開発技術と開発プロセス

アーキテクチャを支えるコアコンポーネント

金融システムビジネスユニット
年金業務ソリューション第二構築センター
湯浅　隆

1．はじめに

住友信託銀行殿の年金管理システム（以下、本システム）はプログラミング言語としてMicrosoft Visual Basic .NETを採用し、全面的にオブジェクト指向で実装された大規模システムである。ピーク時のプログラマは120名を超える大規模プロジェクトであり、システム全体の統制感をとるために、当然と言えば当然であるが、アーキテクチャ中心の開発アプローチを採用して開発を行った。オブジェクト指向に精通した少数精鋭のメンバーにより設計されたアーキテクチャは、オブジェクト指向の未経験者の割合が多い本プロジェクトにおいて、揺るがない拠りどころとしてプロジェクト全体を成功へと誘導する役割を十分に果たしたと実感している。

ここでは、本システムのアーキテクチャをいくつかにカテゴライズし、それを支える主要なコンポーネント（フレームワーク）について紹介していきたい。紹介にあたっては、UML*1等による図式表現もいくつか挿入したが、理解のしやすさを優先し、オブジェクト構造やメソッド等には簡略化や名称変更を施した。

2．基本処理パターンとアプリケーション構成要素

コンポーネントの詳細に入る前に、まず、本システムの基本的な処理パターンとアプリケーションの構成要素の概略について記述する。

2.1　基本処理パターン

本システムでは処理方式を大きく3つのパターンに分類して設計し、個々のアプリケーション（業務機能）では機能要件や処理の特性によりいずれかのパターンを選択して、そのパターン上に実装することとした。3つのパターンの概略は以下のとおりである。

・オンライン処理と日中バッチ処理による構成
・オンライン処理と夜間バッチ処理による構成
・オンライン処理のみによる構成

一つ目は、オンライン処理と日中バッチ処理により全体処理を構成するパターンである（図1）。このパターンでは、オンラインは、画面入力されたデータについて、その妥当性に関する基本的なチェックだけを行い、業務データベースの更新や帳票の作成といった本質的な処理は、オンラインから起動されたバックグラウンド処理が、オンライン処理とは切り離されて独立的に実行されるという形態をとる。このバックグラウンド処理は、日中に実行されるバッチ処理という意味で「日中バッチ」と呼ぶが、実際、その処理の実装は夜間バッチと全く同一の様式で実装される。

二つ目は、オンライン処理と夜間バッチ処理により全体処理を構成するパターンである（図2）。このパターンは「オンライン処理と日中バッチ処理による構成」のパターンと似たものであるが、バッチ部分がオンラインから起動されるのではなく、夜間バッチとしてジョブスケジューラから起動されるという点が異なる。

三つ目は、バッチ処理を一切使わず、オンライン処理がすべての処理を行うパターンである（P.20図3）。主に画面照会機能などで使用されるものだが、更新系機能でもこの

＊1）Unified Modeling Language　Grady Booch氏、James Rumbaugh氏、Ivar Jacobson氏の3人によって開発された、オブジェクト指向分析、設計の統一記法。OMGにより標準として認定されている。

