

Programmers' Technical Reference Guide for Titanium

Kristóf Szabados, Ádám Knapp

Version 11.2.1, 2026-08-05

Table of Contents

1. About the Document . . . 2
   1.1. Purpose . . . 2
   1.2. Target Groups . . . 2
   1.3. Typographical Conventions . . . 2
2. Adding a new Code Smell . . . 3
   2.1. Adding the new Code Smell class . . . 3
      2.1.1. The location of the new Code Smell class . . . 3
      2.1.2. The description of the new Code Smell class . . . 3
      2.1.3. The superclass of the new Code Smell class . . . 3
      2.1.4. The code of the new Code Smell class . . . 4
   2.2. Register the new Code Smell . . . 5
      2.2.1. Register the new Code Smell Type . . . 5
      2.2.2. Add the new Code Smell to semantic problem related map . . . 5
      2.2.3. Register the new Problem Type Preference . . . 6
      2.2.4. Initialize the preference of the Code Smell . . . 6
      2.2.5. Refresh Markers Preference Page . . . 7
   2.3. Refresh Titanium documentation . . . 10
3. Titanium metrics . . . 11
   3.1. Overview . . . 11
   3.2. Metrics . . . 11
   3.3. MetricData . . . 12
   3.4. ModuleMetricWrapper . . . 12
   3.5. MetricsView . . . 12
   3.6. TopRiskView . . . 12
   3.7. Interaction with the titan designer . . . 12
4. Graph generation and display . . . 13
   4.1. The generation of graphs on the UI . . . 13
   4.2. The generation of graph on the headless interface . . . 14
   4.3. How graph data is obtained? . . . 15
      4.3.1. Obtaining data for module graph . . . 15
      4.3.2. Obtaining data for component graph . . . 16
5. Graph clustering . . . 17
   5.1. Algorithms . . . 17
      5.1.1. Clustering by folder name . . . 17
      5.1.2. Clustering using regular expressions . . . 17
      5.1.3. Clustering by module name . . . 17
      5.1.4. Automatic clustering . . . 17
   5.2. Running the algorithms . . . 18

5.3. Connection with TITAN designer . . . 18
6. Titanium DAG layout algorithm . . . 19
6.1. Basic idea . . . 19
6.2. Versions . . . 19
6.3. Display . . . 19
7. Searching for parallel paths and cycles . . . 20
7.1. Finding parallel paths . . . 20
7.2. Finding circles . . . 20
8. Code smell table merging . . . 21
8.1. Algorithm . . . 21
8.2. Limitation . . . 21
9. References . . . 22

Abstract

This document describes detailed information on writing components of executable test suites for the TITAN TTCN-3 Toolset.

Copyright

Copyright (c) 2000-2026 Ericsson Telecom AB.
All rights reserved. This program and the accompanying materials are made available under the terms of the Eclipse Public License v2.0 that accompanies this distribution, and is available at https://www.eclipse.org/org/documents/epl-2.0/EPL-2.0.html.

Disclaimer

The contents of this document are subject to revision without notice due to continued progress in methodology, design and manufacturing. Ericsson should have no liability for any error or damage of any kind resulting from the use of this document.

Chapter 1. About the Document

1.1. Purpose

The purpose of this document is to provide detailed information on writing components, for example, test ports, and so on, for executable test suites.

1.2. Target Groups

This document is intended for programmers of TTCN-3 test suites with information in addition to that provided in the TITAN User Guide [3]. It is recommended that the programmer reads the TITAN User Guide before reading this document.

1.3. Typographical Conventions

This document uses the following typographical conventions:

Bold is used to represent graphical user interface (GUI) components such as buttons, menus, menu items, dialog box options, fields and keywords, as well as menu commands. Bold is also used with "+" to represent key combinations. For example, **Ctrl+Click**

The character "/"" is used to denote a menu and sub-menu sequence. For example, **File / Open**.

`Monospaced` font is used represent system elements such as command and parameter names, program names, path names, URLs, directory names and code examples.

`Bold monospaced` font is used for commands that must be entered at the Command Line Interface (CLI).

Chapter 2. Adding a new Code Smell

This chapter guides you through the world of Code Smells. It contains a tutorial like description on how to add a new Code Smell, which are the important things you have to pay attention to, and why. A concrete example is provided to help understanding the concepts and usage.

2.1. Adding the new Code Smell class

2.1.1. The location of the new Code Smell class

The new code smell-s class is actually a code smell spotter implementation. That is why you have to add the new class in the

package org.eclipse.titanium.markers.spotters.implementation .

The name of the class has to refer to the role of the code smell.

In this description we study about Code Smells using a concrete code smell, which is implemented in the class named Goto.java.

2.1.2. The description of the new Code Smell class

Do not forget to add a description about the role of your new code smell. This should contain your id as author.

In the Goto example code smell a description could be:

```
/**

* This class marks the following code smell:
* The code contains goto statements, which is not recommended.
* @author <XY_id>

*/
```

2.1.3. The superclass of the new Code Smell class

The new Code Smell has to extend the class BaseModuleCodeSmellSpotter. This class is an abstract class in the **package** org.eclipse.titanium.markers.spotters .

