Software testing

JaBUTi

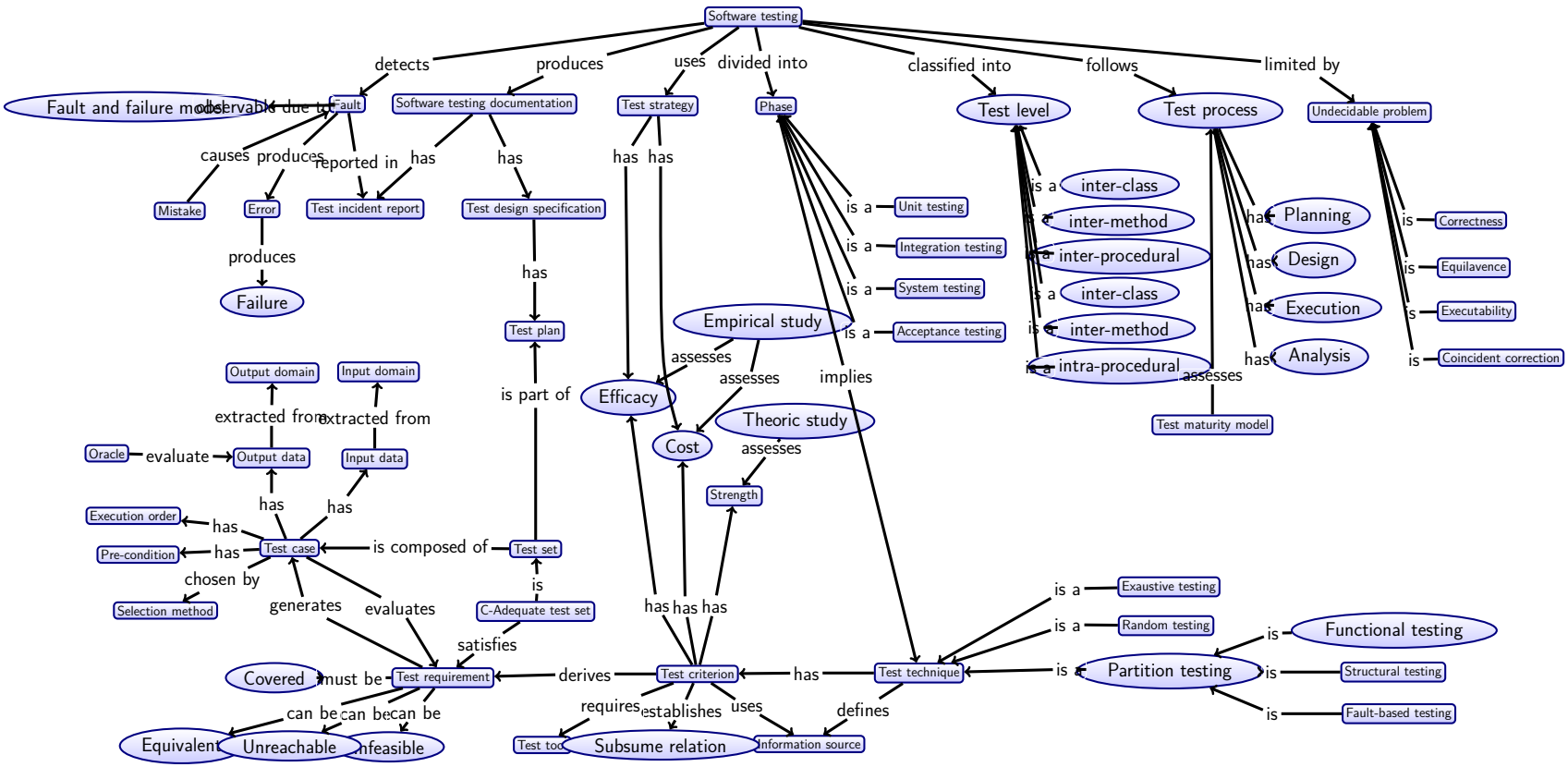
Marco Aurélio Graciotto Silva[1], Ellen Francine Barbosa[1],
Márcio Eduardo Delamaro[1], Auri Marcelo Rizzo Vincenzi[2],
José Carlos Maldonado[1]

[1]University of São Paulo (USP)
São Carlos, SP, Brazil

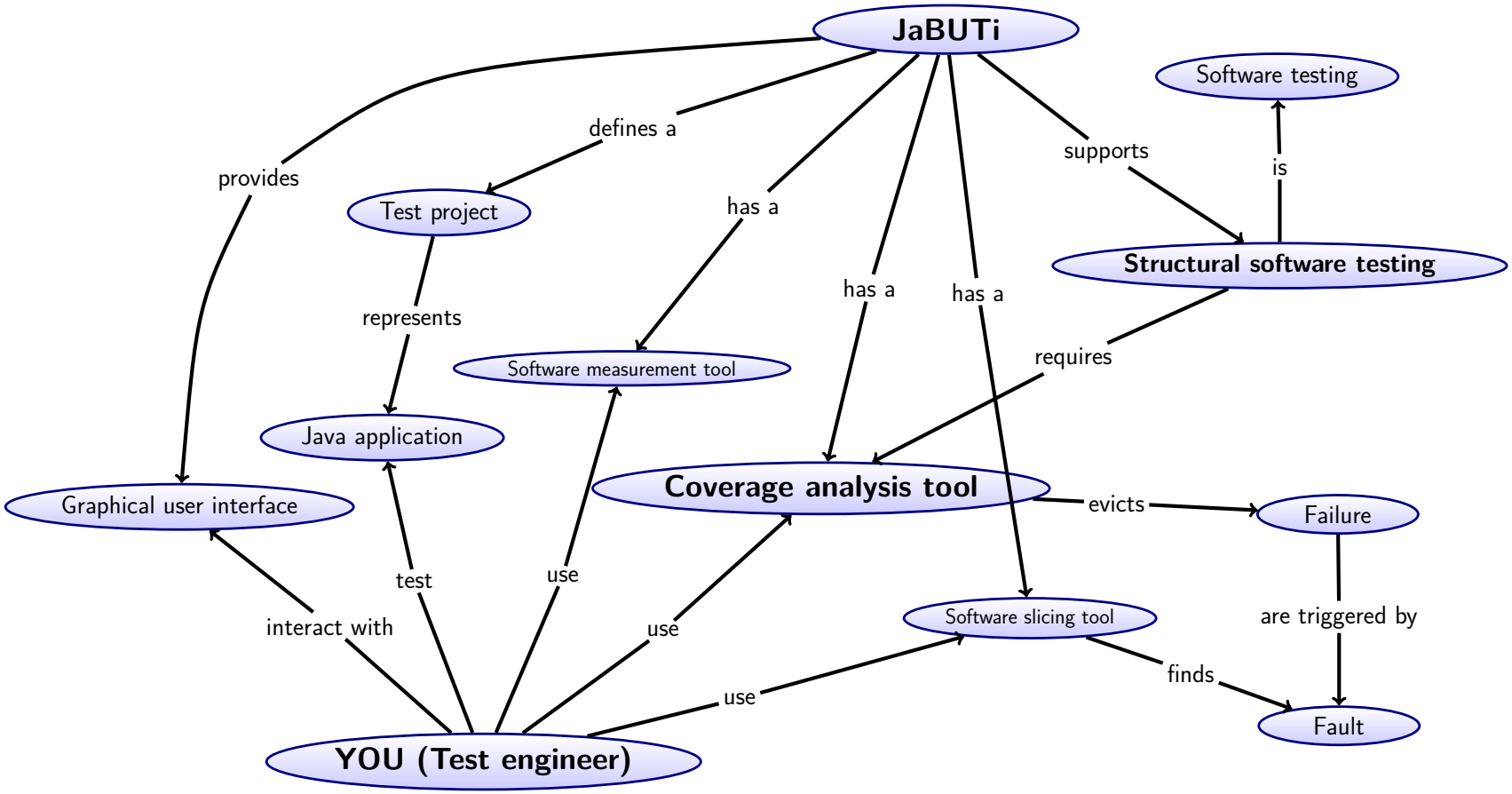
[2]Federal University of Goiás (UFG)
Goiânia, GO, Brazil

