

# Rational Software Analyzer による静的解析

## Part 1 – 静的解析を始めよう

Level: 初級

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 1 番目のものです。ここでご紹介する静的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、静的解析のハイレベルな概念にフォーカスして説明します。本シリーズ中の他の記事では、いくつかの分析機能の使い方および拡張方法について解説します。この製品の全ての機能を使いこなしたい方に中、上級レベルの機能を学習する助けになります。

原文 URL:

[http://www.ibm.com/developerworks/rational/library/08/0429\\_gutz1/](http://www.ibm.com/developerworks/rational/library/08/0429_gutz1/)

テクノロジー業界で働く人なら誰でも、自分たちを最高の開発者だと考えるもので、ほとんどの部分においてそれは正しいです。しかしながら、どれだけの教育や訓練を受け、集中して作業しても、私たちはバグのあるコードを書き続けます。ほとんどのソフトウェア・アプリケーションはそれほどに複雑になりました。要求を理解し、それに合わせて意図しない動作やバグがないソースコードを書くことは、事実上不可能とも言えます。

あなたの組織のミッション・クリティカルなアプリケーションのことを考えてみてください。数百のクラスや数千行ものコードで作られていることでしょう。そのような複雑なシステムを理解し、実装するのを助けるため、この業界で我々はアジャイル開発プロセスのような新しいテクニックを作り上げました。短い期間の反復型開発とピアコードレビューをもってしても、私たちが顧客に納品するソフトウェアには依然として多くの欠陥が潜んでいることでしょう。現在の開発プロセスは十分なものではないのです。だんだんと静的解析ツールの助けが必要になってきているのです。

私たちは何十年も動的開発ツールを使ってきました。誰もがランタイム・デバッガーでコードをデバッグし、コードプロファイラーでパフォーマンスボトルネックを発見してきました。しかしこれらのツールは修正にコストがかかる開発サイクルの遅い時期にしか使用できません。問題を発見するのに最適なタイミングは、まさにソースコードをレビューする時なのです。静的解析ツールを使うことにより、多くの手戻り作業の原因となる問題を自動的に取り扱うことができます。ただし、静的解析ツールはコード品質改善のための別手段であるわけではないことを心にとどめてください。手作業のコード・レビューの完全な代替となるものではありません。

この記事は、**IBM® Rational® Software Analyzer** という静的解析ツールを紹介します。目的は、自動化されたコード解析について易しく解説し、ソフトウェア開発プロセスにもたらすメリットを説明することです。

**Rational Software Analyzer** はコード品質の改善を自動化するプロセスを促進、単純化するために設計されています。最初は他の **Rational** ツール (**Rational® Application Developer** や **Rational® Software Architect** のような) に静的解析機能を統合するための **API** とユーザー・インターフェースでしたが、現在では単独で完全な機能を提供する製品へと進化を遂げました。**Analyzer** は開発者が自動化されたコード・レビューや構造解析といった機能を日々の開発プロセスに取り込むことを容易にします。

## 我々が“静的解析”によって意味するもの

静的解析の解釈は人によってさまざまです。もしあなたが製品の観点に立てば、何十もの会社が静的解析ツールを販売しているのに気づくことでしょう。静的解析の概念がとてつもなく広範囲であるためにマーケットには多くの会社が参入しているのです。ある会社は **C++** のコード・レビューのみにフォーカスし、一方で他の会社は **Java** 開発言語のソフトウェア・メトリックのみを提案します。**Web** アプリケーションのセキュリティ問題のみを解析する、依存問題のためだけにソースコードをスキャンする、などなど。このように、静的解析は多様かつ混乱した、分類が必要なコンセプトとなっています。

それでは、静的解析とは何でしょうか。*静的解析* は、変化していないものの研究を意味します。しかし、ソフトウェア用語では、この定義はソースや現在動作していないバイナリー・コードを研究するものとして洗練することができます。コードを動作させて分析するためにはデバッガーやプロファイラーが必要です。しかし、コードを動作させず、あるいはコンパイルさえしていなくても、あなたはコードから多くのことを学ぶことができます。

例えば、プログラムのための全てのソース・ファイルを単純に構文解析すれば、ソースコードが予め定義されたコーディング標準に準拠しているかどうかを確認することができます。また、結果に変更がないのに同じメソッドを何回も呼んでいるような、よくあるパフォーマンス問題についても検出することができます。

多くのタイプの静的解析がありますが、提供する価値に基づいていくつかの共通のカテゴリーに分割することができます。もちろん、他にも多くの種類の静的解析が存在しますが、表1は **Rational Software Analyzer** に関する一連のこのシリーズの記事の主題である静的解析の主要なタイプと様式を捉えています

表1. 静的解析のカテゴリー

Type	Value
コード・レビュー	このタイプのツールはコードの構文解析を自動化し、所定のルールセットに違反するコード・パターンを探するという典型的なものです。 <b>C++</b> のような言語では、多くのルールがコンパイラーに組み込まれ、あるいは <b>Lint</b> のような外部プログラムとして利用可能です。 <b>Java</b> のような言語では、コンパイラーは自動化されたコード・レビューの役目をほとんどしません。
コード依存関係	個々のソース・ファイルのフォーマットを調査するのではなく、コード依存関係ツールはプログラムの全体的なアーキテクチャーのマップを作るためにクラス間の関係を調査します。コード依存関係ツールは主に、広く知られている良いデザイン・パターンと悪いアンチ・パターンをコードから発見します。
コードの複雑さ	複雑さツールはプログラムを解析し、所定のソフトウェア・メトリックと比較して不必要に複雑でないかどうかを判定します。もしコードの特定の部分がしきい値を超えれば、メンテナンス性を改善するためのリファクタリングの候補として識別され得ます。
トレンド	トレンド分析は直接コード成果物を使用しません。むしろ、他の分析に基づき、コード品質の改善やデグレードを調べます。（特に他の分析の結果を分析します。）これらのツールから作成される結果は品質改善の方向性を示しており、「コードは良くなっているのか悪くなっているのか？」という説明に答えられるので、開発者よりもむしろマネージャーやエグゼクティブや顧客によりアピールします。

