オペレーティング・システム（OS）の アーキテクチャと開発技法： アーキテクチャからみた LINUX の含意

Architectures and Developing Methods of Operation Systems:
Implication of LINUX

竹田陽子
国際大学グローバル・コミュニケーション・センター助教授
Associate Professor
Center for Global Communications, International University of Japan

小山 裕司
国際大学グローバル・コミュニケーション・センター専任講師
Hiroshi Koyama
Assistant Professor
Center for Global Communications, International University of Japan

要約

本稿では，増大する複雑性に対処するため，機能の配分を単純化し，構成単位の相互関係に規則を設定し，インターフェース部分と内部構造を分離するという，モジュール化の進展という観点から，コンピュータ・ソフトウェア，とりわけ OS（オペレーティング・システム）のアーキテクチャと開発技法について論じ，近年サーバ OS で大きなシェアをしめる Linux の成功の含意について考えてみたい。

Linux は，オープンソース，つまり内部構造を公開することで多くのプログラマーの参加を可能にし，高い安定性を実現している。これは，一見，内部構造をブラック・ボックス化してインターフェースのみを規定するというモジュール化とは逆の方向性である。

Linux の含意は，従来のモジュール化が部品間の関係など，客観的な基準としての相互依存関係の大きさに従っておこなわれるものと考えられていたのに対し，複数の参加者が共通の関心と知識体系を持ち，十分なコミュニケーションが出来る範囲に開発目標の規模と内容を設定することが構成単位の基準となりうるのではないかということである。コミュニケーションを基準にしたモジュール化である。

Summary

Architectures and developing methods of software can be regarded as the devices of reducing complexity. From this point of view, we discusses about LINUX, a successfully diffused operating system.

Comparing the other Unix family operating systems, the architecture of LINUX cannot be seen as more modular. Developers of LINUX all over the world share the information of its internal structures not only that of interfaces. In terms of coordinating developers closely, Linux is developed to a highly stable OS.

Linux itself could be thought as a module in the total system that is divided based on the size or range in which communication is the most effective.

わずか 30 年前には，企業や大学でおこなわれるすべて計算処理を 1 台の大きなメインフレーム・コンピュータで行っていた。現在では、オフィスの机上や各家庭にあるパーソナル・コンピュータで、これをはるかに上回る計算処理を行うことができ、インターネットを通じて地球の裏側の相手とも自由自在に情報交換や協働活動を行うことができる。この間，ハードウェアの分野では，いくつもの技術革新が行われ，真空管からトランジスタ，集積回路，LSI，VLSI と指数関数的に成長を遂げた。しかし、ソフトウェアの分野では，ハードウェアの進化に対応して従来に比べ 10 の 3 乗倍程度の計算機資源を扱ったり，ネットワーク上で数百台のマシンを接続する必要が出てきたのに関わらず，ハードウェア分野で起こったほどの革新は無かった。注1

ソフトウェアがハードウェアの進化に追いつけない大きな要因は，ソフトウェアが大規模になるに従って，ソフトウェア開発のコストは線型以上で上がり，生産性は線型以下で下がることにある。ソフトウェアの規模が増加するほど，構成要素間の相互依存関係，すなわち複雑性が線形以上で増すのである。

増大する複雑性に対処するため，コンピュータ・ソフトウェアのアーキテクチャと開発技法においては，機能の配分を単純化し，構成単位の相互関係に規則を設定し，インターフェース部分と内部構造を分離するという，モジュール化の試みが早くからおこなわれてきた。しかし，ソフトウェア分野におけるモジュール化は理論通りには進んでいない。例えば，近年サーバ OS で大きなシェアをしめる Linux は，オープンソース，つまり内部構造を公開することで多くのプログラマーの参加を可能にし，高い安定性を実現している。これは，一見，内部構造をブラック・ボックス化してインターフェースのみを規定するというモジュール化とは逆の方向性である。

本稿では，モジュール化の進展という観点から，コンピュータ・ソフトウェア，とりわけ OS（オペレーティング・システム）のアーキテクチャと開発技法について論じ，Linux の成功の含意について考えてみたい。

