# APMツールを活用したシステム見える化への取り組み

野村総合研究所
開発技術部　副主任テクニカルエンジニア

**藤田 由起**（ふじた ゆき）

情報技術本部にて開発技術の調査、研究と実プロジェクトへの適用支援を行うテクニカルエンジニア。現在はレガシーシステムの近代化手法に関する研究開発を実施している。



**要旨**

**今日、システムリリースしてから10年、20年が経過した所謂レガシーシステムを抱える企業は少なくない。レガシーシステムは、長年のメンテナンス活動によりロジックが複雑化し、アプリケーションのメンテナンスコスト上昇やシステム知識の属人化といった問題をもたらしている。また、現状ではシステムの問題点を客観的に捉えられておらず、何が本当に問題なのか関係者同士で共有できていないことが多い。**

**このような状況を受け、野村総合研究所（NRI）では、現状を可視化し、問題発見や解決に役立てるためAPM（Application Portfolio Management）ツールを活用している。本稿では、その取り組みについて述べる。**

キーワード：APM、定量化分析、可視化、ソフトウェアメトリクス、メインフレーム、COBOL

Nowadays quite a few companies continue to use some legacy systems during a couple of decade after their system release. Legacy system has complicated logic due to maintenance activities throughout long period of time, and cost increment of application maintenance and system knowledge reliance on specific people has been brought about by it. Problems concerning systems have not been objectively perceived, persons involved have not been able to share the idea of what the real problems are. In the light of this situation, Nomura Research Institute, Ltd.(NRI) is utilizing APM (Application Portfolio Management) tool for visualizing the current situation and suiting the purposes of finding problems and reaching resolution. This article will describe about our approach.

Keywords : APM, Quantification analysis, Visualization, Software metrics, Mainframe, COBOL

## 1．はじめに

現在、コンピュータシステムが誕生して久しく、システムを導入していない企業はほとんどないと言えるだろう。むしろ1980年代～1990年代にリリースされ、リリース以来数々の機能追加、他システムとの統合など多くの改良、改変を繰り返してきたレガシーシステムを抱える企業が少なくない。また、そのようなシステムはハードウェアの保守費や保守期限といった問題や、アプリケーションのメンテナンスに関わる問題など、様々な問題を抱えている。野村総合研究所（NRI）ではAPM（Application Portfolio Management）ツールを、アプリケーションのメンテナンスに関わる問題の発見や解決に活用する取り組みを行っている。本稿では、その取り組みについて紹介する。

レガシーシステムが抱えているアプリケーションのメンテナンスに関わる問題点は、次のようなものがある。

### （1）プログラムロジックの複雑化

リリースされてから何年も経過しているシステムのプログラムソースコードは、長年のメンテナンスの繰り返しにより、その多くが複雑化している。

ロジックが複雑化する原因は様々であるが、メンテナンスの際に標準化を無視したコード修正を行ったり、デグレードのリスクを避けるために既存ロジックは触らず冗長なコードを追加したりなどが考えられる。そのようなメンテナンスが繰り返されると、ベテラン担当者以外は理解不能な複雑なプログラムになってしまい、システム知識の属人化を助長させる。また、複雑であるために仕様理解や開発、テスト実施に工数がかかってしまい、アプリケーションのメンテナンスコストの上昇という問題も発生する。

### （2）管理対象プログラムの増加

さらに、長年のメンテナンス期間を経て、管理対象となるプログラムが増加してきているという現状もある。機能が増えてコード量が増えるのは必然であるが、既に不要となったプログラムを削除せず放置しておいたり、似たような処理があった場合に、プログラムを共通化するのではなく、プログラムを丸ごとコピーして部分的に修正したりすることで必要以上にプログラムが増えてしまっている。この場合も修正時の影響調査に工数がかかってしまったり、1つの仕様変更が入ると多くのプログラムを修正しなくてはいけなかったりという問題が発生する。こういったことも、アプリケーションのメンテナンスコスト上昇につながっている。

上述の問題を解決するためには、システムを再構築する、問題のある部分を修正しドキュメントを整備するといった対応策が考えら

れる。しかし、対策を講じる前にシステムをきちんと分析し、具体的に「何が問題なのか」という点を客観的に把握し、それを関係者同士で共有することが重要である。

筆者らは、現状を可視化し、問題を客観的に捉えるために、APM ツールを活用している。単にツールで数値を算出するだけでなく、問題にどう対処すればよいかを示すところまで実施している。また、プロジェクト適用を通じ、様々なノウハウを蓄積してきた。

なお、APM 自体は特定の言語に限った手法ではないが、本稿で取り上げる事例は主に COBOL、PL/I といった古い言語で開発されたシステムを対象としている。

## 2．APM とは

APM とは、金融資産のポートフォリオ管理と同様の考え方で IT 資産を管理しようという手法である（図 1）。

投資家は自分の資産の現状がどうなっているのかをリアルタイムで把握し、その情報を分析して投資戦略を練るが、IT 資産も金融資産と同様に捉え、可視化して IT 投資の最適化を図ろうというものである。

APM を実現するために使用されているツールが APM ツールである。APM ツールは自らが所有する IT 資産の現状を可視化し、システムの問題発見を可能とする。さらに、その問題を解決するための意思決定材料となる情報も提供する。

APM ツールは「IT 投資の意思決定の判断材料となる情報を提供し、効率的な IT 投資を実現するもの」と言え、現在 1 節で述べたようなレガシーシステムの抱える問題を解決するための 1 つの道具として注目を浴びている。

具体的には、APM ツールは IT 資産（プログラムソースコード、JCL、画面定義体など）を取り込んで解析し、様々な情報をリポジト

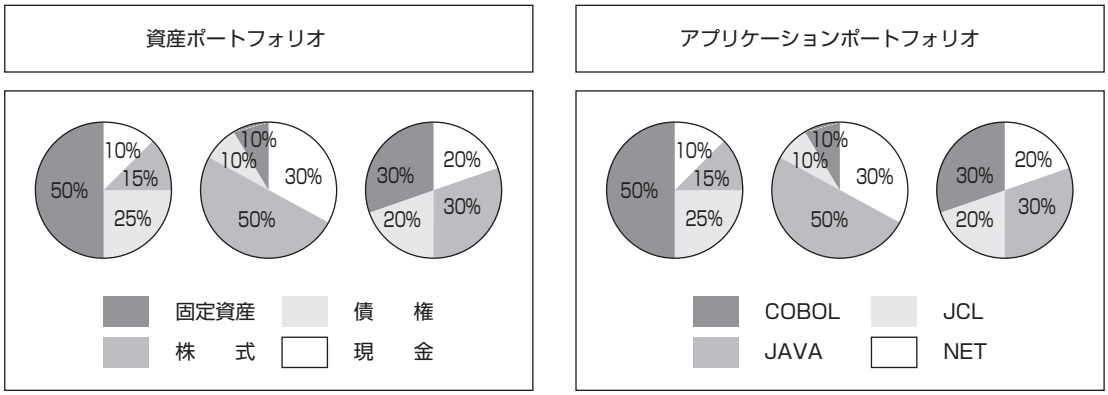



**図 1　資産ポートフォリオと APM**

リへ格納する。場合によってはプログラムなどのIT資産だけでなく、アプリケーションの利害関係者の意見やシステムのビジネス価値、アプリケーションの変更頻度といったソースコードからは分からない情報も合わせて取り込む。そして、それらの情報を元に様々なレポートをユーザの役割（経営者、プロジェクトマネージャ、開発者、品質監理者など）に合わせて提供する（図2）。