## 静的解析のメリット

この記事では既に、開発プロセスに静的解析を取り入れたほうがよい理由について触れました。繰り返すと、2つの基本的で注目すべき理由があります。時間を節約することとお金を節約することです。静的解析ツールによる時間の節約はとて明確でしょう。少ない時間でよりよい品質のコードを得ることができます。IBM 社内での使用も含む多くの事例は、単純なコード・レビューの自動化だけでも、コードの欠陥のうち5~10%を発見できることを示唆しています。

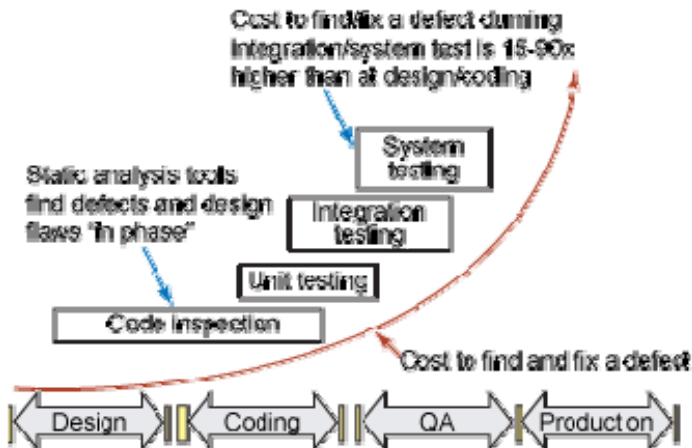
同じ事例は、顧客から報告される1つの欠陥は、1万2千~1万8千 US ドルのコストを消費し得ることを示唆しています。ライフサイクルで数千もの欠陥を持つ典型的な大きなソフトウェアを考えると、自動化されたコード・レビュー・ツールが270万 US ドルのうち60万 US ドルを節約できることがわかるでしょう。

こうしたパーセンテージやコストをあなたが信じるかどうかにかかわらず、静的解析ツールによるコスト削減のポテンシャルは驚異的なものです。

確かに、顧客によって報告される欠陥の減少はコストを削減する最も明らかな手段であり、一般的には包括的なテスト・プロセスにより達成します。しかしながら、コード・レビューのような静的解析ツールはさらにいっそうコストを削減する方法となるでしょう。

私たちは皆、図1に示したようなグラフを見てきました。その理論的根拠について異論を唱える人はほとんどいないでしょう。開発プロセスのより早期に欠陥を発見することはコストを抑えます。そして、シンプルで自動化されたコード・レビューを使用することによって、プロジェクトのコーディング・フェーズから---おそらくは、開発者がコードをタイプしている間にさえ---欠陥を探し始めることができます。

図1. 開発ライフサイクルの異なる段階で発見された欠陥の修正コストの比較



ですが、明白とはいえないかもしれない節約の別の面があります。これまでのところは、我々は、開発者が静的解析を使うことによって時間とお金を節約できることに注目しました。しかし、顧客はどうでしょう？ あなたのソフトウェアが顧客に余計な時間またはお金を負担させることになれば、彼らは、喜んであなたを告訴することでしょう。あなたは信頼性が高いという評判によって、顧客にソフトウェアを買って使って欲しいはずです。開発プロセスに静的解析を追加すれば、顧客も時間とお金のアドバンテージを得ます。高品質なコードは、顧客がレポートした欠陥を、修正されるまで待っている時間を無駄にしないことを意味します。さらに言えば、顧客の収益力は待っている間にも妨げられます。

**Rational Software Analyzer** は下記のような要求を満たすようにデザインされています。

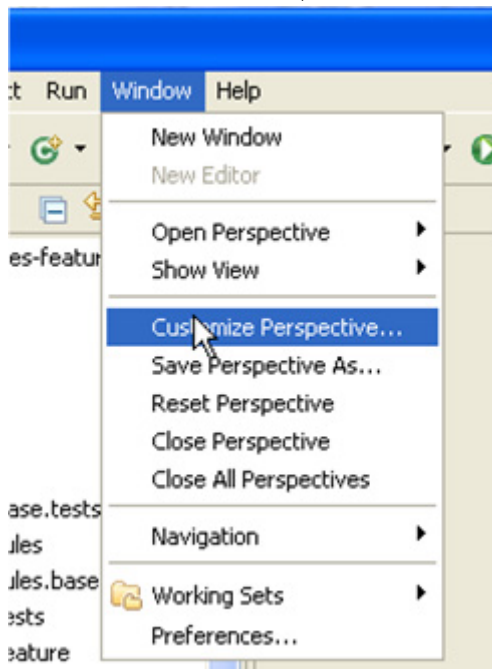
- 第一に、**Eclipse** や、**Rational Software Architect**・**Rational Application Developer** のワークベンチとタイトに統合すること。これにより、開発者はコードを書いている時に静的解析機能にフルアクセスすることができます。
- 第二に、既存のビルドシステムと統合できるよう、コマンドラインまたは **ANT** タスクとして利用できること。完全な **API** が、ビルトインの解析だけではなく、独自の利用方法にも対応します。  
(※**Enterprise Edition** のみ)
- 最後に、解析結果を抽出してワークベンチ内と **HTML** のようなエクスポートされたフォームでレポートを生成できること。これにより、開発者、マネージャー、エグゼクティブがコード品質の全てを評価できるようになります。

## 解析を実行するためのルールを定義する

ワークベンチから Rational Software Analyzer を使えるようになったら、Java, Debug, C++, Plugin 開発パースペクティブに新しいメニューとツールバーオプションが追加されます。他のパースペクティブでは、手動でフィーチャーをえるようにする必要があります。

