B-035

ソフトウェア設計の保守性を評価・改善するための構造診断手法の提案
Software Diagnosis Technique to Evaluate and Improve its Maintainability

岡本 渉† 亀田 佳代子† 山下 剛† 砂田 徹也†
Wataru Okamoto Kayoko Kameda Tsuyoshi Yamashita Tetsuya Sunata

1. はじめに

多くのソフトウェア開発は, 既存機種に対する仕様変更や機能拡張など, ベースとなるソフトウェアを修正する派生開発によって占められる. 保守性に優れたソフトウェア設計であれば, 高い派生開発効率を実現できるが, その際, アドホックな設計変更を積み重ねると, ソフトウェアの構造は複雑化し, その保守性は失われていく. その結果, 派生開発効率も低下する.

ソフトウェア設計の高い保守性を維持するためには, 設計の良否を定量的に評価し, 問題箇所を検出・改善していく必要がある. 従来から, 設計を定量的に評価するための指標としては, 例えばオブジェクト指向設計であれば CK メトリクス[1]など様々なものが提案されている. しかし, 実際の製品ソフトウェアに適用してみると, 期待とは異なる評価結果となることも少なくない. この原因のひとつとして, 従来指標では個々のモジュールやクラスなどソフトウェア設計上の構成要素を同規模・同質なものと扱っている点が考えられる. 実際のソフトウェアでは, 個々のモジュール規模には極端なバラつきが見られることも多く, 設計を評価する上では, このバラつきは無視できない要因となる.

本稿では, モジュール規模のバラつきを考慮した評価指標を提案するとともに, それによって検出された設計上の問題に対する効果的な改善ポイントを絞り込むための手法を提案する.

2. 従来の評価指標とその課題

ソフトウェア設計の保守性評価においては, モジュール間の結合性と, モジュール内の凝集性の評価が中心となる.

モジュール間の結合性を評価する指標として, アーキテクチャ分析を目的とした市販ツール Lattix[2]で扱われている「影響度平均」がある. ここで, 影響度とは, ソフトウェア構造において, あるモジュールが変更された際, どれくらいの範囲にその影響が及び得るか評価する指標である. Lattix ではそれを影響が及び得るモジュールの合計数で表現している. そして, 影響度平均は各モジュールの影響度の平均値である. なお, 影響が及び得るモジュールとは, 変更されるモジュールに直接・間接に関わらず依存しているモジュール群を指す. 影響度平均が高い, すなわち, 設計変更時の影響範囲が平均的に広いソフトウェア構造ほど, 広範囲に渡るリグレッションテストが必要になるなど, 多くの派生開発工数を必要とし, 保守性は低いと評価される.

一方, モジュール内の凝集性を評価する指標としては, CK メトリクスのひとつである LCOM* (Lack of Cohesion in Methods: メソッド間の非凝集度)が挙げられる. これは, クラス内のメンバ変数 1 つあたりにアクセスするメソッドの数の平均割合を示した数値である(正確には, 1 からその値を引いた値). メンバ変数を介して, より多くのメソッドが相互に関連しているほど, LCOM*値は低く, 凝集度は高いと評価される.

これらの指標を実際のソフトウェアに適用する際, 課題となるのは, これらの指標が「各モジュールの規模(もしくは各メソッドの規模)が同程度に揃っていること」を前提としている点である. モジュール規模やメソッド規模に極端なバラつきがあると, 上述した指標では適切に保守性を評価できない可能性がある. そして, 実際のソフトウェアでは, モジュール規模, メソッド規模に極端なバラつきがあることが少なくない.

† (株)東芝 ソフトウェア技術センター,
TOSHIBA Corporate Software Engineering Center

3. 保守性評価指標

3.1 影響度

本稿で提案する「影響度」は, 概念としては前節で紹介した Lattix の影響度と同一である. すなわち, ソフトウェア構造に対して, 設計変更があった際, 影響が及び得る範囲の広さによって保守性の良否を評価する. ただし, モジュール規模のバラつきを加味して評価する点に特徴がある.

モジュール規模にバラつきがあるとき, 影響範囲をモジュール数で評価すると, 以下のような問題が生じる. 図 1 のような状況を考えてみると, 構造 1 では, モジュール X が影響を及ぼすモジュールの数は 3, 構造 2 では 1 となる. したがって, 構造 2 の方が, 影響範囲が狭く保守性に優れた設計であるとの評価結果が導かれる. しかし, 実は, モジュール X が影響を及ぼすソースコードの総量は構造 1 でも構造 2 でも同じであり, 設計変更時に行われるソースコード調査やリグレッションテストの量に変わりはない. したがって, 保守性という観点からは構造 1 と構造 2 は等価と評価した方が妥当である.

