FindBugs 第2回: カスタムのチェック機能を書く

アプリケーション特有の問題発見にカスタムのチェック機能をどう書くか

Chris Grindstaff
Software Engineer
IBM

2004年 5月 25日

FindBugsはそれぞれのチーム特有の要求に合わせて拡張、カスタム化できる静的解析ツールです。このシリーズ第2回目の今回は、上級ソフトウェア・エンジニアのChris Grindstaffがアプリケーション特有のバグを検出するチェック機能（detectors）を書く方法を説明します。このシリーズ 第1回 の記事も忘れずに読んでください。

このシリーズ 第1回 ではFindBugsをどのように設定し、実行するかを説明しました。今度はFindBugsの最も強力な機能、カスタムのバグ・チェック機能を見て行きます。カスタムのバグ・チェック機能がなぜ便利なのかその理由を最初に説明し、その後で詳細例を説明して行きます。

カスタムのバグ・チェック機能を書く

なぜカスタムのバグ・チェック機能を書く必要があるのでしょう。私はあるチームのパフォーマンス問題を調査するよう依頼された時に、この疑問に突き当たりました。調べてみると、このチーム自前のログ・フレームワークは（ログ・フレームワークは皆そうですが）、時間と共に大きくなりすぎたことが明白だったのです。元々、`Loggers` は何も考えずに呼ばれていました。不幸なことにチームが成長するにつれ、アプリケーションのパフォーマンス低下もだんだんひどくなっていきました。このチームでは高価なログ・メッセージを常に生成していたためです。それもログ・メッセージは、ログ・フレームワークがログ使用不可と認識した時に捨てられるだけで、何も使われていないのです。この問題を標準的に扱うには、高価なログ・メッセージを構築する前に、最初にログが使用可能になっているかどうかをチェックすべきなのです。言い換えれば、リスト1にあるような保護文（guard clause）を使うのです。

リスト1. 保護されたログの例

```
if(Logger.isLogging()) {
    Logger.log("perf", anObjectWithExpensiveToString + anotherExpensiveToString);
}
```

このシリーズ第1回の記事もお忘れなく

このシリーズ第1回の記事「 コード品質を改善する 」では、クラスやJARファイルを検査して潜在的な問題を見つける静的解析ツールとしてのFindBugsと、その効果的な使い方を説明しています。

このチームはこの方法がログのための表現方法として適切と判定し、この新しい方法を反映して既存のコードを変更したのです。驚くに当たらないかも知れませんが、この巨大プロジェクトの締め切りが近づくにつれ、見落としたコード部分がいくつもあることが分かったのです。足りない部分を特定するためにはもっと良い方法が必要でした。この記事はFindBugsに関するものなので、FindBugsを使ってこの問題を解決しましょう。

目標としては、保護文にラップされることなくログ・フレームワークを呼んでいるコード部分をすべて見つけるFindBugsチェック機能を書くことです。

私がこのチェック機能を最初に書いた時には、次のようにいくつか別々のステップに分解しました。

1. 最初に非保護ログ・ステートメントでテストケースを書く。
2. 次にFindBugsのソースコードをながめ、自分が書きたいものに似たチェック機能を探す。
3. 次に、非保護チェック機能のロード方法がFindBugsに分かるように、正しくパッケージしたJARファイルを（ビルド・スクリプトを使って）作る。
4. テストを実行し、テストが通るようにコードを実装する。
5. 最後に、さらにテストを追加し、終わるまでこのシーケンスを続ける。

コードを眺めている時には特に、 `BytecodeScanningDetector` と `ByteCodePatternDetector` のサブ・バイトを調べました。スキャニングのチェック機能は実装にかかる手間も多いのですが、検出できる問題もより一般的です。検出しようとしているものがバイトコードのシーケンスとして表現できる時には、パターンのチェック機能を選ぶ方が良い選択と言えます。この良い例が `BCPMethodReturnCheck` チェック機能です。このチェック機能は、色々なメソッドの戻りタイプが、よく分からない理由で無視されている部分を探します。 `BCPMethodReturnCheck` は、 `POP` または `POP2` 命令が後に続く特定なメソッドの呼び出しを探すパターン・シーケンスとして容易に記述できます。圧倒的多数のチェック機能はスキャニングのチェック機能として書かれていますが、これは単にその内の多くを `ByteCodePatternDetector` に動かしているだけの時間の余裕が開発者に無いためだと私は思っています。