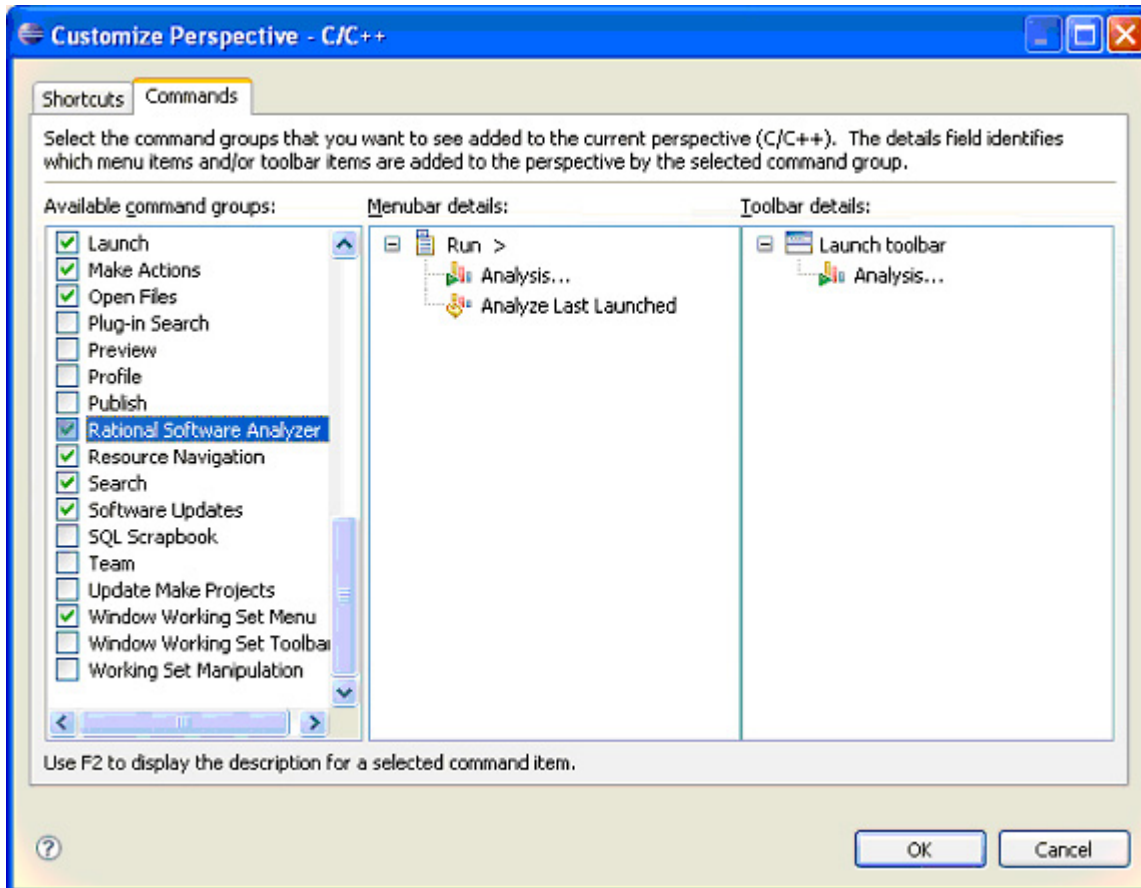
1. Eclipse のメニューバーから **Window > Customize Perspective** を選択する (図 2 を参照)

図 2. パースペクティブオプションのカスタマイズ



2. ダイアログが表示されたら、**Commands** タブを選択し、Rational Software Analyzer チェックボックスをクリックする。
3. **OK** をクリックする。 (図 3)

図 3. Commands タブ



Eclipse のツールバーとメニューに静的解析が追加されます。これにより、分析構成を作成、更新、実行することができます。

#### 4. Run > Analysis メニューを選択してメイン分析構成ダイアログを表示する

Eclipse ワークベンチでコードを実行したりデバッグしたりする際に使用するのと非常によく似たダイアログが表示されます。簡単に使えるようにするため、既に使用しているダイアログに似せて機能をデザインしてあります。ダイアログの左上にあるボタンを使って分析構成を追加したり削除したりすることができます。分析構成は適切な名前をつけて、どのような形態の分析をどのルールを使用してどの範囲（例えば、プロジェクトやワーキングセットやワークスペース全体）に実行するかを決定します。

