データベースメトリックスの活用
－システム内部品質の向上に向けて－

陰山 泰之†

† 日本アイ・ビー・エム株式会社 〒103-8510 東京都中央区日本橋箱崎町 19-21
E-mail: † e01104@jp.ibm.com

あらまし 本稿はアプリケーションのデータベースシステムの内部品質を計測し，評価するための指標を客観的な観点から数値として取得し，それをデータベースの内部品質の維持・向上に役立たせるための方法について論じる．最初にソフトウェアの内部品質について説明し，データベースシステムの品質評価の指標として，主キー項目比率，テーブルあたり項目数，CRUD マトリックスの密度の三つを選ぶ．次にあるシステムの計測値をもとにこれらの指標で得られた数値からデータベースシステムの内部品質についての推測を立て，最後に，そこからの品質維持・向上策について論じる．

キーワード データベース，内部品質，客観的指標，正規化，CRUD マトリックス

Utilization of Database Metrics
－Towards Enhancement of System Internal Qualities－

Yasuyuki KAGEYAMA†

† IBM Japan Ltd. 19-21, Nihonbashi Hakozaki-cho, Chuo-ku, Tokyo, 103-8510 Japan
E-mail: † e01104@jp.ibm.com

Abstract In this paper I argue about the method of utilizing database metrics as objective indices for evaluating database internal qualities, and for maintaining and enhancing them. I first explain the internal qualities of software systems, and then select the ratio of key items, number of items per a table, and the density of CRUD matrix as main metrics for evaluating database internal qualities. Based on the values obtained from real measurements of a certain system, I conjecture its internal qualities, and finally argue about the methods for enhancing them.

Keyword Database, Internal Qualities, Objective Indices, Nominalization, CRUD Matrix

1. はじめに

アプリケーションシステムを構成する既存のアプリケーションプログラムに対してリファクタリングが唱えられるようになってきた昨今では，「システムは動きさえすればよい」，「動いているシステムには触るな」という旧来の考え方は成り立たなくなってきている．たとえ外からは見えなくても，プログラムの保守性，可読性，拡張性などのシステムの内部品質を重視する動きが浸透しつつあり，プログラムの内部品質を測る指標が活用され，これを評価する動きも盛んになりつつある．

けれども，システムはアプリケーションプログラムだけで成り立っているわけではない．日々の業務をこなすシステムにおいては，業務を処理するアプリケーションプログラムの他に，業務の情報を格納してあるデータベースも，システムの構成要素として極めて重要な位置を占めている．むしろ，多くのプログラムが共通にアクセスする構成要素としてデータベースは共通基盤的な役割を担っており，この意味ではデータベースは個々のプログラムより重要なコンポーネントであると言える．

この点からすると，システムの内部品質を語るうえでは，プログラムの内部品質だけでは不十分であり，データベースについてもその内部品質を客観的に評価し，維持・改善していく動き，活動が必要である．

本稿では，このような観点から，全体の議論のベースとなるシステムの内部品質についてポイントとなる事項を明らかにすることから始めて，データベースの内部品質を測るための指標をどう評価し，そこからなにが推測できるか，またデータベースの内部品質の改善のためどのような手を打つことができるかなどについて，実際のアプリケーションシステムのデータベースについて調査し，その調査結果を活用した事例をもとに報告し，検討する．

2. ソフトウェアシステムのバスタブ曲線

2.1. 故障率の経時変化

時間の経過とともに故障率などのハードウェアとしての工業製品の品質がどのように変化するかは，一般に「バスタブ曲線」で表わされる．

製品を使い始めた頃は，製造過程で紛れ込んだ不具合が顕在化する初期故障が発生する時期（第Ⅰ期）となる（もっとも，最近では製造技術が進んだため，初期故障の発生率は低くなってきている).やがてその時期を過ぎると，故障が発生しにくくなる比較的安定した時期（第Ⅱ期）に入る．しかしながら，やがて製品を構成する部品の摩耗，劣化，金属疲労などにより摩耗故障が増えて，また故障の発生頻度が高くなっていく（第Ⅲ期).

