# **“Code Scent”, Customizable Code Smell Detection Plugin for IntelliJ IDEA**

4nix (Team 4)

Jinyoung Kim, Chanho Song, Jinmin Goh

Hyunbin Park, Seokhwan Choi, Gwanho Kim

## **1. Description of Our Project**

### **1.1 Goal**

The primary goal of "Code Scent" is to enhance code quality in software development projects by detecting and addressing code smells within the developer's workspace. It aims to assist in maintaining clean, readable, and well-organized code, which is essential for the scalability, maintainability, and performance of software.

Our plugin offers customization options, allowing users to set their own parameters for detecting various code smells, making it adaptable to different coding styles and preferences.

### 

### **1.2 Problem and Solution**

The problem addressed by "Code Scent" is the presence of "code smells" in software development projects. These are suboptimal coding patterns that, while not stopping the code from functioning, make it harder to understand, maintain, and modify. Such issues can lead to difficulties in bug fixing, feature addition, and general code alteration, potentially resulting in new bugs.

"Code Scent" offers an innovative solution by introducing an automated, intelligent code smell detection system within the IntelliJ IDEA environment. This is further enhanced by its unique customization feature, which allows users to customize the detection parameters to their specific needs and preferences.

This level of customization ensures that developers can optimize the tool to better align with their individual coding styles and project requirements, thereby maximizing the advantage and utility of "Code Scent" in various software development scenarios.

## 

## **2. Architecture and Design**

### **2.1 Major Components of Our Implementation**

For our project “Code Scent”, major implementations are divided into different components as detecting, ui, utils, and test. detecting component includes classes for detecting code smells.ui component includes classes for GUI configurations and is further grouped into analyzing, and customizing parts. utils component includes a LoadPsi class that is commonly used in detecting classes. test component includes classes for testing our detecting package.

More detailed description of our major components will be provided later in the next following sections.

### **2.2 Code Smell Detection Functionality Model**

#### **2.2.1 Model of Detecting and utils Package**

The BaseDetectAction.java file is an abstract class belonging to detecting package. It's designed to provide a framework for code smell detection techniques.

The class is abstract, so it's intended to be subclassed by other specific detection classes. It defines abstract methods like storyID(),description(), which would be implemented by subclasses to provide specific functionality for different types of code smells. BaseDetectAction serves as a base class for other code smell detection classes, such as those for detecting comments, dead code, duplicated code, etc. Each specific detection class would extend this base class and implement its abstract methods.

Each of these subclasses (Comments, DeadCode, DuplicatedCode) extends the BaseDetectAction class. This inheritance would allow them to utilize the common framework provided by BaseDetectAction for code smell detection, while implementing specific strategies for their respective focused areas.

**BaseDetectAction (Abstract Class):**

* Central class in the diagram. It is an abstract class with abstract methods (like storyID(), storyName()) that need to be implemented by its subclasses.
* The class is part of the detecting package due to its role in detecting various code smells.

**Code smell detecting classes (subclasses of BaseDetectAction):**

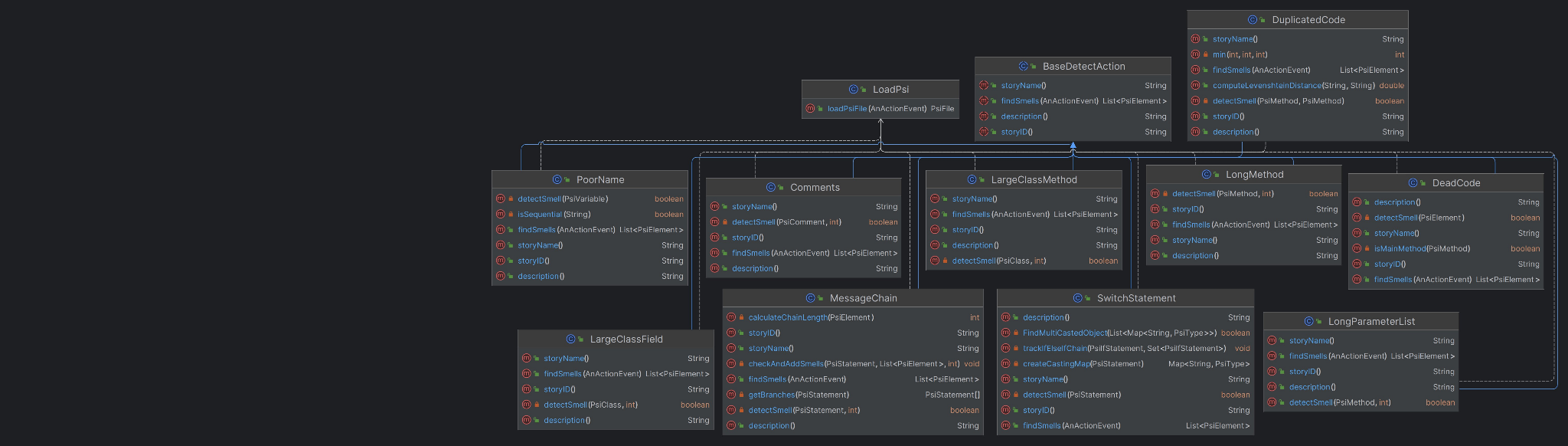
Each of the following subclasses extend the BaseDetectAction class, thus they have inheritance relationship. They are also part of the detecting package.

1. Comments: Implements specific methods for detecting comments code smells
2. DeadCode: Focuses on identifying unused code.
3. LargeClassMethod: Deals with the "Large Class" code smell, where a class does too many methods.
4. LongMethod: This would handle the "Long Method" smell, identifying methods that are too long
5. PoorName: Deals with naming conventions and identifies variables, methods, or classes that are poorly named and therefore could make the code harder to understand and maintain.
6. LargeClassField: Similar to LargeClassMethod, but focused on classes that may have too many fields (properties), suggesting that it may be violating the single responsibility principle by holding too much state.
7. LongParameterList: Looks for methods that have too many parameters, which can make methods difficult to understand and maintain.
8. MessageChain: Concerned with the "Message Chain" code smell, which occurs when a client requests a chain of method calls to navigate through multiple objects.
9. SwitchStatement: Deals with the "Switch Statement" smell. Excessive use of switch statements can be a sign that polymorphism could be utilized instead.
10. DuplicatedCode: Concerned with the "Duplicated Code" code smell, which occurs when a similar implementation of methods are used. The similarity is measured using Levenshtein distance.

**Utils LoadPsi (Dependency relationship with code smell detecting classes):**

The LoadPsi.java file contains a single public static method loadPsiFile which is designed to retrieve a PsiFile instance from the IntelliJ IDEA IDE environment when an action is performed. This class is used for all code smell detecting classes, thus they have dependency relationship.

UML class diagram for these relationships are shown below. All subclasses are inheritance of BaseDetectAction, and all of them also have dependency relationship with LoadPsi.