#### 5. 分析構成を開始するには、分析構成ダイアログの左側にある分析構成リストから **Rational Software Analyzer** を選択し、**New** ボタンをクリックします。

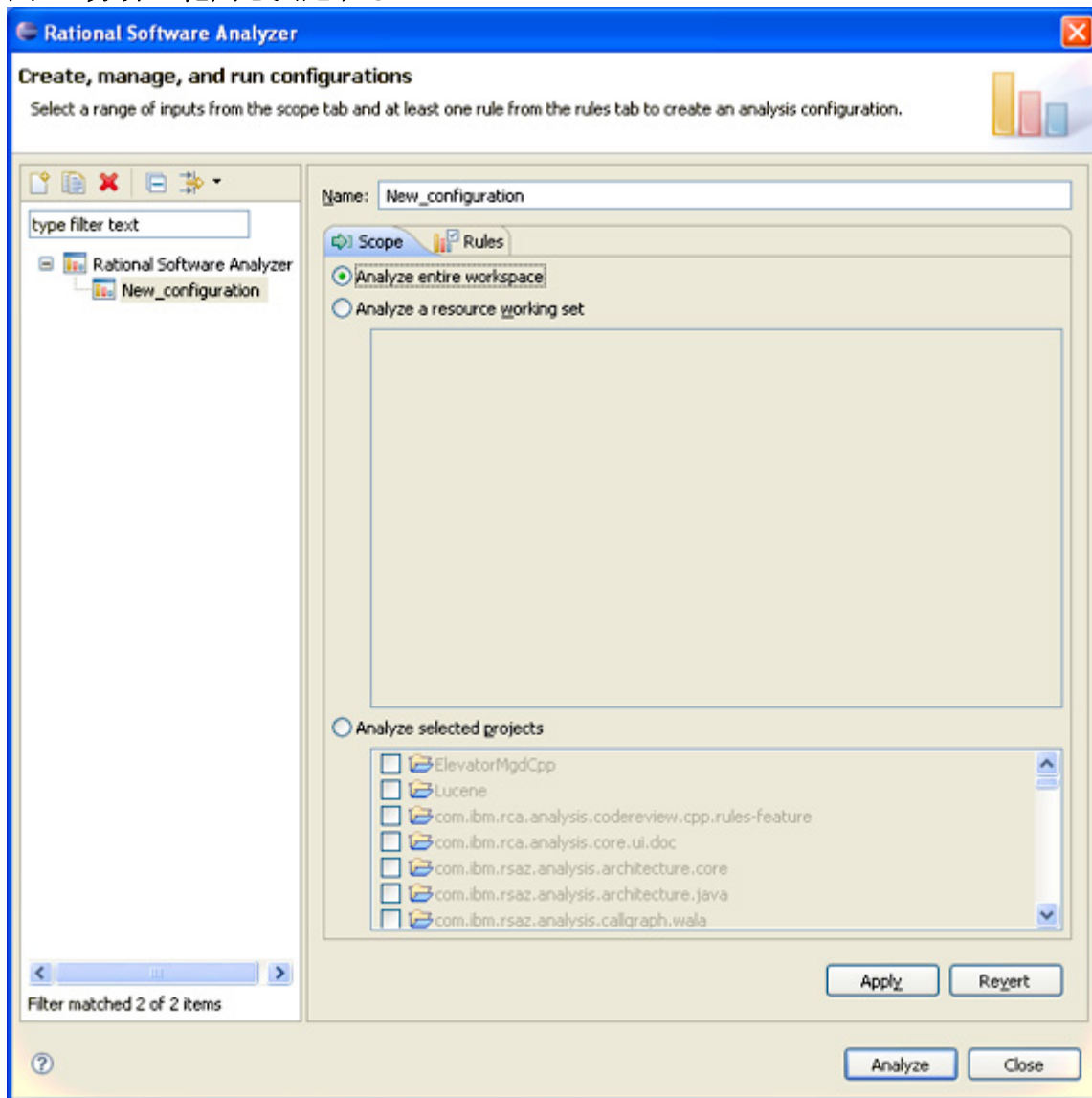
ダイアログの右側の画面が基本的な構成を行う画面に変わります。

## 分析を構成する

分析構成を作る最初の一步は分析対象となるデフォルトのリソース範囲を決定することです。スコープタブから希望する範囲を選択することができます。選択可能なオプションは、ワークスペース全体、ワーキングセット、プロジェクトのセットのいずれかです。

1. 新しい構成を作成し、図 4 のように、スコープタブでワークスペース全体を選択する。

図 4. 分析の範囲を決定する



ルールタブで、実行したい分析の形態を指定します。このタブは分析要素を選択したり選択解除したりできるディレクトリ（ツリー）を表示しています。また、ルール選択の内容をインポートしたりエクスポートしたりするボタンも備えています。ツリーの最上位のノードは解析フレームワークが認識する解析ツールのタイプを表現する分析プロバイダーです。プロバイダーはカテゴリーを含みます。ルールは、解析の結果の状態を定義します。

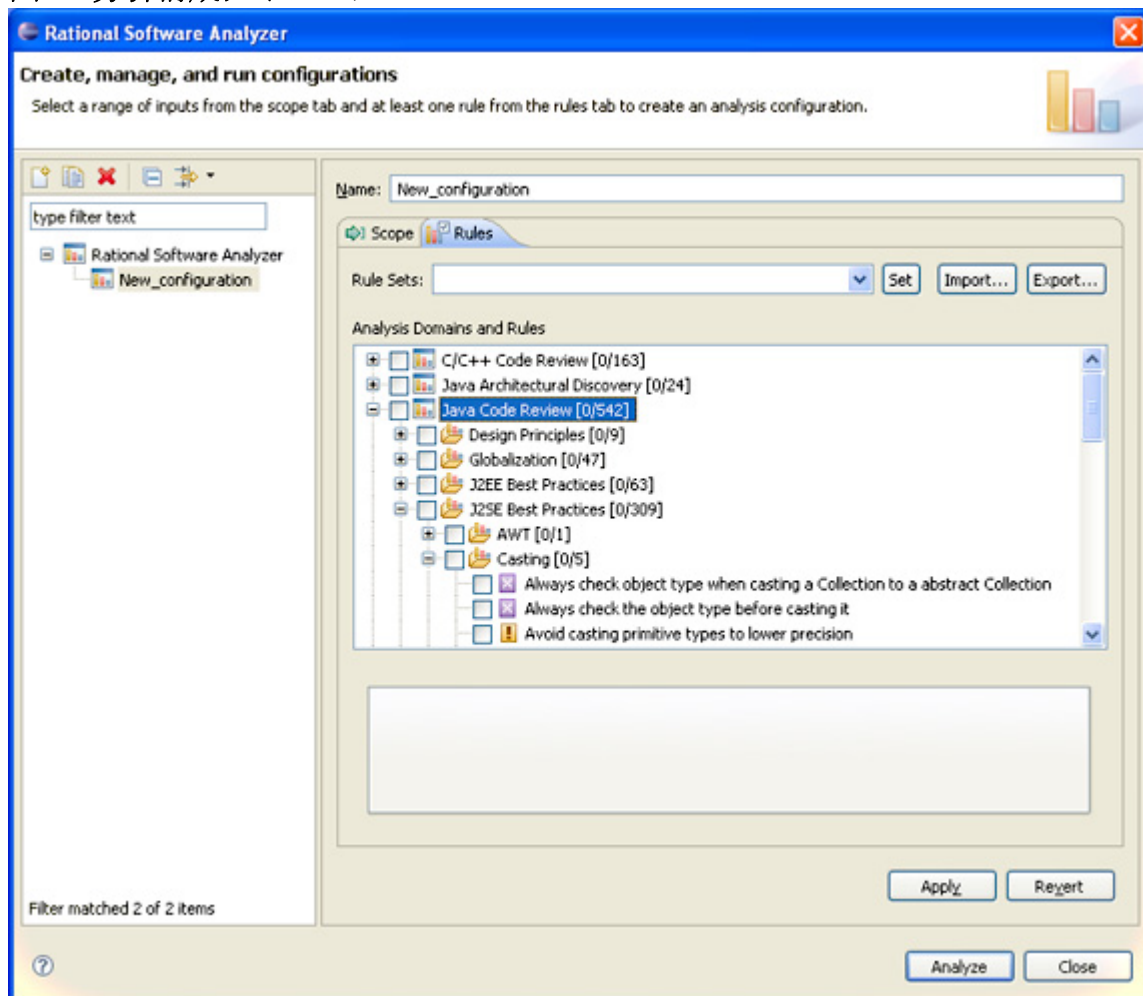
ツリーの各ノードのチェックボックスは有効か無効かを選択できるようになっています。子ノードは親ノードと同じ状態に設定されます。例えば、

