

コード品質を追求する: 多弁なコードを黙らせるために コードの冗長性を測るためのツールとメトリック

Andrew Glover
CTO
Stelligent Incorporated

2006年 6月 30日

開発者のなかには、壮大なコード・ブロックを遠目から眺めただけですくみあがってしまう人もいますが、これは当然のことです。。ほとんどの場合、多弁なコードは複雑さの証明であり、その結果、テストやメンテナンスが困難になります。。今月は、メソッドの長さ、クラスの長さ、そしてクラス内の結合を基準としてコードの複雑さを測定する、3つの重要な方法について説明します。今回の「コード品質を追求する」では、品質のエキスパートである Andrew Glover が過剰なコードを見分けるためのヒントを説明し、次に、PMD やJavaNCSS などのツールを使って、さらに正確にコードの複雑さを見極める方法を紹介します。

ためらわずに認めますが、私は複雑なコードのブロックを見ると本能的にすくみあがってしまいます。敢えて言うなら、私以外の人も長々としたメソッドや雑多なクラスに出くわすと、多少なりとも身震いするものだと確信しています。そんなとき、逃げ道を探してあたふたするのは人間として当然であるだけでなく、開発者の本能でもあります。あまりにも複雑なコードはテストやメンテナンスが難しいため、通常は問題が起こる確率も高くなります。

[このシリーズの以前の記事](#)で説明したように、厄介なコードの尺度として循環的複雑度 (cyclomatic complexity) が使われるようになっていきます。循環的複雑度が高いメソッドをテストすると、ほとんど間違いなく、簡単に出口が見つからない迷路に迷いこんだようになります。先月は、メソッド抽出パターンを使用して迷路からの逃げ道をリファクタリングする方法を説明しました。図1 に示すように、複雑なメソッドを小さく分けることによって、コードのテストやメンテナンスが簡単になります。

図 1. 複雑さの軽減によるコードのテストとメンテナンスの簡単化



循環的複雑度だけが、ハイリスクなコードを見極めるための測定値ではありません。クラスの長さ、メソッドの長さ、そしてクラス内の結合にも着目することができます。これらの測定値は

相互に関連し、見分けるのも簡単です。今月は、この3つがなぜ重要なのかを説明し、PMD と JavaNCSS を使って追跡する方法をご紹介します。

過剰なコード

単純なコードと簡単なコードの違いを知ることは重要です。単純なコードは、必ずしも単純化されているわけでも、簡単に作成しやすいわけでもありません。単に理解しやすいだけです。単純なコードは、VisualBasic で書くのと同じくらい簡単に C++ でも書くことができます。一方、どんな言語であってもコードを複雑にする最も簡単な方法は、一度にさくさんのコードを書くことです。

コードの品質向上

最も急を要する質問に対する答えは、Andrew の[ディスカッション・フォーラム](#)を参照してください。

この法則がメソッドとクラスにどのように当てはまるかを考えてみてください。ほとんどの人はクレジット・カードの番号を記憶するのに苦労します。その理由は単純で、私たちが一度に覚えらるデータは7つ (プラスマイナス 2) のみだからです。このことを念頭に置けば、過剰な条件文を理解するのは困難で、そのためにテストやメンテナンスがしにくくなることもわかります。これと同じ原理が論理ブロックにも適用されます。

コード本体には通常、同じ目標に向けて動作するグループ化されたステートメントが含まれます。例えば、コレクションを作成し、そこに項目を追加するなどの目標です。ただし、たくさんの論理ブロックをグループ化して1つの長いメソッドにすると、メソッドの意図全体があっという間に不透明になってしまいます。そのような大規模なデータ・セットを効率的に扱える人はほとんどいないからです。この弱点こそが、コード・ベースにおけるメンテナンスの問題を引き起こします。巨大なメソッドを効率的に構文解析できる人は数限られているため、巨大なメソッドは不具合の温床と言えます。長いメソッドはたくさん作業をしてくれるだけでなく、理解するのにもたくさんの作業が必要になります。

長いメソッドが開発者を混乱させるように、冗長なクラスにも同じことが言えます。これと同じ議論はコード全体にも当てはまります。長々としたクラスはたくさん作業をしすぎるため、責任も過多になります。

何が余分なのか?