February 2011

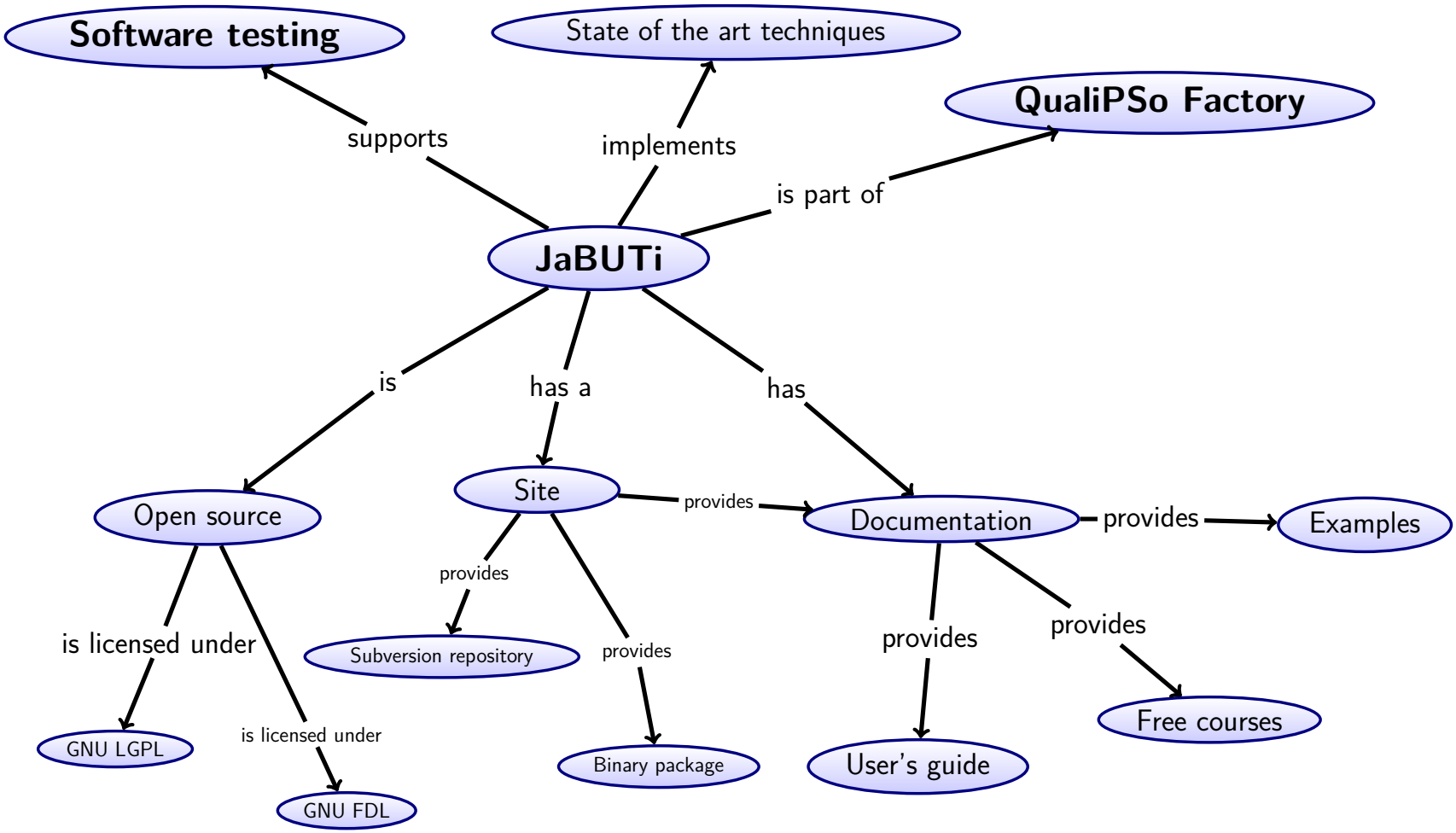
Software testing



JaBUTi and software testing



JaBUTi


Software testing
State of the art techniques
QualiPSo Factory
supports
implements
is part of
JaBUTi
is
has a
has
Open source
Site
provides
Documentation
provides
Examples
provides
Subversion repository
provides
provides
provides
is licensed under
Free courses
GNU LGPL
is licensed under
Binary package
User's guide
GNU FDL

About

JaBUTi is an **open source** software **testing tool** that implements state of the art **techniques** for **control-flow** and **data-flow** based testing of Java applications.

Users

It is currently used by industry and in research settings:

- QualiPSo Project,
- FLOSS Competence Centers,
- Software Engineering Laboratory (ICMC/USP).

Web site

- JaBUTi is hosted at CCSL/ICMC-USP.
  - Source code hosted in the Subversion repository.
  - Packaged binary, ready to run.
  - Documentation

Documentation

- JaBUTi User's Guide.
- This course home-page (slides, related material, and exercises).
- Applications used as examples (source code, test sets and binary downloads).

General terms

It is free to modify and share JaBUTi (if kept the same license and acknowledge the original authors).

Licenses

- Software:
  - GNU Lesser General Public License 2.1
- Documentation:
  - GNU Free Documentation License Version 1.3.