## 2. Java コード・レビューブランチ全体を選択する

注：

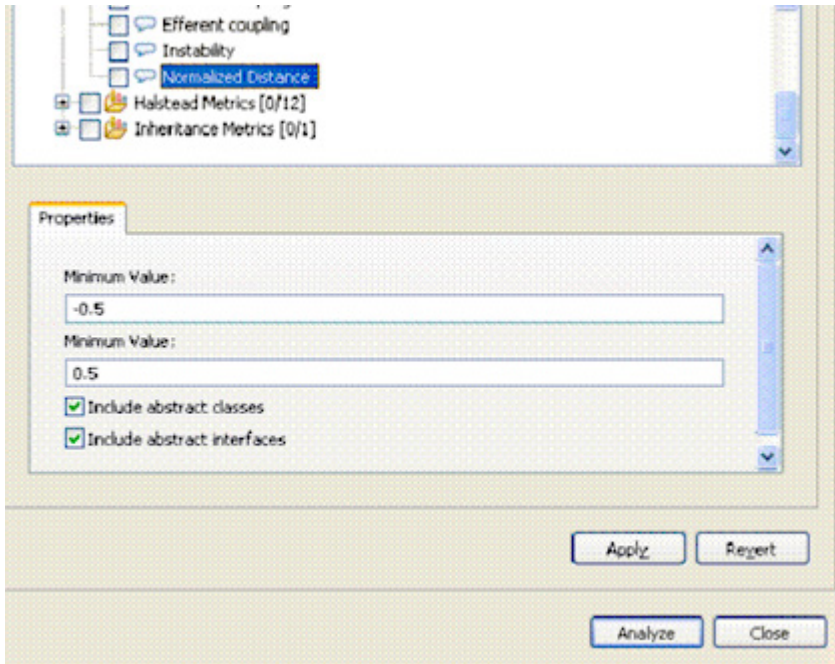
ルール数が図 5 のダイアログに表示されているものと異なっても心配しないでください。解析機能は複数の Rational プロダクトで利用され、含まれるルールは可変です。

図 5. 分析構成ダイアログ



いくつかのルールには追加の構成オプションがあります。このケースでは、ルールタブに表示される最下層のルールに設定項目があります。設定する内容がない場合はこの部分は空白になります。図 6 は Java メトリック・ルールのパラメーターの一例です。

図 6. ルールパラメーターの例



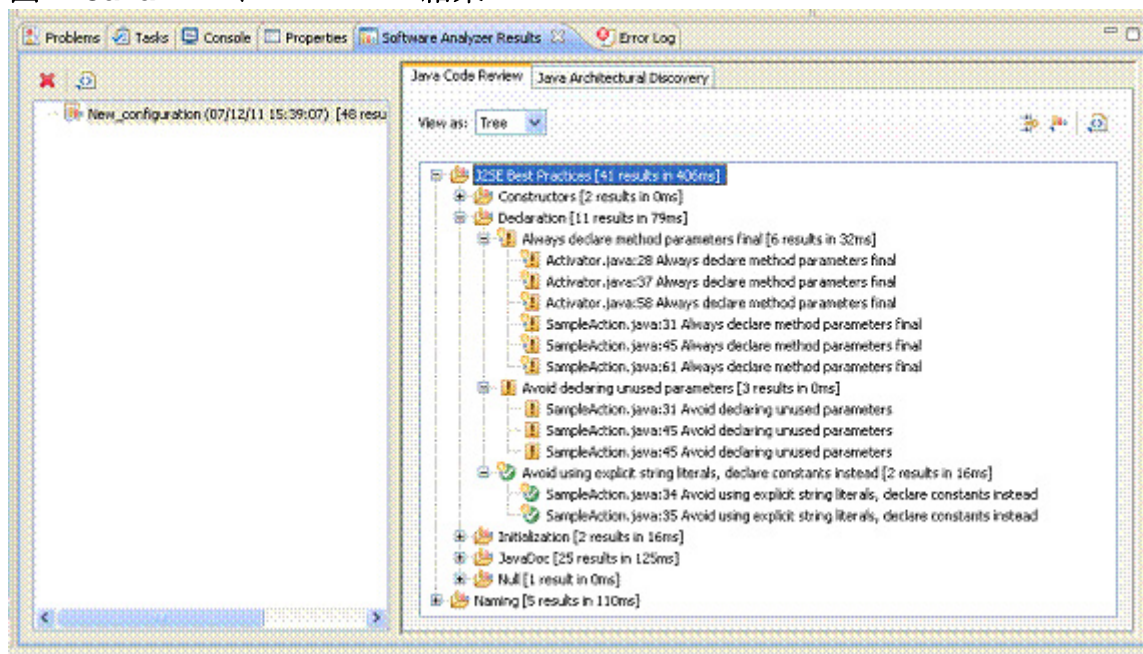
## 解析を実行して結果を表示する

1. 解析を実行するには、**Analyze** ボタンを押します。

Eclipse ワークベンチに解析結果のビューが表示されます。どんな解析を選んだかによって、結果のビューは異なります。例えば、**Java** コード・レビューで提供されるルールであれば、結果は複数のフォーマットで見ることができます。（例えば、テーブル、もしくはツリー）