APMツールはツールのカバーする範囲が様々で、ツールによって重要視するアウトプットが異なるが、アウトプットの例として以下のようなものがある。

- コスト分析
  適切なところに適切なコストがかけられているかを可視化する。
- データCRUD分析
  どのプログラムがどのデータにアクセスしているかを分析する。
- 定量化分析
  様々なメトリクスを算出する。
- プログラム構造図
  プログラムの構造を図式化する。
- 波及分析
  ある特定の項目が変更された場合、その影響がどのプログラムのどの部分まで影響するかを分析する。

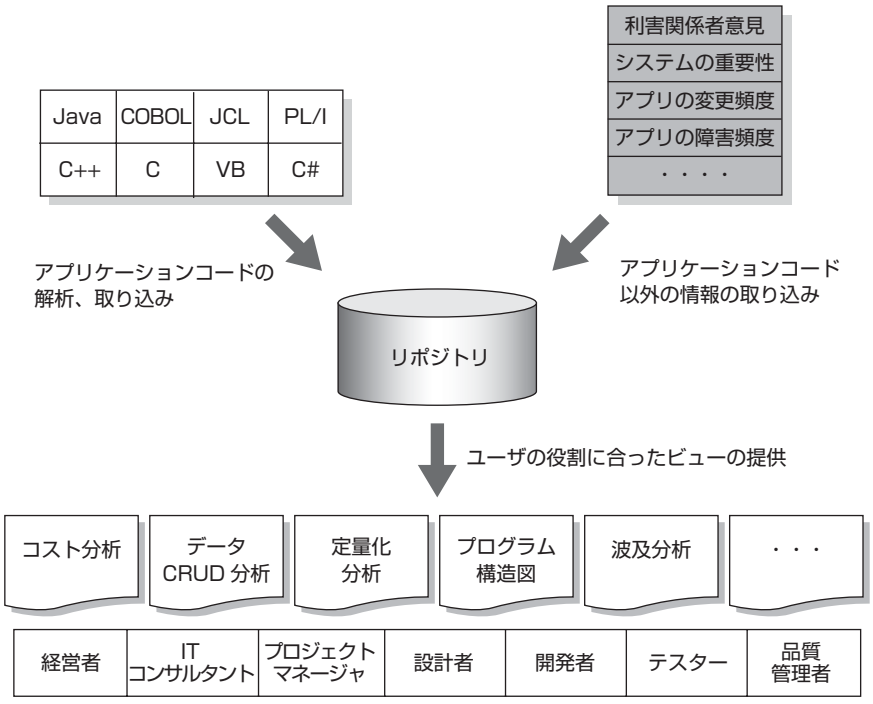



**図2　APMツール**

## 3．NRIのAPM分析手法

### （1）NRIのAPM分析手法の特徴

筆者らは、APMツールの様々な機能のうち、社内のニーズの高かった定量化分析機能に着目し、システムを可視化する取り組みを行っている。使用している数値とそれぞれの意義、目的を表1に示す。

NRIのAPM分析手法の特徴は、単にツールで算出した数字をレポートするだけでなく、問題の対処方法まで示すところにある。表1に4つの数値を示したが、実際はツールで算出した数値だけをレポートしても、開発現場はそれを元に何をすればよいのか分からない。筆者らは、数値をどのように解釈し、さらにどう踏み込んで分析すれば開発現場に有効かを検討した。分析手法を確立するにあたり、具体的に工夫した点について以下にまとめる。

- 定量化するだけでなく、定量化で全体を把握した後、問題の詳細は個別に分析

定量化はシステム全体の傾向を把握するのに非常に有効な手法ではあるが、それだけでは具体的に何が問題かまでは十分に捉えられない。筆者らは、定量化数値を算出することで問題のあるプログラムの当たりをつけ、その後具体的な問題や課題を特定するためにソースコードを個別に見て分析するといったアプローチをとっている。

- プロジェクト適用によるノウハウ蓄積

開発現場にとって有効な手法を確立するため、筆者らは様々なプロジェクトで分析を実施し、成果物を元に開発現場の人々と議論を重ねて手法を改善してきた。その結果、多くのノウハウも蓄積した。

- 基本は市販のAPMツールを使用するが、足りない部分は自作

世の中には多くのAPMツールが存在し、それらはインプットとして与えたプログラム

**表1　使用する定量化数値**

| 定量化数値 | 数値の意味 | 数値の意義・利用方法 |
|---|---|---|
| 規模 | ソースコードのステップ数、ファイル数 | システムの規模を捉えるために利用する |
| 複雑度 | モジュールがどれくらい複雑かをあらわす数値<br>この数値が大きいモジュールは、テストが困難で、正しく修正しにくくなるとされている | システム全体としてプログラムの構造が適切であるかどうかを把握するために利用する<br>また、複雑度の高いモジュールに絞ってソースコードを読むことにより、効率的にシステムの問題点を抽出できる（個別モジュール分析） |
| 重複度 | 「似た」プログラムがシステム中にどの程度の割合で存在するかを表す | 規模の見積りを補正するために利用する<br>また、ソースコードのスリム化にも利用できる |
| デッドコード率 | 存在するが呼ばれていない、参照されていないコードが全体のどの程度の割合で存在するかを表す | 規模の見積りを補正するために利用する<br>また、ソースコードのスリム化にも利用できる |

を解析して多くの情報をリポジトリへ格納する。筆者らは市販のツールを機能性や数字の信頼性といった観点で評価し、使えると判断したものを使用している。ただし、市販ツールで算出できない数値に関しては、必要に応じて筆者らでツールの開発を行った。また、数値を集計し、レポート作成を行う部分についても有効な方法について検討を行い、独自にツールを開発した。

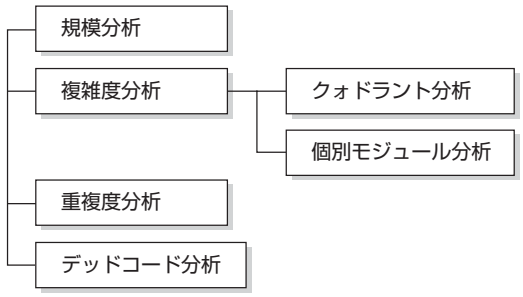



**図3　NRIのAPM分析手法の全体像**

### (2) NRIのAPM分析手法の詳細

NRIのAPM分析手法の全体像を図3に示す。複雑度分析はクォドラント分析と個別モジュール分析という2段階の分析を行う。

以下、各分析方法について順番に説明する。

なお、以降の説明においてツールとしてはMcCabe & Associates社のMcCabeIQというツール[3]を使用している。本ツールは数字を算出するだけでなく、モジュール構造を表示する機能も備えている。

| システム全体 | | |
|---|---|---|
| 総ステップ数 | 4,530,671 | ※コメント・空白以外のステップ数 |
| 全行数 | 6,133,365 | |
| コメント・空白行数 | 1,602,694 | |
| コメント率 | 26.1% | |
| ファイル数 | 9.478 | ※解析対象のファイル総数 |
| 概算FP値 | 26.651 | ※言語ごとの調整値を用いてステップ数から換算したFP値 |

