Vol.2019-SE-203 No.5
2019/12/12

ソフトウェア品質管理への形式的アプローチ

宮永 照二[1,†1,a)]

概要：ソフトウェアはその構成の複雑さ多様性により、本質的に品質の形式的扱いが難しい領域である。その一方で、ソフトウェアの品質にメトリクスを与え、定量的な評価を行う試みがなされている。本稿では、モジュールとモジュール間連携により構成されるソフトウェアについて、定量的な品質評価手法を提案する。

キーワード：形式的アプローチ, ソフトウェア品質管理, ソフトウェアメトリクス

Formal Approach for Software Quality Assurance

Shoji Miyanaga[1,†1,a)]

Abstract: Software is, due to its complexity and diversity, essentially hard to formally deal with. On the other hand, giving software quality metrics, quantitative evaluation attempts are being made. This paper presents quantitative quality evaluation method for software, which is composed of modules and connections among them.

Keywords: Formal Approach，Software Quality Assurance，Software Metrics

1. はじめに

昨今、ソフトウェアはあらゆるものの構成要素として組み込まれている。そのソフトウェアはあらゆる分野・領域で適用・運用されており、社会基盤の一部としてみなされている。このような状況においては、ソフトウェアの品質はサービスの品質に直結するため、高い品質が要求される。

ソフトウェアの開発においても、品質は極めて重要なファクタとされている。昨今のいずれの開発モデルにおいても、上流工程・下流工程という開発のステージを分離して運用されることが多いが、下流工程はソフトウェアの製造およびテストを実施する工程であり、品質に対しては特にセンシティブである。さらに、下流工程では不具合の 1 つも許さないような品質を目指してるものの、現実には仕様の考慮漏れや、想定されない用途による不具合の発生、また、他システムとの連携などでは、想定されないデータの状態（これも想定されない用途に含むことができるが）により、やはり不具合として観測されることがある。

特に業務系システムの開発においては、既存システム／アプリケーションとの構成や運用面での整合性、開発要員のスキルセットや成熟性などにより構成が決定されるが、多くの場合、短納期でありながら高機能かつ高使用性が要求され、そのためにアプリケーションが大規模化かつ複雑化しがちである。

このような現況において、問題解決のための様々なアプローチがなされている。そのうちの 1 つは人材育成があげられる。すなわち、高い品質のソフトウェアを提供可能な人材要員を多数育成し、その要員にアーキテクチャ設計を含む上流工程の設計を任せる。それによりソフトウェアの信頼性を担保しようとする試みである。この方法はビジネスの面でも純粋にソフトウェア品質改善の面でも有効な試みであるが、輩出可能な人材の数が限られるという課題は残る。

[1] (株) テクノネット
TechnoNet Corp.
[†1] 現在，三菱電機インフォメーションネットワーク株式会社
Presently with Mitsubishi Electric Information Network Corp.
[a)] myngshj3@gmail.com

その他では、アジャイル等の開発方法論によりプロセス改善を推進し、より迅速に要求を満たすソフトウェアを提供しつつ品質を高めるというアプローチがある。

少し、視点を変えてみてみることにする。ソフトウェア品質の定義の前に、サービスレベル合意 (Service Level Agreement: SLA) の定義が優先的に存在し、SLA を満たすソフトウェア品質が定義される。このような観点にたったとき、品質要求という観点からソフトウェアの品質仕様が見えてくる。

高度人材育成も、アジャイル方法論の適用も、作業者の直感にもとづいた問題解決方法が主であり、システマチックなプロセスとしてアルゴリズムを記述する対象として認識されていない。これが高品質なソフトウェアを恒常的に提供するための仕組みのなさ、提供の難しさであると考える。

その一方で、さまざまなプロセスに対して計算機の支援のもと、効率化がなされており、品質管理も例外ではない。

本稿では、まずソフトウェア品質管理は計算機により支援可能、すなわち、アルゴリズムを記述することにより、自動化できるプロセスが存在するという観点に立ち、ソフトウェア品質管理について形式的なアプローチを試みる。

以下、次のような構成で論じる。2 章でソフトウェア品質の関係について概説する。3 章で、モジュールとモジュール間連携で構築されるソフトウェアに対して、その品質評価指標を予測するための構造を定義し、具体例を示す。4 章でその方法論を評価する。

