業務システムのソフトウェア品質向上のために
（プログラム開発工程の品質管理）

新日石インフォテクノ株式会社

■ 執筆者Ｐｒｏｆｉｌｅ ■

柳　豊

1989 年　日本石油（株）入社
日石情報システム（株）配属
システム開発部　システム担当
2002 年　ホスト廃止に伴う業務システム再構築担当
2007 年　現在 システム開発部開発１グループ所属 人事システム開発・保守担当

■ 論文要旨 ■

ＩＴは企業の経営戦略と密接に関係しており，企業経営にとって無くてはならないものとなっている．新日本石油(株)においてもパッケージの採用や自社開発によりさまざまな業務システムを導入しており，それらのほとんどは弊社において開発・保守，運用しているため，その品質維持管理については我々の重大な責務となっている．現在，私の部署で開発・保守を担当している新日本石油(株)の人事システムは 2002 年から 2003 年にかけて，上流工程から下流工程までの全工程を自社開発で行い，現在も保守工程を繰り返しながら安定稼動を継続している．

業務システムとして稼動するソフトウェアのいわゆる品質を確保するには，要求仕様の的確な把握や設計フェーズでの品質管理や，開発工程においてはテスト工程がシステム品質のチェックポイントとして重視されるため，その技法などについては多くの書籍も出版されており，情報も豊富でノウハウも蓄積されている．

今回は，プログラムの開発プロセスや開発基準やルールの策定など，システムの品質を維持するために，テスト工程以外の開発工程で実施しなければならない内容について述べてみる．

■ 論文目次 ■

■ 図表一覧 ■

１．　はじめに

1.1　当社の概要

当社は日本石油（株）（現，新日本石油(株)）の情報システム部門を，同社の 100%子会社として 1985 年に分社化し，設立された日石情報システム(株)を前身として，2003 年に新日本石油(株)と富士通(株)との合弁会社として発足し，現在は新日本石油グループに関わるシステムの開発・保守，運用やパッケージの販売を主力事業としている.

弊社は 2006 年 1 月に ISO9001 の認証を取得しており，全社目標や部門目標を掲げ，品質活動に全社をあげて取り組んでいるところである。

1.2　業務システム開発環境の変化

私が入社した頃（約 20 年前），業務処理システムといえばホストコンピュータのバッチ処理が中心であり，COBOL と JCL でその機能のほとんどを実装できていた.

弊社においても当時は開発・保守，運用していたシステムの大半が COBOL 資産であり，その品質確保・維持管理するためのシステム開発基準や，プログラムのコーディングルールなどは社内基準として一つ決められており，どの業務システムの開発担当者もその基準に沿って開発・保守を行うことができた.

しかし，1990 年を過ぎたころからコンピュータ関連の技術はめまぐるしく進歩し始める.

そのため，さまざまな環境で，それぞれに適合したインフラやアーキテクチャーを採用した業務システムが企業に導入されるようになり，また，この頃から社員へのパソコンの普及も進んでいった.

2000 年頃から，弊社においても全面的に基幹業務システムの再構築を行うこととなり，全ての基幹業務の再構築終了とともにホストコンピュータは廃止されることが決定した.

基幹業務システムの再構築を行うにあたり，社内の統一基準として採用していた開発基準やルールを全面的に見直す必要があったが，システムごとに再構築の時期や採用するインフラ，OS，アーキテクチャー，プログラム言語などの技術も異ったため，再構築するシステムの開発プロジェクトごとに開発基準やルールを作成することになった.

私が担当している新日本石油(株)の人事システム(以下人事システムと略記)も，販売・物流や経理システムなどといった，他の大規模基幹系業務システムとは別プロジェクトとして再構築を行ったため，アーキテクチャー，OS（Windows 2000 Server），開発言語(Visual Basic6.0（以下 VB と略記），Active Server Pagese(以下 ASP と略記))などの採用した技術が異なっており，人事システム開発プロジェクトとして開発基準やコーディングルールなどを決めている.

