外部のコード・チェッカーを Eclipse CDT に統合する

Eclipse の Codan で C/C++ コードの解析ツールを実行する

Alex Ruiz
Software engineer
Google

2012年 9月 20日

Codan は C/C++ プロジェクト用のコード解析フレームワークとして Eclipse の CDT (C/C++ Development Tooling) に組み込まれています。静的なコード解析を実行するためのインフラストラクチャーを提供する Codan には、そのまま使用できる問題チェッカーが付属しています。Eclipse の Juno リリースでは Codan が拡張され、外部のコード解析ツールを自動的に実行できるようになっています。この記事では著者の Alex Ruiz が、この拡張がなぜ Eclipse CDT のユーザーにとって朗報なのかを説明し、また Java コードと簡単な XML を使用することで、お気に入りのコード・チェッカーを Eclipse の C/C++ 開発環境に統合する方法について説明します。

Codan は C/C++ プロジェクトのコード・チェックを実行するコード解析フレームワークです。2011年以来、Codan は Eclipse CDT (C/C++ Development Tooling) スイートの一部として、静的なコード解析に必要なインフラストラクチャーのすべてを提供しているのみならず、そのまますぐ使用できる上に有用な問題チェッカーもいくつか提供しています (「参考文献」を参照)。

2012年 6月の Eclipse Juno リリースで Codan は更新され、Eclipse の中で外部のコード解析ツールを自動的に実行できるようになりました。これは Eclipse CDT や C/C++ を使用する開発者にとって画期的な前進です。Codan が以前提供していた問題チェッカーは優れていたものの、外部のコード解析ツールと同等の機能を持たせるには、はるかに多くの機能が Codan には必要でした。そして現在では、Codan は Cppcheck や clang_check などの成熟した外部ツールと容易に統合することができます。

この記事と関連する内容の developerWorks の記事

この記事の他にも developerWorks には、Eclipse CDT を使用したアプリケーション開発に関する以下の記事があります。

- 「Eclipse Platform を使用した C/C++ 開発」
- 「Eclipse C/C++ Development Toolkit を使ってアプリケーションを開発する」

外部のコード解析ツールを Eclipse CDT に統合すると、Codan 単体で行う場合に比べ、数多くの優れたコード・チェック機能がもたらされます。また外部ツールとの統合により、開発全体の生産性を大幅に改善することができます。そして現在では、外部のコード解析ツールを Codan の

「Preferences (設定)」ページから構成できるようになりました。これらのツールは Codan と統合されると自動的に呼び出され、外部ツールの出力がエディター・マーカーとして表示されます。

この記事では、Java コードと簡単な XML を使用して皆さんのお気に入りのコード解析ツールを Eclipse の C/C++ 開発環境に統合する方法について説明します。ここでは例として Cppcheck を Codan に統合する方法を説明しますが、ここで説明するプロセスは皆さんが選択した他のツールにも応用できるはずです。

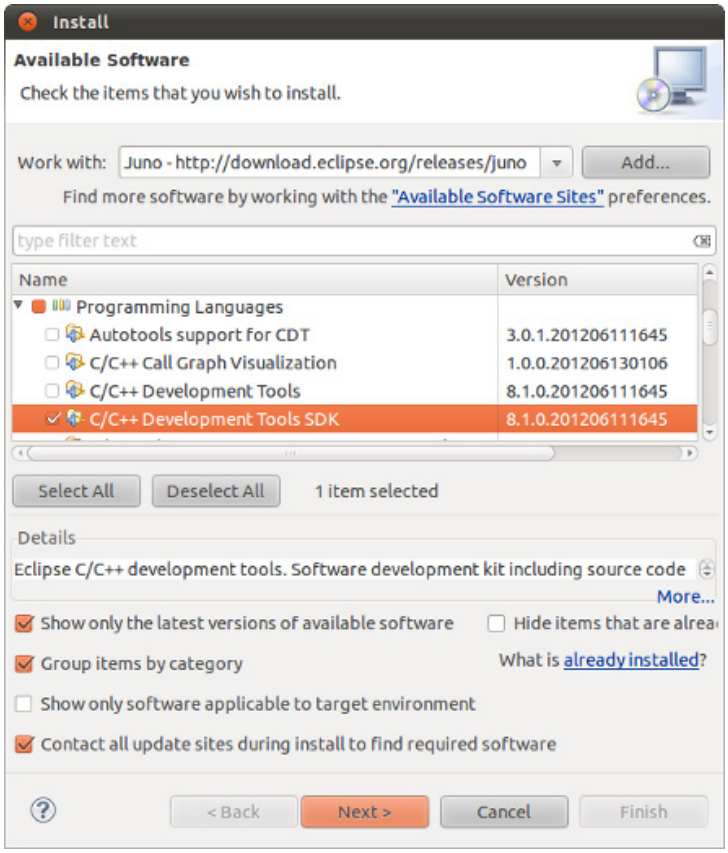
Eclipse Juno と CDT をインストールする

この記事で説明する例を実際に試すには、Eclipse Juno と CDT の両方をインストールしておく必要があります。まだ Eclipse をインストールしていない場合には、CDT がプリインストールされたバージョンをインストールすることができます。そのためには、単純に Eclipse のダウンロード・ページから Eclipse IDE for C/C++ Developers を選択します。