1. 相互依存関係をいかに削減するか

ソフトウェア開発では，目的のソフトウェアを構築するために，どこからソフトウェア構成要素を集めてきてどのように組み立てるか，構成要素間の相互依存関係をいかにして削減するかが設計上の中心課題となる。

相互依存関係を削減する手段としては以下のものが挙げられる。

i. 機能水準を落す。
ii. 機能水準を落とさずに、
    1. 機能の配分を単純にする。

2. 相互関係に規則を設定する。(→インターフェースの形成)
3. インターフェース部分と内部構造を分離する。

ii の 1→2→3 はモジュール化がおこなわれていく過程そのものであり，ソフトウェアの設計方法の進化もこの順番を辿っている。

1.1 機能配分の単純化:プログラムの構造化

1970 年代の初め、データ構造，プログラム構造，制御構造という 3 つの概念によって，整理された構造的なプログラムを書こうとするプログラミング・スタイルが提唱された。これによって，プログラムを単純な構造にすることができ，ソフトウェアを統一的に開発することが目指された。これは，ii の 1 の機能の配分を単純化する手法といってよいであろう。

データ構造とは，各種のデータを簡潔で明示的に表現する方法である。プログラム構造とは，複数の文をひとまとめにしたり，繰り返し・分岐の単位を指定したり，変数の有効範囲を指定したりすることである。制御構造とは，厳選された制御だけを使ってプログラミングを行うことである。

例えば，制御構造では，分岐(if～else)，繰り返し(while)を使うことによって，論理的でシンプルな構造をつくりだすことができる。逆に，プログラムの任意の場所に飛ぶコマンドである goto を多用することは，論理的に一貫性がなくなるので望ましくない。現在，これらの指針はプログラミングの常識として普及している。

1.2 相互関係の規則化:オブジェクト指向

1980 年代の終わりには，「オブジェクト指向」という，インタフェースが規定されたソフトウェア構成要素を集めてきて，組み立てることで，目的のソフトウェアを構築するプログラミング・スタイルが現われた。これによって，ソフトウェアの生産性を飛躍的に改善できると考えれ，C++，Java など，オブジェクト指向の考え方が登場した以降に設計されたほとんどのプログラミング言語にこのスタイルが反映された。

構成要素から目的のソフトウェアを構築する作業の概念図を図1に示す。ここで、新しいソフトウェアは，一般に使われているベース・ライブラリと Jim，Alex という他の二人のプログラマーが作ったライブラリから構成要素となる小さなソフトウェアを集め，相互に関連付けることによって設計されている。

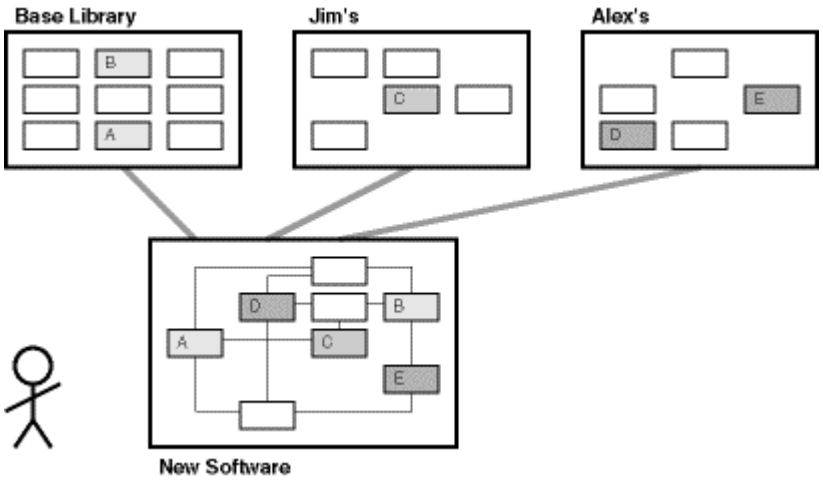

図 1: 再利用によるソフトウェア構築

UNIX には，オブジェクト指向という言葉が生まれるかなり以前から，フィルタと呼ばれる再利用できる小さいソフトウェア(図1の A, B, C, D, E・・・)を組み合わせて使うことによって，目的のソフトウェアを構築するという概念があった。フィルタは入力からデータを受け取り，何らかの処理を行い，結果を出力に流していく。複数のフィルタは，パイプラインと呼ばれるデータの流れで互いに関連付けられる。この仕組みはソフトウェアの再利用の走りである。

