メトリクス利用によるリファクタリング対象の自動抽出

Candidate refactoring point filter uses metrics

小林　光一

kouichi.kobayashi@rolanddg.co.jp

ローランドディー.ジー.株式会社　第 4 開発部 SC02

発表要旨：

本発表の目的は、メソッドのインターフェースの複雑度に着目しメトリクスを利用にすることでレガシーコードからリファクタリング対象を自動抽出することである。

市場で不具合が発生した時、修正箇所は正常に動作するようにするが、将来のことを考えるとメンテナンス性を向上させたいと思うことが現場では多々ある。その結果リファクタリングしようという考えに至るのだが、修正対象のコードがレガシーコードであるため、どこから手をつけて良いものかわからないという壁にぶつかる。現場においては人的リソースや時間が限定される。そのような状況下では効果的なリファクタリング対象を素早く検出する必要がある。そこでメトリクスを使ってリファクタリング対象を自動抽出する仕組みを考案した。

本発表ではその方法と実施結果を紹介する。

キーワード：

メトリクス、リファクタリング、レガシーコード、コード静的解析、コーディング規約、レビュー

想定している聴衆

ソフトウェア開発に従事していて、ソースコードのリファクタリングに興味を持っている方

発表者の紹介（全角１００文字）：

ローランドディー.ジー.株式会社にて幾何学図形作成ソフトウェア開発に従事。主に、アーキテクチャの策定や共通コンポーネントやフレームワークの開発を担当。

社外活動は TEF 東海に所属。メトリクス勉強会に積極的に参加する。設計から実装までが主な業務のため、メトリクスを活用しての実装工程での品質改善に取り組む。

* 副題は不要であれば行ごと削除してください

メトリクス利用による
リファクタリング対象の自動抽出

ローランドディー.ジー.株式会社

第4開発部SC02

〇小林　光一

e-mail：kouichi.kobayashi@rolanddg.co.jp

概要

- 市場で不具合が発生にした時、修正箇所は正常に動作するようにしたけど将来のことを考えるとメンテナンス性を向上させたいと考えた
- リファクタリングを実施して改善しようと考えた
- レガシーコードなのでどこから手をつけて良いものかわからない
- メトリクスを使ってリファクタリング対象を自動抽出する仕組みを考えた
- その結果、メソッドのインターフェースの複雑度限定で手がかりを抽出することができた

言葉の定義

- リファクタリング
  - 外部から見たときの振る舞いを保ちつつ、理解や修正が簡単になるようにソフトウェアの内部構造を変化させること
- リファクタリング・カタログ
  - 「リファクタリング」の第6章以降に記載されている改善するためのテクニックのリスト(「リファクタリング―プログラムの体質改善テクニック」からの引用)
- リファクタリング対象
  - リファクタリングの効果の大きい場所、コード、メソッド
- 手がかり
  - 人手で確認することができる程度にしぼりこまれたリファクタリング対象の候補
- レガシーコード
  - 自動テストのないコードのこと(「レガシーコード改善ガイド」からの引用)

言葉の定義

- メトリクス
  - ソフトウェアを定量的に測定すること
- メソッドのインターフェースの複雑度
  - メソッドに入力される情報量の多さ
- 組込み型
  - プログラミング言語にあらかじめ用意されている変数の型
- ユーザー定義型
  - ユーザーが任意で定義した構造体やクラス
- 引数ポイント
  - メソッドの引数に型に応じて重みをつけて換算したもの
- Doxygen
  - C++、C、Java、Objective-C、Python、IDL 、Fortran、VHDL、PHP、C# 向けのドキュメンテーション・システム

言葉の定義

- JQuery
  - ウェブブラウザ用のJavaScriptコードをより容易に記述できるようにするために設計された軽量なJavaScriptライブラリ
- LOC
  - Line of Code、コード行数（※1）
- WMC
  - Weighted Method per Class 、クラスごとの重みづけされたメソッド数、この値が大きい場合、クラスが大きすぎるので分解する必要がある（※1）
- CBO
  - Coupling Between Object Classes、クラス間の結合度（※1）