Example:

```
public class Goto extends BaseModuleCodeSmellSpotter { ... }
```

2.1.4. The code of the new Code Smell class

- It is recommended for your code smell class to have as private attribute named `ERROR_MESSAGE`, this containing the message which should appear to the user of the Code Smell, when the Code Smell is detected.

  Example:

  ```
  private static final String ERROR_MESSAGE = "Usage of goto and label statements is
  not recommended ";
  ```

- The constructor of the Code Smell should contain `super(CodeSmellType.<Code_Smell_Id>);` as first row, where `Code_Smell_Id` is choosen by the author, and will be used in registering a the new code smell. Information on this id is in section Register the new Code Smell Type.

  Example:

  ```
  public Goto() {
    super(CodeSmellType.GOTO);
  }
  ```

- As the code smell-s superclass, the `BaseModuleCodeSmellSpotter` has two abstract methods, we have to override these. The method `getStartNode()` will have to return the list of AST nodes on which the spotter will work, on which we are interested to analyze the appearance of our code smell.

  Example:

  ```
  @Override
  public List<Class<? extends IVisitableNode>> getStartNode() {
      List<Class<? extends IVisitableNode>> ret =new ArrayList<Class<? Extends
  IVisitableNode>>(1);
      ret.add(Goto_statement.class);
      return ret;
  }
  ```

- The method `process(IVisitableNode node, Problems problems)` has to be overriden as well. The actual work for matching the code smell is done here. This method would surely contain a `problems.report(s.getLocation(), ERROR_MESSAGE);` row.

  Example:

```
@Override
public void process(IVisitableNode node, Problems problems) {
    if (node instanceof Goto_statement) {
        Goto_statement s = (Goto_statement) node;
problems.report(s.getLocation(), ERROR_MESSAGE);
    }
}
```

2.2. Register the new Code Smell

2.2.1. Register the new Code Smell Type

The *CodeSmellType* enum in ***package*** *org.eclipse.titanium.markers.types* contains all the Code Smell Types as instances. We have to add the new Code Smell, using the <*Code_Smell_Id*> which we have chosen in the constructor of our code smell's class.We have to add a short description of the problem detected by our code smell, and we must specify three numeric parameters as well:

1. the minimum time needed to improve the detected code smell;
2. the average time needed to improve the detected code smell;
3. the maximum time needed to correct on instance of the detected code smell.

Example:

```
package org.eclipse.titanium.markers.types;
public enum CodeSmellType implements ProblemType{
    ...
    GOTO("Goto", 1.0, 5.5, 26.0)
    ...
}
```

2.2.2. Add the new Code Smell to semantic problem related map

The class StaticData has the newSpotters() method which returns an unmodifiable map. This map contains the code smell spotters,that are related to each semantic problem. The map's key is the relevant instance of the CodeSmellType, the value is new instance of the code smell class.

This information is used to gather the active code smells spotters, for a code smell type, during the analysis.

Example:

```
package org.eclipse.titanium.markers.spotters.implementation;
class StaticData {

    public static Map<CodeSmellType, BaseModuleCodeSmellSpotter[]> newSpotters() {
        ...
        m.put(CodeSmellType.GOTO, new BaseModuleCodeSmellSpotter[] { new Goto() });
        ...
        return Collections.unmodifiableMap(m);
    }
}
```

2.2.3. Register the new Problem Type Preference

The ProblemTypePreference contains an item related to each Code Smell. This enum is going to serve in the process of setting the preferences of a Code Smell. It also contains a short description about the effect of the related Code Smell. This description going to appear when the user browses in the Titanium Preferences / Code Smell window. Try to be short and very clear.

Example:

```
package org.eclipse.titanium.preferences
enum ProblemTypePreference


public enum ProblemTypePreference {
    ...
    GOTO("Report the usage of label and goto statements",
EnumSet.of(CodeSmellType.GOTO)),
    ...
}
```

2.2.4. Initialize the preference of the Code Smell

All Code Smells have a default preference. This must be given in the class PreferenceInitializer.

There are three kind of preferences: *ERROR, WARNING, IGNORE*. If *ERROR* is set, the detecting of the referred Code Smell is going to occur an error. In the case of *WARNING* the user is going to get a warning message if the referred Code Smell is detected. *IGNORE* means that the user is not going to receive any sign, even if the Code Smell exists.

Example:

```
package org.eclipse.titanium.preferences;
class PreferenceInitializer

    public final void initializeDefaultPreferences() {
        IPreferenceStore preferenceStore = getPreference();
        ...
            preferenceStore.setDefault(ProblemTypePreference.GOTO.getPreferenceName(),
                GeneralConstants.IGNORE);
        ...
}
```

2.2.5. Refresh Markers Preference Page

The class MarkersPreferencePage is the place where users can set the severity of each code smell. This page also contains the explanation on the program code containing the specific Code Smell, could be written more clearly, correct, why is it important to pay attention on it. This explanation appears to the user in the Titanium Preferences / Code Smell window, when the mouse is set above the short description of the code smell.