CDT が含まれていない Eclipse を既にインストールしてある場合には、その開発環境を以下の手順で更新します。

1. Eclipse の中で、メニューから「Help (ヘルプ)」 > 「Install New Software... (新規ソフトウェアのインストール...)」の順に選択します。
2. 「Install (インストール)」ダイアログでドロップダウン・リストから「Juno」を選択します。
3. 「Programming Languages (プログラミング言語)」カテゴリーで「C/C++ Development Tools SDK」を選択します。

図 1. CDT をインストールする



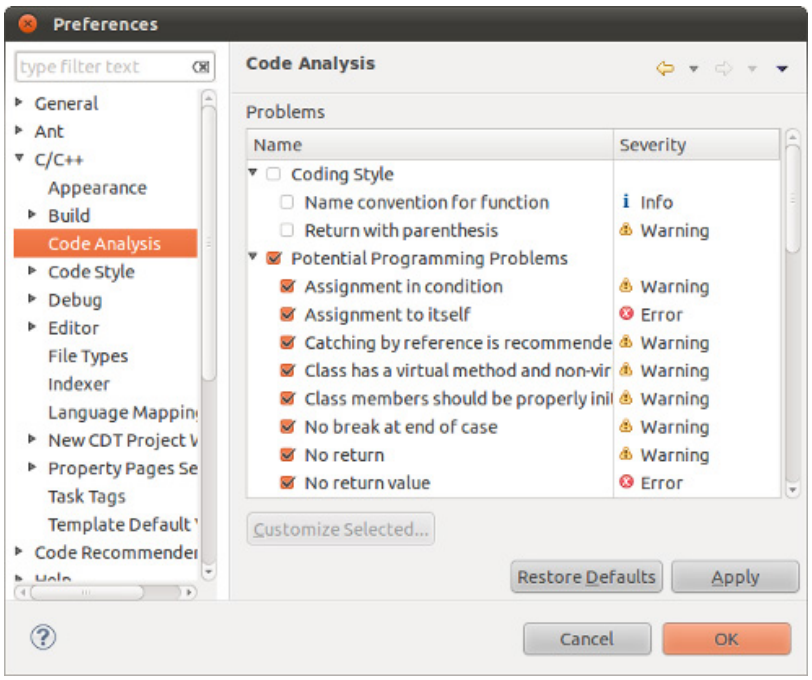
CDT の他に、コードのコンパイル、ビルド、デバッグのための標準的な GNU C/C++ 開発ツールが必要です。これらのツールのインストール方法については「参考文献」を参照してください。

Codan を起動する

Codan によるコード・チェックの大部分はデフォルトで有効になっています。Eclipse の「Preferences (設定)」ページまたはプロジェクトの「Property (プロパティー)」ページを使用することで、それぞれワークスペース・レベルまたはプロジェクト・レベルで Codan のコード・チェックを個別に構成することができます。

Codan の「Preferences (設定)」ページ (図 2) には、利用可能なすべてのコード・チェッカーと、各コード・チェッカーがレポートしたコードの問題のすべてを表示することができます。

図 2. Codan の「Preferences (設定)」ページでのコード・チェッカー



このページから、問題の重大度のチェックを有効化、無効化、または変更することができます。個々の問題の他のプロパティーを構成したい場合には、1 つの問題を選択し、「Customize Selected... (選択された項目のカスタマイズ...)」ボタンをクリックします。図 3 は「Name convention for function (関数の命名規則)」という問題に対する構成オプションを示しています。

