オンライン ISSN 1347-4448
印刷版 ISSN 1348-5504

赤門マネジメント・レビュー 3 巻 12 号 (2004 年 12 月)

〔研 究 会 報 告〕コンピュータ産業研究会　2004 年 9 月 10 日 [1]

ソフトウェアのアーキテクチャ分析とその測定について [2]

立本　博文
特定非営利活動法人グローバルビジネスリサーチセンター
E-mail: tatsu@mmrc.e.u-tokyo.ac.jp

要約：オープンソース・ソフトウェアとは、ソフトウェアの設計情報（ソースコード）を公開して流通しているソフトウェアのことである。製品の設計情報の分析を行うことは、通常非常に難しい。しかしながら、オープンソース・ソフトウェアを題材に用いることにより、そのハードルを越えることができる。今回は、実際にアーキテクチャ指数というインデックスを定義してみることで、ソフトウェアがバージョンを経る毎にどのように変化したかを観察した。

キーワード：ソフトウェア、アーキテクチャ、モジュール化

1. はじめに

既存研究では、開発プロセスや生産プロセスといった「プロセス」とパフォーマンスの間の因果関係が測定されてきた。本研究の目的は、プロセスとともにパフォーマンスの決定要因となるものとして製品アーキテクチャの概念を取り入れ、このアーキテクチャの部分を数値化して測定することである。

本報告は、製品アーキテクチャの製品にソフトウェアを対象にしている。ソフトウェア製品の設計図とも言えるソースコードファイルが公開されるということはまれであるので、特殊な事例としてオープンソース・プロジェクトを用い、そのソースコードを分析することでアーキテクチャの数値化を試みた。

[1] 本稿は 2004 年 9 月 10 日開催のコンピュータ産業研究会での報告を竹迫まり恵（東京大学大学院）が記録し、本稿掲載のために報告者の加筆訂正を経て、GBRC 編集部が整理したものである。文責は GBRC に、著作権は報告者にある。内容の引用または複製には著作権者の許可を必要とする。

[2] この研究は「市場とボランタリの協働としてのリナックス・モデル」（事業番号 S-202-22）として特定非営利活動法人グローバルビジネスリサーチセンター（GBRC）が笹川平和財団から研究助成を受けた調査研究の一部である。ここに記して謝意を表したい。

2. 問題意識

本報告の問題意識は、ソースコードを分析すれば背後にある開発スタイルなどが見えてくるのではないかということである。従来ソースコード分析は、「社内では行われている」というように設計技法としては存在すると言われてきたものの、実証研究の例は非常に少ない。その理由としては、ソースコードはソフトウェア製品の設計図そのものであり、ソースコードがあれば、まったく同じソフトウェアが作ることが可能であり、企業としては外部に公開することが困難であることが考えられる。

このような制限を取り払うため、今回はオープンソース・ソフトウェアのプロジェクトを研究対象として取り上げた。その利点としてはソースがオープンであること、また今回扱った Apache に関しては、開発記録が非常にしっかりしており、その背後にある開発体制なども見えやすいことなどがある。