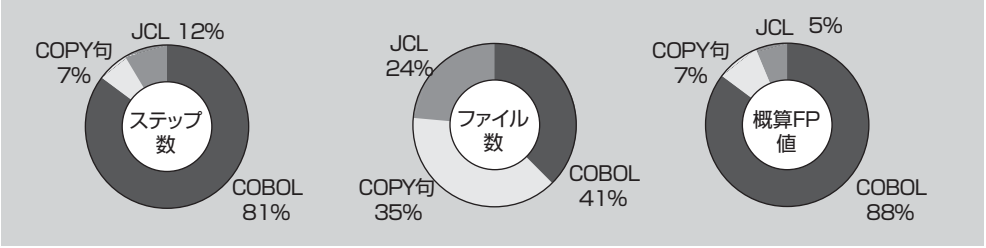



**図4　規模レポートの例（システム全体）**

①規模分析

規模とは、ソースコードのファイル数やステップ数である。言語別やサブシステム別、オンライン/バッチ別など様々な切り口で集計する（図4、図5）。

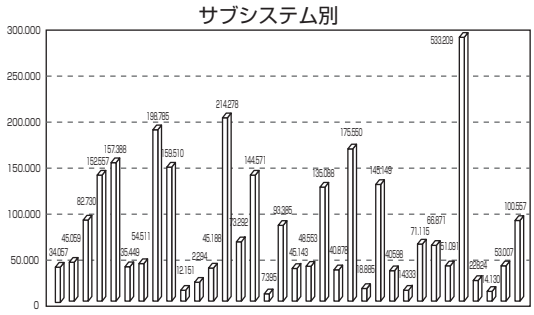



**図5　規模レポートの例**
**（サブシステム別ステップ数）**

サブシステム間のばらつきなどからシステムの特性を把握することが可能である。また、ステップ数を用いて以下の公式より概算FP値を計算することもできる。

概算FP値 = ステップ数 / 調整係数

調整係数は言語ごとに異なる。値はQSM社やDavid Consultingの数値[1]を参考に、NRIとして適切な数値を検討中である。

②複雑度分析

複雑度は「モジュール」単位に算出し、その「モジュール」がどのくらい複雑かを表わす数値である。なお、「モジュール」とは、COBOLで言うところの「セクション」で、C/C++では「関数」、Javaでは「メソッド」にあたる。

筆者らは、McCabeの循環複雑度と本質複雑度の2つの数値[2]を使用し、その組み合わせで複雑度を評価する。

● 循環複雑度

循環複雑度とは論理的な複雑度を示し、分岐数が大きければ大きな値になる。図6に示したように、循環複雑度はモジュールのパスをグラフ化し、そのエッジとノードの数を用いて求められる。また、一般に50を超えるとテストで全パスを通すのが難しく、保守が困難と言われている（表2）。

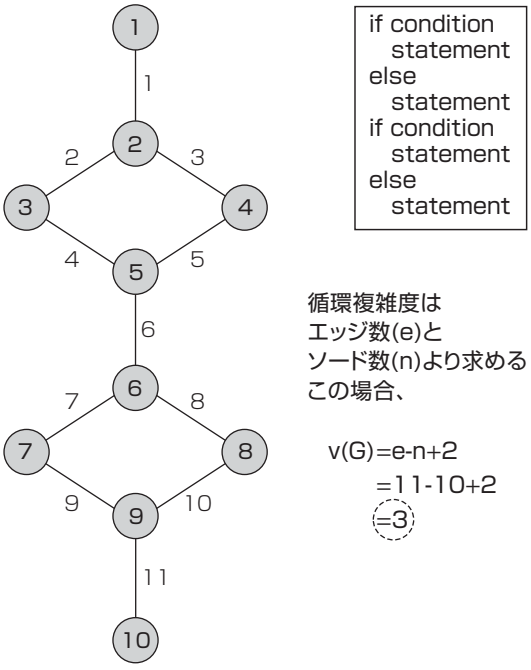



**図6　循環複雑度の求め方**

**表2　循環複雑度の目安**

| Cyclomatic Complexity | Risk Evaluation |
|---|---|
| 1-10 | a simple program, without much risk |
| 11-20 | more complex, moderate risk |
| 21-50 | complex, high risk program |
| greater than 50 | untestable program (very high risk) |

**（出典：カーネギーメロン大学）**

● 本質複雑度

本質複雑度とは、循環複雑度のうち非構造化処理部分のパス数である。図7で示すように、循環複雑度を求めた際のパスの構造化部分を収縮して求められる。残った非構造化部分はGOTO文を使用している箇所と言えるため、本質複雑度に着目することでGOTO文を多用しているモジュールを検出できる。

筆者らは、「クォドラント分析」により全体の傾向を把握し、その後「個別モジュール分析」により複雑なモジュールの原因を調査するといった方法をとっている。以下に、これらの分析方法を解説する。

● クォドラント分析

クォドラント分析では、循環複雑度を縦軸、本質複雑度を横軸にとり、個々のモジュール

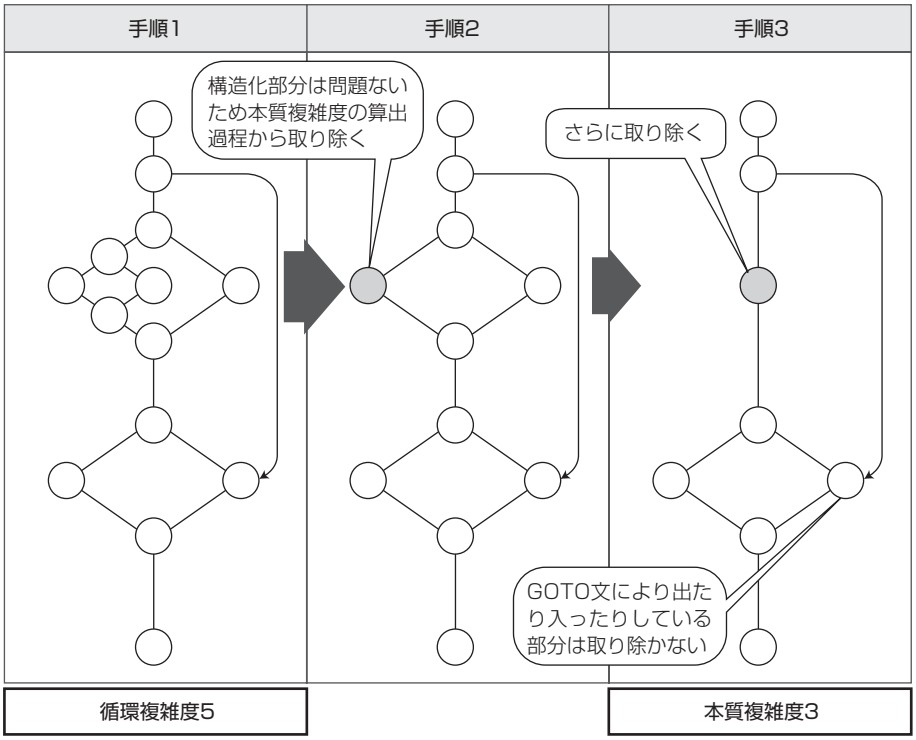
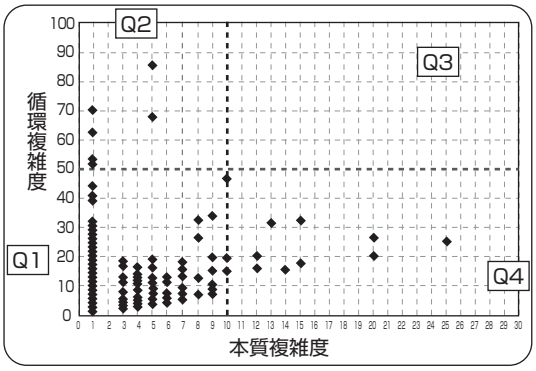