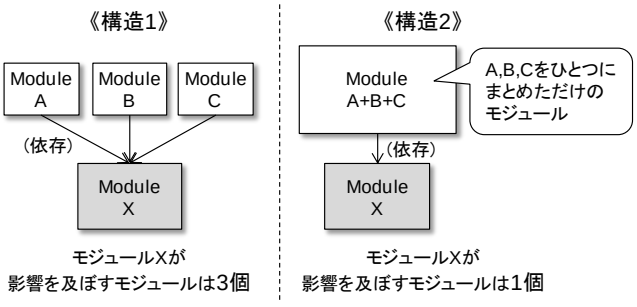

図 1 モジュール「数」で計測する影響範囲

さらに, モジュール規模の違いに起因した, 各モジュールの設計変更の起こりやすさの違いも考慮すべきである. 従来指標に見られる, モジュール数で割って影響度の平均値を算出するという処理は, どのモジュールも等確率で変更が発生することを前提としている. しかし, 単純に, ソースコード一行あたりの変更確率が等しいと仮定すると, モジュールとしての変更確率は, モジュールの規模に比例する. すなわち, 設計変更が発生するのであれば, それは巨大モジ

ュールを対象とする確率が高く，極小モジュールが対象となる確率は低い．したがって，あるソフトウェア構造における「平均的な影響範囲」としては，モジュール数で割った平均値ではなく，モジュール規模で重みを付けた平均値とした方が妥当と考える．(※モジュール規模が揃っていれば，モジュール数で割っても，モジュール規模で重みを付けても同一の結果となる．)

なお，ここではモジュール規模をもとに変更確率を考えているが，他の設計情報や過去の変更履歴を利用できる場合は，それらに基づいて変更確率を導出してもよい．その意味で，より本質的なのは「モジュールの変更確率を重みとして影響度を算出すべき」という点であり，モジュール規模はその一材料として使っているにすぎない．

上記の点を踏まえて，我々は次の 2 点を算出式に組み込み，影響度を定義することとした．

- 設計変更時に影響を与えるモジュールの「数」ではなく，影響を与える「ソースコード行数」で評価する．
- 各モジュールに対する設計変更の発生しやすさ，変更確率を重みとして加味する．

具体的には以下の手順で算出する．

1. 各モジュールの影響範囲を算出する: モジュール間の依存関係に基づいて，各モジュールが影響を及ぼす「ソースコード行数」を算出する．これは，各モジュールに直接・間接に関わらず依存しているモジュール群の総行数と，当該モジュールの行数の和である．
2. 各モジュールの影響範囲を，モジュールごとの相対的な変更確率で加重平均する: ここで「モジュールごとの相対的な変更確率」とは，全ソフトウェア中の任意の一箇所に変更があるとしたとき，その変更が当該モジュールに含まれる確率を意味する．変更確率は，当該モジュールの規模に比例するものとして扱う．

定式的には以下となる．

$$\text{影響度} = \sum_{m \in S} \left(\left(\sum_{n \in Dep(m)} LoC(n) \right) \cdot \frac{LoC(m)}{LoC(S)} \right) \Big/ LoC(S)$$

- S : ソフトウェアを構成する全モジュールから成る集合
- Dep(x) : 直接・間接に関わらず，モジュール x に依存している全モジュールから成る集合．モジュール x も含む．
- LoC(x) : モジュール(集合) x のソースコード行数．

図 2 には影響度の算出例を示した．

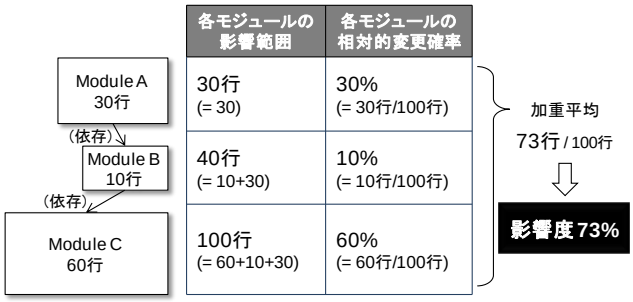

図 2 影響度算出例

3.2 凝集度

2 節において，モジュールの凝集度を計測する従来指標として例示した LCOM*についても，前節の影響度と同様，構成要素の規模(この場合，クラス内の各メソッドの規模) にバラつきがあると，妥当な保守性の評価とならない場合がある．

例えば，図 3 に示す 2 つのクラス C1, C2 に対しては，ともに LCOM*は 0.5 であるが，クラス C2 はそのほとんどが Ma, Mb という相互に関連のあるメソッドで占められており，保守性の観点からは，凝集性は C1 よりも高いと考えることができる．