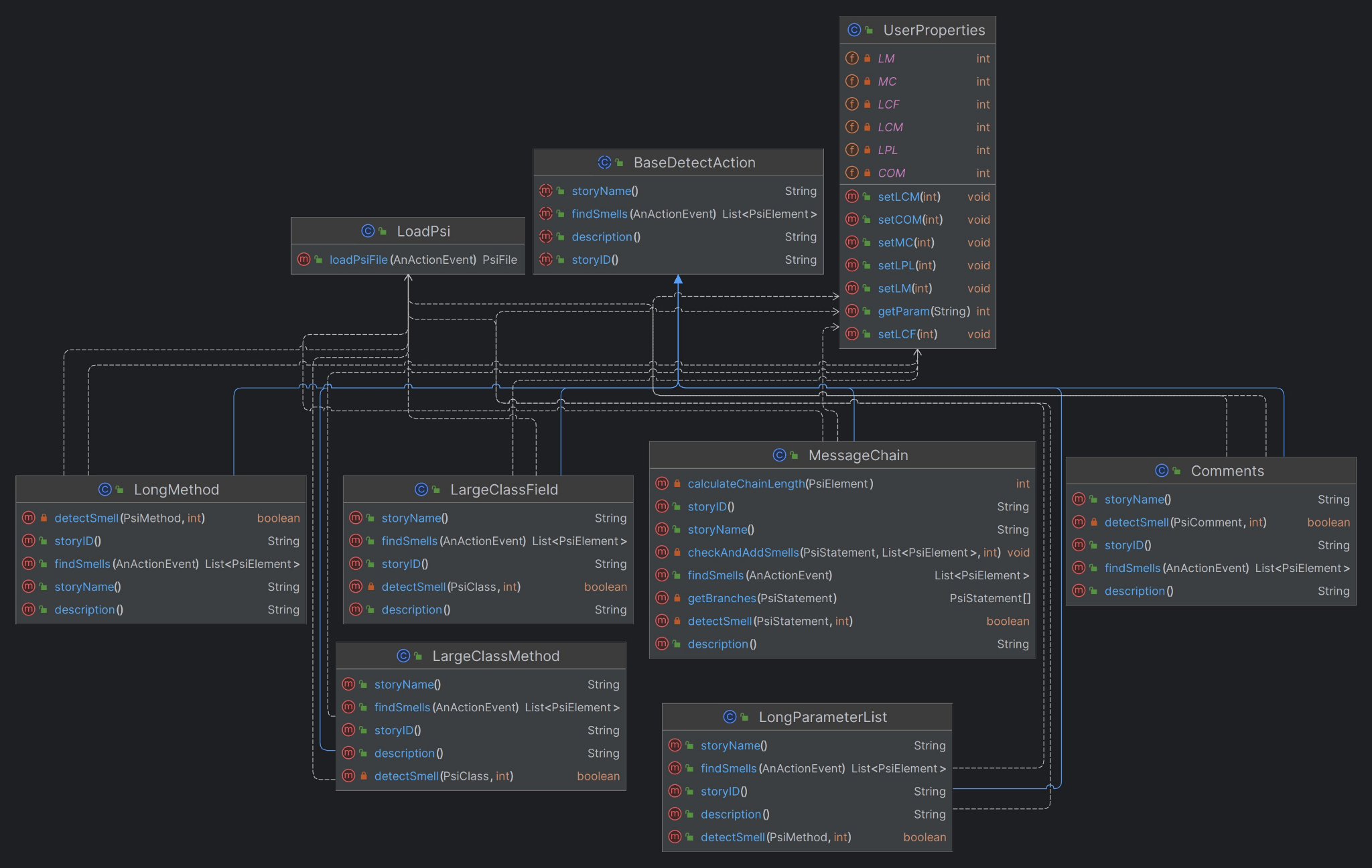
#### **2.2.2 Model of Detecting and UserProperties Package**

UserProperties class represents a pattern for managing configuration settings in an application, encapsulating the settings in static variables and providing a standardized way to access and modify them, with persistence handled by another class HandleConfig.

For example in the LongMethod class, which extends BaseDetectAction, UserProperties class is used to get a user-defined threshold for what constitutes a "long method." This is done by the getParam method of the UserProperties class. The findSmells method calls UserProperties.getParam(storyID()) to get the user-defined maximum line count for a method to be considered long. The storyID() method returns the string "LM", which corresponds to the "Long Method" threshold in the UserProperties class.

UML class diagram for relationships between user customizable code smell detection classes and UserProperties are shown below. All of the code smell detection classes have dependency relationships with UserProperties and LoadPsi.

More details about UserProperties are given in **2.3.2 Model of ui.customizing package.**



### 

### **2.3 GUI Model**

#### **2.3.1 Model of ui.analyzing package**

ui.analyze package is related to displaying code smell results. This package shows a tool window to users, enabling them to identify the results.

The MyToolWindowAction.java class in the ui package extends AnAction class user’s actions with the GUI. This class is connected with plugin.xml to respond to the user's action.

**MyToolWindowAction:**

* **actionPerformed Method**: A public method that activates a specific tool window named "Code Scent" within a project, if the window is available, when an action event occurs.

The MyToolWindowFactory.java class in the ui package extends the ToolWindowFactory class to create and configure a specific tool window within the user interface.

**MyToolWindowFactory:**

* **Static Variables:** The class defines the following static variables.
  + customizeButton: JButton object for ‘Customize’ button.
  + analyzeButton: JButton object for ‘Analyze’ button.
  + analyzeallButton: JButton object for ‘Analyze all’ button.
  + buttonPanel: Jpanel object contains ‘Customize’, ‘Analyze’ and ‘Analyze all’ buttons.
  + treeWindow: JBScrollPane object for tree.
  + mainPanel: JPanel object, which is the main container for other UI components in the class.
  + project: Project object of current project context.
  + tree: JTree object, which is used to display a hierarchical structure of data.
  + resultAll: This field serves as a storage for results, mapping strings to lists of PsiElement objects.
* **getActiveProject Method:** A private method that iterates through open projects to find and return the active one. If no active project is found, it returns the first open project as a fallback.
* **createToolWindowContent Method:** A public method that sets up the content for a tool window within a given project. It initializes UI components like panels, buttons, and a scroll pane, and arranges them in the mainPanel. The method also defines actions for buttons and adds the fully assembled mainPanel to the tool window's content manager. This setup is essential for creating the interactive part of the tool window.
* **updateUIWithAnalyzeResult Method:** A public method that updates the UI to display analysis results. It removes the old tree view, fetches the active project, creates a new TreeStructureWindow class with the current results, and then adds this updated tree view to the mainPanel.
* **updateUIWithAnalyzeAllResult Method:** A public method that updates the UI based on analysis results. It merges the new results with the existing resultAll, refreshes the treeWindow with a new TreeStructureWindow class reflecting the updated data, and repositions it in the mainPanel. This method ensures the UI reflects the most current analysis state.
* **customDataContext Method:** A public method that creates a custom DataContext. It processes a VirtualFile object, manages file editors, and retrieves the Editor and PsiFile associated with the file. The method creates and returns a new DataContext instance, providing data relevant to the current file, editor, and PSI file, alongside other existing data from the original context.
* **mergeMaps Method:** A public method to merge two Map<String, List<PsiElement>> objects. It combines the lists associated with each key, adding entries from the first map to the second.
* **listFilesForFolder Method:** A public method to recursively gather VirtualFile objects representing Java source files from a specified folder and its subdirectories, focusing on files within the "main\\java" path.

The AnalyzeAction.java class in the ui.analyze package extends AnAction class to connect user’s actions with the GUI. This class is connected with plugin.xml to respond to the user's action.

**AnalyzeAction:**

* **setResultListener Method:** A public method to set an AnalyzeResultListener instance to the resultListener field, enabling customized event handling.
* **actionPerformed Method:** A public method overrides a specific event handling method. It initializes necessary resources, processes a predefined list of storyID to detect code smells, and accumulates the results. If a result listener is set, the method triggers it with the compiled results.

The AnalyzeResultListener.java class in the ui.analyze package is an interface class that stores analysis results from AnalyzeAction.java and returns the results to MyToolWindowFactory.java.

