Duplicates Finder (Java)

The Duplicates Finder (Java) Build Runner is intended for catching similar code fragments and providing a report on discovered repetitive blocks of Java code. This runner is based on IntelliJ IDEA capabilities, so an IntelliJ IDEA project file (.ipr) or directory (.idea) is required to configure the runner. Since TeamCity 2017.1, in addition to the bundled version, it is possible to install another version of JetBrains IntelliJ Inspections and Duplicates Engine and/or change the defaults using the Administration | Tools page.

The Duplicates Finder (Java) can also find Java duplicates in projects built by Maven2 or above.

> In order to run inspections for your project you should have either an IntelliJ IDEA project file (.ipr)/project directory (.idea), or Maven2 or above pom.xml of your project checked into your version control.

This page contains reference information about the following Duplicates Finder (Java) Build Runner fields:

IntelliJ IDEA Project Settings

Option	Description
Project file type	To be able to run IntelliJ IDEA inspections on your code, TeamCity requires either an IntelliJ IDEA project file\directory, Maven pom.xml or Gradle build.gradle to be specified here.
Path to the project	Depending on the type of project selected in the Project file type, specify here: • For IntelliJ IDEA project: the path to the project file (.ipr) or the path to the project directory the root directory of the project containing the `.idea` folder). • For Maven project: the path to the `pom.xml` file. • For Gradle project: the path to the `.gradle` file. This information is required by this build runner to understand the structure of the project. ⓘ The specified path should be relative to the checkout directory.
Detect global libraries and module-based JDK in the *.iml files	This option is available if you use an IntelliJ IDEA project. In IntelliJ IDEA, the module settings are stored in *.iml files, thus, if this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDKs when saved. This helps ensure that all references will be properly resolved. Warning When this option is selected, the process of opening and saving the build runner settings may become time-consuming, because it involves loading and parsing all project module files.
Check/Reparse Project	This option is available if you use an IntelliJ IDEA project. Click this button to reparse your IntelliJ IDEA project and import the build settings right from the project, for example the list of JDKs. If you update your project settings in IntelliJ IDEA (e.g add new jdks, libraries), remember to update the build runner settings by clicking Check/Reparse Project.
Working directory	Enter a path to a Build Working Directory if it differs from the Build Checkout Directory.Optional, specify if differs from the checkout directory.

Unresolved Project Modules and Path Variables

This section is displayed, when an IntelliJ IDEA module file (.iml) referenced from an IPR-file:

- cannot be found

- allows you to enter the values of path variables used in the IPR-file.

To refresh values in this section click Check/Reparse Project.

Option	Description
<path_variable_name>	This field appears if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In Set value to field, specify a path to the project resources to be used on different build agents.

Project JDKs

This section provides the list of JDKs detected in the project.

Option	Description
JDK Home	Use this field to specify the JDK home for the project. ⚠ When building with the Ipr runner, this JDK will be used to compile the sources of corresponding IDEA modules. For Inspections and Duplicate Finder builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process, the JDK specified in the `JAVA_HOME` environment variable will be used.
JDK Jar File Patterns	Click this link to open a text area where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns. The default value is used for Linux and Windows operating systems: `jre/lib/*.jar` For Mac OS X, use the following lines: `lib/*.jar` `../Classes/*.jar`
IDEA Home	If your project uses the IDEA JDK, specify the location of the IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

⚠ You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a requirement for the corresponding property.

Java Parameters

Option	Description
JDK	Select a JDK. This section details the available options. The default is JAVA_HOME environment variable or the agent's own Java.
JDK home path	The option is available when <Custom> is selected above. Use this field to specify the path to your custom JDK used to run the build. If the field is left blank, the path to JDK Home is read either from the JAVA_HOME environment variable on agent the computer, or from the env.JAVA_HOME property specified in the build agent configuration file (buildAgent.properties). If these values are not specified, TeamCity uses the Java home of the build agent process itself.

JVM command line parameters	You can specify such JVM command line parameters, e.g. maximum heap size or parameters enabling remote debugging. These values are passed by the JVM used to run your build. Example: `-Xmx512m -Xms256m`

Duplicate Finder Settings

Option	Description
Test sources	If this option is checked, the test sources will be included in the duplicates analysis. Tests may contain the data which is duplicated intentionally, and verifying tests for duplicates may yield a lot of results creating long builds and "spamming" your reports. We recommend you not select this option.