長いメソッドや長いクラスは、当然どこか主観的な構成になっています。目安として、コメントを除いたコードが100行を超えるメソッドは長すぎると言えるでしょう。この数字には無論、個人差があります。私にとっての限界は、50行のコードですが、別の開発者はスクロールダウンしないとメソッドの本体全体が見えなければ、そのメソッドは長すぎると言うかもしれません。この基準を決めるのは、あなた自身です。

同様に、クラスの適切なサイズを決めるのにも個人の判断が必要です。大多数が主張する目安は、1つのクラスにつき1,000行のコードで、これより多くの行を持つクラスは大きすぎるとされます。一方、500行のコードであれば、耐えられるという人もいます。

クラス内の結合

あるオブジェクトが別のオブジェクトと関係を持つようになる場合、複雑さを増加させるパターンは繰り返されます。外部依存関係をたくさんインポートするクラスや、public メソッドを多く持つクラスを理解するのは難しいだけでなく、それによってクラスの責任の負担が増えるため、クラスが脆弱になります。

まず、依存関係から説明しましょう。オブジェクトが 80 を超える外部クラスをインポートする場合(標準 JavaTM システム・ライブラリーを除く)、そのクラスの遠心性結合の値は高いことになります。つまり、インポートされたクラスを変更すると、クラス自体に影響が出ます。最悪の場合、具体的なクラスがインポートされ、それぞれの動作が変更されると、インポートを行うクラス自体が壊れてしまいます。(遠心性結合について詳しくは、参考文献を参照してください。)

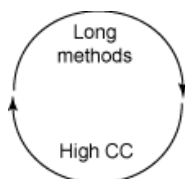
脆弱性を予測するための有効な手段はインポートするオブジェクトの数に注意することですが、パッケージ全体が * 表記でインポートされている場合 (例えば、com.acme.user.*) は誤解を招くおそれがあります。正確を期すには、オブジェクトが所有する固有の型の数に注意を払うことです(これは、import 文ではなく、コードを構文解析することによって行います)。固有の型のメトリックは、アプリケーションのパッケージ構造が、少数のパッケージに多くのクラスを組み込めるように大まかにレイアウトされている場合にも役立ちます。

多くの public メソッドが含まれるクラスも、インポートの数が多くなりがちです。このようなクラスは通常、ファサードまたはユーティリティ・クラスのいずれかとして、コード・ベースの中心になります。この責任(多数の public メソッドによって公開される)により、このようなクラスは高い求心性結合を持つとされます。高い求心性結合も、逆の脆弱性につながります。これらのクラスのいずれかが変更されると、関連性のないように見えるアプリケーションのさまざまな部分が壊れてしまう可能性があります。

複雑さの相関関係

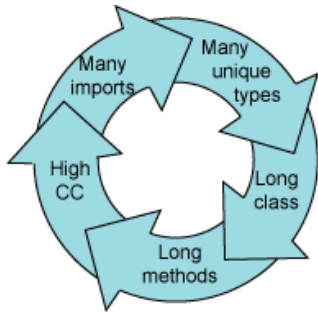
これまでの説明で、貪欲なコード (長いメソッド、多すぎる公開メソッド、過剰な条件文とインポートなど)は理解し難く、テストやメンテナンスの容易性も損なわれるというパターンが見えてきました。このパターン自体がさまざまなメトリックで繰り返されるため、メトリックはすべて相関する傾向にあります。例えば、図2 に示すように、長いメソッドでは通常循環的複雑度が高くなります。

図 2. 長いメソッドと循環的複雑度の相関関係



相関関係はこれだけにとどまりません。過剰なインポートを持つクラスには、固有の型がたくさんあります。そのようなクラスは一般的にかなり大きくなります。大きなクラスは通常長いメソッドを持ち、さらに、長いメソッドが高い循環的複雑度を持つことは珍しくありません。図3 に、複雑度のメトリックがどのように相関しているかを示します。