図 7 にあるように、分析構成が複数の分析タイプを含む場合には（この場合はコード・レビューとアーキテクチャル・ディスカバリー）、結果のビューは分析プロバイダー毎のタブで表示されます。

図 7. Java コード・レビュー結果ビュー

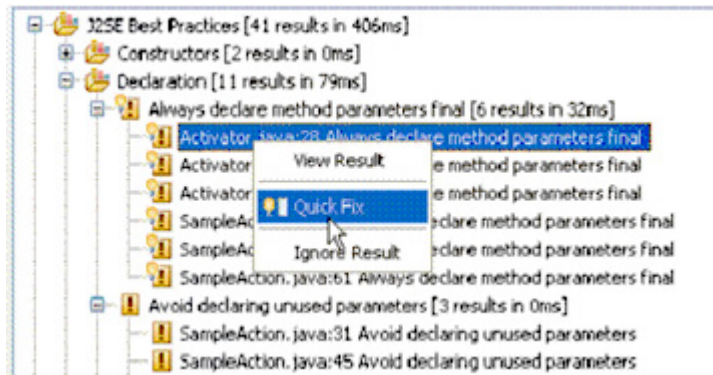


### ヒント:

結果を右クリックすると問題が発生しているソースコードの表示や、クイックフィックスといったタスクを実行することができます。クイックフィックスは、特定のルールに用意される、比較的単純な問題を自動的に修正する機能です。

2. もし、**Quick Fix** メニューオプションが使用可能であれば、選択することで問題を訂正することができます。

図 8. Quick Fix オプション



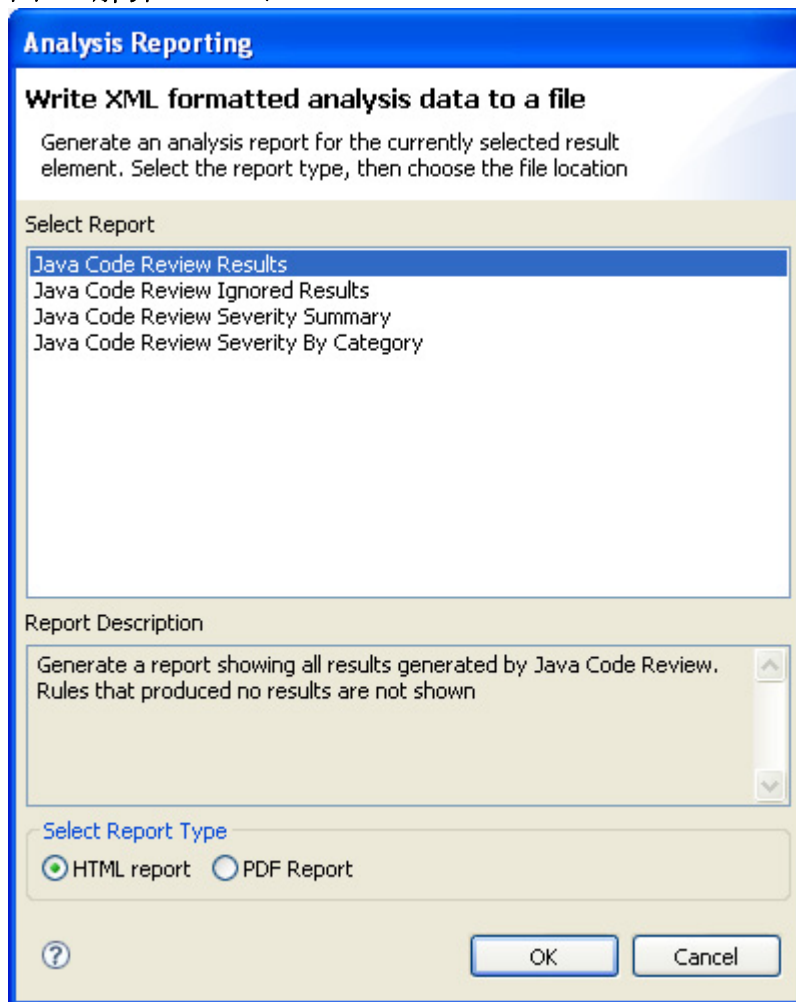
ビューワーが結果を描写するために使用するのはデータタイプ毎の機能であるということに注意してください。結果を参照するとき、エディターでテキストがハイライトされたソースファイルや、UML ダイアグラムや、統計データのテーブルが表示されます。結果を見るのに共通の方法はありません。それは、ルールの記述者によって決定されます。Java コード・レビュープロバイダーでは、全ての結果は編集可能な Java ソース・ファイルとして表示されます。

## エクスポートとレポート

その他に結果ビューで利用可能な 2 つの機能があります。（実行した解析のタイプに依存します） データのエクスポートとレポートです。

データエクスポートによって、解析結果をそのまま XML のようなファイルへエクスポートすることができます。エクスポートされるデータのタイプは解析プロバイダーによって決定されており、データエクスポートのリストとして提供され、1 つを選択して実行することができます。

図 9. 解析レポートビュー



多くの点で、レポートはエクスポートに似ています。実際、両方の機能はエクスポーターを共有します。しかしながら、レポートはローカルやリモート **Web** サイトに格納できるきれいなフォーマットのページを生成します。既存のレポートファイルを取り出して目的に合わせて更新することができます（例えば、会社のロゴを追加するなど）。生成されるレポートは図 10 や図 11 に似ているでしょうが、レポートエンジンはフレキシブルなので他のバリエーションも利用可能です。