図 3. 問題を構成する

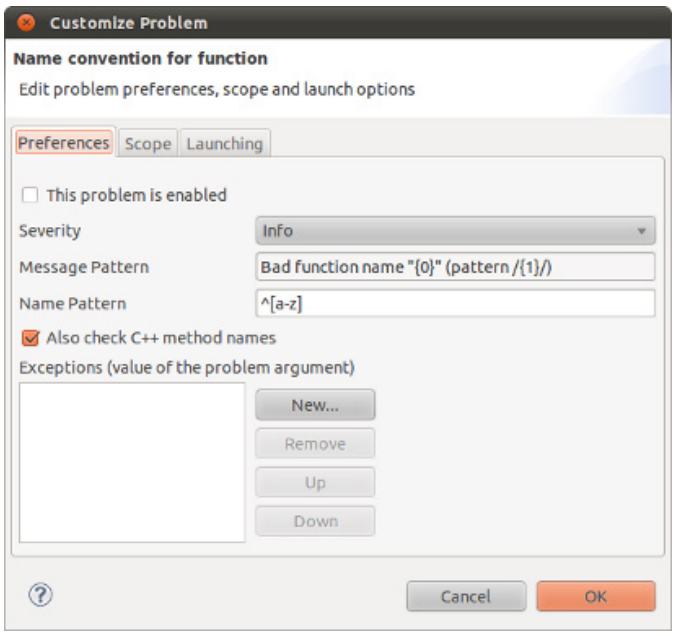


図 3 に示されている 3 番目のタブにより、どのようにして問題のチェックを開始するかを指定することができます。

- 「Run as you type (入力と同時に実行)」: ユーザーが CDT エディターでファイルに変更を加えた時に開始します。
- 「Run on file open (ファイルを開く時に実行)」: CDT エディターでファイルを開く時に開始します。
- 「Run on file save (ファイルを保存する時に実行)」: CDT エディターでまだ保存していない変更を保存する時に開始します。
- 「Run on incremental build (インクリメンタル・ビルド時に実行)」: インクリメンタル・ビルドが行われた時 (通常は、プロジェクト・レベルのオプション「Build Automatically (自動的にビルド)」が有効になっていて、ファイルが保存された時) に開始します。このオプションと「Run on file save (ファイルを保存する時に実行)」を同時に有効にすると、コード・チェックは 2 度実行されます。
- 「Run on full build (フル・ビルド時に実行)」: フル・ビルドが行われた時 (例えば、プロジェクトがクリーンアップされた場合など) に開始します。
- 「Run on demand (オンデマンドで実行)」: ユーザーがコンテキスト・メニューの項目「Run C/C++ Code Analysis (C/C++ コード解析の実行)」から手動でコード・チェックをトリガーすると開始します。

Codan を使用してコードをチェックする

Codan がどのように動作するかを理解できるように、ここでは簡単な C/C++ ファイルによる C++ プロジェクトを作成します。このファイルの中で、変数の自己代入を行います。Codan には「Assignment to itself (自己代入)」というコード・チェックが含まれており、デフォルトで重大度が「error (エラー)」に設定された状態で有効になっています。このチェックは、入力と同時に実行するように構成されているため、エラーは即座にポップアップ表示されます。