これらの時期の故障の発生率を曲線として描くと，右肩下がりで始まって水平線になり，やがて右肩上がりの曲線になって，全体として故障の発生率のグラフは図 2-1 のようにバスタブに似たかたちになる．

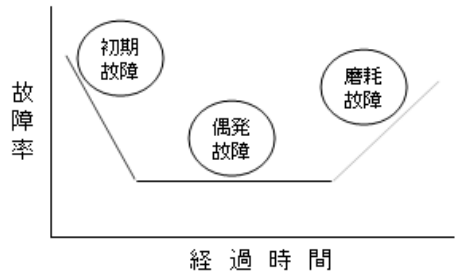

図 2-1．故障のバスタブ曲線

2.2. ソフトウェアのバスタブ曲線

このようなバスタブ曲線は，ふつうハードウェア製品に対してあてはまるものと考えられており，摩耗，劣化しないソフトウェアシステムにはあてはまらないとされている．

しかしながら，どのようなシステムも長い間使い続けていれば，とくによく使われているシステムほど，その品質はハードウェアと似たようなバスタブ曲線を描き,その品質は経過時間にともなって劣化していく．ただし，それは部品の摩耗や劣化によってではなく，改修，機能拡張などの積み重ねにともなって内部品質が劣化していくということである(図 2-2 参照).

どんなによく管理されたソフトウェアでも，長い間使い続けていれば、機能改修や機能拡張にともなって自然にコード量は増加し，保守性は低下していく（たとえば，UNIX の Free BSD のカーネルとユーザープログラムについても，このような傾向が見られることが報告されている[1]).

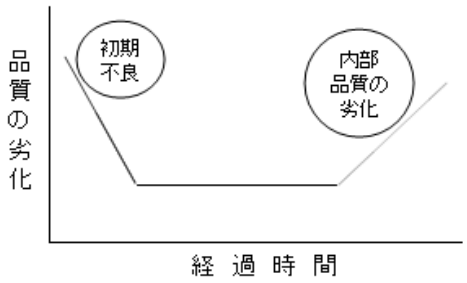

図 2-2．ソフトウェアのバスタブ曲線

たとえば，機能改修や機能拡張の際に，適切な設計を行わないで該当箇所にいきなり必要なコードを書きこむようなことをしてしまうと，図 2-3 に示したように，メソッドやクラスのサイズや複雑さが徐々に肥大化していき，その結果としてコードの品質が低下していく可能性があるわけである．長年保守されたソースコードを見ると，日付と担当者名を含んだコメントで挟んであるコード片を見かけることがあるが，そのようなコードを含むメソッド，関数はたいていの場合，かなりの長さ，複雑さに達している．

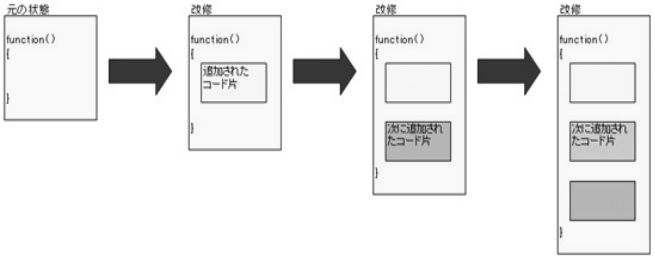

図 2-3．改修の積み重ねによるコードの肥大化

とくに機能的に重要なコードは改修の頻度が高いので，重要な部分ほどコードの品質が低下するという傾向があるが，そのような重要な部分は利用頻度が高い機能なので，その部分の品質低下はシステム全体の品質を左右することにもなりかねない．

3. 外部品質と内部品質

3.1. ソフトウェアの内部品質

一般に品質について語られる場合，それは使うときに主にユーザーから分かる品質,つまり機能の豊富さ，機能の正しさ，機能の使いやすさなどの外部品質に言及されている．

これに対して，ソフトウェアシステムを作るとき・直すとき・拡張するときに主に開発者に分かる品質，構造のわかりやすさ，機能改修・追加のしやすさ，テストのしやすさ，整合性のとりやすさなどの特性は内

部品質ということになる(図 3-1 参照).

