

Rational Software Analyzer による静的解析

Part 2 – Java コード・レビューを拡張するためのルールとルールフィルターの作成

Level: 中級

Steve Gutz (sgutz@ca.ibm.com), Manager, Rational Software Analyzer

2008 年 4 月 29 日

この記事は IBM® Rational® Software Analyzer の静的解析機能について解説する 4 部構成の記事の 2 番目のものです。ここでご紹介する静的解析機能のサブセットは Rational® Application Developer および Rational® Software Architect にも含まれています。ここでは、Java コード・レビューの使い方にフォーカスし、ルールとフィルターの作成過程についてウォークスルーを行います。

IBM® Rational® Software Analyzer は、自動化されたコード品質改善プロセスを促進し、単純化します。最初は他の Rational ツールに静的解析ツールを統合するための API とユーザー・インターフェースとして設計されました。その後完全に独立したコード・レビュー、構造分析を行う製品として発展してきました。

原文 URL :

http://www.ibm.com/developerworks/rational/library/08/0429_gutz2/

本シリーズの最初の記事では、ハイレベルな観点から静的解析機能を調べ、IBM Rational® Software Analyzer のユーザーインターフェースを紹介しました。また、プロバイダーやカテゴリ、ルールを選択して分析構成を作成する過程についても説明しました。静的解析の結果が表示されるリザルト・ビューや基本的なレポート機能についても議論しました。

パート 2 とパート 3 では、Java コード・レビューのルール・セットを拡張するためにルールを書く方法を学びます。Rational Software Analyzer の解析フレームワークはいくつかの拡張ポイントと、新しい解析フォームとルールを追加するための完全な API を含んでいます。しかしもっと重要なことは、Java ソースコードに対してコード・レビューを実行するための機能解析プロバイダーが完全に提供されていることです。本記事と次の記事ではこのフレームワークを解説し、あなたが必要とするルールを書くことができますようにします。

この記事はルールを作成するコンポーネントの説明からはじめます。それから、カテゴリやルール拡張ポイント、**Java IDE** の簡単な概説を行います。最終的には、あるグループの簡単なルールを実装することにより、あなたが独自のルールを作成する方法を理解できるようにプロセス全体のウォークスルーを行います。

Java ツールの概要

既にご存知のように **Eclipse** プラットフォームは **Eclipse** 内部の機能が利用する **Java** ソースコードの完全なパーサーを提供しています。

(例えば、**Java** ソースアウトラインの表示や依存の発見やシンタックスハイライト等で使われています)

幸運にも **Eclipse** ではこの **IDE** をプラットフォーム開発者が利用できるように公開しています。**JDT** は **Java** ソースファイルを解析し、**API (application programming interface)** から紹介可能なメモリ内ツリー構造表現を生成します。**Java** ファイルのトップレベルの抽象シンタックスツリー (**AST**) はおおまかにいうとこの方法で構造化されています。

```
CompilationUnit:  
  [ PackageDeclaration ]  
    { ImportDeclaration }  
      { TypeDeclaration | EnumDeclaration | AnnotationTypeDeclaration | ; }
```

Compilationunit はルートタイプであり、概念的には **Java** ファイルそのもので **TypeDeclarations** (クラスやインターフェース) のコンテナです。ルールを書く際には **MethodInvocations** や **MethodDeclarations**、その他多くのノードタイプを使うことになります。すべてのノードの記述が **Eclipse** ヘルプで参照可能です。ドキュメントには **100** 以上の **Application Server Toolkit (AST)** ノードタイプがありますが、**Javadoc** 情報がよく整備されているので心配することはありません。

AST パースツリーを照会するために **Java IDE** は **Visitor** パターンを使います。これはかなりのパフォーマンスが望めますが、有益な結果のためには **Visitor** クラスを書かなければいけないという脅迫めいた感じを受ける可能性があります。解析 **API** は、共通の **Visitor** クラスと照会メカニズムの抽象化により、**AST** ノードに関する作業のほとんどを削減しています。この **API** はこの記事の全体を通じて言及されます。さしあたり、必要なのは **AST** ツリー構造の基本的な理解が全てであり、**Eclipse Java IDE** のヘルプを参照することで事足ります。ヘルプを精読することに少し時間をとってください。なぜなら、より高度なルールのためには、結局は **AST** コードを書くことが必要になるからです。

Java コード・レビューの拡張ポイント

ここでは、Analyzer の解析フレームワークで利用可能なすべての拡張ポイントを詳細に記述することはしません。なぜなら、数が多すぎるからです。その代わりに、新しいカテゴリーやルールを作成するために必要となる少数の拡張ポイントに焦点を絞ります。プロバイダーやリザルトのようなその他の拡張ポイントはこのシリーズの次の記事でカバーします。

Java プロバイダーのためのコード・レビューは、Rational Software Analyzer、Rational Software Architect、Rational Application Developer で利用でき、あなたが書くどのようなルールでもこれらのツールに配備可能です。分析構成ダイアログを開き、Java プロバイダーを見ると数個のカテゴリーがあることがわかります。これらはそれぞれ、Analyzer の `com.ibm.rsaz.analysis.codereview.java.rules` プラグインのように `plugin.xml` 内で拡張として与えられます。例として、`Plugin.xml` ソースファイルからひとつのカテゴリーを見てみましょう。

```
<analysisCategory
  class="com.ibm.rsaz.analysis.core.category.DefaultAnalysisCategory"
  id="codereview.java.category.awt"
  label="%label.category.j2sebestpractices.awt"
  category="codereview.java.j2sebestpractices"
  help="com.ibm.rsaz.analysis.codereview.java.rules.awt"/>
```

`class` 属性はカテゴリーを管理するクラスを定義します。カスタムカテゴリーを書いた時でさえ、このコードは API によって提供されるデフォルトカテゴリーとして利用され、ほとんどのケースで効果を発揮します。カスタムカテゴリーはカテゴリーの共通の特徴を取り扱わなければなりません。含まれる子階層のリストやラベルやアイコン名の管理等です。これはこの記事のスコープ外ですが、デフォルトカテゴリーはどのようなカテゴリーが要求するサービスでも提供できるということを心にとどめてください。ほとんどすべての状況において、カテゴリーのために提供されたクラスを利用することになります。

カテゴリーはまた、ユニークな ID を持ち、ほかのカテゴリーやルールをまとめるのに使われます。もちろん、カテゴリーはユーザーへの表示に必要なラベルや記述を持たなければなりません。

この例では、拡張は `category` 属性をもっています。このカテゴリーは、ユニーク ID `codereview.java.j2sebestpractices` を持つ他のカテゴリー内にネストされることを示しています。もしトップレベルカテゴリーを作りたければ `category` 属性を `provide=provider.id` で置き換えます。この例では、`codereview.java.analysisProvider` が Java コード・レビュープロバイダーのユニーク ID です。

```
<analysisCategory
class="com.ibm.rsaz.analysis.core.category.DefaultAnalysisCategory"
id="codereview.java.j2sebestpractices"
label="%label.category.j2sebestpractices"
provider="codereview.java.analysisProvider"
help="com.ibm.rsaz.analysis.codereview.java.rules.j2sebestpractices"
</analysisCategory>
```

