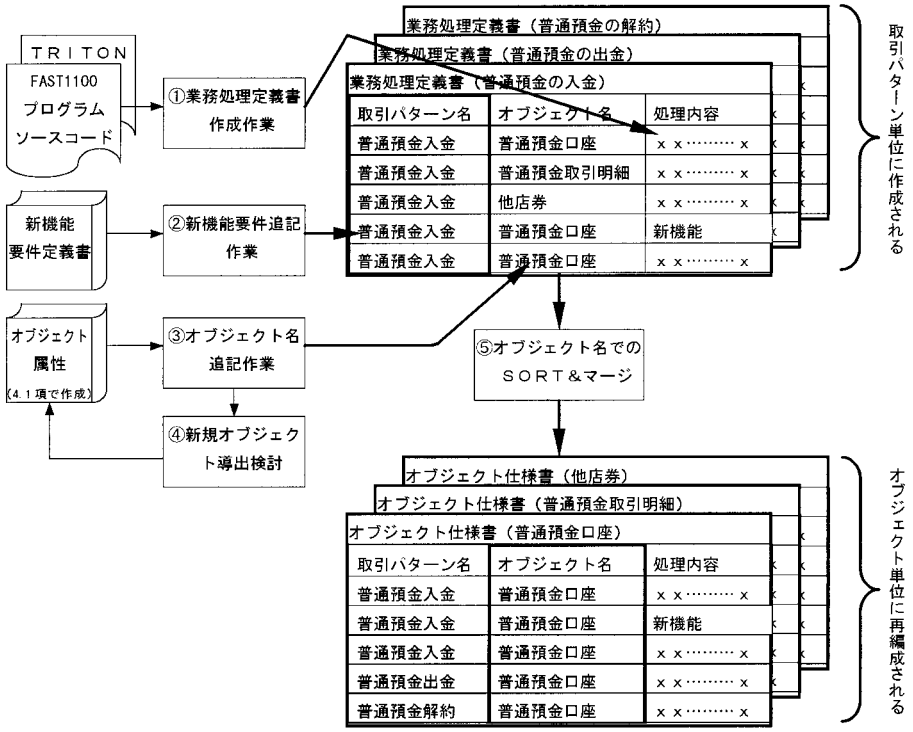

図 4　オブジェクトサービスの導出手順

① まず，既存システムのプログラムソースコードを取引パターン単位（例：普通預金の入金，出金，解約等）に解読し，解読した内容を EXCEL のワークシートに記述していく．このワークシートを本システムでは業務処理定義書と呼ぶ．

② 上記業務処理定義書に対して，新機能要件定義書に記載されている新機能実現のための処理を補記する．

③ 次に，その業務処理定義書の処理内容を見ながら，その処理は，今回の構造においてはどのオブジェクトで行うべきかを判断し，そのオブジェクト名を追加定義していく．どのオブジェクトで行うべきかの判断は，その処理内容で操作している属性が 4.1 節で導出したどのオブジェクトに帰属しているかを目安に行うことができる．

④　この時点で，どのオブジェクトにも帰属できそうもない処理が出現する場合がある．その理由は，大別すると以下の二つである．

ａ．4.1 節では，あくまで既存システムのデータベース項目を元にオブジェクトの導出を行っているが，新機能を追加したことにより新たに必要となるデータベース項目が発生する場合があり，それを受け持つオブジェクトが存在しない場合がある．

ｂ．オブジェクトとオブジェクトの関係上発生している処理がある．例えば，総合口座は１人の顧客では１口座しか持つことができないというチェックを行っているような処理は，顧客オブジェクトと口座オブジェクトとの関係をチェックしている処理であり，どちらか一方にそのチェック処理を帰属させるには無理がある．

このような場合は，再度オブジェクトの導出の検討を行うことになる．特に，b. の場合は，オブジェクトとオブジェクトの関連自体を受け持つオブジェクトの導出が必要か否かという検討を行うことになる．