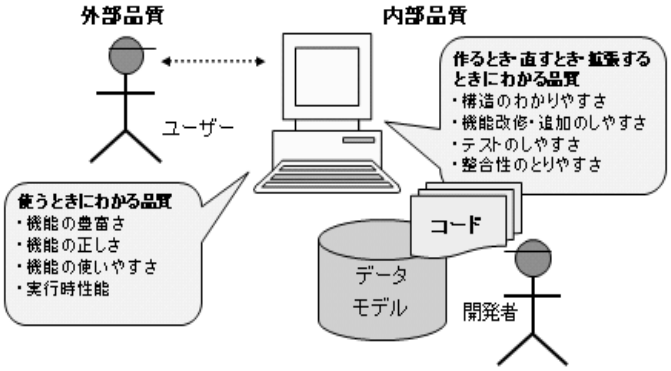

図 3-1. 外部品質と内部品質

3.2. ソフトウェアの内部品質

外部品質の低下は不具合としてただちに顕在化するので、すぐに対策がとられる一方で，内部品質の低下は外からは見えにくいので、放置される場合も少なくない．だが，内部品質が伴わなければ、実質的には外部品質の確保も困難である．内的品質の構造に関する品質が低下すると，保守性・生産性が低下し，結果として改修の影響を十分に調べ切れないうちに改修を加えたり，影響範囲が不明なままテストを実施したりするなど，内部品質が低下して信頼性などが低下し，最終的にはそれが外的品質の低下として顕在化してくるからである(図 3-2 参照).

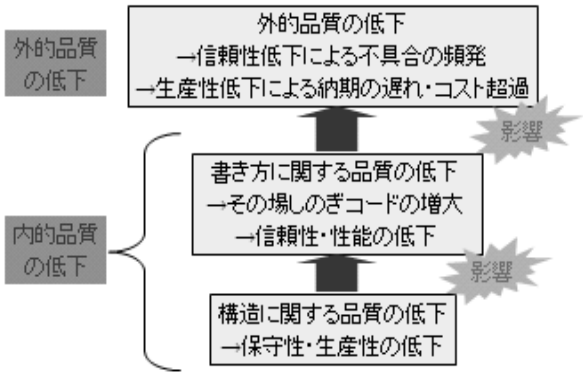

図 3-2. 品質低下の連鎖

バスタブ曲線の第Ⅲ期はソフトウェアの陳腐化によってもたらされるという見方もあるが[2]，何十年も使われているレガシー系のシステムに見られるように，保守が可能な間は，必ずしも第Ⅲ期が訪れるとは限らない.むしろ第Ⅲ期の招来を不可避にしてしまうのは，内部品質の劣化であると言える.

内部品質の低下にともなう外部品質の低下という事態を避けるためにも，構造に関する内部品質を早い段階で評価してその低下を早期に防止しておく必要がある．そしてこのことは，冒頭でも述べたように，アプリケーションプログラムだけでなく，データベースシステムについても当てはまることである.

4. データベース評価の指標と方法

4.1. 品質評価の指標

プログラムコードについては，早くから LOC, NCSS などの規模に関する指標があり，さらにコードの複雑さを表わすサイクロマチック数などの指標，さらにはハルステッドのメトリックスなども使われるようになっている．たとえば，サイクロマチック数と欠陥発生率の間にはある程度の強い相関があることが研究により明らかになっているので，コードの複雑さはプログラムの内部品質を測る重要な指標のひとつであると考えられている[3].

このように，プログラムについては，その内部品質を評価するための客観的な数値で表わされる指標がすでにいくつか開発されており，評価の目安となる数値も公表されていて，実際のプロジェクトの現場で活かされている．一方，データベースについてはどうであろうか.

プログラムコードの内部品質が良くない場合の比喩として，ファウラーはリファクタリングを行うきっかけとしての「コードの不吉な匂い」に言及している[4]．これとまったく同じように，アンブラーとサダラージはデータベースについて以下のような不吉な匂いを指摘している[5].

1.複数の目的に使われるカラム.
2.複数の目的に使われるテーブル.
3.冗長なデータ.
4.カラムの多すぎるテーブル.
5.行の多すぎるテーブル.
6.「スマート」カラム.
7.変更の恐怖.

コードの不吉な匂いと同じく，ここに挙げたデータベースの不吉な匂いも，客観的で定量的な性格のものもあれば，主観的で定性的な性格のものもある.

ソフトウェアシステムも，人間がかかわってくるものである以上，主観的で定性的な評価はおろそかにはできない．けれども，いきなり主観的で定性的な評価から議論を始めると，不毛な結果に陥ってしまう恐れがあるので，内部品質評価の導入部としては，評価・意見の不一致が生じにくい客観的で定量的な属性から着目すべきであろう.

