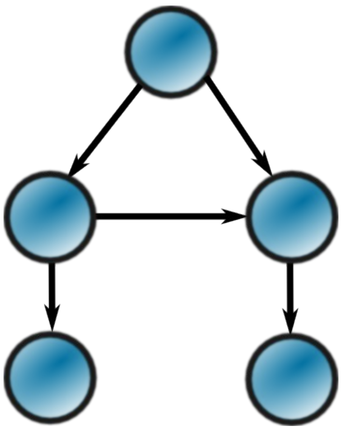
HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Control Flow Graph Generator



Documentation

Aldi Alimucaj

alalimuc@htwg-konstanz.de

June/2009

Version 1.0

Table of Contents

1 General information about the Project

The main idea of this project was to build a plug-in for Eclipse which enables the user to generate control flow diagrams for measurement and learning purposes. Eclipse was a good choice because it is wide spread and offers excellent interfaces to extend it. This documentation is aimed to readers who want to know more about the way it works and why. The main algorithms are explained in details as they make the most important functionalities. Other useful information can be retrieved from the Java-Doc or the web page as well. To explain how the plug-in works in practice, many code and diagram examples are shown. But they do not have specific meaning or are subject of special algorithms. They represent a normal or at some point a special case, which might explain the best the Plug-in's features.

Since the beginning of this project it was conceived as open source so that it could contribute to the eclipse community. This plug-in is published under the EPL and includes no warranty.

This documentation is submitted in partial satisfaction of the requirements for the degree of Bachelor of Science.

2 Overview

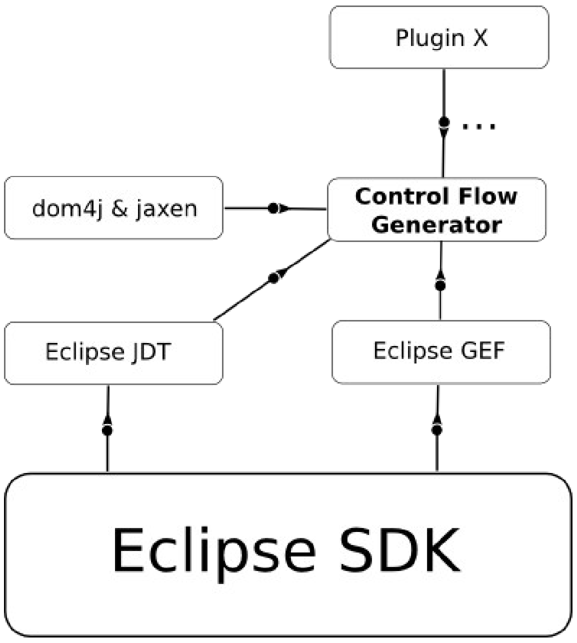
A control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. It refers to the order in which the individual statements, instructions, or function calls of an imperative or functional program are executed or evaluated. If the graph is built from code execution the hole program is first compiled than is executed with special parameters so that a log file is generated to obtain the necessary information to build a graph. This makes it only possible if the code can be complied and run, which is not always the case during testing when parts of the implementation might miss. For this reason the evaluation of the code looked like a nice try to see if it is possible to build a correct graph from static analysis only and not compiling or running any of the code which needs to be analyzed. The methods to do it are explained in details to get a better understanding of how the evaluation of code can lead to a CFG. Normally when building control flow graphs we focus on a specific method and very seldom on hole classes. Classes can get big, but methods as well, thats why for this plug-in special functions like node collapsing and expanding or node hovering where made available to help viewing the graph. The aim was to build CFG-s form methods in the quickies and best way possible without extra clicking and compiling, an all-in-one-click operation.

Purpose

Reachability or different coverage strategies are some of the most common objectives. CFG-s are also build for static analysis reasons. Static analysis is the type of examination which does not consist on the execution of the code being analyzed but rather gaining information from other sources like documentation, reviews, formal methods or automated tools for analysis. From the evaluation only we can not get such information like reachability or coverage. This leads to the idea to use a specialized coverage tool to supply the necessary information about this measurements. After examining some know tools like EMMA, Coverlipse and others [linkCTools] to expand the functionalities, CodeCover turned out as the most fitting white box coverage tool for this plug-in and has been successfully integrated. It is a fully optional feature and can be activated or not without having an impact on the CFG generator.

3 Plugin architecture

Eclipse is a complex application but at the same time has a simple to understand architecture. It is build on its own implementation of the OSGi notion, called Equinox. The framework itself just provides the environment for the modules to operate. With the exception of the run-time kernel, everything else in Eclipse is composed in plug-ins. They are called bundles form OSGi or Plug-ins from Eclipse and have a specific interface to connect to the framework. They can consume and or offer functionalities. Each bundle is loose-coupled with other bundles so that they can be installed, changed or removed while the framework is running [OREclipse]. The next picture illustrates the idea.

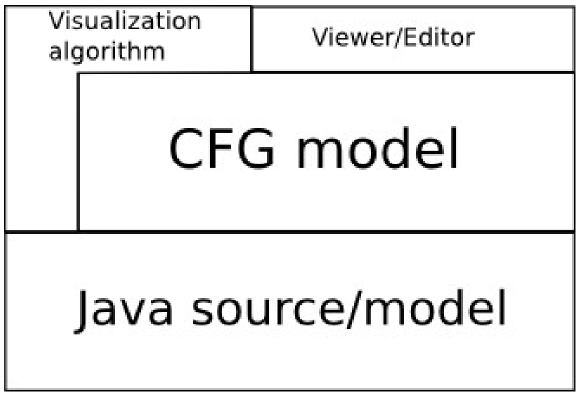


[Figure 3.1 Simplified view plug-in dependencies]

Figure 3.1 is a simplified representation and an example of the the control flow generator dependencies. As a bundle it depends on some bundles which it has to declare as extension and the other bundles can depend on other modules as well to fulfill their tasks. The exact list of the dependencies together with their version as a minimum of acceptance for the plug-in is:

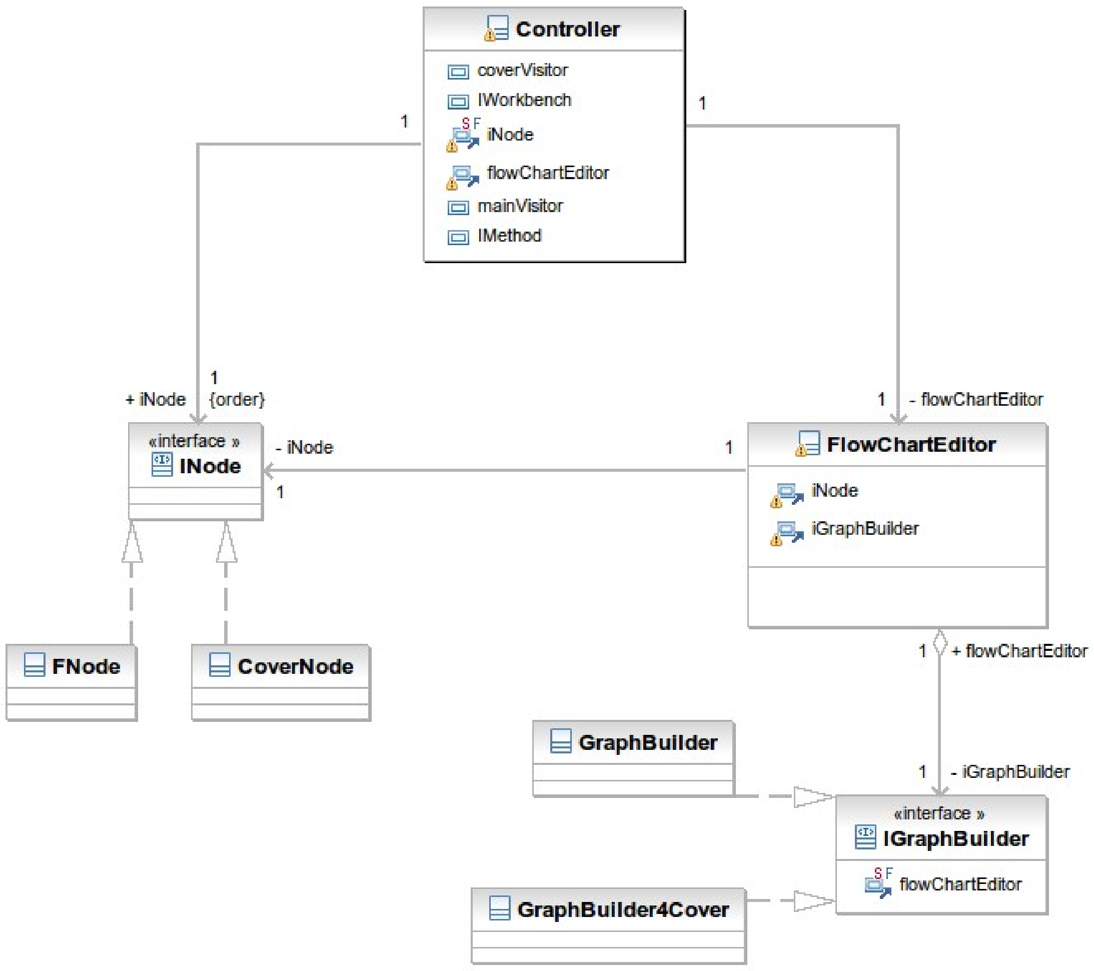
```
org.eclipse.ui,
org.eclipse.core.runtime,
org.eclipse.jdt.core;bundle-version="3.4.4",
org.eclipse.ui.ide;bundle-version="3.4.2",
org.eclipse.gef;bundle-version="3.4.1",
org.eclipse.zest.core;bundle-version="1.0.0",
org.eclipse.zest.layouts;bundle-version="1.0.0",
org.eclipse.core.resources;bundle-version="3.4.2",
org.eclipse.ui.ide.application;bundle-version="1.0.1",
org.eclipse.core.filesystem;bundle-version="1.2.0",
org.d4j_jaxen;bundle-version="1.0.0"
```

The `org.d4j_jaxen` bundle is a module created from the dom4j and jaxen libraries to support the plug-in. If the functionalities of these libraries are loaded as bundles then they are accessed faster then if they were just from a jar.
The plug-in from the bundle point of view as an entity can be divided into 3 major parts which are shown in the next graphic. The algorithms as we will see later, builds the CFG model from the Java source and links the viewer with the the model. The CGF model is a tree representation of the Java source in Java objects which can be persisted and reviewed but not changed.