オープンソース・プロジェクトでは、ソースコードファイル、構成設定ファイル、マニュアルや開発ログ（更新履歴）などのファイルをインターネット上で配布している。

3. Apache プロジェクトの概要

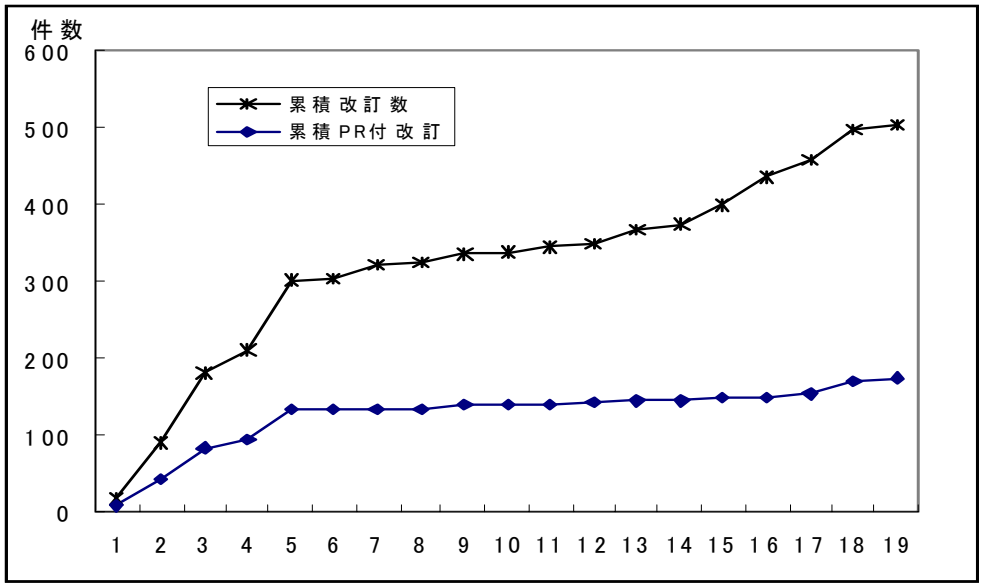
今回具体的に取り上げたのは、Apache というウェブサーバーのプロジェクトである。Apache の最初のリリースは 1995 年 3 月 18 日であった。もともとはその名前の通り、[3] 似たような働きをする NCSA httpd 1.3 に対してパッチ（修正差分）を当てたものであった。現在、1.0 系列から 1.1 系列、1.2 系列、1.3 系列、2.0 系列までバージョンアップしながらリリースされてきており、今回は 1.3 系列を扱う。最新の 2.0 系列よりひとつ古いバージョンであるため、データが豊富にあるということ、また 1.2 系列から 1.3 系列の間に爆発的なマーケットシェアを得たと考えられることから、1.3 系列を扱うこととした。

1.3 系列の最初のバージョンは 1998 年 6 月にリリースされ、現在までリリースは続いているが、本報告では 2003 年 10 月まで（1.3.0 から 1.3.29 まで）をデータとして扱っている。[4] 基本的な機能はこの 1.3 系列であればおおよそ同じだが、リリースごとに機能追加や機能削除が行われている。

ソースコードファイルより、リリース 7 前後[5] でライン数[6] の大幅な増加と減少が起きて

[3] Apache という名称は A Patchy WWW Server から来ている。
[4] 1.3.0 から 1.3.29 までであるから実際には 30 リリースであるが、リリースしてすぐ公開をやめてしまったもの、リリースが中止されたものがあるので、それらをまびくと 5 年間で 19 回リリースされていることになる。
[5] 脚注 4 にある通り、リリースを 19 回として時期の早い順から各リリースに ID 番号をふった。
[6] ソースコードのライン数（行数）はプロジェクトの大きさを示す。

図 1　開発ログから得られる情報（改訂数）



いることがわかる。ここで何か起こったのではないかと調べてみると、その時期付近で開発者の分裂があったという話が聞けた。また、開発ログには変更箇所等の記録が書かれており、それをグラフ化したものが図 1 である。[7]

改訂は、変更部分の説明であり、不具合修正・新機能追加・他のプラットフォームへの移植情報・既定の設定値の変更などが含まれる。PR 付改訂は、ユーザーがコメントしていた箇所に対する改訂であり、不具合や新機能要望などがある。最初からリリース 5 くらいまでは改訂数が多く、その後いったん各リリースにおける改訂の数は少なくなるが、リリース 13 前後から再び回数が増えている。それに対して、ユーザーコメントによる変更はリリース 5 くらいで増加が頭打ちとなり、その後は全体の変更数にしめる割合が減少していることが見て取れる。