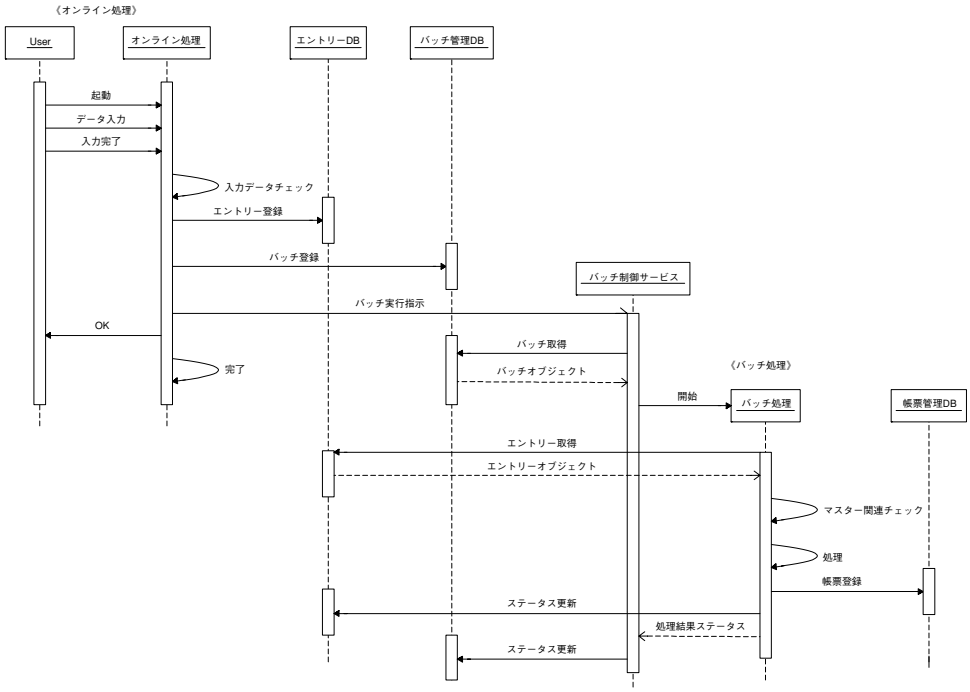

図1　基本処理パターン：オンライン処理と日中バッチ処理による構成

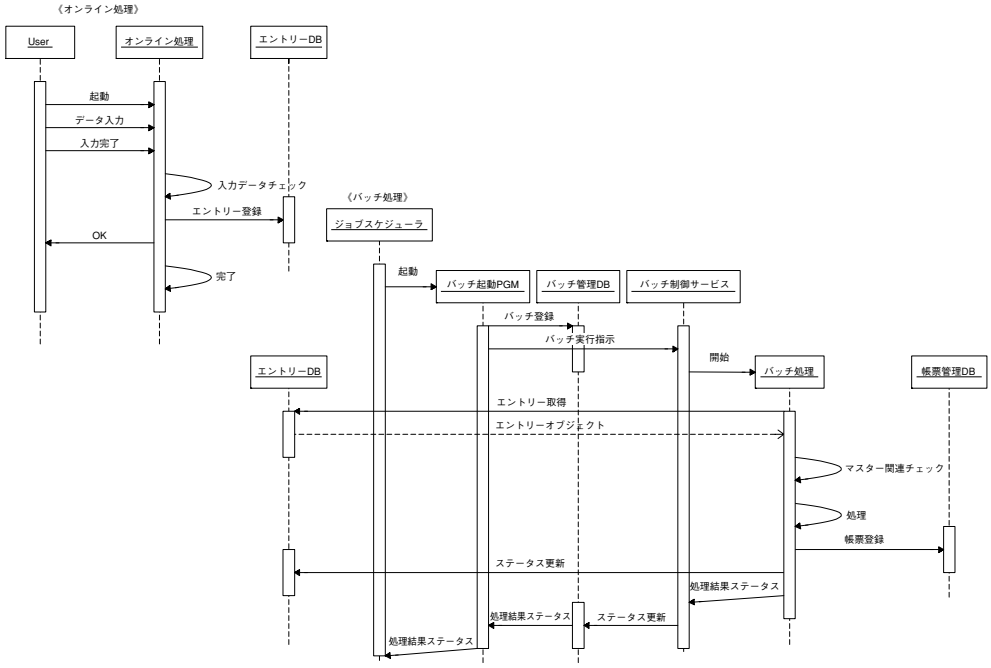

図2　基本処理パターン：オンライン処理と夜間バッチ処理による構成

パターンを使用したものがいくつか存在する。ただし、オンライン処理のレスポンス低下を避けるため、このパターンでの実装を許可される更新処理は処理時間の短いものに限定される。

2.2　オブジェクトの切り出し

本システムでは、エントリー（入力伝票のアナロジー）、業務（台帳のアナロジー）、帳票（論理的な帳票）、バッチ（バッチ処理）、画面（GUI）の5種類を主要オブジェクトとして管理することとした。それら基本インタフェースを規定するクラスとして、エントリー、業務、帳票、バッチ、BusFBase（Pageオブジェクトを継承した画面のスーパークラス、Business Façade Baseの略）というスーパークラスを設計し、個々のアプリケーション機能（オンライン処理やバッチ処理）は、これらスーパークラスを継承する具象クラスを定義することで構築することとした。もちろん、

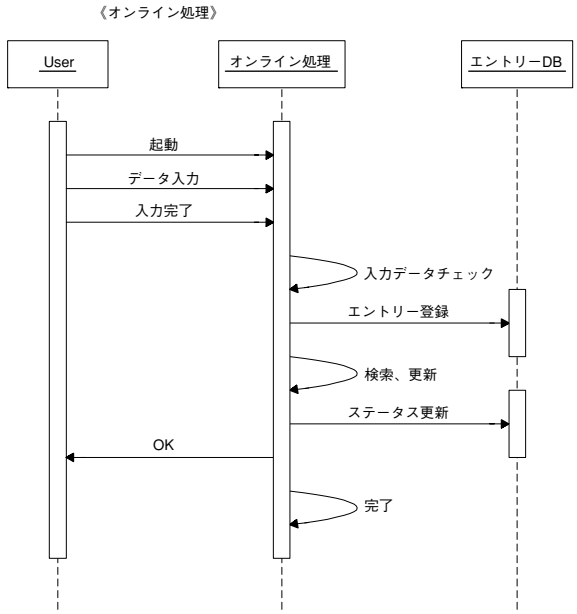

図3　基本処理パターン：オンライン処理のみによる構成

上記カテゴリに属さない大小様々なビジネスコンポーネントもクラスとして実装されているが、メインとなるオブジェクトは上記5種のいずれかに属している。スーパークラスの設計にあたっては、具象クラスの統一性や開発の容易性を考慮し、常套手段ともいえるがTemplate Method Pattern [GoF*2]などを積極的に適用した。以下に切り出された主要オブジェクトについての概略を記述する。

個々の入力画面については、画面とエントリーの具象クラスを一対として定義することで構築することとした。例えば、「新規加入員届」の登録画面であれば、新規加入員届画面クラスと新規加入員届エントリークラスという2つのクラスを定義する。画面とエントリーの関係は、一般的に良く知られているMVC*3フレームワークのViewとModelの関係に相当すると考えてよい。画面の具象クラスはエントリーの具象クラスをインスタンス化し、互いに協調する形で、画面制御に関する処理とビジネス処理を分担しながら、個々の入力を扱っていく。