私は例として `FindRunInvocations` をずっと使っているのですが、その理由はこれが一番小さなチェック機能だからです。パターンのシーケンスを使ってチェック機能をどう実装すべきかは、私にとっては明白なものではありませんでした。

FindBugsはByte Code Engineering Library（BCEL 参考文献 ）を利用してチェック機能を実装しています。バイトコード・スキャニング・チェック機能は全てvisitorパターンに基づいていますが、FindBugsはこれを実装しています。visitorパターンはこうしたメソッドのデフォルト実装を提供していますが、デフォルト実装はカスタムのチェック機能を実装する時には上書きされます。詳細については `BetterVisitor` とそのサブクラスを見てください。ここでは2つのメソッド、`visit(Code)` と `sawOpcode(int)` のみに注目します。FindBugsがクラスを解析する際、メソッドの

内容をウォークする時には `visit(Code)` メソッドを呼びます。同様にFindBugsは、メソッド・ボディ内部の各命令コードを解析する時には `sawOpcode(int)` メソッドを呼び出します。

こうしたことを頭に置いた上で、非保護ログのチェック機能を作るために使われている、こうしたメソッドの実装を見てみましょう（ リスト2 ）。

リスト2. 非保護ログのチェック機能：visit()メソッド

```
18      public void visit(Code code) {
19          seenGuardClauseAt = Integer.MIN_VALUE;
20          logBlockStart = 0;
21          logBlockEnd = 0;
22          super.visit(code);
23      }
```

箱から出しただけで手を付けていないチェック機能のコードを読んでいて目に付くのは、解析中に状態を構築をする必要があるかどうかに注目している、という点です。言い換えれば、メソッド、クラス、階層構造あるいは全プログラムのレベルで見るものを、記憶しておく必要があるかどうか、という点です。例えば `Inconsistent Synchronization` チェック機能は全プログラムに関する状態を構築するので、同期化の面から一貫性に欠ける方法でフィールドがアクセスされると、それがいつなのかを判別できるのです。ここで書こうとしているチェック機能ではメソッド・レベルの問題を探しているので、状態を維持する必要があるのはバイトコードのスキャニング・フェーズの期間中のみです。

チェック機能はメソッド特有の状態を累積していきますが、これは `visit(Code)` メソッドで吐き出し、またはリセットすることができます（ リスト2 ）。これはFindBugsがこのメソッドを呼び出すのは、あるメソッドのバイトコードをスキャンする前だからです。この場合では、チェック機能が保持する状態には次のような3ビットがあります。

- `seenGuardClauseAt` : 解析したコードの中にログ保護文が見つかった時のプログラム・カウンターの値
- `logBlockStart` : 保護文開始のインデックス
- `logBlockEnd` : 保護文終了後の命令のインデックス

`visit(Code)` メソッド実装に関して重要な点を2つ指摘しておく必要があります。最初の点は `super.visit()` へのコールですが、このメソッドのスーパークラスの実装が、解析対象のメソッドの内容を調べる責任があるので、これは重要です。もしスーパークラスの実装を呼ばないと、解析対象のメソッドは調べられません。

2番目は、累積された状態はスーパークラスの実装を呼ぶ前にリセットされるという点です。こうした変数は、これから見ようとしている次のメソッド、 `sawOpcode()` メソッドが使うので、これは重要です。こうした変数が、 `sawOpcode()` メソッドを使う前にリセットされていることを確認する必要があります。リスト3は `sawOpcode()` メソッドの実装を示します。

リスト3. 非保護ログのチェック機能：sawOpcode()メソッド

```
25      public void sawOpcode(int seen) {
26          if ("cbg/app/Logger".equals(classConstant) &&
27                  seen == INVOKESTATIC &&
28                  "isLogging".equals(nameConstant) && "()Z".equals(sigConstant)) {
29              seenGuardClauseAt = PC;
30              return;
31          }
32          if (seen == IFEQ && (PC >= seenGuardClauseAt + 3 && PC < seenGuardClauseAt + 7)) {
33              logBlockStart = branchFallThrough;
34              logBlockEnd = branchTarget;
35          }
36          if (seen == INVOKEVIRTUAL && "log".equals(nameConstant)) {
37              if (PC < logBlockStart || PC >= logBlockEnd) {
38                  bugReporter.reportBug(
39                          new BugInstance("CBG_UNPROTECTED_LOGGING", HIGH_PRIORITY)
40                          .addClassAndMethod(this).addSourceLine(this));
41              }
42          }
43      }
```

