# FindBugs 第 1 回: コード品質を改善する

## なぜ、どのように FindBugs を使うか

Chris Grindstaff
Software Engineer
IBM

2004年 5月 25日

静的な分析ツールは開発者の側に大きな努力を要求することなく、コード中にあるバグを見つけてくれます。当然ながら長年プログラミングをしている人ならば、こうしたうたい文句が必ずしも正しいものではないことを知っているでしょう。とは言え静的分析ツールは、良質なものであればツールボックスの中に入れておくだけの価値があるものです。2 部構成のこのシリーズでは、上級ソフトウェア・エンジニアの Chris Grindstaff が、FindBugs がコードの品質改善や、隠れているバグを取り去るのにどれほど役に立つかを説明します。2 回シリーズの本記事の後半、第 2 回目も忘れずにお読み下さい。

コード品質ツールの問題の一つは、実際には問題ではないようなもの、つまり false positives まで拾い上げ、開発者を圧倒してしまうことにあります。false positive が起きると開発者はツールの出力を無視するようになるか、あるいは結局ツールを使わなくなってしまいます。FindBugs を作ったDavid Hovemeyer と William Pugh はこの問題をよく考え、false positive を減らすように大いに努力しています。他の静的解析ツールとは異なり、FindBugs はスタイルやフォーマットには注目せず、本当のバグや潜在的なパフォーマンス問題のみを見つけようとするのです。

## FindBugs とは何か？

FindBugs はクラスや JAR ファイルを検査する静的解析ツールであり、バイトコードとバグ・パターン・リストを比較することで潜在的な問題を探し出します。静的解析ツールを使うと、実際にプログラムを実行せずにソフトウェアを解析することができます。プログラムの意図する所を確かめるために、プログラムを実行する代わりに多くの場合 Visitor パターン (参考文献) を使って、クラスファイルの形式や構造を解析します。図 1 はある無名のプロジェクトを分析した結果を示しています。

図 1. FindBugs の UI


FindBugs で検出できる問題のいくつかを見て行きましょう。

## 見つけられる問題の例

次のリストは FindBugs で見つけられる問題の全てではありませんが、より皆さんに関心のありそうななものに注目しています。

Detector: Find hash equals mismatch
このチェック機能 (detector) は `equals()` と `hashCode()` の実装に関連した問題を見つけます。この 2つのメソッドはほとんど全ての Collections ベースのクラス、つまり List、Maps、Sets などから呼ばれるので、非常に重要です。一般的に、このチェック機能は次のような2種類の問題を検出します。

- クラスが Object の `equals()` メソッドを上書きするが、その `hashCode` は上書きしない時。又はその逆の時。
- クラスが `equals()` や `compareTo()` メソッドの co-variant を定義する時。例えば `Bob` クラスがその `equals()` メソッドをブール値の `equals(Bob)` として定義すると、これはObjectで定義した `equals()` メソッドをオーバーロードになります。Java コードによるコンパイル時のオーバーロードメソッド解決方式のために、(`equals()` への引数をタイプ `Bob` に明示的にキャスト

しない限り）ランタイムに使うメソッドはほとんど必ず Object で定義するものになり、`Bob` で定義したものではありません。その結果このクラスのインスタンスの一つが集合クラスに入れられる時には、このメソッドの `Object.equals()` 版が使われ、`Bob` で定義したものは使われません。この場合では `Bob` クラスは、タイプ Object の引数を受け付ける `equals()` メソッドを定義すべきなのです。

Detector: Return value of method ignored
このチェック機能はメソッドの戻り値が（無視されるべきではないのに）無視されている場所を探し出します。このシナリオの一般的な例は `String` メソッドを呼び出す時に見ることができます (リスト1)。

## リスト 1. 無視された戻り値の例

```
1  String aString = "bob";
2  b.replace('b', 'p');
3  if(b.equals("pop"))
```

この間違いはごく一般的なものです。2行目で、このプログラマーはストリングにある全ての b を p で置き換えたと思っています。確かに置き換えているのですが、ストリングの不変性 (immutable) を忘れています。このタイプのメソッドは全て新しいストリングを戻し、このメッセージの受信者を変えません。