ここで、前述した「オンライン処理と夜間バッチ処理による構成」による処理パターンを考えると、エントリーは生成されてから処理が終了するまでの非常に長い時間システムにとどまるオブジェクトであり、また、オンライン処理とバッチ処理の連携を実現する意味においても、永続化（データベースへの保存）が必要なオブジェクトである。このため、本システムでは、.NET FrameworkのSOAP*4フォーマッタを利用し、オブジェクトをシリアライズ（Unicode テキスト化機能を利用したXML変換）して、データベース上の1テーブル（厳密には親子2テーブル）に保存するという方式を採用した。この方式を採用した利点として、エントリーを保存するテーブル（エントリーDBと呼ぶ）が1つで良いという点があげられる。エントリーが持つデータ構造は個々の具象エントリーによって異なるため、各データ項目をデータベースの1カラムに対応させるという通常の方式を採れば、具象エントリーの数だけテーブルが必要ということになってしまう。これに対し、エントリーをシリアライズして保存するという方式によれば、エントリー全体を1カラムに保存することができるため、テーブル構造は具象エントリーの構造に依存しない形とすることができる。具体的なエントリーDBの構造としては、エントリーそのものを格納する「本文」というカラムと、すべての具象エントリーについて共通な管理項目（ステータス等）を格納する個別のカラムで構成することとした。また、エントリーDBに対する保存、更新、検索等の処理をつかさどる、エントリー管理というクラス（具象クラス）を汎用コアコンポーネントとして導入した。

バッチについても、エントリーと同様に永続化が必要なオブジェクトである。これは、例えば「オンライン処理と日中バッチによる構成」では、オンライン処理にてバッチの具象クラスをインスタンス化してデータベースに登録し、バッチ制御サービスが非同期でバッチ処理を実行するという方式を採るためである。このため、バッチ管理DBとバッチ管理クラスを導入し、エントリーと同様の方式でデータベースへの格納を行うようにした。

帳票についても同様に帳票管理DBと帳票管理クラスを導入し、DBへのマッピングを実現している。ただし、帳票についてはオブジェクトをシリアライズして格納するのではなく、CSVやPDFといった帳票ファイルの実体を保存対象とした。

最後に業務クラスについて述べる。業務クラスはリレーショナルデータベース上のエンティティを表現するクラスであり、データモデルと密接な関わりを持つものである。どのような具象業務クラスを導入するかについては、データベース上のテーブル構造に深く関わってくるが、そもそも、リレーショナルデータベースの論理単位であるテーブル表現と、オブジェクト指向言語の実装単位であるクラス

＊2）Gang of Four　書籍「オブジェクト指向における再利用のためのデザインパターン」の4人の共著者。

＊3）Model-View-Controller　「Model」「View」「Controller」という3種類のオブジェクトを構成要素としてソフトウェアを実装するモデル。Modelは本質的な処理、Viewは画面出力、Controllerはユーザー入力の制御をつかさどる。

＊4）Simple Object Access Protocol 他のコンピュータ上のデータ、サービスを呼び出すための、XMLをベースとした通信プロトコル。Webサービスなどで利用される。

の対応付けの方針は、アーキテクチャ確定上しばしば困難な決断が必要となる部分である。本プロジェクトでは、CRUD分析*5に基づいて、テーブルをある程度のブロック（原則的に親テーブル1つと、いくつかの子や孫関係にあたるテーブルの集合） にグルーピングし、各ブロックに1つの具象業務クラス（業務クラスのサブクラス）を対応付けることとした。具象業務クラスのインスタンスには、基本的に親テーブルの1レコードと関連テーブル（子、孫）のレコードの束を管理する役割を担わせている。業務データベースへのインタフェースには、業務セットというクラスを導入し、このクラスがデータベースに対する挿入、更新、削除、検索を担当する。ただし、エントリー、バッチ、帳票とは違い、業務セットが扱うテーブルは1つではないため、個々の具象業務クラスに対し、1つの具象業務セットクラスを作成することとした。

以上のとおり、本システムでは、エントリー、エントリー管理、バッチ、バッチ管理、帳票、帳票管理、業務、業務セット、およびBusFBaseの 9 クラスを用いることでアプリケーション機能を構成している。

2.3　実装上必要となった重要なコンポーネント

前述したBusFBaseは、実は画面の実装上必要となった重要なコンポーネントである。フレームワークと言えば、画面の遷移を構成定義ファイルで管理し、フレームワークから画面のコンポーネントを呼び出すような形でのソフトウェア部品を想像する技術者が少なくないと思うが、本システムでは画面間の連携はそれほど複雑ではないこともあり、このような方式は採用していない。前述したとおり、個々の画面は対になるエントリーと共にMVCフレームワークに似た形で定義することで構築する。ただし、本システムでは画面遷移が複雑ではない反面、個々の画面は数ページにわたって入力が続くような複雑なものが多いため、1つの画面で扱うデータ項目の数が膨大となる。このため、画面とエントリーの間で必要なデータ項目の相互反映のコードが膨れ上がってしまう。これをいかに簡略化できるかが実装効率を上げるための大きな鍵となった。簡略化を実現するため、ASP.NETのPageオブジェクト上のコントロール（TextBoxほか）と、エントリーが持つメンバー変数間の相互反映の仕掛けをBusFBaseに導入した。これにより、アプリケーションのコーディングにかかる負荷をかなり軽減することができた。BusFBaseはこういった背景で導入されたクラスである。

その他、データベース更新のトランザクション管理を行うDACServicedComponent（DACはDatabase Access Componentの略、以下DAC）というクラスも実装上重要なコンポーネントである。DACはトランザクション管理をつかさどるクラスとして、本システムの実装を陰ながら支えた重要なクラスである。DACによって、個々のアプリケーション処理にはトランザクション制御の実装がほとんど不要となった。DACの詳細についてはコアコンポーネント実装概略のところで記述する。