図 3. 複雑度のメトリックの相関関係



PMD と JavaNCSS

PMD と (それほどではないにしろ) JavaNCSS では、冗長なコードを簡単に見分けられます。この2つのツールはどちらも、Ant や Maven などのビルド・プラットフォームに簡単に組み込むことができます。

PMD はいわばルールに基づいたエンジンで、ソース・コードを分析してルールに違反しているインスタンスを報告します。PMDは現在、メソッドの長さ、クラスの長さ、固有の型、および public メソッドの数それぞれに対する固有のルールを200 近く定義しています。カスタム・ルールを定義したり、既存のルールを (例えば、ドメインの必要性に合わせて) 変更することもできます。

PMD のカスタマイズ

例えば、長いメソッドを見つけるには PMD の ExcessiveMethodLength ルールを使います。このルールでは、メソッド長のしきい値のデフォルトは100 (スキャンしたメソッドの長さが 100 行を超えていると、PMD がルール違反を報告します) ですが、このしきい値は必要に合わせて小さい値にすることができます。

PMD ルールではプロパティを定義できます。PMD 開発チームの優れた先見の明により、定義したプロパティは実行時にルール・セット・ファイルを使ってオーバーライドすることができます。ExcessiveMethodLengthルールのデフォルト値 100 を 50 にするには、properties 要素をルール定義に追加して、プロパティの名前を参照するだけで済みます。リスト1 では、minimum という名前のプロパティが PMD の rule の定義に追加されています。

リスト 1. ExcessiveMethodLength ルールのカスタマイズ

```
<rule ref="rulesets/codesize.xml/ExcessiveMethodLength">
  <properties>
    <property name="minimum" value="50"/>
  </properties>
</rule>
```

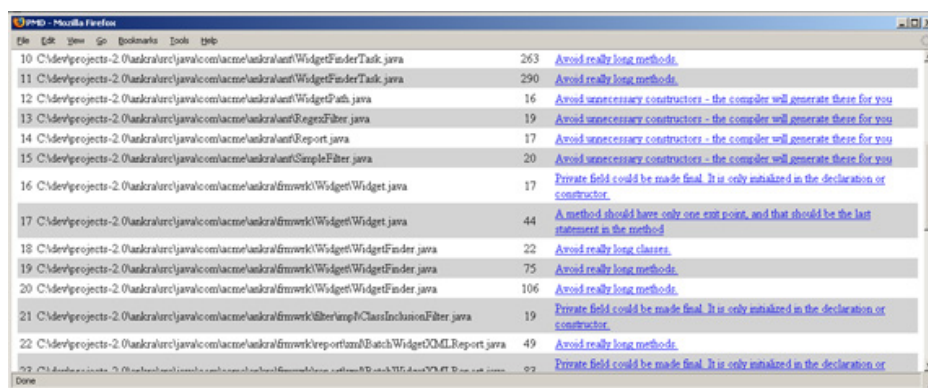
Ant のカスタム・ルール・セット・ファイルで PMD を呼び出すには、リスト 2に示すように、PMD タスクの rulesetfiles 属性を使用して、そのカスタム・ファイルのパスを指定する必要があります。

リスト 2. カスタム・ルール・セット・ファイルの参照

```
<pmd rulesetfiles="./tools/pmd/rules-pmd.xml">
  <formatter type="xml" toFile="${defaulttargetdir}/pmd_report.xml"/>
  <formatter type="html" toFile="${defaulttargetdir}/pmd_report.html"/>
  <fileset dir="./src/java">
    <include name="**/*.java"/>
  </fileset>
</pmd>
```

