

1 ソフトウェア管理技術の現状

松本健一

奈良先端科学技術大学院大学
matumoto@is.aist-nara.ac.jp



本稿では、ソフトウェア管理技術に関するトピックスを「プロセス」、「プロダクト」、「リソース（資源）」の3つの観点から1つずつ取り上げる。プロセス観点からは「CMMI」、プロダクト観点からは「ソフトウェア脆弱性」、そして、リソース観点からは「プロダクトライン」である。さらに、今後の展望の1つとして、オープンソースとソフトウェア管理のかかわりを3つの観点から述べる。

◆ CMM から CMMI へ

◆ 成熟度モデル

成熟度モデル（Capability Maturity Model: CMM）とは、米国 Carnegie Mellon 大学のソフトウェア工学研究所（Software Engineering Institute: SEI）が米国国防省の要請により作成した、よりよいソフトウェア開発を行うための一連のモデルである。ソフトウェア開発を行う組織（会社やその中のグループ）がソフトウェアプロセス改善を行うための数々の指針が記述されている^{1)・2)}。なお、当初 CMM はソフトウェア開発のみを対象とされていたが、その後、ソフトウェア開発以外を対象とするモデルがいくつか提案された。それらと区別するため、現在では、ソフトウェア CMM（Capability Maturity Model for Software: SW-CMM）と呼ばれている。

一方、CMM 統合（Capability Maturity Model Integration: CMMI）とは、ソフトウェア中心のシステムや製品に対して開発プロセスの改善を支援する道具を提供するために、同じく SEI が進めているプロジェクトである³⁾。具体的には、SEI がこれまで提案してきたモデルのうち次の3つを統合する活動である。

- ソフトウェア CMM（Capability Maturity Model for Software: SW-CMM）
- システムエンジニアリング CMM（Systems Engineering Capability Maturity Model: SE-CMM）
- 統合製品開発 CMM（Integrated Product Development Capability Maturity Model: IPD-CMM）

なお、ソフトウェア調達 CMM（Software Acquisition Capability Maturity Model: SA-CMM）は、CMMI の現行バージョン（V1.1）では統合されていない。また、ピープル CMM（People Capability Maturity Model: P-CMM）は、当初から統合の対象にはなっていない。

CMMI の主な目的は、成熟度モデルの「統合」と「改良」である。近年、SW-CMM の成功もきっかけとなって、SEI が提案する他の CMM も利用されるようになってきた。モデル適用分野の広がりや、結果として、複数モデルの利用による混乱と高コストを招くこととなった。「モデル統合」による冗長性の排除、用語・様式の統一は、プロセス改善活動そのものの統合でもある。また、CMM を利用するプロセス改善コミュニティの成熟に伴って、高い成熟度レベルが実現され、多様なプラクティスが得られるようになってきた。これらをモデルに取り込むとともに、ISO/IEC 15504 との整合性も確保するという「モデル改良」も CMMI が目指すところである。

◆ CMMI におけるモデル統合

(1) 4つの専門分野のサポート

次に示す4つの専門分野（Discipline）の組合せ（SW, SE/SW, SE/SW/IPPD, SE/SE/IPPD/SS）でモデル文書が用意されている。

- システムエンジニアリング（System Engineering: SE）：システム全体の開発（ソフトウェアを含む、含まないにかかわらず）を取り扱う。

	SW-CMM	CMMI: Staged
Level	Maturity Level 1.Initial 2.Repeatable 3.Defined 4.Managed 5.Optimizing	Maturity Level 1.Initial 2.Managed 3.Defined 4.Quantitatively Managed 5.Optimizing
# of Areas	Key Process Area 18	Process Areas 25 (22 for SW)
Goal	Goal	Goal • Specific Goal • Generic Goal
Practice	Key Practice	Practice • Specific Practice • Generic Practice
# of Common Features	5	4

表-1 モデル構成要素の比較

- ソフトウェアエンジニアリング (Software Engineering: SW) : ソフトウェアシステムの開発を取り扱う。
- 統合プロセス成果物開発 (Integrated Process and Product Development: IPPD) : 成果物開発に対する系統的なアプローチで、顧客ニーズをより満足するために、成果物のライフサイクル全般にわたる関係者間でのタイムリーな協調を実現する。
- 供給者ソーシング (Supplier Sourcing: SS) : 外部からの調達に特に重要な場合、供給者の分析、選定、監視と成果物の調達を取り扱う。