JaBUTi components

JaBUTi architecture is comprised of several components:

- **program representation**, **visualization**, and **instrumentation**,
- **test case execution** and **management**,
- **software measurement**,
- **software slicing**, and
- and **report** components.

JaBUTi tools

Those components are organized in three tools:

- software **coverage analysis tool**,
- software **measurement tool**,
- software **slicing tool**.

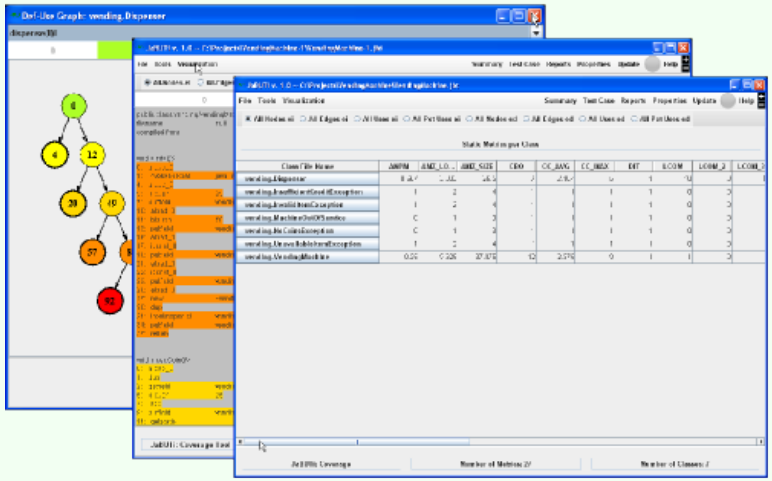
Why so many tools?

- The coverage analysis tool comprises of most of JaBUTi's features: test case management, execution, instrumentation, visualization and report.

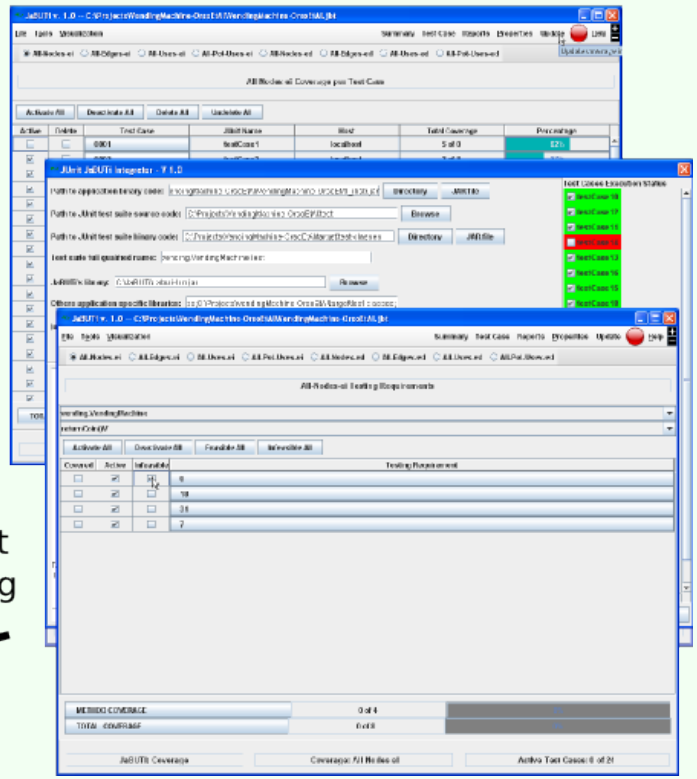
Why so many tools?

- However, some guidance must be provided to define a good test strategy.
  - The software measurement tool collects information about the complexity of the software under testing, which is a good hint of which features should be covered first.
- Once detected a **failure**, it must be traced to the **fault** that triggered it.
  - The slicing tool uses the execution trace of test cases to detect the best spots to look for faults.

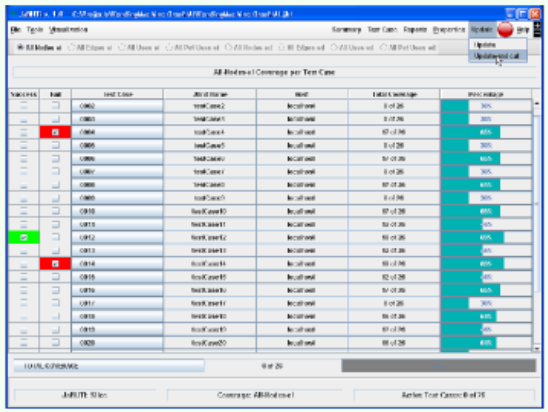
JaBUTi tools relationship



Graph and code visualization, static metrics, and test requirement weights are used to design test cases and to improve coverage metrics.