Detector: Null pointer dereference and redundant comparisons to null
このチェック機能は 2種類の問題を見つけます。コードのパスが null ポインター例外を引き起こすか、引き起こす可能性がある場合、また null との冗長比較がある場合を見つけ出します。例えば 2つの値を比較して両方とも明らかに null だった場合、この 2つは冗長ということになり、コーディング間違いの可能性があります。FindBugs は一方の値が null で他方が null ではないと判定できる時にも、似たような問題を検出します (リスト2)。

## リスト2. null ポインターの例

```
1  Person person = aMap.get("bob");
2  if (person != null) {
3      person.updateAccessTime();
4  }
5  String name = person.getName();
```

この例では、1行目の `Map` が「bob」という名前の人を持っていない場合には、5行目で `person` の名前を尋ねられた時にヌル・ポインター例外が発生します。FindBugsはmap に「bob」が入っているかどうかを判別できないため、5行目に対してヌル・ポインター例外の可能性があるというフラグを立てるのです。

Detector: Field read before being initialized
このチェック機能はコンストラクターで読まれるフィールドのうち、初期化前に読まれてしまうフィールドを見つけます。この間違いは (常にではありませんが) 多くの場合、コンストラクター引き数の代わりに間違ってフィールド名を使ってしまう結果起こります (リスト3)。

### リスト 3. コンストラクターにあるフィールドを初期化前に読んでしまう

```
1  public class Thing {
2      private List actions;
3      public Thing(String startingActions) {
4          StringTokenizer tokenizer = new StringTokenizer(startingActions);
5          while (tokenizer.hasMoreTokens()) {
6              actions.add(tokenizer.nextToken());
7          }
8      }
9  }
```

この例では `actions` が初期化されていないので、6行目でヌル・ポインター例外が起きることになります。

こうした例は FindBugs が検出する問題のごく一部にすぎません (他の問題に関しては参考文献 を見てください)。この記事の執筆時点で FindBugs には合計 35 のチェック機能があります。

## FindBugs を使ってみる

FindBugs を実行するには Java Development Kit (JDK), version 1.4 以上が必要です。ただし、古い JDK で生成したクラスファイルも解析は可能です。まず最初に FindBugs の最新リリース (現在は 0.7.1です。参考文献) をダウンロードしてインストールします。幸いダウンロードもインストールもごく単純です。zip または tar を適当なディレクトリにダウンロードしたら unzip します。それが終わればインストールは完了です。

インストールができたので、サンプルクラスに対して実行してみましょう。多くの記事と同様、この記事でも Windows ユーザーを念頭に書いていますが、Unix 派の人も苦もなく理解できると想定しています。コマンドプロンプトを開き、FindBugs をインストールしたディレクトリに行きます。私の場合は C:\apps\FindBugs-0.7.3. です。

FindBugs のホーム・ディレクトリにはいくつか面白いディレクトリがあります。説明ファイルは doc ディレクトリにあります。もっと大事なこととして、bin ディレクトリには FindBugs を実行するためのバッチファイルがあります。これを次に説明します。

## FindBugs を実行する

最近のツールはどれもそうですが、FindBugs もいくつかの方法で実行できます。GUI でも実行できますし、コマンドラインからでも、Ant を使っても、Eclipse のプラグインとしても、または Maven を使って実行することもできます。ここでは FindBugs を GUI で実行する方法を簡単に説明しますが、中心的に説明するのは Ant やコマンドラインから実行する方法です。理由は、GUI ではコマンドラインにあるようなオプションが全て使えるわけではないためです。例えば UI では、フィルターに対して特定のクラスを含むか含まないようにするかを指定することができません。もっと大事な理由として、私の意見では FindBugs はビルドに統合された一部として使う時に最高の力を発揮するものですが、自動ビルドに UI は向かないからです。

### FindBugs の UI を使う

FindBugs の UI はごく単純に使えますが、いくつか注意すべき点があります。図 1 で分かるように FindBugs の UI を使う利点の一つは、検出された各問題のタイプに対して説明が表示されることです。図 1 では Naked notify in method というバグの説明を示しています。他のバグのパターンにも