[Figure 2.2 Plugin architecture]

Figure 2.2 shows at the bottom the Java source, from which the CFG model is built. The CFG model than is linked to the Editor, which is the last step of the generation action. How the model gets to the editor is explained in the next chapters.



[Figure 2.3 Simplified UML Class diagram]

From the UML class diagram we can get a first idea about the most important classes involved in the process. Eclipse plug-ins working with GEF, EMF and GMF generally adopt the MVC pattern but for this case the model is supposed to be generated and not changed so that indirect connections used for that purpose were left out as they are unnecessary.

4 Model

How the model should look like, is trivial. The way to build it by evaluation takes the real effort. If we analyze the Java code we can clearly see its tree shape. It is build on classes that contain methods, attributes and other Java elements. If an element has brackets than it most probably is a parent element with descendants otherwise its a leaf. As long as the elements do not have a special meaning, the data structure of the model is clear. The tree gets just than more complicated and turns int a graph if it contains jumps from elements of one node element to its other siblings, children, parent or other elements. This is the case when loops contain Break/Continue statements or Switch-Cases, where the break statements can decide whether the next statements are going to be executed or not. If-else statements, where one of the blocks is not executed and other cases that turn a tree into a graph and so on. We are going to talk about the processing of this nodes further ones again when we explain their behavior. For now we know that our model is a tree that needs to be processed into a graph.

INode

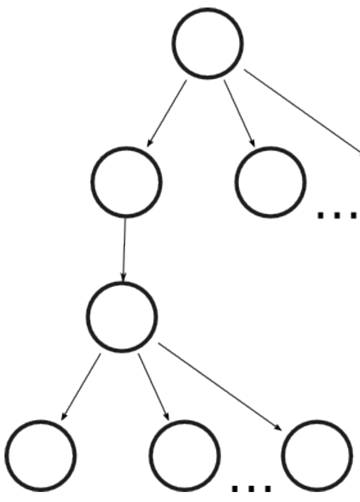
INode is the interface that model classes implement. These classes have a number of functionalities to implement like node name, source code, type and others. For this version there are two classes that implement this interface

- FNode
- CoverNode

For more information please check the Java doc.

FNode and CoverNode

FNode is the class that implements the INode interface and offers the basic functionalities which are needed for the simple graph (simple meaning without coverage information). For the graph which must contain information about the coverage we have the CoverNode class. It extends the FNode and adds some supplementary attributes and functionalities. FNode is a linked list, where every element points at its children, with an unlimited (as far as Java collections can handle) number of children. The visual interpretation of a tree model should look like in the picture 4.1.



[Figure 4.1 Model]

5 Visualization algorithms

We can divide the functionalities into two main parts. The first one is the collecting of information from the Java source code and the second part is the processing of the model form a tree into a graph.

Execution order of the functionalities:

– gaining the Java source
– parsing into a AST
– traversing the AST and building the tree model
– persisting the model
– normalizing
– viewing

Reading the model

To retrieve the information from the Java editor we use the outline view of this editor. By selecting a method the listener changes the focus so that its content can be obtained by other listeners. Eclipse offers a toolkit called JDT(Java Development Toolkit) to handle operations with Java code. To do so we first have to build an AST (Abstract Syntax Tree). After defining the AST the best way to traverse the tree is by using a visitor (ASTVisitor). The abstract class needs to be implemented so that when desired elements are encountered, specific operations are executed. The implementations for this plug-in are called *ASTNodeMainVisitor* and *ASTNodeVisitor4Cover*. The visitor traverses the AST in a depth first way and adds the defined elements to the model. The next table shows the implemented functions and hence the interesting Java elements.

Methods	Information
`visit(ExpressionStatement node)`	Insertion of a node into the model if a expression statement is encountered. This case covers ternary declarations as well. But they are inserted as if statements in the model.
`visit(VariableDeclarationFragment node)`	Insertion of a node into the model if a variable declaration, such as field declaration, local variable declaration or others are encountered.
`visit(IfStatement node)`	Insertion of a node into the model if a if-statement is encountered. This Node contains at least two children as for the graph empty if statements need to be considered as well.
`visit(TryStatement node)`	Insertion of a node into the model if a try-statement is encountered. Contains as many children as catch statements plus the finally statements and the try block.
`visit(ForStatement node)`	Insertion of a node into the model if a for-statement is encountered. It has two children.

Methods	Information
`visit(WhileStatement node)`	Insertion of a node into the model if a while-statement is encountered. There is no difference regarding the possible paths between the for and while statements so that we treat them the same.
`visit(DoStatement node)`	Insertion of a node into the model if a do-statement is encountered. The difference between the do statement and the for or while statements is that loop can be escaped from the end of the block without referencing the top which has no decision power.
`visit(SwitchStatement node)`	Insertion of a node into the model if a switch-statement is encountered. The switch node contains all the cases and/or the default.
`visit(BreakStatement node)`	Insertion of a node into the model if a break-statement is encountered. There is no difference between the loop-break and the switch-break. Cases are treated differently by the time the graph is build. Contains no child nodes.
`visit(ContinueStatement node)`	Insertion of a node into the model if a break-statement is encountered. Contains no child nodes.

ASTNodeMainVisitor

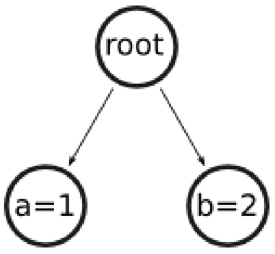
This class implements `org.eclipse.jdt.core.dom.ASTVisitor` and builds the simple tree model. The model has some differences from the original Java source sequence due to optimization for the building of the graph. The children of the root element do not hang on it but point to each other as the next nodes to come.

Example:

```
void expressionTestMethod(int b,int a) {
      a = 1;
      b = 2;
}
```

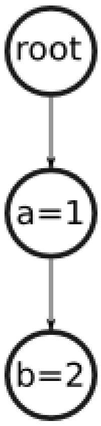
[Code 5.1 expression method]

From the code above we can build a tree representation as is the next picture. This model type would fit for the viewer but it would make it very hard and require extra computations to link children from one branch to anther.



[Figure 5.2: Java tree model]

For this reasons the code in example 5.1 is not represented as in the picture 5.2. Instead it is build as show in the picture 5.3.



[Figure 5.3: FNode representation]

We can agree that this model has a higher level of readability for humans as well. From this example we can see two expression statements which contain no children but the next node. For other nodes it is necessary to use a convention setting the first node of every parent to be the next node from the root and not a child node. Remembering that children of a node can be just nodes within the declared brackets of a statement. An exception is the if-statement but due to the AST parsing these shortcuts are transparent to us.

ASTNodeVisitor4Cover

The class ASTNodeVisitor4Cover does the same operations the class ASTNodeMainVisitor does and adds information useful for the recognition of the covered nodes from the CodeCover sessions such as the offsets to compare the exact position of the nodes, covered boolean and others. For more information please check the Java doc.

Persistence

Ones the model has been built, it gets serialized and saved as a file with the .ff3 extension in a subfolder of the project called cfg. The name is given by convention from the <classname>_<methodname>.ff3. To perform this action a serializing class called NodeSerializer was created. If an older version already exists or same class names with same method names were already examined then a confirmation is shown asking if it should be overwritten or not. Changes to the model are not planed so that the reallocation of the graphical nodes or node folding information are discarded as the editor is closed.
The tree model as it is saved is the plain representation of the Java code with the order changes shown above. Therefore a reorganization of the node must be done before it is been shown.

NodeNormalizer

The node normalizer goes through the main stream and checks if there is a node without its end node. If this is the case it adds the end node which is the last node in the model. The main stream is the straight way to the end of the tree without considering the children. It is necessary to add a common end node where one or many nodes can directly or indirectly connect to.

6 Editor

Eclipse offers a base implementation of all workbench editors called `EditorPart` which can be extended to offer two basic editor types

- textual
- graphical

The editors apply in the most cases the MVC architecture if the model is supposed to change during viewing. In our case the user can not draw a control flow graph on its own but uses the plug-in to generate it, so that a change in the model is not planed. In this case a very useful library from GEF is adopted. The Zest library was developed for this purpose, to show graphics which are not supposed to change their model. The controller or like its called for GEF editors, the `EditPart` implementation in overridden by the `GraphViewer` which is the Zest default controller. Ones its done the Zest-conform model can be loaded. It is based on the two graphical elements which are drawn using the `Draw2D` library, `GraphNode-s and GraphConnection-s.`

Zest Layout Algorithms

Zest is a visualization toolkit for Eclipse. It has a predefined set of classes, interfaces and operations to help building graphics on a GEF editor. It also has drag and drop support for its elements and uses animations in the initial time. The layout algorithms which the library offers are the one listed below

- Spring Layout Algorithm
- Fade Layout Algorithm
- Tree Layout Algorithm
- Radial Layout Algorithm
- Grid Layout Algorithm

None of them is developed to build a control flow graph so that a new one had to be implemented. The way to do that is by extending the `AbstractLayoutAlgorithm` class. The new class is called `GraphLayoutAlgorithm` but not all algorithms were written from scratch. The tree layout algorithm offers a good way to represent a Graph. It is directional and offers a similar flow as the control flow graph. So it made it a good choice to extend and not to start a new one from zero. The nodes in the pictures or snapshots made for this documentation have been repositioned to better fit in the document. They may be arranged differently from the generation.