⑤　このようにして完成した業務処理定義書を今度はオブジェクト名で SORT することにより，オブジェクト単位にとりまとめ，抽出する．このような加工が行えるということが業務処理定義書を EXCEL で作成した理由である．一つのオブジェクトの処理は複数の業務処理定義書に定義されている．例えば，普通預金口座オブジェクトの処理は，普通預金の入金という取引パターンにも出金という取引パターンにも定義されている．従って，各業務処理定義書から抽出したものをオブジェクト単位にマージするという作業を行う．取引パターン＝サービスと考えれば，これで，オブジェクトのサービスの洗い出しとそのサービスの処理内容を記述した仕様書が完成したことになる．これをオブジェクト仕様書と呼ぶ．

現実の作業内容は上述したほど単純ではない．一つの取引パターンが複数のサービスに分割される場合もあるし，また，複数のサービスで同一の処理を行っている場合は，その処理を内部的な共通処理として再整理していく等の作業が発生する．しかしながら，既存システムのプログラムソースコードが元になっているため，従来のように設計書から作成し最終的にプログラム仕様書を作成するという手順に比べ，必要とする処理の網羅性という観点においては非常に高い．

5.　リエンジニアリングの効果と誤算

5.1　既存システムのプログラム構造と手順のミスマッチ

4.2 節に記載した手順であれば，プログラム仕様書ができるまでの生産性は，単純には，

既存システムの総ステップ数/1 日に解読できるステップ数

となる．この生産性であれば，従来のような設計書から作成し最終的にプログラム仕様書を作成するという手順よりは，遙かに高い生産性が期待できる．

しかしながら，既存システムでは共通サブルーチンが多用されている．それに対し，

業務処理定義書は取引パターン単位に作成していく．従って，複数の取引パターンで使用されている共通サブルーチンはその取引パターン数の回数分解読対象となってしまう．すなわち，上述の計算式の分母の部分は単純に総ステップ数ではなく，

共通サブルーチンのステップ数×サブルーチンが使用されている取引パターン数

という部分を加算した数値となってしまい，複数の取引パターンで使用されている共通サブルーチンが多ければ多いほどその生産性は低下していく．

一方，取引パターンによっては極めて処理内容が類似しているものが存在する．例えば，普通預金の口座開設という取引パターンは，その処理の途中から入金という取引パターンとほぼ同様の処理となる．また，出金と入金訂正という取引パターンも類似性が高い．このように類似性が高い処理に関しても画一的に１取引パターンずつ業務処理定義書を作成していくのでは，結果的にほぼ同一の処理内容を複数の業務処理定義書に記述することになり，作業の無駄が多くなる．特に，業務スキルの高い要員にとっては，このように無駄の多い単調な作業は耐え難い．

従って，最終的な手順としては，主要な取引の業務処理定義書の作成が完了した時点で，4.2 節の⑤に記載したオブジェクト単位への分割処理を行い，一旦，オブジェクト仕様書を作成し，それ以降の取引パターンに対しては，プログラムソースコードの解析結果を直接オブジェクト仕様書に追記するという手順とした．これにより，既存システムの共通サブルーチンに定義されていた処理は早い段階でオブジェクト仕様書内の共通処理として定義されることになり，取引パターン単位に何度も共通サブルーチンの内容を業務処理定義書に定義するという作業の無駄を排除することができた．

5.2 既存システムのプログラム構造と要員スキル

今回の手順は，一つの取引パターンに沿って既存システムのプログラムソースコードを解読し，その処理内容を業務処理定義書に転記していくという手順のため，当初は，業務に精通していなくてもプログラムを解読できるスキルがあれば，ある程度作業可能と考えていた．