２．　システムの品質について

2.1　ソフトウェア品質の規格

ソフトウェアの品質を評価するための国際規約や国内規格について，どういったものがあるか確認してみる．

品質管理と品質保証に関する国際規約としてISO(International Standardization for Organization：国際規格認証機構）が定めたISO9000シリーズがあるが，このうちソフトウェアの品質評価のための規格として，1991年にISO/IEC9126-1が制定されている．

日本国内ではこの規格に基づいて，JIS（Japanese Industrial Standard：日本工業規格）が定めたJIS X0129-1（ソフトウェア製品の評価－品質特性およびその利用要領）という規格がある．

（企画書の全文はJISCのWebサイト　http://www.jisc.go.jp/index.html　で参照可能）

「ISO/IEC9126」および「JIS X0129-1」の中では，評価が可能なソフトウェアの品質特性を６種類に分類し，図 1 のように定義している．

品質特性	品質副特性				
機能性	合目的性	正確性	相互運用性	セキュリティ	機能性標準適合性
信頼性	成熟性	障害許容性	回復性		信頼性標準適合性
使用性	理解性	修得製	運用性	魅力性	使用性標準適合性
効率性	時間効率性	資源効率性			効率性標準適合性
保守性	解析性	変更性	安定性	試験性	保守性標準適合性
移植性	環境適用性	設置性	共存性	置換製	移植性標準適合性

図 1 品質特性と副特性

（１）機能性：ソフトウェアが指定された条件下で利用されるときに，明示的および暗示的必要性に合致する機能を提供するソフトウェアの製品の能力
　→要求を満たす機能を正しく提供し，かつ安全に保護しているか？

（２）信頼性：指定された条件下で利用するときに，指定された達成基準を維持するソフトウェアの能力
　→欠陥が少なく，障害発生時には目標値内に回復できるか？

（３）使用性：指定された条件下で利用するときに，理解・習得・利用でき，ユーザーにとって魅力的であるソフトウェアの能力
　→利用するユーザーにとって理解しやすく，かつ使いやすいか？
　　たとえば，用意したマニュアルと利用環境だけで，想定時間内に操作方法を習得することが可能か？

（４）効率性：明示的な条件下で，使用する資源の量に対比して適切な性能を提供するソフトウェア製品の能力
→時間や資源を適切に利用しているか？
（５）保守性：修正のしやすさに関するソフトウェア製品の能力
→解析や修正がしやすく，その修正の妥当性を判断できる手段（テストなど）があるか？
（６）移植性：ある環境から他の環境へ移すためのソフトウェア製品の能力
→決められたコスト内で新しい環境へ移植できるか？また、移植を実現するために必要な規格に準拠しているか？

ソフトウェアの品質を考える際には上記の「ISO/IEC9126」，「JIS X0129-1」といった「ソフトウェアの品質評価のための規格（ソフトウェア製品の評価品質特性およびその利用要領）」で定義されている6つの品質特性やその副特性について確認する必要がある．

また，システム全体の品質を考える際に密接に関係してくるのがセキュリティである．

セキュリティの示す範囲は非常に多義に渡るが，情報セキュリティについてまとめたものに「JIS X5080」（情報セキュリティマネジメント）がある．

「JIS X5080」（情報セキュリティマネジメント）は「ISO/IEC17799」を翻訳し，情報セキュリティマネジメントのためのガイドラインとして制定されたもので，ISMS(Infomation Security Management System)適合性評価制度の基盤として利用されており，10のマネジメント領域を定義しており，その中でも「システムの開発および保守」や「通信および運用管理」などはシステム開発の品質と密接に関係してくる内容であるため，これらの内容についても確認する必要がある．

このような標準の規格などを確認せずに我流でルールや基準を決めてしまうと，本来決めるべき事項が抜けてしまい，品質を維持管理することが困難になることも考えられるため，上記規約や規格を前提におき，社内あるいは開発プロジェクトで定める開発基準やルールを明確にすることが重要である．

2.2 システム品質の定義

業務システムに求めるられる品質には，システムを利用するユーザー側が客観的に評価する「外部品質」と，システムを開発・保守，運用する開発者側に必要となる保守性や信頼性といった「内部品質」といわれる品質があり，ユーザーが満足のいく外部品質を保障するためには，内部品質が高いシステムである必要がある．