**図7　本質複雑度の求め方**

をグラフにプロットする。そして、循環複雑度、本質複雑度それぞれの閾値を「50」「10」に設定し、4つに分けた区画をそれぞれ、「Q1」「Q2」「Q3」「Q4」と定義する（図8）。

モジュールがどの区画にどれだけ含まれているかを分析し、システムの複雑度の傾向を把握する。複雑度分析より生成されるレポートの例を図9に示す。




| | モジュール数 | | コード行数 | | 循環複雑度 | | 本質複雑度 | |
|---|---|---|---|---|---|---|---|---|
| | | 構成比 | | 構成比 | | 構成比 | | 構成比 |
| Q1 | 4362 | 99.6% | 83787行 | 96.2% | 17803 | 95.4% | 4756 | 96.6% |
| Q2 | 9 | 0.2% | 2503行 | 2.9% | 642 | 3.4% | 25 | 0.5% |
| Q3 | 0 | 0.0% | 0行 | 0.0% | 0 | 0.0% | 0 | 0.0% |
| Q4 | 10 | 0.2% | 831行 | 1.0% | 217 | 1.2% | 144 | 2.9% |
| | 4381 | 100.0% | 87121行 | 100.0% | 18662 | 100.0% | 4925 | 100.0% |

**図9　複雑度分析レポートの例**

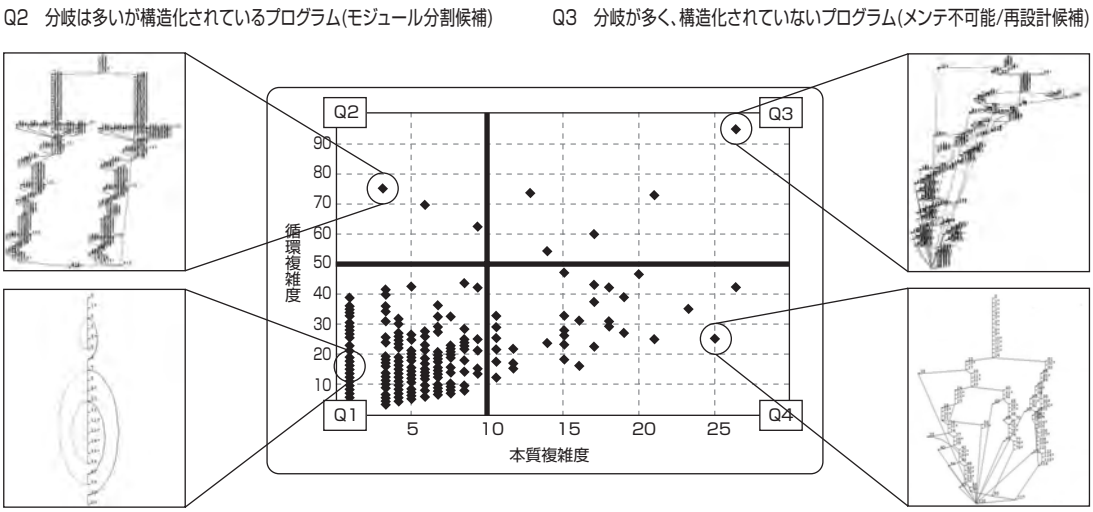
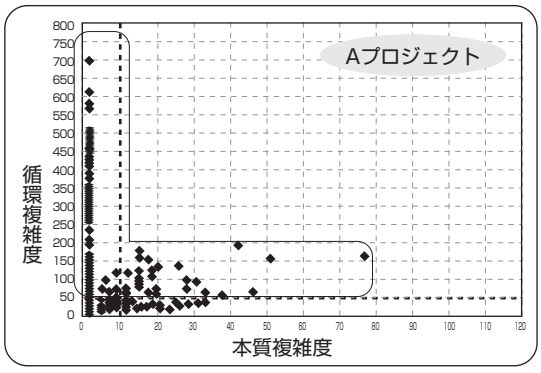


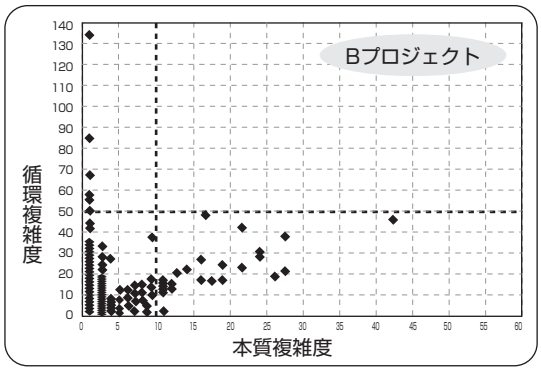

**図8　複雑度のクォドラント分析**

クォドラント分析のグラフの下の表では、各区画に含まれる「モジュール数」とその割合、「コード行数」とその割合、「循環複雑度」とその割合、「本質複雑度」とその割合を示している。筆者らは傾向を見るために以下のような手法を使用している。

1. Q2とQ3に所属するモジュールの「コード行数の割合」の合計、つまり、複雑なモジュール（循環複雑度が閾値の50以上のモジュール）を構成するステップは全ステップの何パーセントかを示す値に着目し、そのシステムがどの程度複雑か判断する。

Q2とQ3のコード行数割合に着目する理由は、一般的に複雑なモジュールは、数は少なくても、巨大でステップ数が多く、ステップ数換算にすると全体のかなりの割合を占める傾向にあるためである。傾向比較している例を図10に示す。Bプロジェクトが0.5%に対しAプロジェクトが12.6%でAプロジェクトの方が複雑なことが分かる。Aプロジェクトの方が複雑なことは、グラフ上のモジュールの散布状況でも確認できる。




| | モジュール数 | 構成比 | コード行数 | 構成比 | 循環複雑度 | 構成比 | 本質複雑度 | 構成比 |
|---|---|---|---|---|---|---|---|---|
| Q1 | 100,194 | 99.2% | 1618335行 | 84.7% | 328,991 | 83.7% | 109,685 | 93.4% |
| Q2 | 341 | 0.3% | 215097行 | 11.3% | 49,637 | 12.6% | 375 | 0.3% |
| Q3 | 48 | 0.0% | 24058行 | 1.3% | 4,945 | 1.3% | 1,091 | 0.9% |
| Q4 | 454 | 0.4% | 53653行 | 2.8% | 9,261 | 2.4% | 6,228 | 5.3% |
| | 101,037 | 100.0% | 1911143行 | 100.0% | 392,834 | 100.0% | 117,379 | 100.0% |

| | モジュール数 | 構成比 | コード行数 | 構成比 | 循環複雑度 | 構成比 | 本質複雑度 | 構成比 |
|---|---|---|---|---|---|---|---|---|
| Q1 | 34050 | 99.8% | 764008行 | 98.5% | 70159 | 98.2% | 41649 | 98.4% |
| Q2 | 7 | 0.0% | 3028行 | 0.4% | 396 | 0.6% | 9 | 0.0% |
| Q3 | 1 | 0.0% | 555行 | 0.1% | 50 | 0.1% | 17 | 0.0% |
| Q4 | 44 | 0.1% | 8063行 | 1.0% | 857 | 1.2% | 670 | 1.6% |
| | 34102 | 100.0% | 775654行 | 100.0% | 71462 | 100.0% | 42345 | 100.0% |