View

The viewer/editor class is called `FlowChartEditor` and as we already know, it extends `EditorPart` and implements `IAdaptable.` There are to ways to provide it with the model data.

- To call the editor with the data
- To execute the file which opens the editor

The necessary changes to the graph viewer are made from the editor algorithms to adapt the model to the viewer. The classes which implement this functionalities are those implementing `IGraphBuilder.` It defines the method `createView(Graph g, INode node)` for drawing the graph into the editor which defines a graph to be painted on. The method reads the model in a recursive way and builds it from the bottom. As it is defined to have just one end the editor takes advantage of the `NodeNormalizer`. This step was necessary because of the convention that defined the first node as the next node and not as a child node. After the model has been normalized the graph builder is called to paint it into the editor.

GraphBuilder

The `GraphBuilder` as one of the main algorithms is going to be explained in details. To start painting, the builder needs two arguments

- a graph
- the model

The Graph holds the nodes and connections for the editor and the model is a interface of type INode. Series of node types are defined to be processed differently as they occur. The first and easiest one to explain is the node type for an *`EXPRESSION_STATEMENT`*.

Expression statement

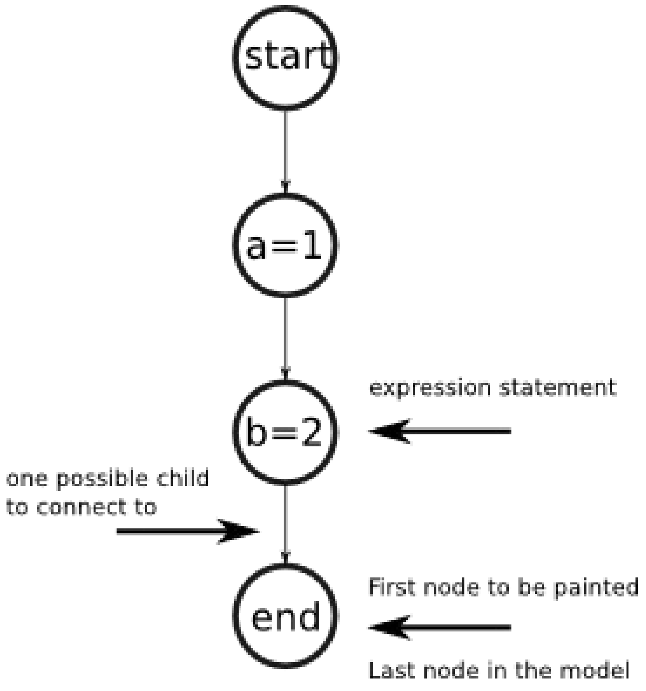
Expression statements are the nodes that do not have child nodes and for this reason the node is created and connected to its next node coming from the recursion.

Example:

```
void expression_statement(int b,int a) {
      a = 1;
      b = 2;
}
```

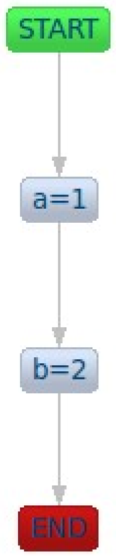
[Code 6.1 expression method]

Figure 6.1 shows the graph model which is generated and ready to be painted.



[Figure 6.1 graph model for the expression statement]

We can see that the recursion path of this model goes strait to the end because no node has children and after painting the last one, a reference of this node is returned so that the previous node can connect to it. After knowing every node to which they should point to, the graph is easily processed. Picture 6.2 shows the result from code sample 6.1 that is built from the plug-in. The start node is painted green to identify it better and for the same reason the end node is colored red.



[Figure 6.2 Flow chart editor graphic, expression statement]

If statements

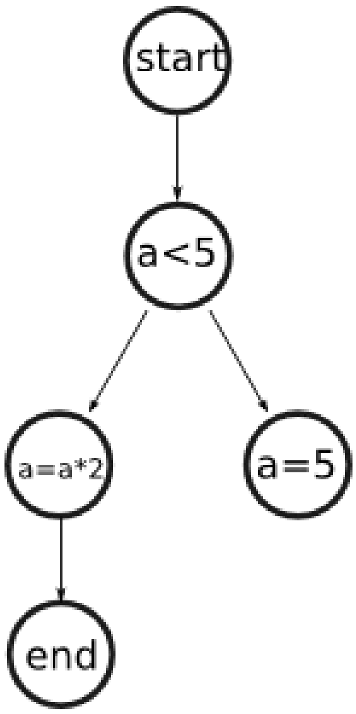
The difference between expression-statements and if-statements is that the last ones can contain child nodes. The second child is the content of the then-expression and the third one of the else-expression but there is no difference between them. We know that just the first child of a node is the next node and not a real child of that node but, as if-statements contain children they need to be referenced as well. Depending on the existence of an else statement there could not be a direct connection between if node and its next node.

In the picture 6.3 we are going to see a simple case of an if statement for the INode model without the else part.

Example:

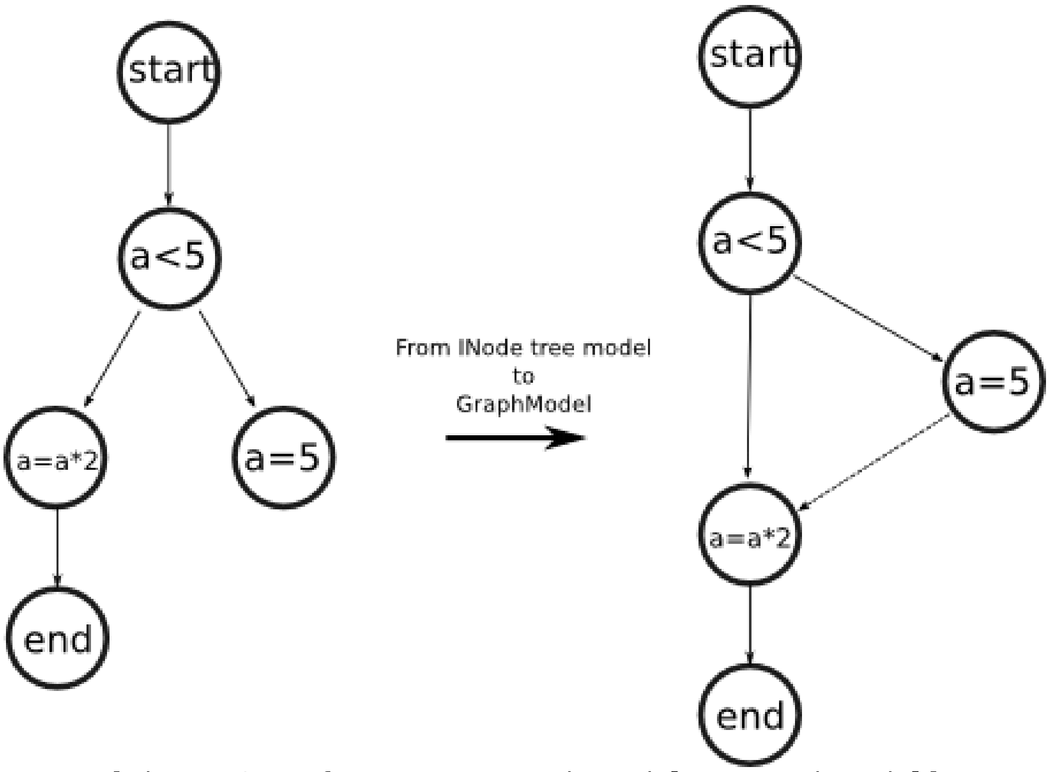
```
void ifTestMethod(int a) {
      if (a < 5) {
            a = 5;
      }
      a = a*2;
}
```

[Code 6.3 If statement]



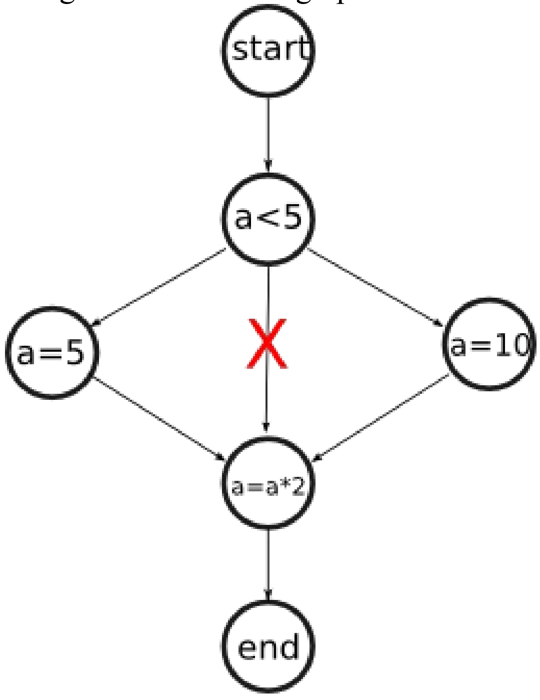
[Figure 6.3 If statement INode representation]

From the node a=5 point of view, we know that its parent's first node is not a sibling but the next element which they should connect to. In the case there are no break and continue statements which can be set if the statement is a child of a for, do or while node. This feature can be attributed the its children as well. Those are going to be discussed when we talk about the loops. In picture 6.4 we can see the transformation from the tree to the graph model.