しかしながら，既存システムのプログラムソースコードの中には，既存システムの処理上の仕組みやデータベース構造に起因する処理が随所に現れてくる．従って，プログラムを解読し業務処理定義書を作成する作業者は，その処理が金融機関の勘定系システムの業務処理として必要な処理なのか，あるいは，既存システムの処理上の仕組み/データベース構造等に起因する処理なのかを選別できるスキルがなければならない．その上で，業務処理の部分のみを抽出して業務処理定義書に転記するということを行う必要がある．その選別を行うためには，やはり，ある程度の業務スキルが必要であり，業務スキルの不足している要員が作成した業務処理定義書に関しては，業務スキルの高い要員による入念なチェックが必要という事態に陥った．かつ，この業務処理定義書はプログラムソースコードから作成した関係で，処理の詳細まで定義されており，非常に量の多い成果物である．従って，チェック作業には相当の時間を要す結果となり，業務スキルの高い要員に作業負荷が集中するという従来型の開発と同様の現象を引き起こした．

5.3 プログラム作成段階における高生産性の実現

最終的に作成されたオブジェクト仕様書の処理の記述内容は，元をただせば業務処理定義書であり，更にその元を辿ればプログラムソースコードである．かつ，業務処理定義書もオブジェクト仕様書も EXCEL のシートに定義している．また，プログラムソースコードから業務処理定義書に処理内容を転記する際の記述形式に関してはあらかじめルールを定めてあった．

これにより，簡単な EXCEL マクロを作成するだけで，オブジェクト仕様書の処理内容の約 60〜80% を COBOL のプログラムソースコードに自動変換することができた．従って，プログラム作成は，変換できなかった残りの 20〜40% の処理内容を COBOL に変換するだけの作業となり，プログラマが仕様書からプログラムを作成するという従来型の開発に比べ数倍の生産性を実現できた．しかも，仕様書の記述レベルが詳細であるため，プログラマによる判断が必要な部分がほとんどなく，従ってコードレビューの必要がないほど正確に仕様書からプログラムが作成できた．

6. 本手順の他システムへの適用

本手順は他の既存システムをオブジェクト指向技術を用いてリエンジニアリングする際でも有効であると思われる．しかも，手順書も成果物様式も，更には，その作業負荷を軽減できる PC 上での開発環境も用意されている．全てをそのまま踏襲することは無理だとしても，一から手順等を検討するよりは遙かに近道である．

しかしながら，適用に際しては，いくつかの留意点，及び改良点が存在すると思われるため，以降にそれを記述する．

6.1 要員教育と開発期間

本開発における懸念事項の一つに設計要員がどの程度オブジェクト指向技術を理解できるかということがあった．一応の教育を行い設計に臨んだのだが，その結果，以下に示すような傾向が現れた．

1) 今回のシステムの実行環境であるシリーズ 2200 系メインフレームの知識/経験が豊富な要員の方が，オブジェクト指向技術を受け入れにくい．
2) 勘定系システムの設計経験豊富な要員の方が，オブジェクト指向技術を受け入れにくい．

上記の 1）に関して言えば，シリーズ 2200 系メインフレームの特性がわかっているだけに，論理モデル設計段階で既に頭の中で実装設計を行ってしまうからだと思われる．その結果，例えば処理効率上の問題等に目が行ってしまい，無意識のうちに業務を忠実にモデルに写像するということより実装上問題がないことの方を優先してしまいがちである．

上記の 2）に関してもほぼ同様のことが言える．勘定系システムの処理を詳細に知っているために，オブジェクトの導出とその関係を定義する段階で，既に非常に例外的な処理まで考えてしまう．その結果，論理モデル作成段階では，業務を素直にモデルに写像するということに専念すべきにもかかわらず，例外的処理への対応の方が優先してしまう場合がある．