(2) 2つの表現形式のサポート

プロセス改善の達成度を、次に示す2つの形式で表現することができる。

- 段階表現 (Staged Representation) : SW-CMM で用いられてきた形式で、組織の「成熟度レベル (Maturity Level)」によって改善の達成度を表す。成熟度レベルにより、達成する「プロセス領域 (Process Area)」をグループ化し、達成度を1次元 (レベル 1, レベル 2, ..., レベル 5) で表現する。
- 連続表現 (Continuous Representation) : SE-CMM で用いられてきた形式で、プロセス領域の「能力レベル」によって改善の達成度を表す。プロセス領域は、「プロセス管理」, 「プロジェクト管理」, 「エンジニアリング」, 「支援」の4つである。「プロセス領域」と「能力レベル」の2次元で達成度を表現する。

(3) モデル構成要素の変更

モデルの主要な構成要素である「成熟度レベル」, 「プロセス領域 (エリア)」, 「目標 (ゴール)」, 「プラク

ティス」, 「コモンフィーチャ」の内容が変更された。SW-CMM と CMMI 段階表現との比較を表-1 に示す。

◆ CMMI におけるモデル改良

(1) プロセスエリアの変更

表-1にも示したとおり、SW-CMM が定めた18個のキープロセスエリアが統合、分割され、22個のプロセス領域となった (4つの専門分野全体では25個)。特に、レベル4に関しては視点の変更が行われ、「定量的プロセス管理」, 「ソフトウェア品質管理」から「定量的プロジェクト管理」, 「組織プロセス実績」に変更された。

(2) 2種類のゴールの導入

表-1にも示したとおり、プロセス改善のゴール (達成目標) が次の2種類に分類された。

- 固有ゴール (Specific Goal) : プロセス領域に固有の達成目標。例: 「見積もりを確立する」, 「プロジェクト計画を策定する」, 「計画に対するコミットメントを獲得する」 (いずれもプロセスエリア「プロジェクト計画策定」の固有ゴール)。
- 共通ゴール (Generic Goal) : すべてのプロセス領域に共通の達成目標。「制度化」の程度を表す。例: 「管理されたプロセスを制度化する」, 「定義されたプロセスを制度化する」。

さらに、プラクティスも2種類に分類され、固有ゴールに対する「固有プラクティス」、共通ゴールに対する「共通プラクティス」、とされ、個々のゴールとプラクティスの対応関係も明確化された。

◆ CMMI への期待^{☆1}

3つのモデルが統合されたことにより、ソフトウェア開発からみると、CMMの適用範囲がシステムレベルにまで広がったことになる。組み込み系の開発、インフラ系の開発にも適用可能となった。ただし、「構成要素を組み合わせて最適解を提供する」というシステムエンジニアリングの側面が強くなった結果、ソフトウェアのみを対象とした場合は、解釈しづらいという懸念はある。また、CMMIが提供するモデルは抽象度が高く、どのような開発組織でも適用可能であるという意見がある反面、そもそもCMMIは大規模組織におけるプロセス改善を念頭に開発されたものであり、単一分野の組織や小規模組織に対する配慮が弱いという意見もある。

日本国内には、経済産業省を始めとして、CMMI（あるいはCMM）を政策的に活用したいという動きがあり、CMMI活用のための環境整備に関する調査も行われている⁴⁾。プロセス成熟度の認証や認定を期待する向きもあるが、CMMIは、「認証・レベル認定」といった仕組みを持たず、ISO9000のような継続的な審査の仕組みもない。

◆ 信頼性から脆弱性へ

◆ ソフトウェア脆弱性

ソフトウェアの脆弱性（Vulnerability）はセキュリティホールとも呼ばれる。情報セキュリティ分野の用語で、

脆弱性：システム、ネットワーク、アプリケーション、または関連するプロトコルのセキュリティを損なうような、予定外の望まないイベントにつながる可能性がある弱点の存在や、設計もしくは実装の欠陥

と定義されている⁵⁾。なお、広い語感を与える脆弱性を整理し、予定されたセキュリティ仕様を満たさないものを狭義の脆弱性とし、仕様上のセキュリティの欠如を露出（Exposure）として区別する動きもある。露出の代表例としては、fingerや、クリアテキストパスワードをネットワーク上に流してしまうtelnet等がある。また、ソフトウェアの脆弱性がソフトウェアの主要な品質特性の1つであることは、従来から広く知られている。「ソフトウェア製品の評価－品質特性およびその利用要領（JIS X0129 ISO/IEC 9126）」においても、脆弱性という表現は使われていないものの、機能性（Functionality）の一特性