図 4. コード・チェックを実行する Codan

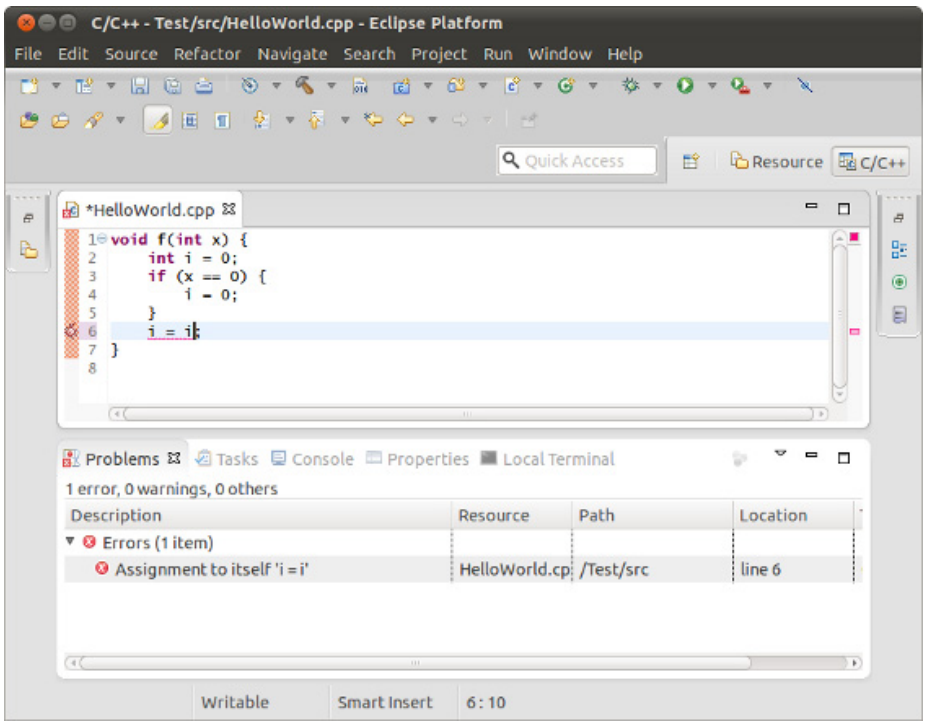


図 4 を見ると、Codan は私がファイルを保存しようとする前に自己代入エラーを検出し、それをレポートしていることがわかります。完璧です！

Codan の使い方の詳細については Codan プロジェクトのホームページを訪れてください (「参考文献」を参照)。

Cppcheck を Eclipse CDT に統合する

外部のコード解析ツールを Codan に統合するには、そのツールを呼び出す機能を持った特殊なチェッカーを作成する必要があります。チェッカーというのは Codan の `IChecker` インターフェースを実装したもので、指定された `IResource` (通常は `IFile`) に対して何らかのコード・チェックを実行します。

ここでは外部ツール・ベースのチェッカーを容易に作成する方法を示すために、よく使われているツールである Cppcheck (「参考文献」を参照) を呼び出すチェッカーを作成します。そのためには以下のことを行います。

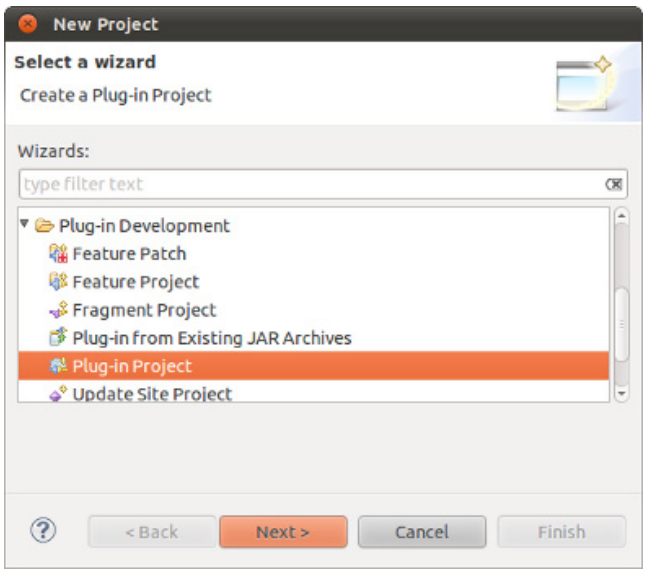
- Eclipse プラグイン・プロジェクトを作成し、依存関係として Codan を追加します。
- エラー・パーサーを作成します。このパーサーは Cppcheck の出力を構文解析し、必要に応じてエディター・マーカーを作成します。
- コード・チェッカーを作成します。コード・チェッカーは Cppcheck を呼び出すクラスです。

ステップ 1. Eclipse プラグイン・プロジェクトを作成する

Codan チェッカーを作成するために、まず新しい Eclipse プラグイン・プロジェクトを以下の手順で作成します。

1. メニューから「File (ファイル)」 > 「New (新規)」 > 「Project... (プロジェクト)」の順に選択します。
2. 「Plug-in Development (プラグイン開発)」カテゴリーで「Plug-in Project (プラグイン・プロジェクト)」を選択し、「Next (次へ)」をクリックします。
3. プロジェクトの名前 (ここでは「CppcheckChecker」) を入力し、「Next (次へ)」をクリックします。
4. デフォルトを受け入れ、「Finish (完了)」をクリックします。

図 5. プラグイン・プロジェクトを作成する



新しいプラグイン・プロジェクトを作成するとすぐに、Eclipse は自動的に `MANIFEST.MF` ファイルを開きます。このファイルに Codan の依存関係を追加します。

エディターで「Dependencies (依存関係)」タブを選択し、「Required Plug-ins (必須プラグイン)」のリストに以下のプラグインを追加します。

```
org.eclipse.cdt.codan.core

org.eclipse.cdt.codan.core.cxx

org.eclipse.cdt.codan.ui

org.eclipse.cdt.codan.ui.cxx

org.eclipse.cdt.core

org.eclipse.cdt.ui

org.eclipse.core.resources

org.eclipse.core.runtime

org.eclipse.ui
```

Eclipse プラグイン

この記事では Eclipse が話題の中心というわけではないため、Eclipse プラグインの作成方法の詳細については説明しません。Eclipse プラグインの作成方法を紹介した記事については「参考文献」を参照してください。

ステップ 2. エラー・パーサーを作成する

Cppcheck の出力からエディター・マーカーを作成するエラー・パーサーが必要なので、次のステップでは Eclipse の C/C++ ツールをプラグインで拡張します。Eclipse 自体が Java アプリケーションなので、拡張のためには Java コードを使用します。