例えば，**grep** というフィルタを使うと，巨大のファイルから特定の文字列を抜き出すことが出来る。パソコンにたまってしまった膨大な受信メールの（圧縮された)記録ファイル messages の中から自分のアドレス（koyama@glocom.ac.jp）に関係するものだけを抜き出して調べたいときは，次のようなコマンドを実行すれば良い。

```
zcat messages | grep koyama@glocom.ac.jp | more
```

zcat は圧縮されているファイル(messages)からデータを取り出すという働きを，more は結果を 1 画面単位で表示するという働きをする。ここでは，zcat，grep，more という 3 つのフィルタがパイプライン「|」で連結され，一連のデータ処理の流れを作っている。ここで，重要な点は，新しくソフトウェアを作る人は，構成要素となる小さなソフトウェアの働きだけを知っていればプログラミングができることである。grep というフィルタの詳細を知らなくても，grep を利用したソフトウェアの設計はできるのである。構成要素間の相互関係の規則化がおこなわれている段階（前述 ii の 2）にあると言える。

1.3 相互関係の規則化：データ・フォーマットとネットワーク・プロトコルの標準化

ソフトウェアが扱うデータやコンピュータ間の通信に関する規則（ネットワーク・プロトコル）を標準化するのも，構成要素間の相互関係の規則を設定（前述 ii の 2）を実現し，ソフトウェア開発の生産性を向上させるひとつの方法である。

各種アプリケーション・ソフトウェアのデータフォーマットのうち，標準が存在したり，仕様が公開されているものは，複数のアプリケーション・ソフトウェア間で相互のデータの交換が可能になる。例えば，ホームページを記述する HTML がこれに相当する。HTML のデータを扱うことができるソフトウェアは多数存在するが，異種のソフトウェアを持ったユーザーが互いに作成したホームページを閲覧したり，データを再加工するのに，変換ソフトを用意する必要はない。

コンピュータ間のデータのやりとりの規則であるネットワーク・プロトコルの標準化も同じような作用がある。インターネットでは，電子メールやファイル転送等で使用するネットワーク・プロトコルの標準が決まっているため，数多くのソフトウェアが存在し，これらを自由に選択して利用することができる。

1.4 インターフェースの分離：ソフトウェア IC

インターフェース部分を内部構造から分離し，構成要素を結びつけるのに内部構造に一切関わらなくてもすむようにするのが，相互依存関係削減の究極の姿である。

インターフェースを内部構造から完全に分離し，再利用可能性を高める手法として，ソフトウェア IC の概念が提唱されている。ソフトウェア IC は，名前の通りハードウェア IC のように汎用で再利用できるソフトウェア構成要素のことである。外部に公開する必要が無いデータを構成要素の内部に隠蔽し，制限されたインターフェースを準備することで，利用者が構成要素を利用するにあたって理解すべきものはこのインターフェースだけに減らすことができる(図 2)[Carroll 1995]。