4.2. 正規化の観点

ソフトウェアシステムは複雑であるため，複雑さを管理する定石としては分割統治（divide and conquer）の考え方が有効であり，ソースファイルや関数・メソッドなどのコンポーネント・モジュールの規模の指標は，この分割統治の設計がどれだけ適用されているかを測る指標となりうる．

この考え方によって，プログラムの機能は管理可能なレベルにまで分割され，単純化され，それらの組合せで複雑な機能が実現されることになるので，結果として機能のムダな重複も排除される．

データベース設計においてこの分割統治法に相当するものは正規化であろう．この正規化がどこまで進められているかによって，データモデルがどのような状態かをある程度評価，判断することができる．データベースの正規化の結果はキー項目に関するものなので，数値としては主キー項目の比率などのかたちで現われてくる．また正規化が進めば，エンティティが小さくなっていく傾向があるので，上掲の不吉な匂いの 4 でも言及されているように，テーブルあたりの項目数も品質の指標として取り上げることができる．

4.3. CRUD マトリックスの利用

主キーの比率やテーブルあたり項目数などは，データモデルを設計したてのシステムとしてまだ稼働していない時点，データベースにアクセスするアプリケーションプログラムがまだない時点でも,数値を取得し，評価することが可能である．しかしながら，システムのバスタブ曲線の第Ⅲ期に見られるような時間経過にともなう内部品質の劣化についても評価するためには，これらとは異なる観点の指標も欲しいところである．

そこでそのために，ツールで生成した CRUD マトリックスから取得した値を品質評価の指標として利用することを考えてみた．ここで利用したのは，データディクショナリ（テーブル ID、項目 ID）と SQL 文の情報を元に、DB 項目単位，SQL 文単位に CRUD マトリックスを生成するツールである(図 4-1) [5].

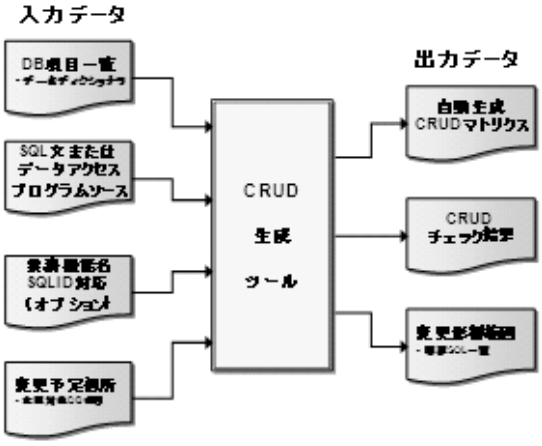

図 4-1. CRUD マトリックス生成ツール

この CRUD マトリックス生成ツールを利用して，主キー項目比率、従属項目平均数、CRUD マークパターンなどを分析したが，ここでは CRUD マークの密度という概念を導入しよう．つまり，CRUD マトリックス全体のなかで C， R，U，D のマークがどれほど生成されているかという指標である．

こうして，正規化の程度と CRUD の密度をキーとして，データベースの品質を評価してみる．

4.4. 複数システムの比較による評価

プログラムの規模を表わす LOC や NCSS，複雑さを表わすサイクロマチック数などについては，得られた値を評価する際の目安として一般的になっている値がある[7]．これに対して，上記で選んだデータベースの指標については，たとえばテーブルあたりの主キー項目の目安などはあるが，評価の目安はプログラムの指標ほどには必ずしも充実していないようである．