最初に、`org.eclipse.cdt.core.IErrorParser` を実装した `CppcheckErrorParser` クラスを作成します。まず、コードの問題をレポートする際に Cppcheck が使用するパターンを見つけます。エラー・パーサーは問題レポートを表現する出力行をこのパターンを使用して特定し、その出力から、エディター・マーカーの作成に必要な情報を抽出します。

リスト 1. Cppcheck の出力と突き合わせるためのパターン

```
  // sample line to parse:
  //
  // [/src/HelloWorld.cpp:19]: (style) The scope of the variable 'i' can be reduced
  // ----------1--------- -2    --3--  ------------------4------------------------
  //
  // groups:
  // 1: file path and name
  // 2: line where problem was found
  // 3: problem severity
  // 4: problem description
  private static Pattern pattern =
      Pattern.compile("\\[(.*):(\\d+)\\]:\\s*\\((.*)\\)\\s*(.*)");
```

エラー・パーサーがこのパターンを使用して、チェック対象のファイルのパスと名前、そして検出されたエラーの場所、説明、重大度を抽出する方法をリスト 2 に示します。この情報を基に、エラー・パーサーは新しい `ProblemMarkerInfo` を作成し、指定された `ErrorParserManager` に渡します。`ErrorParserManager` はエディター・マーカーを作成するクラスです。

リスト 2. Cppcheck の出力を処理する

```
  @Override
  public boolean processLine(String line, ErrorParserManager parserManager) {
    Matcher matcher = pattern.matcher(line);
    if (!matcher.matches()) {
      return false;
    }
    IFile fileName = parserManager.findFileName(matcher.group(1));
    if (fileName != null) {
      int lineNumber = Integer.parseInt(matcher.group(2));
      String description = matcher.group(4);
      int severity = findSeverityCode(matcher.group(3));
      ProblemMarkerInfo info =
          new ProblemMarkerInfo(fileName, lineNumber, description, severity, null);
      parserManager.addProblemMarker(info);
      return true;
    }
    return false;
  }
}
```

問題の重大度をマッピングする

Cppcheck は独自に問題の重大度を定義しますが、この重大度はエディター・マーカーが使用する重大度と同じではありません。例えば、Cppcheck の「style」という重大度に対応する重大度は Eclipse の世界にはありません。この問題に対処するには、Codan と Cppcheck の間での、問題の重大度に関するマッピングを作成する必要があります。`findSeverityCode` メソッド (リスト 3) は、このマッピングを直接的な方法で実装しています。

リスト 3. 問題の重大度をマッピングする

```
private static Map<String, Integer> SEVERITY_MAPPING = new HashMap<String, Integer>();

static {
  SEVERITY_MAPPING.put("error", IMarkerGenerator.SEVERITY_ERROR_RESOURCE);
  SEVERITY_MAPPING.put("warning", IMarkerGenerator.SEVERITY_WARNING);
  SEVERITY_MAPPING.put("style", IMarkerGenerator.SEVERITY_INFO);
}

private int findSeverityCode(String text) {
  Integer code = SEVERITY_MAPPING.get(text);
  if (code != null) {
    return code;
  }
  return IMarkerGenerator.SEVERITY_INFO;
}
```

上記のようにマッピングを作成すると、Cppcheck が「style」という重大度でレポートする問題はすべて、Eclipse では「`SEVERITY_INFO`」という重大度で表示されます。このマッピングによって定義されるのは問題の重大度に対するデフォルトの値のみです。後で説明するように、このマッピングは Codan の「Preferences (設定)」ページから構成することができます。

Codan が `CppcheckErrorParser` を認識するようにするためには、以下のように拡張ポイント `org.eclipse.cdt.core.ErrorParser` を使用して plugin.xml ファイルに CppcheckErrorParser を登録する必要があります。

リスト 4. エラー・パーサーを登録する

```
<extension id="com.developerworks.cdt.checkers" name="Cppcheck error parsers"
    point="org.eclipse.cdt.core.ErrorParser">
  <errorparser class="cppcheckchecker.CppcheckErrorParser"
      id="com.dw.cdt.checkers.CppcheckErrorParser"
      name="Cppcheck">
    <context type="codan" />
  </errorparser>
</extension>
```

リスト 4 で、`ErrorParser` 拡張ポイントは元々 CDT のビルド・ツールに対するパーサーを登録するために作成されたものであることに注意してください。ErrorParser 拡張ポイントは Codan 専用ではないため、`CppcheckErrorParser` は Codan でのみ使用すべきであることを示すために、ここではコンテキスト "`codan`" を追加しています。

ステップ 3. コード・チェッカーを作成する

`AbstractExternalToolBasedChecker` は、外部ツールをベースにした任意の Codan コード・チェッカーのスーパークラスです。このクラスは外部のコード解析ツールを呼び出すために必要なインフラストラクチャーの大半を提供します。ここでは Cppcheck を統合するので、このクラスを `CppcheckChecker` と呼ぶことにします。