Example:

```
package org.eclipse.titanium.preferences.pages;
class MarkersPreferencePage

static {
    Map<ProblemTypePreference, String> m = new EnumMap<ProblemTypePreference, String>
        (ProblemTypePreference.class);

        ...
        m.put(ProblemTypePreference.GOTO,
        "In almost all cases the usage of goto should be forbidden as it can very
easily
        breaks the principles of structured/well designed source code.");
        ...

}
```

The function createFieldEditors() is responsible for creating the fields. Only the fields created in this function going to be seen by the user in the Titanium Preferences / Code Smell window. Be sure you put the new field in the block corresponding to the type of problem detected by your Code Smell.

Example:

```
package org.eclipse.titanium.preferences.pages;
class MarkersPreferencePage


protected void createFieldEditors() {
  ...

    addField(b1);
    {
        ...
        sec.setText("Potential Structural problems");
        sec.setFont(fonts.getBold(""));
        Composite comp = new Composite(sec, 0);
        comp.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
        comp.setLayout(new FillLayout(SWT.VERTICAL));
        createField(comp, ProblemTypePreference.GOTO);
        ...
    }
}
```

Using parameters

It is also possible to ask for parameters from the user on this page. In which case the parameter must also be a preference setting, and have a place on the preference window, preferably next to its main option. Please don't forget to provide a default value for each parameter, so that the code smell can work out of the box if needed.

In the following example we would like to have a minimum length for identifiers. This threshold has a default value, but it should be changeable by the user. Figure 1 shows a sample window, consisting of all the parameters which are responsible for the minimum lengths of the identifiers. The default value is 4, but the user can change the threshold values any time here.