Include / exclude patterns| Optional, specify to restrict the sources scope to run duplicates analysis on. For details, refer to the section below| #IdeaPatterns]|

Detalization level	Use these options to define which elements of the source code should be distinguished when searching for repetitive code fragments. Code fragments can be considered duplicated if they are structurally similar, but contain different variables, fields, methods, types or literals. Refer to the samples below:
Distinguish variables	If this option is checked, the similar contents with different variable names will be recognized as different. If this option is not checked, such contents will be recognized as duplicated: `public static void main(String[] args) {` `int i = 0;` `int j = 0;` `if (i == j) {` `System.out.println("sum of " + i + " and " + j + " = " + i + j);` `}` `long k = 0;` `long n = 0;` `if (k == n) {` `System.out.println("sum of " + k + " and " + n + " = " + k + n);` `}` `}`

Distinguish fields	If this option is checked, the similar contents with different field names will be recognized as different. If this option is not checked, such contents will be recognized as duplicated: `myTable.addSelectionListener(new SelectionListener() {` `  public void widgetDefaultSelected(SelectionEvent e) {` `  }` `  /*.....**/` `});` `myTree.addSelectionListener(new SelectionListener() {` `  public void widgetDefaultSelected(SelectionEvent e) {` `  }` `  /*.....**/` `});`

Distinguish methods	If this option is checked, the methods of similar structure will be recognized as different. If this option is not checked, such methods will be recognized as duplicated. In this case, they can be extracted and reused. Initial version: ``` public void buildCanceled(Build build, SessionData data) { /* ... **/ for (IListener listener : getListeners()) { listener.buildCanceled(build, data); } } public void buildFinished(Build build, SessionData data) { /* ... **/ for (IListener listener : getListeners()) { listener.buildFinished(build, data); } } ``` After analysiing code for duplicates without distinguishing methods, the duplicated fragments can be extracted: ``` public void buildCanceled(final Build build, final SessionData data) { enumerateListeners(new Processor() { public void process(final IListener listener) { listener.buildCanceled(build, data); } }); } public void buildFinished(final Build build, final SessionData data) { enumerateListeners(new Processor() { public void process(final IListener listener) { listener.buildFinished(build, data); } }); } private void enumerateListeners(Processor processor) {/* ... **/ for (IListener listener : getListeners()) { processor.process(listener); } } private interface Processor { void process(IListener listener); } ```

Distinguish types	If this option is checked, the similar code fragments with different type names will be recognized as different. If this option is not checked, such code fragments will be recognized as duplicates. `new MyIDE().updateStatus()` `new TheirIDE().updateStatus()`
Distinguish literals	If this option is checked, similar line of code with different litarels will be considered different If this option is not checked, such lines will be recognized as duplicates. `myWatchedLabel.setToolTipText("Not Logged In");` `myWatchedLabel.setToolTipText("Logging In...");`
Ignore duplicates with complexity lower than	Complexity of the source code is defined by the amount of statements, expressions, declarations and method calls. Complexity of each of them is defined by its cost. Summarized costs of all these elements of the source code fragment yields the total complexity. Use this field to specify the lowest level of complexity of the source code to be taken into consideration when detecting duplicates. For meaningful results start with value 10.
Ignore duplicate subexpressions with complexity lower than	Use this field to specify the lowest level of complexity of subexpressions to be taken into consideration when detecting duplicates.
Check if Subexpression Can be Extracted	If this option is checked, the duplicated subexpressions can be extracted.

Include / exclude patterns are newline-delimited set of rules of the form:

```
[+:|-:]pattern
```

Where the pattern must satisfy these rules:

- must end with either ** or * (this effectively limits the patterns to only the directories level, they do not support file-level patterns)
- references to modules can be included as `[module_name]/<path_within_module>`

Some notes on patterns processing:

- excludes have precedence over includes
- if include patterns are specified, only directories matching these patterns will be included, all other directories will be excluded
- "include" pattern has a special behavior (due to underlying limitations): it includes the directory specified and all the files residing directly in the directories above the one specified.

Example:

```
+:testData/tables/**
-:testData/**
-:testdata/**
-:[testData]/**
```

For the file paths to be reported correctly, "References to resources outside project/module file directory" option for the project and all modules should be set to "Relative" in IDEA project.