4. 設計情報の分析

以上のように、オープンソース・プロジェクトで配布しているソースコードファイルや開

[7] 改訂数は、各リリースに付属の CHANGES ファイル（改訂箇所の説明ファイル）をもとにカウントした。

発ログからはいろいろな情報を得ることができる。そしてさらに、詳細設計図を分析すると製品設計の構造が分析可能である。

ソフトウェアを作成する際、まずユーザーはインターネット上で配布されているソースコードファイルと構成設定ファイルをダウンロードし、これをあわせることで実行可能なファイル、すなわちソフトウェアを作る（この作業をビルド (Build) という）。

ソースコードはテキストファイルで書かれた多数の関数が記述されており、関数は、その関数内で他の関数を呼ぶ（関数をコール (Call) する）。つまり、ソフトウェアとは関数が集まったものと言うことができる。この関数同士の呼び合い構造をグラフ化したものがコールグラフと呼ばれるものである。グラフのまま分析するのは非常に難しいので、分析する場合はまず行列化し、最終的に指標化するという手順をふむ。

まず、要素間の関係がある場合を 1、ない場合を 0 とし、要素間の関係を行列化する。例えば、関数 A が F 関数を呼んでいれば 1 をつけ、そうでなければ 0 をつける。このようにして作成した行列を構造行列と呼ぶ。しかし、ソフトウェアというのは関数の集まりであるから、関数が何千もある。それをそのまま評価するのは非常に難しく、またあまり意味がないので、縮約化を行う。縮約化（ブロックモデル）とは、大きな構造行列のもつ関係構造を残しつつ小さな行列にすることである。呼び合いの関係をなるべく壊さず、可能な限り情報が失われないようにするためには、似たような構造の関数をまとめる。縮約化された結果の行列をイメージ行列という。

縮約化には大きく分けると二つの方法がある。ひとつは「演繹的ブロックモデリング」であり、すでに要素間にある程度の関係が想定される場合にそれを元に要素をまとめる方法である。もうひとつは「帰納的ブロックモデリング」と呼ばれる方法であり、この場合、データとある一定のアルゴリズムから事後的にブロックを求める。帰納的ブロックモデリングのアルゴリズムにはいくつかあり、CONCOR (Breiger, Boorman & Arabie, 1975) などが有名である。本報告では CONCOR アルゴリズムを採用した。

5. イメージ行列とアーキテクチャ指数

イメージ行列を構成するモジュール間の相互依存関係は様々である。

図 2 は完全モジュール型である。要素 1 は、要素 1 にしか依存していない（自ブロック内部で完結している）。要素 2、要素 3 についても同様で、他のモジュールと相互依存関係がない。図 3 は完全インテグラル型であり、要素 1 は、要素 1 とともに、要素 2、要素 3 にも依存している。図 4 は、モジュールが成立していない例である。要素 1 が要素 1 に依存していない、すなわち要素 1 に関しては、モジュールとして成立していない。これは共通部品な

図 2　完全モジュール型

	1	2	3
1	1	0	0
2	0	1	0
3	0	0	1

図 3　完全インテグラル型

	1	2	3
1	1	1	1
2	0	1	1
3	0	0	1

図 4　一般的な例

	1	2	3
1	0	1	1
2	0	1	0
3	0	0	1

どに見られるタイプである。[8]

ここで、設計構造が全体としてインテグラルあるいはモジュラルかをあらわす指数アーキテクチャ指数を以下のように定義する。

$$\text{アーキテクチャ指数} = \frac{\text{対角成分が 1 である行の、対角成分が 1 である列への関係数の合計}}{\text{行列の大きさ (分子・分母とも対角成分はカウントしない)}}$$

すなわちアーキテクチャ指数とは、イメージ行列において、対角成分が 1 である部分の個数をモジュール数とすると、成立しているモジュール数が行列の大きさに対してどれくらいであるかを数値化したものである。その性質としては 0 から 1 の連続変数になっており、完全にモジュラーの場合は 0、完全にインテグラルの場合は 1 となる。