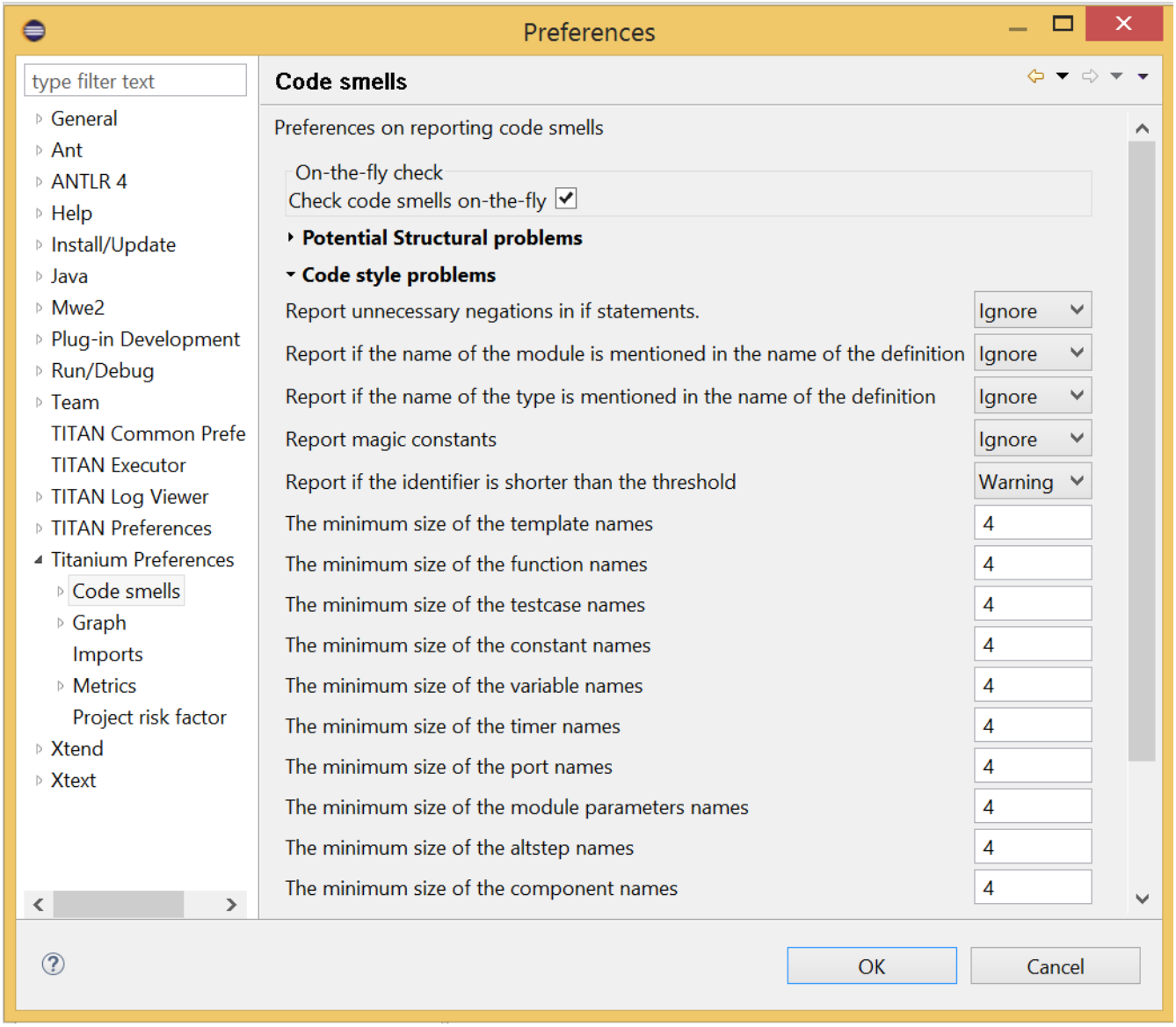


Figure 1. Preferences window with user parameters

Initializing process of a parameterized code smell looks like the previously shown code smells, however there are some differences about the GUI parameters.

To define the parameters, the PreferenceInitializer must be completed.

Example:

```
package org.eclipse.titanium.preferences;
class PreferenceInitializer

preferenceStore.setDefault(
    PreferenceConstants.IDENTIFIER_EXCESSIVELY_SHORT_TEMPLATE_SIZE, 4);
```

Constant need to be added to PreferenceConstants.

Example:

```
package org.eclipse.titanium.preferences;
class PreferenceConstants

public static final String IDENTIFIER_EXCESSIVELY_SHORT_TEMPLATE_SIZE =
    "IDENTIFIER_EXCESSIVELY_SHORT_TEMPLATE_SIZE";
```

2.3. Refresh Titanium documentation

The last, important step is to refresh the documentation. In the chapter Titanium Preferences / Code Smell Preferences are presented several Code Smell categories. Search for the category your Code Smell belongs in the Titanium Preferences/Code Smell window, use the short description of the code smell from the Preferences/Code Smell window l to introduce the new description.

Example:

In docs/Titanium_Description

Titanium Preferences / Code Smell Preferences / Potential structural problems

- **Report the usage of label and goto statements**: goto statements and labels usually indicate incorrect programing practices. Developers are encouraged to used elements of structured programming practices.

Chapter 3. Titanium metrics

3.1. Overview

One of the core features of Titanium is the code metrics for TTCN-3 projects. Code metrics (like block nesting of a test case, or cyclomatic complexity of a function) are indicators of code quality, that can tell the user about which are the complex modules of a project, and may serve as a lead in refactoring by showing overly complicated, bloated parts of the project.

Calculating metrics is planned to happen rarely, only when the user opens explicitly an eclipse view that requires the metrics, thus it was not designed to be lightweight.

The code of the implementation resides in the `org.eclipse.titanium.markers` and its submodules. A rough overview of some notable classes and its associations is depicted below.