2. ソフトウェア品質管理

2.1 ソフトウェア品質の概観

ソフトウェアの開発者の観点では不具合の発生は極めてセンシティブな問題である。不具合がサービス停止に直結する場合、その不具合は経済的損失にも直結するため、品質問題に関してセンシティブにならざるを得ない。ISO/IEC 9126 によると、ソフトウェアの信頼性 (reliability) の指標として、障害許容性 (fault tolerance) や回復性 (recoverability) が存在するが、ソフトウェア品質を担保するためには、内部的に障害が発生してもシステム停止に陥らず、復帰が可能であるような設計が望まれることがよくある。なにか内部的に重要な障害らしき現象が観測されたときには、データベースをコミットせずにエラー画面に復帰し、ユーザに再入力を促すような設計アプローチがこの例にあたる。

ソフトウェア開発にはいくつもの意思決定要因が存在し、それは異なるステークホルダーの存在ゆえの場合もあるが、それらの制約の中で最適解としての設計を得る必要がある。そこでは、プロジェクトマネージャがリードしてプロジェクト管理を行うが、納期・工数・サービスレベルはそれぞれに依存関係が存在し、容易に最適な方法を決めることができない。その理由のひとつとして、ソフトウェアの品質指標の定量化が難しい点にあると考えられる。

しかしながら、昨今の研究では様々なソフトウェアメトリクスが提案されているが、品質標準 SQuaRE においても品質の測定について規定されており、その重要性が認識されている。さらに、プロジェクト管理の観点からは、ソフトウェアプロダクト品質がどのような状態にあるのか、可視化される必要があると考える。すなわち、現在の主な開発手法がプロジェクトマネージャーの直感的な理解に依存してマネージメントの方向性が決定されるのに対して、ソフトウェアメトリクスを算出して品質を可視化し、それをもとにプロジェクトマネージメントを計算機システムにより支援するという方法論は、プロジェクトマネージメントおよび品質管理の手続きをアルゴリズムとして記述し計算機により支援するというアプローチであり、CMMI のレベル 3 以上、すなわち反復再現可能なプロセスの定義を与えることにつながる。

2.2 ソフトウェア品質モデル

ソフトウェアの品質に関しては、様々な研究が存在する [2][4][3]。またソフトウェアの品質にはソフトウェア要求が強く関係するが、それらについての研究も存在する [7]。

品質指標の一般的な普及を考えた場合、ISO 等の標準化された品質テンプレートを利用するのが好ましい。ソフトウェア品質の規格の一つである ISO/IEC 9126 は、以下からなる。

(1) 品質モデル; quality model
(2) 外部測定法; external metrics
(3) 内部測定法; internal metrics
(4) 利用時品質測定法; quality in use metrics

ここで品質モデル (quality model) は、以下の構造を持つ。

- 機能性（functionality） - 機能とその特性に影響する特性群。機能には、必要性を明確に述べているものと、暗に示しているものがあり、次の性質に細分化される。合目的性（suitability），正確性（accuracy），相互運用性（interoperability），機密性（security），標準適合性（compliance）
- 信頼性（reliability） - ある状況がある時間続いたときにソフトウェアがどの程度機能するかに影響する特性群であり、次の性質に細分化される。成熟性（maturity），障害許容性（fault tolerance），回復性（recoverability），標準適合性（compliance）
- 使用性（usability） - 利用するのにかかる手間、個人の努力などに影響する特性群であり、次の性質に細分化される。理解性（understandability），習得性（learnability），運用性（operability），注目性 (attractiveness)，標準適合性（compliance）
- 効率性（efficiency） - ソフトウェアの性能やそれに要

するリソース量に影響する特性群であり、次の性質に細分化される。時間効率性（time behaviour），資源効率性（resource behaviour），標準適合性（compliance）

- 保守性（maintainability） - 何らかの変更を加えるのにかかる手間に影響する特性群であり、次の性質に細分化される。解析性（analyzability），変更性（changeability），安定性（stability），試験性（testability），標準適合性（compliance）
- 移植性（portability） - 別の環境にソフトウェアを移行させる可能性に影響する特性であり、次の性質に細分化される。環境適応性（adaptability），設置性（installability），共存性 (co-existence)，置換性（replaceability），標準適合性（compliance）