**AnalyzeResultListener:**

* **onAnalyzeResult Method:** A method in an interface, requiring implementation to handle analysis results, provided as a Map<String, List<PsiElement>>. It processes the results, typically representing code smells or similar findings, each mapped to a string key.

The BaseDetectManager.java class in the ui.analyze package handles storyID, which are defined at each Code Smell Detection. This class is connected to AnalyzeAction.java and ProjectTreeModelFactory.java to fetch BaseDetectAction method names by storyID.

**BaseDetectManager:**

* **Static Variable:** The class defines the following static variable.
  + manager: A variable storing the singleton instance of the BaseDetectManager class.
* **Constructor:** A protected constructor of the BaseDetectManager class, used to prevent external instantiation and support the singleton pattern.
* **getDetectActionByID Method:** A public method in the BaseDetectManager class that returns a BaseDetectAction object based on the provided id. It utilizes a switch statement to select and instantiate specific types of detect actions. If the id does not match any case, the method returns null.

The CodescentPopUp.java class in the ui.analyze package extends JPopupMenu class to create a context-specific popup menu. It specializes in displaying information related to BaseDetectAction objects.

**CodescentPopUp:**

* **Constructor:** A public constructor that adds a menu item with the description of a BaseDetectAction object to the popup, if the provided object is an instance of BaseDetectAction.

The TreeStructureWindow.java class in the ui.analyze package extends Tree class that is connected with MyToolWindowFactory class to show tree structure of result analyzing code smell.

**TreeStructureWindow:**

* **Static Variables:** The class defines the following static variables.
  + Icon1: An icon loaded from /general/projectStructure.svg
  + Icon2: An icon loaded from /nodes/configFolder.svg
  + Icon3: An icon loaded from /nodes/editorconfig.svg
  + currentHighlighter: RangeHighlighter object, which is used for managing text highlighting in the UI.
* **Constructor:** In Constructor It gets a model to set a Tree, and set a cell render to display the name and icon of each node, and set a mouse listener to handle click events.

1. **Getting a tree model**

by setModel **Method**, it gets tree model from createProjectTreeModel

in the ProjectTreeModelFactory class.

1. **Setting a cell renderer**

For each node, the renderer assigns different icons according to feature of each node - icon 1 for project node, icon 2 for the BaseDetectAction object node, icon 3 for PsiElement object node. And for the PsiElement object node, it appends fileName, lineNumber, and if possible name of PsiElement information of each PsiElement that the user can easily check where and which part of the code code smell occurs.

1. **Setting a mouse listener**
2. **one mouse click**

When the mouse click once it gets the clicked object of the node. Then it gets TextAttributes and offset of the element to make background color of the code smell element Yellow. Using MarkupModel class and currentHighlighter that is an instance of RangeHighlighter class, it removes the current highlighter in the scrollbar and add new highlighter according to the offset of the element with color pink. Also With the method of RangeHighlighter class the comment “code smell detected” is shown when the mouse hovered on the pink colored scrollbar stripe mark.

1. **double mouse click**

When mouse double click it gets the clicked object of the node.When the object is an instance of PsiElement it gets the OpenFileDescriptor object that element belongs to by getTextOffset method and getContainingFile method of object. Then, it navigates to the descriptor to move to the location of the element.

1. **mouse right click**

When mouse right click it gets the clicked object of the node.to show

popup that shows description of clicked code smell, it create

CodescentPopUp class object for the code smell type.

The ProjectTreeModelFactory class is statically used by TreeStructureWindow.java. Instance of ProjectTreeModelFactory class is not used but the method is just used to make a TreeModel object of result from AnalyzeAction class.

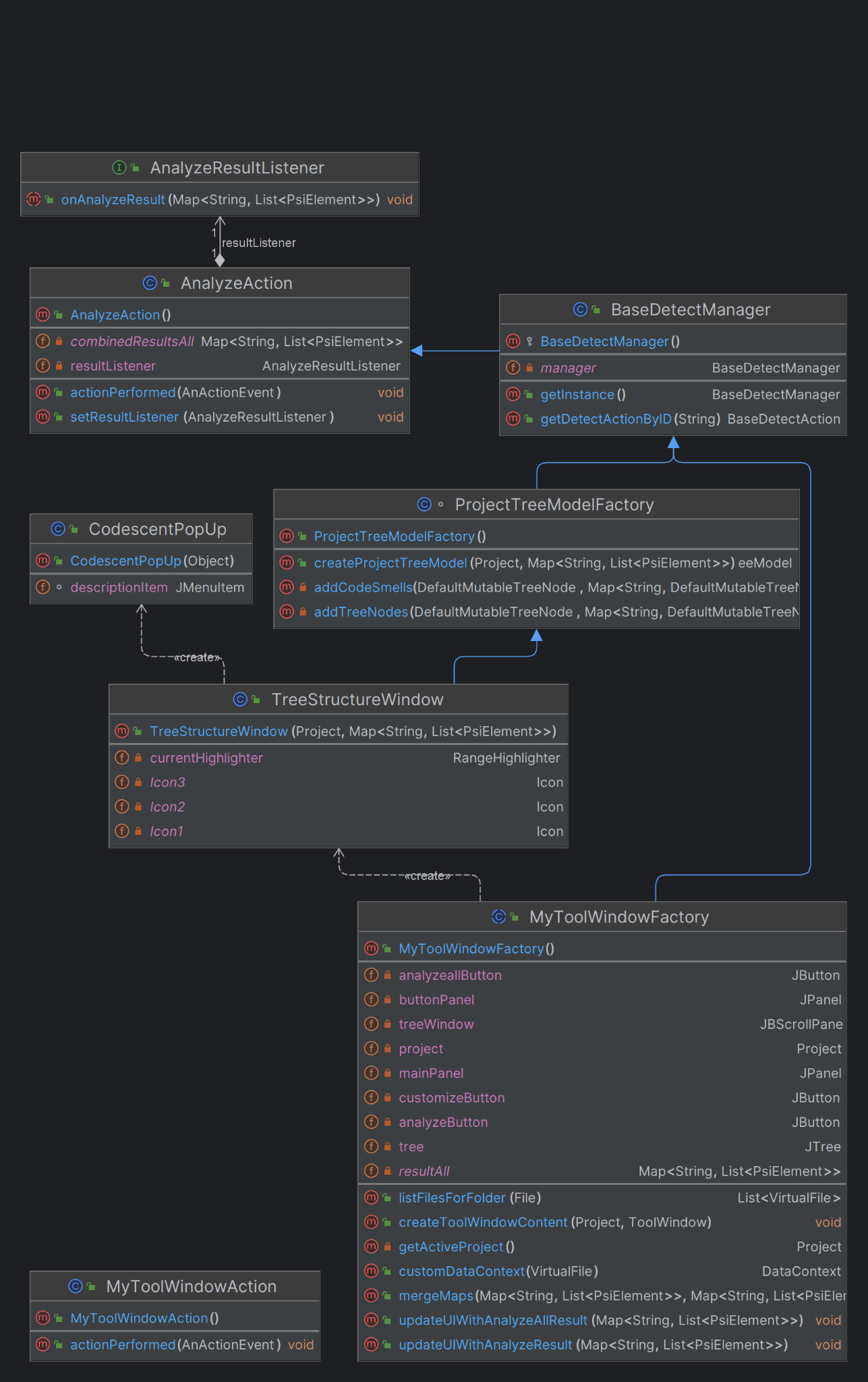
**ProjectTreeModelFactory**

* **createProjectTreeModel Method:** This method builds treemodel and returns it based on the active project and result provided as a Map<String, List<PsiElement>> from AnalyzeAction. At first the root node corresponding to the active project is created. Then for each List<PsiElement> of the code smell from result, corresponding nodes are added as children under codesmell type nodes by **addTreeNodes** method.
* **addTreeNodes Method:** This method is called by **createProjectTreeModel** method to add node for each psiElement in psiElements provided as List<PsiElement>. If node for code smell type already exists, node is just added under the node. If not, a node for code smell type is added under root node calling **addCodeSmells** method.
* **addCodeSmells Method:** This method is for adding node for the code smell type under the root node. Since this method make node based on BaseDetectAction object of code smell type, BaseDetectManager is used to get BaseDetectAction object.