似たような説明が用意されており、FindBugs に慣れるには大いに役立ちます。ウィンドウの下側の区画にある Source code タブも同様に便利です。このタブに切り替えて、「ソースがどこにあるかを探せ」と FindBugs に言うと、違反しているコード行がハイライトされます。

もう一つ大事な点は FindBugs を Ant のタスクとして、またはコマンドラインから実行している時に `output` オプションとして `xml` を選択すると、以前実行した結果を UI にロードしてくることができるのです。こうすることでコマンドライン・ベースのツールと UI ツールを同時に使う利点を最大限に生かすことができます。

## FindBugs を Ant のタスクとして実行する

Ant のビルド・スクリプトから FindBugs をどのように使うかを見て行きましょう。まず FindBugs の Ant タスクを Ant の lib ディレクトリにコピーし、Ant が新しいタスクを認識するようにします。FIND_BUGS_HOME\lib\FindBugs-ant.jar を ANT_HOME\lib にコピーします。

FindBugs タスクを使うために、ビルド・スクリプトに何を追加すべきかを見てみましょう。FindBugs はカスタム・タスクなので、Ant にどのクラスをロードすべきかが分かるように `taskdef` タスクを使う必要があります。それにはビルドファイルに次の行を追加します。

```
<taskdef name="FindBugs"
classname="edu.umd.cs.FindBugs.anttask.FindBugsTask"/>
```

`taskdef` を定義した後は、それを `FindBugs` という名前で呼ぶことができます。次に新しいタスクを使うビルドにターゲットを追加します (図 4)。

## リスト4. FindBugsターゲットを作る

```
1  <target name="FindBugs" depends="compile">
2      <FindBugs home="${FindBugs.home}" output="xml" outputFile="jedit-output.xml">
3          <class location="c:\apps\JEdit4.1\jedit.jar" />
4          <auxClasspath path="${basedir}/lib/Regex.jar" />
5          <sourcePath path="c:\tempcbg\jedit" />
6      </FindBugs>
7  </target>
```

このコードが何をしているかを少し詳しく見てみましょう。

**1行目**:`target` がコンパイルに依存することに注意してください。ここは覚えておくべき重要な点ですが、FindBugs はソースファイルではなくクラスファイル上で動作するので、target をコンパイル・ターゲット依存とすることで FindBugs がどんな最新クラスファイルででも確実に実行できるようになります。FindBugs は入力に関して柔軟であり、クラスファイル一式、JAR ファイル、一連のディレクトリなども入力として受け付けます。

**2行目**: FindBugs を含むディレクトリを指定する必要があります。これは Ant プロパティを使って次のようにします。

```
<property name="FindBugs.home"
value="C:\apps\FindBugs-0.7.3" />
```

オプションとしての属性 `output` は FindBugs が使う出力フォーマットを指定します。使える値としては `xml` か `text` または `emacs` です。もし `outputFile` を何も指定しないと、FindBugs は標準出

力に出力します。先に書いたように、XML フォーマットは UI 内で見られるという点でより有利です。

3行目: どのセットの JAR、クラスファイル、ディレクトリが必要なのかを指定するために `class` 要素を使います。複数の JAR ファイルやクラスファイルを解析するには、それぞれに対して別々の `class` 要素を指定します。 `projectFile` 要素が含まれているのでない限り `class` 要素は必要です。詳細については FindBugs のマニュアルを見てください。

4行目: ネストされた要素 `auxClasspath` を使ってアプリケーションの依存性をリストアップします。これらのクラスはアプリケーションに必要なものですが FindBugs には解析させたくないものです。アプリケーションの依存性をリストアップしなくても FindBugs はやはりそうしたクラスを解析しますが、探しているクラスが見つからない時には文句を言ってきます。`class` 要素の場合と同じく、FindBugs 要素にも複数の `auxClasspath` 要素を指定することができます。`auxClasspath` 要素はオプションです。

5行目:`sourcePath` 要素が指定されている場合には、`path` 属性はアプリケーションのソースコードがあるディレクトリを指します。ディレクトリを指定することで、GUI で XML の結果を見る時に FindBugs がエラーのあるソースコードをハイライトするようになります。この要素はオプションです。