ルール拡張は似たように動作しますが、もう少し詳細な内容を含みます。ルール拡張のサンプルは下記のようなものです。

```
<analysisRule
category="codereview.java.category.awt"
class="com.ibm.rsaz.analysis.codereview.java.internal.rules.awt.RuleAwtPeer"
id="codereview.java.rules.awt.RuleAwtPeer"
label="%label.relrule.j2sebestpractices.awt.awtpeer"
severity="2"
help="com.ibm.rsaz.analysis.codereview.java.rules.awtpeer">
</analysisRule>
```

ルール拡張の最初のパートはほとんどカテゴリ拡張と同じです。このケースでは、**class** 属性は拡張対象のルールの正確なクラスパスを含みます。

ルールはまた付加的な情報を含みます。ヘルプタグは関連する **help.xml** ファイル内のヘルプコンテキストの **ID** を提供します。これにより、ルールはよく整備された例とソリューション情報をユーザーに提供できます。ヘルプと他の高度なルールの概念については次の記事で議論します。

コード・レビュールールの構造

前のセクションで議論したとおり、ルール定義の前半は **Eclipse** の拡張です。後半は、ルール・クライテリアにマッチするすべての問題に対してクラスの内容を照会し、結果を生成するために、**rule API** か **JDT** の一部、またはその両方を使う **Java** クラスです。ルールクラスはきわめてシンプルで、通常は1つのメソッドしか含みません。ルールは通常 **Rational Software Analyzer** 解析フレームワークにより提供されるデフォルトルールクラスの拡張となるため、必要な機能のほとんどは既に実装されています。リスト1のコードは、2つのオブジェクトを比較するために**==**操作を使用しているすべてのコード行について結果を作成する完全なルールです。

リスト 1. 2つのオブジェクト比較のためのルール

```
public class RuleComparisonReferenceEquality
    extends AbstractAnalysisRule
{
    private static final String[] OPERATORS = { "=", "!=" };
    private static final int[] OPERANDS = { ASTModifier.TYPE_PRIMITIVE,
ASTModifier.TYPE_NULLTYPE };

    // Rule filters
    private static IRuleFilter[] EXPFILTERS = {
        new OperatorRuleFilter( OPERATORS, true ),
        new LeftOperandRuleFilter( OPERANDS, false ),
        new RightOperandRuleFilter( OPERANDS, false )
    };

    /**
     * Analyze this rule
     * @param history A reference to the history record for this analysis
     * @throws CoreException
     */
    public void analyze(final AnalysisHistory history,
        final CodeReviewResource resource )
    {
        List list = resource.getTypedNodeList( resource.getResourceCompUnit(),
ASTNode.INFIX_EXPRESSION );
        ASTHelper.satisfy( list, EXPFILTERS );

        for(Iterator it=list.iterator(); it.hasNext(); ) {
            InfixExpression tempInf = (InfixExpression)it.next();
            ITypeBinding leftBinding = tempInf.getLeftOperand().resolveTypeBinding();
            ITypeBinding rightBinding = tempInf.getRightOperand().resolveTypeBinding();
            if( (leftBinding != null && leftBinding.isPrimitive()) ||
                (rightBinding != null && rightBinding.isPrimitive()) ) {
                it.remove();
            }
        }

        resource.generateResultsForASTNodes( this, history.getHistoryId(), list );
    }
}
```

このルールクラスは解析フレームワークの抽象クラス

`com.ibm.rsaz.analysis.codereview.java.AbstractCodeReviewRule` を拡張するため、派生ルールは`##analyze()`メソッドを実装しなければなりません。

このメソッドはどのルールが実行されたかを表す1つのパラメーター (`history`) をとります。ユーザーが解析を実行するたびに、結果を収集するための新しい履歴が登録されます。これにより `history` 要素はどのプロバイダー、カテゴリー、ルールが選択されたかを追跡し続けます。

`Analyze()`に渡されるパラメーターには、すべての結果と解析される全てのリソースが蓄積されたスキャン履歴に関する情報が含まれます。Java コード・レビュープロバイダーの一部である `AbstractCodeReviewRule` クラスは、解析されているクラスの `Compilationunit` と接触を保ち、クエリーを `AST` ツリーの中で管理します。

例に挙げたルールは内挿表現 (`infix Expression` : 比較のように左右の間に挿入する文字列) に関連するので、`Compilationunit` にあるそのような表現のリストを取得するようにコード・レビューリソースを使います。

```
List list = resource.getTypedNodeList( resource.getResourceCompUnit(),
ASTNode.INFIX_EXPRESSION );
```

次に、ルールは与えられたクワイテリアに合わない表現をフィルターするために `ASTHelper.satisfy()` メソッドを使います。クワイテリアはクラスの先頭の `static` ブロックに定義されています。

```
// Rule filters
private static IRuleFilter[] EXPFILTERS = {
    new OperatorRuleFilter( OPERATORS, true ),
    new LeftOperandRuleFilter( OPERANDS, false ),
    new RightOperandRuleFilter( OPERANDS, false )
};
```

`EXPFILTERS` 配列は 3 つのタイプのルールフィルターを含んでいます。1 番目のルールフィルターは操作が `"=="` でも `"!="` でもない式を取り除きます。2 番目のフィルターは式の左側にプリミティブ型か `null` が指定されている式を取り除きます。最後のフィルターは式の右側について同様に行います。`Satisfy()` メソッドは入力リストを受け取り、全てのフィルタリングを 1 ステップで行います。このソースコード行が実行された後のリストは、左側と右側がともにオブジェクトで、操作が `"=="` か `"!="` のいずれかである式だけを含みます。このルールの目的からして、リストに残っている全ての項目が結果リストにレポートされるべきです。これはルールの最後の行によって実行されています。

```
resource.generateResultsForASTNodes( this, history.getHistoryId(), list );
```