UML class diagram for these relationships are shown below. You can see in the UML class diagram that **MyToolWindowFactory** class have reference to **AnalyzeAction** class and **TreeStructureWindow** class to detect code smells in the file and to show Tree object that show the result of detecting in the window. **AnalyzeAction** class have reference to **BaseDetectManager** to get every **BaseDetectAction** class object to check for every each code smell type. Also **AnalyzeAction** class is connected with **AnalyzeResultListener** in

**MyToolWindowFactory** class since it can get the result of detecting dynamically.

On the other hand, **TreeStructureWindow** class has reference to both **ProjectTreeModelFactory** class and **CodescentPopUp** class to get **TreeModel** object and to show popup when the codescent type node in the tree is right clicked reference. **ProjectTreeModelFactory** class have reference to **BaseDetectManager** class to get **BaseDetectAction** class object when adding node for each code smell type.

****

#### **2.3.2 Model of ui.customizing package**

ui.customizing package is related to the user's parameter customization. This package shows pop-up to users and enables them to customize thresholds used in code smell detection.

The HandleConfig.java class in the ui.customizing package handles codescent.properties file, which is located right below the root of the project.

**HandleConfig:**

* **Static Variables:** The class defines following static integer variables.
  + configPath: A path of codescent.properties file.
  + configFile: File object of config file.
  + prop: Properties object of config file.
  + fis: FileInputStream object used when user changes parameter.
  + handler: HandleConfig object, which is implemented as a singleton design pattern.
* **initializeConfig Method:** Designed to check whether config file exists and initialize parameters in Userproperties class. If there is no config file, call the createConfig method to create a new config file. After creating a file or file exists, parses parameters from the config file and sets parameters in UserProperties class.
* **getHandler Method:** Designed to make class as a singleton design pattern. If there is no handler variable created, call the constructor of the class to create a variable. Returns the handler variable of HandleConfig class.
* **getConfigParam Method:** Takes String parameter storyID. Loads parameter of storyID and returns it.
* **setConfigParam Method:** Takes String parameter storyID and int parameter param. Set storyID’s parameter into param write to config file.
* **createConfig method:** Creates codescent.properties file with configFile into configPath. The file is set with default parameters.

The UserProperties.java class in the ui.customizing package is a **utility** class that stores and manages user-customized properties related to the thresholds for various code smell condition parameters. Each of these parameters represents a customizable setting that a user of the system can modify, presumably to alter the behavior code smell detection functions.

**UserProperties:**

* **Static Variables:** The class defines the following static integer variables.
  + LM: Threshold for determining if a method is considered long.
  + LCF: Threshold for the number of fields in a class.
  + LCM: Threshold for the number of methods in a class.
  + LPL: Threshold for the number of parameters in a method's signature.
  + MC:Threshold for the length of a message chain.
  + COM: Threshold for length of comments in the code.
* **getParam Method:** This method takes a String parameter storyID, which is expected to match one of the predefined keys corresponding to the static variables.
* **initializeParam Method:** Designed to initialize the parameters from configuration source. It calls HandleConfig.initializeConfig(), which reads configuration file to set up initial values.
* **Setter Methods:** There are setter methods for each of the static variables (setLM, setLCM, setLCF, setLPL, setMC, setCOM). Each method updates the corresponding static variable with the new value entered by the user as a parameter and then calls a method in HandleConfig to persist this new value in the configuration setConfigParam.

The CustomizeAction.java class in the ui.customizing package extends AnAction class to connect user’s actions with customization GUI. This class is connected with plugin.xml to respond to the user's action.

**CustomizeAction:**

* **actionPerformed Method:** This method overrides AnAction class’s same method. This method takes AnActionEvent e. We can get Project class object from AnActionEvent, and give as parameter when create new CustomizePopipGUI object to show pop-up window.

The CustomizePopupGUI.java class in the ui.customizing package is a **GUI** class that shows a user customizable pop-up window. This class extends JFrame class and implements TableModelListener class.

**CustomizePopupGUI:**

* **Variables:** The class defines the following static integer variables.
  + customizeTable: A table that the user can change its contents.
  + scrollPane: An object that actually shows customizeTable visually.
  + columnType: Two column types, which are parameter and input.
  + data: Actual data which is shown at pop-up.
  + project: Project object, which represents this project.
* **Constructor:** The constructor initializes the table which is shown at pop-up. First, constructor initializes the UserProperties class’s parameters to include in the table. After initializing parameters, create data and set other settings related to the pop-up window. The important part is making only the parameter column editable. After setting, we call setVisible(true) to make the pop-up table visible.
* **tableChanged Method:** This method overrides TableModelListener class’s same method. In this method, we parse the user's input and check whether input is valid(positive integer form). If input is not valid, we show a warning message. If the input is valid, we apply the input into UserProperties class’s parameter.

UML class diagram for these relationships are shown below. CustomizeAction

creates CustomizePopupGUI class object. HandleConfig and

UserProperties class have dependency relationship with CustomizePopupGUI

and each other. HandleConfig class has a singleton design pattern.