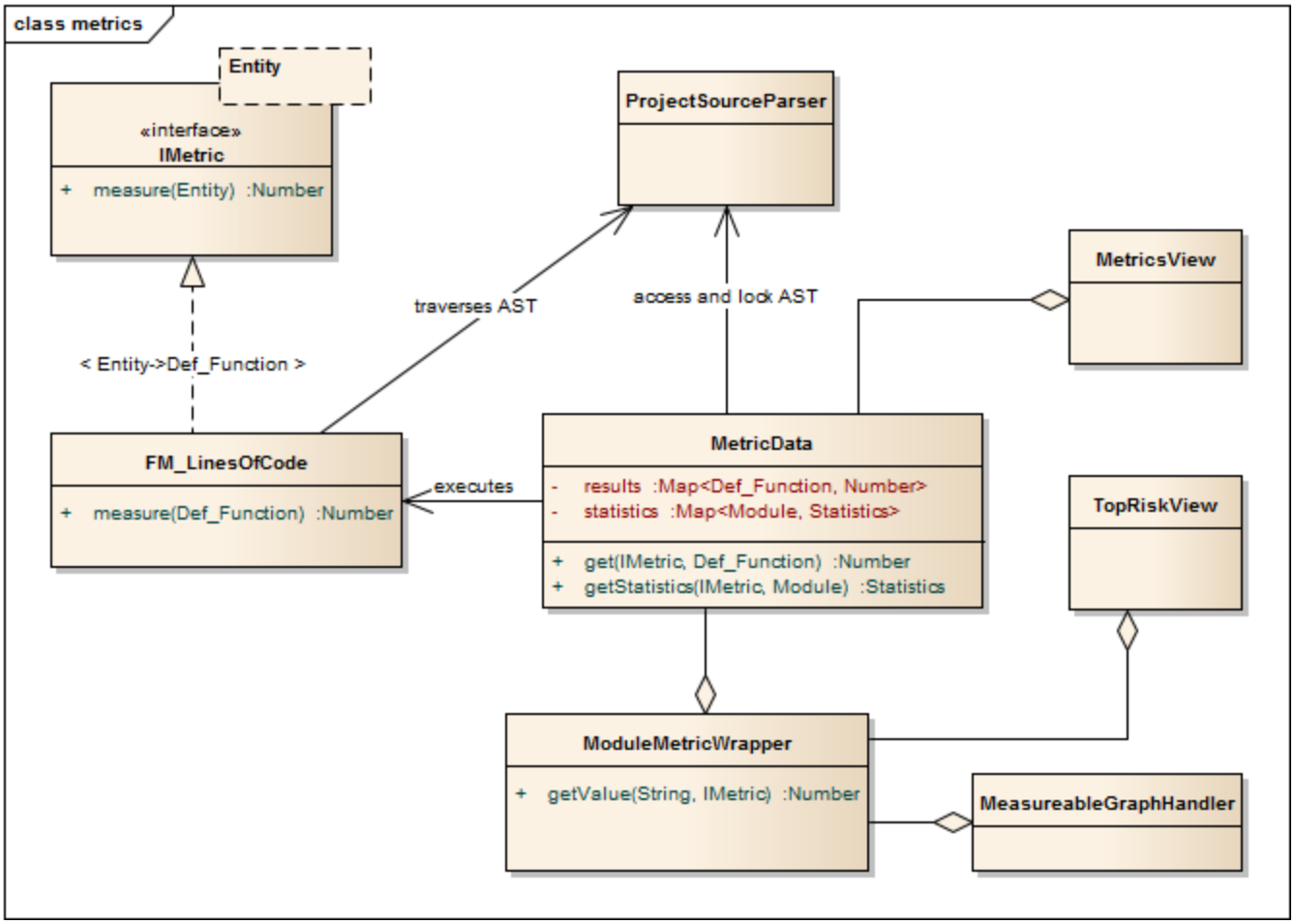


Figure 2. Associations of notable classes related to the metrics package

3.2. Metrics

Metrics are similar to the code smell spotters in the `marker` package. Their purpose is to measure a TTCN-3 entity (e.g. count the length of a function definition). They can contain local state (to ensure flexibility), but after initialization this state should not be changed, so their measure method is intended to serve a stateless behavior.

3.3. MetricData

This is the core class of the package that controls the execution of the metrics, stores the results of them, calculates its statistics and risk factors. It is immutable, in fact a snapshot of the project's state and quality at the time of creating the `MetricData` object. As a result, it is quite a heavyweight object, which should be created sparingly and cached when possible.

Note that constructing an instance requires locking the project to prevent modifications during the measurements. This locking is handled internally.

3.4. ModuleMetricWrapper

During development it was an important aspect to ensure an easy-to-use interface for the graph package, and also that the graph package and the `MetricData` class is not tightly coupled. The `ModuleMetricWrapper` class is the façade that solves this.

3.5. MetricsView

This view gives the user an overview of the project from the point of view of the metrics. The project is explorable via a tree view, where the user can see all the metrics, and under those nodes the entities related to these metrics, and also the detailed results of the metrics.

The view uses a `MetricData` instance as its data source, constructed when the view is opened.

3.6. TopRiskView

This view is designed to show the users the modules that are potentially dangerous according to the metrics, that is, that have bad quality index for the metrics. When using this view, the user can select which metrics to count in the quality index, and than we list the modules of the project ordered by this quality index (along with the details of the metrics).

As here we are interested only in the modules, this class is not in direct association with a `MetricData` instance, but rather uses a `ModuleMetricWrapper` to show the metric details.

3.7. Interaction with the titan designer

This module has a single point where the designer is directly touched. The `MetricData` instance must access to the AST of the project, so it can execute the metrics on the entities that should be measured. This happens in the `measure()` method of the `MetricData`, where the module nodes are queried from the `ProjectSourceParser` associated with the project that we measure.

On the other hand, the metrics themselves rely on the methods of the AST nodes, and AST traversal, so this should be also considered as part of the interface.

Chapter 4. Graph generation and display

4.1. The generation of graphs on the UI

The graph generation can be basically divided into two parts:

1. The generation of component graph
2. The generation of module graph

These two finally use the same methods for the display, but the graph creation is totally different.

Clicking in the appropriate menu entry first launches an `AbstractHandler`, this can be `ModuleGraphAction` or `ComponentGraphAction` according to the claimed graph. These classes only do a search for the appropriate graph drawing window, set a reference to the opened window, or create a new window if needed. After opening the window the `EditorPart` takes the control. Both the component and module windows are inherited from `GraphEditor` class, this class implements basic `EditorPart` methods, and besides it creates the UI elements needed for showing the graph (only the common UI elements are built here).

For the graph building (generation) we use `GraphGenerator`, this class implements a frame how a generator should look like: It can generate graph, return the generated graph (this method is synchronized with the generation), and it can refresh the graph searching for changes. The most important method here is `createGraph()` which is protected, and abstract.. According to the claimed graph we make instance of the appropriate subclass, thus all the differences are solved in the graph drawing. The two present subclasses of `GraphGenerator` are `ComponentGraphGenerator` and `ModuleGraphGenerator`.

After this step the editor window takes back the control, and displays the graph through JUNG API. Because of synchronization issues the editor doesn't wait for the generator, the generator can set a new graph and thus launch the process of display through a method call on the inherited `SetGraph()` method. Here the most important steps are the following:

1. Handling of `DrawArea` (this is a Swing component that displays the graph).
2. Handling of `SatelliteView` (this is handled through a reference).