6. 実際の手順

ソースコードのファイルをビルドするときに次々に関数を呼び出す過程をコンパイルと言う。このコンパイルを監視して関数の呼び合い構造を記録し、その関数に ID をつけて構造行列を作る。その結果、今回の 19 回のリリースでは、構造行列の大きさがおおよそ 1000×1000 から 1300×1300 の範囲で収まった。1.3.0 の場合は 984×984 という構造行列であったが、CONCOR アルゴリズムを用いて 4×4 のイメージ行列に縮約化した。CONCOR アルゴリズムは行同士の相関をとり、同じ行要素間の相関パターンをもつものをまとめて、最終的なイメージ行列を 2 の n 乗以内に分割する。ここでは $n = 2$ (4 × 4) を採用したので、イ

[8] 図 2、図 3、図 4 は、すべて有方向関係で記述されているが、これはソフトウェアの Call 構造（親子構造がある）を反映したものであり、特殊な形である。ソフトウェア以外の製品の場合は、図 2～4 までの行列を対称化し、無方向関係で記述することになる。

表 1　イメージ行列

	1.プロトコル解析関数群	2.プロセス制御エンジン	3.リクエスト処理エンジン	4.共通関数群
1.プロトコル解析関数群	0.002	0.001	0.001	0.002
2.プロセス制御エンジン	0.001	①0.008	0.001	0
3.リクエスト処理エンジン	③0.007	②0.007	0.004	0.003
4.共通関数群	0.004	0.002	0.003	④0.001

関係性の高いセル

メージ行列は 1 から 4 に分割された。

そして 1 から 4 までのラベルに含まれる関数を、筆者が関数名と機能を考慮し、各関数群毎に以下のように名前をつけた。

① プロトコル解析関数群：http プロトコルの解析およびそれに関係のある文字列操作
② プロセス制御エンジン関数群：http リクエストのサイクルの制御（プロセス制御）およびその記録
③ リクエスト処理エンジン関数群：http リクエストと関係する各処理モジュール（コンテンツ種類の判別や権限制御）のネゴシエーション
④ 共通関数的関数群：設定ファイルやサーバの状態から、変数を設定する

上記のような名前を付け、数値化したものが表 1 である。

この表から関係性の高いセルが三つあることがわかる。①はプロセス制御エンジンであり、処理効率を向上させるためにモジュール内での依存度が高く、数値にもそれが表れていると考えられる。②、③はそれぞれプロセス制御エンジンとリクエスト処理エンジン、プロトコル解析とリクエスト処理エンジンであり、役割的に見て非常に結びつきが強いために高い数値となっている。一方、④のセルは関係性が低い（0.001）。ここに分類されているのは共通関数的なものであり、文字列処理関数やメモリを確保する関数である。そのため、どこからも呼ばれるが自らで自らを呼ぶことは少ないので数値が低いと考えられる。