また，営業店の端末からデータが入力され，実際に処理がなされ伝票や通帳に記帳

されるまでの各処理のプロセスを全て熟知しているが故に，オブジェクトの役割を越えたサービス設計を行う場合があるという問題もある．例えば，普通預金口座オブジェクトの入金のサービス設計段階で営業店端末の無通帳キーが押下された場合/押下されなかった場合で処理が異なるということが一瞬のうちに思い描けてしまう．そのため，入金のサービスの入力パラメタに無通帳キーが押下されたか否かという情報を保有するという設計を行ってしまう．こうなると，普通預金口座オブジェクトは営業店端末のキーの状態まで知っていなければならないオブジェクトになり，その役割が拡大されてしまう．その結果，本来普通預金口座の役割とは無関係な営業店端末の形態が変化した場合でも影響を受けるオブジェクトになってしまう．

蛇足ながら，システム開発に際してよく言われる部品化による再利用に関して考えてみると，上記のように作成してしまった普通預金口座オブジェクトは，全く異なる形態の営業店端末が接続されているシステムではそのまま使うことが難しく，その端末に合わせた修正が発生するということになる．すなわち，部品としての再利用度が低下してしまう．実際に部品として再利用できるものを作成するということは非常に高度な設計技術を要するということがこの例でもわかると思う．

以上のことから，オブジェクト指向技術で分析設計を行う場合，従来型開発における優秀な設計者が必ずしもそのまま優秀な設計者として通用するわけではないということが言える．また，このようなことを教育だけで納得させることは非常に難しく，実際に設計したオブジェクトをレビューし，是正しながら体得していってもらうということが必要となる．従って，今回と同じような手順で行う場合で，かつ，設計者がオブジェクト指向技術の経験がない場合は，設計の初期段階は非常に生産性が悪いということをある程度覚悟しておく必要がある．そして，それに対応するためには当初は設計要員を大量に確保せず，代わりに設計期間を通常より長めに設定するということが重要である．

6.2 本システムの構造上の改良点

開発言語にオブジェクト指向を支援する言語機能を有しない COBOL を採用したことにより，かなり初期の段階でオブジェクト指向技術の適用はカプセル化という考え方に限定するというように無意識に結論づけてしまっていた．確かにそれにより，それほど深くオブジェクト指向技術を知らない人間でも設計要員として十分に戦力になったという利点はあった．しかしながら，思い返すと早々と断念しすぎてしまったという部分もある．そのいくつかを以下に記述する．

1） クラスデータとクラスメソッド

COBOL では，オブジェクト指向技術でいうインスタンスという考え方は実装できない．同一クラスから複数のインスタンスを生成するという言語機能がないためである．インスタンスを複数生成できない以上，クラスデータとかクラスメソッドという考え方の適用には無理がある．従って，本システムにおいては敢えてクラスデータ/クラスメソッドとインスタンスデータ/インスタンスメソッドというような区分けは行わなかった．その結果，実際に完成したプログラムを見ると両者が同一プログラムの中に混在することになってしまい，そのことがオブジェクトの役割を曖昧にする一因となってしまった．少なくとも例えばクラスデー

タ/クラスメソッドに関してはオブジェクトとは別プログラムに定義するというような区分けはできたように思われる．そのようにしておけば，オブジェクトのプログラムとは明確な識別ができ，いらざる混乱を招かずに済んだように思われる．

2) 継　　承

継承に関しては，当初，本システムが提供する PC 開発環境にて，スーパークラスの属性/メソッドをサブクラス側のプログラムにインラインコードとして埋め込むということを想定していた．しかしながら，インラインコードで実現した場合，スーパークラスの属性/メソッドが変更になった場合，結果的に全てのサブクラスのプログラムが変更になるという問題があると共に，PC 開発環境側での対応負荷の問題もあり，結果的に継承に関しては断念せざるを得なかった．

これに関しては，スーパークラスの属性はスーパークラスでしか操作できないというオブジェクト指向技術での約束事にあまり厳密なこだわりを持たず，図 5 に示すような実装形態を考えてもよかったように思う．オブジェクト指向技術で勘定系システムを構築すること自体が目的ではなく，あくまで手段であるという旨を前述したが，このような考え方を採れなかったのは，知らず知らずのうちに自分自身の中で手段と目的がすり替わっていたのかもしれない．