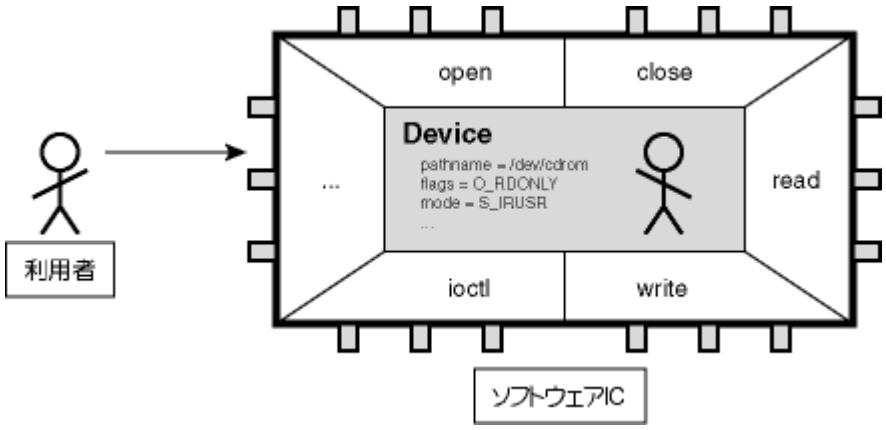

図 2: ソフトウェア IC の概念図

2. モジュール化の障害

ソフトウェアのアーキテクチャと開発技法にオブジェクト指向等のモジュール化の思想が唱えられてから久しいが，ソフトウェア開発の生産性は飛躍的に向上しているとはいえない。その原因は相互依存関係の規則化が完全におこなわれていないということに尽きる。相互依存関係の規則は，通常，仕様（ドキュメント）という形で明示化されるが，仕様にはしばしば曖昧さが残っていたり，仕様には書かれない暗黙の相互依存関係が存在することがある。

例えば，「関数 A を呼ぶ前に～を行うこと」等の暗黙の依存関係が仕様から漏れている場合がある。意識して情報を隠蔽しているとは限らないが，内部情報を知っている人はいつのまにか内部情報に依存したコードを書いてしまうことがあり，それが重要なノウハウにつながるかもしれない。クリアな仕様を書くということは簡単ではないである。

バージョンアップの際に前のバージョンの仕様との整合性がとれていない場合や，各社が独自に仕様を拡張した結果，仕様が変わってしまうこともしばしば起きる。インターネットでホームページを見るのに，Netscape Navigator を使う場合と Microsoft Internet Explorer を使う場合で見え方が異なることを経験した方は多いのではないかと思われるが，これは後者の例で，ブラウザによって HTML の解釈の違いがあるためである。

意識的に一部の機能が隠蔽されることもある。このことは，普及した OS（オペレーティング・システム）の開発会社がアプリケーション・ソフトを開発する際の公正競争上の問題になっている。OS 開発会社にしかわからないノウハウが存在するため，OS 開発会社がアプリケーション・ソフト市場でも常に競争上優位に立つことが危惧されているのである。

3. OS（オペレーティング・システム）の開発とアーキテクチャ

OS とは，MS-DOS，UNIX，Windows など、アプリケーション・ソフトウェアとハードウェアの間をつなげる基盤ソフトウェアのことで，ファイルとディレクトリの管理，利用者管理，資源管理，入出力処理，通信等の機能がある。以下では，各 OS がどのように開発され，どのようなアーキテクチャを持っているかについて紹介する。

3.1 MS-DOS

MS-DOS は 1980 年～1990 年代前半に普及したマイクロソフト社の商用 OS(ソースコードは非公開)である。構造自体は非常に単純であり，D（=Disk）OS という名前の通り，ディスクの入出力管理を中心に設計された OS であるために，ディスク資源以外の管理に関する機能は不十分であった。また，当初この OS は数名のエンジニアによって，これほど普及することを念頭に置かずに開発されたため，ある程度の構造はあったが，アプリケーション・ソフトウェアとハードウェアをつなげる各種機能に明確な階層構造がなかった。

このため，MS-DOS では，アプリケーション・ソフトウェアがディスプレイ等のハードウェアを直接制御することが可能で，システム全体をクラッシュさせる原因を生み出すソフトウェアを書くことも可能であった。

MS-DOS のアーキテクチャは，機能配分の単純化，構成要素間の相互関係の規則化が不十分で，インターフェースがはっきりしない段階にあったといえる。

3.2 UNIX

UNIX の原形は，MS-DOS 以前の 1960 年代の終わりごろに，AT&T のベル研で商用製品にすることを意図されずに開発された。初期にはソースコード（プログラムの中身）がオープンにされており、ほとんど無料で教育・研究機関に配布されたこともあり，多くのプログラマーが開発に寄与して，実用に足る OS に発展していった。特に，仮想記憶やネットワークの機能は有名で，当時 UNIX で導入されたアーキテクチャ及び概念の多くは現在でも通用する。その後、ワークステーションの OS として採用されるようになると，サン・マイクロシステムズ社など各社が独自の UNIX を開発するようになり，ソースコードは非公開になった。