先に書いた通り、FindBugsがメソッドを解析する時には、そのメソッドに含まれる各バイトコード命令に対して `sawOpcode()` を呼びます。このメソッドは3つのことをしています。元のコードは実際には3つのメソッドにリファクターされているのですが、私はこの記事での便宜上、スペース節約のためにインライン化しています。このメソッドは次のような3つのことをします。

1. 静的メソッド `Logger.isLogging()` が呼ばれたかどうかを判定する。もし呼ばれていれば、プログラム・カウンター（ program counter: PC ）の値は何か。
2. Determines if an if instruction follows the call made to Logger.isLogging() `Logger.isLogging()` へのコールの後に `if` 命令があるかどうかを判定する。
3. 保護文の外で `log()` メソッドが呼ばれている場合を探す。

リスト4はこれらの部分をそれぞれ、より詳細に示しています。

リスト4. 非保護ログのチェック機能：sawOpcode()、isLogging()が呼ばれる

```
25      public void sawOpcode(int seen) {
26          if ("cbg/app/Logger".equals(classConstant) &&
27                  seen == INVOKESTATIC &&
28                  "isLogging".equals(nameConstant) && "()Z".equals(sigConstant)) {
29              seenGuardClauseAt = PC;
30              return;
31          }
```

`classConstant` と `nameConstant` 、それに `sigConstant` フィールドはチェック機能がそのスーパークラスから継承する、保護されたフィールドです。こうしたフィールドには現在の命令コードの詳細が含まれています。チェック機能を書く時には、こうしたフィールドの値を出力するようにしておくと便利なことがよくあります。 `BytecodeScanningDetector` の階層構造をながめると、より有用なフィールドやメソッドが `DismantleBytecode` クラスにあることがわかります。チェック機能を書く時に使うツールとしてもう一つ非常に便利なのは、永遠なる `javap` です。チェック機能を書くにあたって、論理の流れやメソッド名を理解するために、Javaの逆アセンブラーは非常に重宝なツールです。一般的な手法としては、探そうとするパターンを書き（ この場合ではJavaファイルで保護文を書き ）、保存してからコンパイルします。次に `javap -c` を使って逆アセンブ

ルしたバイトコードを見て、 `sawOpcode(int)` メソッドの構成方法を考えるのです。例えばリスト5は、私がテストケース (ログ保護文を正しく使っているメソッド) のために使ったクラスに対して `javap` を実行させた後の出力を示しています。:

リスト5. 保護文の逆アセンブル例をソースと共に示す

```
public void methodWithLogging_guarded();
  Code:
   0:   invokestatic    #28; //Method cbg/app/Logger.isLogging:()Z
   3:   ifeq    18
   6:   new     #16; //class Logger
   9:   dup
   10:  invokespecial   #17; //Method cbg/app/Logger."<init>":()V
   13:  ldc     #19; //String bob
   15:  invokevirtual   #23; //Method cbg/app/Logger.log:(Ljava/lang/Object;)V
   18:  aload_0
   19:  invokespecial   #31; //Method doWork:()V
   22:  return
 corresponds to the Java source code
   public void methodWithLogging_guarded() {
        if (Logger.isLogging()) {
            new Logger().log("bob");
        }
        doWork();
    }
```

`javap` の出力を調べるとメソッドの制御フローが理解しやすくなり、また `sawOpcode()` メソッドで指定すべきクラスやシグニチャー、名前定数などの構成方法も分かりやすくなります。例えばリスト6は、リスト5にある `javap` のコードの1行目です。

リスト6. メソッド呼び出しの逆アセンブル

```
0:   invokestatic    #28; //Method cbg/app/Logger.isLogging:()Z
```