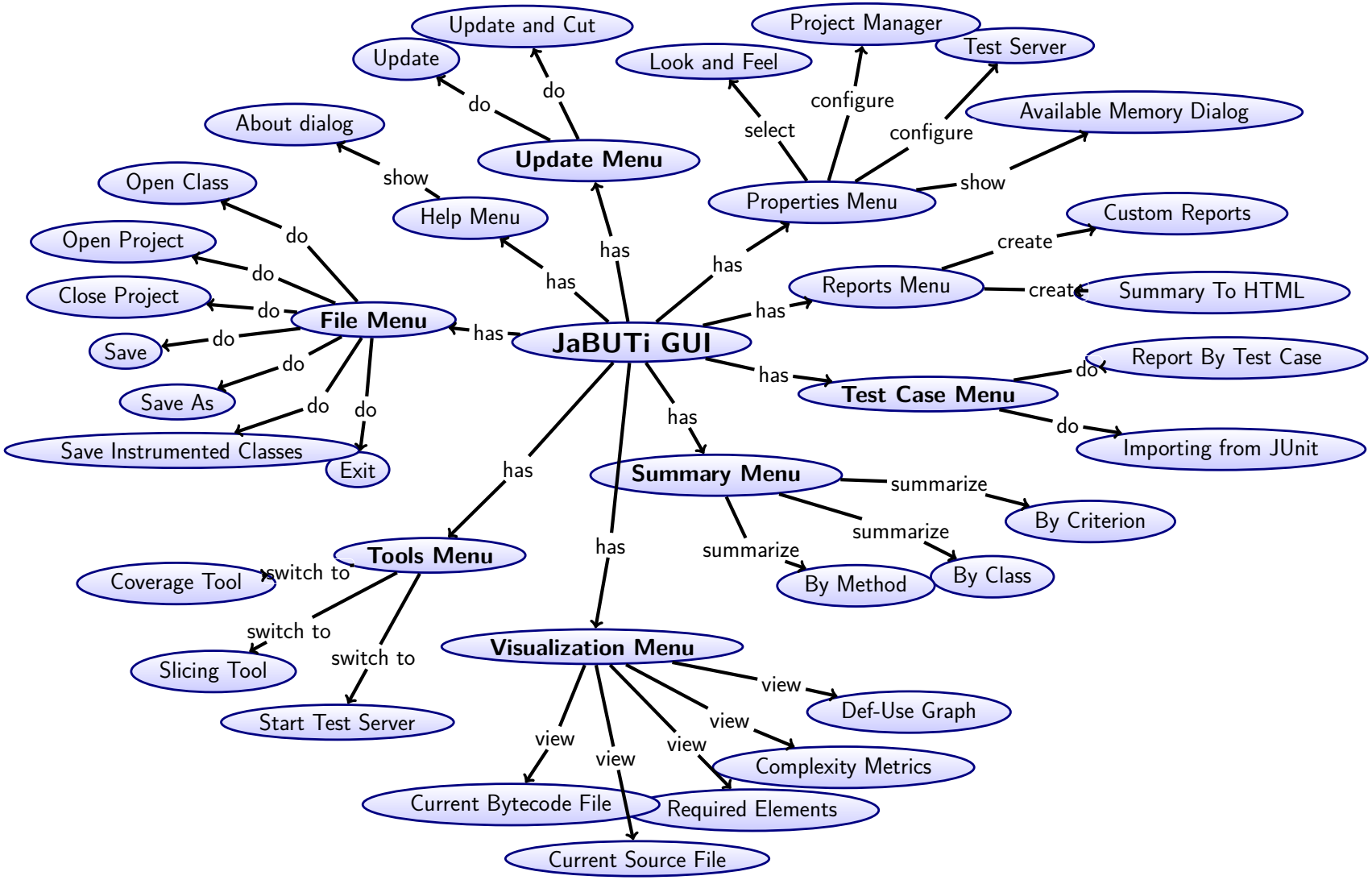


Once identified the fault, the software testing activities can be restarted.