As Jung is compatible with Java AWT (it finally returns a subclass of AWT `Component`), we add everything to these two swing objects.

To make a `Component` from our generated graph we use `GraphHandler` class. This class provides a quite complex functionality, let's see the main ones:

1. Visualizing a graph (make a drawable object from inner representation)
2. Save the graph to an output file (pajek or image)
3. Change the graph's layout
4. Set custom rendering for graph nodes (font style, node shape, node colour,...)
5. Other, less important lightweight UI features.

Finally the visualized graph is added to the `DrawArea`, and `SatelliteView` in the `GraphEditor` class (this class calls the `GraphHandler`).

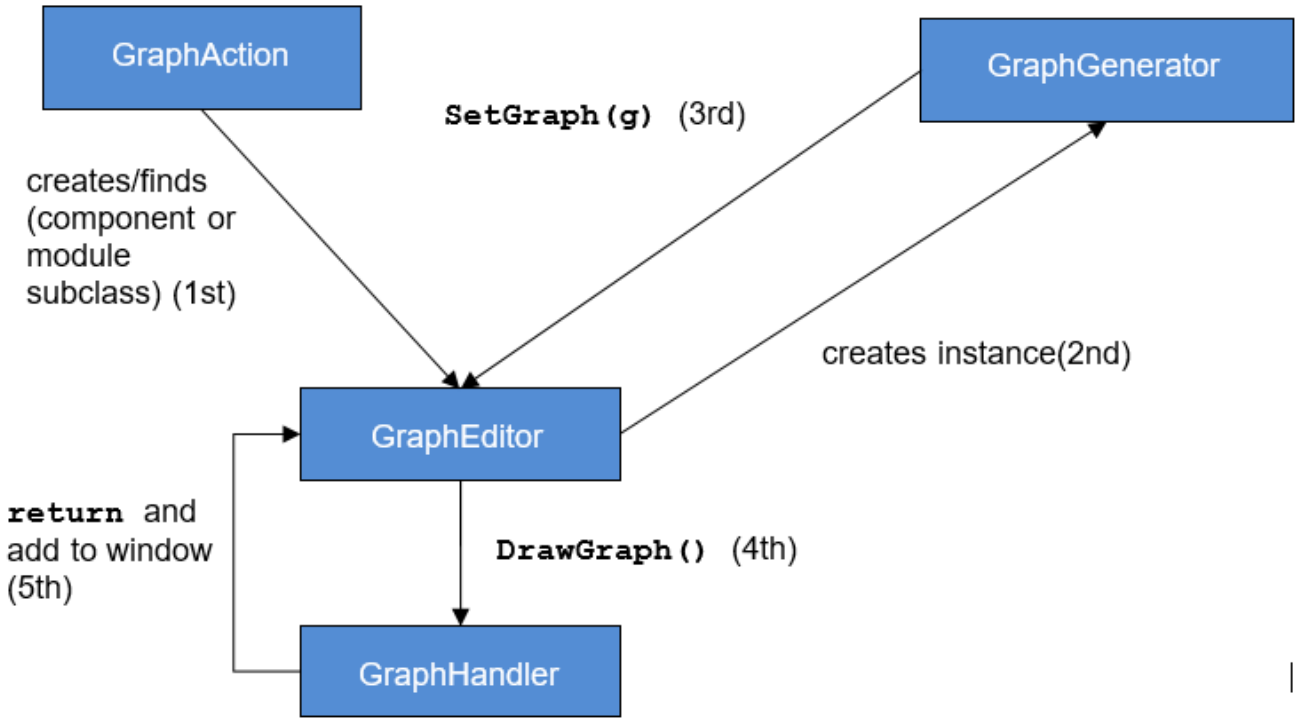


Figure 3. The 5 steps of graph drawing

4.2. The generation of graph on the headless interface

Compared to the UI graph building headless building is simplier. `SaveModuleGraph` or `SaveComponentGraph` launches the generation. The called application looks for all projects in your workspace, and generates a graph for each TTCN project.

To generate graph an appropriate `GraphGenerator` subclass is instanced and called (`ModuleGraphGenerator` or `ComponentGraphGenerator`). This class generates the inner representation of the claimed graph.

After this the application calls the `SaveGraphToPajek` method of `GraphHandler`, this method is static, therefore it needs no instance to build up. This call takes as parameter the graph, which will be claimed from the generator through `getGraph()` call (note that this is a significant difference from the UI case). As this method is synchronized with the generation the call will only return after the generation is done, so this call maybe longer. After all the `GraphHandler` writes out the graph to the disk in Pajek .net format. If there was no problem during the run the application returns here, and the run terminates.