3．コアコンポーネント実装概略

3.1　画面・エントリー・バッチ・データベースの関係

次ページ図4は、ASPX*6を利用して画面から入力が行われてから、データベースを更新するまでのクラスの関連を概念的に表わしたものである。

画面の具象クラス（Code Behind*7）に実装される処理は、ページオブジェクトの初期化処理とボタンに対応するイベント処理である。具象画面クラスの主な役割は、個々の画面に対応した具象エントリーオブジェクトをエントリーDBに保存することであり、業務ロジックや業務データベースの更新処理等が直接的に実装されることはない。また、「2.1　基本処理パターン」のところでも示したが、業務データベースの更新は画面ではなくバッチに委ねるというのが基本であり、一部の例外を除いて画面から業務データベースの更新処理を呼ぶこともない。業務データベースを更新する役割を担うのは基本的にバッチの具象クラスであり、具象バッチクラスはエントリーDBから具象エントリーオブジェクトを取得し（1つまたは複数）、その内容を業務データベースへ反映する。

エントリーは 1 件のトランザクションを表現するオブジェクトであり、業務はデータベース上の1レコードを表現するオブジェクトである（前述したとおり、複数の子や孫テーブルのレコードを束ねて管理するものもある）。

＊5）エンティティと機能の相互関係をマトリックス化して整理する分析作業。機能がどのエンティティにどういう操作（Insert, Select, Update, Delete）を及ぼすかについて、エンティティと機能を縦軸および横軸に置いたマトリックス上に、その交点に“C”,“R”,“U”,“D”の文字を埋めて整理する。CRUDは、Create/Refer/Update/Deleteの略。

＊6）Microsoft のWebアプリケーション開発フレームワーク（ASP.NET） における、Webフォームページファイルのこと。テキストファイルで拡張子に .aspx を持つ。

＊7）Microsoft のWebアプリケーション開発フレームワーク（ASP.NET） における、Webフォームページのコード（処理）のこと。ASP.NETでは画面デザインとコードは ASPX と Code Behind に分離して定義する（コードビハインドモデル）。Code Behind は .NET Framework で提供されるWebページのクラスを継承する形で定義する。

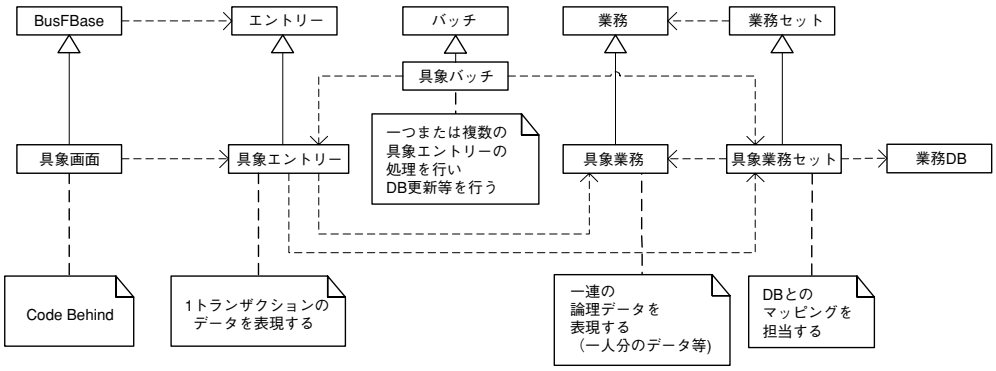

図4　画面・エントリー・バッチ・データベースの関係

データベースを更新するトランザクションの場合、具象エントリーにはデータベース更新を行う「マスター反映」といったメソッドを用意することとし、このメソッドには、トランザクションを表わす具象エントリーの内容をまず具象業務オブジェクトへ反映し、その後、具象業務セットを通じて具象業務オブジェクトの内容をデータベースに反映するといった実装を施すこととした。

業務データベースに対するアクセス機能は、すべて業務セットのサブクラスである具象業務セットクラスに、アクセス方式としてはADO.NET*8を利用して実現している。データベースに対するアクセス回数を減らし、検索系および更新系メソッドの処理効率を上げるため、業務セットにはキャッシュの仕組みを実装した。具象業務クラスのインスタンスを返却する検索系メソッドでは、検索結果をクラス内部のキャッシュにも貯めておき、2度目以降はデータベースへのアクセスを行わずにキャッシュからインスタンスを生成する。逆に更新系メソッドでは、受け取った具象業務クラスのインスタンスの内容をキャッシュに反映するだけでデータベースへの反映は行わない。データベースへの反映には、キャッシュに溜め込まれた内容をデータベースへ一括反映する「DB更新」という専用メソッドにより行うこととした。業務セットのクライアント（具象バッチクラス等）は、このキャッシュの仕組みを利用するよう設計することで、処理効率を上げることができる（図5）。