図 4 に示すように、PMD はソース・ファイルごとに違反を報告します。この例では、ソースコードが50 行を超えているメソッドがいくつか見つかっています。

図 4. PMD Ant レポートの例



File	Line	Message
C:\dev\project-2\src\java\com\acme\ankra\ant\WidgetFinderTask.java	263	Avoid really long methods.
C:\dev\project-2\src\java\com\acme\ankra\ant\WidgetFinderTask.java	290	Avoid really long methods.
C:\dev\project-2\src\java\com\acme\ankra\ant\WidgetPath.java	16	Avoid unnecessary constructors - the compiler will generate these for you
C:\dev\project-2\src\java\com\acme\ankra\ant\RegexFilter.java	19	Avoid unnecessary constructors - the compiler will generate these for you
C:\dev\project-2\src\java\com\acme\ankra\ant\Report.java	17	Avoid unnecessary constructors - the compiler will generate these for you
C:\dev\project-2\src\java\com\acme\ankra\ant\SimpleFilter.java	20	Avoid unnecessary constructors - the compiler will generate these for you
C:\dev\project-2\src\java\com\acme\ankra\framework\Widget\Widget.java	17	Private field could be made final. It is only initialized in the declaration or constructor.
C:\dev\project-2\src\java\com\acme\ankra\framework\Widget\Widget.java	44	A method should have only one exit point, and that should be the last statement in the method.
C:\dev\project-2\src\java\com\acme\ankra\framework\Widget\WidgetFinder.java	22	Avoid really long classes.
C:\dev\project-2\src\java\com\acme\ankra\framework\Widget\WidgetFinder.java	75	Avoid really long methods.
C:\dev\project-2\src\java\com\acme\ankra\framework\Widget\WidgetFinder.java	106	Avoid really long methods.
C:\dev\project-2\src\java\com\acme\ankra\framework\filter\amp\ClassInclusionFilter.java	19	Private field could be made final. It is only initialized in the declaration or constructor.
C:\dev\project-2\src\java\com\acme\ankra\framework\report\ant\BatchWidgetOMLReport.java	49	Avoid really long methods.
C:\dev\project-2\src\java\com\acme\ankra\framework\report\ant\BatchWidgetOMLReport.java	92	Private field could be made final. It is only initialized in the declaration or constructor.

長いクラスに対しては、PMD には ExcessiveClassLength ルールがあり、デフォルトで1,000 行のコードに設定されています。ExcessiveMethodLength ルールと同様に、このデフォルト値はより適切な値で簡単にオーバーライドできます。さらに、PMDには CouplingBetweenObjects と名付けられた固有の型の数に対するルールがあります。インポートの数については、ExcessiveImports ルールを調べてください。どちらのルールも構成が可能です。

JavaNCSS による冗長性の測定

明確なルールを定義してソース・コードを分析する PMD とは対比的に、JavaNCSSはコード・ベースを分析して、クラス・サイズ、メソッド・サイズ、そしてクラス内で検出されたメソッド数などコード長に関するすべての項目を報告します。JavaNCSSでは、しきい値は関係ありません。検出されたすべてのファイルをカウントし、サイズに無関係にその値を報告します。PMDと比べ、この種のデータはありきたり（で冗長）にも見えますが、事実からかけ離れたものにはなりません。

JavaNCSS ではすべてのファイル・サイズが報告されるため、相対値を理解することができます。これは、PMDではなかなか困難です。例えば、PMD は違反が含まれるファイルのみを報告するため、コード・ベースの一部に関するデータしか理解できませんが、JavaNCSSではコンテキストにおけるコード長データが提供されます。図 5 を参照してください。

図 5. JavaNCSS Ant レポートの例

Packages

Nr.	Classes	Functions	NCSS	Javadocs	Package
1	8	38	223	40	com.acme.phnix.ant
2	2	11	75	10	com.acme.phnix.frmwrk.dependency
3	1	4	10	5	com.acme.phnix.frmwrk.dependency.exception
4	1	2	4	1	com.acme.phnix.frmwrk.filter
5	3	13	57	12	com.acme.phnix.frmwrk.filter.impl
6	1	0	3	1	com.acme.phnix.frmwrk.report
7	3	8	88	14	com.acme.phnix.frmwrk.report.xml
19	76		460	83	Total