（１）システム品質の定義

- 使いやすい
- 不具合がない（少ない）
- 汎用性、柔軟性が高い
- 独立性が高く、変更が容易
- 作りがシンプルで、保守・拡張性が高い　など

（2）「最適な品質」

- 品質は高いほうが良いということはいうまでもないが，システム開発にはほとんどの場合，限りあるリソース（予算と納期）で実施する必要があるため，そのリソースに見合った提供可能な品質レベル，いわゆる「最適な品質」について，システム企画・設計のフェーズから十分検討し，ユーザーの理解を求め合意しておく必要がある．

（3）「システム品質の評価」

- ソフトウェアの品質は一般的にテスト工程で評価するが,テストで不具合が無いということだけではシステム全体としての品質が良いということにはならず,システムをリリースしてからでなければ発見できない不具合もあるため，リリース時点ではソフトウェア品質の直接的な評価は困難な場合もある．
- そのため，品質の最終的な評価はシステム稼動後にユーザーがその機能性や使用性について納得するもので且つ，そのシステム保守を担当する側にとっても保守性や信頼性の高いものになっているかどうかで評価するべきである．

2.3 システムの保守性

冒頭でも述べたように弊社の業務内容はシステムの開発・保守，運用が主力事業であり，私の所属する部署でも，業務システムの開発･保守が主な業務である．

そういった立場から，上記6つの品質特性を確認してみると，その中でも「保守性」については業務遂行上，特に重視しなければならない特性であるといえる．

「保守性」には以下の副品質特性が定義されており，保守性の高いシステムとはこれらの特性を満たしている必要がある．

（1）解析性：障害や改訂の対象箇所の識別しやすさ
（2）変更性：実際の改訂、障害除去のしやすさ
（3）安定性：改訂時の予期せぬ影響の出にくさ
（4）試験性：改訂時の妥当性検証のしやすさ

業務システムに限らずシステムは，開発に利用する技術の推移や法改正，ビジネスルールやモデルの変化など，さまざまな要因により要求の追加・変更が発生するため，常に保守・メンテナンスを施しながら稼動しており，そのライフサイクルのほとんどを，保守工程で過ごすといっても過言ではない．

また，システム開発，保守・メンテナンスの要因となる「ビジネス要求」と「開発技術」は，ともにそのサイクルを縮め，日々変化していることも事実である．

システム開発とは，「ビジネス要求」と「開発技術」を結ぶものといえるが，この二つが変化のない状態を想定して，開発や保守・メンテナンスを行うことは現実的ではない．

このような変化を柔軟に受け入れられるようなシステムにするためにも，企画や設計フェーズの段階から稼動後の保守性を意識して，システム開発を行う必要がある．

３． プログラム開発の品質

3.1 コーディングルール

コーディングルールは，プログラムの品質のよさや保守性を高めるために作成されるもので，開発者がコードを書くうえでよりバグの少ない，また後の保守効率を上げるためにプログラムコーディングの具体的な指針をまとめたものである．

プログラム開発言語は提供する機能が豊富で，同一機能を複数の方法（ロジック）で構築することができる反面，開発者それぞれが自分の知識や経験のみで開発すると，本来ならば簡単なロジックで実装できる機能（処理）でも我流で実装し，プログラム全体を解り辛いものにしてしまうこともある．また，同一システム内に異なるロジックで実装した同一機能が混在すると，プロジェクトあるいはシステム全体で纏まりが無く，保守性や生産性が著しく低下することになるため，避けるべきである．

特に弊社では，保守業務を行う割合が高く，他の開発者が作ったプログラムをメンテナンスするケースが多いため，万一品質の悪いシステムの保守を担当するようなことになってしまった場合，担当開発者あるいはその開発プロジェクトメンバー全体のモチベーションは下がり，保守・生産性は更に低下することになる．