リスト4 にある `sawOpcode()` メソッドの26行目から28行目をよく見ると、 `javap` からとった リスト5 の中に見えるものと一致する方法を記述していることが分かるでしょう。 `javap` はこうした形式をどのように一致させるかを決めるために有用なツールです。

`Logger.isLogging()` メソッドが呼ばれたことが判定できたらば、プログラム・カウンターの値を保存します (リスト7)。プログラム・カウンターは、 `if` 文が `Logger.isLogging()` へのコールの後に来ているかどうかを判定するために必要です。これが次のコード部分につながって行きます。

リスト7. プログラム・カウンターの値を保存する

```
32          if (seen == IFEQ && (PC >= seenGuardClauseAt + 3 && PC < seenGuardClauseAt + 7)) {
33              logBlockStart = branchFallThrough;
34              logBlockEnd = branchTarget;
35          }
```

リスト3 からとったこのコードは、先程の `Logger.isLogging()` へのコールから3～7バイトコード離れた所に分岐命令があるのを調べています。こうした値は `javap` の出力を見たり、試してみたりすることで判定します。試してみて？ そうです。false positivesと有効な結果とのちょうど良いバランスをとるためには、時には試行錯誤が必要なのです。このプロセスはコンピュー

ター・サイエンスではなく、経験論によるコンピューター技術だと思ってください。この命令が `if(Logger.isLogging())` 命令だと判定したら、 `if` のコードブロックの境界を判定する必要があります。これは `branchFallThrough` と `branchTarget` を保存することで行います。 `branchFallThrough` は `if` 文の初めであり、 `branchTarget` は `if` 文の外にある最初の行を表します。この情報で、このメソッドの最終部分に進むことができます（リスト8）。

リスト8. log()へのコールをチェックする

```
36          if (seen == INVOKEVIRTUAL && "log".equals(nameConstant)) {
37              if (PC < logBlockStart || PC >= logBlockEnd) {
38                  bugReporter.reportBug(
39                          new BugInstance("CBG_UNPROTECTED_LOGGING", HIGH_PRIORITY)
40                          .addClassAndMethod(this).addSourceLine(this));
41              }
42          }
```

このコードのブロックも リスト3 からとったものですが、 `Logger` の `log()` メソッドに対するコールを探します。 `log()` メソッドに対するコールが見つかったら、プログラム・カウンターが先程決めた `if` ブロックの外側かどうかをチェックします。外側であれば、新しいバグ・インスタンスを作り、バグのタイプ（後ほど詳しく説明します）とその優先度を規定することで、バグをレポートします。ユーザーが問題を修復するにはどこに行けばよいかが分かるように、クラスやメソッド、バグに至る部分のソース行を追加しておくと便利です。

コードが書けたら、FindBugsがプラグインJARとして認識するよう特別にパッケージしたJARファイルを作る必要があります。リスト9は、JARファイルを作って正しい場所にコピーするために私が使った、ビルド・スクリプトのターゲットを示します。

リスト9. FindBugsチェック機能をパッケージするためのビルド・スクリプト

```
<property name="FindBugs.home" value="C:\apps\FindBugs-0.7.3"></property>
<target name="build">
 <jar destfile="cbgFindbugsPlugin.jar">
  <fileset dir="bin"/>
  <fileset dir="src"/>
  <zipfileset dir="etc" includes="*.xml" prefix=""></zipfileset>
 </jar>
 <copy file="cbgFindbugsPlugin.jar" todir="${FindBugs.home}/plugin" />
</target>
```