UNIX では，各種機能に階層構造が見られ，機能間のインターフェースの規定が行われている。UNIX の本体は，単一のファイルから構成されているが，プリンタ機能やネットワーク関係の機能の一部は OS 本体から分離され（サーバと呼ばれる），基本コマンドの多くも外部コマンドとして分離されている。必要に応じてこれらの機能が呼び出されるのである。MS-DOS のようにハードウェアを直接制御するコードを書くことは出来ない。

UNIX は，MS-DOS に比べて構成要素間の相互関係の規則化がなされているといえる。しかし，本体部分はひとつのファイルとして一体となっており，さまざまな機能を組み合わせて使うという段階には至っていない。

3.3 Windows

今日広く使われている Windows は，当初， MS-DOS をグラフィックによって直感的に使いやすくするためのソフトウェア（GUI：グラフィカル・ユーザ・インターフェース）としてマイクロソフト社により開発された。1985 年の初期バージョン Windows 1.0 から一般に普及した Windows3.1 までは，MS-DOS の GUI である。その後，Windows 95 以降は，MS-DOS を補完するソフトウェアとしてではなく，新規の OS として開発されるようになった。しかし、MS-DOS 上で動くソフトウェアを使えるようにするために，Windows のアプリケーションに対するインターフェースだけでなく，MS-DOS のソフトウェアを動かすためのインターフェースを別に用意しなければならなかった。このため，構造が複雑になることが避けられず，Windows は，いまだに UNIX 等に比べて動作が不安定である。注 2

MS-DOS と Windows では，ソースコードを公開せずマイクロソフトの社内のみで開発するという開発戦略が貫かれているが，アーキテクチャの面から見ても，過去の経緯により機能配分を単純化してインターフェースを規定することが難しいため，もともと外部のプログラマーと協力しにくいという性質を持っている。今後も，要求される機能が拡大するにつれますます複雑になる構造を，すべて社内で開発しなくてはならないという宿命を負っているのである。

3.4 マイクロカーネル OS

各種の機能が取り込まれ，巨大化する一方の OS の現状に対し，1980 年代の初めごろから，「カーネル(基本)部分に本当に必須の基本機能だけを残し，従来の OS の残りの機能は必要に応じ利用できる部分として分離する」というマイクロカーネル OS という基本概念が提唱されるようになった。

カーネル部分に残る機能は，基本的な入出力，メモリの管理，通信機能，複数のソフトウェアを同時に動かすための機能程度であり，ファイルとディレクトリの管理，利用者管理等も分離される(図 3)。「カーネル部分から取り除くことができるものはすべて取り除く」という哲学でマイクロカーネル(最小)のカーネルが実現されるわけである。