**図10　複雑度分析傾向比較の例**

2. 続いて、Q3とQ4に所属するモジュールの割合を確認し、構造化がされているかを確認する。

   ただし、GOTO文が使用されている場合でも、エラー処理の場合のみGOTO文の使用を認めるよう標準化されている場合などは、構造に問題があるとは言えない。つまり、クォドラント分析のみでは「GOTO文の使い方に問題あり」とは言いきれず、この後で行う個別モジュール分析を行って明らかにする必要がある。Q3+Q4の割合が高い場合は、個別モジュール分析でその原因を調査する。

● 個別モジュール分析

全体傾向を把握した後、複雑なモジュールは具体的に何が問題なのか、システムにどういった課題が存在しているかを分析する。これを「個別モジュール分析」という。この分析は自動化できず、実際に複雑なモジュールのソースコードを読むことになる。

その際、どのモジュールに着目してソースコードを読めばよいかを判断するために、先述のクォドラント分析の結果を使用する。具体的には、Q2、Q3、Q4に所属するモジュールの中で循環複雑度や本質複雑度の値が特徴的なモジュールを選択する。さらに、ツールを活用してモジュール構造を表示することで大まかな構造把握が可能なため（図11）、ソースコードを読むのに役立つ。

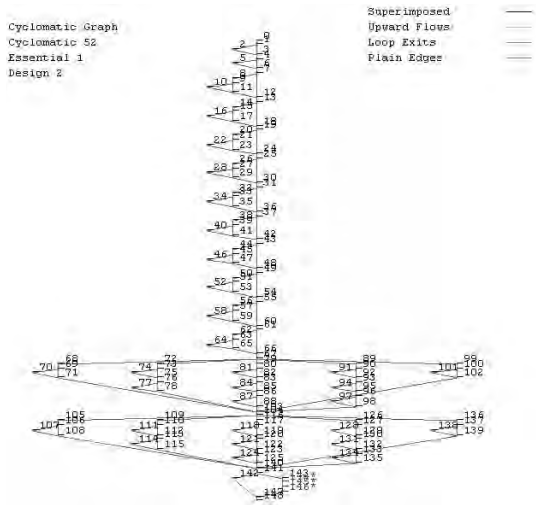



**図11　モジュール構造の例**

単純に「循環複雑度の高いモジュール＝問題のあるモジュール」というわけではない。循環複雑度は、単純なロジックの繰り返し（例えば、似たような単純処理を項目ごとに実施していて、かつ項目数が大量にある場合）によっても高くなるためである。

この区別をするためにもツールのモジュール構造表示機能を使用した分析が有効である。単純なロジック繰り返しの場合は、構造を表示させると縦長かつ綺麗な構造になる（図12）。一方、ロジックが複雑な場合は規則性のない複雑な構造になる。

加えて、個別モジュール分析では本質複雑度が高い理由についても確認する。前述の通り、エラー処理などで同一箇所に飛ぶためにGOTO文が使用されている場合は問題ないが、ソースの様々な場所に飛んでいて、構造

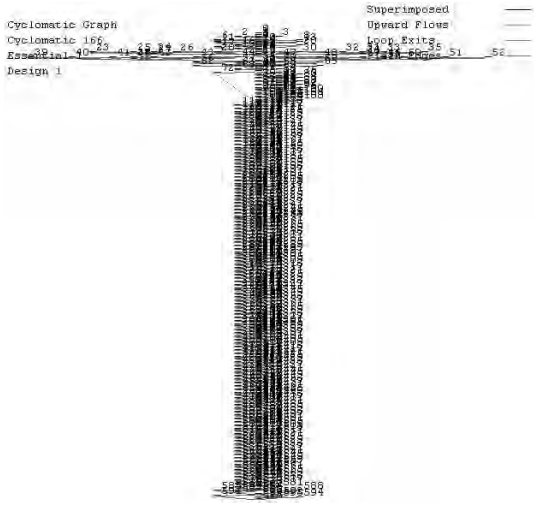



**図12　モジュール構造（単純繰り返し）**

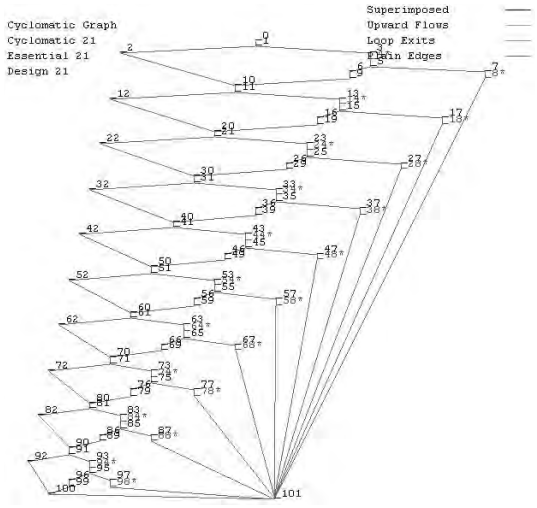



**図13　モジュール構造**
**（同一箇所に飛ぶGOTO文）**

化ができていないスパゲッティ状態の場合は問題として抽出できる。図13のモジュールは、本質複雑度が21であるが、GOTO文はモジュールの最後に飛ぶ場合にしか使用されていないのが分かる。

③重複度分析

重複度とは、プログラム単位でステートメントの並びがどれくらい一致しているかを表わす。一字一句重複しているかを判定するわけではなく、あくまでステートメントの並びを見ているため、使用している変数名やCOPY句、実行するプログラム名が異なっていても一致していると判断する。

重複度100％のプログラムに注目することで、別プログラムをコピーして新規作成（プログラムIDやCOPY句、変数名のみ変更）しているプログラムの抽出に役立つ。図14は重複度100%の例である。左のプログラムは、右のプログラムをコピーし、計算式を1箇所変更しただけのものであることがわかる。

④デッドコード分析

デッドコードとは、どこからも呼ばれることのないコードを意味し、2つの分析観点を利用する。

1つ目は「デッドプログラム」であり、JCLを拠点として絶対に呼ばれることのないプログラムを分析する。ただし、特別な場合にのみ使用されるプログラムは必ずしもJCLに登録されているとは限らないため、注意が必要である。また、オンラインプログラムの場合はJCLから呼ばれないため、別途他の方法で分析する必要がある。デッドプログラムと判