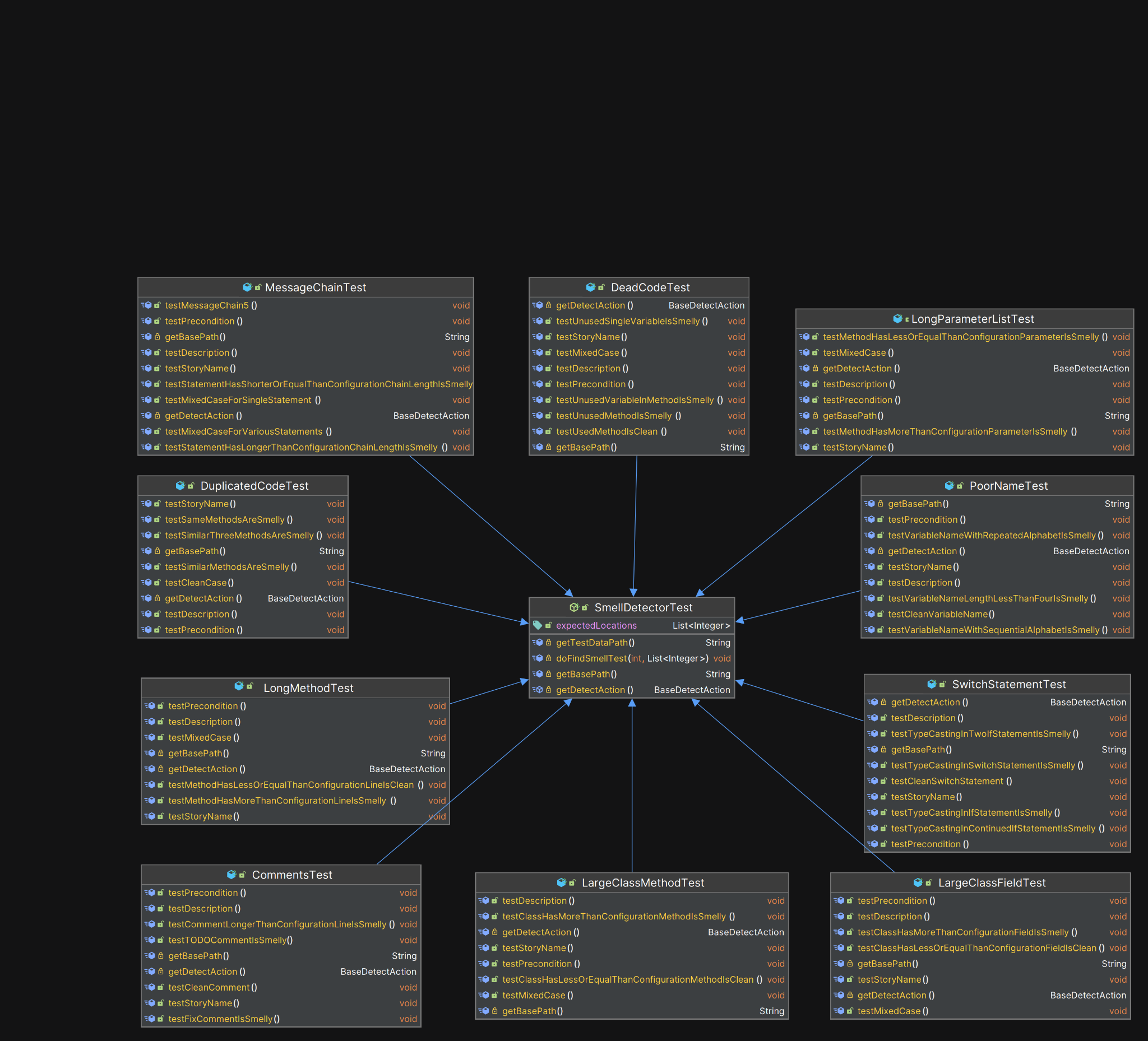
### 

### 

### **2.4 Test Model**

#### **2.4.1 Model of test package**

There are java and test packages in test package. The java package has SmellDetectorTest that is an abstract class for testing, and extended test classes for testing functionality in detecting package. UML class diagram for relationships of classes in Test package are shown below.



#### **2.4.2 SmellDetectorTest class**: **The core class for testing**

We inherited the LightJavaCodeInsightFixtureTestCase class to build a test environment. This structure is Inspired by the *2020 wanted team’s test structure*. The LightJavaCodeInsightFixtureTestCase class is a class that provides a test environment for the development of the IntelliJ plug-in. By inheriting this class, the SmellDetectorTest class acquires the ability to perform light code insights fixture tests in the IntelliJ environment. The part in our code that the class was used is as follows:

* Load test data files: When inherited, the getTestDataPath, getBasePath, and configureByFiles methods make it easy to locate test data files.
* Use myFixture Object: You can use the myFixture object to get insights on code modification and analysis during code testing. myFixture.doSomething()
* Utilize the IntelliJ API: The IntelliJ IDEA API enables you to perform a variety of functions. You can create AnActionEvent and use Data Manager and Action Manager to set the required context.

#### **2.4.3 Fields and methods in the SmellDetectorTest class**

* **expectedLocations Variable**: List<Integer> that stores the expected location of the detected smelly codes. Use this variable in the doFindSmellTest method to compare with the actual results.
* **getTestDataPath Method**: Specifies the directory path where the test data file is located. The test data files used in the test case are loaded based on this path.
* **getBasePath Method**: Specifies the default path for test data files. The test data files are stored relative to this path.
* **getDetectAction Abstract Method**: returns a BaseDetectAction object. This method must be implemented in a specific subclass.
* **doFindSmellTest Method**: detect smelly codes and verify results for a given test case number and a list of expected locations. Load a specific test data file using the configureByFiles method, and set AnActionEvent to create the required context. Call the findSmells method in BaseDetectAction to detect smelly codes and receive the results as a list of PsiElements. Calculate the location of the detected smelly codes and perform the test against the expected location list. This class provides a general test capability that sets up the test environment for the plug-in, detects smelly codes from specific test data files, and compares the detected location to the expected location.

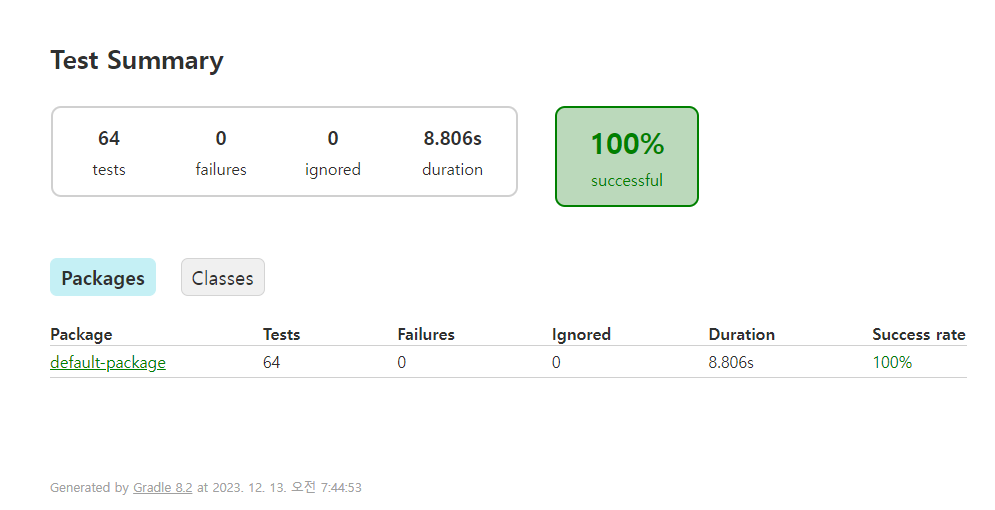
## 

## **3. Appendix**

### **3.1 How to run automated tests**

To run automated tests, type ./gradlew test command in terminal.

you can get below test report in *csed332project/build/reports/tests/test*



To test manually with GUI, follow below steps.

### **3.2 Ready to get into CodeScent**

1. git clone our project in your local machine.
2. Run Plugin (Either run manually by Gradle -> Run configurations -> runIde or type ./gradlew runIde command in terminal)

2.1 **(If new project is created after step 1, and you want to analyze from this new project)** Delete initially existing Main.java

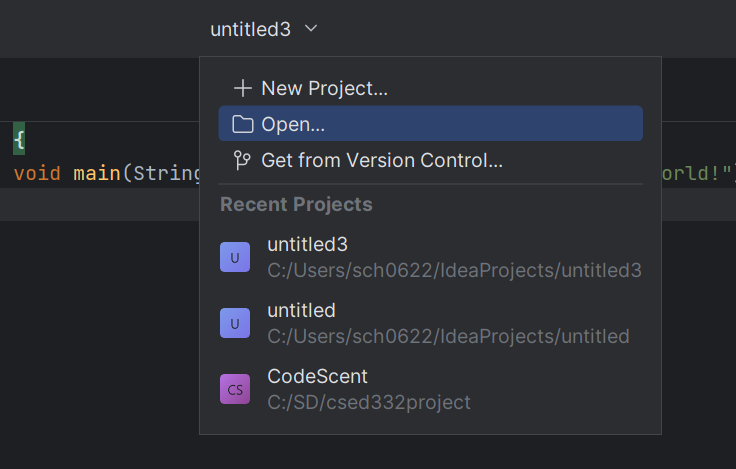


2.2 Create the *main/java* folder under the *src* folder and code under it. **Our plug-in recognizes the .java file in src/main/java.**

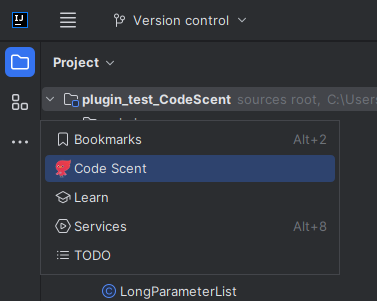
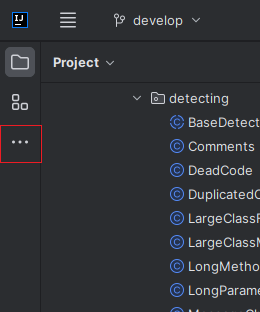
****

3.1 **(If you want to open an existing project to analyze after step 1)** Click the title of the project on the upper middle of the screen, then click open.