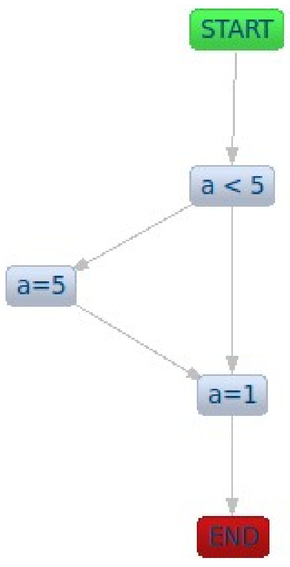
[Figure 6.4 If statement INode model to graph model]

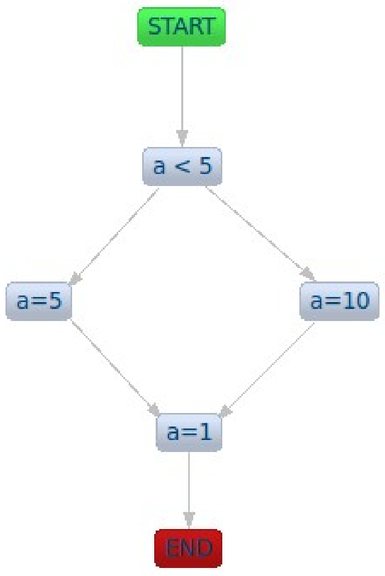
If there is an else-statement then the if-node would have another child and the link between the if-node and its next node would be ignored so that the graph would look like in the picture 6.5.



[Figure 6.5 If else statement]

Both cases as they are built from the plug-in are shown in picture 6.6.





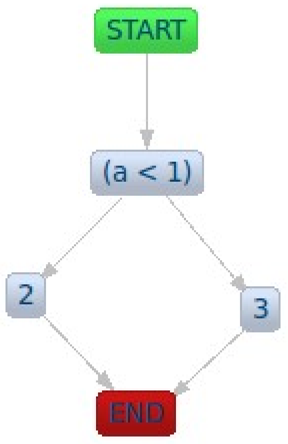
[Figure 6.6 If else statement]

Ternary operator

The ternary operator is basically a short form of the if-else-statement. It has three arguments on the right side and the first one is the expression evaluation. The second and the third arguments are executed depending on this evaluation. For this reason as the choice is made in an if-else-form where both of them are always present, it was good to think of it as an if-else-statement and treat it as such. The ternary statement is added to the tree as an if-node and from that point it is used as such by the painting algorithm.

Example:

```
void ternaryTestMethod(int b,int a) {
      b = (a<1)?2:3;
}
```
[Code 6.4 If statement]



[Figure 6.7 Ternary statement]

For and While statements

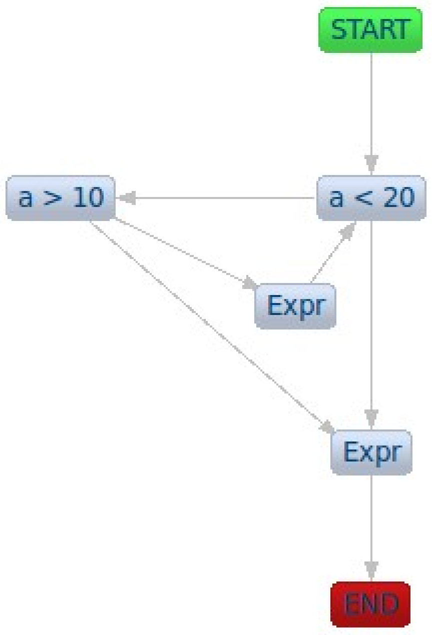
From the logical interpretation there is no difference in the treatment of this two nodes. The expression evaluation is done at the beginning of the block for both cases. So that they are both processed as one case. They have one more child which is the inner part of the statement. It can be of any type and they are recursively build knowing that there is just a parent node and that is the for/while node. This is important for the case that a break or continue statement is encountered and at that point it must be decided whether it should connect to the parent (in case of a continue node) or to its next node (in case of a break node) to exit the block. The next example will show this in practice.

Example:

```
void forTestMethod(int a) {
      for (a = 1; a < 20;) {
            if (a > 10) {
                  break;
            } else {
                  a = a + 2;
            }
      }
      a = a*2;
}
```

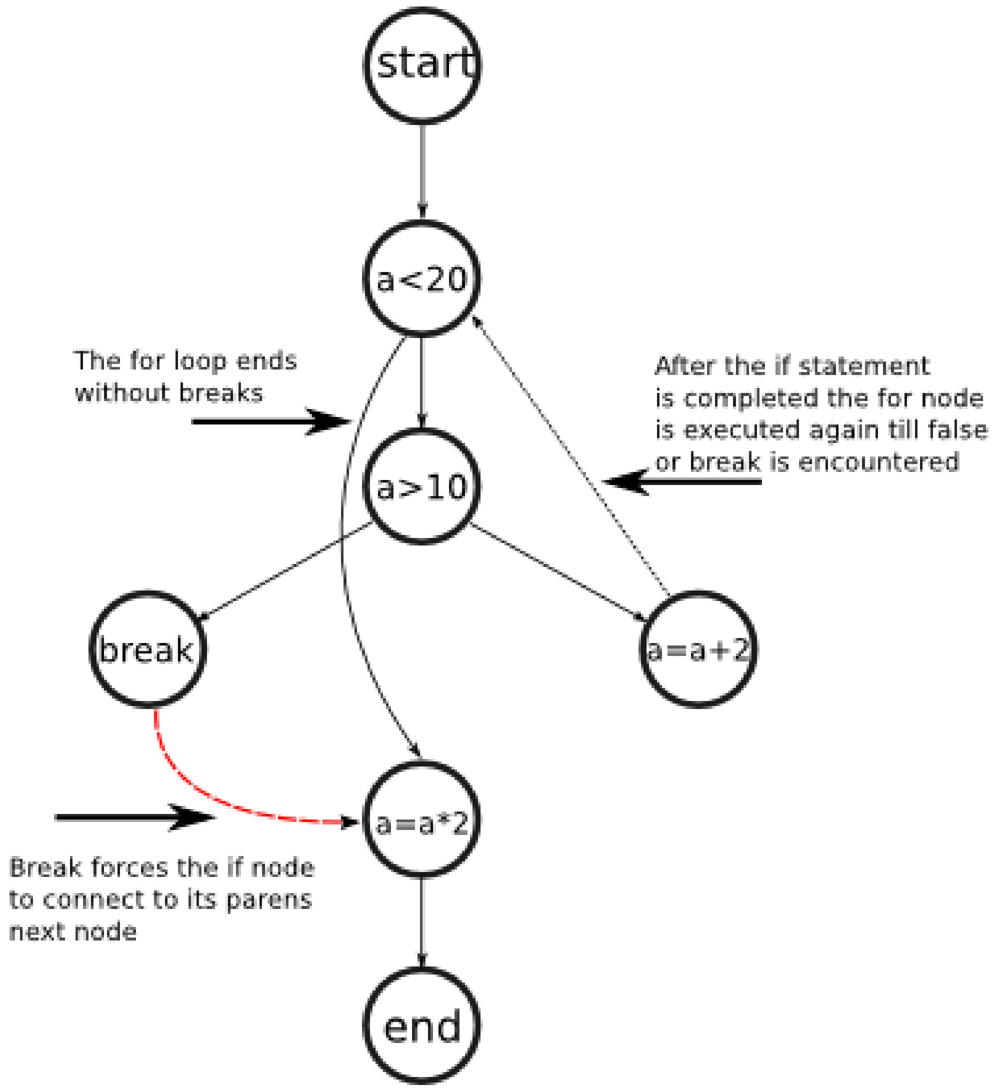
[Code 6.5 For statement]

We can see that the if statement at some point has to resolve to true and the break statement is called. This wouldn't lead to connect the if node to its next node, but to its parents next node. To illustrate this better we can check the result generated from the plug-in in picture 6.8.



[Figure 6.8 For-while statement]

The graphs drown by the plug-in like in this case contain some nodes called Expr which is an abbreviation of veryLongExpressionStatements. From the tree point of view the nodes were connected as shown in the next picture 6.9.


start
a<20
The for loop ends without breaks
After the if statement is completed the for node is executed again till false or break is encountered
a>10
break
a=a+2
a=a*2
Break forces the if node to connect to its parens next node
end

`[Figure 6.9 For-while statement. Tree to Graph model]`

The for node has two children. The first one is the roots next node a=a*2 and the other one is its inner part if-node. When the inner part is called, a reference is given to mark the existence of a loop parent node so that break and continue statements are referenced correctly. Although break statements can only appear from for, do, while and switch statements the reference is useful to indicate which one in case there are more loops nested into each other. Inspecting the if-node we realize that it has just two children instead of tree like any other if-node. This is the case when there is no next node from the root and the node isn't part of the main stream. The main stream is the direct way to the end without considering the children nodes (see *NodeNormalizer)*. It is assumable that its next node should be the parents next node or deviations like in the cases showed above.