易しいと思いませんか？ `Java` ルールを書くための `API` はほんの数個のメソッドしか含みませんが、極めてパワフルです。

スクラッチからルールを書く

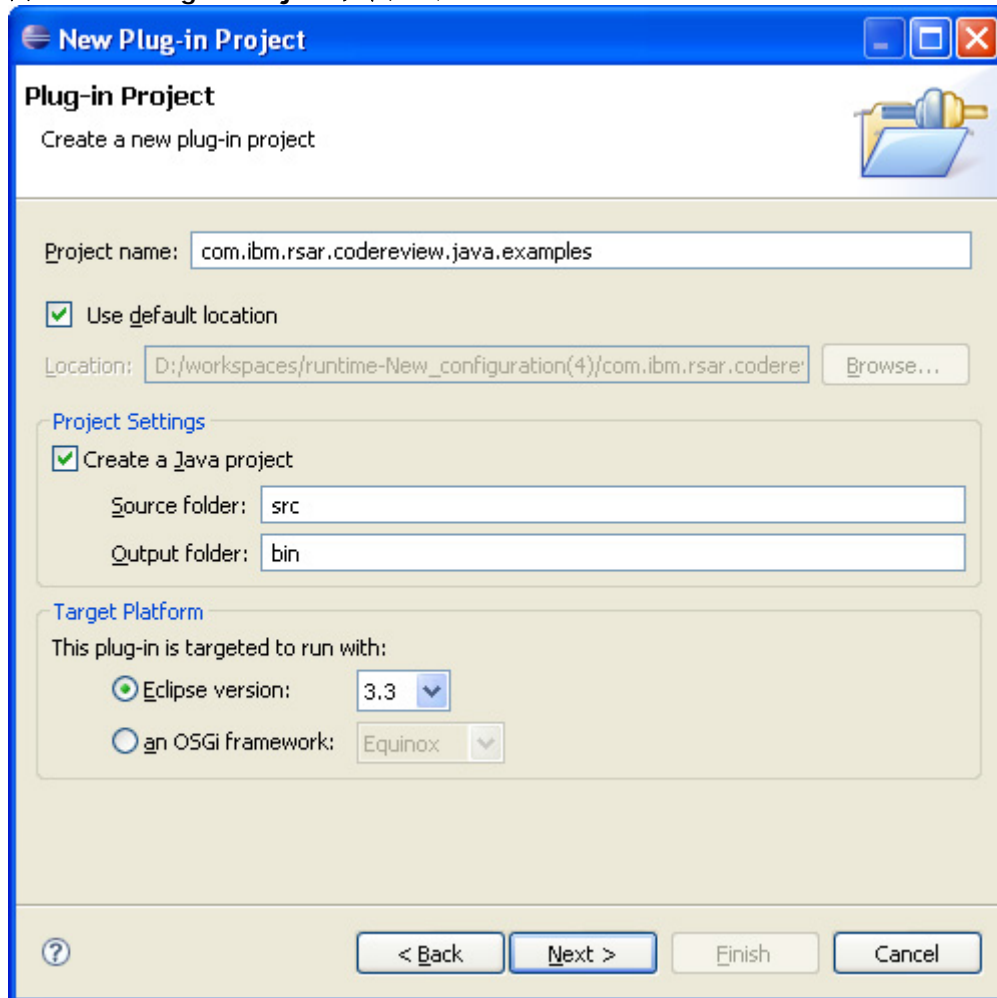
ここまではコード・レビュールールの単純さについて見てきましたが、もう少しチャレンジしてみましょ。ここまで見てきたルールは 1 つのルールができる単純なものでした。`AST` ツリーの 1 種類のノードを照会し、フィルター後の結果のみを見てきました。新しいカテゴリーとルールで完全なルールプラグインを書くために必要なすべてのステップについて考えてみましょう。

プラグインを作成する

ルールを作成するためには、まずプラグインを作成する必要があります。

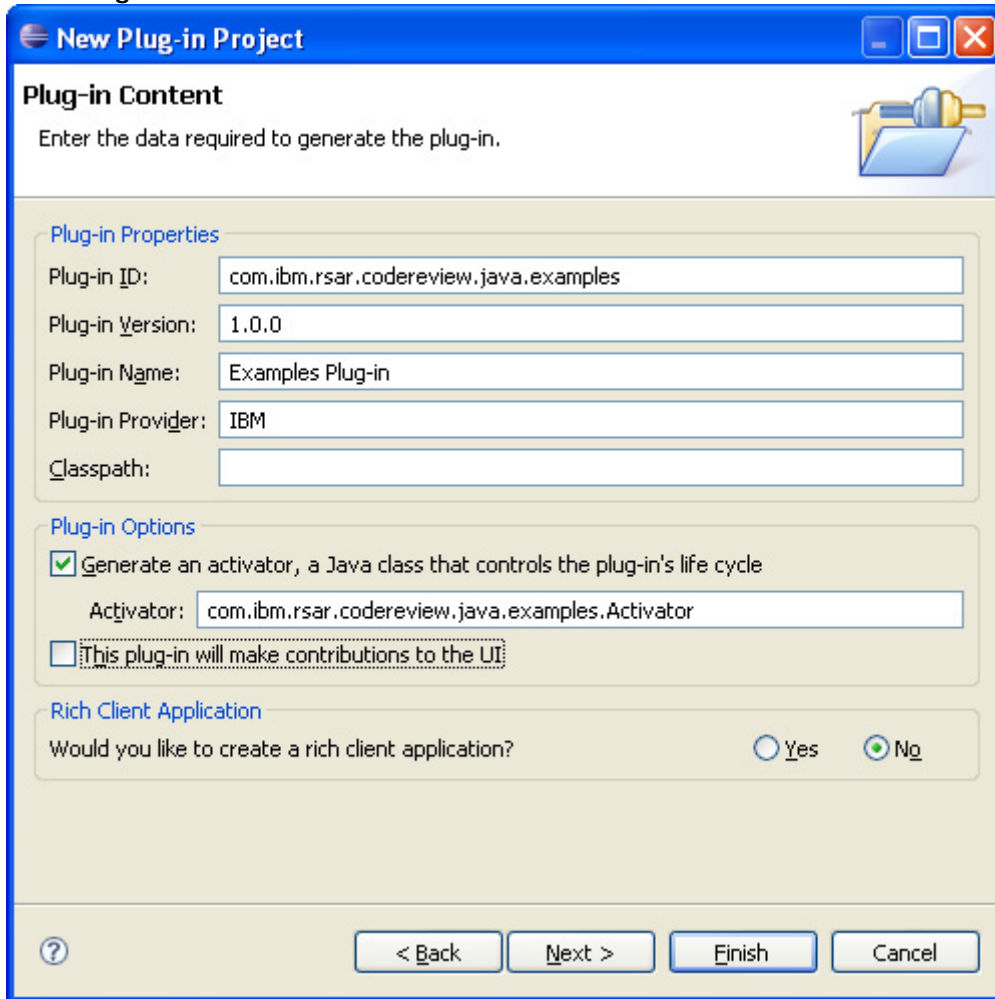
1. Eclipse の **New Project** ウィザードを使ってワークスペースに `#com.ibm.rsar.codereview.java.examples` という名前のプロジェクトを作成します。

図 1. New Plug-in Project ウィザード



2. ウィザードの 2 枚目の画面（図 2）で、“This plug-in makes contributions to the UI”チェックボックスを外します。
3. **Finish** ボタンを押してプラグインを作成します。

2. Plug-in Content view



The screenshot shows the 'New Plug-in Project' dialog box in Eclipse IDE. The title bar reads 'New Plug-in Project'. The main title is 'Plug-in Content', and the subtitle is 'Enter the data required to generate the plug-in.' There is a folder icon with a plug-in symbol in the top right corner.