## 図 10. Java コード・レビュー

### Java Code Review

May 14, 2007 2:04 PM

#### Design Principles

##### ✔ Avoid methods with more than 3 parameters

```
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/AnalysisLaunchConfigurationDelegate.java:62  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/AnalysisLaunchConfigurationDelegate.java:206  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/Views/AnalysisResultView.java:26  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/Views/ResultTab.java:50  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/Views/ResultViewDefault.java:108
```

##### ✔ Avoid using the negation operator "!" more than 3 times

```
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/Views/ResultsFrameView.java:314
```

#### Globalization

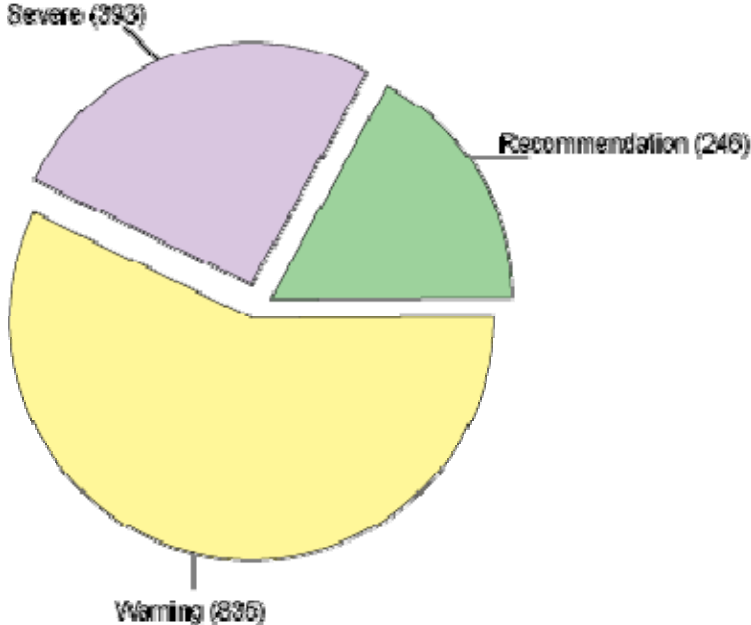
##### ⚠ Avoid using java.lang.String + operator

```
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/UITMessages.java:20  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/actions/AbstractResultAction.java:100  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/actions/AbstractResultAction.java:125  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/dialogs/ExportDialog.java:204
```

##### ⚠ Avoid using java.lang.String.equals() for multilingual strings

```
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/actions/AbstractResultAction.java:87  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/actions/AbstractResultAction.java:112  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/Views/ResultViewDefault.java:220  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/Views/ResultViewDefault.java:346  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/Views/ResultViewDefault.java:355  
/org.eclipse.ttp.platform.analysis.core.ui/src/org/eclipse/ttp/platform/analysis/core/ui/Views/ResultViewDefault.java:361
```

図 11. パイチャートで表示された Java コード・レビューの重要度サマリー

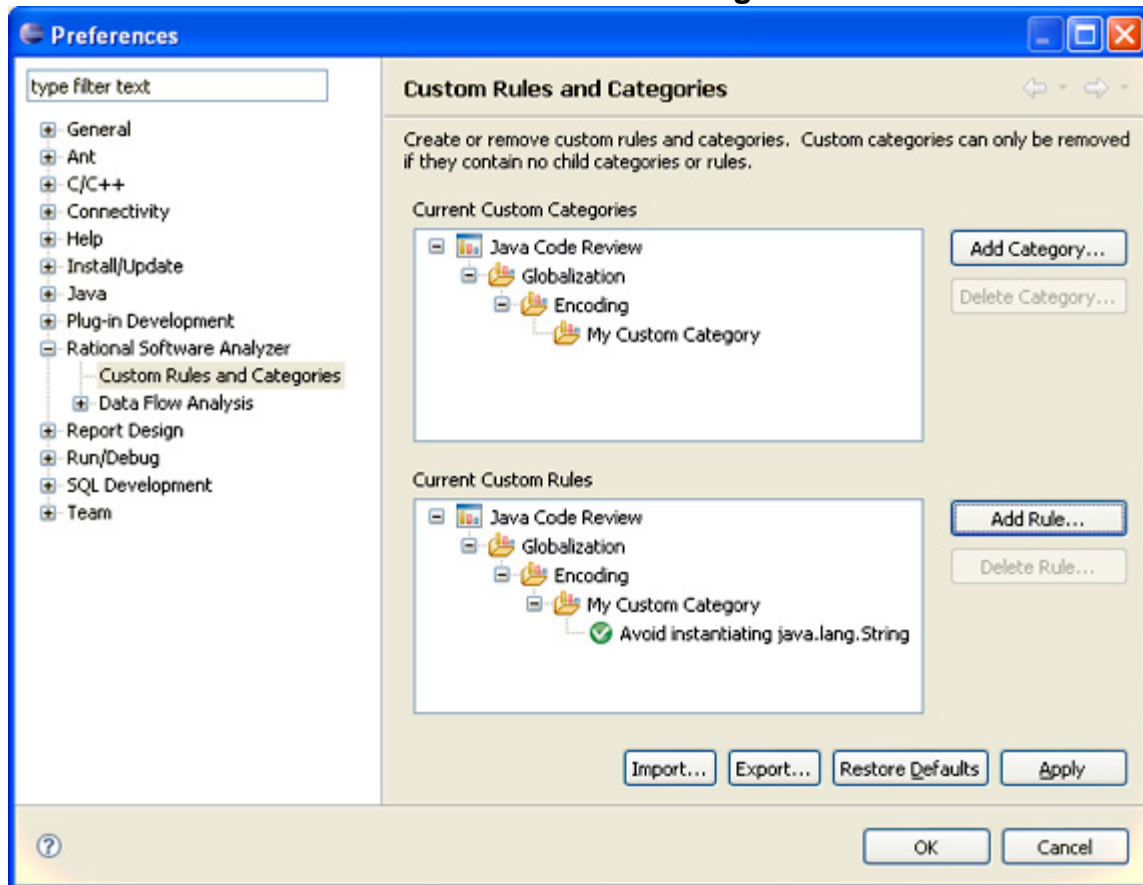


(オプション) カスタムルールとカテゴリの作成

Rational Software Analyzer が提供するルールとサード・パーティが提供するルールに加えて、カスタムカテゴリとテンプレートベースのルールを、一行もコードを書かずに作成できます。新しいカスタムルールとカテゴリを作成するには、