このような事態に陥らないためにも，しっかりとした根拠のある開発基準やコーディングルールをプログラム開発工程に入る際に決めておく必要がある．

これらを決める際には，プロジェクト全体として掲げているシステム全体の品質を基準として決める必要がある．

いうまでもないが，開発工程の途中で開発基準やコーディングルールを大幅に変えることは非常にリスクを伴い，無駄なコストもかかることになり，品質も下がる．

（１） コーディングルールを決める際の考慮点

- 開発メンバーが理解・利用しやすい
- ベンダーが決めている規約や基本的約束事（暗黙のルールとなっている）は守る
- 自分勝手な書式やルールは決めない
- 品質やセキュリティについて考慮する

（２） コーディングルールとして決める基本的項目

- 命名規準（オブジェクトや変数名など）書式の統一
- 基本的なソースコードのフォーマット（改行，字下げの量など）の統一
- 機能の配置（ビジネスロジック，データロジック）
- 共通部品（ロジック，アルゴリズム）の提供，および利用方法（サンプル）

（３） コーディングの際の考慮点

- 他の人に解りやすくすることを常に意識する
- ソースで内部仕様を他の人に伝えらるぐらいの気持ちでコーディングする
- 論述的にならず，仕様書に書かれたとおりの流れがソース上で整然と簡潔に見えるものを目指す
- 無駄（意味の無い）や余計なコードは書かない

次節以降では，これらの内容を考慮して人事システムで採用した開発基準やコーディングルールの一部を例にあげ，そのルールを決めた際の観点などについて説明する．

3.2 共通部品

プログラムで多用される機能は共通部品として提供し利用することで，開発者の負担軽減が図れ，処理ロジックのソースコードへの実装レベルが統一されるため可読性も向上し，結果的にシステム全体の統制が取れ，システムの品質が向上する．

人事システムでも共通部品化可能な機能については積極的に対応を行っており，その一例として，システム稼動の源ともいえる「入力データ」の属性チェック処理について確認してみる．

オンラインあるいはバッチ処理の入力データは，セキュリティ面からみても取り込み時にチェックを行い，想定外のデータがシステムに取り込まれるのを確実に防ぐような仕組みを実装しておく必要がある．

これらのチェック処理を各プログラムごとに，それぞれ実装した場合，機能の実装レベルがバラバラとなり，脆弱性の原因となってしまう恐れもある．またプログラムソースの可読性が下がり，システム開発における生産性や保守性などの品質も下がる原因となる．

これらを解決するために，表 1 のような入力データの属性ごとの，入力必須項目か任意入力項目かで使い分けられるように，チェックパターンごとの処理機能を部品化して提供している．また，これらの入力データの属性チェック処理はセキュリティの観点から ASP やデータロジックではなく，ビジネスロジック内の入力データの相関チェック処理前に実装することをルール化した．

（各データ間の相関チェックについては業務システムに依存する内容であるため，ここでは割愛する．）

属性	チェックパターン	
文字列	文字固定	最大文字数指定
半角文字列	文字サイズ固定	最大文字数指定
全角文字列	最大文字数指定	
日付	日付時刻	日付文字
長整数	長整数	長整数文字列
電話番号	電話番号	
郵便番号	郵便番号	
半角カナ文字	最大文字数指定	
１０進数データ	１０進数データ	１０進数文字列
半角数値文字列	文字サイズ固定	最大文字数指定
テキストエリア文字	最大文字数指定	
入力不許可		

表 1 入力データチェックパターン

3.3 端数計算

人事システムでは円単位の金額計算処理を多く行うため，四捨五入や切上げ・切捨てといった端数計算処理ロジックを多く実装する必要がある．このような言語の提供する関数を使って実装できる処理についても，共通部品化と同様の観点から具体的な実装ルールを決めることで，実装が開発者のスキル任せにならないようにしなければならない．

人事システムで採用したルールでは，デバックのしやすさや可読性の向上を図る目的でLIST 1 のように，あえて言語の提供する四捨五入等の関数を使わず，ロジックを明示的に記述することにしている．