Plug-in Properties

Plug-in ID:

Plug-in Version:

Plug-in Name:

Plug-in Provider:

Classpath:

Plug-in Options

Generate an activator, a Java class that controls the plug-in's life cycle

Activator:

This plug-in will make contributions to the UI

Rich Client Application

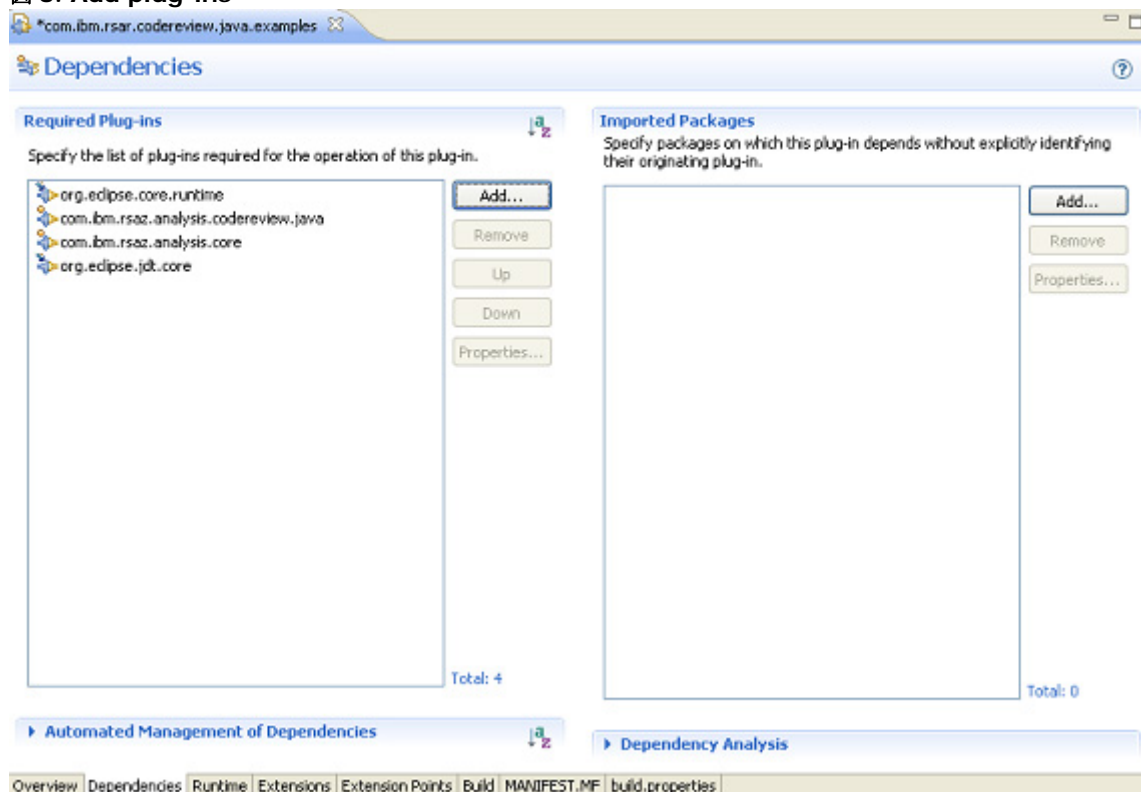
Would you like to create a rich client application? Yes No

At the bottom, there is a help icon (question mark) and four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

うまくルールを作成するためには、プラグインに依存関係を追加する必要があります。

4. プラグインマニフェストファイルを開き、**Dependencies** タブを選択します。
5. **Add** ボタンを使って図 3 のように `org.eclipse.core.runtime` と下記の 3 つのプラグインを追加します。
 - **com.ibm.rsaz.analysis.core**: 基本的な解析フレームワーク API と拡張ポイントを含みます。
 - **com.ibm.rsaz.analysis.codereview.java**: Java コード・レビューを書くための API を含みます。
 - **org.eclipse.jdt.core**: AST パーサーやクエリーフレームワークを包含する JDT API を含みます。

図 3. Add plug-ins



一般的に、新しいルールを生成する際にはこれら 3 つのプラグインの依存関係が必要です。

ルールクラスを作成する

最初のルールは実用的なものです。このセクションでは、`super.finalize()`を呼ぶ以外何もしない `finalize` メソッドを Java コードから検出するルールを作成します。もし `finalize()`メソッドが親を呼ぶだけであれば、そのメソッドは単純に除去されます。例えば：

```
public class Example
{
    private void method() {
        // Do something
    }

    // Don't do this
    protected void finalize() throws Throwable {
        super.finalize();
    }
}
```

こういった問題を検出するために何をする必要がありますでしょうか？ 以下の2つのステップを見てみましょう。

- **Step 1**
まず最初に、ソースコードから `finalize` メソッド宣言のリストを取得する必要があります。また、それらのメソッドはパラメーターなしであるべきです。
- **Step 2**
必要なメソッド宣言のリストを取得したら、それらのコード行を参照して、`super.finalize()`の1行だけが記述されているかどうかを見る必要があります。

メソッド宣言のためのルールを作成する

以下のステップでルールを作成することができます。

1. まず、プラグインの `com.ibm.rsar.codereview.java.example` パッケージに新しいクラスを作成します。このクラスを `RuleFinalizerSuper` と呼ぶことにします。
2. このルールクラスが `com.ibm.rsaz.analysis.codereview.java.AbstractCodeReviewRule` を継承していることを確認します。

```
import com.ibm.rsaz.analysis.codereview.java.AbstractCodeReviewRule;
import com.ibm.rsaz.analysis.codereview.java.CodeReviewResource;
import com.ibm.rsaz.analysis.core.history.AnalysisHistory;

public class RuleFinalizersuper extends AbstractCodeReviewRule
{
}
```

3. #analyze メソッドをクラスに加えます。

```
/**
 * Analyze this rule
 *
 * @param history A reference to the history record
 */
public void analyze(AnalysisHistory history, CodeReviewResource resource ) {
}
```

アルゴリズムの **Step1** に従って分析対象のクラスからメソッド宣言のリストを収集する必要があります。

4. analyze()メソッドがこれを行うために以下の行を追加します。

```
List list = resource.getTypedNodeList( resource.getResourceCompUnit(),
ASTNode.METHOD_DECLARATION );
```

この行は **AST** ツリーを参照し、**Compilationunit** (最上位ノード) からスタートしてすべてのメソッド宣言のリストを照会します。 **getTypedNodeList()**メソッドが複数のシグネチャーを持つことに注意してください。このメソッドはデフォルトで再帰的に呼び出され、すべてのインナークラスのメソッド宣言も発見します。

アルゴリズムの **Step1** を完了するために、メソッド宣言リストのうち、不必要なものをフィルターするメソッドが必要です。これは、ルールフィルターのリストを作成すれば可能です。

5. 以下の行をルールクラスの先頭に加えます。

```
// Rule filters
private static final IRuleFilter[] MIFILTERS = {
};
```

6.