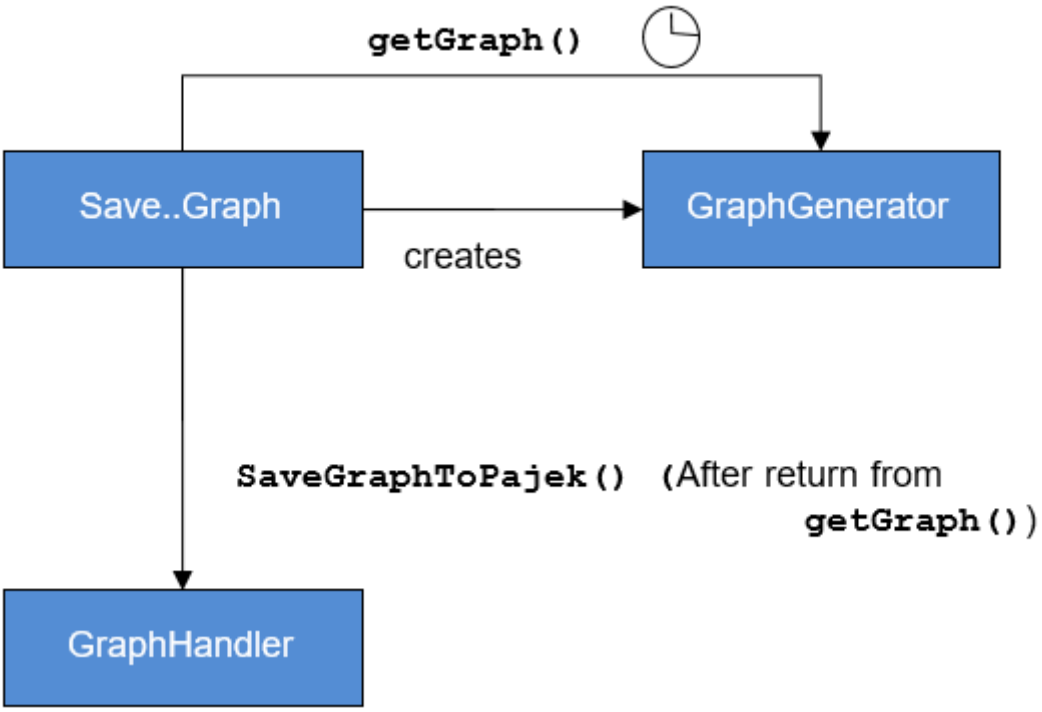


Figure 4. Generating graph on headless interface

4.3. How graph data is obtained?

In this section we are going to speak about the connection among Titanium, and TITAN designer. As the graphs always represent the current state of the TTCN code naturally this connection is unavoidable. Just like in the former sections we are going to discuss two cases, which are now quite different:

1. Module graph
2. Component graph

Both generations are implemented in the `GraphGenerator` classes, more precisely at the `createGraph()` method.

4.3.1. Obtaining data for module graph

Module graph data can be obtained a bit easier. The following steps are done in the appropriate method:

1. Creating an instance of `ProjectSourceParser`
2. Checking whether it is up to date
3. Analyzing if it wasn't up to date
4. Iterating through all the visited projects, here visited project are got through `ProjectBasedBuilder`
5. Getting a `ProjectStructureDataCollector` object through `GlobalProjectStructureTracker` object for the current project

6. Get `knownModules`, `missingModules` and importations from the collector. From these values the graph can be directly generated

4.3.2. Obtaining data for component graph

For component graph we need to use a bit more complicated interface. Up to step 4 we do the same as at generating module graph. After this we do the following steps:

1. Iterate through `knownModules`
2. Override the `accept()` method of the current module by a new `ASTVisitor`
3. Check whether the visited node is a `Component`
4. If it was a component then we get its `Identifier`, otherwise we return
5. We can create the base node through the datas provided in the Identifier
6. We get the extensions and the extension attributes through the original `Component_Type` object
7. We override there the `accept()` method again, and we work with `Reference` objects only (otherwise we return)
8. Build up a new node through the `Reference`'s `Identifier`, and add an edge from the first node to the second one.

Chapter 5. Graph clustering

5.1. Algorithms

The classes of the implemented algorithms are all subclasses of `BaseCluster`. The algorithms are the following:

5.1.1. Clustering by folder name

This algorithm is implemented in the `FolderNameCluster` class.

We recursively check the Resources beginning from the project root directory.

If the currently checked resource is a directory, we create a cluster and continue the recursive search.

If the currently checked resource is a file, we extract the contained module and assign it to the appropriate cluster.

5.1.2. Clustering using regular expressions

This algorithm is implemented in the `RegexpCluster` class.

We iterate through the nodes, and check which regular expressions match the module name.

Then we create the clusters by iterating through the nodes again.

If no matches were found for a given node, then we assign that node to the cluster that contains the nodes which have no matches.

If one match was found, we assign the node to the cluster belonging to the matching regular expression.

If more than one match was found, we display them, and the clustering fails.

5.1.3. Clustering by module name

This algorithm is implemented in the `ModuleNameCluster` class.

First, we create the clusters by iterating through the nodes and splitting the module names according to the settings. A cluster is created for every name segment.