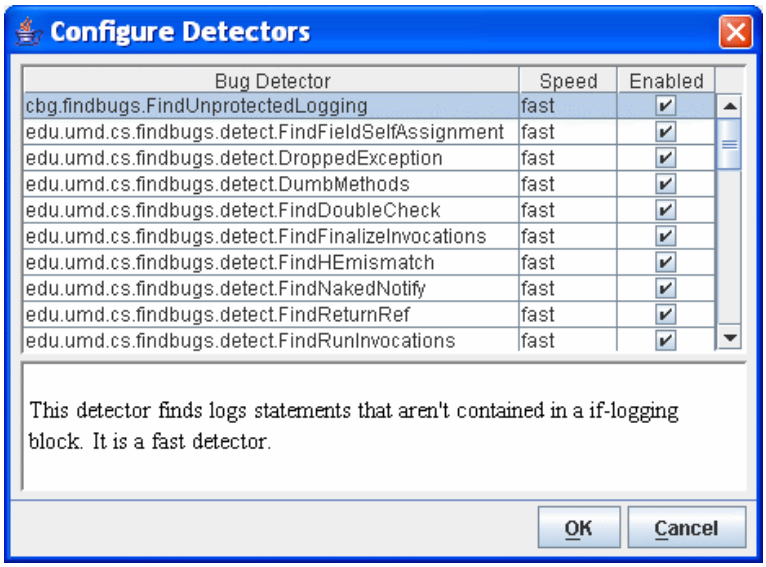
このコードでソースファイルやクラスファイル、それにFindBugs.xmlとmessages.xmlを含むJARファイルができます。リスト10と11はこの2つのXMLファイルの内容を示します。

リスト10. FindBugs.xmlの内容

```
<FindbugsPlugin>
  <Detector class="cbg.findBugs.FindUnprotectedLogging" speed="fast" />
  <BugPattern abbrev="CBGL" type="CBG_UNPROTECTED_LOGGING" category="PERFORMANCE" />
</FindbugsPlugin>
```

新しいチェック機能それぞれに対して、FindBugs.xmlファイルに `Detector` 要素と `BugPattern` 要素を追加します。 `Detector` 要素はチェック機能の実装に使うクラスと、それが高速なチェック機能か低速なものかを規定します。スピード属性はUIでチェック機能を見る時に使います（図1）。スピード属性の値として取りうるのはslow（低速）、moderate（中速）、fast（高速）です。

図1. チェック機能UIを設定する



`BugPattern` 要素は3つの属性を規定します。 `abbrev` 属性はチェック機能の短縮形（short abbreviation）を規定します。短縮形はコマンドライン・クライアントから実行する時に、検出したバグを規定するのに使います。同じ短縮形を持ったチェック機能をいくつかまとめてグループ化することができます。

`type` 属性はユニークな規定子で、2つの目的に使います。FindBugsのAnt版またはコマンドライン版で出力フォーマットをXMLとして使う時には、問題を規定するために `type` 属性を使います。バグの正しいタイプを作るために（チェック機能の）Javaコードで規定するのも `type` 属性です。ここでリストに挙げたタイプは、 リスト8 の39行目で使われている名前と一致することに注意してください。

`category` 属性は列挙型です。これは下記の内のどれかです。

- `CORRECTNESS` : 一般的な正しさの問題
- `MT_CORRECTNESS` : マルチスレッドの正しさの問題
- `MALICIOUS_CODE` : 悪意のコードにさらされた場合の潜在的な脆弱性
- `PERFORMANCE` : パフォーマンスの問題

FindBugs.xmlファイルに関してはこれだけです。リスト11はmessages.xmlファイルの内容を示しています。

リスト11. messages.xmlファイルの内容

```
<MessageCollection>
  <Detector class="cbg.FindBugs.FindUnprotectedLogging">
    <Details>
<![CDATA[
<p> This detector finds logs statements that aren't contained in an if-logging block.
It is a fast detector.
]]>
    </Details>
  </Detector>
  <BugPattern type="CBG_UNPROTECTED_LOGGING">
    <ShortDescription>Found unprotected logging</ShortDescription>
```

```
    <LongDescription>Found unprotected logging in {1}</LongDescription>
    <Details>
<![CDATA[
<p> This method logs without first checking that logging is enabled; for example
... more text omitted...
]]>
    </Details>
  </BugPattern>
  <BugCode abbrev="CBGL">Found unprotected logging</BugCode>
</MessageCollection>
```

messages.xmlファイルは `Detector` 、 `BugPattern` 、 `BugCode` という3つの要素から成ります。

Detectorの `class` 属性はチェック機能のクラス名を規定します。Details要素にはチェック機能についての簡単なHTML記述があり、従って `CDATA` 部分に含まれるべきものです。この記述は図2で示すように、UIで使われます。