としてセキュリティ（Security）が次のように定義されている。

セキュリティ：プログラムおよびデータに対して、偶発的かまたは故意にかかわらず、不当なアクセスを排除する能力をもたらしソフトウェアの属性

◆ 脆弱性のライフサイクル

Arbaughらは、脆弱性が取り得る7つの状態で構成されるライフサイクルモデルを提案している⁶⁾。7つの状態が出現する一般的な順序は次の通りである。

発生（Birth）：欠陥（flaw）が形成される。大規模プロジェクトにおいては、一般に、（故意ではなく）過失によって欠陥が作り出される。

発見（Discovery）：ソフトウェアにセキュリティ上の問題があることを誰かが発見することで、欠陥（flaw）は脆弱性となる。もし、欠陥が故意に作り出されたものであれば、「発生」と「発見」は同時に起こることになる。

発覚（Disclosure）：発見者が欠陥の詳細を広く公開することで、脆弱性は多くの人々の知るところとなる。欠陥の公開は、脆弱性に関するメーリングリスト（Bugtraqなど）への投稿というかたちで行われる場合もある。また、ソフトウェアベンダや開発者が、脆弱性に関する情報を直接受け取る場合もある。

修正（Correction）：ソフトウェアベンダや開発者が、脆弱性の原因となる欠陥の修正方法をパッチ（修正プログラム）などとして公表することで、脆弱性は修正可能な状態となる。

周知（Publicity）：インシデント対応センター（Incident response center）から、欠陥や脆弱性の詳細に関するレポートが公表されることなどにより、より多くの人々が当該脆弱性を知ることになる。「発覚」よりもはるかに多くの人々の知るところとなり、隠蔽等の情報操作は事実上不可能になる。

スクリプティング（Scripting）：脆弱性に付け込む手順がスクリプト化されることで、ソフトウェアシステムへの侵入（Intrusion）が非常に容易になる。一般に、新しく発見された脆弱性を利用してソフトウェアシステムへ侵入するためにはある程度の技術力が必要である。しかし、付け込む手順が一旦スクリプト化されると、技術力のない者でも進入が可能となり、その数（攻撃者数）は劇的に増加する。

消滅（Death）：脆弱性を利用した侵入がほとんどのソフトウェアで不可能になると、その脆弱性は消滅する。

^{☆1}「CMMIへの期待」を始めとして、CMMIに関する記述の多くは、乗松聡氏（乗松プロセス工房）にご提供いただいた情報に基づくものです。ここに深く謝意を表します。

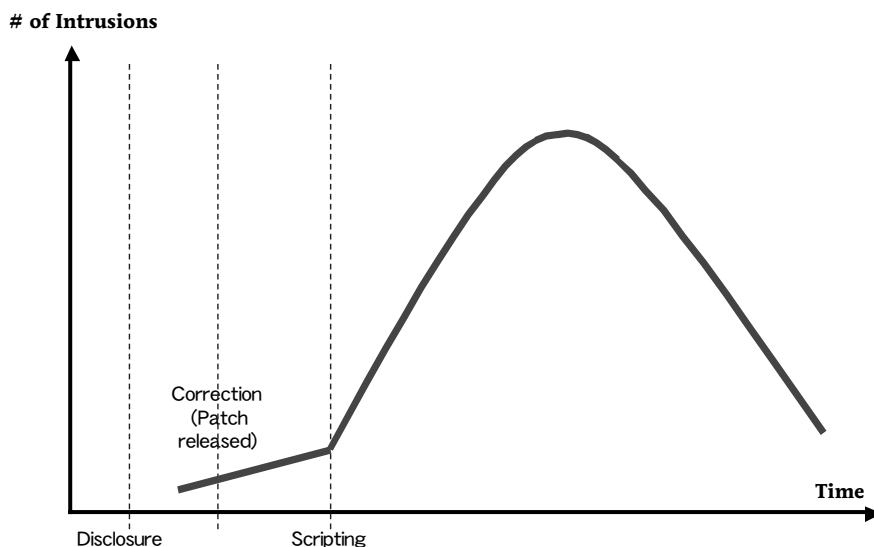


図-1 侵入件数の時間変化

◆脆弱性管理の必要性

ライフサイクルモデルの最初の3つの状態「発生」, 「発見」, 「発覚」はこの順に出現するが, 「修正」, 「周知」, 「スクリプティング」の出現順は一定ではない。ソフトウェア管理上問題となるのは, 「修正」や「周知」よりも先に「スクリプティング」が行われた場合である。スクリプティングの直後から, 脆弱性を持つソフトウェアシステムへの侵入件数は爆発的に増加する(図-1参照)⁷⁾。また, そもそも, 発覚した脆弱性に対してパッチを公開するというアプローチ(Penetrates-and-patch approach)には次のような管理上の問題点がある⁷⁾。