Then we iterate through the nodes again and choose the narrowest cluster the module belongs to, because we want the nodes to belong to only one cluster.

5.1.4. Automatic clustering

This algorithm is implemented in the `AutomaticCluster` class.

This clustering uses a similar algorithm to the one in the article here: S. Mancoridis, B. S. Mitchell, C.

Rorres, Y. Chen and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242), Ischia, Italy, 1998, pp. 45-52, doi: 10.1109/WPC.1998.693283. Available: https://www.cs.drexel.edu/~spiros/papers/iwpc98.pdf (2020-11-09)

The changes are documented in the javadoc of the class.

5.2. Running the algorithms

Running the algorithm is simple. We create the appropriate clustering object and call its `run(monitor, group)` method.

The monitor parameter is the progress monitor. The group parameter is a Boolean value which tells the tool whether only the clusters are needed, or we create a graph from the clusters.

If we want to display the results, we simply override the `drawGraph()` method as it will be called at the end of the `run` method.[1]

It is possible to circumvent the `run` method and create and obtain the clustering using the public interface (for more information consult the javadoc), but using the above method is much simpler.

5.3. Connection with TITAN designer

Only folder name based clustering uses TITAN directly. This algorithm uses a `ProjectSourceParser` object to obtain module paths, thus it can create a directory hierarchy.

[1] These overrides are already done at **ModuleGraphEditor** class

Chapter 6. Titanium DAG layout algorithm

The layout algorithm, which creates a tree-like layout for directed acyclic graphs in the `jung` graph library, was inefficient and could not handle cycles in the graph, so we created our own.

6.1. Basic idea

Since strongly connected components (cycles in this case) are not common in a module or component graph, it makes sense to search for nodes that are not in a cycle.

So we begin by creating a topologic ordering of the nodes by choosing the node that has no arcs going into it (in-degree = 0). We "delete" this node, and decrease its neighbours' in-degree.

If the next node (ordered by in-degree) does not have 0 in-arcs, then it is in a cycle. We find this cycle using DFS.

6.2. Versions

There are two versions. Basically the topologic order can be created by finding sources (no in-arcs) or by finding sinks (no out-arcs). These are implemented in the `TitaniumDAGLayoutAlgorithm` and `TitaniumDAGLayoutReverseAlgorithm` classes respectively.

The obtained order of the nodes and the level at which they will be drawn differ. If we search for sinks, modules that are not importing other modules come first. If we search for sources, modules that are not imported come first.

6.3. Display

The display is done through `TitaniumDAGLayout` class. It implements the `Layout` interface of Jung. After this everything works the same way as any other Jung layout.

Chapter 7. Searching for parallel paths and cycles

The module and component graphs of a project should be trees. A tree does not contain cycles and parallel paths.

7.1. Finding parallel paths

The tool implemented in `CheckParallelPaths` finds the arcs that are contained in parallel paths from a given source node or from every node.

We use a modified DFS algorithm. If we find an already checked node, then we have either found a parallel path or a cycle. We check if it is not a cycle and put the arcs of the two parallel paths in a set.

This method does not find every parallel path, but every arc contained in a parallel path will be in the set.

7.2. Finding circles

`CircleCheck` class implements a circle searching algorithm for graphs. It can be instanced by providing a Jung graph. After this `IsCyclic` method returns whether the graph contained any circle. And `getCircles` method returns all the found circles.

Note that this class may not find all circles, however it finds if there is any circle for sure. Finding all circles is quite a difficult exercise, as circles may even overlap, which causes difficulties.

Chapter 8. Code smell table merging

For the project analysis we export code smells to an excel table. This table will only contain the current code smells. We would like to examine how the figures changed in time.

8.1. Algorithm

The merging algorithm is implemented in `MergeExportedMarkerTables`.

First we collect the dates and code smell names contained in the different files. This way, if a code smell name changes, the change will be handled. We can also merge tables containing more than one column.

Then we iterate through the found dates in ascending order and write the appropriate information to a new excel table.

8.2. Limitation

The `.xls` format used by the `jxl` library only supports 255 columns in a table.

Chapter 9. References

- [1] Installation guide for TITAN TTCN-3 Test Executor
- [2] Installation Guide for TITAN Designer and TITAN Executor for the Eclipse IDE
- [3] User Guide for TITAN TTCN-3 Test Executor
- [4] Programmers Technical Reference for TITAN TTCN-3 Test Executor
- [5] Release Notes for TITAN TTCN-3 Test Executor
- [6] TTCN-3 Style Guide
- [7] TTCN-3 Naming Convention
- [8] Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3.Part 1: Core Language European Telecommunications Standards Institute. ES 201 873-1 Version 4.5.1, April 2013
- [9] Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3.Part 4: TTCN-3 Operational Semantics European Telecommunications Standards Institute. ES 201 873-4 Version 4.4.1, April 2012
- [10] Methods for Testing and Specification (MTS);The Testing and Test Control Notation version 3.Part 7: Using ASN.1 with TTCN-3 European Telecommunications Standards Institute. ES 201 873-7 Version 4.5.1, April 2013