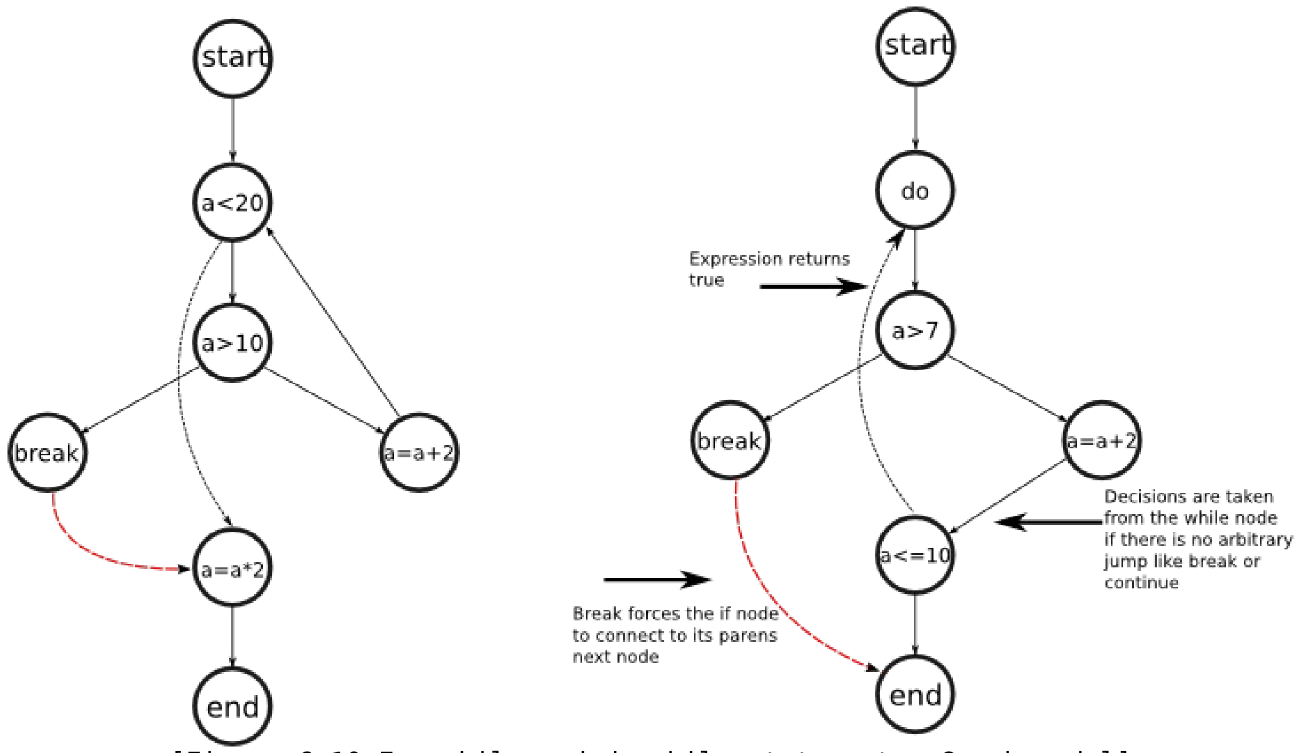
Do-while statement

Do statements are similar to the for and while statements with the exception that we have do nodes to represent this statement and the expression examination is done at the end of the block. This doesn't lead to a reaction change for its children. Basically the expression goes at the bottom of the block and it can be escaped from the bottom of the block without referencing the top. To make this more clear check the next picture.

Example:

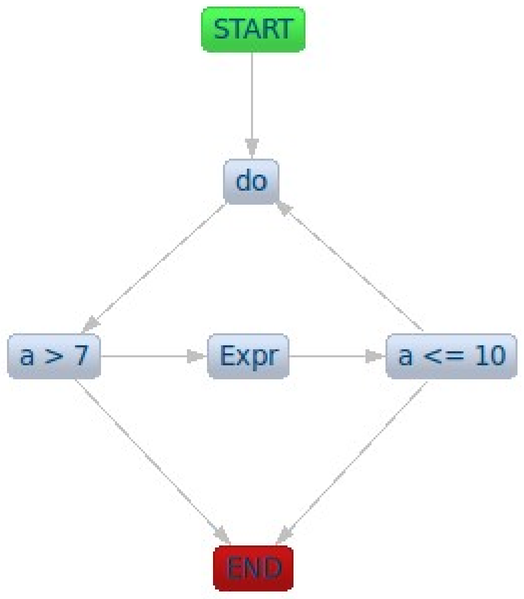
```
void doWhileTestMethod(int a) {
      do {
            if (a > 7) {
                  break;
            } else {
                  a = a + 2;
            }
      } while (a <= 10);
}
```

[Code 6.6 Do-while statement]


start
a<20
a>10
break
a=a+2
a=a*2
end
start
do
Expression returns true
a>7
break
a=a+2
Decisions are taken from the while node if there is no arbitrary jump like break or continue
a<=10
Break forces the if node to connect to its parens next node
end

[Figure 6.10 For-while and do-while statements. Graph model]

Ones again to make the differences clear we see the for while-statement at the left side of the picture and to the right is the do-while one. We can see at the top, that the do-node which is run ones no matter if the while expression returns false. This fact takes the control features of this node and puts it at the bottom which is the while-node. Inner nodes after having been executed connect to the while node and not to the do node at the top for escaping the block like in the for loops. Than after the examination the node decides whether to go back at the top of the block or escape the loop. Break nodes then have to reference to the next child of the while node as that is the first node to be executed after the block. Another picture will illustrate this case from the plug-in.



[Figure 6.11 Do-while statement]

Switch case statement

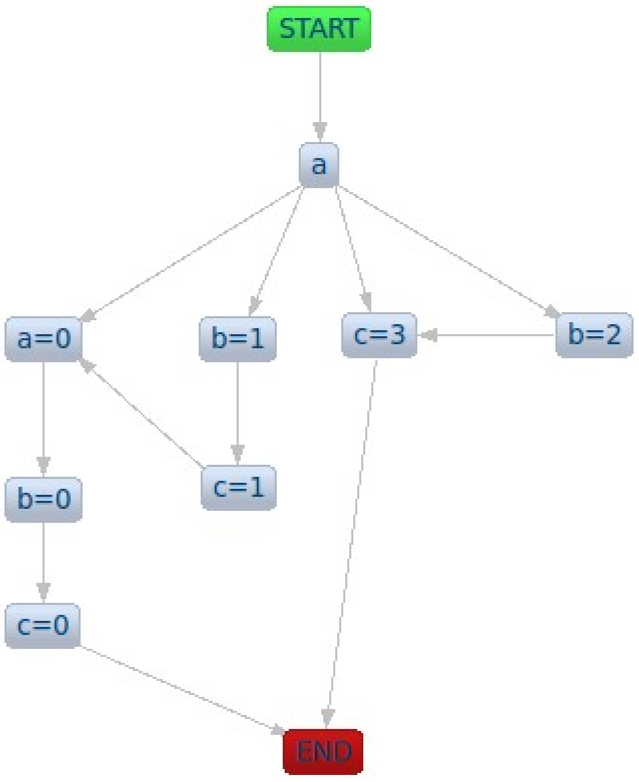
If the if-statement allows just one of two possible choices, with the switch-case statement it is possible to chose out of an undefined number of choices. The switch node has as much children +1 where the first one is the next node and the rest are the cases or the default case. As every branch represents a decision, every case should have its own branch. If one case ends with a break statement, than it connects strait to the end. But depending on the switch variable it can have previews cases leading to it or not. The following example explains it in details.

Example:

```
void switchTestMethod(int a, int b, int c) {
      switch (a) {
      case 1:
            b = 2;
      case 2:
            c = 3;
            break;
      case 3:
            b = 1;
            c = 1;
      default:
            a = 0;
            b = 0;
            c = 0;
      }
}
```

[Code 6.7 Switch case statement]

In case the variable `a` has an integer value of 1 than we know that case 2 is going to be executed as well. But if it is 2, than case 1 is not going to be executed. Therefore we do not want to discriminate any of the cases and assume they are executed directly because of the integer value so that indirect connections from previews cases are possible. A switch case graph of the code 6.7 is represented as in picture 6.12 from the plug-in.



[Figure 6.12 Switch case statement]

Case 1 and case 3, starting from the right, do not have a direct connection to the parents next node, in this case the end node, because they do not have a break statement. The last node, which in this case is the default node but as mentioned before there is no difference between default and case nodes, connects to the end node like any other case which concludes with a break.

Try-catch statement

Try-catch-finally like switch nodes have an undefined number of child nodes. It can have many catch block and/or a finally block. Each of the catch blocks is interpreted as a branch which is executed if an error occurred in the try block. The finally block is the one which gets executed no matter if all the statements in the try-block or catch-blocks were fully and successfully executed. Therefore this is good reason to treat the finally-block as the next node in the raw.
Try blocks are a bit more complicated concerning the unpredictable jumps that can occur during the execution and need to be examined in more details. A try-catch statement can almost be seen as a if-else statement, and sometimes incorrectly used as such instead of if-else statements even though Java literature discourages the usage of try-catch statements as normal control flow. Lets take a closer look at this statement and how it can be drown as a control flow graph. If the try-block is executed successfully than none of the catch block is entered and the next node in the raw is referenced. If it has a problem than only one catch block is executed which turns it practicaly into a if else similar form. A pseudo code interpretation of the try block can be represented as in code 4.8.

```
While [more statements to execute]
      If [there is a problem during execution]
      then
            goto most specialized catch block
      end if
If [there is finally]
```

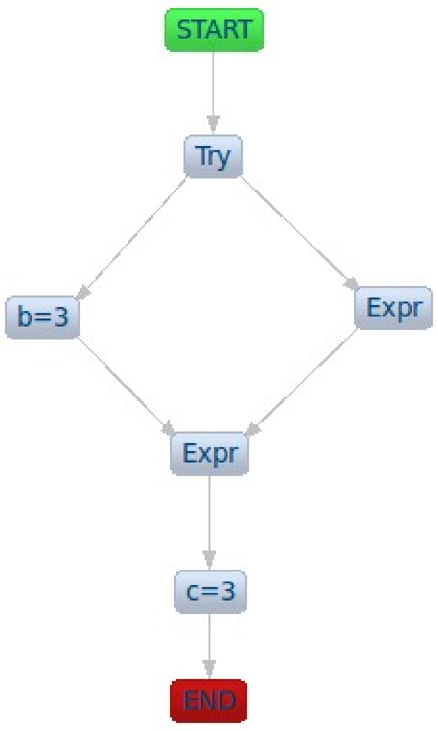
```
then
      goto finally block
else
      goto end of try-catch block
end if
```
[Code 6.8 Try-catch-finally pseudo code]

Unfortunately it can not be represented 100% as a if-else-statement because an exception can be thrown at any point of the try block (assuming that there are many statements in the try block that can throw different exceptions) and some exceptions are almost never thrown. It makes it really hard to know it in advance or predict which of the paths is going to be executed so that it would be inappropriate to connect every child node from the try block which can throw an exception with any possible catch block. A simplification of the problem is needed and the try-catch blocks are represented as separate branches.