マイクロカーネル OS の概念は，機能配分の単純化，相互依存関係の規則化がかなり徹底され，モジュール化がすすんだ状態にある。

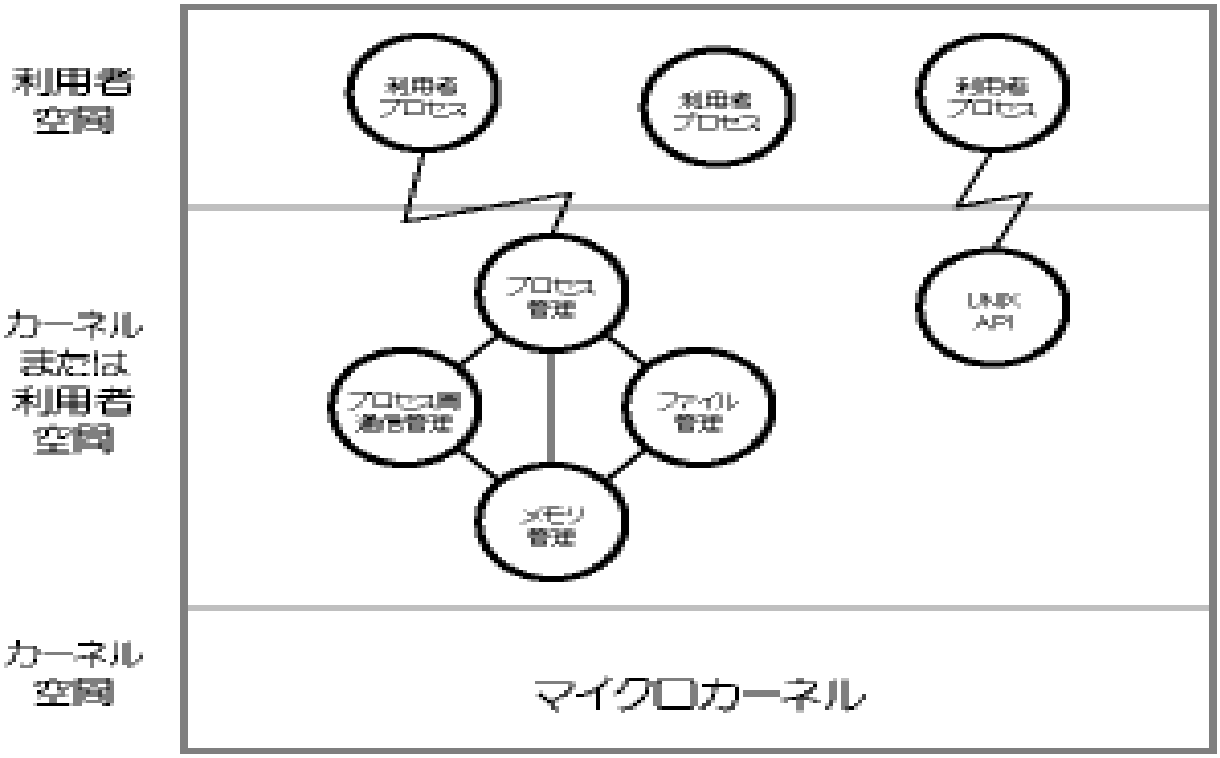

図 3: マイクロカーネル

3.5 Linux

Linux（リヌックスあるいはリナックスと発音する）は，ソフトウェアの内部構造であるソースコードが公開され，世界中のプログラマーが開発に参加しているオープンソース・ソフトウェアの代表格として知られている。インターフェース部分だけでなく内部構造がすべて公開されているという意味で，モジュール化の流れとは逆を行っているわけである。

Linux の歴史は 1991 年に最初の開発者が 386 CPU の動作の勉強をすることから始まった。この後，ソースコードがインターネットで公開され，誰でもが開発に参加できる開発スタイルを取ったために，インターネット上でプログラマーが積極的に機能改良・拡張を行い，現在のように発展を遂げた。現在，Linux は，サーバ OS の約 20%程度のシェアを占めている。

Linux の構造は，当時積極的に研究が行われていた新しいマイクロカーネル構造ではなく，従来の単層カーネル構造（多くの機能が OS 本体にひとつになっている構造）であった。Linux が開発の初期段階にあった 1985 年 11 月には，OS 研究の権威から「Linux is obsolete(Linux は時代遅れだ)」だと非難され，ネットワーク上で論争が行われる事件が起こっている。それにも関わらず，Linux は，UNIX 互換 OS として完成度を次第に高め，普及，発展を遂げてきた 。

Linux の最初の開発者は当時の論争に関して，「当時，マイクロカーネルは実験段階で，従来の単層カーネル構造よりも複雑で，従来の単層カーネル構造よりもかなり遅かった。現実の OS は速度が重要である。マイクロカーネルが速度を稼ぐために使っている手法は従来のカーネル構造ではごく当り前に使われているものだ。」と書いている[DiBona 1999]。Linux の開発は，すでに確立した構造の中で多くの開発者が参加して精緻に性能を高めるプロセスであったといえる。マイクロカーネルなどのモジュール化された構造ではどうしても冗長な部分が増え，無駄のない最適の設計をするのは難しい。要求される機能が従来と変わらなければ，内部構造を公開して最適の設計を追及するほうが効率が上がるのである。

しかしながら，1991 年 9 月 17 日に公開された最初の linux-0.01 のソースコードは 319,488 バイト(100 ファイル)だったものが，2000 年 05 月 25 日に公開された最新版 linux-2.4.0-test1 では 89,487,360 バイト(7,563 ファイル)にもなっている。約 8.5 年で 285 倍に成長してきたわけである。