この他にも年齢の算出ロジックや条件分岐・反復などの基本的なアルゴリズムの記述方法についても同様の目的でルール化を行い，サンプルを提供している．

```
Dim decGaku   As Variant

'*****---------------------------------------*****
strFxxCxxxZExp = "桁揃え処理"
decGaku = CDec(decGaku) * CDec(100)

'*****---------------------------------------*****
strFxxCxxxZExp = "四捨五入処理"
decGaku = CDec(Int(Abs(decGaku) + CDec(0.5))) * Sgn(decGaku)

'*****---------------------------------------*****
strFxxCxxxZExp = "切上処理"
decGaku = CDec(Int(Abs(decGaku) + CDec(0.99999))) * Sgn(decGaku)

'*****---------------------------------------*****
strFxxCxxxZExp = "切捨処理"
decGaku = CDec(Int(Abs(decGaku))) * Sgn(decGaku)

'*****---------------------------------------*****
strFxxCxxxZExp = "桁戻し処理"
decGaku = decGaku / CDec(100)
```

LIST 1 端数計算ロジック

3.4 データベース操作

業務システムで扱うデータのほとんどは，リレーショナルデータベース（以下DBと略記）を用いて管理するのが一般的である．

システムでDB内のデータを操作するためにはSQL文を記述する必要があるため，その記述方法についてもルール化する必要がある．

プログラムソース可読性の向上や保守性，セキュリティなどの観点から検討し，以下のような内容をルール化し，その記述方法についてLIST2のようなサンプルを提供をしている．

（1） ASP内にDBへのアクセスロジックは記述しない

（２） 複雑なSQL文は特別な場合を除き記述しない
- 異なるテーブル同士を結合する「JOIN」句
- SELECT文を統合する「UNION」
- 副問い合わせ（サブクエリー）相関副問い合わせ（相関サブクエリー）

（３） データ抽出条件はパラメタクエリで指定する

（４） Where句でのデータ抽出条件は最低限にし，データ判定条件はプログラムロジック内に記述する　　など

```
レコードの取得(select文)
  '*****-----------------------------------*****
  strExp = "レコードの取得内容を記述する"
  With comSQLServer
      .CommandText = "select " & 項目名１ & "," & 項目名２ & ･･･_
                     " from " & MVstrTarget & ".." & ﾃｰﾌﾞﾙ名 & " (nolock) " & _
                     " where " & 項目名ａ & " = ? and " & _
                                 項目名ｂ & " = ? ･･･"
      .Parameters(0) = 項目名ａの抽出条件の値を記述する
      .Parameters(1) = 項目名ｂの抽出条件の値を記述する
              ⋮
  End With
  Set ﾃｰﾌﾞﾙ名 = comSQLServer.Execute

   (ﾃｰﾌﾞﾙ名／項目名は、標準ﾓｼﾞｭｰﾙで定義されているﾃﾞｰﾀﾍﾞｰｽ項目名を使用)
```

LIST 2 SQL 文の記述例

3.5 プログラムソースのコメント

プログラムソース内に処理概要など，コメントの記述やロジック内の非標準的な部分に適切なコメントを記述することで，プログラムソースの可読性を向上させることができるが，むやみやたらにコメントを記述し過ぎたり，記述したコメント内容とロジックの振る舞いが異なっていると，他の開発者がメンテナンスや影響調査を行う際に混乱を招き，保守性を下げることになるため，コメントは適格かつ正確に記述する必要がある．

人事システムでは，後述する例外および障害発生時に迅速に対応する目的で，多少コメントは増えることになるが，意図的にプログラム内の処理ステップの内容および異常終了時の内容をコメント記述用の変数にセットするようルール化している（LIST 3）．

（１）プログラムソースコメントを書く際の考慮点
- 非標準的な記述をした場合の説明を記述する
- 必要最低限にする：ソースコードで容易に読み取れるものは、かえってソースを読み辛くするので記述しない
- 嘘を書かない：ソースコードとコメントが一致しなければ混乱を招き保守性の低

下を招く

- コメント変数には処理ステップの内容および異常終了内容を正確に記述する

```
処理のコメントおよび異常終了時のメッセージを記述する。
   '*****-----------------------------------*****
   strFxxCxxxZExp = "処理内容をコメントとして記述する"
                ⋮
       strFxxCxxxZMsg = "異常終了時の内容を記述する"
       GoTo LabelError
                ⋮
```