そこでデータベースの評価に際しては，複数のデータベースシステムについて取得した同じ指標の数値を

	今回のシステム	自動車メーカーA 設計部品表システム その1	自動車メーカーA 設計部品表システム その2	自動車メーカーA 設計部品表システム その3	自動車部品メーカーB 設計部品表システム その1	自動車部品メーカーB 設計部品表システム その2	自動車メーカーC 生産ライン管理システム	自動車メーカーD 設計部品表システム	精密機器メーカーE 特許情報管理システム	情報通信会社F 料金管理システム
全テーブル数	580	124	194	242	124	156	165	193	321	72
全項目数	15,791	1,619	2,648	4,455	1,996	3,010	1,406	2,928	5,350	2,027
全実質項目数	14,241	875	1,484	3,003	1,252	1,918	1,406	2,757	4,387	1,591
全主キー項目数	1,101	404	736	839	352	443	370	1,111	867	253
全実質従属項目数	13,140	471	748	2,164	900	1,475	1,036	1,646	3,520	1,338
主キー項目比率(対全実質項目数)	7.7%	46.2%	49.6%	27.9%	28.1%	23.1%	26.3%	40.3%	19.8%	15.9%
実質従属項目比率(対全実質項目数)	92.3%	53.8%	50.4%	72.1%	71.9%	76.9%	73.7%	59.7%	80.2%	84.1%
1テーブルあたり平均項目数	27.2	13.1	13.6	18.4	16.1	19.3	8.5	15.2	16.7	28.2
1テーブルあたり平均実質項目数	24.6	7.1	7.6	12.4	10.1	12.3	8.5	14.3	13.7	22.1
1テーブルあたり主キー項目数	1.9	3.3	3.8	3.5	2.8	2.8	2.2	5.8	2.7	3.5
1テーブルあたり実質従属項目数	22.7	3.8	3.9	8.9	7.3	9.5	6.3	8.5	11.0	18.6
アクセスSQL本数	10,188		3,140	1,063		1,755		3,265		457
CRUDマーク数	145,828		27,042	16,865		14,702		38,977		5,068
マトリックス上CRUDマーク発生率	0.091%		0.325%	0.356%		0.278%		0.408%		0.547%

表 4-1. データベースメトリックスの比較対照表

互いに比較して評価するというやり方をとることにした．こうして，実績データに照らして極端な値を示している部分の品質について，その値が極端になった原因を推測していくようにした．

このような準備のもと，あるサービス会社の営業系システムのデータベースのメトリックスを取得した．その結果と，他の企業システムのデータベースの指標を比較対照させたのが表 4-1 である(この表で実質項目とは，テーブル中の項目のうち，更新日時，更新者 ID などのシステム管理上の項目を除いたアプリケーショントランザクションで利用される項目のことを意味している)．

SQL については，必ずしもすべてのデータベースで値を取得できたわけではなかったが，それでも比較するのに十分なデータは取得できている．

表 4-1 の中で，今回の評価対象のデータベースのメトリックスは表の一番左の列にある．

4.5. 主キー項目の比率

まず，この表から一見して分かることは，評価対象の今回のデータベースは，他と比べるとその規模の大きさが際立っているということである．これの右側に並んでいる他のデータベースの規模と比べると，テーブル数，データ項目数，SQL 本数，どれも数倍の規模に達している．もっとも，このシステムの過去の履歴を見ると，そのデータベースは，最初に設計した段階からある程度大規模なものであったが，すでに数年使っていて，その間に徐々に大きくなってきた部分も確かにあった．

このような規模を示す今回の評価対象のデータベースについて，これらの値から計算した他の比率などに関する値について見てみると，次に目立つのは，今回の評価対象のデータベースは，他と比べて主キー項目の比率が異様に低いという点である．

たとえば，そのすぐ右隣にある自動車メーカーA 設計部品表システムその 1 の 46%を超える比率は高すぎるにしても，表の右端の情報通信会社料金管理システムの一番低い比率 16%と比べても，今回のシステムの 7.7%はおよそその半分である．

自動車メーカーA 設計部品表システムその 2 の場合，主キー項目比率がおよそ半分に達するほど高いが，これはデータモデルを最初に設計した時に，かなり徹底して正規化を進めたために，ひとつひとつのエンティティのサイズが小さくなり，その代わりにエンティティ同士の間にリレーションが増えてきた結果であることが分かっている（図 4-2 参照）．

リレーションが増えると，親エンティティの主キー項目が外部キーとして子エンティティに降りてきたり，連関エンティティには連関を結ぶエンティティの主キー項目が外部キーとして含まれてきたりするので，リレーションを表現するための主キー項目が増えてくる．そうすると，相対的に従属項目の比率は下がってくることになる．

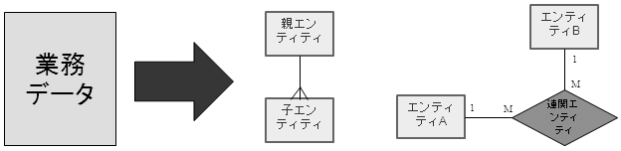

図 4-2．正規化の進展

実際,このシステムもその 2 の直後から運用に入り，その 3 ではすでに実運用に入ってしばらく経った段階で，テーブル数が増えた一方で，主キー項目比率が 27.9%に減っている．傾向の度合いに違いはあるものの，これと同じ現象は自動車部品メーカーB 設計部品表システムにおいても見られる．こちらでもテーブル数が増える一方で，主キー項目比率は減っている．