7. まず、finalize()宣言がないメソッドをフィルターする必要があります。空の static ブロックに以下を追加します。

```
new MethodNameRuleFilter( "finalize", true ),
```

#MethodNameRuleFilter クラスはメソッド名をフィルターするメカニズムに **#finalize** を指定します。 **#true** は **finalize()**メソッドがリストに含まれるべきだということを示しています。もしこのフィールドが **#false** だったら、フィルターは **finalize()**を取り除き、その他の行を保持します。

また、真に対象とする **finalize()**メソッドのみを取り扱うことを確実にする必要があります。これは、ルールが無視する必要がある **finalize()**メソッドが値を返すようにすれば可能です。

8. void 型の戻り値を持たない finalize メソッドを取り除くフィルターを追加するために、ReturnTypeRuleFilter クラスを使用します。

```
new ReturnTypeRuleFilter( "void", true ),
```

フィルターされたリストのサイズを削減するために、パラメーターを持たない `finalize()` メソッドのみを取得します。パラメーターを持つメソッドはクラスファインライザーではありません。(実際には、ルールはこれらのメソッドについて無効であることをレポートすべきです)

9. `static` ブロックに次の行を追加します。

```
new ParameterCountRuleFilter( 0, true )
```

このケースでは、フィルターコードに対してパラメーター数が `0` であることを指示しています。

最終的なフィルター配列はリスト 2 のようになります。

リスト 2. 最終的なフィルター配列

```
// Rule filters
private static final IRuleFilter[] MIFILTERS = {
    new MethodNameRuleFilter( "finalize", true ),
    new ReturnRuleFilter( "void", true ),
    new ParameterCountRuleFilter( 0, true )
};
```

アルゴリズムの **Step1** を完了するために実際にフィルターを実行する必要があります。

10. `analyze()` メソッドの最後に以下の行を追加します。

```
ASTHelper.satisfy( list, MIFILTERS );
```

スーパーメソッドの呼び出しが 1 行だけであることを確認する

ルールの易しいパートは終了しました。ここからは少し話が複雑になります。

メソッドのコード行が 1 行しかないことを確かめる必要があります。取得したいのはメソッド宣言のリストであることを思い出してください。リストに含まれるアイテムは 1 つだけであるべきです。なぜなら、シグネチャーが同じ複数のメソッド宣言を **Java** はサポートしないからです。

1. 安全のため、以下の行を **analyze()** メソッドの最後に追加し、**list** イタレーターを作成します。

```
for( Iterator it = list.iterator(); it.hasNext(); ) {
    MethodDeclaration md = (MethodDeclaration)it.next();
}
```

すべてのメソッド宣言はボディを含みます。(カッコ内の記述です)メソッド宣言を取得した時、**getBody()**メソッドを使用して **Block** (これは 1 つの **AST** ノードです) を取得することができます。

2. イタレーターループの最後に以下のコードを追加します。

```
Block block = md.getBody();
if( md.getBody().statements().size() == 1 ) {
```

もう少しです。あとは **super finalize()** の呼び出しを発見した時にステートメントをチェックするだけです。

3. **if** ステートメントに以下のコードを追加します。

```
List superList =
resource.getTypedNodeList( block, ASTNode.SUPER_METHOD_INVOCATION );
if( superList.size() > 0 ) {
```

ここで少し説明が必要です。メソッド宣言のボディはステートメントからなっており、あらゆる種類のアクションであり得ます。(例えば、**for** ステートメント、**if** ステートメント、比較など)。型を指定したスーパーメソッド呼び出しがないかどうか、メソッドボディを参照する必要があります。これはステートメントブロックから **SUPER_METHOD_INVOCATION** タイプの **AST** ノードのリストを取得することで可能になります。既にボディが 1 行のみであることはわかっているので、このリストは **0** か **1** いずれかの要素を持ちます。リストが **1** つかそれ以上の要素を持つならば、ボディは **super.finalize()** への呼び出しのみであると想定することができます。

最後に、解析結果ビューに結果を生成します。このために、コード・レビューエンジンは単純な **API** を提供します。

4. 空の **if** ステートメントに以下の行を追加します。

```
resource.generateResultsForASTNode( this, history.getHistoryId(), md.getName() );
```

このメソッドコールの最初の 2 つのパラメーターは常に同じです。結果を生成するルールと、結果が位置すべき場所の履歴のコレクションです。3 つめのパラメーターはハイライト表示されるべき AST ノードを示します。このケースでは、マッチするすべてのメソッド宣言 (`finalize()`メソッド) で名前がハイライト表示され、結果が生成されます。

完成したルールはリスト 3 のようになります。

リスト 3. 完成したルール

```
package com.ibm.rsar.codereview.java.examples;

import java.util.Iterator;
import java.util.List;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.Block;
import org.eclipse.jdt.core.dom.MethodDeclaration;

import com.ibm.rsaz.analysis.codereview.java.AbstractCodeReviewRule;
import com.ibm.rsaz.analysis.codereview.java.CodeReviewResource;
import com.ibm.rsaz.analysis.codereview.java.IRuleFilter;
import com.ibm.rsaz.analysis.codereview.java.ast.ASTHelper;
import com.ibm.rsaz.analysis.codereview.java.rulefilter.MethodNameRuleFilter;
import com.ibm.rsaz.analysis.codereview.java.rulefilter.ParameterCountRuleFilter;
import com.ibm.rsaz.analysis.codereview.java.rulefilter.ReturnTypeRuleFilter;
import com.ibm.rsaz.analysis.core.history.AnalysisHistory;

public class RuleFinalizersSuper extends AbstractCodeReviewRule
{
    // Rule filters
    private static final IRuleFilter[] MIFILTERS = {
        new MethodNameRuleFilter( "finalize", true ),
        new ReturnTypeRuleFilter( "void", true ),
        new ParameterCountRuleFilter( 0, true )
    };

    /**
     * Analyze this rule
     * @param history A reference to the history record for this analysis
     */
    public void analyze
    ( final AnalysisHistory history, final CodeReviewResource resource ) {
        List list =
            resource.getTypedNodeList( resource.getResourceCompUnit(),
                ASTNode.METHOD_DECLARATION );
        ASTHelper.satisfy( list, MIFILTERS );

        for( Iterator it = list.iterator(); it.hasNext(); ) {
            MethodDeclaration md = (MethodDeclaration)it.next();
            Block block = md.getBody();

            if( md.getBody().statements().size() == 1 ) {
                List superList =
                    resource.getTypedNodeList( block, ASTNode.SUPER_METHOD_INVOCATION );

                if( superList.size() > 0 ) {
                    resource.generateResultsForASTNode( this, history.getHistoryId(), md.getName() );
                }
            }
        }
    }
}
```