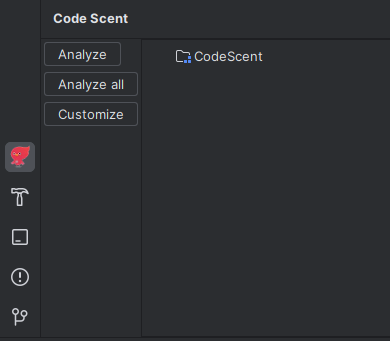
3.2 Then locate the project location that you want to open to analyze code smells.



4. You may find CodeScent icon from more tool windows button.

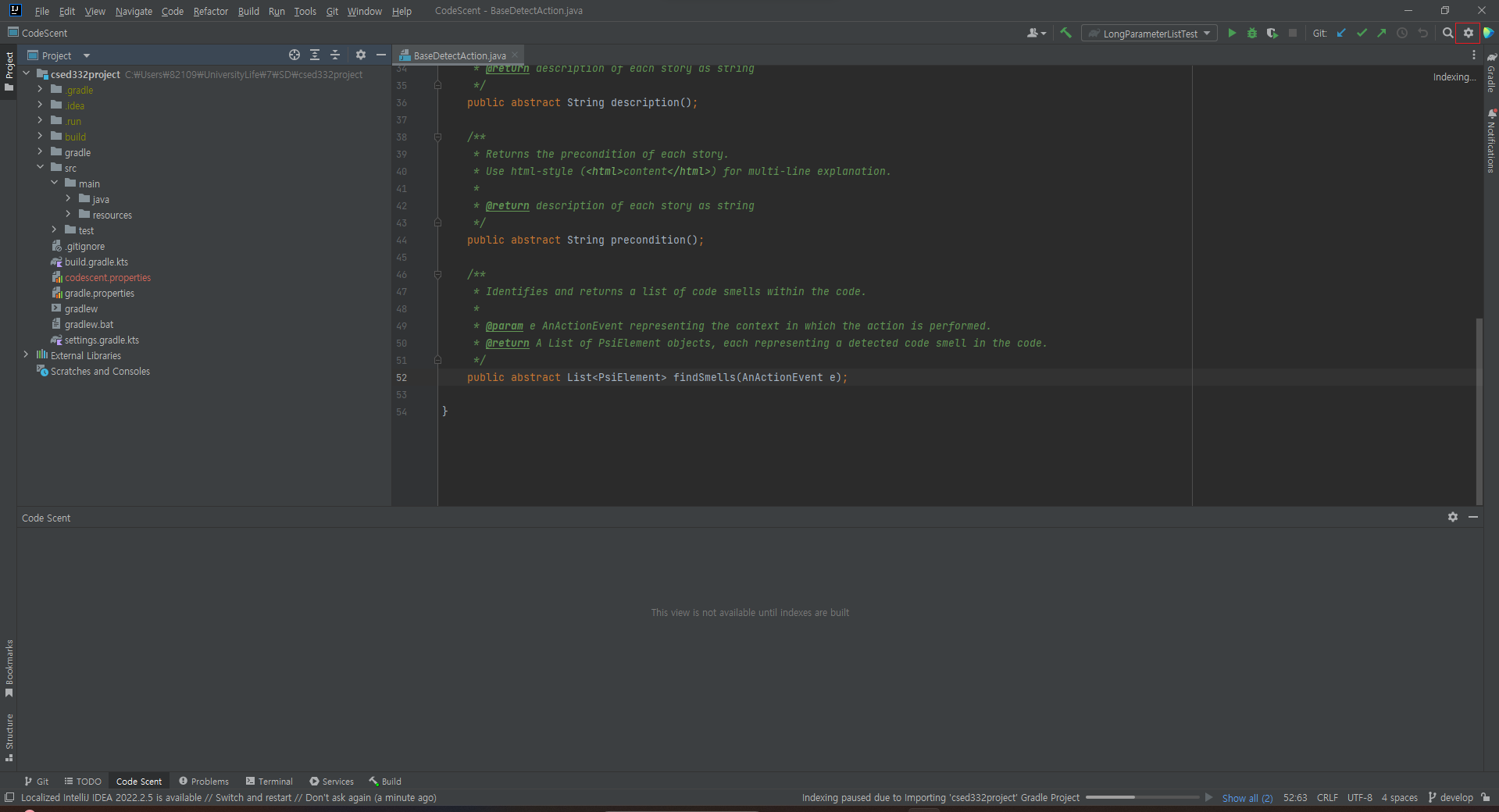


5. Click the ponix Code Scent icon in the left-bottom toolbar.



**If your IDE screen UI are like below, follow the steps to synchronize us.**

1. Click the settings button located at the upper right corner.

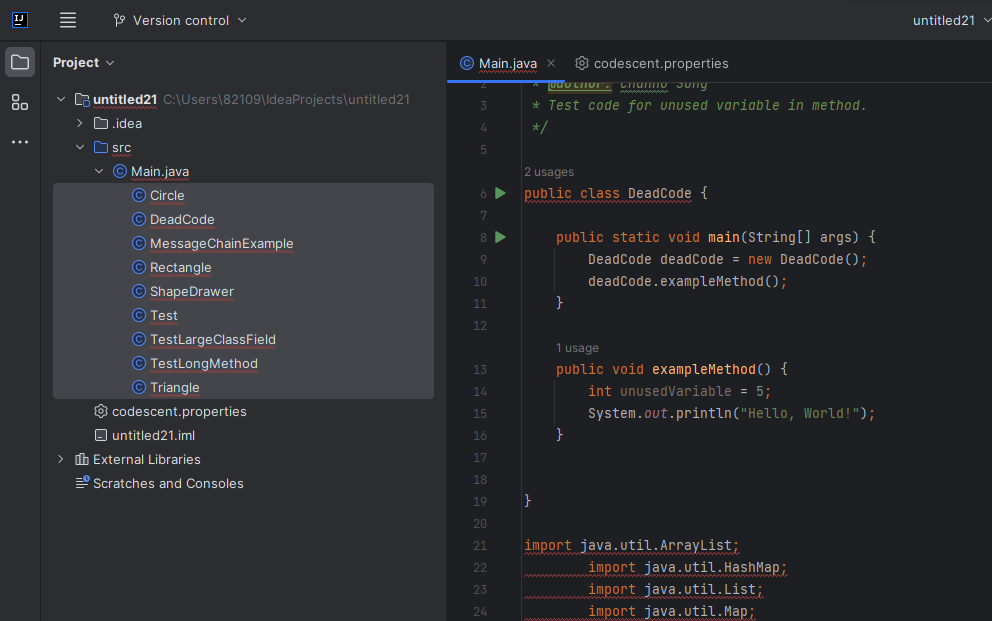


1. Appearance & Behavior > New UI Preview > Check Enable New UI > OK > Shutdown now and reload plug-in IDE
2. Start from **step 2.1**