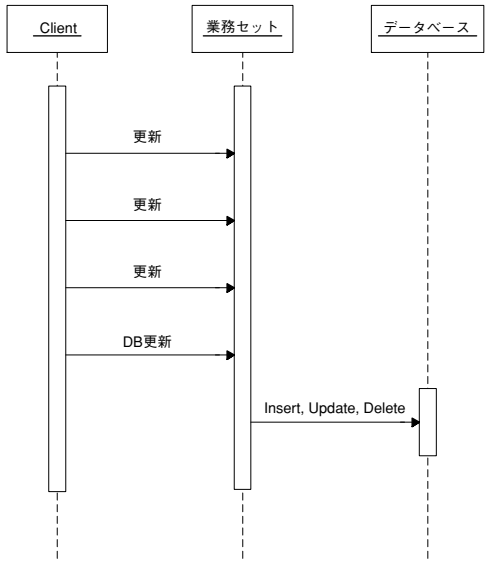

図5　データベースへの一括反映

3.2　オンライン処理

オンライン処理は、画面から入力されたデータをエントリーDBに収めることが基本的な役割となる（図6）。

画面からボタン押下等のアクションが行われた場合、画面オブジェクトはまずhttpRequestオブジェクトに含まれているデータ（画面入力されたデータ）を具象エントリーのメンバー変数へ反映する。「実装上必要となった重要なコンポーネント」のところでも触れたが、このデータ反映の仕掛け自体はBusFBaseに実装され、これによりアプリケーション側でのデータ反映コードの削減が図られている。アプリケーション側では画面項目とエントリーメンバー変数の対応関係、すなわちマッピングルールをBusFBaseに与える必要があるが、これは具象エントリーのプロパティメソッドにカスタムアトリビュートという形で埋め込むこととした。

画面からエントリーへのデータ反映後、次に実施するのはトランザクションレコードとしての正当性チェックである。ここで、チェック処理は画面ではなくエントリーに責任を持たせることとし、画面側ではエントリーのチェックメソッドを呼び出すという役割分担とした。エントリーに実装するチェック処理については、4つのメソッドに分類して実装することとし、各チェックメソッドの実行順序に

＊8）Microsoft .NET Framework で提供されるデータアクセスのアーキテクチャ。Microsoft ActiveX Data Objects （ADO） の発展系で、中心的なオブジェクトにDataSetやDataAdapterなどがあり、データ転送のフォーマットには XML が採用されている。

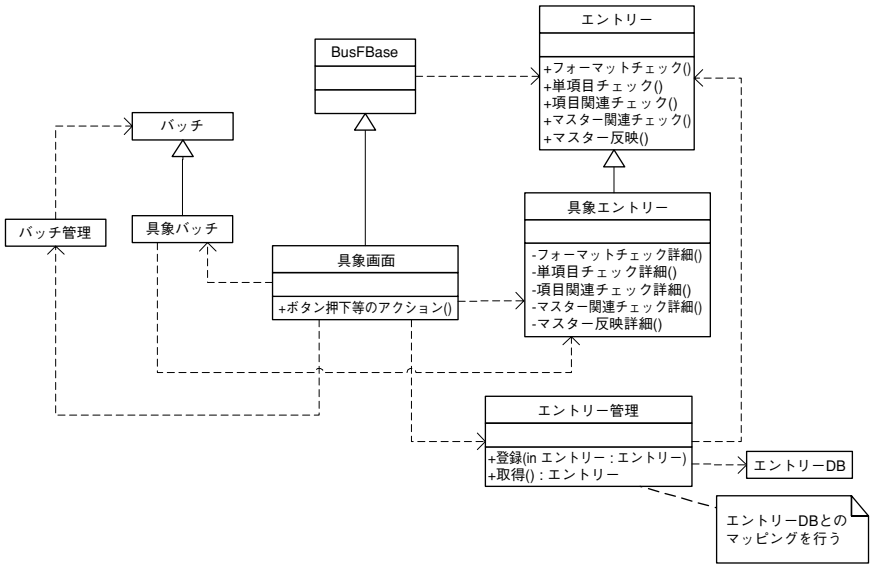

図6　オンライン処理

関してもエントリークラス（スーパークラス）で規定を行った。例えば、項目関連チェックは単項目チェックをパスしていないと実行できないといった具合である。具象エントリークラスでは、スーパークラスで定義されたインタフェースに従い、各チェック処理の本体を実装することとなる。

エントリーによるチェックをパスしたら、具象画面クラスはエントリー管理クラスの登録メソッドを呼び出し、具象エントリーのインスタンスをエントリーDBに登録する。この際、エントリー管理クラスの登録メソッドでは、受け取ったエントリーをシリアライズ（XML化）して、エントリーDBの単一カラムに収納する。この後、引き続き日中バッチによる継続処理を委託する場合、具象画面はバッチのインスタンス（具象バッチクラスのインスタンス）を生成し、そこに処理対象のエントリーを識別するキー情報をセットした上でバッチ管理DBへ登録を行う。図では省略したが、帳票をクライアントに返す画面では、具象帳票クラスと帳票管理クラスのインスタンスを利用して機能を実現する。

ここで、オンライン処理に関する実装上の工夫として、Microsoftが提供するMultiPageControlの利用についても言及しておきたい。年金管理の入力トランザクションは、繰り返し構造を含む多くの項目からなるため複数ページを必要とするケースが多い。今回の実装ではMultiPageコントロールを利用することにより、1入力トランザクションを構成する画面群をASPX上では単一ページとして扱うことにした。これにより、1入力トランザクション内でのページのリダイレクトが不要となり、複雑になりがちなASPXの状態遷移制御の実装を簡略化することができた。

3.3　バッチ管理コンポーネント

本システムのパフォーマンス確保上、バッチ管理は非常に重要なコンポーネントである。バッチを投入するサーバーとバッチを実際に実行するサーバーを分離し、1つのバッチプログラムを複数台のサーバーで実行する仕組みを構築した（次ページ図7）。バッチの投入には2種類あり、ジョブスケジューラがバッチ管理サーバー上で「バッチ生成.EXE」を実行して行う夜間バッチのケースと、Webサーバー上のアプリケーション画面（ASPX）から行う日中バッチのケースの2つである。どちらの場合も、具象バッチクラスのインスタンスを生成してバッチ管理DBに登録するとともに（バッチ管理クラスの登録メソッドを使用）、バッチ実行リクエストとしてMSMQ*9にメッセージ送信を行う。この際、メッセージには登録したバッチインスタンスの識別子を埋め込んでおく。