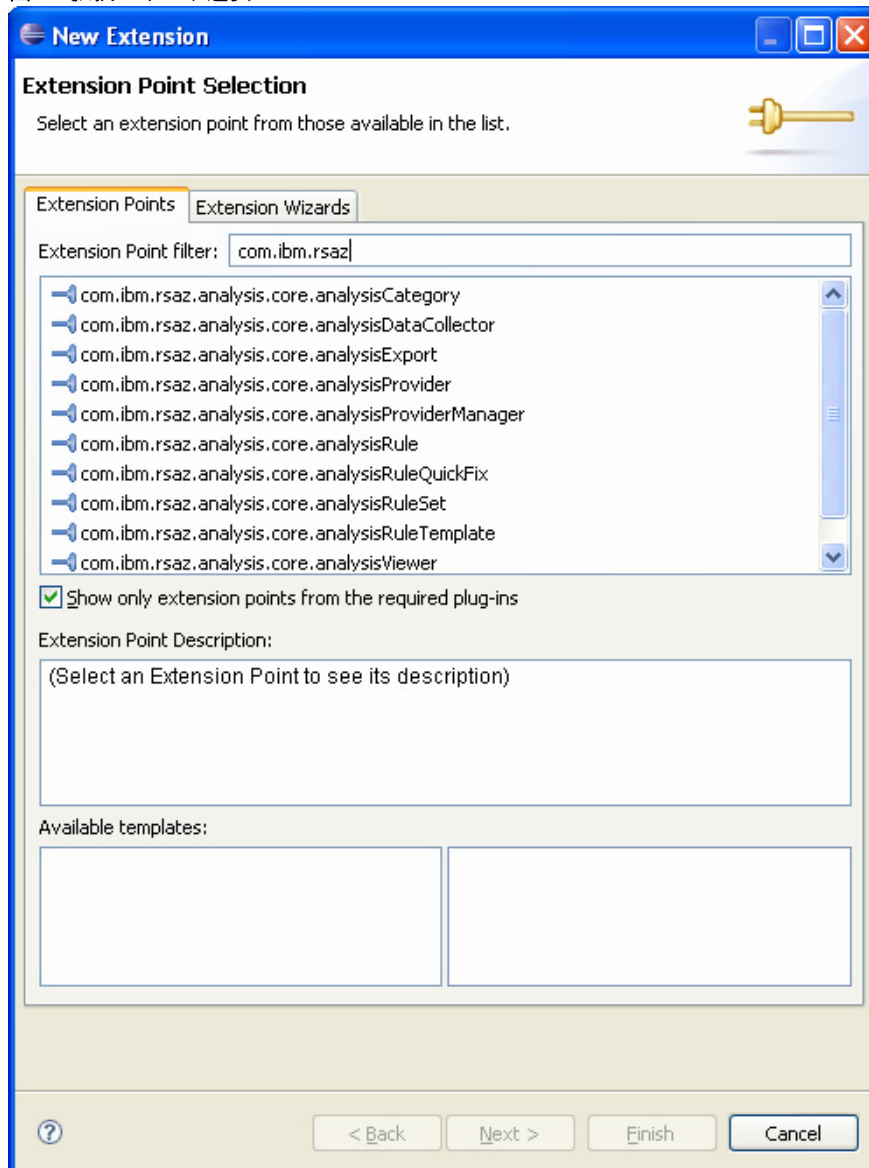
拡張ポイントを作成する

ルールクラスが完成したら、コード・レビューエンジンがルールを発見できるように解析フレームワーク内で提供される拡張ポイントを通して記述する必要があります。この例では、まず独自のカテゴリを作成し、ルールをそのカテゴリと関連付けます。

新しいカテゴリを作成することは比較的単純なタスクです。

1. このセクションの最初で作成したプラグインのマニフェストを開きます。
2. エディターの **Extensions** タブを選択し、**Add** ボタンをクリックします。
3. リストから図 4 のように **com.ibm.rsaz.analysis.core.analysisCategory** 拡張ポイントを選択します。

図 4. 拡張ポイント選択ビュー



4. **Extension** タブに新しい拡張ポイントが追加されます。右クリックし、新しい **analysisCategory** を追加します。**Extention** フィールドを下記のように指定します。(図 5 を参照してください)
- **id*** : analysis.codereview.java.examples
 - **label*** : Examples
 - **category** : codereview.java.j2sebestpractices
 - **class** : com.ibm.rsaz.analysis.core.categoryDefaultAnalysisCategory
(ブラウザボタンを使って選択します)

図 5. Extension Element Details ビュー

Set the properties of "analysisCategory". Required fields are denoted by "*".

id*: analysis.codereview.java.examples

label*: Examples

icon:

provider:

category: codereview.java.j2sebestpractices

class: com.ibm.rsaz.analysis.core.category.DefaultAnalysisCategory

viewer:

help:

このカテゴリーは親カテゴリー (codereview.java.j2sebestpractices) を持っています。それは Java™ 2 プラットフォーム内でネストされ、ユーザーインターフェース上では **Special Edition Best Practices** カテゴリーとして表示されます。クラス **DefaultAnalysisCategory** はユーザーが作る可能性があるほとんどのカテゴリーで必要とされる機能を提供するために予め定義されています。

結果として以下のテキストが **plugin.xml** に挿入されます。

```
<extension
  point="com.ibm.rsaz.analysis.core.analysisCategory">
  <analysisCategory
    category="codereview.java.j2sebestpractices"
    class="com.ibm.rsaz.analysis.core.category.DefaultAnalysisCategory"
    id="analysis.codereview.java.examples"
    label="Examples">
  </analysisCategory>
</extension>
```

次に、新しいカテゴリにルールを追加する必要があります。

5. カテゴリと同様に新しい拡張をプラグインに追加します。今回は `com.ibm.rsaz.analysis.core.analysisRule` 拡張点を選択します。
6. **Extensions** タブでルール拡張を右クリックし新しい **analysisRule** を追加します。
7. ルールの詳細を指定します。（図 6 を参照してください）
 - **id*** : `codereview.java.example.finalSuper`
 - **label*** : スーパーメソッドのみを呼び出す `finalize()` メソッドを避ける
 - **category** : `analysis.codereview.java.examples`
 - **class** : `com.ibm.rsar.codereview.java.examples.RuleFinalizerSuper`
(ブラウズボタンを使って選択します)
 - **severity** : 2 (ドロップダウンメニューを使って選択します)

図 6. analysisRule のプロパティ

Set the properties of "analysisRule". Required fields are denoted by "*".

| | |
|---------------|--|
| id*: | <input type="text" value="codereview.java.example.finalSuper"/> |
| label*: | <input type="text" value="Avoid finalize() methods that call only super"/> |
| category*: | <input type="text" value="analysis.codereview.java.examples"/> |
| icon: | <input type="text"/> |
| class: | <input type="text" value="com.ibm.rsar.codereview.java.examples.RuleFinalizerSuper"/> <input type="button" value="Browse..."/> |
| viewer: | <input type="text"/> |
| severity: | <input type="text" value="2"/> <input type="button" value="v"/> |
| help: | <input type="text"/> |
| quickFixIcon: | <input type="text"/> |