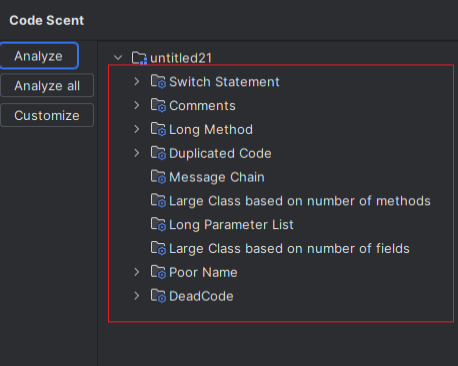
#### 

#### **3.2.1 Analyze**

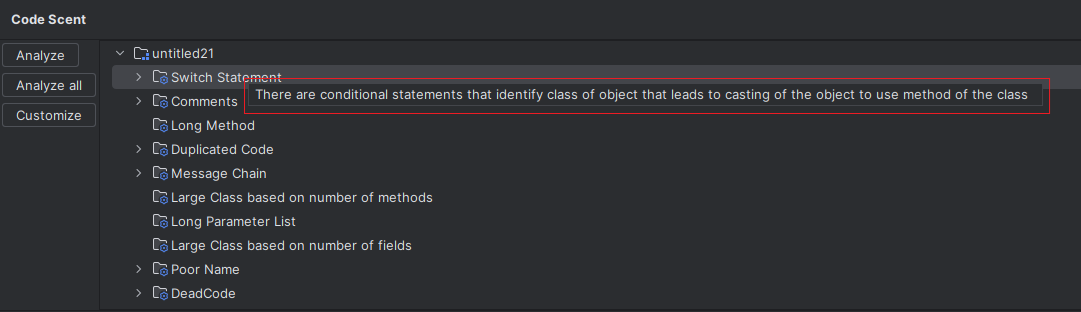
1. In Idea project, open a project file that you want to analyze, or like below image, open .java files that you want to analyze under src Main.java directory.



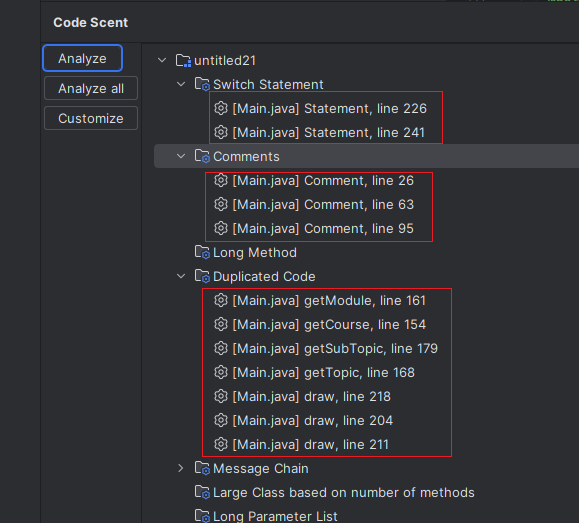
1. Double-click Analyze button. You can see code smell directories which have smelly code components in **currently open file**.



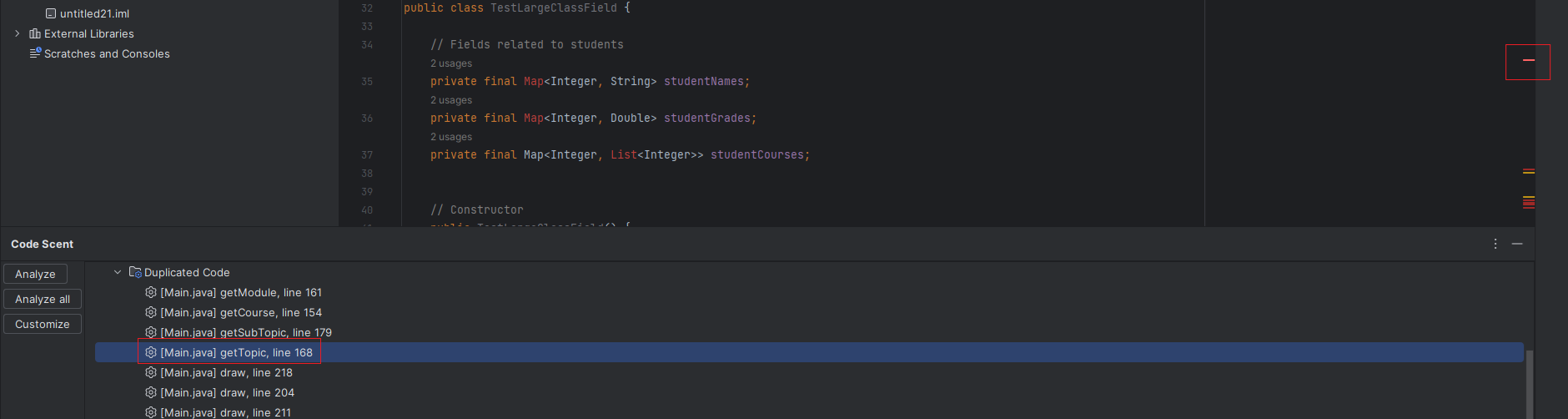
1. Right-click each directory to view the description of that code smell directory.



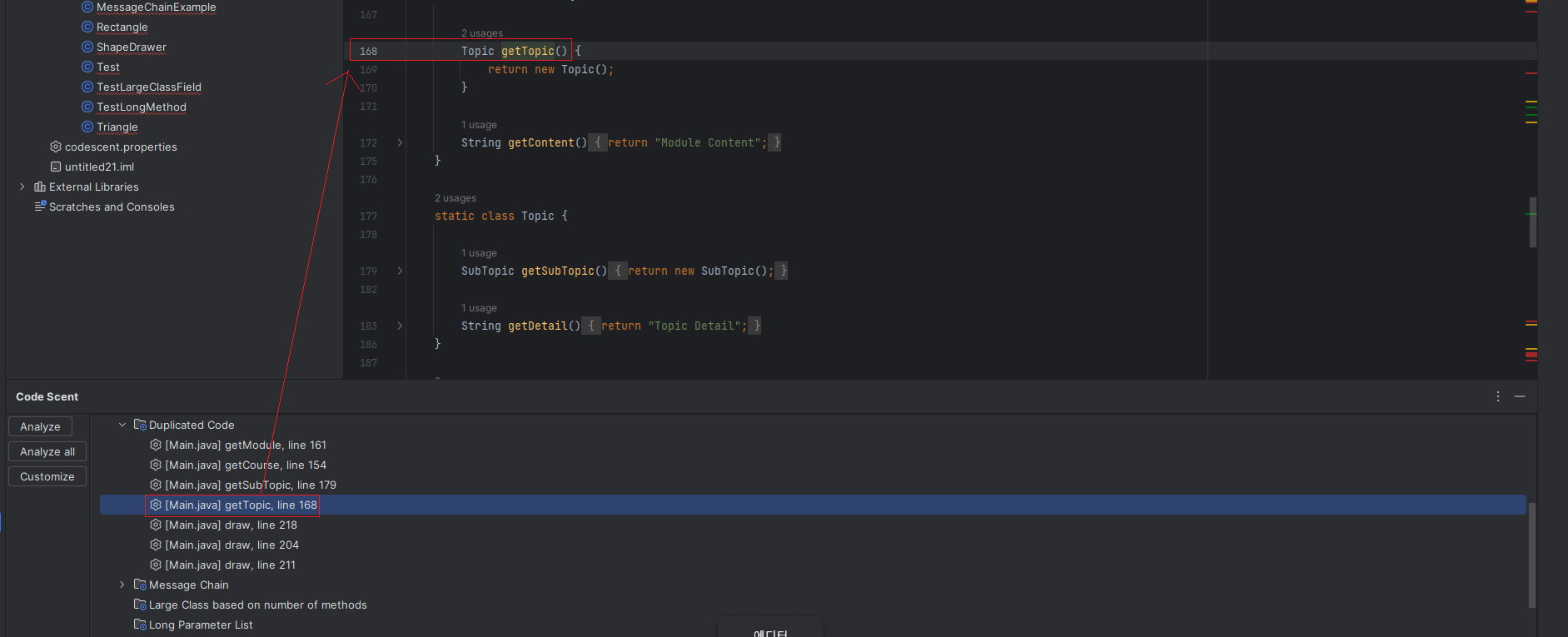
1. Double-click each directory to show the components belonging to the corresponding code smell directory.