Packages	Classes	Functions	NCSS	Javadocs	Iper
7.00	19.00	76.00	460.00	83.00	Project
	2.71	10.86	65.71	11.86	Package
		4.00	24.21	4.37	Class
			6.05	1.09	Function

Objects

Nr.	NCSS	Functions	Classes	Javadocs	Class
1	6	2	0	1	com.acme.phnix.ant.Dependencies
2	139	20	0	17	com.acme.phnix.ant.DependencyFinderTask
3	8	3	0	4	com.acme.phnix.ant.DependencyPath
4	10	4	0	5	com.acme.phnix.ant.RegexFilter
5	13	5	0	6	com.acme.phnix.ant.Report
6	10	4	0	5	com.acme.phnix.ant.SimpleFilter
7	1	0	0	1	com.acme.phnix.ant.SourceDependencies
8	1	0	0	1	com.acme.phnix.ant.TargetDependencies
9	19	6	0	5	com.acme.phnix.frmwrk.dependency.Dependency
10	43	5	0	5	com.acme.phnix.frmwrk.dependency.DependencyFinder
11	9	4	0	5	com.acme.phnix.frmwrk.dependency.exception.DependencyFinderException
12	3	2	0	1	com.acme.phnix.frmwrk.filter.Filter
13	8	3	0	2	com.acme.phnix.frmwrk.filter.impl.ClassInclusionFilter
14	22	6	0	7	com.acme.phnix.frmwrk.filter.impl.RegexPackageFilter
15	11	4	0	3	com.acme.phnix.frmwrk.filter.impl.SimplePackageFilter
16	1	0	0	1	com.acme.phnix.frmwrk.report.IDependencyReport
17	39	3	1	7	com.acme.phnix.frmwrk.report.xml.BatchDependencyXMLReport
18	13	2	0	3	com.acme.phnix.frmwrk.report.xml.DependencyXMLReport
19	16	3	0	4	com.acme.phnix.frmwrk.report.xml.XMLReportHelper
Average Object NCSS:					19.58
Average Object Functions:					4.00
Average Object Inner Classes:					0.05
Average Object Javadoc Comments:					4.37
Program NCSS:					460.00

まとめ

開発チームが空白の IDE コンソールを起点として、見事で簡潔なコードを入力するグリーンフィールド開発は、ソフトウェア・アプリケーションの生涯においては非常に小さな部分です。今日、世界中の多数の組織では、いまだにCOBOL をベースとしたアプリケーションを実行しています。これは開発者の側からすると、大昔の見知らぬ人が書いたコードと格闘するという事です。

そんな怪物に直面して嫌気が差すのは当然のことですが、何日も続けて病欠の電話をすることしかできません。ある時点で、その巨大なコード・ブロックと対決し、征服しなければなりません。敵を知る最初のステップは、クラスの長さ、メソッドの長さ、そしてクラス内の結合(つまり、オブジェクトのインポートと固有の型)に対する複雑度メトリックを用いることです。まず始めにクラスとメソッドのサイズに関する目安を知り、PMDおよびJavaNCSSなどのツールを使って詳細に焦点を合わせてください。

長年使われたコード・ベースで複雑度メトリックを始めて使うと、非常に多くのことが分かりますが、そこで終わらせてはなりません。複雑度メトリックを引き続き監視することで、長期にわたってコード・ベースを拡張およびメンテナンスすると同時に、より賢い判断を下してリスクを低減させることができます。

著者について

Andrew Glover



Andrew Gloverは合衆国ワシントン特別区にある、Vanward TechnologiesのCTO（最高技術責任者）です。Vanward Technologiesは自動化テスト・フレームワークの構築を専門としており、ソフトウェアのバグ発生数や統合時間やテスト時間の減少、また全体的なコード安定性改善に貢献しています。

© Copyright IBM Corporation 2006

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)