まず、外部ツールに関連して Codan の「Preferences (設定)」ページに表示される情報のデフォルトの値を指定する必要があります。

この情報はチェッカーのコンストラクターに渡す必要があり、以下の内容が含まれています。

- 外部のコード解析ツールの名前 (この場合は Cppcheck)。
- そのツールの実行ファイルの名前 (この場合は `cppcheck`)。ここでは実行ファイルがシステムの `PATH` にあるという前提なので、実行ファイルのパスを指定する必要はありません。
- 実行ファイルに渡す引数 (1 つの `String` の中に含まれています)。ここでは Cppcheck のチェックをすべて有効にするために、「`--enable=all`」を指定しています。

リスト 5. Cppcheck によるチェックにデフォルトで含まれる情報

```
public CppCheckChecker() {
  super(new ConfigurationSettings("Cppcheck", new File("cppcheck"), "--enable=all"));
}
```

Codan の「Preferences (設定)」ページを使用すると、実行ファイルのパスと渡される引数の両方を変更できることに注意してください。

問題の重大度を問題 ID にマッピングする

次に、ここで使用するエラー・パーサーの ID を指定する必要があります (リスト 6)。これらの ID は plugin.xml ファイルで使用される ID と同じでなければなりません。

リスト 6. 使用するエラー・パーサーの ID を指定する

```
@Override
protected String[] getParserIDs() {
  return new String[] { "com.dw.cdt.checkers.CppcheckErrorParser" };
}
```

先ほどのリスト 2 では、エディター・マーカーを作成するために、`ProblemMarkerInfo` を作成して指定の `ErrorParserManager` に渡しました。`ProblemMarkerInfo` を渡された `ErrorParserManager` は新たに作成されたチェッカーにエディター・マーカーを作成させます。

チェッカーにエディター・マーカーを作成させるには、`addMarker(ProblemMarkerInfo)` メソッドをオーバーライドする必要があります (このメソッドは別のタイプのミスマッチを検出するためのものです)。Codan チェッカーは `ProblemMarkerInfo` から直接エディター・マーカーを作成することはできません。Codan チェッカーには、作成されたエディター・マーカーに対応する重大度を、問題 ID を使用して判断するための独自のメカニズムがあります。

問題 ID は、コード・チェッカーがレポートしたコードの問題を特定するために Codan が使用する一意の ID です。コードに関するすべての問題は Codan の「Preferences (設定)」ページに表示されます (図 2 を参照)。

リスト 7. エディター・マーカーを作成する

```
@Override
public void addMarker(ProblemMarkerInfo info) {
  String problemId = PROBLEM_IDS.get(info.severity);
  String description = String.format("[cppcheck] %s", info.description);
  reportProblem(problemId, createProblemLocation(info), description);
}
```

`ProblemMarkerInfo` で示される重大度に対応する問題 ID を知るためには、重大度と問題 ID とのマッピングを作成する必要があります。リスト 8 は、このマッピングの実装方法を示しています。

リスト 8. 問題の重大度を問題 ID にマッピングする

```
private static final String ERROR_PROBLEM_ID =
    "com.dw.cdt.checkers.cppcheck.error";

private static final Map<Integer, String> PROBLEM_IDS =
    new HashMap<Integer, String>();

static {
  PROBLEM_IDS.put(
      IMarkerGenerator.SEVERITY_ERROR_RESOURCE, ERROR_PROBLEM_ID);
  PROBLEM_IDS.put(
      IMarkerGenerator.SEVERITY_WARNING, "com.dw.cdt.checkers.cppcheck.warning");
  PROBLEM_IDS.put(
      IMarkerGenerator.SEVERITY_INFO, "com.dw.cdt.checkers.cppcheck.style");
}
```

外部のコード解析ツールを使用するコード・チェッカーは、そのチェッカーによる問題 ID のうちのどれを「リファレンス」とみなすのかを示す必要があります。リファレンスとなる問題 ID はチェッカーの設定値 (例えば、リスト 5 に示した外部ツールの名前など) を取得するために使用されます。すべての問題で設定が共有されるため、どの問題 ID がリファレンスであっても関係はありません。

リスト 9. リファレンスとなる問題 ID を指定する

```
@Override
protected String getReferenceProblemId() {
  return ERROR_PROBLEM_ID;
}
```

定数 `ERROR_PROBLEM_ID` はリスト 8 で定義されます。

チェッカーを登録する

Codan の「Preferences (設定)」ページに、コード・チェッカーとともに、コード・チェッカーがレポートするすべての問題が表示されるように (また、そうすることでユーザーがそれらを利用できるように) するには、チェッカーを Codan の plugin.xml ファイルに登録する必要があります。