このように、ソフトウェア品質モデルは複雑な構造をしており、あらゆる品質要求に対してそれらを常に満足するような品質指標を与えることは困難と考えられる。たとえば、ソフトウェアの使いやすさという観点から言えば、それは使用性 (usability) に該当し、直感的に使用できる GUI を工夫することにより、理解性 (understandability)，習得性 (learnability)，注目性 (attractiveness) という性質について高い評価を与えることができるが、その反面、効率性 (efficiency) の観点からは、時間効率性 (time behavior) や資源効率性 (resource behavior) を損なうことが考えられる。また、昨今はやりの Web インタフェースでは、そのソフトウェア構成上の問題から、保守性 (maintenability) の観点において、解析性 (analyzability)，変更性 (changeablility)，安定性 (stability)，試験性 (testability) の性質を損なう場合も往々にある。そのうえ、高い品質要求は開発・保守コストとトレードオフするため、品質管理はプロジェクト管理と密接な関係がある。

2.3 ソフトウェア品質モデルと SLA の関係

導入で述べたように、ソフトウェア品質はソフトウェアが提供するサービスレベル合意 (SLA) に基づいて、SLA を満たすように定義されるべきものである。その点でいえば、そのような順序関係を厳密に満たすようにプロセス管理がなされているとは、必ずしもいえない。筆者の経験からは、下流工程の担当者は自担当のソフトウェアの品質に関しては注力するが、それがより上位工程においてなされる SLA という意思決定については、必ずしも認識しているとは限らない。

コードクローンのように、ある種の仕様を満たすと思われるモジュールのクローニングは開発の効率化に対して一面的には寄与するが、SLA というサービスレベルの要求事項を満たすソフトウェア品質要求という観点にたてば、クローンの作成にもケアが必要だろう。

2.4 ソフトウェア品質とプロジェクト管理

一般的にはソフトウェアプロジェクト管理は上流工程から下流工程までを俯瞰し、たとえば進捗などのプロジェクトの性質をモニターしながら、プロジェクトをコントロールする。多くの場合、ソフトウェアの機能面に着目し、その機能がソフトウェア要求を満たすように設計へさらに製造へブレークダウンする。

この機能面への着目は、前述したソフトウェア品質モデルにおける機能性 (functionality) に該当する。そしてサブ項目である合目的性 (suitability) や正確性 (accuracy)、他システム連携を考慮する場合などの相互運用性 (interoperability) などにより、ソフトウェア要求ひいては SLA を満たすかどうか、そしてどれほどよく満たしているかを評価することになる。

3. ソフトウェアの構成的品質指標

3.1 ソフトウェアの構成的品質指標の定義

本項では、モジュールとモジュール間連携により構築されるソフトウェアの品質は、構成的方法により品質指標を構築できるという観点に立ち、その方法を提示する (構成的方法とは、ここでは、基本的な構造とそれらの関係をもちいて、逐次積み上げることによって、総合的な指標を得るという方法を意味している)。

いま問題を単純化するために、図 1 のグラフで与えられるような構造をもつソフトウェアを考える。

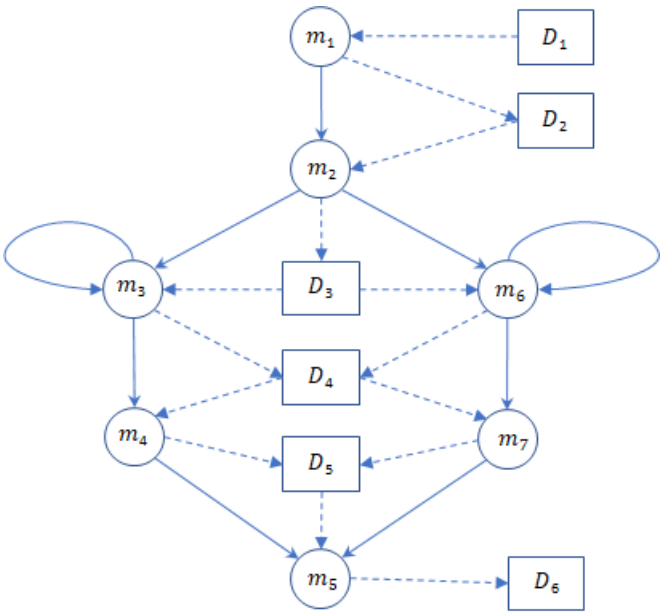

図 1 Software Model

つまり m_i を i により索引つけされたモジュールとし、モジュールが逐次呼び出されることよって総合的な機能、ひいてはサービスを実現するとする。D_i はモジュールが使用するデータ、実線の矢印はモジュール実行順序、破線