バッチの実行主体として複数のバッチサーバーに常駐しているWindowsサービスのバッチ制御は、MSMQにメッセージが残っていないかを常時監視している。メッセージが見つかった場合、バッチ管理の取得メソッドを用いて、メッセージ中に埋め込まれた識別子をもとに、該当するバッチのインスタンスをバッチ管理DBから取り出す。そして、取り出したインスタンスのバッチ処理メソッドを呼び出して、バッチ処理を実行する（厳密にはバッチ処理詳細メソッドに具体的な処理が記述されており、バッチ処理

＊9）Microsoft Message Queuing 非同期の分散型アプリケーションを構築可能な、Microsoftが提供するMessage Queuing Servicesのコンポーネント。

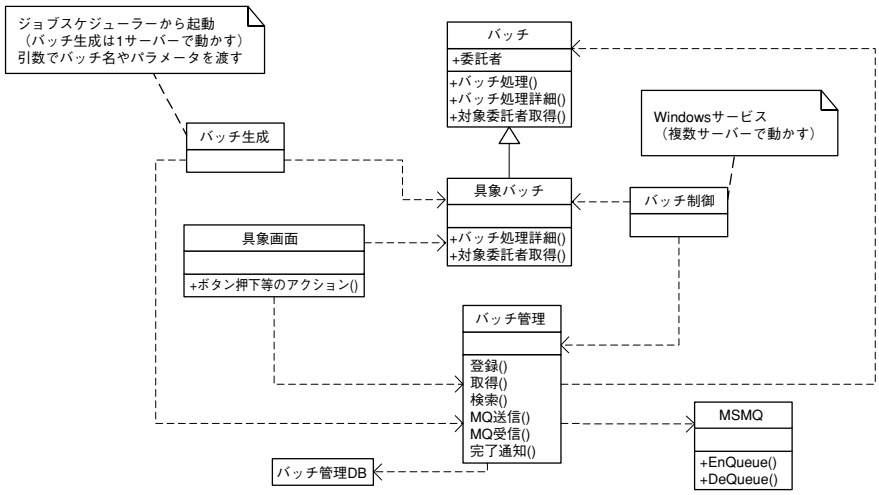

図7　バッチ管理コンポーネント

メソッドが内部的にこれを呼び出す)。バッチ処理メソッドが完了すると、バッチ制御は処理結果ステータスをバッチ管理DBに格納し（バッチ管理の完了通知メソッドを使用)、次のバッチ実行に備えてMSMQメッセージの監視に戻る。

MSMQのキューは、日中バッチ用と夜間バッチ用の２つに分けて設定しており、バッチ制御もそれぞれ別々に立ち上げている。ここで、夜間バッチは複数サーバーで並列実行させる必要があるため、夜間バッチ用バッチ制御は複数のバッチサーバーで動作させている（複数のバッチ制御が1つのキューを監視している)。逆に、日中バッチについては基本的に直列実行としているため、日中バッチ用バッチ制御は１台のバッチサーバーのみで動作させている。

次に、バッチの並列化向上策として実装した委託者分割の仕組みについて述べる。年金管理のアプリケーションでは、委託者（基金）が異なれば処理対象レコードが衝突することは基本的にない。この性質を利用し、委託者をキーにして処理対象レコードを水平分割し、それぞれ別々のバッチとして並列処理させることにより処理効率を上げるというのが、委託者分割の仕組みである。この仕組みの一部として、バッチクラスには対象委託者取得というメソッドを作成した。バッチの投入時、バッチ管理の登録メソッドを起動すると、対象委託者取得メソッドによって処理対象委託者のリストが作成される。そして、得られた対象委託者が20件であればMSMQに20件のメッセージを投入し、20個のバッチにより処理を行うというものである（バッチのインスタンスには「委託者」というメンバー変数があり、自分が処理すべき委託者を知ることができる)。これにより、１バッチの対象レコードをスライスして複数のサーバーで同時に実行できるようになった。ここで、分割のキーは、実は委託者に限定されるものではないという点にも言及しておきたい。対象委託者取得メソッドはキーとなる文字列のリストを返すだけであり、その文字列に意味的な制約は持たせていない。したがって、さらに細かい単位のリストを返却するよう、具象バッチクラスで対象委託者取得メソッドを実装すれば、例えば、同一委託者の処理を加入員番号のFrom、To等でさらに細かく分割するようなことも可能である。実際、基金解散等で非常に大量の届けや指図が発生するケースについて、この細分割の方式を適用することによりパフォーマンス問題を乗り切ったこともあった。

バッチ管理コンポーネントの締めくくりとして障害復旧について記述する。個々のバッチ処理に異常が発生した場合の復旧を容易にするため、バッチ制御は１トランザクションとして具象バッチのバッチ処理詳細メソッドを実行するようにした。具体的には、後述するDACを利用し、バッチ処理詳細メソッドが自動トランザクション制御下で実行されるよう、バッチクラス（スーパークラス）に仕掛けを導入してある。これにより、障害発生時にはデータベースがロールバックされ、障害復旧を容易にしている。特定の委託者で発生した障害が他の委託者に影響しないだけでなく、その委託者のバッチを実行しなかったのと同じ状態になるため、異常時にデータベースの復旧処理を実施する必要はない。