これらの部品表システムでは，実運用に入ってからは，業務上の必要に応じて後からいくつかテーブルを追加していくことになったが，その際に，時間も限られていたため，他のテーブルとの関連をあまり考慮することができず，必要なアプリケーションとセットで新しいテーブルを設定するというケースが少なくなかった．つまり，正規化の度合いが低下したということである．部品表データはもともと組合せが多いので，他の業務データと一概に比較することは難しいかもしれないが，今回の評価対象のシステムでも，低い主キー項目比率はこれと同じことが起こっているのではないかと推測できる．

この推測は，次に述べる指標によっても裏書されるだろう．

4.6. テーブルあたり項目数

今回の評価対象のデータベースについてさらに言えるのは，一テーブルあたりの項目数が多いということである．これは，先に挙げたデータベースの不吉な匂いの 5 にあたるが，ここから，テーブルの凝集度が低いということがうかがえる．そして，テーブルの凝集度が低いと，以下のような問題が潜在化している可能性がある．

・項目の重複，繰り返しが存在する
・主キーに従属しない項目がある
・推移従属がある

正規化を進めれば，主キー項目によって決まる項目

だけが従属項目として残るので，ひとつひとつのエンティティは小さくなって，そこに本当に必要な項目だけが残る.結果としてエンティティの凝集度が高まる.そして相対的に従属項目の比率は下がってくる.

このように見ると，従属項目が多いエンティティは，そこに本来は必要のない項目が多く含まれている可能性がある．そして，必要のない項目は，複数のエンティティに重複して存在している可能性がある.これは，データ不整合の温床ということもできる．つまり，主キー項目の比率が低くて，従属項目が多いようなデータベース，そのような値を示しているものは，高いサイクロマチック数を示しているプログラムと同じく，問題ありと言えそうである.

4.7. CRUD マトリックスの密度

最後に，経過時間にともなう指標として，CRUD マトリックス上のマーク発生率を見てみる．これは，全項目数×SQL 本数のマトリックス上の全セル数に対して，C，R，U，D のマークがどれだけ発生しているかの比率である.

今回の評価対象のデータベースは表 4-1 の中で行列ともに最大の規模であるにもかかわらず，意外なことに，マークの発生率が最も低い．つまり，評価対象のデータベースは他と比べてCRUDマトリックスでのマークがまばら，疎であるということである.

ここからは，ある特定の目的のために設定されていて,複数のアプリで共有されていないテーブルが多く,同じ意味の項目が複数のテーブルに散らばっているのではないかということが推測できる.

たとえば，システムの機能を拡張するにあたって，参照，格納しなければならないようなデータ項目が既存のテーブルのどこに存在するかをあまりよく調査せずに，新しいアプリケーションプログラム専用のテーブルを新しく設けて，そこにアクセスするようにしてしまうと，そのデータベースはアプリケーションプログラムごとの固有のテーブルの寄せ集めのようになってしまう．このようなデータベースの CRUD マトリックスは，たとえば各プログラムがアクセスしている順にテーブルを並べると，極端に表現すれば図 4-3 のようになるだろう.この CRUD では CRUD マークは対角線上にしか現れないので，マークの発生率は減ってしまい，疎なマトリックスになってしまっている.

	アプリA	アプリB	アプリC	アプリD	アプリE	アプリF
テーブルA	■					
テーブルB		■				
テーブルC			■			
テーブルD				■		
テーブルE					■	
テーブルF						■

図 4-3．疎な CRUD マトリックス

これは，プログラムコードで言えば，機能改修が必要になった際に，図 2-3 に示したように，他に似たような機能がすでに既存のプログラム中に存在するかどうかを調べもせずに，必要な機能を必要な個所にだけ直接実装してしまい，結果として同じような機能を実現するコードがプログラム中に分散して存在してしまう事態に相当するだろう.

実際に，データベースの規模（テーブル数，項目数）が大きくなればなるほど，それだけ調査範囲が広くなるので，調査し切れずにこのように新規アプリケーションごとのテーブルが新設されてしまう可能性が高くなってくる.