If a failure is detected, the fault can be found using the slicing tool

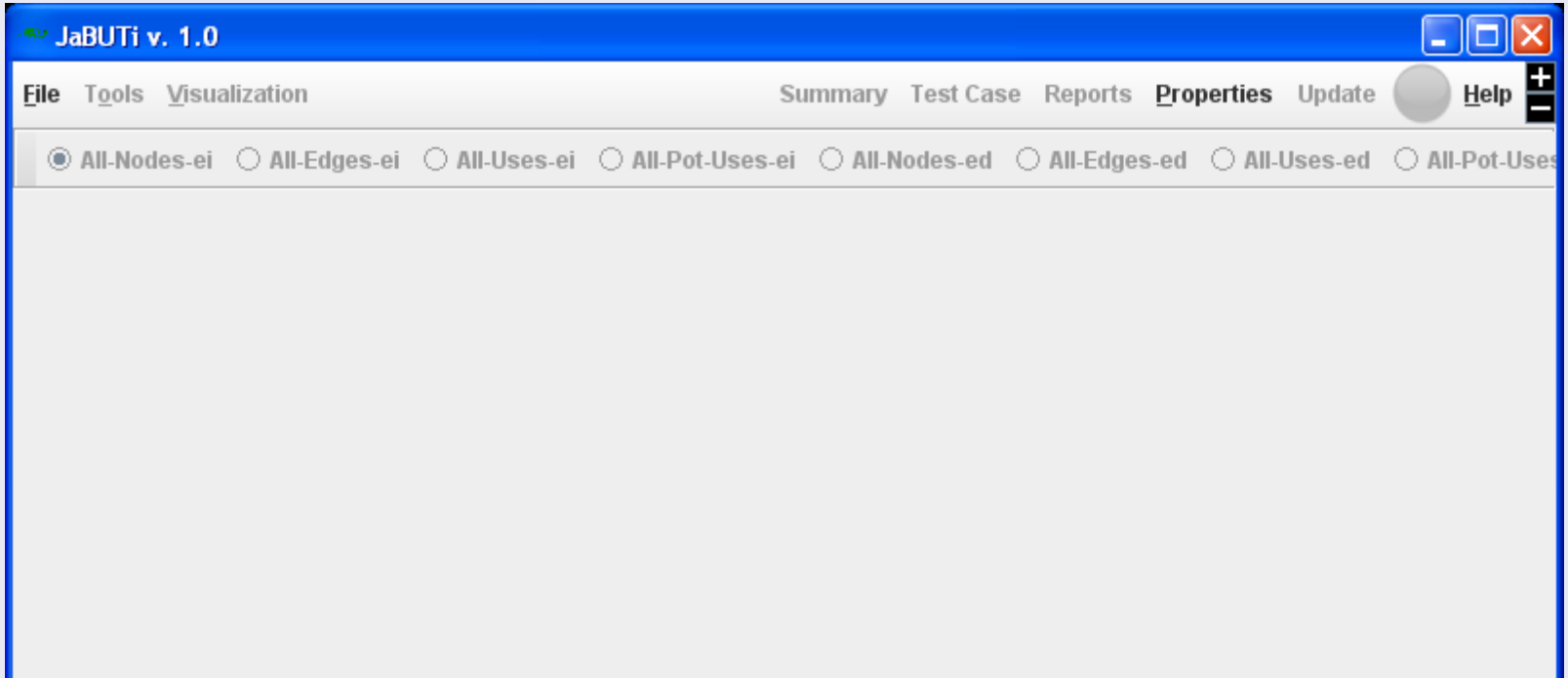
Concepts

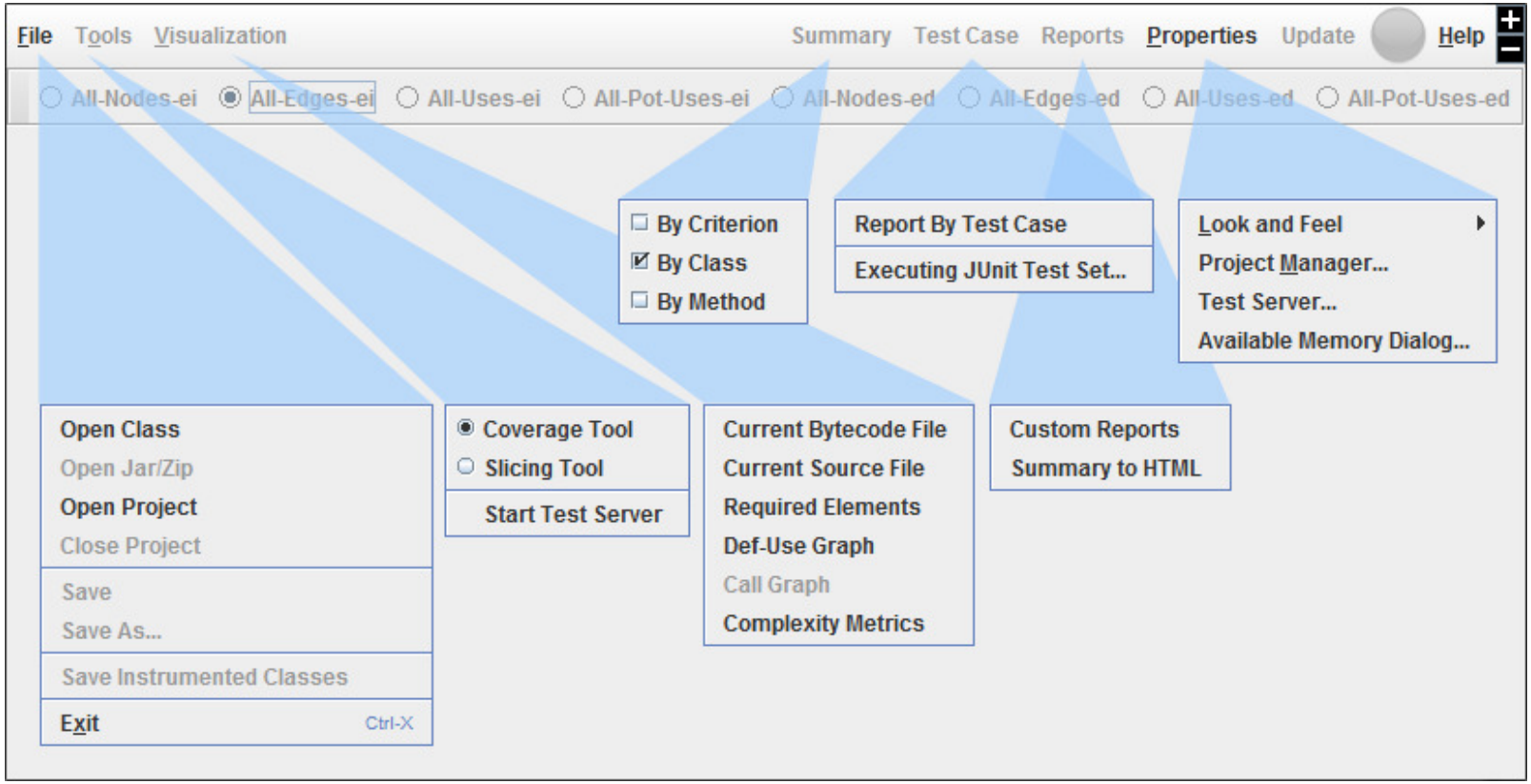

JaBUTi GUI
has
File Menu
Open Class
Open Project
Close Project
Save
Save As
Save Instrumented Classes
Exit
do
Update Menu
Update
Update and Cut
Help Menu
About dialog
show
Properties Menu
Look and Feel
Project Manager
Test Server
Available Memory Dialog
select
configure
Reports Menu
Custom Reports
Summary To HTML
create
Test Case Menu
Report By Test Case
Importing from JUnit
Summary Menu
By Criterion
By Method
By Class
summarize
Tools Menu
Coverage Tool
Slicing Tool
Start Test Server
switch to
Visualization Menu
Def-Use Graph
Complexity Metrics
Current Bytecode File
Required Elements
Current Source File
view

JaBUTi GUI

- Allows the beginner to explore and learn the concepts of control-flow and data-flow testing.
- Provides a better way to visualize which part of the classes under testing are covered and which are not.