3.4　帳票管理コンポーネント

帳票管理コンポーネントとしては、帳票クラスと帳票管理クラスを導入した。帳票クラスは論理的な１帳票を表現するものであり、帳票作成メソッドのインタフェースなどを定義したスーパークラスである。アプリケーション側では、このクラスを継承する形で１つの帳票に対して１つの具象帳票クラスを導入し、「PDFファイル出力」等の帳票作成メソッドにレンダリングコードを実装する。帳票の元データは、具象帳票クラスに「帳票データ」という

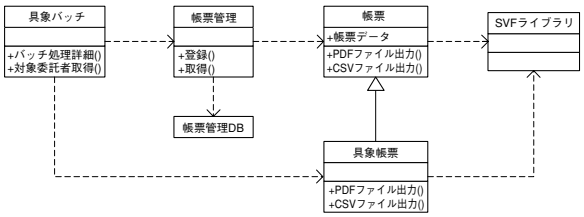

図8　帳票管理コンポーネント

DataSet型のメンバー変数として持たせることとし、帳票作成メソッドを実行する前にクライアント側（バッチ、あるいは画面）からデータをセットすることとした（図8）。

エントリー管理やバッチ管理と同様、データベースへのマッピングを行うクラスとしては帳票管理というクラスを導入した。本システムで生成した帳票ファイル（例えばPDFファイル）は、帳票生成の一時的な過程を除き、その保管先をWindowsのファイルシステム上ではなくデータベースとしているため、帳票管理クラスには帳票ファイルそのものをデータベースに格納するインタフェースも実装している。ここで、帳票ファイルはデータベースの１カラムに格納されるが、巨大な帳票ファイルへの対応として、ファイルを数MBずつに分割しながら格納し、最終的に１カラムへまとめるよう格納処理の実装に工夫を施した。

3.5　トランザクション管理コンポーネント

最後にトランザクション管理の汎用コンポーネントであるDACについて触れたい。DACは、業務セットクラスなどのデータベース更新処理を行うクラスの設計時に、必要に迫られ導入されたクラスである。当時、業務セットの基本設計がある程度進んだころ、本システムでのトランザクション管理にMicrosoft がCOM+ Serviceの形で提供する自動トランザクションの方式が採用されることとなったが、これは業務セットクラスの設計に多少なりとも問題を発生させるものだった。何故なら、自動トランザクションのサービスを受けるためには、データベース更新を行うクラスをSystem.EnterpriseServices.ServicedComponent（以下ServicedComponent）のサブクラスとして実装する必要があるが、ServicedComponentのサブクラスはステートレスなオブジェクトでなくてはならないという制約があったからである。この制約に反し、業務セットクラスはデータベース上のデータをDatasetに一部キャッシュするような形、つまりステートフルなオブジェクトとして設計が進んでおり、これをステートレスに設計しなおすことは非常に困難であった。業務セットの設計は基本的に変更せず、かつ自動トランザクションの仕掛けを使用できる良いアイデアはないか検討した結果として、DACが設計された（図9）。

DACはServicedComponentのサブクラスである。データベース更新を行うクラスは次ページ図10のようなシーケンスでDACを利用する。まず、DACのインスタンスを生成し、更新処理の実行をDACに依頼する（DACのDB更新メソッドを呼び出す）。呼び出されたDACは自動トランザクションのコンテキストを作成した後、自分を呼び出したオブジェクトのDB更新処理を呼びなおす（具体的には、DACにパラメータとしてわたってきたオブジェクトのDB更新詳細メソッドを呼び出す）。DACはDB更新詳細の完了を待ち、処理が成功していればSetComplete（Commitのフラグを立てる）、失敗した場合はSetAbort（Rollbackのフラグを立てる）を行うという仕掛けである。この仕掛けにより、具象業務セットクラスはServicedComponentのサブクラスである必要がなくなり、ステートフルな設計のまま、データベース更新処理を実装できるようになった。

また、DACのサービスを受けられるクラスを業務セットに限定したくなかったため、IDACObjectというインタフェースを導入し、これをDACの相手とした。これによ