の矢印はデータの流れる向きを表す。モジュール m_i に対する障害発生率 f_i が存在するとする。一般的には、データ量 x_i に依存して f_i は増加すると考えらるので、$\frac{df_i}{dx_i} > 0$ とみなせる。その一方で、保守性を考慮したソフトウェアは、例えば逐次処理の中に変数の初期化処理を入れることにより $f_i = 0$ を強制する処理を実装することができる。

このような処理を挿入することは、ソフトウェアの信頼性を高めると同時に、運用性や正確性を高めることができる。

まず、ソフトウェアの最小構成要素である、モジュールに対応する品質構成部品として構成的品質モジュール (constructive quality module) を定義する。構成的品質モジュール CQM は以下のタプルであらわされる。

$$CQM = (Module, Reader, Writer, Indicator)$$

ここで、$Module$ はモジュール、$Reader$ は $Module$ への入力インタフェース、$Writer$ は $Module$ からの出力インタフェース、$Indicator$ を $Module$ の品質指標関数を表す。

このときソフトウェアの構成的品質モデル (constructive quality model of software) は $CQMS$ は以下の構造を持つ。

$$CQMS = (M, D, S_r, S_w)$$

ここで、$M = \{CQM\}$ を CQM の集合、 D をソフトウェアが使用するデータの集合、$S_r : M \to D$ をモジュール M が使用するデータソースを求める関数 (data source function), $S_w : M \to D$ を M が使用するデータデスティネーションを求める関数 (data destination function) とする。

このとき、あるソフトウェアを構成するモジュール $m_i \in M$ が存在して、$m_1, m_2, .., m_n$ というふうに実行されるとすれば、この実行時の品質は以下であらわされると想定される。

$$1 - \prod_i (1 - Indicator_i)$$

3.2 ソフトウェアの構成的品質指標の構成例

前章で説明したように、ソフトウェア品質モデルは複雑な構造をしており、すべてについて適切な指標を与えることは困難と考えらえる。その場合においても、システム開発者だけでなく、サービスの提供者がもっとも危惧するのは、不具合によるサービスの停止である。

そこで本項では、ソフトウェアの構成的品質指標として、障害すなわちサービス停止をもたらす不具合についてその発生率を指標として与え、その発生率の予測に寄与するソフトウェアの品質モデルに指標を与えることを、構成的な方法で表現する。

一般的にモジュール直列型のソフトウェアの場合、どのかのモジュールが障害を起こすと、ソフトウェアが完全には動作しなくなり、ソフトウェア品質モデルの機能性 (functionality) の一つである合目的性 (suitability) を満たさなくなる。

図 2 に、モジュール $m_i (1 \leq i \leq n)$ の直列接続からなるソフトウェアの構造を示す。

図 2 Simple Path

この例であらわされるソフトウェアの実行におけける障害が発生しない確率は、パス $p = m_1, .., m_n$ のノードのいずれでも発生しない確率の総積であり、その確率 s_p は以下であらわされると考えられる。

$$s_p = \prod_i (1 - \int_{X_i} f_i(x_i) dx_i)$$

したがって、障害が発生する確率 g_p は以下であらわされる。

$$g_p = 1 - s_p = 1 - \prod_i (1 - \int_{X_i} f_i(x_i) dx_i)$$

ここで、f_i はモジュール m_i の障害発生率の密度関数，x_i はモジュール呼び出し契機に該当する入力イベント，X_i はモジュール m_i が呼ばれるときのイベントの平均的な集合である。