これに対して，新しい機能を追加する際に既存のテーブル，DB 項目を綿密に調査して，既存の項目を使うようにすれば，複数のアプリケーションから共通にアクセスされるテーブル，DB 項目が多いと，各テーブル，DB 項目行あたりの CRUD マークは増えるようになり,結果としてCRUDマトリックスとしては図4-4のように密になるだろう.

	アプリA	アプリB	アプリC	アプリD	アプリE	アプリF
テーブルA	■	■	■			■
テーブルB		■	■		■	
テーブルC			■			
テーブルD		■		■		■
テーブルE			■	■	■	
テーブルF	■	■	■			■

図 4-4．密な CRUD マトリックス

CRUD マトリックスが疎になってしまったことの原因として以上の推測が当てはまるとすると，そのようなデータベースはたとえば同じ意味のデータ項目が複数個所に散らばってしまっている可能性がある．このような状態はやはり，先の低い主キー項目比率や，項目数の多いテーブルなどと同じく，データ不整合の温床となりうる．プログラムコードで言えば，コピーペーストが積み重ねられていった結果，コードクローンが多数存在するような状態とほぼ同じようなケースであると言うこともできるだろう.

5. 内部品質向上へ向けての対応策

5.1. データベース内部品質改善の難しさ

以上のような指標によって，プログラムコードと同じく，データベースについてもその内部品質を客観的な数値で計測し,評価し,その結果を手がかりとして，データベースの内部品質を維持・向上させていくことができるだろう．そして継続的にそうした値を計測，評価することにより内部品質の推移を知ることも可能である．

しかしながら，データモデルを設計しなおして品質を向上させる場合には，個々のプログラムのリファクタリング以上に影響範囲が広く，なかなか難しい面がある．

データベースはふつう複数のアプリケーションプログラムからアクセスされていて，それらとの結合度が高い．このため，すでにシステムとして稼働してしまっているデータベースで，メトリックスの値がよくない部分を改善するためには，データベーステーブルだけでなく，それにアクセスするアプリケーションプログラムも改善する必要があるからである．

けれども， 先に述べたように，システムの内部品質は放置しておけば時間とともに劣化していく．とくにメトリックス値のよくない部分のデータベースにアクセスしなければならないアプリケーションプログラムは，たとえそのプログラム自体をリファクタリングして改善できたとしても，アクセス先のデータベースがそのままであれば，いつまでもデータ不整合の問題などに悩まされ続けてしまう恐れがある．この意味では，やはりデータベースの内部品質も放置すべきではなく，なんらかの対策を講じるべきである．

5.2. 内部品質改善のためのポートフォリオ

このようにデータベースの内部品質については，改善の難しさと改善の必要性の両方がつきまとう．そこで，実際の内部品質改善作業にあたっては状況を整理して具体的に対応していくべきだろう．

表 4-1 はデータベース全体としてのメトリックスであったが，計測対象をある程度，テーブルごと，あるいはテーブル群ごとに分類して値を取得し，得られた結果を改善の難易度と改善の効果を二軸にして，改善対象を四分類にした図 5-1 に示したようなポートフォリオに配置して，改善作業に取り組むのが効率的だろう．

このポートフォリオにおいて，改善の効果がある部分とは，内部品質のメトリックス値がよくない部分である．値がよくなくても，あまり使われていない部分であれば，大した問題ではないが，問題が解決されて効果が出るということであれば，やはり利用頻度の高い部分であるということもできる．

難易度	高	①	②
	低	③	④
		小	大
		効果	

図 5-1．品質改善作業のポートフォリオ

これを逆に考えてみると，改善効果の高い部分は，これを放置した場合に危険度の高い部分でもあるとも言える．したがって，効果が大で難易度の低い④の部分から始めるのがよいだろう．順番としては，④から始めて③に至り，最終的には②を目指す形になる．

なお①の部分については，改善に使える工数との兼ね合いによって，場合によっては当面は放置しておなざるをえないかもしれない．

5.3. CRUD による未使用テーブルの除去

以上のポートフォリオを使って，改善に取り組む順番が決まったら,まずてっとり早くできる対策として，メトリックス値を取得するのに利用した CRUD マトリックスを使って，どの SQL からもアクセスされていない項目・テーブルを除去することができる．