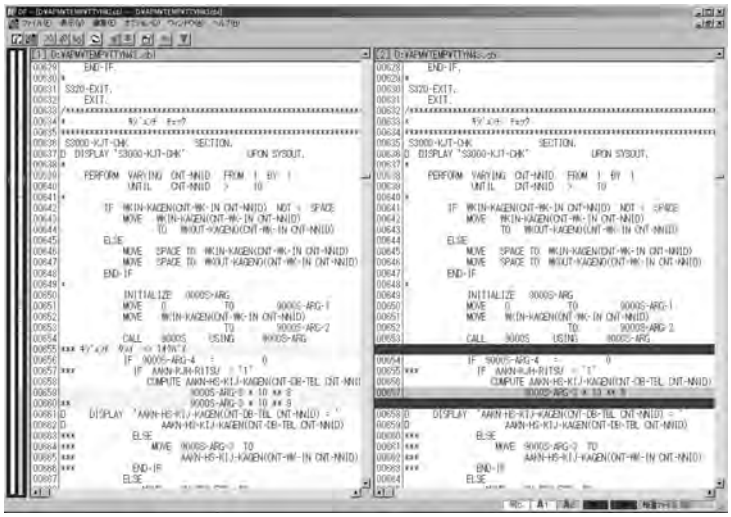


**図14　重複度100%の例**

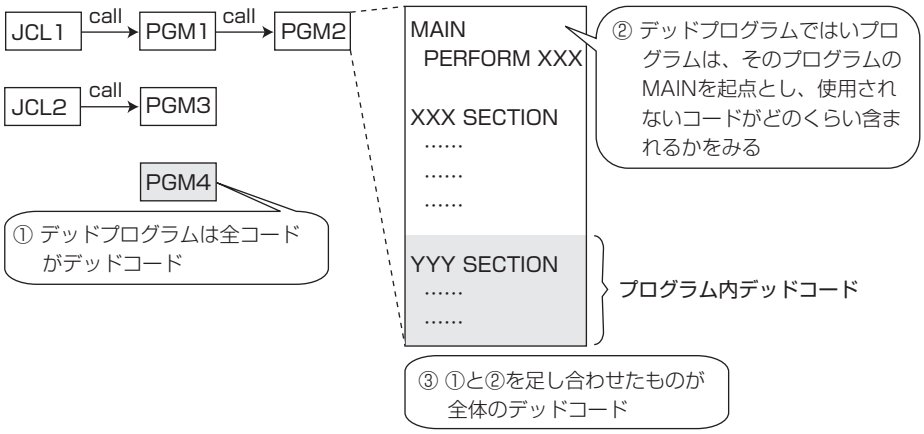



**図15　デッドコード率とは**

定されたプログラムは全コードがデッドコードと言える。

2つ目は「プログラム内デッドコード」であり、各プログラムのエントリポイントであるMAINを起点として呼ばれることのないモジュールや行、使用されないデータ定義を分析する。

両方の結果を考慮して最終的なデッドコードを算出する（図15）。

### (3) 分析結果からわかること

筆者らが行っているAPM分析手法で使用している数値の意味とそれらを用いた分析方法については3(2)で解説した。次に、分析結果より発見可能な問題について具体例を挙げて解説する。なお、規模やデッドコードは結果からわかることが自明なため、複雑度と重複度のみを取り上げる。

①複雑度

- モジュール分割すべきモジュール候補

図16のモジュールは循環複雑度が250で非常に高い。このモジュールはループの中の処理(図中の①、②部分)が4階層のIF文になっている。また、ステップ数が巨大で、どこがIF文の終わりなのかが分かりにくく可読性が低い。このようなモジュールは、階層の深い処理はサブルーチン化するなどしてすっきりさせるべきである。

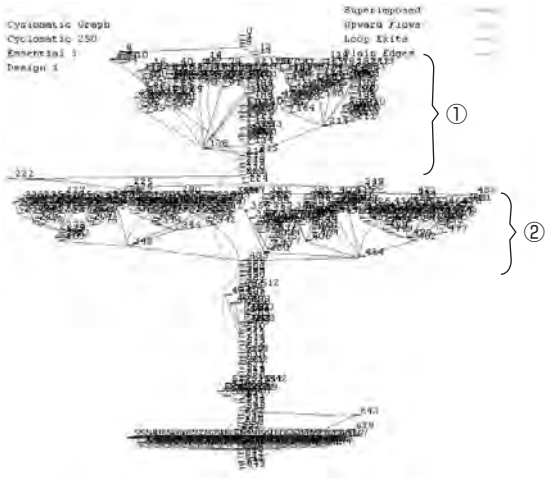



**図16　モジュール構造(モジュール分割すべき)**

- ロジックを整理すべきモジュール候補

図17のモジュールは、度重なる仕様追加や仕様変更が原因で、ロジックがスパゲッティ化した例である。こういったモジュールは仕様情報も喪失して誰も触れない状態になり、実際メンテナンスの際に苦労している。ロジックを整理して修正すべきモジュールである。

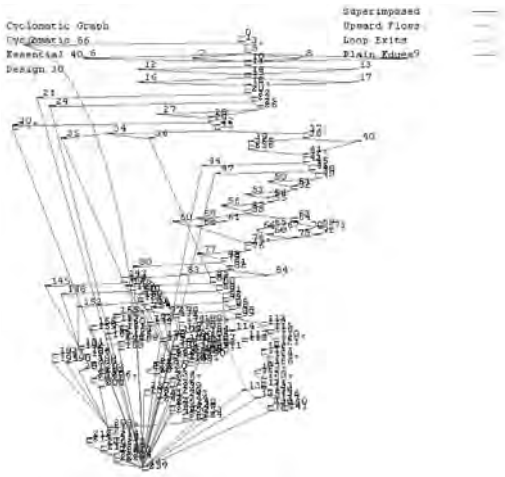


**図17　モジュール構造(ロジックを整理すべき)**

- 標準化の守られていないモジュール

複雑度分析によって、標準化の守られていないモジュールが発見できることがある。実際のプロジェクト適用で得られた事例を2つあげる。1点目は図18に示したモジュールである。このモジュールは、処理の先頭である区分を判定し、区分ごとに処理を分けている。本来分岐先の処理は外部モジュールに実装し、本モジュールはそれをコールするだけと

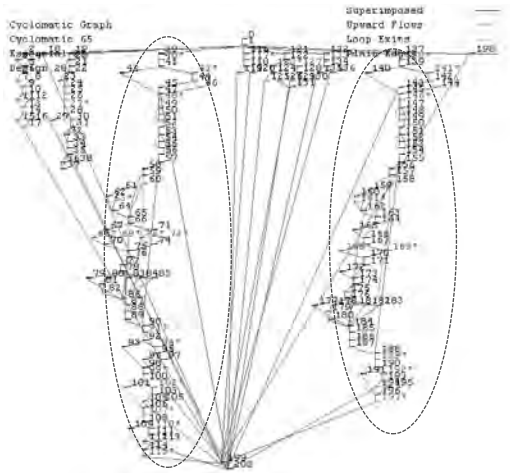



**図18　モジュール構造（標準化を無視①）**

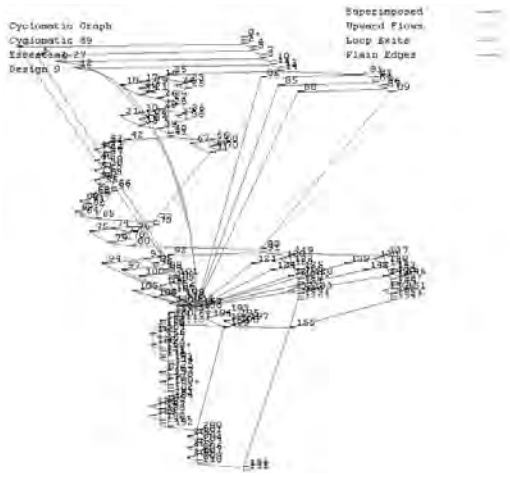



**図19　モジュール構造（標準化を無視②）**

いう構造に標準化されていた。しかし、モジュール修正の際に標準化を無視し、外部モジュールに実装すべき処理を本モジュールに実装してしまったのが分かる（破線の囲み部分）本モジュールはそれゆえに循環複雑度が高くなっていた。

2点目は図19に示したモジュールである。本モジュールは3階層のコントロールブレーク処理を、GOTO文を多用して独自に実装しており、循環複雑度、本質複雑度がそれぞれ94、28と共に大きな値だった。本来は独自に実装するのではなく、コントロールブレーク処理用のテンプレートを使用するように標準化されていた。

● 構造化されていないモジュール

本質複雑度は、GOTO文を多用していて構造化されていないモジュールを発見するのに役立つ。しかし、前述の通り必ずしも「GOTO文を使用している＝構造の悪いモジュール」というわけではなく、個別分析してみないと断定できない。例えば、項目チェック処理において、エラーを検出したらその場で処理を抜けてモジュールの最後にGOTO文で飛ばすようにエラー処理方式を標準化することがよくある。この場合本質複雑度は高くなるが、構造に問題があるわけではない。GOTO文の使用方法を確認する場合もモジュール構造の参照が有効である。モジュールの最後に飛ぶGOTO文しか使用していない場