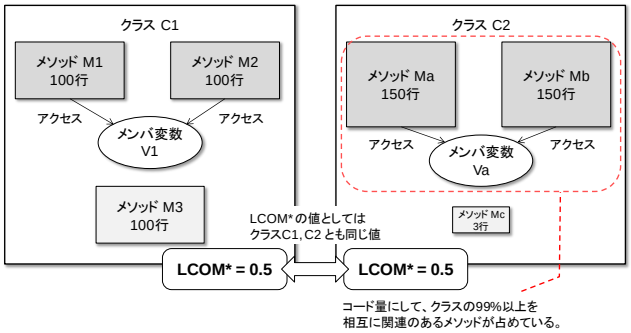

図 3 メソッドの規模と凝集度

そもそもモジュールの凝集性の低さが保守性に及ぼす影響は次のように考えることができる．モジュールを変更する際には，モジュール全体を解読したり，変更後にはリグレッションテストを行ったりするが，変更箇所に関連のない要素がモジュール内に多く存在すると，それだけ無駄な解読作業やテスト作業を行うことになる．すなわち，モジュールの凝集性の低さは，保守作業における無駄な作業の割合を示すものと言える．図 3 のクラス C2 では，そのような無駄な作業が発生する割合は低く，その意味で凝集性は高いと評価できると考える．

このような考え方に沿って，我々は凝集度を「モジュール内の任意の 2 箇所が相互に関連する確率」として定義した．ここで，「モジュール内の箇所」とは，メソッドや関数の粒度ではなく，ソースコード一行を想定している．この定義により，モジュール内のどこか一ヶ所に変更を加えたとき，発生し得る無駄な作業量を示す値として凝集度を扱うことができる．

具体的には以下の式に従って凝集度を算出する．

$$\text{凝集度} = \sum_{f1 \in F} \sum_{f2 \in F} Rel(f1, f2) \cdot SizeRatio(f1) \cdot SizeRatio(f2)$$

・F　：　モジュール内のすべての機能から成る集合(なお，簡便のため，「機能」=「関数 or メソッド」とする)
・Rel(f1,f2)　：　機能 f1 と f2 の間に関連があれば 1，なければ 0
・SizeRatio(fx)　：　機能 fx のソースコード行数の，全ソースコード行数に占める割合

「機能間の関連」とは明示的には定義していないが，共通の変数にアクセスしている，機能間で関数の呼び出し関係がある，といった状況を指す．

この算出式は，図 4 に示した表をイメージするとわかりやすい．この表は，「任意の 2 箇所」が含まれるメソッドを行・列それぞれに取っている．相互に関連を持つ組合せは，(Ma,Ma), (Ma,Mb), (Mb,Ma), (Mb,Mb), (Mc,Mc) の 5 つであり，それぞれの組合せの発生確率の和を凝集度としている．

		メソッドMa	メソッドMb	メソッドMc
	SizeRatio	0.495	0.495	0.01
メソッドMa	0.495	1×0.495×0.495	1×0.495×0.495	0×0.495×0.01
メソッドMb	0.495	1×0.495×0.495	1×0.495×0.495	0×0.495×0.01
メソッドMb	0.01	0×0.01×0.495	0×0.01×0.495	1×0.01×0.01

合計: 0.9801

図 4 凝集度算出方法

このように凝集度を定義すると，図 3 に示した 2 つのクラスの凝集度はそれぞれ C1=0.56, C2=0.98 となり，クラス C2 の方が高い凝集度となる．

3.3 モジュール規模バランス

これまで述べてきたように，実際のソフトウェアではモジュール規模や関数の規模に大きなバラつきがあり，そのバラつきのために従来の指標では適切に保守性を評価できないケースがあった．本節で述べる「モジュール規模バランス」は，そのモジュール規模のバラつきを数値化するものである．

我々はモジュール規模のバラつきや偏りを表現するために，富の偏在性などを表現するために用いられる「ジニ係数」を利用することとした．これは，「中・小規模のモジュールが大多数を占め，一部巨大モジュールが存在する」といった，現実によく見られるパターンを表現するのに適していると考えたためである．標準偏差では，大多数を占める中・小規模のモジュールの特性が強調され，巨大モジュールの存在が隠れがちになってしまう．