Demo



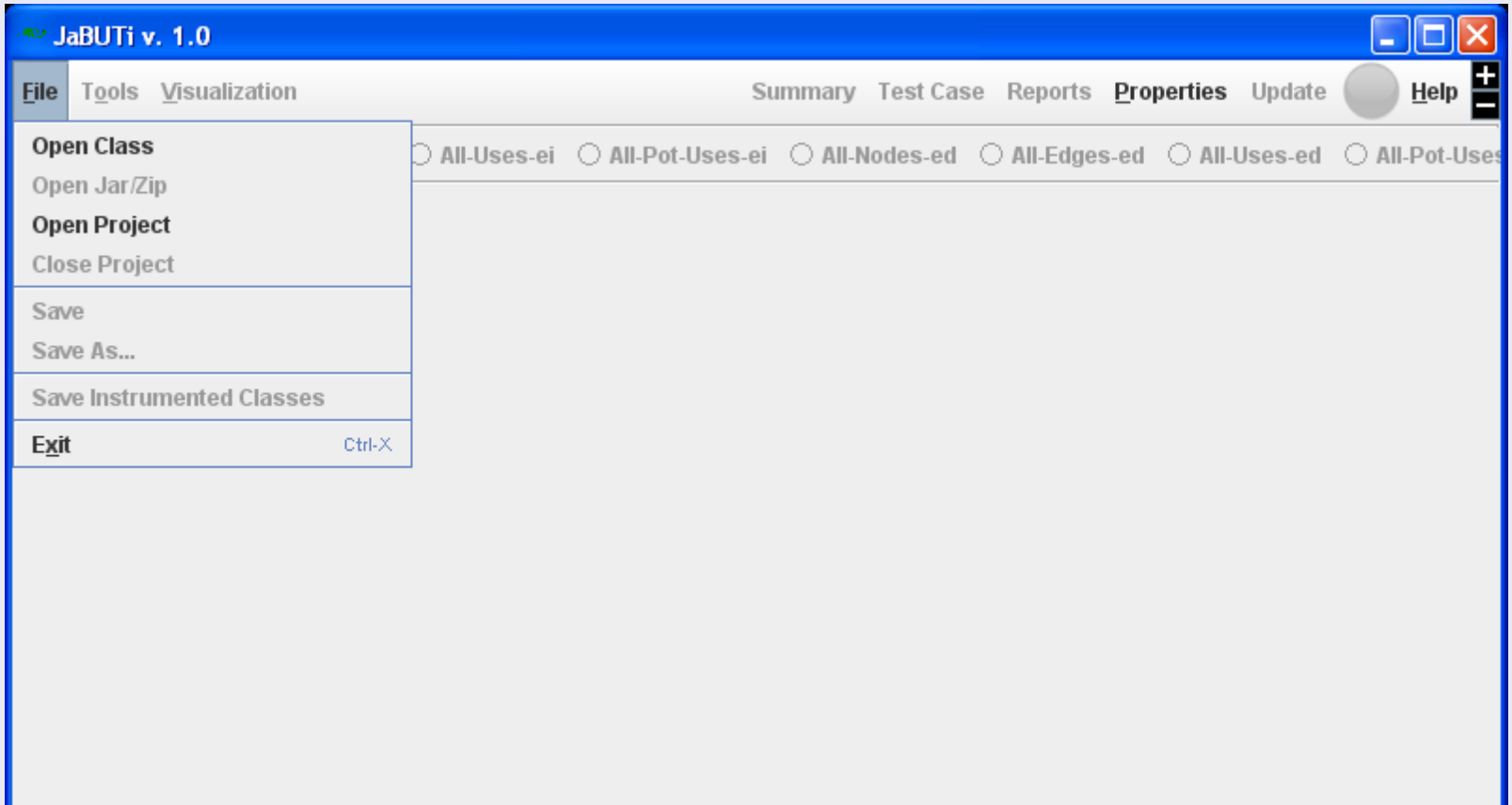


Demonstration of JaBUTi capatibilities

File

The **File** menu provides options to create and manipulate a JaBUTi project.

Demo



Open Class

The **Open Class** menu option allows to select the base class file from where the classes to be tested are identified.

Demo

Open Project

The **Open Project** option opens a previously created project.

Demo

Close Project

The **Close Project** option closes the current project.

Demo

Save

The **Save** option saves the current project.

Demo

Save As

The **Save As** option saves the current project with a different name.

Demo

Save Instrumented Classes

The **Save Instrumented Classes** option saves the classes from the current project, already instrumented for testing in production or a different context.

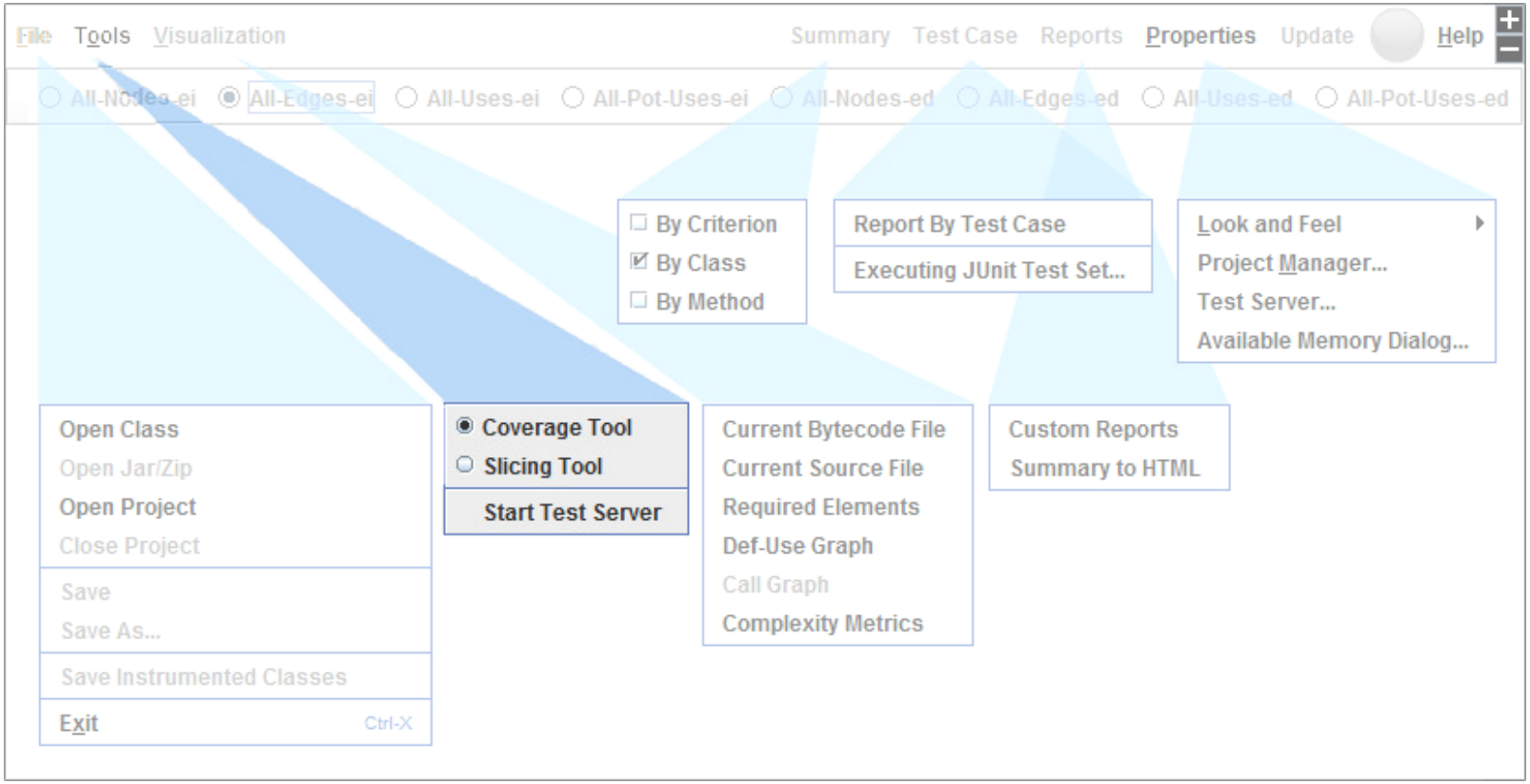
Demo

Exit

The **Exit** option exits of the tool.