図2. FindBugsのUIが非保護ログのチェック機能をハイライトしている

`BugPattern` 要素はFindBugs.xmlで定義される `BugPattern` 要素と似ています。type属性は必要であり、FindBugs.xmlやチェック機能のJavaコードで使ったのと同じユニークな規定子と一致する必要があります。 `BugPattern` は、チェック機能に関する情報をUIでどう表示するかに影響する要素を3つ持っています。 `ShortDescription` と、 `LongDescription` と `Details` ですが、どれも内容は自明でしょう。

`ShortDescription` はUIでView > Full Descriptionsが切り替えてオフにされた時に使います。同じように `LongDescription` はView > Full Descriptionsが切り替えてオンにされた時に使います。注記を使うことで、バグ・チェック機能のJavaコードからの情報を、full descriptionとして渡すことができます。descriptionの中では、 `{0}` で最初の注記を表し、 `{1}` で2番目の注記を表し、というようにして変数を規定します。実行してバグが見つかると、バグ・インスタンスに付加した注記は何でもdescriptionとして置き換えられるのです。 リスト8 の40行目で、 `BugInstance` にクラスとメソッドの注記が追加されていることに注意してください。クラス注記は位置0、メソッド注記は位置1です。詳細については、 `BugInstance` のいろいろな `add*()` メソッドを見てください。

前と同じく、 `Details` 要素は `CDATA` 部分にHTML記述があります。図2は今回のチェック機能の詳細例です。View > Full Descriptionsはオンになっています。

`BugCode` 要素はUIでBy Bug Typeタブを使う時に使います。この要素のテキストが、 図2 にあるような赤いノードとしてツリー上に現れるものです。一般的なチェック機能は同じ短縮形を使うので、 `BugCode` 要素はその短縮形を要素の属性として規定する必要があります。

この2つのXMLファイルができたので、完全なJARをパッケージ化する準備が整いました。JARをビルドしてFIND_BUGS_HOME\pluginディレクトリに置くと、新しいチェック機能を使ったテストの準備が完了です。

アプリケーション専用のバグ・チェック機能

FindBugsは持っていると便利なツールです。ただし他のツールと同じように、どういう場合に使うべきかを知っている必要があります。とは言っても静的解析ツールはユニット・テストやシステム・テスト、コード見直し等を補うものと言えます。

FindBugsにはコード品質改善のためのユーティリティに加えて、アプリケーション専用の使い方もいろいろあるので、皆さんにも試してみることをお勧めします。例えば、新人が犯しがちな問題を見つけるチェック機能を書くことができます。コードがチームの指針に沿ったものになっているかどうかを調べるチェック機能を書くこともできます。あるフレームワークを構築した場合に、パッケージ中の全クラスが必ず引き数ゼロのコンストラクターを持つようにするとか、先頭にアンダースコアーがついたフィールドにはゲッターはあってもセッターは無いようにする必要があるかも知れません。あるいは `Thread` 生成も `Socket` 生成も無い、というような制限にJ2EEコードが従っていることを証明するための一連のチェック機能を書くこともできます。

非保護ログの例に挙げたチームでは、例外を捉えるのにも問題を抱えていました。称賛すべき事に、このチームでは単純に例外を無視したわけではありません。例外が、そのスタック・トレースを出力するようにしたのです。これはアプリケーションの構築中やデバッグ中であれば良いことなのですが、展開時には得策とは言えません。何千もの例外の可能性がある場合には特にそうです。（当然ながらアプリケーションが何千もの例外を投げるのであれば、ログファイルが大きいことよりもずっと重大な問題があるはずですが、説明のためですので我慢してください。）このチームでは、例外を捉えてそのスタック・トレースを出力するように言っているコード部分を見つけるチェック機能を必要としていたのです。そうすれば例外をログ・フレームワークの方に渡すように、コードを変更することができたのです。

私は非保護ログのチェック機能の変種として面白いものを作りました。このチェック機能は、ログすべきメッセージが保護文の外側で作られている（これも良くある問題で、カッコいい `toString` があると非常に高くつきます）コード部分を全て見つけるために使いました。

まとめ

FindBugsが初めての人も既に慣れている人も、自分のアプリケーション専用のチェック機能を試してみることをお勧めします。この記事ではカスタムのチェック機能の実装方法を明快に説明できたと思います。こうした概念を皆さんのチーム特有の問題にも応用できることを祈っています。

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）。