Cppcheck によってレポートされる問題のすべてを認識できるわけではなく、また Cppcheck の今後のバージョンで何らかのコード・チェック機能が追加または削除されるのを防ぐことはできないため、個々の問題をすべて登録することはできません。問題をすべて登録する代わりに、ここでは重大度によって問題をグループ分けし、各グループを個々の問題として扱うことにします。リスト 10 では、errors (エラー)、warnings (警告)、style violations (スタイル違反) を個々の 3 つの問題として登録します。

リスト 10. チェッカーと問題レポートを登録する

```
<extension point="org.eclipse.cdt.codan.core.checkers">
  <category id="cppcheckChecker.category" name="Cppcheck" />
  <checker class="cppcheckchecker.CppcheckChecker" id="cppcheckChecker.cppChecker"
    name="CppcheckChecker">
    <problem id="com.dw.cdt.checkers.cppcheck.error" name="Error"
      defaultEnabled="true" defaultSeverity="Error" messagePattern="{0}"
      category="cppcheckChecker.category"/>
    <problem id="com.dw.cdt.checkers.cppcheck.warning" name="Warning"
      defaultEnabled="true" defaultSeverity="Warning" messagePattern="{0}"
      category="cppcheckChecker.category"/>
    <problem id="com.dw.cdt.checkers.cppcheck.style" name="Style"
      defaultEnabled="true" defaultSeverity="Info" messagePattern="{0}"
      category="cppcheckChecker.category"/>
  </checker>
</extension>
```

ユーザーに対して表示されるチェッカーの名前を `category` 要素の中で指定します。問題の ID はチェッカーで使用される問題 ID と同じでなければなりません (リスト 8 を参照)。

plugin.xml ファイルの中で、3 つの問題すべてに関して以下の内容を指定します。

- 問題はデフォルトで有効です。
- 問題のデフォルトの重大度はそれぞれ “Error” (エラー)、”Warning” (警告)、”Info” (情報) です。
- 問題のメッセージ・パターンは “`{0}`” です。このパターンにより、Codan は Cppcheck がレポートした問題の説明をそのまま使用することになります。