ジニ係数は近似的に以下の式に従って算出することができる．この値は 0 から 1 の間となり，0 だとすべてのモジュールが完全に同規模である状態，1 だと全ソースコードが 1 つのファイルにのみ占められている状態を表す．

$$\text{モジュール規模バランス} = 1 - 2 \cdot \sum_{i=1}^{N} \sum_{j=1}^{i} \left(\text{SizeRatio}(m_{j-1}) + \text{SizeRatio}(m_j) \right) \cdot \frac{1}{2N}$$

・N　：　モジュール数
・SizeRatio(m)　：　モジュール m のソースコード行数の全ソースコード行数に占める割合

※ j < i ならば SizeRatio(m_j) ≦ SizeRatio(m_i) を満たすものとする．ただし，SizeRatio(m_0) = 0 とする．

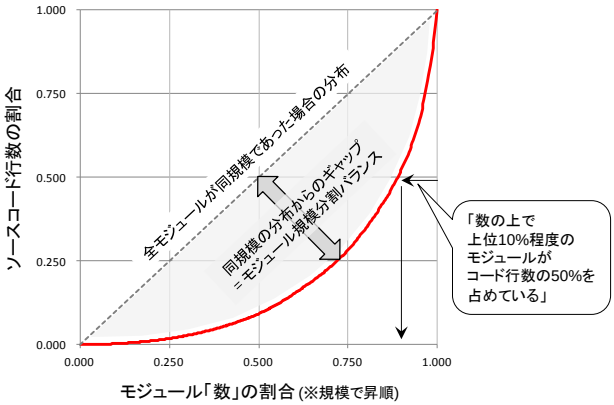

図 5 モジュール規模分割バランス

各モジュールの分布の様子は，図 5 のように表現される．例えば，巨大モジュールの上位 10%によって，全ソースコード行数の半分が占められている，といった見方をする．

4. 指標計測結果に基づく設計改善手法

本節では，3.1 節で述べた「影響度」によって評価される「モジュール間の結合性」に問題があった場合，それを改善するための手法について述べる．

影響度が高くなる原因としては，モジュール間の結合が密である，循環依存があるなど，様々なものが考えられるが，その中には，少しの要因が大きな悪影響を及ぼしている場合もある．例えば，1 本の依存関係のために，全モジュールが循環依存の関係になってしまうケースなどがそれにあたる．見方を変えると，そのような問題は少量の修正で大きな改善効果をもたらせる箇所であり，このような箇所をピックアップするのが，本節で述べる設計改善手法である．

ここでは，次の 3 つの観点から改善対象箇所を絞り込む．

- 多くの依存関係を生みだすグローバル変数の検出
- モジュール間の双方向依存における「逆流依存」の検出
- 影響度改善効果-コストグラフに基づく改善対象モジュールのピックアップ

4.1 多くの依存関係を生みだすグローバル変数の検出

複数のモジュールの間に直接的な関連はなくても，グローバル変数を介して，暗黙的に依存関係が生じている場合がある．図 6 に示した例では，モジュール A と B の間に直接的な関連はないが，モジュール A は，B が任意に書き込んだグローバル変数の値を参照しているため，「B に依存している」と言える．同様に，モジュール B も A に依存している．

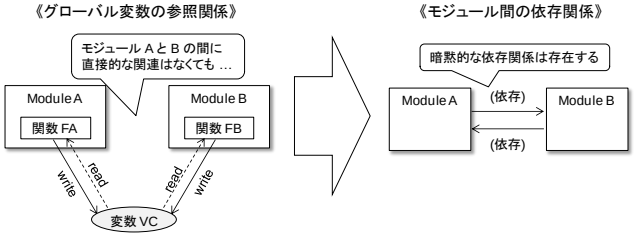

図 6 グローバル変数を介した暗黙の依存関係

このようにグローバル変数が介在すると，モジュール間に直接の関連はなくても依存関係は発生するが，特に，多く

のモジュールから書き込まれ，多くのモジュールから読み出されるグローバル変数は，暗黙的な依存関係を多く作り出す．そのようなグローバル変数をピックアップし，見直すことができれば，依存関係の効果的な改善に繋がる．

本手法では，対象ソフトウェア内に存在する各グローバル変数に対して，それに書き込んでいるモジュール数，および，その値を読み出しているモジュール数を取得し，その積が大きいグローバル変数をピックアップしている．

ここでピックアップされたグローバル変数に対して，例えば，アクセッサ関数を定義し，書き込まれる値の妥当性を保証する等の改善ができれば，暗黙的な依存関係は大きく削除できる．

4.2 双方向依存における逆流依存の検出

次に，モジュール間の双方向依存に着目する．2 つのモジュール A, B の間に双方向の関数呼び出しがあるとき，A から B への関数呼出しの数に比べて，B から A への関数呼出しの数が極端に少ない場合，後者の関数呼出しは，本来意図していた設計から逸脱している可能性がある．すなわち，本来の設計意図としては，A から B への，一方向の依存であったにも関わらず，実装やデバッグの都合により，止むを得ず B から A への関数呼び出しを行っている可能性がある．