現実にはモジュール m_i は必ずしも直列接続された単純パスの構造を持つとは限らない。図 3 に示すように複数の分岐を用いて実行されることのほうが多いだろう。

この図では、ノード m_s から分岐が発生し、そのノードから n 個のパスに分岐してたのち、ノード m_e で合流する。いま、パス $p_j = m_s, m_{j1}, m_{j2}, .., m_s \ (1 \leq j \leq n)$ を定義すると、パス p_j 中のノードのいずれかで障害が発生する確率密度関数は、

$$g_{p_j} = 1 - \prod_i (1 - \int_{X_{ji}} f_{ji}(x_{ji}) dx_{ji})$$

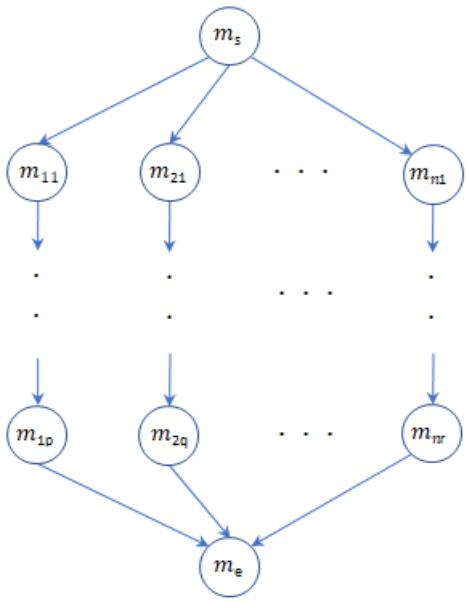

図 3　Parallel Path

いまこの分岐において、パス p_j が r_j の割合で実行されるとすると、この場合、このシステムの障害発生確率 g は以下であらわされる。

$$g = \sum_{j=1}^{n} r_j g_{p_j}$$

ここで、$\sum_j r_j = 1$ である。

3.3　事例研究

一般的に、Java/C#/Python 等のオブジェクト指向言語では、例外処理機能を実装し、想定外のアクションについては、汎用例外を検知したとして処理し、システム停止をもたらさないような配慮を行う。ただし、システム停止をもたらさないまでも、ユーザが意図する操作が実現できなくなることがある。そのため、本項では、例外捕捉機能によりシステム停止を回避する一般的な例を挙げ、さらに、例外処理時にはログを出力するという一般的な障害原因検知の方法を例に挙げ、品質指標の構築について事例研究する。

図 4 に、事例のプログラムモデルを示す。

さらに、Listing 1 に、その Java のプログラム例を示す。

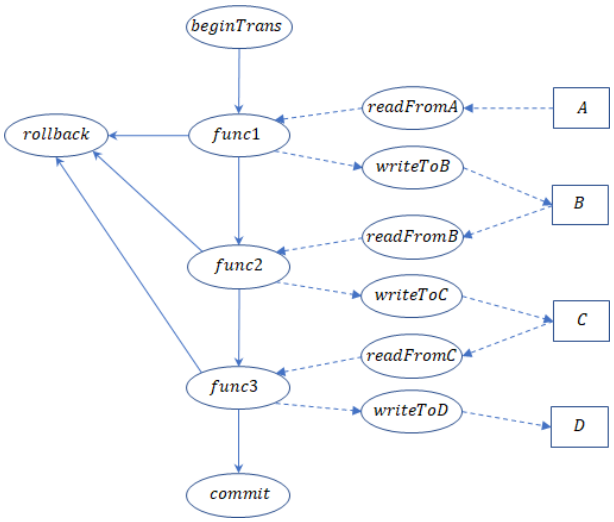

図 4　Case Study Model

Listing 1

Source

```
1  public class Foo {
2      static void main(String[] args) {
3          Foo foo = new Foo();
4          try {
5              outputLog("start\n");
6              beginTrans();
7              func1();
8              func2();
9              func3();
10             commit();
11             outputLog("finish\n");
12         } catch (Exception e) {
13                 rollback();
14                 outputLog(e);
15         }
16     }
17     public void func1() {
18         readFromA();
19         writeToB();
20     }
21     public void func2() {
22         readFromB();
23         writeToC();
24     }
25     public void func3() {
26         readFromC();
27         writeToD();
28     }
29     public void beginTrans();
30     public void commit();
31     public void rollback();
32     public void outputLog(object obj);
33 }
```