Codan で Cppcheck を使用する

これで、Codan の「Preferences (設定)」ページに `CppcheckChecker` が表示されるようになりました (図 6)。

Codan の「Preferences (設定)」ページに表示されている Cppcheck のスクリーン・ショット

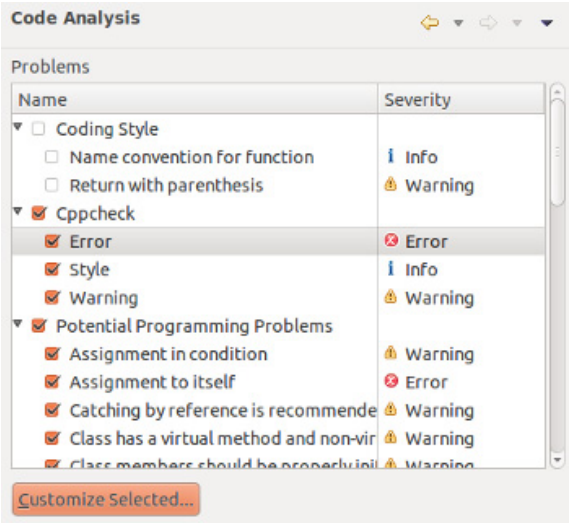


図 7 に Cppcheck がコードのエラーをどのようにレポートするかを構成するためのオプション設定の画面を示します。

図 7. Cppcheck のエラー・レポートを構成する

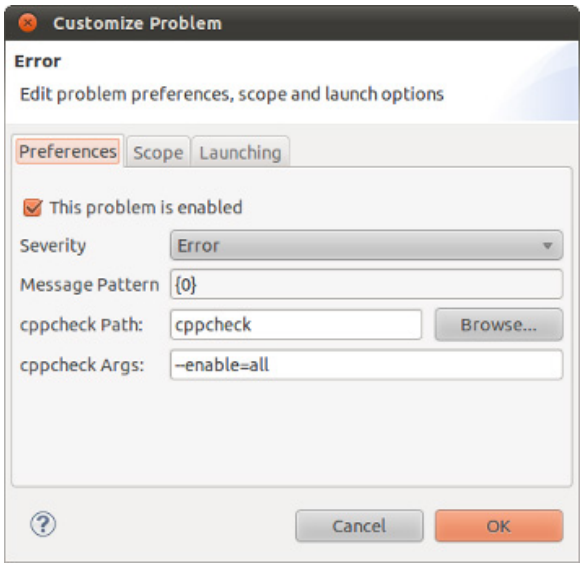
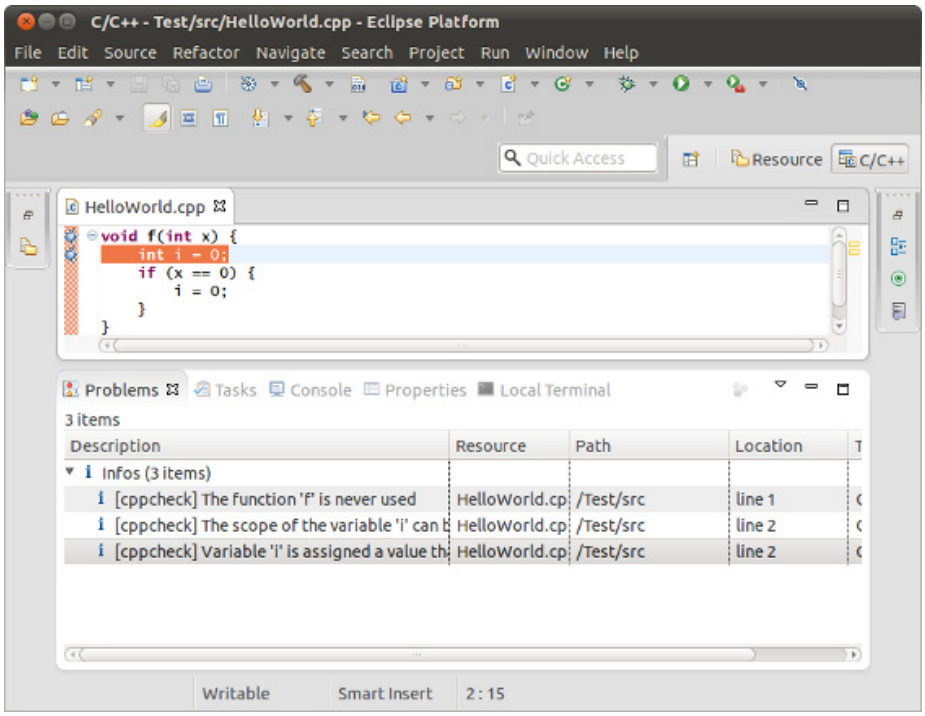


図 8 は Cppcheck がどのようにコードの問題をレポートするかを示しています。ファイルが保存されると自動的に Cppcheck が呼び出されたことに注意してください。

図 8. Cppcheck によるレポート



統合による欠点

Codan を外部のコード解析ツールと統合することによる 1 つの制約は、ユーザーが入力している間は外部ツール・ベースのチェッカーを実行できないことです。これは単純に、ファイルに加えられた変更が保存されない限り外部ツールが変更を把握することができないことによるもので

す。そのため、ファイルが開かれた時とファイルが保存された時に外部チェッカーを実行する必要があります。

ただしこの制約を上回るほど、成熟したコード解析ツールを使用することには大きなメリットがあります。また、通常のチェッカーを作成するよりも、外部ツールを Codan に統合する方がはるかに容易で単純です。通常のチェッカーを作成するには C 言語または C++ 言語、さらには CDT の AST 実装について十分理解する必要がありますが、それとは対照的に、ここでは (2 つのクラスの中に含まれた) 約 100 行の単純な Java コードと 30 行の XML で `CppcheckChecker` を作成することができました。

まとめ

Eclipse の Juno がリリースされる以前は、Codan 用のカスタム・コード・チェック機能を作成するには C/C++ 言語と CDT 独自の AST 実装を十分に理解している必要がありました。Eclipse CDT の Juno リリースではその問題が解決され、開発者が Codan 用のコード・チェッカーを作成することにより、そのコード・チェッカーを通じて負荷の重い作業を外部のコード解析ツールに任せられるようになりました。

この記事では、簡単な Java コードと XML のみを使用して Codan に Cppcheck を統合しました。その結果、よく使用される C/C++ コード解析ツールと、Eclipse に組み込まれている C/C++ プログラム・コード解析用フレームワークがを統合されました。先ほど触れたように、ここで説明したプロセスは皆さんのお気に入りのコード解析ツールにも応用できるはずです。詳細については「参考文献」を参照してください。

著者について

Alex Ruiz

Alex Ruiz は Google Inc. の Engineering Tools 組織に所属しています。彼は時間に余裕がある時にはオープンソース・プロジェクトの作業を行い、またブログや技術記事を書いたり国際会議で講演をしたりしています。彼のブログをご覧ください。この記事に書かれた意見は彼自身の意見であり、彼が所属する企業の意見ではありません。