これで基本的な所を説明しました。では何週間分かを早送りしましょう。

## フィルター

あなたは FindBugs をチームに導入し、日々のビルド・プロセスの一部として使い始めました。チームがこのツールに慣れてくるに従って、(何らかの理由で) 検出されるバグの一部は重要ではないと判定したとしましょう。あなたは一部のクラスが、悪意に変更される可能性があるオブジェクトを戻しても気にしないかもしれませんし、あるいは JEdit のように、神に誓って正直に `System.gc()` を呼び出すだけの理由があるかもしれません。

特定のチェック機能を指定して、いつでもそのチェック機能をオフすることができます。さらにきめ細かなレベルとして、チェック機能を設定して特定のクラス群、または特定のメソッド群であっても、その一群の中にある問題をチェックしないようにすることができるのです。 FindBugs では include と exclude フィルターを使って細かな制御ができます。Exclude と include フィルターは現在 FindBugs のコマンドライン版とAnt版でのみサポートしています。その名前の示す通り、exclude フィルターはある種のバグのレポートを含まないようにするために使います。Include フィルターは exclude ほどには使われませんが、やはり便利なもので、特定なバグのみをレポートします。フィルターは XML ファイルで定義します。Includeやexclude を指定するにはコマンドラインで exclude/include のスイッチをつけるか、Antビルド・ファイルの中で `excludeFilter` または `includeFilter` を使います。下の例では exclude スイッチを使っていると想定してください。また、この先の説明では「バグコード」「バグ」「チェック機能」をそれぞれを、ある意味で置き換え可能なものとして使っていることに注意してください。

フィルターには様々な定義の仕方があります。

- あるクラスに一致するものをフィルターにかける。このフィルターは、ある特定なクラスで見つかる全ての問題を無視するのに使います。

- あるクラスの中にある、特定なバグコードに一致するものをフィルターにかける。このフィルターは、ある特定なクラスで見つかる一部のバグを無視するのに使います。
- 一連のバグに一致するものをフィルターにかける。このフィルターは、解析するクラス全体に渡る一連のバグを無視するのに使います。
- 解析するクラスの中にある、特定なメソッドに一致するものをフィルターにかける。このフィルターは、あるクラスの一連のメソッドで見つかる全てのバグを無視するのに使います。
- 解析するクラスの中のメソッドで見つかる、一部のバグに一致するものをフィルターにかける。このフィルターは非常にバグの多い一連のメソッドで見つかるバグの一部を無視するのに使います。

始めるにあたって必要なのはこれだけです。FindBugs のタスクをカスタム化する他の方法の詳細については FindBugs の説明資料を読んでください。さてこれでビルドファイルの設定方法が分かったので、FindBugs をビルドプロセスに統合するにはどうするかを見て行きましょう。

## FindBugs をビルドプロセスに統合する

FindBugs をビルドプロセスに統合するにはいくつかのオプションがあります。FindBugs をコマンドラインから実行することももちろんできますが、おそらくビルドに Ant を使っているでしょうから、FindBugs Ant タスクを使うのが一番自然でしょう。FindBugs Ant タスクを使うための基礎は先に説明したので、FindBugs をビルドプロセスに加える理由と、進める中で突き当たりそうな問題をいくつか説明することにします。

### なぜ FindBugs をビルドプロセスに統合すべきなのか

私が頻繁に受ける質問の中で最初に聞かれるのが、なぜ FindBugs をビルドプロセスに加える必要があるのか、というものです。理由はたくさんありますが、最も明らかな答えとしてはビルドを実行したらできるだけ早く問題を検出しておきたい、というものです。チームが成長するにつれて必然的により多くの開発の初心者が加わることになるので、FindBugs を安全策として既知のバグ・パターンの検出に使うことができます。FindBugs の資料の一つにも説明があるのですが、開発者がある程度の数になるとコードにバグが入り込むことになります。確かに FindBugs のようなツールは全てのバグを検出するわけではありませんが、一部を見つける手助けになります。(特に FindBugs をビルドプロセスに組み込むためのコストが非常に低いことを考えれば) 一部でも今見つけておいた方が、後で顧客に見つけられるよりもずっと良いと言えるでしょう。