- 開発者は既知の脆弱性しか修正することができない。攻撃者は, 一般に, 自分たちが見つけた脆弱性を開発者に報告したりはしない。
- ソフトウェアベンダは市場からの圧力に屈するかたちでパッチを性急に公開する場合が多い。そのような場合, 結果として, 新たな脆弱性を生むことになる。
- 多くの場合, パッチは対処療法にすぎず, 脆弱性の根本原因を取り除くわけではない。
- システム管理者はパッチをシステムに当てたがらない。作業が煩雑であることもあるが, 正常に動作しているソフトウェアシステムに変更を加えることを望まない場合が多い。

結果として, ソフトウェアの脆弱性に付け込まれる被害は増える一方である。2002年に情報処理振興事業協会セキュリティセンターに届け出られた, コンピュータシステムに対する不正アクセスの件数は619件で, 2001年の届出件数550件の約1.1倍となり, 過去最多となっ

た⁸⁾。しかも, 実際に被害にあった届出を原因別分類に見ると, 「古いバージョン, パッチ未導入など」, 「設定不備」など既知の対策をとっていれば被害を未然に防げたケースが全体の約6割を占めている。別の見方をすれば, それだけ多くのシステム管理者が, ソフトウェアの脆弱性に関する情報を十分に得ておらず, また, パッチを当てることに積極的でなかった, ともいえる。

これまで, 特定分野のソフトウェアを除き, 脆弱性(あるいは, セキュリティ)の重要度は, 信頼性(Reliability)や使用性(Usability)に比べて高いものではなかったのかもしれない。しかし, オープンソースの利用や開発したソフトウェアのオープン化を前提とした場合, 開発したソフトウェアが脆弱性を持つ可能性は否定できず, また, 攻撃者によって欠陥が発見され, 侵入されるかもしれない。侵入されれば被害は甚大である。ソフトウェアの脆弱性対策について検討することは, ソフトウェア管理上, 非常に有益である。たとえば次のような対策である。

- ソフトウェアの脆弱性やそれに対するパッチに関する情報を組織的に収集する体制を整備する。インシデント対応センターの利用はもちろんであるが, 自分たちが運営管理する膨大なソフトウェアの中に, 周知された脆弱性を持つものがないかどうか, 常時チェックするための自動化ツール等を実現する。
- システム管理者が安心してパッチを当てることのできる体制を整備する。パッチ適用後のテストの自動化ツール, 必要があればパッチ適用前の状態にソフトウェアを復元するツール, 等を実現する。
- 発覚した脆弱性に対してパッチを公開するだけでなく, ソフトウェア開発時に脆弱性が混入するのを防ぐ