LIST 3 ロジック内のコメント変数

3.6 例外および障害発生時の振る舞い

アプリケーションは想定された，あるいは想定外の障害が発生した場合に適切に振る舞うように作られている必要があり，エンタープライズレベルの業務システムではデータチェックや例外処理ロジックがプログラムソース内の大半を占るといわれている．

ただ，機能要求仕様を決める際に正常系処理は十分検討されるが，例外処理については重要であることは理解されているにも関わらず，十分な検討がされないまま，あるいは後回しにされたまま開発作業を行ってしまうことが少なくない．

人事システムでは例外処理や障害（以下エラーと記載）が発生した場合に，その発生時刻，場所および内容と原因を迅速，且つ容易に特定できるように，コメント変数の採用や（ LIST 3）前節に掲載））エラー処理の記述方法についてルールを決めており，その記述方法についてLIST 4 のようなサンプルを提供している．

言語に依存する内容になるが，VBにはOn Errorステートメントという例外によって処理を分岐する機能があり，このOn Errorステートメントを正しく利用することで，例外処理の対応を実装できる．

On Errorステートメントは，Resume Nextキーワードを使ってエラーが発生しても処理を次のステップへ進める使い方と，GoToキーワードを利用して指定したラベルへ分岐させる方法があるが，ResumeNextを利用した場合のロジックは煩雑で，コンポーネントに対する呼び出しの都度，呼び出し元のプログラムで複数行のエラーチェックロジックを記述する必要があるなど，可読性も悪くなり現実的ではない．

また，プロシージャの先頭でOn Error Resume Nextを宣言し，そのままエラーチェック無しで処理することもVBの機能としては実装可能ではあるが，業務システムで使うプログラムとしては特殊な場合を除き，そのような利用はしない．

業務システムでは，少なくともエラーを検出したらその内容をエラーログに出力し，そのプロシージャを抜けるようにGoToキーワードを利用すべきである．

```
ﾌﾟﾛｼｰｼﾞｬ内にはｴﾗｰ処理を記述する。
   On Error GoTo LabelError
       Dim conSQLServer        As Connection
       Dim comSQLServer        As Command
       Dim objFxxCxxx1         As FxxPxxxx.FxxCxxx1
       Dim strFxxCxxx1ConStr   As String
       Dim objFxxCxxxZ         As FxxPxxxx.FxxCxxxZ
       Dim strFxxCxxxZExp      As String
       Dim strFxxCxxxZMsg      As String
       Const strFxxCxxxZMethod As String = "ﾌﾟﾛｼｰｼﾞｬ内()"

            ⋮

       ﾌﾟﾛｼｰｼﾞｬｰ名 = 0

       Exit Function

   '*********************************
   LabelError:

       ﾌﾟﾛｼｰｼﾞｬｰ名 = 9

       Set objFxxCxxxZ = New FxxPxxxx.FxxCxxxZ
       With objFxxCxxxZ
           .Iam = MVstrIam
           .Server = MVstrServer
           .Target = MVstrTarget
           .Location = MCstrLocation
           .Method = strFxxCxxxZMethod
           .Explanation = strFxxCxxxZExp
           .ErrNo = Err.Number
           .Description = Err.Description
           .Source = Err.Source
           .Msg = strFxxCxxxZMsg
       End With
       Call objFxxCxxxZ.RaiseError
       Set objFxxCxxxZ = Nothing

       Exit Function

   End Function
```

LIST 4 エラー処理の記述

VBのErrオブジェクトにはエラーの発生個所を示す情報が含まれない（最新バージョンのVBでは、もっとエラートラップが容易になっているはずである．）ため，人事システムで採用した，コメント記述用変数にセットしている処理内容や，異常終了時の内容などをErrオブジェクトの情報と併せてエラーログ（図 2）に出力することにより，運用担当者がエラーの対応を迅速且つ，容易に行えるようにしている．

このエラーログは運用監視画面で確認可能であるが，迅速な対応を実現するためにエラー発生の都度，運用担当者が登録したメールアドレスへも，その内容を通知する機能を実装している．