どのフィルターを入れるか、どのクラスを入れるかが固まれば FindBugs を実行するためのコストはごく僅かであり、しかも新しいバグを検出できるという利点が付きます。アプリケーション専用のチェック機能を書けばその利点はさらに大きいはずです。

### 意味のある結果を生成する

このコスト対利点の分析は、false positives を大量に生成することがない、という前提でのみ意味を持つことを認識する必要があります。つまりビルドからビルドへ繰り返していった時、新しいバグが出てきたかどうかが簡単に判別できないのだとしたらツールの価値は無くなってしまう、ということです。解析は自動化されていれば自動化されているほど良いものです。バグ修正が、検出されても実際には関係のないバグの中をかき分けて進むような作業を意味するのであれば、おそらく誰もツールを使わなくなるか、少なくともうまい使い方はしなくなるでしょう。

意味のある結果を生成するには、どの問題が自分にとっては気にする必要がないのかを確定し、それらをビルドから排除します。あるいは自分が本当に問題にする、一連の小さなチェック機能を選び、それだけを実行するようにします。別の手段としては、ある一連のチェック機能は個々のクラスから排除し、他のものは残すことです。FindBugs はフィルターの使い方に非常に柔軟性があるので、意味のある結果を生成しやすくなっています。これを次にとりあげます。

### FindBugs の結果をどう処理するかを決める

結果をどう処理するか、は明らかな質問に思えるかも知れませんが、FindBugs 風のツールを単に面白いからという理由でビルドに加えているとしか思えないようなチームが、皆さんの想像以上の数にのぼるのです。この質問をもう少し詳しく考えてみましょう。この結果をどうすべきなのか？これはチームがどのように構成されているか、コード所有権の問題をどうするのか、といった問題に大きく依存するので、具体的に答えるのは困難な問題です。参考になりそうな指針としては次のようなものがあります。

- FindBugs の結果をソースコード管理 (source code management: SCM) システムに加えることを考慮する。大まかに言って、ビルドの成果物は SCM システムに入れるべきではありません。ただしこの場合に限っては、時間と共に変わるコード品質を監視することができるので、このルールを破った方が良いのかも知れません。
- チームの Web サイトに掲示できるように XML の結果ファイルを HTML レポートに変換する。この変換はXSLスタイルシートやスクリプトで行うことができます。FindBugs の Web サイトやメーリング・リストにあるサンプルをチェックして見てください (参考文献)。
- FindBugs のようなツールは、チームやある個人を追い立てるための政治的な武器になってしまいがちなものです。そうした使い方を奨励すべきではありませんし、そんな使い方にならないようにしましょう。繰り返しますが、これはコード品質を改善するために意図したツールなのです。そうした精神論は別として、次回の記事ではカスタムのバグチェック機能を書くにはどうすべきかを説明する予定です。

## まとめ

FindBugsであれ、PMDであれ、その他でも、何らかの静的分析ツールを使ってコードを解析してみることをお勧めします。こうしたツールは便利なものであり、本当の問題を見つけることができます。FindBugsはそのなかでも、false positivesを除去してくれるという点で他のものよりは優れています。さらにプラグ可能な構成のため、アプリケーションに特化した重宝なチェック機能を書くためのテスト・ベッドとなります。このシリーズ第2回ではアプリケーション特有の問題を見つけるためのカスタムのバグチェック機能を書くにはどうすべきかを説明する予定です。

---

# 著者について

Chris Grindstaff

Chris Grindstaffはノースキャロライナ州にあるIBM in Research Triangle Parkの上級ソフトウェアエンジニアです。7歳の時に初めてプログラムを書いたのですが、その際に文章をタイプするのは手書きするのと同じくらいの罰になりうるのだと先生に納得させたのです。現在は様々なオープンソース・プロジェクトに興味を持っています。Eclipseで豊富な経験を積んでおり、よく使われるEclipseプラグインをいくつか書いています（彼のWebサイトを参照）。