※1 「演習で学ぶソフトウェアメトリクスの基礎」からの引用

状況の整理

- レガシーコードをリファクタリングする時
  - 市場で不具合が発生し、修正するとき
  - レガシーコードを使って派生開発をするとき
- リファクタリングする時困ること
  - レガシーコードなので、動作を担保するテストが存在しない、規模が大きい
  - リファクタリング以前にテストを準備する必要がある
  - 既に規模が大きくなってしまっていて，どこから手を付けていいものやら途方にくれる
- やりたいこと
  - リファクタリング対象を見つけたいけど既に規模が大きいため人手では無理なので、コード解析して手がかりを見つけたい

コード解析

- コードを解析してどんなメトリクスを計測すれば良いか？
  - LOC,サイクロマティック複雑度、WMC、DIT、CBOなどある
  - より効果的で、簡単なものが良いので引数の数をメトリクスとして計測する
- なぜ、引数の数なのか？
  - 引数の数が多いメソッドから得られる印象は「複雑」、「煩雑」というような印象がある
  - リファクタリング・カタログにおいて引数の数に注目したテクニックがある（引数の削除、引数オブジェクトの導入など）
  - 引数の数を計測＝メソッドのインターフェースの複雑度を計測
  - 将来的に、自組織で展開していきたいので導入は簡単なもので

やること

- コード解析して手がかりを見つけるために引数の数を使ってメトリクスする
- 引数の数を計測するツールを準備する
  - 引数の数を計測するツール
  - 余計な情報を出力しない手がかりだけを表示するツール
  - DoxygenとJqueryを使った自作ツールにて実現する
- 手がかりを見つけるツールとして全体を自動化したシンプルなツールでないと自分の組織で使ってもらえないのでそのようなツールを準備する

doxygen

- ドキュメントジェネレーターである doxygen にソースコードの構文解析を担わせた。構文解析結果はXML で出力させる。これは、次工程で JQuery を使ってフィルターをかけやすいように配慮したためだ。図1のクラスを doxygen で解析すると図 2 のような XML ファイルを得た。

```
class hoge
{
public:
    hoge() : member(0){;}
    ~hoge(){;}
    void foo(){;}
    bool foo(int value){ return true;}
private:
    void bar(int value, int compare){;}
private:
    int member;
};
```

図 1: hoge クラス

```
<memberdef kind="function" >
  <type>void</type>
  <definition>void hoge::bar</definition>
  <argsstring>(int value, int compare)</argsstring>
  <name>bar</name>
  <param>
    <type>int</type>
    <declname>value</declname>
  </param>
  <param>
    <type>int</type>
    <declname>compare</declname>
  </param>
</memberdef>
```

図 2: classhoge.xml(hoge::bar メソッドのみ抜粋)

Jqueryを使った自作ツール

- JQuery を使って、doxygen で出力されたすべての XML ファイルに対してフィルタリングを行い、フィルタリングの結果を一つにまとめる。JQuery はJavaScript ライブラリである。対象となるファイルの一覧を「filelist.xml」（次ページ）として用意する。「filelist.xml」からファイル名を取得したら、先ほどXML内の<param>タグをカウントする。

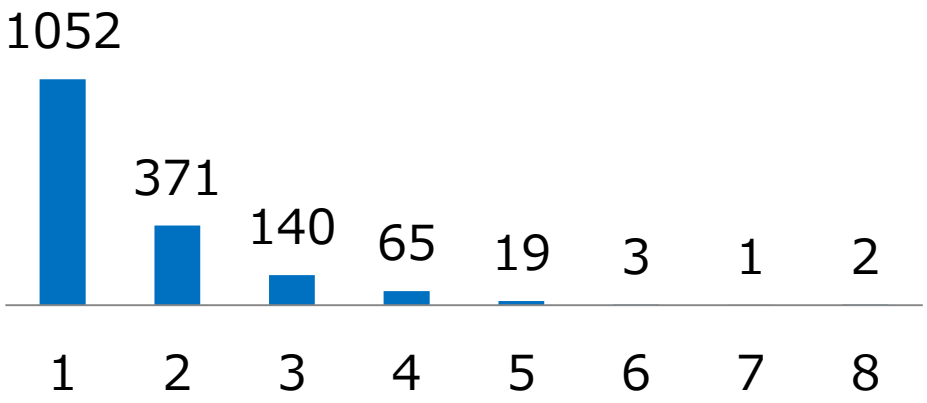
```
<list>
<name>class_c_bold_cross_pt.xml</name>
<name>class_c_bold_le_property.xml</name>
<name>class_c_bold_outline.xml</name>
<name>class_c_bold_polyline.xml</name>
<name>class_c_bold_poly_list.xml</name>
<list>
```