8. ラベルと説明を入力します。（ユーザーがルールを参照するときに表示されます）

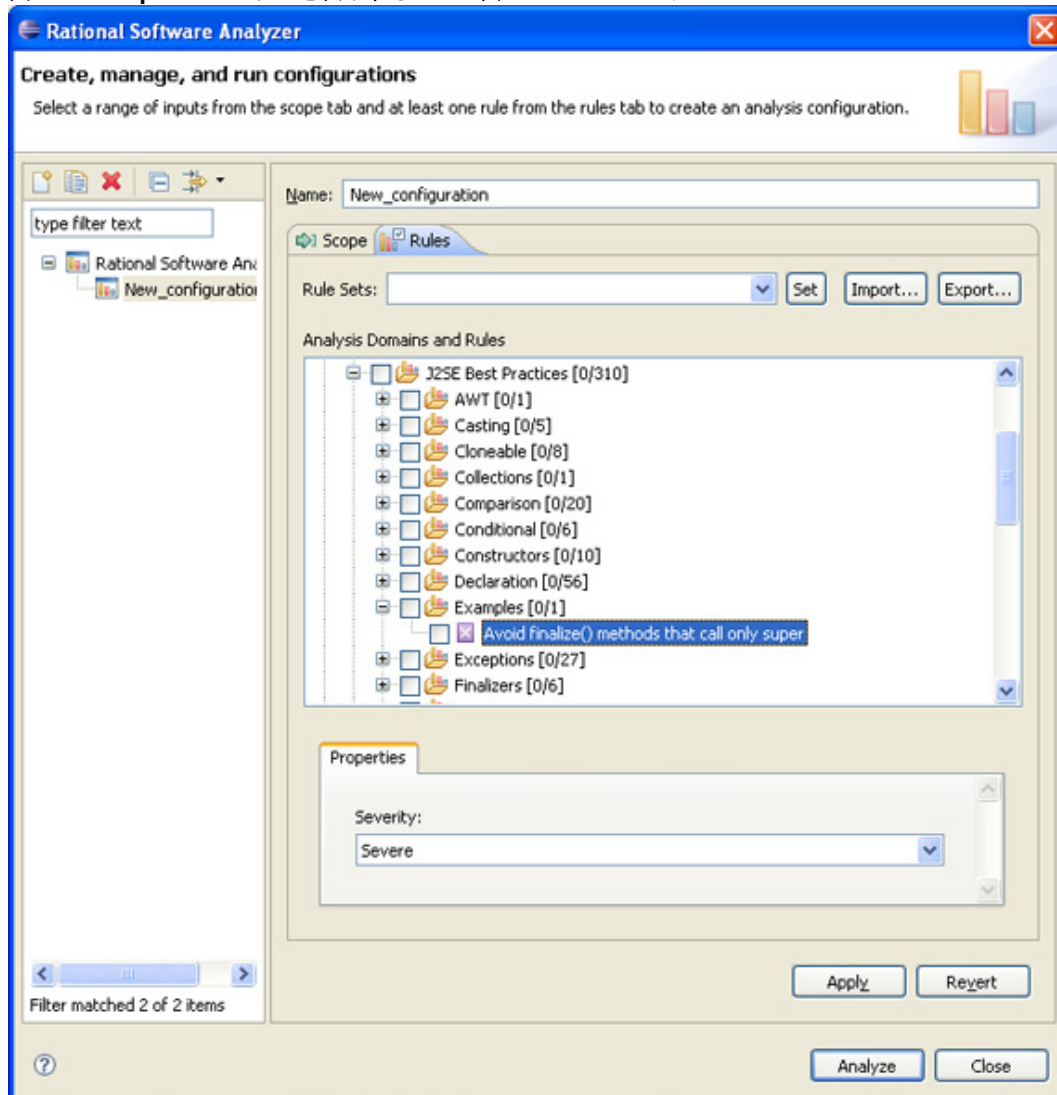
`class` フィールドには作成したルールを指定します。`Category` フィールドには `Examples` カテゴリを指定します。結果として `plugin.xml` には以下のようなテキストを含みます。

```
<extension
  point="com.ibm.rsaz.analysis.core.analysisRule">
  <analysisRule
    category="com.ibm.rca.codereview.java.examples.RuleFinalizeSuper"
    id="codereview.java.example.finalSuper"
    label="Avoid finalize() methods that call only super"
    severity="2">
  </analysisRule>
</extension>
```

ルールの実行

1. Eclipse ビューから **Run** オプションを選択し、新規 Eclipse アプリケーションを生成します。
2. 構成パネルで **Run** ボタンをクリックします。ルールをテストするための新しいランタイムワークベンチが呼び出されます。
3. ワークベンチが表示されたら、レビュー対象となるコードを含むいくつかの Java ソースコードをインポートします。ルールをテストできるように、**super.finalize()**のみを呼び出している **finalize()**メソッドを探します。
4. ソースコードがワークベンチにある状態で、Eclipse の Run メニューから **Analysis** オプションを選択します。
5. ダイアログで新規分析構成を作成し、**Domains** タブを開きます。
6. **JavaCodeReview** ブランチを **Example** サブカテゴリーが表示されるまで開きます。（図 7 を参照してください）

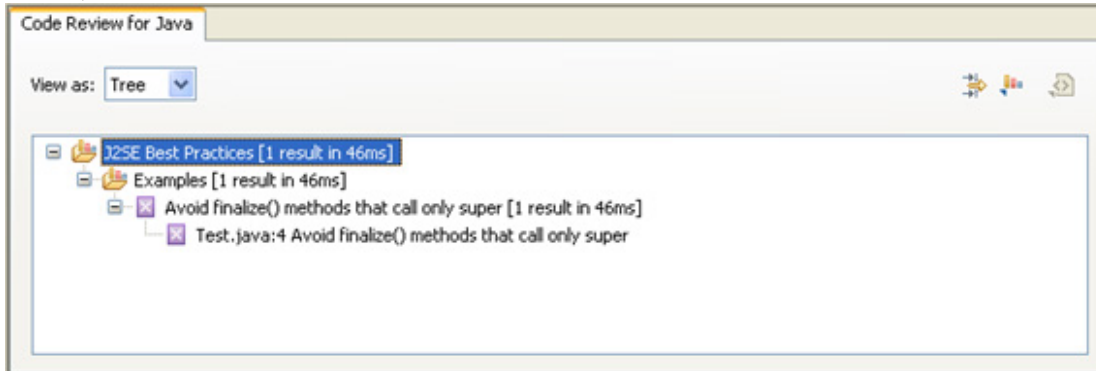
図 7. Example カテゴリーを表示するために開いた Java コード・レビュー



7. 新しいルールが選択された状態でコード・レビューを開始するために **Analyze** をクリックします。

もしワークベンチ内にルールに合致する **finalize** メソッドを持つソースコードがあれば、図 8 のように解析結果のビューが表示されます。

図 8. 解析結果のビュー



再びルールフィルターについて

このセクションでは、ルール、もっと具体的に言えばルールがルールフィルターをどう使うかについて見てきました。API はあなたが書きたいと思うルールのほとんどをサポートするルールフィルターを提供しています。以下は **Rational Software Analyzer** が提供するルールフィルターのリストです。（個々のルールフィルターの使い方に関する情報は解析 API の **Javadoc** にあります）

- ArgumentTypeRuleFilter
- ConstructorRuleFilter
- DeclaringClassRuleFilter
- EnclosingNodeRuleFilter
- ExceptionCountRuleFilter
- ExpressionRuleFilter
- ForInitializerCountRuleFilter
- ForUpdateCountRuleFilter
- FragmentCountRuleFilter
- IfElseStatementCountRuleFilter
- IfElseStatementRuleFilter
- IfThenStatementCountRuleFilter
- IfThenStatementRuleFilter
- ImplementedInterfaceRuleFilter
- LeftOperandRuleFilter
- MethodNameRuleFilter
- ModifierRuleFilter
- OperatorRuleFilter
- ParameterCountRuleFilter
- ParameterTypeRuleFilter
- ReturnRuleFilter
- RightOperandRuleFilter
- SuperClassRuleFilter
- TypeRuleFilter

また、論理演算に関する 2 つのルールがあります (LogicalOrFilter と LogicalAndFilter)。これらのルールを使うことで、似たようなフィルターの配列を比較しなければならないルールの重複を避けることができます。例えば、

```
private static final IRuleFilter[] MIFILTERS = {
    new MethodNameRuleFilter( "testMethod", true ),
    new ReturnRuleFilter( "void", true ),
    new LogicalOrFilter(
        new ParameterCountRuleFilter( 0, true ),
        new ParameterCountRuleFilter( 1, true )
    );
};
```