1. **Windows > Preference** を選択して設定ページを表示します。
2. 設定ツリーで **Analysis > Custom Rules and Categories** を選択します。(図 12 参照)

図 12. 設定ダイアログの **Custom Rules and Categories** ビュー



3. 新しいカスタムカテゴリを追加するには、**Add Category** をクリックします。
4. ルール作成ウィザードを開始するために **Add Rule** をクリックします。

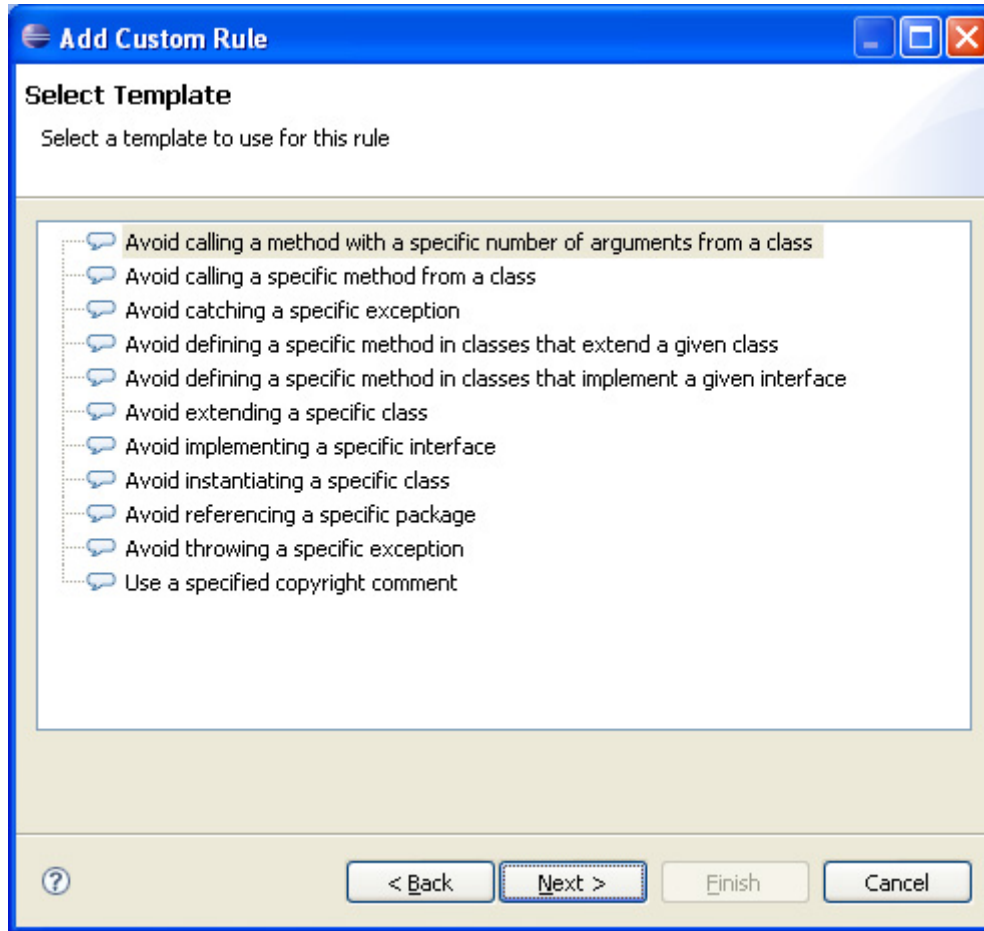
ウィザードの最初の画面でルールをどのカテゴリ・ツリーに配置するか指定できます。

5. ウィザードの最初のページでカテゴリを選択して **Next** をクリックします。

6. ウィザードの 2 ページ目でルールテンプレートの一覧が表示されます。新しいルールのベースとして利用したいルールテンプレートを選択します。

全ての分析プロバイダーがカスタムルールをサポートしているわけではないことに注意してください。Java コード・レビューではいくつかのカスタムルールを提供しています。

図 13. カスタムルールの生成で利用できるテンプレート



ウィザードの最後のページでルールテンプレートに定義された各パラメーターのエントリが表示されます。図 14 の例では、選択されたルールテンプレートは 1 つのパラメーターだけで定義されています。ですから、提供されたフィールドに正しいクラス名を入力するだけで済みます。

7. **Browse** ボタンを使って既存のクラスを選択するか、テキストボックスに有効なクラス名を入力します。

図 14. テンプレート値 (パラメーター) 設定ビュー

The screenshot shows a dialog box titled "Add Custom Rule" with a subtitle "Assign Template Values". The instruction "Fill in the values associated with this template" is displayed. The "Qualified class name:" field contains "java.util.List" and has a "Browse..." button next to it. The "Severity:" dropdown menu is set to "Recommendation". The "Description:" text area contains the placeholder text "Enter the qualified name of a class. For example 'java.lang.String'". At the bottom, there are four buttons: a help icon (?), "< Back", "Next >", "Finish", and "Cancel".

8. **Finish** ボタンをクリックしてテンプレートベースのルールを生成し、ルールツリーに追加します。

このルールは以降の解析構成で選択可能になります。

次は

この 4 パートからなるシリーズの第 1 の記事では、静的解析の概要を紹介し、開発サイクルの早い時期にコード品質の問題を発見するのに役立つ **Rational Software Analyzer** の主要機能を説明しました。

**Part2** では **Rational Software Analyzer** の **Java** コード・レビューについてより詳しく見ていきます。提供される **API** を使い、多様なデータを扱うルールテンプレートやルールを作成できる拡張機能についても解説します。

- **Rational** 製品評価版ダウンロードページは[こちら](#)

## 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.