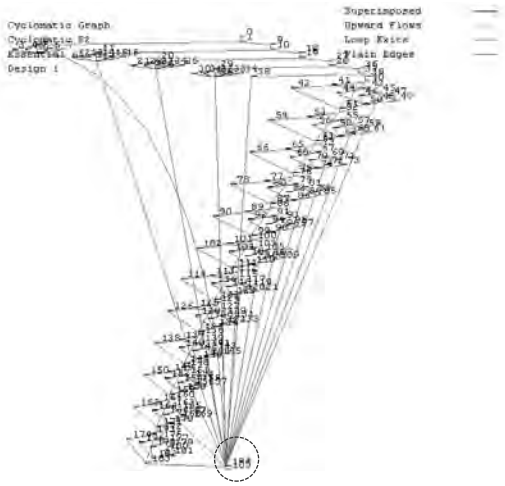



**図20　モジュール構造（標準化されたGOTO文）**

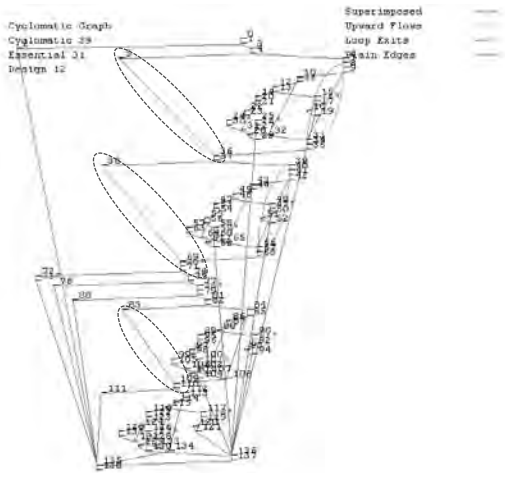



**図21　モジュール構造**
**（標準化されていないGOTO文）**

合は図20のようになるが、モジュールのあちこちに飛んでいる場合は図21のようになる。

以上より、循環複雑度や本質複雑度の高いモジュールに着目して個別に分析することで、様々な問題を発見できることが分かった。

②重複度

● ライブラリ管理の問題

あるプロジェクトで、重複度分析によりライブラリ管理に問題があることが発見できた。そのプロジェクトでは、重複度100%のプログラムがたくさん存在した。実際に中身を分析すると、バックアップや古いバージョンのプログラムが残っていたということが分かった。管理対象とすべきプログラムが把握できていないということは、正確な規模も管理できていないということである。APM分析は、このようなプロジェクト管理上の問題も発見できることがある。

● 共通化可能プログラム候補

重複度が100％のプログラムは、別プログラムをコピーして新規作成された可能性が高い。コピーして新規作成されたプログラムは、本来共通化可能なプログラム候補となる。ただし、意図的にコピーしている場合も少なくないため、本当に共通化すべきかどうかは個別に確認する必要がある。

## 4．NRIのAPM分析の活用場面

APM分析の活用場面の例を、システム開発のフェーズ別に取り上げる（表3）。

これらは、筆者らがAPM分析を様々なプロジェクトに適用し、有効性を確認できたもの及び開発担当者にヒアリングして得られたものである。個々の活用場面について実例を取り上げながら解説する。

### （1）システム化計画時の戦略決定

APMが既存のIT資産を分析し、経営層にも分かりやすい形で可視化するものであることは既に2節で述べた。筆者らが担当したあるプロジェクトは、現行のソースは捨てて一から再構築をするか、それとも再利用できるコードは再利用して単純にメインフレームからオープン系システムへ移行すべきかどうかの判断材料を求められていた。APM分析の結果、多くの問題点が抽出された。

- 無駄なコード、不要なコードがたくさん存在する　→デッドコード率約30％
- 複雑度が全体的に高い
  →Q2 + Q3が約9％
- 重複度が高く、ライブラリ管理がされていない
- 個別分析によって、標準化がされていない、マスタ化されるべきデータがプログラムにハードコーディングされていてデータ設計に問題があり拡張性が低いことがわかった

結果、非常に複雑なため、再利用するより一から再構築する方がよいという判断に至った。

問題点はAPM分析により可視化されたため、経営層にも納得してもらうことができた。

**表3　APM分析の活用場面**

| # | フェーズ | 活用場面 | 説明 | |
|---|---|---|---|---|
| 1 | システム化計画 | 戦略決定 | レガシーシステムに対して何らかの対策をする場合、アプリケーションを再利用すべきか、いちから再構築すべきかといった戦略を決定するのに役立つ情報を提供する | (1) |
| 2 | 概要設計 | 見積（計画） | 戦略決定で決定した行動を計画する際の見積に役立つ情報を提供する | (2) |
| 3 | 開発・テスト | 受入 | 開発したプログラムの受入評価に役立つ情報を提供する | (3) |
| 4 | 保守 | プログラム構造の見直し | 問題のあるプログラム構造を見直す場合、見直し対象とするモジュール候補を抽出する。また、見直し後に本当に綺麗になったかを確認する | (4) |
| | | 見積（メンテナンス） | メンテナンス時の調査、開発、テストにかかる工数見積に役立つ情報を提供する | (5) |
| | | ナレッジトランスファ | 知識移転を行う際に役立つ情報を提供する | (6) |
| | | 定期健診 | 定期的に分析を行い、病気になっていないか測定する | (7) |

### (2) 見積(計画)

再構築を計画する際の見積にもAPM分析の結果を使用できる。例えば、レガシーシステムは必要以上にステップ数が増えてしまっている場合があるが、デッドコード率や重複度の結果より本来どれくらいまでに削減できるかが分かる。また、複雑度の高いモジュールはモジュール分析に時間がかかる傾向にあることがプロジェクト適用より分かった。そのため、複雑度分析の結果は現行システム分析の工数見積に有効と期待できる。

### (3) 受入

プログラムの受入前に複雑度分析を行うことで、構造の悪いプログラムのリリースを防げる。例えば、ある一定の複雑度を超えているプログラムに関してはその理由を確認し、実際に問題がある場合はそれを修正する方法である。

より早い段階で問題のあるプログラムが取り除かれればその後のメンテナンス段階で問題が発生しにくくなることが期待できる。

### (4) プログラム構造の見直し

APMは、メンテナンスの生産性向上のために汚くなってしまったプログラムの構造見直しや使われなくなった不要コード削除を行う際にも役立つ。

あるプロジェクトでは、修正モジュール候補の抽出に複雑度分析の結果を利用し、また構造見直し(構造見直しは人間が手作業で実施)後に本当に綺麗になったかを確認するのにも複雑度分析を利用した。さらに、不要コードを削除するにあたり、デッドコード分析の結果も活用した。