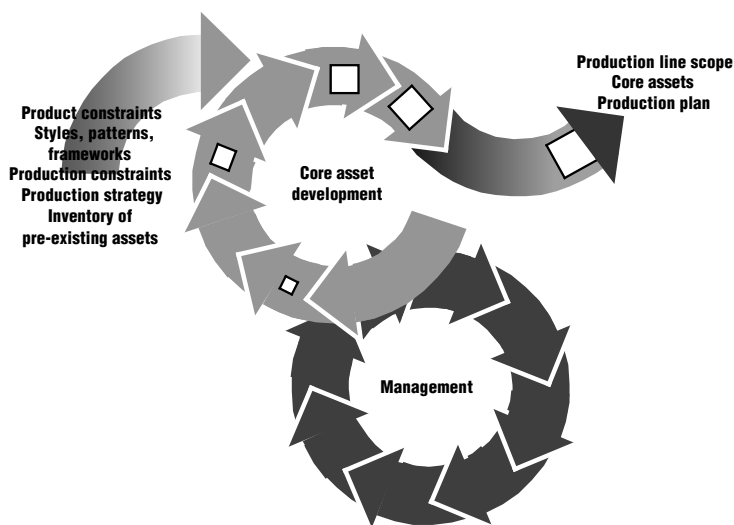


図-2 プロダクトラインにおけるコア資産開発

アプローチをとる。レビューで用いるチェックリスト、脆弱性の原因となるコードパターンを発見するツール、等を整備する。

◆ 資源から資産へ

◆ ソフトウェアプロダクトライン

ソフトウェアプロダクトラインとは、ソフトウェア資源（リソース）の再利用の一形態であり、

特定の市場分野におけるニーズを満たす特徴を共有するソフトウェアシステムの集合であり、「コア資産（Core Asset）」と呼ばれる共通集合から作り出されたもの

と定義されている⁹⁾。コア資産には、再利用可能なソフトウェアコンポーネントだけでなく、アーキテクチャ、ドメインモデル、要求、仕様、性能モデル、スケジュール、予算、テスト計画、テストケース、作業計画、プロセス記述、などが含まれる。

従来から行われてきたソフトウェア部品の再利用やカスタマイズとソフトウェアプロダクトラインが大きく異なるのは、共通性と可変性というマルチパラダイムデザインの概念¹⁰⁾を導入し、再利用の活動を、

- ドメインエンジニアリングによるコア資産開発（資産形成）
- アプリケーションエンジニアリングによる製品開発（資産運用）

に大別した点にある。特に、コア資産開発では、ソフトウェアプロダクトラインを構成するすべてのソフトウェアシステムが持つ機能や特性（共通性）と、個々の

システムが持つ固有の機能や特性（可変性）の境界点（Variation Point）の明確化が重視される¹¹⁾。そして、境界点分析に用いられるのがドメインエンジニアリングの技術である。境界点の明確化を通じて、プロダクトライン全体のアーキテクチャとスコープ（Product line scope）が決定されることになる。一方、製品開発とは、製品に求められる機能や品質を実現するために、プロダクトライン全体のアーキテクチャから製品固有のアーキテクチャを生成することである。このアーキテクチャ生成に用いられるのがアプリケーションエンジニアリングの技術である。このように、ソフトウェアプロダクトラインによるソフトウェア開発は、アーキテクチャ重視の開発である。なお、これは語感だけのことかもしれないが、再利用可能なソフトウェアコンポーネント等を部品や資源ではなく「資産」と呼ぶことで、それらをより積極的、戦略的に活用（運用）し、高い生産性（経済的価値）を実現しようという姿勢が、より明確になっているように思える。

◆ 資産の形成と運用

コア資産開発（Core asset development）の目的は、ソフトウェア製品の生産能力を実現することにある。入力には次の通りである（図-2 参照）⁹⁾。

- 製品制約（Product constraints）：プロダクトラインを構成する製品群の共通性と可変性。
- スタイル、パターン、フレームワーク（Styles, patterns, and frameworks）：アーキテクチャの構成要素。アーキテクチャの定義において、製品および製造に関する制約を満たすために設計者が適用する。
- 製造制約（Production constraints）：プロダクトラインを構成する製品に対して適用される標準や要件。

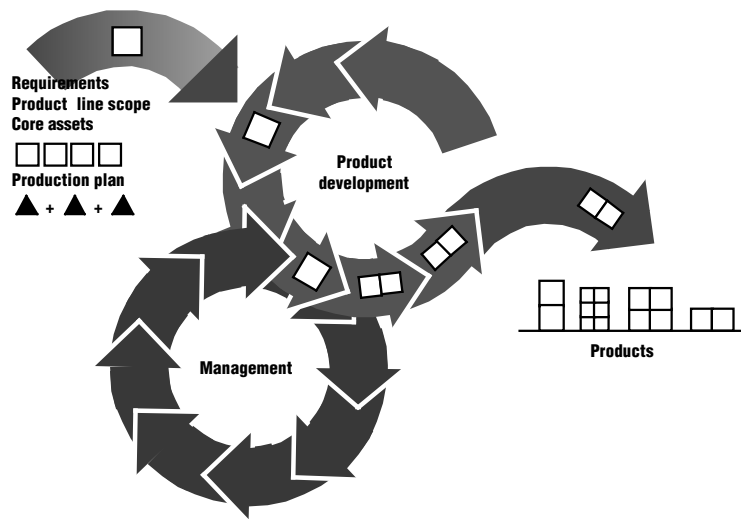


図-3 プロダクトラインにおける製品開発

- 製造戦略 (Production strategy) : コア資産を実現するアプローチ
- 既存資産 (Inventory of preexisting assets) : プロダクトラインの形成開始時に利用可能な資産