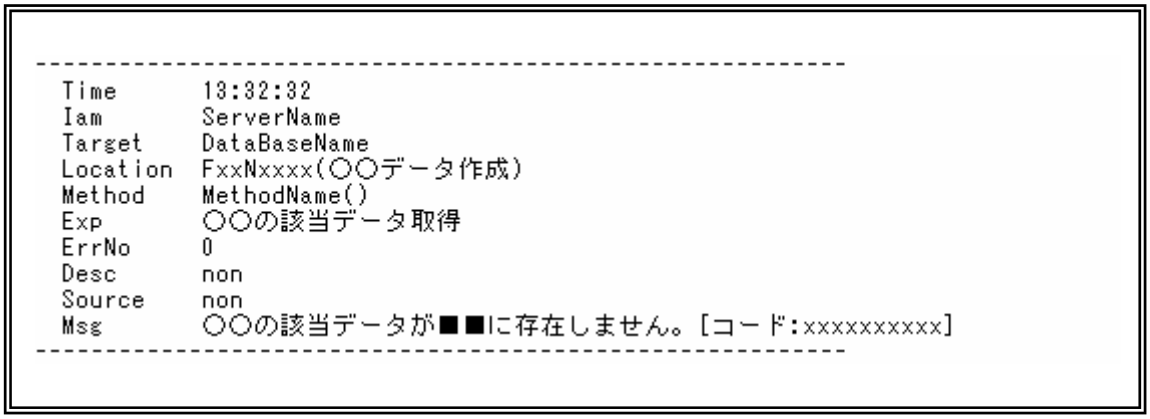

図 2 エラーログ出力内容

４． まとめ

人事システムの再構築プロジェクトは，与えられた開発期間と費用で要求される機能をほぼ全て実装することができた．また，稼動後 4 年間の運用中も毎年のように行われる法改正や社内の制度変更など，多くの変更要求に対応するべく開発・保守・メンテナンスを繰り返しながら，現在も安定稼動を継続している．

このことからも，人事システムで採用した発基準やコーディングルールは品質維持のために有効に機能しており，システム全体としても保守性のよいシステムであると評価できる．

今回はプログラム開発工程の品質管理という視点から，筆者が開発プロジェクトに参画し，現在も保守を担当しているシステムの開発・保守工程で採用している開発基準やルールの一部を紹介させて頂いたが，人事システムというシステムの性質上あまり具体的な表現をしておらず，記載内容が曖昧で理解しづらくなっている部分もあったかと思うが，どうか事情をご理解頂きご了承願いたい．

ただ，今回紹介したほとんどの内容はインフラやアーキテクチャー，言語などの環境が異なっても実装・適用可能な基本的事項であり，業務システムの開発あるいは保守を担当される開発チームのリーダー，あるいは開発者の方でシステムの開発品質について検討されている方の少しでも参考になれば幸いである．

５．　おわりに

弊社においても業務システムの新規導入や再構築におけるプログラム開発などの下流工程の作業は，開発期間やコスト重視の観点からベンダーやソフトウェア会社に外部委託するケースが多く，若手社員がプログラム開発やテスト，デバックなどといった下流工程を経験する機会が減っており，上流工程からプロジェクトに参画するケースが多くなってきている．

プログラム開発やデータベース設計などの実務を経験し，机上の知識のみでは習得できない技術やスキルを習得しておくことは，システム企画や設計などの上流工程の業務を遂行するうえでも必要であり，後の実務に必ず役立つはずである．

これから企業のシステム部門を担う若手技術者の方々は，自ら積極的にチャンスを見つけシステム開発における様々な技術やスキル習得に果敢にチャレンジし，企業経営戦略に必要とされるシステムの導入に自分のスキルやノウハウが有効に活かされるよう，日々努力されることを期待する．

最後に，今回の論文執筆にあたり協力を頂いた，上司をはじめチームメンバーの方々，ならびにアドバイス・情報提供頂いた多くの方々へ，この場を借りて感謝の意を表したい。

参考文献

［１］『開発の現場《特別版》』vol.001 The 実装技術，翔泳社,(2007.03.16)