ファイル名（行番号）	関数名	引数の数
../aaa.h(86)	aaa::aaa	5
../b.h(27)	bool b::Copy	5
../c.h(57)	void c::chander	4

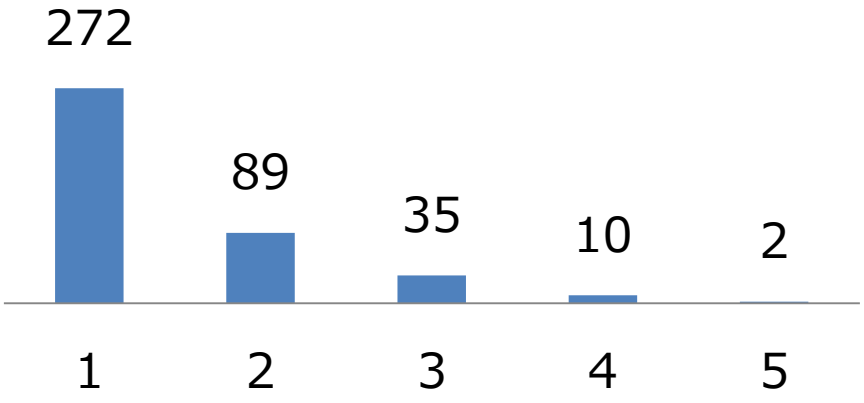
引数の数に注目して実験

- 仮説
  - 引数の多いメソッドの割合が多く、かつそれらのメソッドはリファクタリングの可能性がある
- 実験対象（言語：C＋＋）
  - PJ1 ： アプリケーションソフト
  - PJ2 ： 組込みソフト（オープンソース）

PJ1：n個の引数を持つメソッド数



PJ２：n個の引数を持つメソッド数



引数の数に注目して実験

- 結論
  - 計測結果からは、引数の多いメソッドの全体数に占める割合は、0.1～0.3%程度であった。「引数の数」を単純に分類しただけではリファクタリング対象の絞り込みはできない
- 考察
  - 「引数の数」と「引数の型」と「リファクタリングの可能性」の関係を単純化しまとめた

引数の型	引数の数:少	引数の数:多
組込み型	リファクタリング可能性:C	リファクタリング可能性:B
ユーザー定義型	リファクタリング可能性:B	リファクタリング可能性:A