また、出力は次の通りである。

- コア資産 (Core assets) : 再利用可能なソフトウェアコンポーネント、アーキテクチャ、ドメインモデル、要求、仕様、性能モデル、スケジュール、予算、テスト計画、テストケース、作業計画、プロセス記述など。
- スコープ (Product line scope) : プロダクトラインに含まれる製品の範囲。
- 製造計画 (Production plan) : コア資産から製品を製造する具体的な手順。

コア資産開発は反復作業であり、その入力と出力は互いに影響しあうことになる。たとえば、スコープが拡大すれば、それまでは利用できなかったソフトウェアコンポーネントが、既存資産として利用可能になる場合がある。

一方、製品開発 (Product development) の目的は、コア資産を用いてユーザが要求するソフトウェア製品を実現することにある。入力は、「コア資産開発」からの3つの出力 (コア資産、スコープ、製造計画) に加えて、「個々のソフトウェア製品に対する要求 (Requirements)」であり、出力は「ソフトウェア製品」である (図-3 参照)⁹⁾。コア資産開発と同様に反復作業であり、製造されたソフトウェア製品がコア資産、スコープ、製造計画に大きな影響を与える。

◆資産管理

ソフトウェアプロダクトラインによる開発では、コア資産開発と製品開発に加え、それら活動の管理も重要な活動とされている⁹⁾。たとえば、コア資産開発と製品開発は、共に反復型作業であり、データに基づく進捗把握、工程管理は不可欠である。また、資産によって実現可能な製品が決まり、製品によって必要とされる資産が決まることから、コア資産開発と製品開発の間でのコミュニケーションパスの確保は非常に重要である。パスが確保できなければ、せっかくのコア資産が製品開発で活用されないだけでなく、製品開発では利用しづらい、利用価値の低下した資産を大量に抱えこむ危険性がある。なお、コア資産開発は、プロジェクトに跨った活動であり、必要な人的資源の確保などは、組織単位で行う必要がある。

◆今後の展望

本稿では、ソフトウェア管理技術に関するトピックスを「プロセス」、「プロダクト」、「リソース (資源)」の3つの観点から1つずつ取り上げた。最後に、ソフトウェア管理技術全般にかかわるトピックスとして「オープンソース」を取り上げ、同じく3つの観点から今後の展望を述べる。

◆プロセスのオープン化

オープンソース開発で注目すべきポイントは、ソースコードを媒体として開発者が開発プロセスを共有する点にある。共有の主な手段はメーリングリストと Concurrent Versions System (CVS) である。メーリングリ

ストでは、開発方針、実現機能、作業分担などに関する議論、報告、指示、折衝、質問が、開発者（とユーザ）間で行われる。やりとりの過程はすべて記録され、必要に応じて閲覧や検索が可能である。CVSは、ソースコードに対する変更を、変更内容の説明文とともにすべて記録する。CVSの記録も必要に応じて閲覧や検索が可能である。メーリングリストにおけるやりとりとソースコードの変更履歴を照らし合わせることで、開発者は開発プロセスをより詳細に理解し、開発作業を効率よく進めることができる。なお、オープンソース開発に必要なリソースをウェブ上に提供するサイト（SourceForgeなど）では、メーリングリスト、CVSのほかに、バグ追跡、掲示板・フォーラム、タスク管理、ホストホスティング、ファイル管理、バックアップといった機能も提供されている。また、こうした手法は、オープンソース以外の一般のソフトウェア開発でも利用され始めている。開発プロセスの共有は、開発プロセスのオープン化でもあり、プロセス管理上大きな意味を持つ。すなわち、オープンにされたソースコード上のバグが見逃されにくいと同じように、オープンにされた開発プロセス上のリスクも見逃されにくいことが期待される。特に、開発の予算規模や期間が小さい、要求仕様の変更が頻発する、定義だけの窮屈な開発は開発者に敬遠される、といったプロジェクトでは、開発プロセスを厳密に定義しそれを実行するというアプローチよりも、プロセスをオープンにすることで管理するというアプローチの方が適しているかもしれない。