Demo

Graphical user interface
Main functionalities - Tools



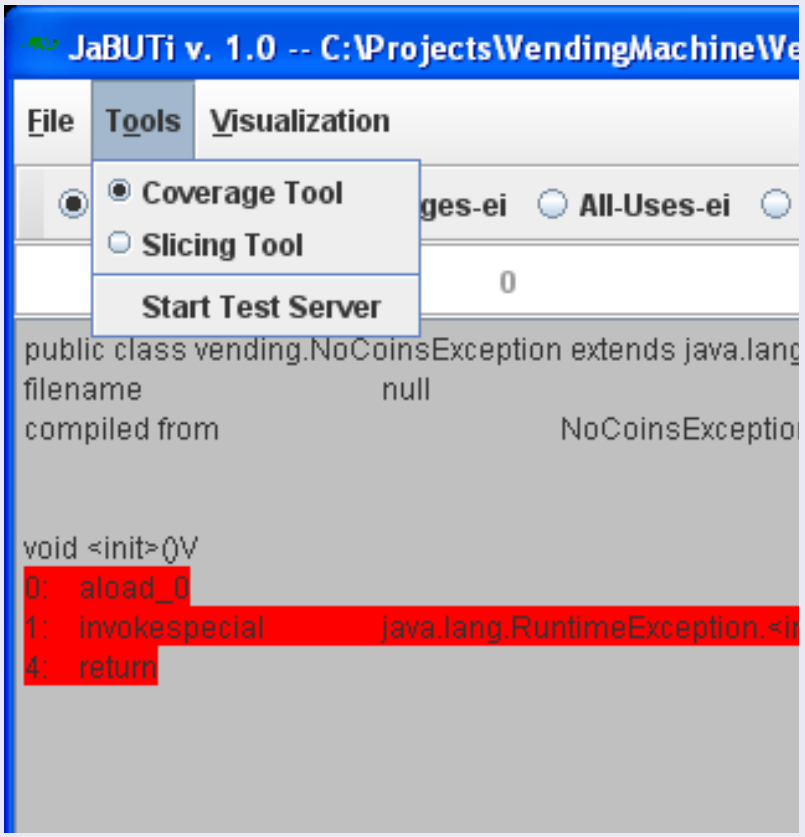
Tools

The **Tools** menu provides access to JaBUTi's tools.

Tools

- **Coverage Tool**: enables JaBUTi's coverage tool.
- **Slicing Tool**: enables JaBUTi's slicing tool.
- **Start Test Server**: starts the test server for mobile devices.

Demo



Visualization

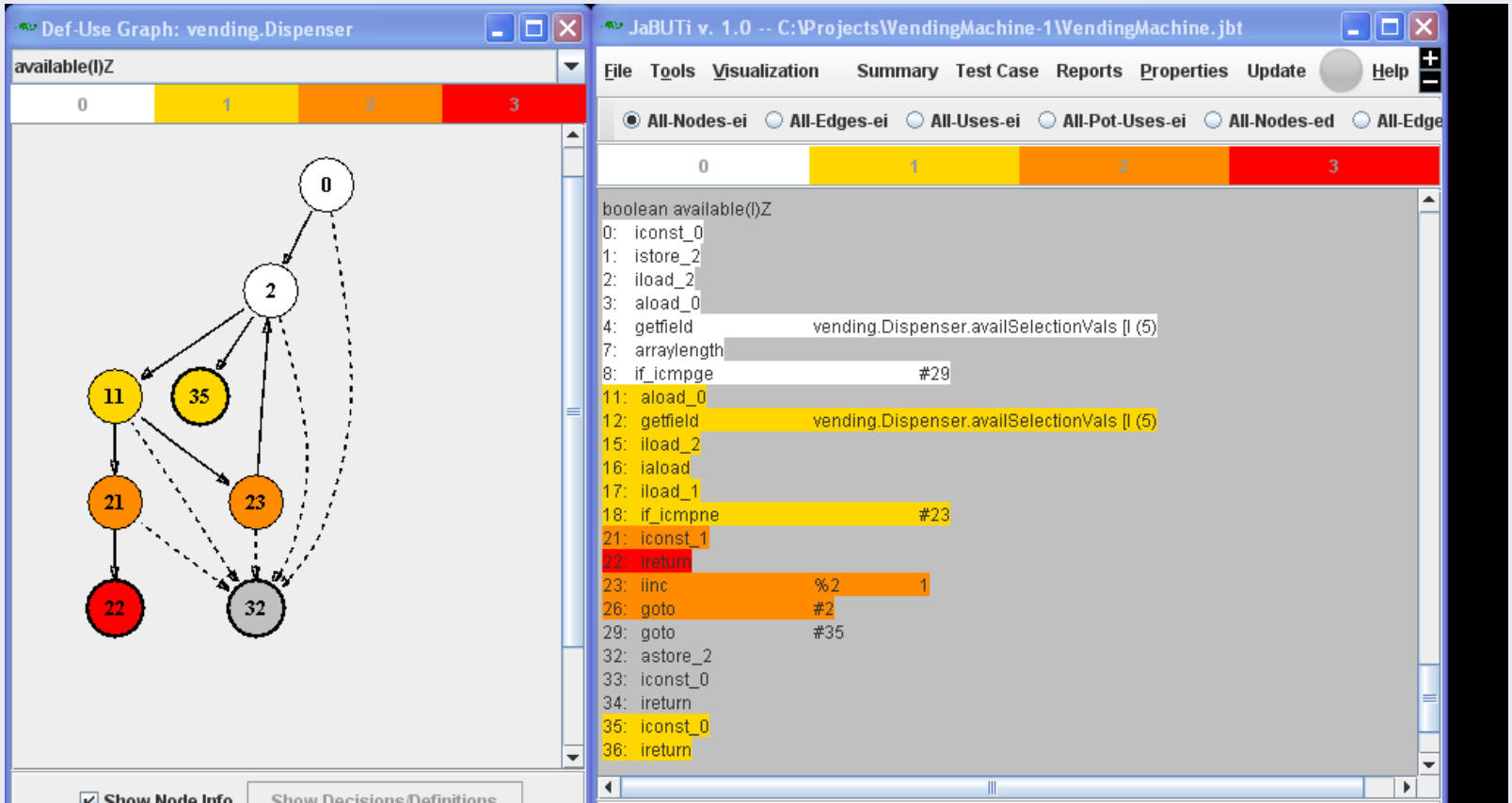
The **Visualization** menu provides different forms of visualization of the classes and methods under testing.

Demo

Current Bytecode File

Current Bytecode File option shows the highlighted bytecode of the currently selected class file.