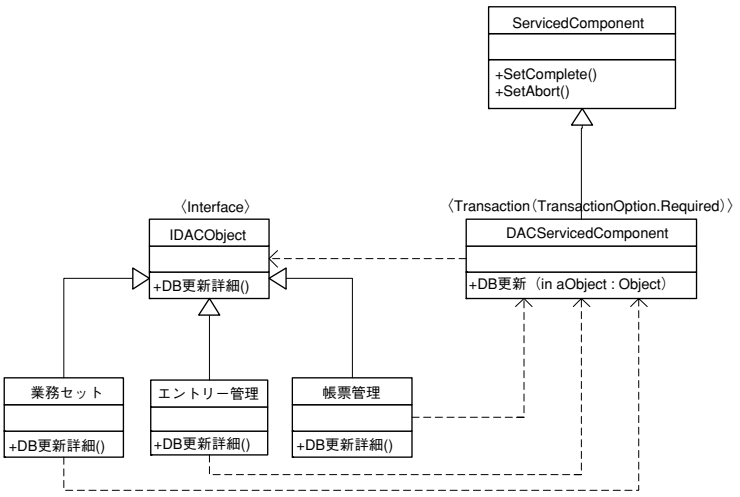

図9　トランザクション管理コンポーネント

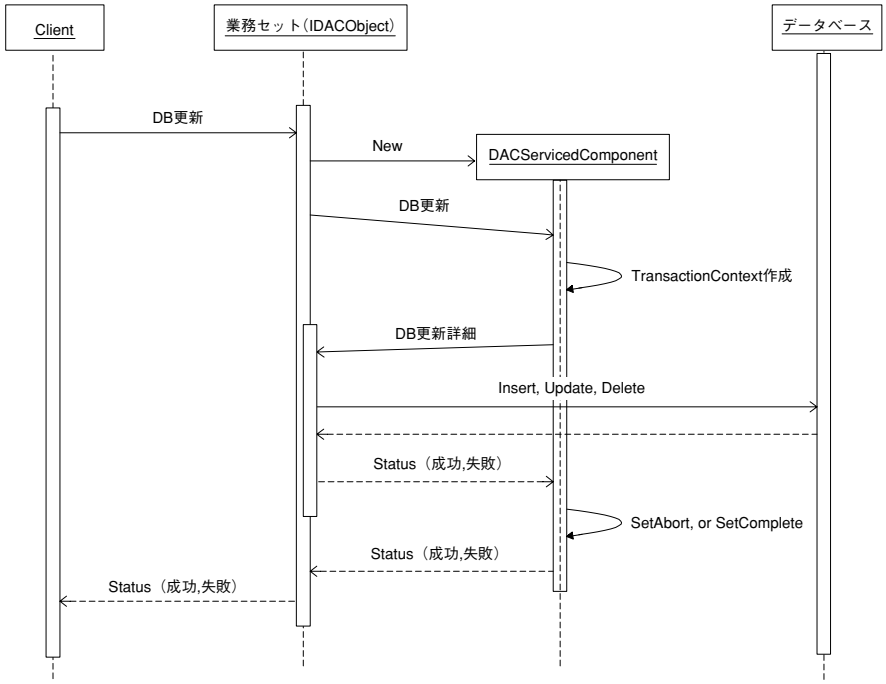

図10　DACによるデータベース更新のシーケンス

り、クラスツリー上のどこに位置するオブジェクトであっても、IDACObjectインタフェースを実装すればDACを利用可能となった。本システムでは、自動トランザクションによるデータベース更新を行うすべてのオブジェクトがIDACObjectインタフェースを実装しており（エントリー管理、帳票管理なども例外ではない）、ServicedComponentのサブクラスはDACが唯一のクラスである。

仮にDACがなかったとしたら、各具象業務セットクラスから更新処理を行う部分を抜き出し、別オブジェクトとして切り出さなくてはならなかったはずである。そうなれば、具象業務セットクラスは、ステートフルな本体のクラスと、ステートレスなServicedComponentのサブクラスの2クラスから構成しなくてはならなくなり、クラス数の増大、ひいては開発工数の増大を招いたと思われる。こういった意味で、DACは非常に重要なクラスである。

4．教　訓

エントリー管理およびバッチ管理を利用した実装は、本システムの柔軟性、パフォーマンス確保の両面で極めて有効に機能したと評価しており、適用領域は年金管理システムに限定されるものではなく、重いバッチ処理が問題となるようなシステムに対して広範囲に適用できる技術と考えている。また、帳票管理においても、データストアにはすべてデータベースを利用するというコンセプトは、運用時の作業軽減という意味でも有効に機能したと考えている。このような点から、本システムにおいて設計されたアーキテクチャおよびフレームワークは、大きくとらえて十分に成功したと評価している。

反面、アプリケーションが本当に組みやすく、容易に高い品質が実現できるようなフレームワークであったかという面で評価すると、まだ改善余地があると言わざるを得ない。例えば、エントリーと画面、エントリーとバッチ、エントリーと業務といった関係の中で、どのクラスに実装すべきか悩むような局面もあり、これは、フレームワークの柔軟性がアプリケーション側へ多くの選択肢を残してしまったということだと考える。また、フレームワークでは規定しきれない、アーキテクチャの根底にあるオブジェクトモデルや思想を、いかにして開発者全体に浸透させるかといったことも、全体の品質や生産性に影響を与える大きな要素であると考える。これについては、アーキテクチャ確定の段階で開発者全体に向けた十分な資料を作成する必要があると痛感した。

実装面で振り返ってみると、ADO.NETのDataSetの利用法が反省点として上げられる。業務クラスやエントリークラスにおいて、データベースデータの保持にDataSetを使用し、これをパブリックな属性として公開したことが、データベーススキーマとアプリケーションの結合度を強めてしまった。作るときには型もゆるく設計も軽くなるが、物理的なデータストアと論理的なクラスの構造の分離が不十分になってしまうという問題があるため、今回のようなDataSetの利用については必要性を十分に検討してから適用すべきであると考えている。実装面での別の反省点とし

て、エントリーオブジェクトのサイズについても、考慮すべき点があると考えている。ASP.NETアプリケーションの特性上、状態保持のために各種オブジェクトをASP.NETのセッションデータへ格納する必要があるが、大きなトランザクションデータがエントリーオブジェクトを肥大化させることにより、セッションデータも大きくなり、IISの機能を用いたロードバランシングを困難にする。また、エントリーオブジェクトが大きくなれば、シリアライズ処理の時間も問題となり、処理能力上の足かせとなる。エントリークラスを十分に小さく設計するための工夫が必要と考えている。

5．終わりに

オブジェクトモデリングやフレームワーク設計はオブジェクト指向開発において最も技量の問われるところであり、その分やりがいもあり、またとても楽しい作業だと思う。筆者はオブジェクト指向の経験が10年を超えるが、オブジェクト指向言語による大規模システム開発は本プロジェクトが初めてであった。本システムのフレームワーク設計において筆者の携わった個所は全体のほんの一部分であるが、作業は非常にエキサイティングなものであり、こういった経験をできたことをとても嬉しく思っている。また、こういった機会を与えてくれた方々、作業中いろいろと助けてくれた方々には、心から感謝の意を表したい。