Example:

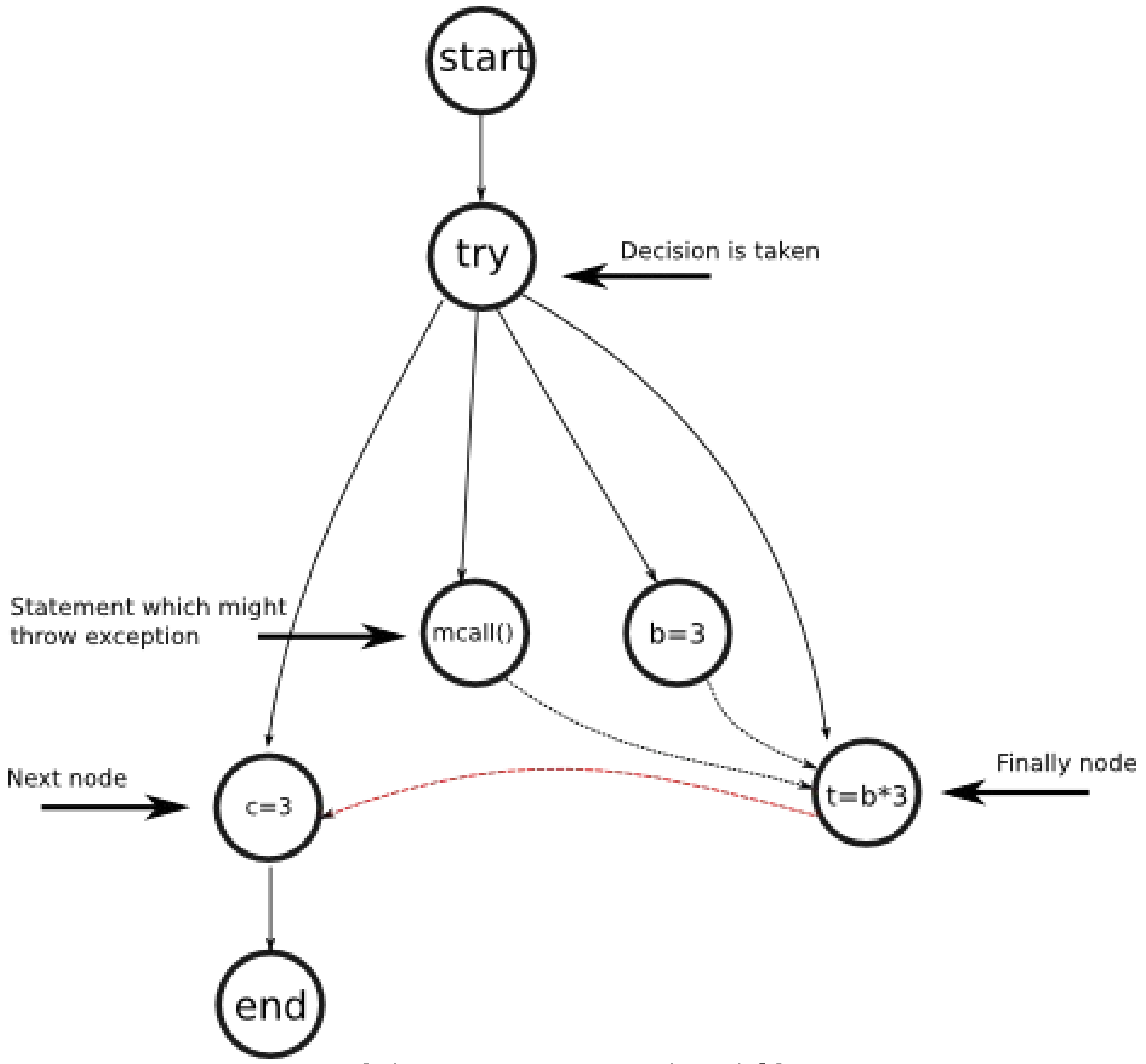
```
void tryCatchTestMethod(int b, int c, int t) {
      try {
            mightThrowAnException(b);
      } catch (Exception e) {
            b = 3;
      }finally {
            t = b*3;
      }
      c = 3;
}
```
[Code 6.9 Try-catch-finally example]

Code 6.9 as other code examples do not aim to build meaningful algorithms but to show how special cases of the current topics are represented in flow chart diagrams.



[Figure 6.13 Try-catch example]

From picture 6.13 we can see that the decision is taken from the try node which is supposed to know in advanced if the try block is going to be executed without exceptions. Again this is not 100% right but as there is no possibility to predict, this is the best possible way of representation. The model looks like in the following picture.



[Figure 6.14 Try-catch model]

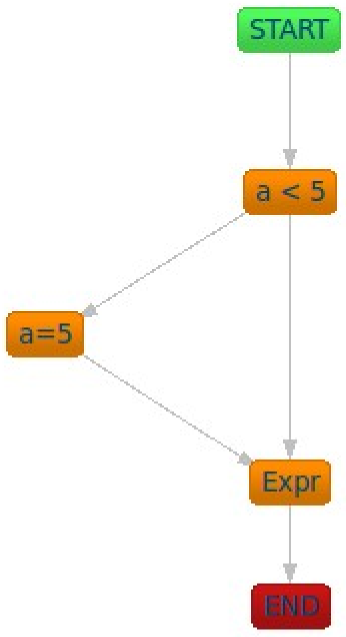
The dotted lines represent the new connections added by the view algorithm. After the block is run successfully or an exception was handled the last statements are run from the finally block and the program continues with the next node.

7 CodeCover

When someone builds control flow graphs, except for the times it is meant just for visualization, it is done with the intention of knowing the code coverage strategy can be applied. By wanting to make the plug-in as convenient as possible it was worthy trying to integrate this feature as well. After searching and testing many suitable code coverage applications, CodeCover was the one chosen for this plug-in. It was developed from the University of Stuttgart and is open source as well. It offered the best possible interface to cooperate but not depend on the application. It also has a plug-in for Eclipse and can be installed from the web site.

It is able to cover branches, conditions, loops and statements. To make the plug-ins independent from each other a simple way of interaction was chosen. After CodeCover finishes collecting the data it saves in the `codecover` folder within the project a xml file containing all the necessary information needed from the CFG. They are processed and the identified nodes then are marked as covered in the editor. The way to do it is by reading the offsets of every covered element from the cover session and compare them with the actual nodes. As the offsets must be 100% right for a node to be marked as covered we can be sure of that. A second menu item is added to the flow chart generator menu indicating that the generation of a graph can be made from a specific `codecover session`. The third menu item just takes the last session and compares it with the graph. If no information about the selected method is found in the session then a error massage is shown indicating the error and the graph is not built.

In the following diagram we can see a graph from the if-statement which is first examined by CodeCover and than build with the cover information it offers. For this example was activated condition and statement coverage.

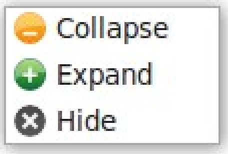


[Figure 7.1 if diagram with condition and statement coverage]

8 Node folding

Methods can get very big and no programmer likes scrolling up and down, so that a node folding feature would be very helpful in this situation. It can be activated by right clicking on a node which can be folded. This are the nodes that have children. In the menu there are three options

- collapse
- expand
- hide



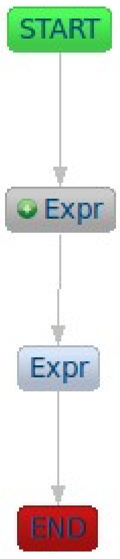
[Figure 8.1: Node folding menu]

Collapsing the node means that every child or descendant node which stays between it and its next node will be hidden.

Expanding makes the opposite and shows the hidden part of that node.

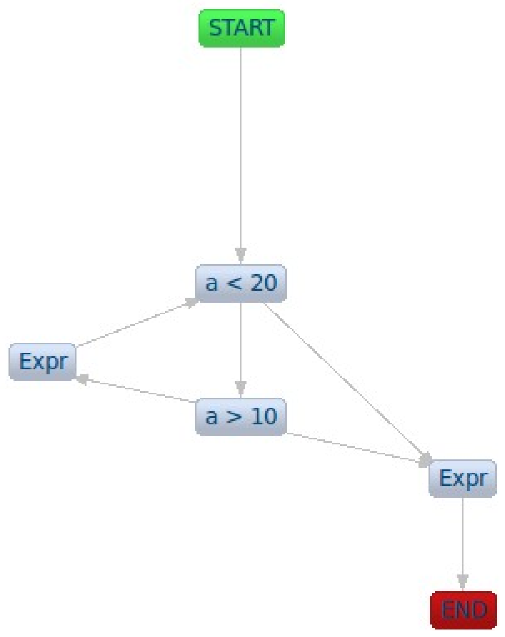
Hiding is the only irreversible function in the menu. After this function is executed successfully on a node, the node itself and every incoming or outgoing connection is disposed. The reason to offer this functionality is to cut the graph for example.