図 5　アーキテクチャ指数（4×4 行列）

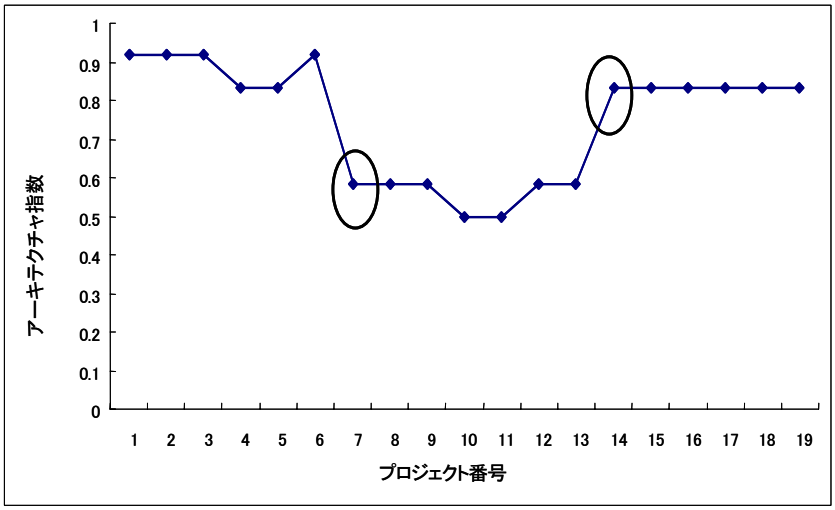


図 6　アーキテクチャ指数の変化（分割レベル別）

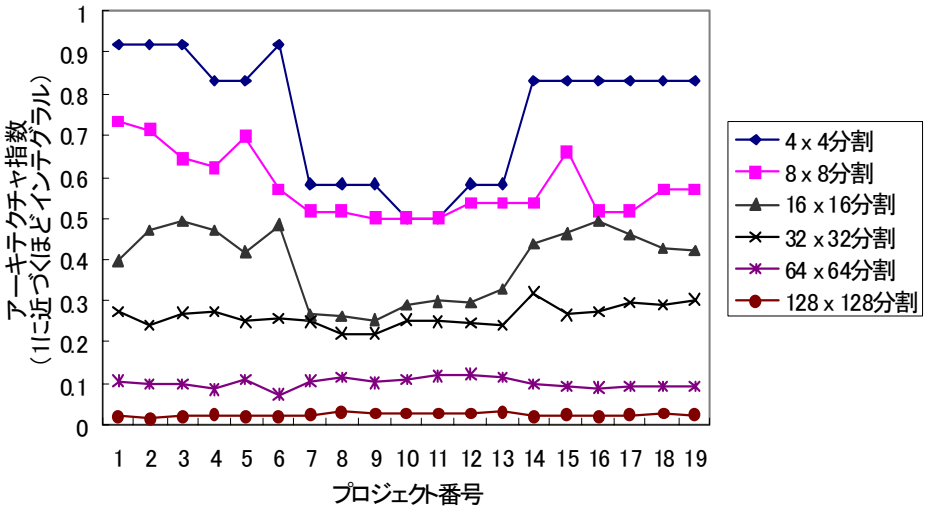


図 5 は、先述のアーキテクチャ指数（1 にちかづくほどインテグラル）を縦軸に、プロジェクト番号を横に取ったグラフである。これを見ると、6 番目のリリースにおいてアーキテクチャ指数が低下しているのがわかる。開発ログを参照すると、ここで新しいモジュールを追加するように設定ファイルを修正したと記載されている。同様に 14 番目のリリースではアーキテクチャ指数が上昇しているが、ここでもセキュリティの問題が生じたので内部構造

の変更があったことが開発ログよりわかる。

図 6 は、4×4 分割以外にも、より分割を細かくした場合にアーキテクチャ指数の推移がどのようになるかを示したグラフである。

分割数を大きくした場合、32×32 分割までは 4×4 分割の場合と同じような傾向が見られる。しかし、下の二つ、すなわち 64×64 分割、128×128 分割に関しては、アーキテクチャ指数の変化がほとんど見られない。これは、あまりに細かく分割しているためにそもそもモジュールの関係性が観察できなくなっているためであると考えられる。

4×4 分割と 16×16 分割においてはグラフが U 字型になっている。8×8 分割はプロジェクト番号が大きくなるにつれて、インテグラルからモジュラルへ推移しているように見える。32×32 は若干ではあるがモジュラルからインテグラルへの推移が見て取れる。これらの背後にある要因はまだ研究の途上であるが、少なくとも、観察する分割レベルによって、モジュール間の関係が変化してしまう可能性がある、ということは言える。

7. まとめ

オープンソースのソフトウェアを研究対象にすることで、開発ログやソースコード（設計情報）が公開されていることにより様々な情報を得られる。さらに、構造行列の縮約という手法を用いることで、設計構造そのものを分析対象とすることが可能である。そこでは 4×4 という分割をしたときに、機能に名前が付くような分割の仕方をすることができた。

アーキテクチャ指数を定義することにより、プロジェクト間の時系列的なアーキテクチャの変化を数量化することができた。ただし、構造行列の縮約レベル（分割レベル）によって、構造の変化の見え方が異なる可能性があるので、どのレベルでの分析なのかを注意する必要がある。