Linux に要求される機能が増えるにつれ，巨大化していくことは避けられない。内部構造をブラック・ボックスにしないことで，多くの開発者が参加し，精緻化を極めるという Linux の開発体制が今後も有効であるかは未知数である。

4. Linux モデルの含意：コミュニケーションを基準にしたモジュール化

ソフトウェアに限らず，多くの人間や組織が参加してひとつの仕事を成すときの大きな制約となる条件は，必要なコミュニケーションが合理的なコストで可能かどうかである。機能配分の単純化→相互関係の規則化→インターフェースと内部構造の分離というモジュール化の流れは，インターフェースのみを知っていればコミュニケーションが可能になるというメリットがあるために考えられた。しかしながら，ソフトウェアのアーキテクチャの歴史は，機能配分や相互関係に関するルール（インターフェース）を事前に厳密に決定することが意外に難しいことを示している。

一方，内部構造を公開する Linux モデルが上手く働いたのは，開発目標が既に存在する OS の互換ソフトウェアという手頃な規模であったことにあるのかもしれない。つまり、1 節で述べた相互依存関係削減の手段の「i. 機能水準を落す」にあたる。そうであるとすると，ソフトウェアをいたずらに巨大化させずにインテグラルな開発プロセスが可能な範囲に開発規模をとどめることが成功要因であると言える。

しかし，現代においては，ますますいろいろな機能が連動することが求められている。シンプルな機能のソフトウェアを単体で使うだけではニーズは満たされない。そのため，オープンソースによって開発された各単位が他の部分とインターフェースによって連結される，つまりモジュール化されることは避けて通れないであろう。

Linux の含意は，従来のモジュール化が部品間の関係など，客観的な基準としての相互依存関係の大きさに従っておこなわれるものと考えられていたのに対し，複数の参加者が共通の関心と知識体系を持ち，十分なコミュニケーションが出来る範囲に開発目標の規模と内容を設定することが構成単位の基準となりうるのではないかということである。コミュニケーションを基準にしたモジュール化である。

そして，各モジュールの内部では，共通の関心と知識体系を持った人間が集まって開発するわけであるから，情報がすべて公開されても，それを処理する能力は極めて高いのである。オープンソース・ソフトウェアはまさにそのような場合に成功する。

しかも，今やインターネットによって，共通の関心と知識体系を持った人間を世界中から集めて共同作業できる。地理や時間的な条件の制約があった頃に比べて，モジュールの決め方にさまざまな可能性が生まれてきたということが，Linux モデルの真の意義ではないであろうか。

注釈

1） いまだにソフトウェアの価値は，開発にかかる人員数と月数の掛け算である「人月」という尺度で表現されていることがその象徴である。人月という尺度の論理自体が間違っていることはプログラミングの経験があればすぐわかる。優れた構造，優れた開発手法で作られたソフトウェアほど，人月は少なくなるはずであるからである。また，1 名が 10 ヶ月かかってできるソフトウェアも，10 名が 1 か月かかってできるソフトウェアも同じ価値であるはずがない。

2）Windows NT と Windows2000 は，Windows 95～98 とは互換性があるが MS-DOS や Windows3.1 とは互換性がない，まったく異なる OS として開発された。複数のアプリケーション・インターフェースを持つ必要がないため、Windows NT～2000 は OS として Windows 95～98 よりも遥かに安定している。

参考文献

[Meyer 1988]
Bertrand Meyer, Object-oriented Software Construction,
Prentice-Hall, 1988.（二木 厚吉監訳，酒包 寛，酒包 順子訳，オブジェクト指向入門，アスキー，1990 年）

[Carroll 1995]

Martine D. Carroll and Margaret A. Ellis, Designing and Coding
Reusable C++, Addison-Wesley, 1995.（小山 裕司, 中込 知之訳, C++言語
書法: コード再利用の奥義と実践, トッパン, 1996 年）

[DiBona 1999]
Edited by Chris DiBona, Sam Ockman and Mark Stone,
Open Sources: Voices of the Open Source Revolution,
O'Reilly & Associations, Inc, 1999.

[Silberschats 1994]
Abraham Silberschats and Peter B. Galvin, Operating System Concepts,
Fourth Edition, Addison-Wesley Publishing Company, 1994.