このように，本来の設計意図からのわずかな逸脱のために双方向依存となっている箇所をピックアップし，それを解消することで，影響度を低減させるのが本手法の狙いである．

図 7 は，あるソフトウェア構造に対して，アンバランスな依存比率となっている箇所を強調表示した例である．実際のソフトウェアにこのような箇所が存在することが確認される．

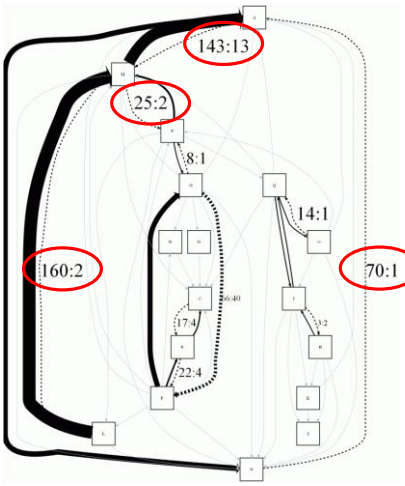

図 7 双方向依存における逆流依存

4.3 改善効果-コストグラフに基づく改善対象モジュールの抽出

ここで述べるのは，どのモジュールに関する依存関係を修正することが効果的な影響度の低下に繋がるかを探るための手法である．ここでは，各モジュールからの依存関係を削除したと仮定すると，影響度はどれほど改善するかをシミュレートする(実際には指標計測ツールの機能を用いる)．一方，実際に依存関係を削除するためには，何本かの関数呼び出しやグローバル変数参照を削除したり移動したりする必要がある．各モジュールについて，影響度の改善効果と，そのための作業量とを併せて検討することによって，低コストで大きな改善効果をもたらすモジュールを絞り込むことができる．これは，コストワース分析(Cost Worth Analysis)と呼ばれる手法であり，本手法はそれをモジュール間の依存関係の改善に応用したものとなっている．

図 8 は，ある実例での適用結果である．横軸に影響度の改善効果，縦軸に各モジュールからの関数呼出し，および，グローバル変数参照の本数をプロットしている．右下に行くほど，効果的な改善対象となる．この例では，右下にあるモジュール A について，3 本の依存関係を削除するだけで，影響度は 30%近く低減できることが分かり，効果的な改善箇所を絞り込めることが分かる．

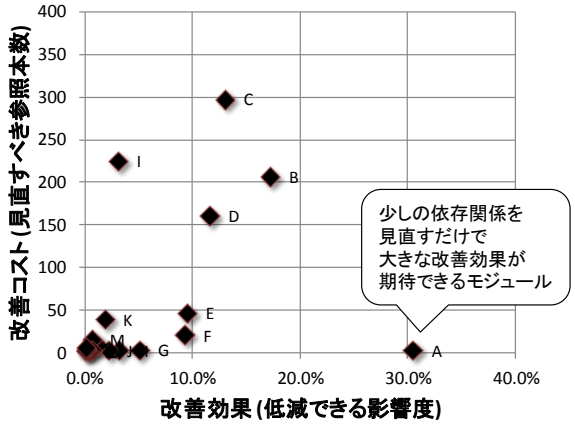

図 8 影響度改善効果-コストグラフ

4.4 適用事例

ソースコード行数約 20 万行の組込みソフトウェアに対して，指標計測および計測結果に基づく改善手法を適用した．

当初のソフトウェア構造では，影響度が 99%となっており，実質的にどこを変更してもソフトウェアのすべてに影響が波及する構造となっていた．それに対して，4.2 節，4.3 節の改善手法を適用し，約 2000 本の関数呼出しの中から 19 本の関数呼び出しを改善対象候補としてピックアップできた．仮にこれらの関数呼び出しを削除できたとすると，影響度は 66%まで低下させることができることを確認している．

5. まとめ

本稿では，実際のソフトウェアにて散見される「モジュール規模のバラつき」を考慮した，ソフトウェア構造の保守性評価指標について提案した．また，その計測結果に基づき，モジュール間の依存関係を効果的に改善するための手法について提案し，実例での適用結果について述べた．

参考文献

[1] Chidamber, S.R., Kemerer, C.F., “A Metrics Suite for Object-Oriented Design”, IEEE Trans. Software Engineering, vol.20, no.6 (1994)
[2] Lattix, http://www.techmatrix.co.jp/quality/lattix/index.html