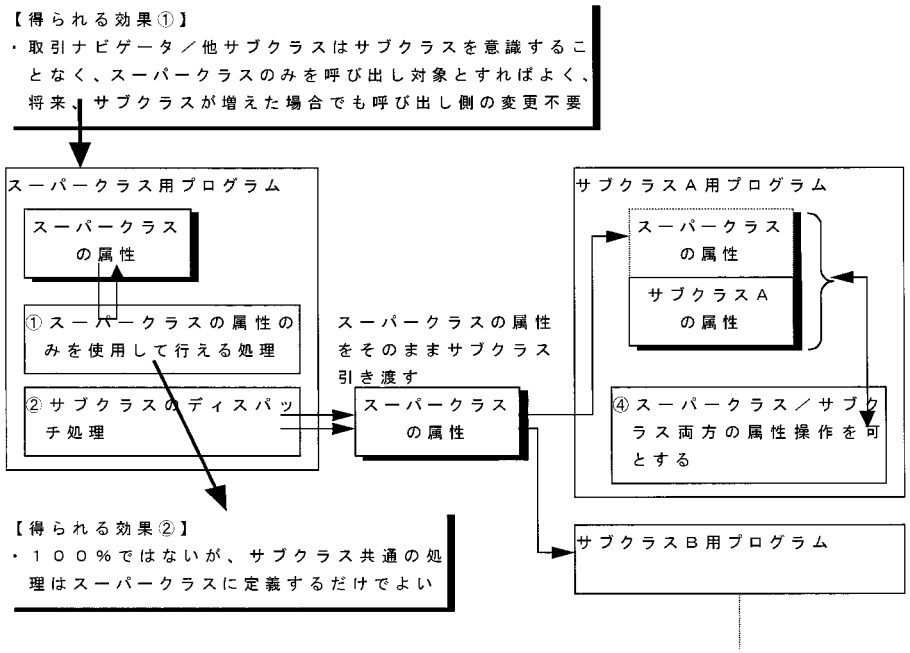

図 5　継承の実装イメージ

7. お わ り に

本システムは 1999 年 2 月現在構築途上であり，単体テストが完了した段階にある．かつ，前述したような改良点も散見される．しかしながら，全体的に見れば，今回の

リエンジニアリングの手順によって，複雑化/肥大化したと言われる勘定系システムをシンプルな構成要素に分解し，部分的/段階的に書き替えることができる構造にしようという当初の目的は達成できる見通しは立ったように思う．

これから，実行環境上でのテストを控え，もう一つの非常に大きなハードルである処理効率との戦いが始まる．現時点の机上計算においては，当初設計目標値の達成は視野に入る範囲になりつつあるものの，この処理効率のハードルを今の構造をどれだけ歪めることなく越えられるかが今後の大きな課題である．

その意味では本稿は，本来，このシステムの実行環境上での稼働確認とその評価が明確になった時点で記述すべきだったかもしれない．しかしながら，金融機関においては日本版ビッグバンを迎え，より迅速な商品開発を要求されるに至り，それに起因し現行勘定系システムの再構築を検討なされている方も多いかと思う．その方々に対する微かなヒントにでもなれば幸いと考え，やや時期尚早にもかかわらず，本稿を記述した次第である．

参考文献 ［1］ J. ランボー/M. プラハ/W. プレメラニ/F. エディ/W. ローレンセン著　羽生田　栄一監約, オブェクト指向方法論 OMT, 株式会社トッパン, 1993 年 10 月 15 日.

執筆者紹介 向　井　　丞（Susumu Mukai）
1955 年生．1976 年国立八戸工業高等専門学校電気工学科卒業．同年日本ユニシス(株)入社．FAST 1100 の開発，信用金庫業界を中心とした SE サービスを経て，1994 年より SBI 21 の開発に従事．現在，SBI 21 の開発部署であるビジネスソリューション一部 SBI 開発室に所属．