  - 「リファクタリングの可能性：B」までは抽出できるようにしたい
  - 「ユーザー定義型」の引数に重みをつける。

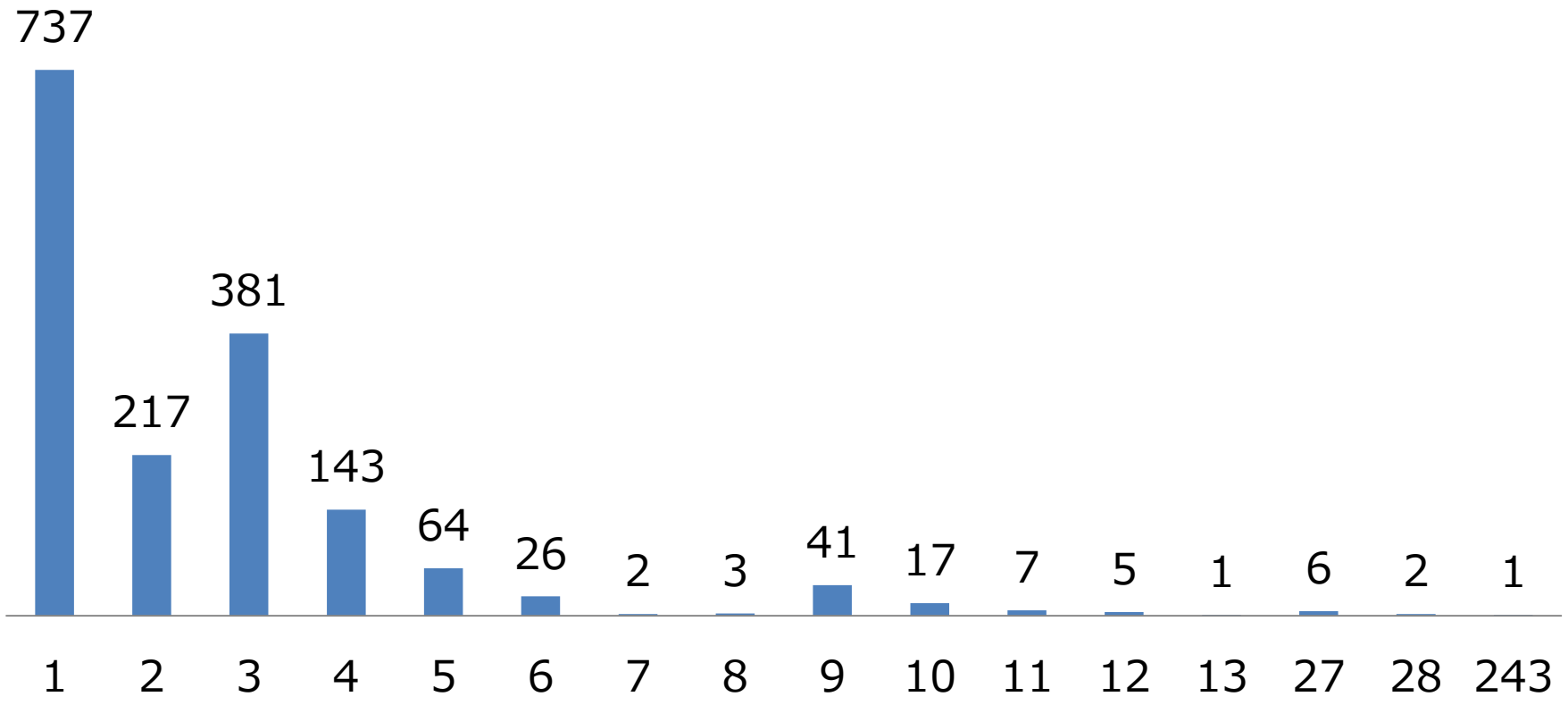
重みをつけて抽出

- 仮説
  - メトリクスを「引数の数」、「引数の型」にすれば、「リファクタリング可能性：A、B」を抽出することができる
- 重みつけ
  - ユーザー定義型ならば一様に重みを3とした。ユーザー定義型に内包されている変数の型について厳密な解析を行えば、ユーザー定義型の重みが分かるかもしれないが、実装が複雑になりそうだったので行わない。代替案としてユーザー定義同士は乗算することにした。
- 例）
  - ユーザー定義型の引数が5個あった場合

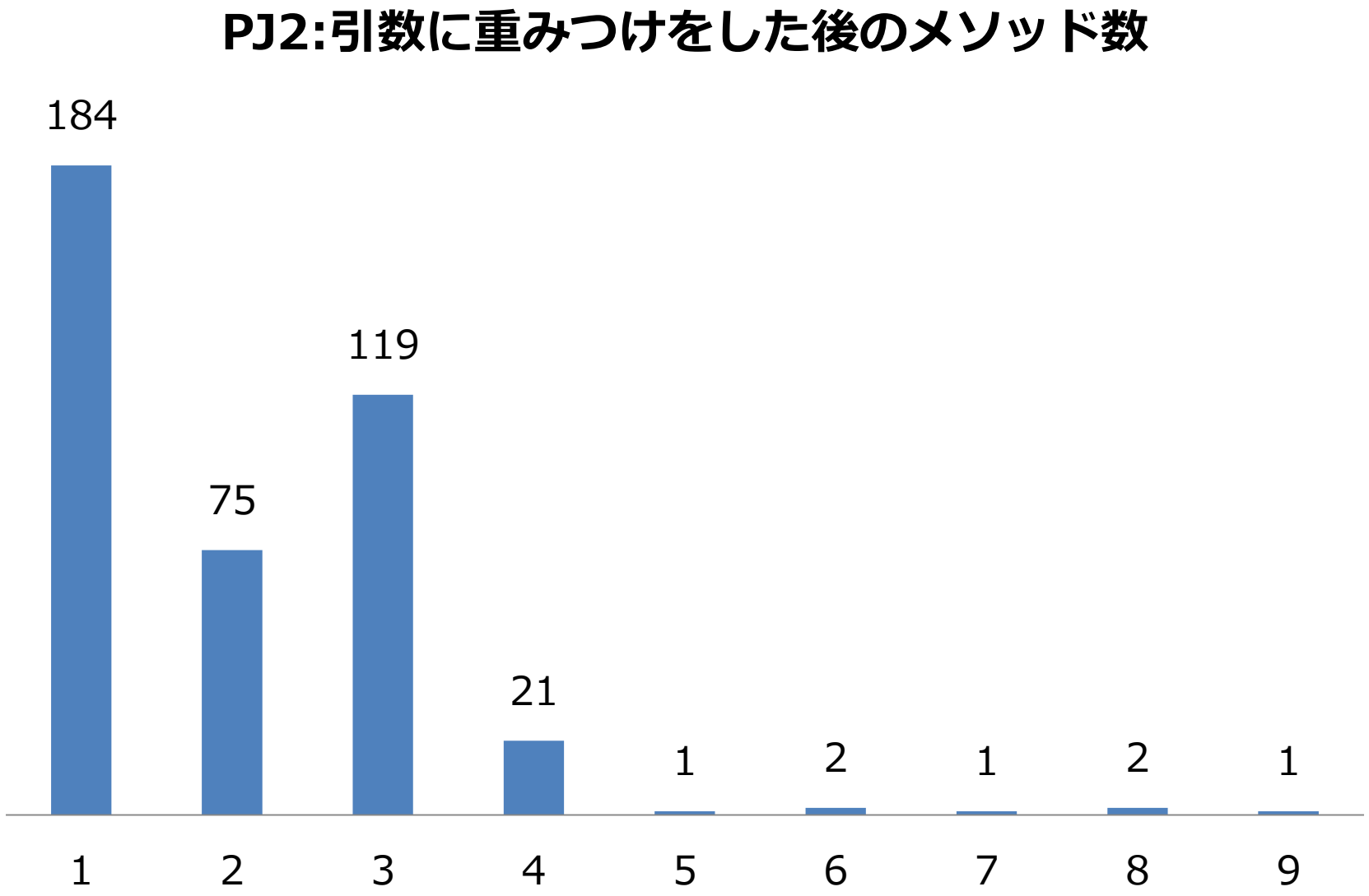
    3X3X3X3X3=243
  - ユーザー定義型2個と組込み型が3個あった場合には