実際に修正対象として抽出されたモジュールと、構造見直し後のモジュールの分析結果を示す。図22が修正対象として抽出されたモジュールの1つである。

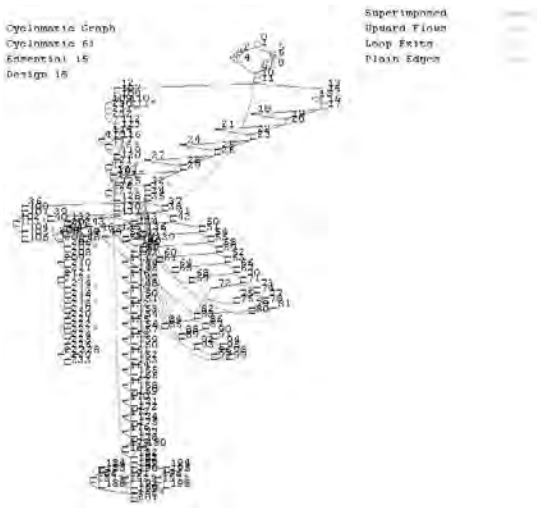



**図22　修正対象モジュール**

このモジュールは、循環複雑度が61、本質複雑度が15であり、個別分析の結果、GOTO文が多用されているために構造が把握し難く、保守に困っており、修正することになった。モジュール修正後の複雑度数値を図23に、モジュール構造を図24に示す。この例では、モジュール見直しの結果、4つに分割され、個々の複雑度も低下したことが確認できた。

修正前

| # | モジュール名 | 循環複雑度 | 本質複雑度 | ステップ数 |
|---|---|---|---|---|
| 1 | S203 'OUT-BREAK-1-1 | 61 | 15 | 304 |

修正後

| # | モジュール名 | 循環複雑度 | 本質複雑度 | ステップ数 |
|---|---|---|---|---|
| 1 | S203 'OUT-BREAK-1-1-2 | 41 | 1 | 144 |
| 2 | S203 'OUT-BREAK-1-1-3 | 21 | 1 | 88 |
| 3 | S203 'OUT-BREAK-1-1 | 19 | 1 | 151 |
| 4 | S203 'OUT-BREAK-1-1-1 | 10 | 1 | 39 |

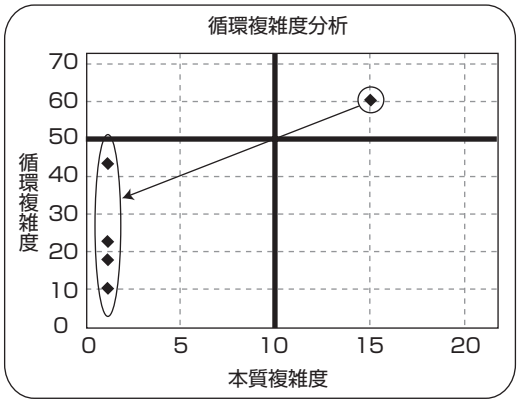



**図 23　修正前後の複雑度**

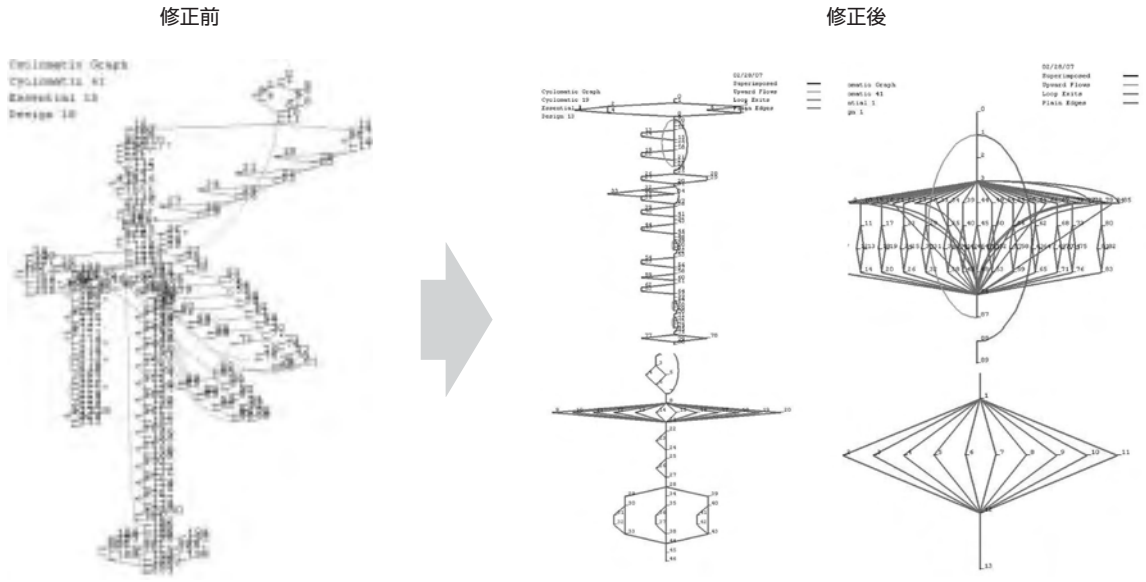



**図 24　修正前後のモジュール構造**

### (5) 見積(メンテナンス)

メンテナンステーマが発生した際の工数見積に複雑度が使用できる。メンテナンス活動には、影響範囲調査、ソース修正、テストというプロセスがあるが、どれも複雑なものほど工数がかかる。開発担当者にヒアリングした結果、多くの場合、開発者の自己申告で修正対象のプログラム難易度を決定し、それを元に工数見積を行っていた。複雑度は、その難易度が妥当かどうか定量的に判断するのに使用できる。より正確に見積ができれば、計画の遅延防止や、テスト不十分によるバグ発生の防止が期待できる。

### (6) ナレッジトランスファ

1節でも述べたようにシステム知識が属人化していることが問題になっており、新規参入者にナレッジを共有させることが大きな課題となっている。APMを用いた定量化分析によって現状を可視化させることは、ナレッジの可視化とも言える。例えば、規模レポートは、そのシステム、また、各サブシステムにはどれくらいのファイル数があって何ステップくらいなのかを一覧で把握できる。

また、複雑度レポートでどのサブシステム、どのプログラムが複雑かを把握でき、さらにツールのモジュール構造表示機能を使用することでどのように複雑なのかのイメージを持つことができる。当然中身を理解するには個別に設計書やソースを読み解いていく必要があるが、ナレッジトランスファするにあたり、まず全体像を掴んでもらうためにAPM分析は有効と言える。

### (7) 定期健診

APMは、本来継続的に実施していくべき活動である。しかし、リアルタイムにというのも現実問題厳しい面もある。

そういった場合は、例えば健康診断のように年に一回など定期的に分析を実施し、過去の結果との推移を見て値が極端に悪くなっていないかをチェックするのに使用できる。再構築などのアクションを起こす際に初めてAPM分析をするのではなく、日頃から定期的に分析を実施し、問題は早期発見、早期対応するのが望ましい。

以上のように、APM分析は多くの場面で活用でき、様々な問題解決に有効な情報を与えてくれる。

## 5. まとめ

本稿では、まずレガシーシステムが抱えるアプリケーションのメンテナンスに関わる問題点を明確にし、APMツールの定量化機能を用いて問題の可視化や問題解決に役立てる取り組みを紹介した。

これまでの活動においても、様々なプロジェクト適用を通じて定量化数値の活用ノウハウを蓄積してきたが、今後も多くのプロジェ

クトに適用し、さらなるノウハウ獲得、活用案の有効性検証、統計データとしての定量化数値の収集を行っていきたい。

また、本手法を用いて現状の問題を把握した後、再構築なり問題箇所の修正なりの活動をすることになった場合に役立つツールや手法の開拓も合わせて行っていきたい。

●参考文献●

［1］ QSM Function Point Programming Languages Table
http://www.qsm.com/FPGearing.html

［2］ McCabe, T.J., "A Complexity Measure," IEEE Transactions on Software Engineering, SE-2, no.4, p.308-320

［3］ McCabe & Associates, Inc
http://www.mccabe.com