◆プロダクトリファレンスの形成

ソフトウェア管理上のリスクを察知、回避する方法の1つとして、管理対象となっているプロダクト、プロセス、あるいは、資源の特性の異常値を検出、解消する方法がある。非常に単純な例としては、関数やモジュールの規模（行数）を計測し、極端に規模の大きな関数やモジュールの内容を調べ、必要があれば分割する、といった方法がある。より厳密に、また、効果的に異常値を検出、解消するためには、比較対象となる基準値（正常値、標準値）が必要となる。関数やモジュールの規模の例でいえば、開発組織全体での平均や標準偏差などが基準値となる。ただし、小さな開発組織では、基準値算出に十分なデータが得られるとは限らない。また、開発組織から収集されたデータのみで基準値を設定し異常値の排除を繰り返していくと、開発組織の特異性を助長する危険性もある。

オープンソースとして公開されているソースコードを利用すれば、組織の壁を越えたより普遍的な基準値の形成が比較的容易に実現できる。開発組織内だけで収集するよりもはるかに多数のデータが可能であり、基準値の

利用目的に応じたデータのみを取捨選択することもできる。信頼性の高い基準値を算出するため、実用規模で、多くのユーザによる利用実績のあるオープンソースのデータだけを用いることも可能である。

◆公共財としての活用

ソフトウェアプロダクトラインにおけるコア資産開発には、既存資産の利用も含まれている。本来、資産とは個人や団体が保有する財産を意味する。しかし、ソフトウェア資産の形成に限って、公共の財産（公共財）の利用も可能という解釈が許されるのであれば、オープンソースとして公開されているソースコードやその他の情報をコア資産開発に利用できることになる。なお、公共財とは、不特定多数の人によって消費・使用され、特定の人を排除しにくい財、とされている。オープンソースとして公開されているソースコードやその他の情報の多くは、公共財としての性質を備えていると考えることができる。

ただし、ソフトウェアプロダクトラインによるソフトウェア開発は、アーキテクチャ重視の開発である。アーキテクチャに合致しないソースコードやコンポーネントを資産として追加することはできない。もちろん、アーキテクチャの異なるソースコードやコンポーネントを単に集めてきただけでは資産を形成したことにはならない。また、公共財にはフリーライディング（公共財の形成に対して協力はせず、できあがった公共財を利用する行為）の問題がつきまとう。資産として活用されることを前提としたオープンソース開発の議論も必要となってくる。

参考文献

- 1) Humphrey, W. S.: *Managing the Software Process*, Addison Wesley (1989). 藤野喜一監訳: ソフトウェアプロセスの成熟度の改善, 日科技連 (1991).
- 2) 井上克郎, 松本健一, 飯田 元: *ソフトウェアプロセス*, 共立出版 (2000).
- 3) CMMI Web Site, <http://www.sei.cmu.edu/cmmi/cmmi.html>
- 4) 平成 13 年度情報技術・市場評価基盤構築事業「CMMI 活用のための環境整備に関する調査」調査報告書, 13 情経第 1504, 情報処理振興事業協会 (2002).
<http://www.ipa.go.jp/NBP/13nendo/13SPI/H13SPI-rep-index.html>
- 5) 情報セキュリティ読本 - 情報セキュリティを理解する -, 情報処理振興事業協会 (2002).
- 6) Arbaugh, W. A., Fithen, W. L. and McHugh, J.: *Windows of Vulnerability: A Case Study Analysis*, IEEE Computer, Vol.33, No.12, pp.52-59 (Dec. 2000).
- 7) McGraw, G.: *Penetrate and Patch is Bad*, IEEE Software, Vol.19, No.1, p.15 (Jan./Feb. 2002).
- 8) 情報処理振興事業協会セキュリティセンター, 2002 年不正アクセス届出状況, http://www.ipa.go.jp/security/crack_report/20030110/02all.html
- 9) Northrop, L. M.: *SEI's software product line tenets*, IEEE Software, Vol.19, No.4, pp.32-40 (July/Aug. 2002).
- 10) Coplien, J. O.: *Multi-Paradigm Design for C++*, Addison Wesley (1999). 金澤典子, 平鍋健児, 羽生田栄一 訳: マルチパラダイムデザイン, ピアソン・エデュケーション (2001).
- 11) McGregor, J.D., Northrop, L. M., Jarrad, S. and Pohl, K.: *Initiating Software Product Lines*, IEEE Software, Vol.19, No.4, pp.24-27 (July/Aug. 2002).

(平成 15 年 3 月 6 日受付)