この、testMethod()と名づけられたコードフィルターメソッドは void を返し、パラメーターに 0 か 1 を含むことを指定します。

必要なルールフィルターがないことはあり得ます。しかし独自のルールフィルターを追加することはさほど難しくありません。API はフィルターシステムにプラグインを実装可能なようにインターフェース (IRuleFilter) を提供しています。リスト 4 は Rational Software Analyzer の Java コード・レビュープロバイダーで提供されているルールフィルターのサンプルです。

リスト 4. Rational Software Analyzer に含まれるルールフィルターの例

```
public class SuperClassRuleFilter extends AbstractRuleFilter {
    private static final String SATISFIES_SUPER_CLASS = "satisfiesSuperClass";
    private String superclassName;

    /**
     * Constructor
     *
     * @param superclassName
     *        The super class name by which to filter
     * @param inclusive
     *        True if filtering will include only nodes that match the
     *        filter criteria, false to exclude matching nodes
     */
    public SuperClassRuleFilter( String superclassName, boolean inclusive ) {
        super( inclusive );

        this.superclassName = superclassName;
    }

    /**
     * Determine if the node is satisfied by the specified filter rule
     *
     * @param node
     *        The ASTNode to test
     * @return
     *        true if the node satisfies the filtering rule
     */
    public boolean satisfies( ASTNode node ) {
        try {
            if (node.getNodeType() == ASTNode.TYPE_DECLARATION) {
                // Get the super class type. It is null if this type
                // declaration does not extend any type
                Type superClassType =
                    ((TypeDeclaration)node).getSuperclassType();
                if (superClassType != null) {
                    return superClassType.resolveBinding().getQualifiedName()
                        .equals( superclassName );
                }
            } else {
                Log.severe (Messages.bind(Messages.RULE_FILTER_ERROR_,
                    new Object[]{ SATISFIES_SUPER_CLASS,
                        node.getClass().getName()}));
            }
        } catch (NullPointerException e) {
            // Do nothing
        }

        return false;
    }
}
```

- ・ ノードの評価に必要な値を受け取るコンストラクターを作る必要があります。ここに示したサンプルでは、コンストラクターはスーパークラスの名前を表現する文字列を受け取ります。

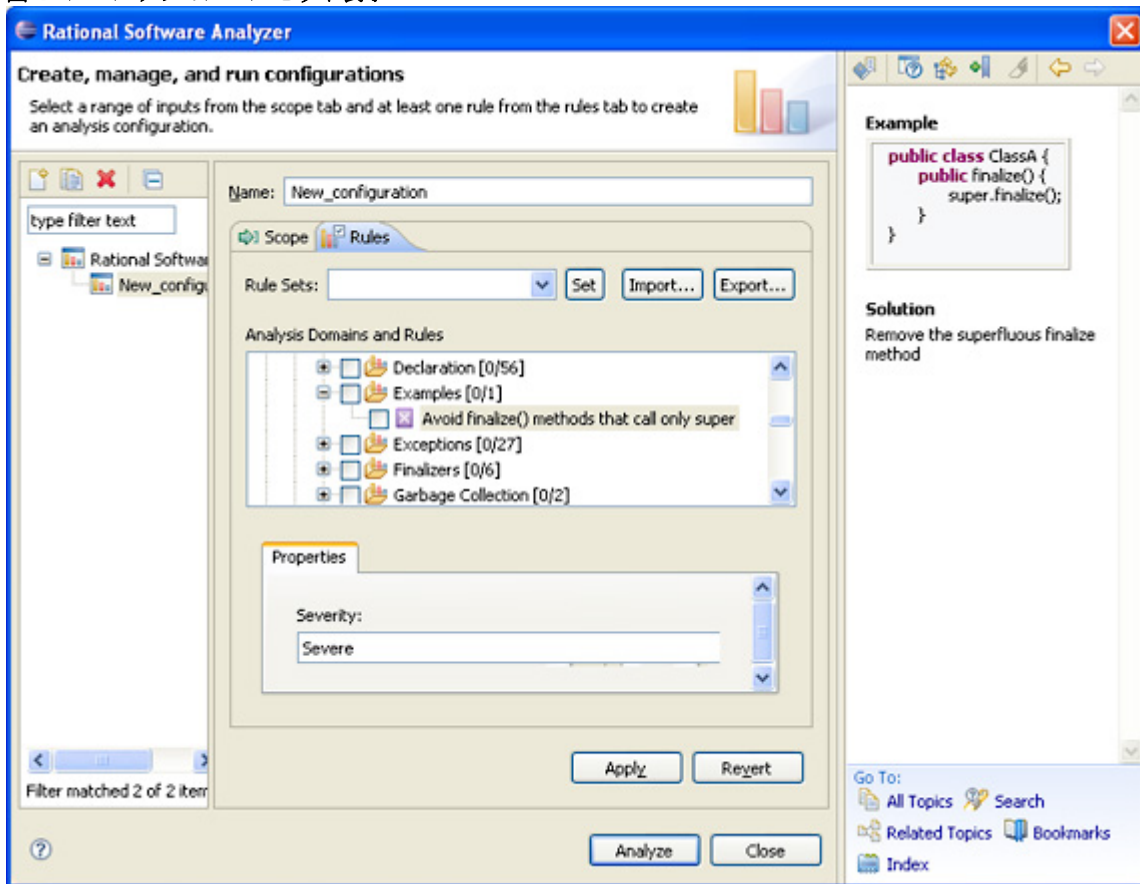
Tip:

インクルード、エクスルードをサポートするために常に **Boolean** パラメーターを受け取ってください。

- ・ テスト対象の **AST** ノードを解釈する **satisfy()** メソッドを実装する必要があります。
- ・ ルールフィルターを信頼できるので、すべての例外は **satisfy()** 内でキャッチできます。ここに示したサンプルでは、コードのバインディングに失敗した場合のために **NullPointerException** をキャッチしています。

ルールフィルターができたなら、他の予め定義されたルールフィルターのように、**IRuleFilter** 配列の中に入れておき、**ASTHelper.satisfy()** をルール内で呼び出すことで使用することができます。

図 9. ルールタブのルールセット表示



要約と次回内容

このチュートリアルでは **Java** の新しいコード・レビュールールを書いて、解析 API の内部動作を説明しました。カテゴリとルールスペックを **plugin.xml** ファイルに作成するコア・コンセプトについて議論し、基本的なルールを作成するためのクラスの書き方について記述しました。次の記事ではルール重要度、カスタムルールパラメーター、ルールテンプレートの作成の概念についてカバーします。

About the author



Steve Gutz is a development manager responsible for IBM Rational's code analysis tools. In addition architecting and managing IBM's commercial products Steve is also a past contributor on the Eclipse Test and Performance Tools project (TPTP), focusing on improvements in implementation and integration of basic code review tools. Previous to joining the IBM Ottawa Lab in 2002, he held senior management and executive positions in several public and private companies including two of his own successful start-ups. He is also the author of two books and many articles, and is a regular conference speaker.