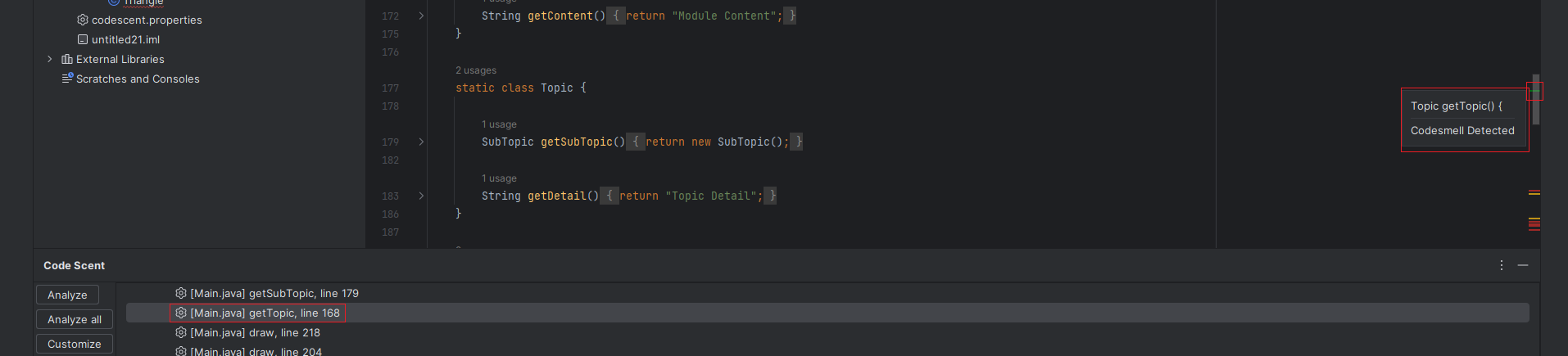
1. If you click on that component, the location of that component will appear as a red bar in the right scroll bar.



1. Double-click each component to move it to the location of that component.



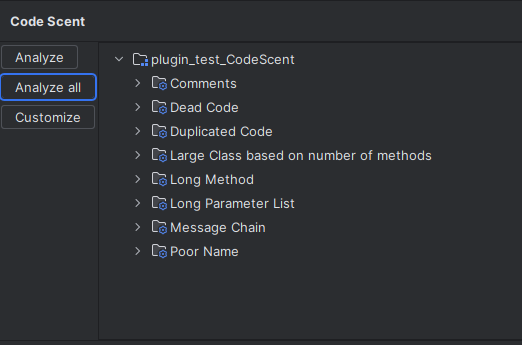
1. In addition, the red bar we saw in 5 changed to the green bar, and the name of the detected component is shown by hovering the mouse on the bar.



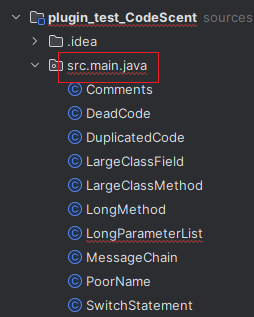
#### 

#### **3.2.2 Analyze All**

1. Open the idea project that you want to analyze for code smells, and run the plugin (same as step 1 in **3.2.1 Analyze**)
2. Click Analyze All button.

****

1. You can obtain code smell analysis results for **all .java files in src/main/java directory**.

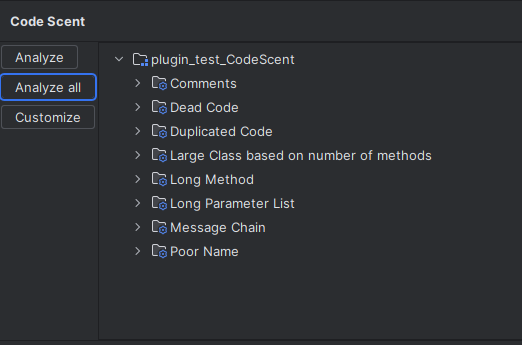
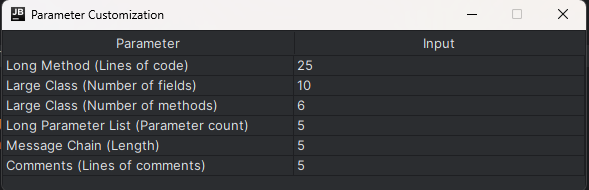


1. Remaining steps work the same as steps from 3 to 7 in **3.2.1 Analyze**.

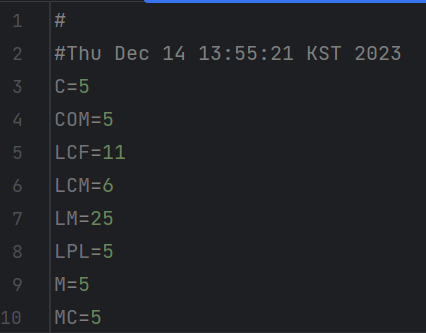
#### 

#### **3.2.3 Customize**

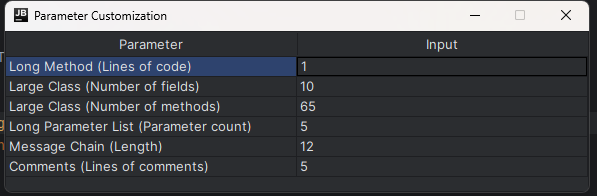
1. If you click the customize button on the bottom left image, you can get **Parameter Customization** **window.** Initially, the values in the window are set to default values.

****

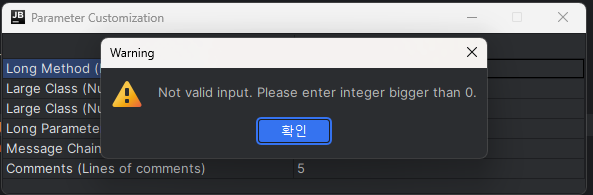
1. If codescent.properties exists and is not in the format below, it may not be loaded. In this case, delete the existing codecent.properties file and then it will be automatically created again correctly.



1. You can **freely customize threshold values** by double-clicking on the input tab of each parameter.



1. When you input invalid value such as a non-positive value or an alphabet, the window rejects your input showing a warning window.



1. If you press the X button, it is set to the desired value.
2. Now, if you analyze your code after customizing the parameters, the detection result for code smells will be based on the parameter values that you customized.