Following pictures will show how it looks like in the plug-in.



[Figure 8.2: Node folding example]

The first node after the start-node is the one which has been collapsed. It changes color into gray and a little plus icon is added to the node indicating that it is expandable. If it is expanded then it returns to its previews shape and no differences can be seen, meaning no minus icon is added.
This graph was build from the following source graph.



[Figure 8.3: Node folding source graph]

The collapsed node represents a for statement and all its descendants were hidden. It was built from the following code section.

```
void forTestMethod(int a) {
      for (a = 1; a < 20;) {
            if (a > 10) {
                  break;
            } else {
                  a = a + 2;
            }
      }
      a = a * 2;
}
```

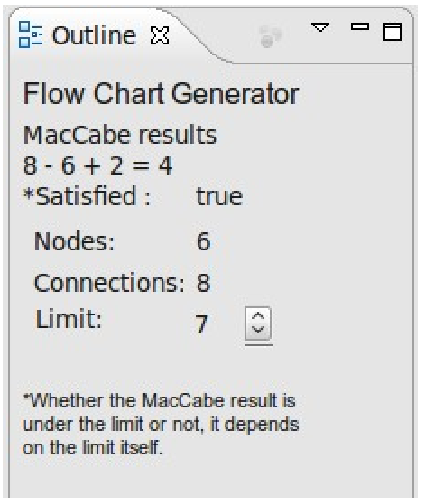
[Code 8.1 For node collapsing example]

9 Outline View

The outline view was implemented to show static information about the graph like the number of the nodes and the connections. It is shown automatically with the appearance of the editor so that no extra activation is needed.

MacCabe metric

The measurement of cyclomatic complexity by McCabe was designed to indicate a program's testability and understandability (maintainability). It is the classical graph theory cyclomatic number, indicating the number of regions in a graph. As applied to software, it is the number of linearly independent paths that comprise the program. As such it can be used to indicate the effort required to test a program. To determine the paths, the program procedure is represented as a strongly connected graph with unique entry and exit points.[AWMeMo02] It is computed using a control flow graph like those build by the plug-in and was a good possibility to extend the amount of information which can be gathered from the control flow graph. Beside the number of nodes and connections in the outline view there is information about the cyclomatic complexity as well. From the nodes and connections we build a formula which gives information about this metric. The formula is simplified because we are examining just one component as an entity and its last version is connections-nodes+2. The result is compared to a limit which can vary in some cases and software phases. In the literature we can find strict examinations starting from a limit of 7 and those more flexible defining it up to 15. For the plug-in the limit is flexible and can be set by the user. The next figure shows the outline view from an example.



Flow Chart Generator

MacCabe results

8 - 6 + 2 = 4

*Satisfied :	true
Nodes:	6
Connections:	8
Limit:	7

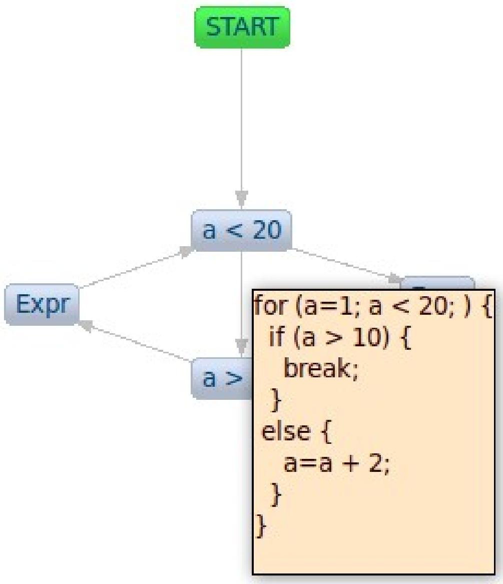
*Whether the MacCabe result is under the limit or not, it depends on the limit itself.

[Figure 8.1 Outline view]

10 Other features

Mouse over

If the node text is longer then a static value of text than the code is hidden and replaced with the “Expr” text. This was introduced for the case very long variable names would interact with the graph positioning. If the mouse over event is fired, the node will show its source code. It embraces the entire block with its descendants and is not just for very long expression statements useful but for other statements as well. An example is shown below.



```
for (a=1; a < 20; ) {
  if (a > 10) {
    break;
  }
 else {
    a=a + 2;
  }
}
```

[Figure 10.1: Mouse over event]

11 Installation and Usage

There are 2 ways to install the plug-in.

- from the website
- manually downloading

From the official website http://eclipsefcg.sourceforge.net/ you can find both possibilities.

The automatic/update method is recommended because it checks for dependencies and compatibility. First step is adding the site to the eclipse update site repository from the Eclipse menu:

Help > Software Updates... > Available Software (tab) > Add Site …

http://eclipsefcg.sourceforge.net/

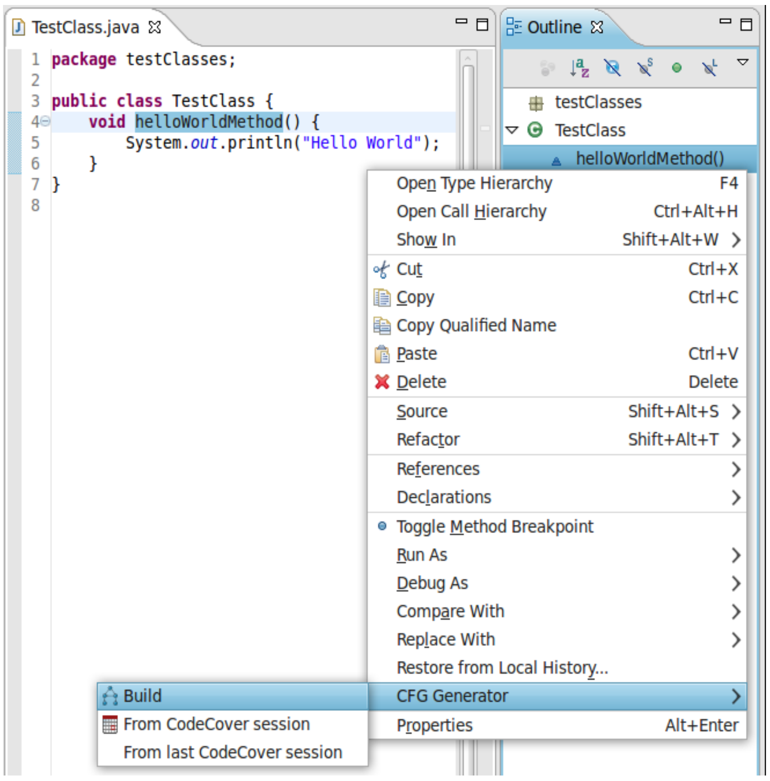
After calculating the dependencies and confirming the add-ons, a restart is recommended and the plug-in is ready to run.

Manual download

If you decide to download it manually and install it from the zipped package than you should check the dependencies listed in this documentation first. After you are sure everything fits then copy the content of the package into your eclipse folder. After a restart it should be ready to use.

How to find it

The Generation is applicable for every Java method and constructors no meter to the access modifiers. From the outline view of the default Eclipse Java Editor you can right-click and then follow the menu item **CFG Generator** like in the next picture.



```
package testClasses;

public class TestClass {
    void helloWorldMethod() {
        System.out.println("Hello World");
    }
}
```

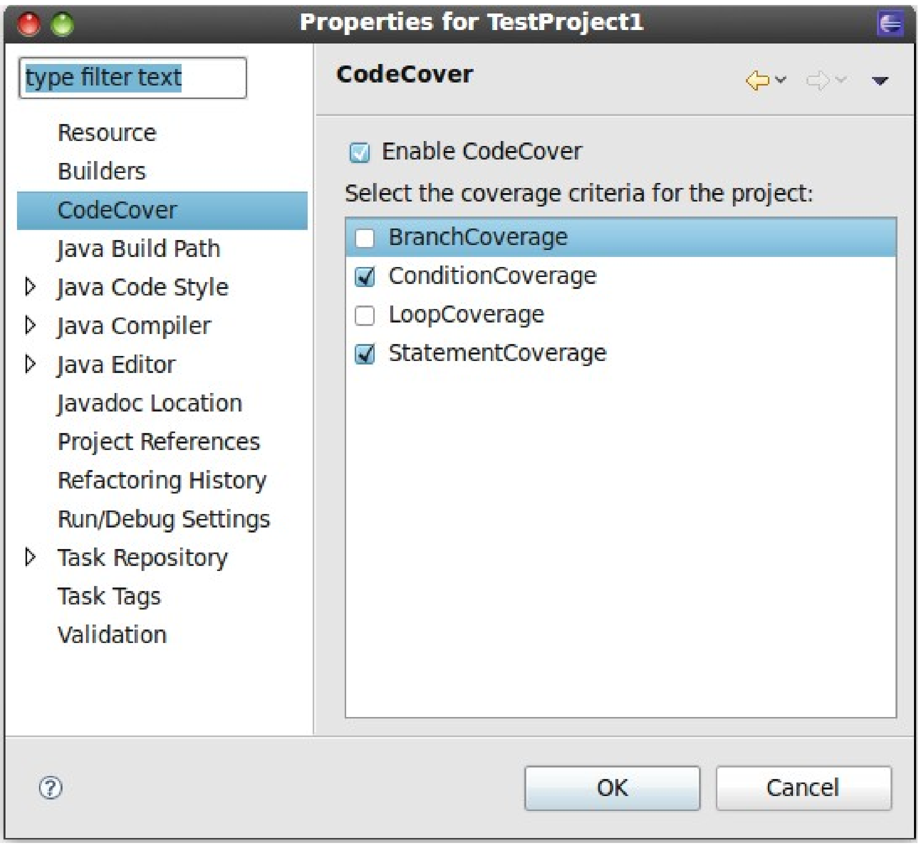
[Figure 11.1: Menu]

CodeCover tool

Detailed information about this tool you can find at the website http://www.codecover.org/ Although a quick tutorial might be handy for this user guide. After the installation, to set up the project for code coverage follow the steps below.