今後の研究を発展させる方向としては、三つ考えられる。

第一に、「ソフトウェア・アーキテクチャ」論や「デザイン・パターン」論との関係性である。『ソフトウェアアーキテクチャ―ソフトウェア開発のためのパターン体系―』 (2000) は、ソフトウェアの設計を抽象度の高いデザインの再利用の観点から考察している。パターンとは頻出する設計上の問題についての有効性が実証された解決策である。設計のパターンは、その抽象度に応じて、最も抽象度の高い「アーキテクチャ」（例：レイヤー構造や MVC 構造など）、最も抽象度の低い「イディオム」（例：C++におけるメモリ管理、メソッドの命名など）、アーキテクチャとイディオムの中間である「デザイン・パターン」の三つに分類できるとしている。デザイン・パターンの例としては、Master－Slave パターン、View Handler パターンなどが代表的である。デザイン・パターンは、コンポーネント間の連係構造を記述

したものであり、汎用的であるがアーキテクチャほど普遍的でなく、特定の状況においては設計上の課題の解決策となる。これがどのように本報告とつながるかは定かではないが、「構造」をいくつかのレベルで考えるという視点は、今後の分析に利用できるのではないかと考えられる。

第二に「デザインと経営戦略」論という方向性である。『デザイン・ルール　モジュール化パワー』(2004) では、デザインと経営上の意志決定との関係性について記述されており、その第 3 部ではリアルオプション論への発展が試みられている。これに従い、イメージ行列の評価、すなわち、良いモジュールと悪いモジュールの数値化が可能かもしれない。（具体的には、組み合わせの価値から組み合わせの検証コストを減じることでオプション価値計算をしているが、それが大枠として正しいかは、また別の議論である。）非常に注意が必要であるのは、この分野は、まだ研究上のフロンティアであって、早急な結論は出せないとおもわれる。

第三に、品質工学（タグチメソッド）との関連性の整理である。『品質を獲得する技術　タグチメソッドがもたらしたもの』(2000) では、品質工学をわかりやすく解説している。品質工学の扱っている問題のうち、例えば、パラメタ設計という概念は、先に挙げた『デザイン・ルール』中のデザイン構造行列（DSM: Design Structure Matrix）とほぼ同じ概念を扱っており、より実践的で精緻である。（端的に言えば、DSM では制御因子も標示因子もすべて同一に扱われているが、品質工学では当然これを区別する。）筆者としては、全く異なる研究分野から同じような概念が提出されていること自体が非常に興味深い。

参考文献

Breiger, R., Boorman, S., & Arabie, P. (1975). An algorithm for clustering relational data, with applications to social network analysis and comparison with multi-dimensional scaling. *Journal of Mathematical Psychology, 12*, 328-383.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996). *Pattern-oriented software architecture: A system of patterns*. Chichester, UK: John Wiley & Sons. 邦訳, F･ブッシュマン, R･ムニエ, H･ローネルト, P･ゾンメルラード, M･スタル (2000)『ソフトウェアアーキテクチャ―ソフトウェア開発のためのパターン体系』 金澤典子, 水野貴之, 桜井麻里, 関富登志, 千葉寛之 訳. 近代科学社.

Baldwin, C. Y., Clark, K. B. (2000) *Design rules: Vol. 1. The power of modularity*. Cambridge, MA: MIT Press. 邦訳, C･Y･ボールドウィン, K･B･クラーク (2004) 『デザイン・ルール　モジュール化パワー』安藤晴彦 訳. 東洋経済新報社.

宮川雅巳 (2000)『品質を獲得する技術　タグチメソッドがもたらしたもの』日科技連出版社.

赤門マネジメント・レビュー 3 巻 12 号 2004 年 12 月 25 日発行

編集　東京大学大学院経済学研究科 ABAS/AMR 編集委員会

発行　特定非営利活動法人グローバルビジネスリサーチセンター

理事長　片平 秀貴

東京都千代田区丸の内

http://www.gbrc.jp