    （3X3）+1+1+1=12

重みをつけて抽出


PJ１：引数に重みつけした後のメソッド数
737
217
381
143
64
26
2
3
41
17
7
5
1
6
2
1
1
2
3
4
5
6
7
8
9
10
11
12
13
27
28
243

重みをつけて抽出



重みをつけて抽出

- リファクタリングの検討
  - 引数ポイント3以上のものを対象にリファクタリングを検討
  - リファクタリング・カタログを摘要する
  - 引数ポイント1～2は細分化できていないと判断し、対象外にする
- プロジェクト1
  - 引数ポイント3以上は699

	メソッド数	理由
必要	231	問い合わせと更新の分離、引数の削除
不要	468	API

- プロジェクト2
  - 引数ポイント3以上は147

	メソッド数	理由
必要	10	問い合わせと更新の分離
不要	137	コンストラクタ

重みをつけて抽出

- プロジェクト1結果
  - 「引数の数」だけよりも細分化できた
  - 引数ポイント1～2のメソッドが多い
  - 問い合わせと更新の分離を検討し引数ポイント5以上はほぼリファクタリング可能性あり
- プロジェクト2結果
  - 引数ポイント5以上はコンストラクタが多いが問い合わせと更新の分離を検討した結果、コンストラクタ以外はリファクタリング可能性あり
- 考察
  - 引数ポイント1～2については未細分化
  - 引数ポイント3～4は組込み型のみ、ユーザー定義型のみ、混在の場合を細分化すると改善が見込める
  - メソッドがコンストラクタなのか、Setter/Getterなのかを判断すると改善が見込める

まとめ/今後の課題

- まとめ
  - 「引数の数」、「引数の型」を計測することでもソースコードを解析することができた
  - Doxygenでインターフェースの構文解析を簡単にできた
  - Jqueryを使った自作ツールは構文解析結果をフィルタリングすること、また簡単にカスタマイズできた
  - 「引数の数」、「引数の型」だけ計測しただけでも手がかりを抽出することができた
- 今後の課題
  - 引数ポイント1～2については未細分化
  - 引数ポイント3～4は組込み型のみ、ユーザー定義型のみ、混在の場合を細分化する
  - メソッドがコンストラクタなのか、Setter/Getterなのかを判断する
  - サンプリングを増やす
  - 手がかりからリファクタリング対象を抽出する評価方法を明確にする