このプログラムを複数回実行し、エラーを発生履歴をトラックすることにより、Listing 2 のような出力を得ることができる。

Listing 2

Log

```
1  start
2  finish
3  start
4  finish
5  start
```

```
6  finish
7  start
8  Exception in thread "main" java.lang.
       ArrayIndexOutOfBoundException: 3
9              at Foo.func3(Foo.java:40)
10             at Foo.readFromC(Foo.java:54)
11 start
12 finish
13 ...
14 start
15 Exception in thread "main" java.lang.
       ArrayIndexOutOfBoundException: 3
16 at Foo.func3(Foo.java:40)
17 at Foo.readFromC(Foo.java:54)
18 ...
19 ...
20 start
21 Exception in thread "main" java.lang.
       ArrayIndexOutOfBoundException: 3
22 at Foo.func3(Foo.java:40)
23 at Foo.readFromC(Foo.java:54)
```

このようなログは、プログラムの実行とその際の各モジュールの実行失敗状況を報告しており、このログを解析することにより、プログラムの内部障害を統計量として算出することができ、障害に関するソフトウェアの構成的品質指標を構成することができる。

4. まとめ

4.1 議論

本項では、いくつかの基本部品を積み上げてソフトウェアを構築する際の品質指標の構築方法を示した。特に、重要な品質として信頼性の観点から、障害許容性と回復性を指標にとり、サービス停止に必ずしもつながっていないが、内部的に発生している障害に起因して発生する障害の可能性を構成的に予測する方法を示した。

その予測方法の例として、一般的にログにより採取される実行状態を分析して得られる、潜在的な障害要因をもとに、統計的な方法を用いた算出例を示した。

各モジュールのもつサービス停止が懸念される内部障害発生率を適切に算出すれば、障害がある頻度で偶発的に起こるという STAMP[1] のようなシステム理論的な考え方で構築できる。

4.2 関連研究

本稿で提案した方法では、ソフトウェアを構成するモジュールに対する品質指標をベースとして、ソフトウェアの品質指標を構築する。この点では、ソフトウェアの持つ性質とソフトウェアの品質との関係を示す研究が存在する。

それらには、機械学習の手法によりコーディングルールと不具合の共起傾向を取得する研究がある [6]。あるメトリクスからソフトウェア不具合の修正予測を行う研究がある [5][8]。また、ソフトウェア品質技術とソフトウェア品質特性の関係の可視化に関する研究がある [9]。

これらの研究にはさらに先行研究が存在し、数々のソフトウェアメトリクスがソフトウェアプロダクトとソフトウェアプロセスに対して与えらえれている。

本稿では事例研究としてソフトウェアの実行履歴から統計的にモジュールの品質指標を構築する例を示したが、これら関連研究により明らかにされたルールに基づいたメトリクスを併用し、ソフトウェア品質指標としてもちいれば、より適切なソフトウェア品質指標が構築できると考えられる。

参考文献

[1] IPA ソフトウェア高信頼化センター：はじめての STAMP/STPA システム思考に基づく新しい安全解析手法，情報処理推進機構 (2016).
[2] 福住伸一, 平沢尚毅: ソフトウェア品質における利用時品質の提案, 情報処理学会研究報告 (2019).
[3] 中島毅, 込山俊博：品質要求フレームワークの IoT システムへの適用例とその評価，情報処理学会研究報告 (2019).
[4] 込山俊博, 東基衛: システムおよびソフトウェア品質の国際的な基準の確立－日本主導の国際標準化への取組み, 情報処理学会デジタルプラクティス (2019).
[5] 野中誠, 中嶋久彰, 伊藤雅子, 山田弘隆: ランダムフォレストを用いたソフトウェア不具合修正予測におけるインパクトスケールの有効性, 情報処理学会研究報告 (2018).
[6] 田口健介, 名倉正剛, 高田眞吾: ソフトウェア変更時のコーディング規約違反と不具合の共起傾向の調査, 情報処理学会研究報告 (2018).
[7] 蛸島昭之, 青山幹雄：ステークホルダー関心事に基づくソフトウェア要求仕様書構成モデルの設計方法，情報処理学会研究報告 (2018).
[8] 野中弘隆, 山田弘隆, 中嶋久彰, 伊藤雅子：インパクトスケールを用いた不具合修正にかかわるソフトウェア変更の予測，情報処理学会研究報告 (2018).
[9] 小島嘉津江, 森田純恵, 若本雅晶, 宗像一樹, 鷲崎弘宜：ソフトウェア品質技術が品質特性に与える効果の見える化，情報処理学会研究報告 (2017).