Step 1

On the project properties go to the menu CodeCover and enable it for the branch and statement coverage. You can activate them all but for this CFG Plug-in version, the loop and branch coverage are not available.



CodeCover

✓ Enable CodeCover

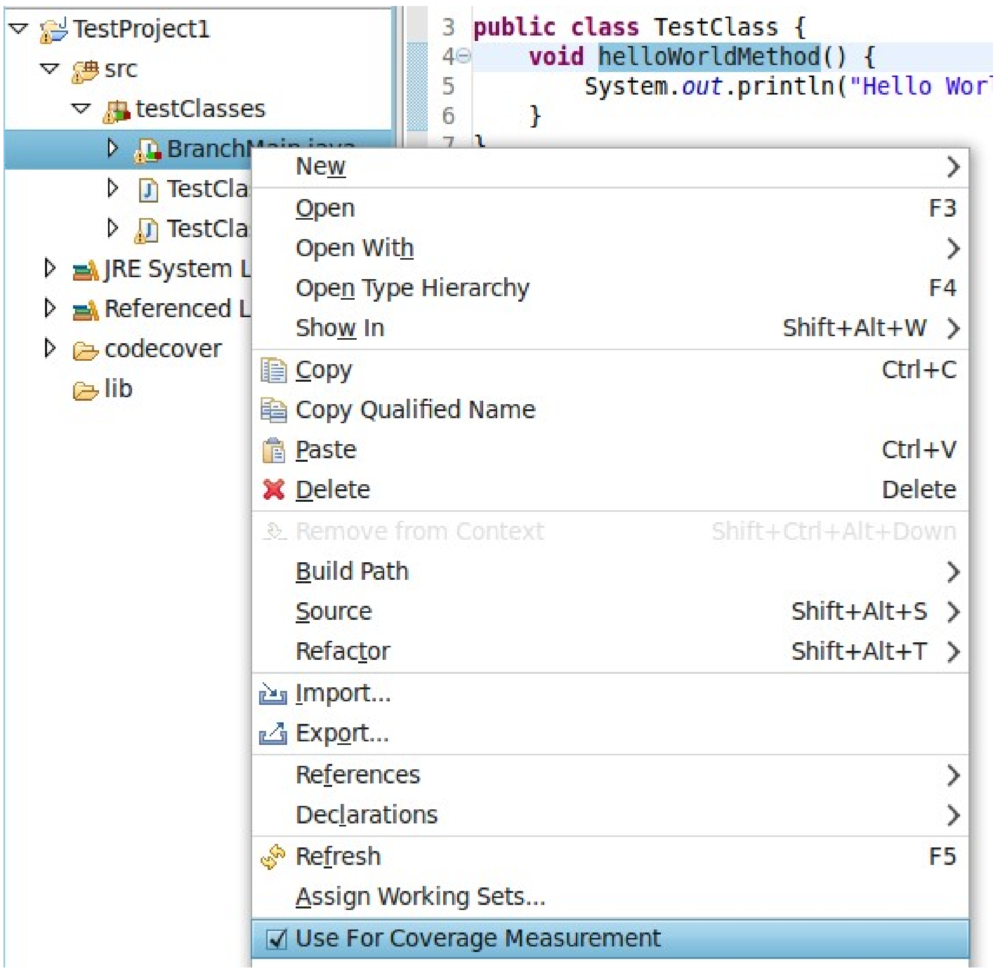
Select the coverage criteria for the project:

- ☐ BranchCoverage
- ✓ ConditionCoverage
- ☐ LoopCoverage
- ✓ StatementCoverage

[Figure 11.2: Project properties]

Step 2

You need to enable for every class you wish to have a log, the menu item **Use For Coverage Mesurement.** Check the picture below.

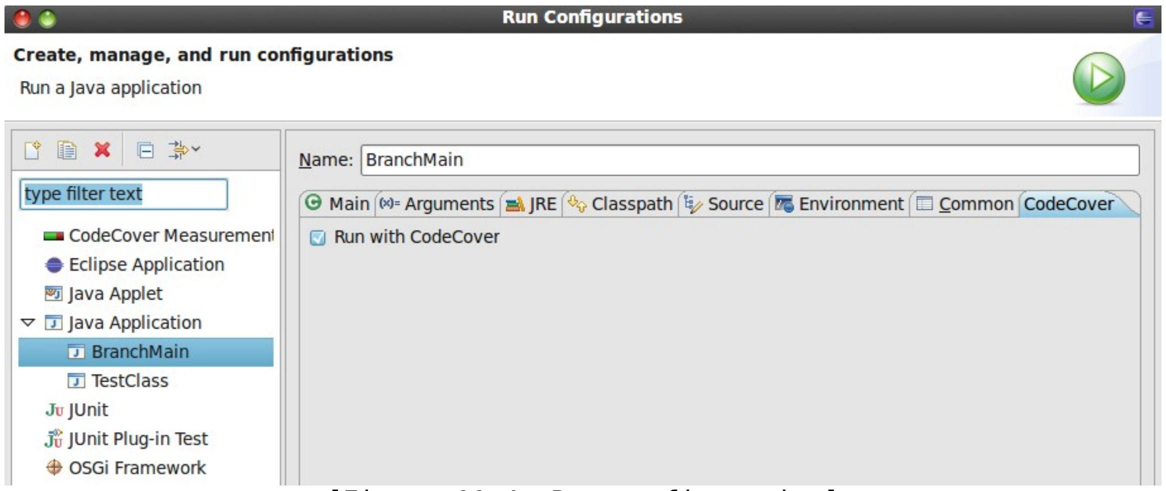


```
public class TestClass {
    void helloWorldMethod() {
        System.out.println("Hello Wor
    }
}
```

[Figure 11.3: Class coverage]

Step 3

And the last step is to set up the run configuration so that it runs with CodeCover like shown below.



Create, manage, and run configurations

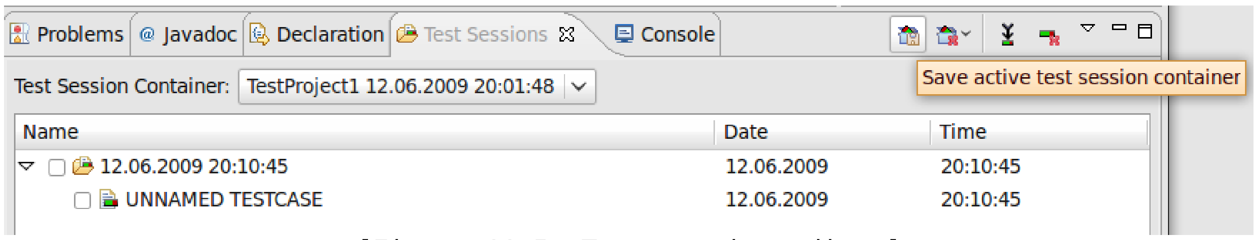
Run a Java application

Name: BranchMain

[✓] Run with CodeCover

[Figure 11.4: Run configuration]

Then the project or program simply needs to be executed. After the program termination the session log is generated but not saved. We have to save it first and just then it is available for the CFG generator.



Test Session Container: TestProject1 12.06.2009 20:01:48

Name	Date	Time
12.06.2009 20:10:45	12.06.2009	20:10:45
UNNAMED TESTCASE	12.06.2009	20:10:45

[Figure 11.5: Test session editor]

CodeCover hast many features like visualization effects and can be used for JUnit test cases as well. For more information about CodeCover please check the website.

12 Similar tools

c1visualizer is a visualization tool for the Java HotSpot client compiler. As we mentioned at the beginning of this document there is a second way to build control flow graphs, which was treated by a project at university of Linz. It was created as a visualization and simplification tool for developers who need to analyze the data structures manipulation during the compilation.

Control Flow Graph Factory is a proprietary licensed plug-in for building CFG. It can perform this operations using both strategies from the Bytecode execution and from source code examination.

13 Sources and References

CFG Generator	[http://eclipsefcg.sourceforge.net/]
Eclipse Application	[http://www.eclipse.org/]
Eclipse SDK API	[http://help.eclipse.org/]
Eclipse GEF	[http://www.eclipse.org/gef/]
GEF Tutorial	[http://www.ibm.com/developerworks/library/os-eclipse-gef11/]
Eclipse Development	[http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf]
CodeCover	[http://www.codecover.org/]
c1visualizer	[https://c1visualizer.dev.java.net/]
Control Flow Graph Factory	[http://www.drgarbage.com/]
[linkCTools]	[http://java-source.net/open-source/code-coverage]
[OREclipse]	O'Reilly Eclipse, Holzner, 2004, Ch 1.3
[AWMeMo02]	Metrics and Models in Software Quality Engineering, Second Edition Kan, 2002, Ch 11.3

Additional

Basiswissen Softwaretest (german) [Spillner, Linz, 2004]

Code Quality Management (german) [Simon, Seng, Mohaupt, 2006]