システムを長い間使っていると，データベースも未使用項目，未使用テーブルが発生してしまう可能性がある.これらはプログラム中の未使用コードと同じく，可読性を落としている．物理的に定義されているが，現状ではアプリケーションから使われていないテーブルの有無をチェックし，あれば，それの必要性を検討して，不必要であれば削除する．これらが本当にどこからも使われていないということが確認できれば，除去してしまう．こうすることによって，テーブルの凝集度が高まる可能性がある．

こうすることにより，不要なテーブル数が減ることで，システムの改修，機能拡張にともなう調査範囲が減り，影響範囲調査などの人的工数が低減する．さらには，固定長項目を多数含むテーブルの場合など，うまく行けば，たとえばディスクの容量に余裕ができる可能性もある．

5.4. CRUD によるテーブル項目の整理

さらに同じ意味の項目が散らばっていて，CRUD マトリックスが疎になっている場合は，同じ意味の項目をまとめる必要があるが，これについても作業工数を削減するためには，アクセスしている SQL を CRUD

で確認して，基本的にはアクセスの多い方に少ない方を寄せるというかたちにすべきだろう．そのための前段階の作業として，物理的に定義されているが，現状ではアプリケーションから使われていない DB 項目の有無をチェックし,あれば,それの必要性を検討して，不必要であれば削除する．

5.5. CRUD による SQL の共通化

CRUD マトリックス上のマークにはデータベースのデータ項目についての情報という意味だけではなく，そこにアクセスしている SQL 文についての情報という意味もある．そこで，CRUD マークを調査して，図 5-2 が示しているように，同じ CRUD マークを示している SQL があれば，それらの SQL は同じ内容であると推測することができる．とすると，こうした調査の結果，重複した SQL を除去できる可能性がある．

図 5-2．同じ CRUD マークパターンを示す SQL

たとえ CRUD マークがまったく同じではなくても，述部（WHERE 句以降）が同じ CRUD マークの SQL を調査し，同じ内容の SQL であれば重複を除去する（図 5-3 参照）．取得項目に差異があっても，検索条件が同じであれば，ひとつにできる可能性がある．

図 5-3．述部が同じパターンの SQL

これにより SQL の共通利用が可能になり,SQL の全体本数が低減する．結果として，機能改修にともなう調査範囲を絞ることができて，調査工数を削減することができる．また，SQL に対して改修・改善を施さなければならなくなった場合に,同じ複数の SQL を改修する必要がなくなるので，改修・テスト工数を削減することができる．さらに，同じ CRUD マークの複数の SQL の実行計画が別々になってしまっている場合には,これらをひとつにすることによって，プロシージャキャッシュの容量を増やすことができる．

6. おわりに

このようにできるところから少しずつ手を付けていくことにより，データベースの内部品質も改善させていくことができるだろう．

放置しておけば時間とともに劣化していく内部品質について対策を施すことで，外部品質の低下を防ぐことにつながる．内部品質維持・向上のためにかける工数は，中長期的には外部品質維持の工数削減，あるいは機能拡張の生産性向上というかたちで効果が現われてくる．

このことは，逆に言うとシステムの内部品質改善作業には即効性はあまりないということである．このため，どうしても対策が遅れがちになり，問題が顕在化してくるころにはもう手遅れに近い状態になってしまっているかもしれない．

このような事態を避けるためには，やはり内部品質を示すシステムの指標を常日頃から取得し，評価するようにして，品質低下の兆候をできるだけ早いうちに捉えるということしかないだろう．

文　献

[1] Diomidis Spinellis,『Code Quality，コードリーディングによる非機能特性の識別技法』，毎日コミュニケーションズ，東京，2007.

[2] 菅野文友,『ソフトウェアの品質管理』,日科技連，東京，1986.

[3] ステファン H. カン,『ソフトウェア品質工学の尺度とモデル』，共立出版，東京, 2004.

[4] マーチン・ファウラー,『リファクタリング，プログラミングの体質改善テクニック』，ピアソン・エデュケーション，東京, 2000.

[5] スコット W. アンブラー，ピラモド・サダラージ,『データベース・リファクタリング』，ピアソン・エデュケーション，東京, 2008.

[6] 蔭山泰之,「CRUD マトリックスの自動生成による品質管理」，ソフトウェアテストシンポジウム 2006 予稿集，100-106 ページ.

[7] リンダ・M・ライルド，M・キャロル・ブレナン,『演習で学ぶソフトウェアメトリクスの基礎』，日経 BP 出版センター，東京, 2009.