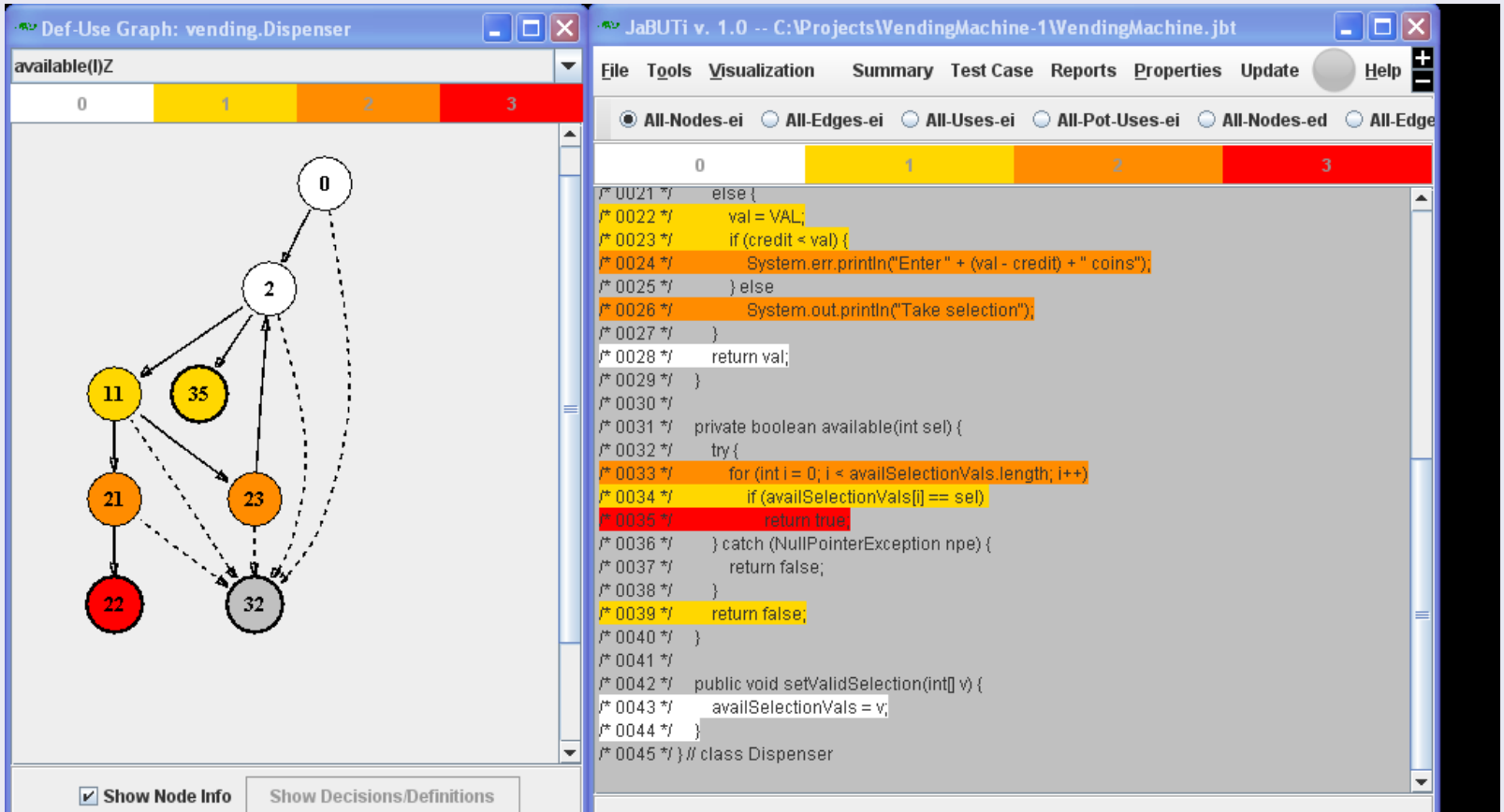
Demo



Current Source File

Current Source File option shows the highlighted source code of the current selected class file.

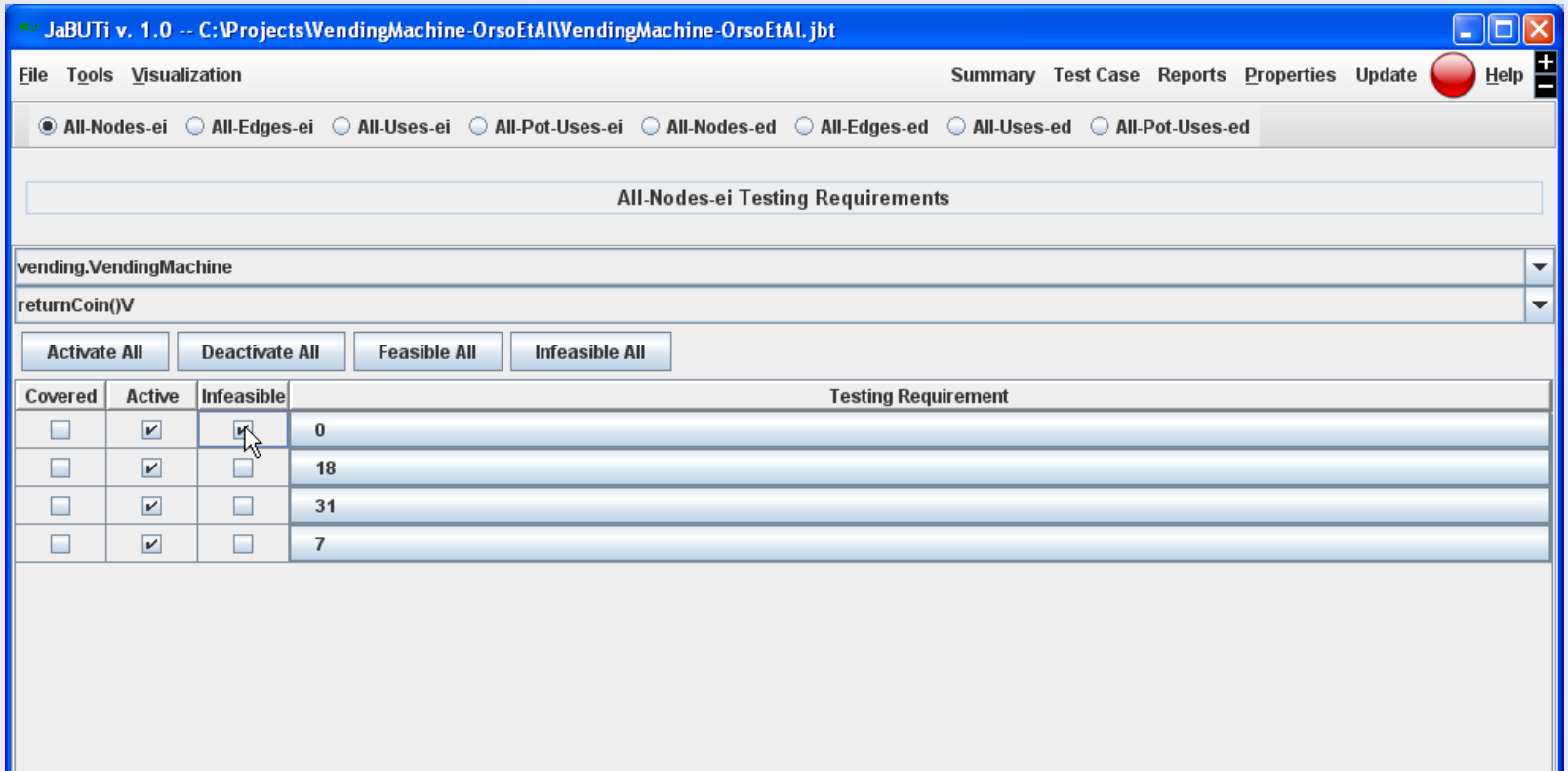
Demo



Required Elements

Required Elements option shows the set of required elements for a given method of a given class, considering the current selected criterion.

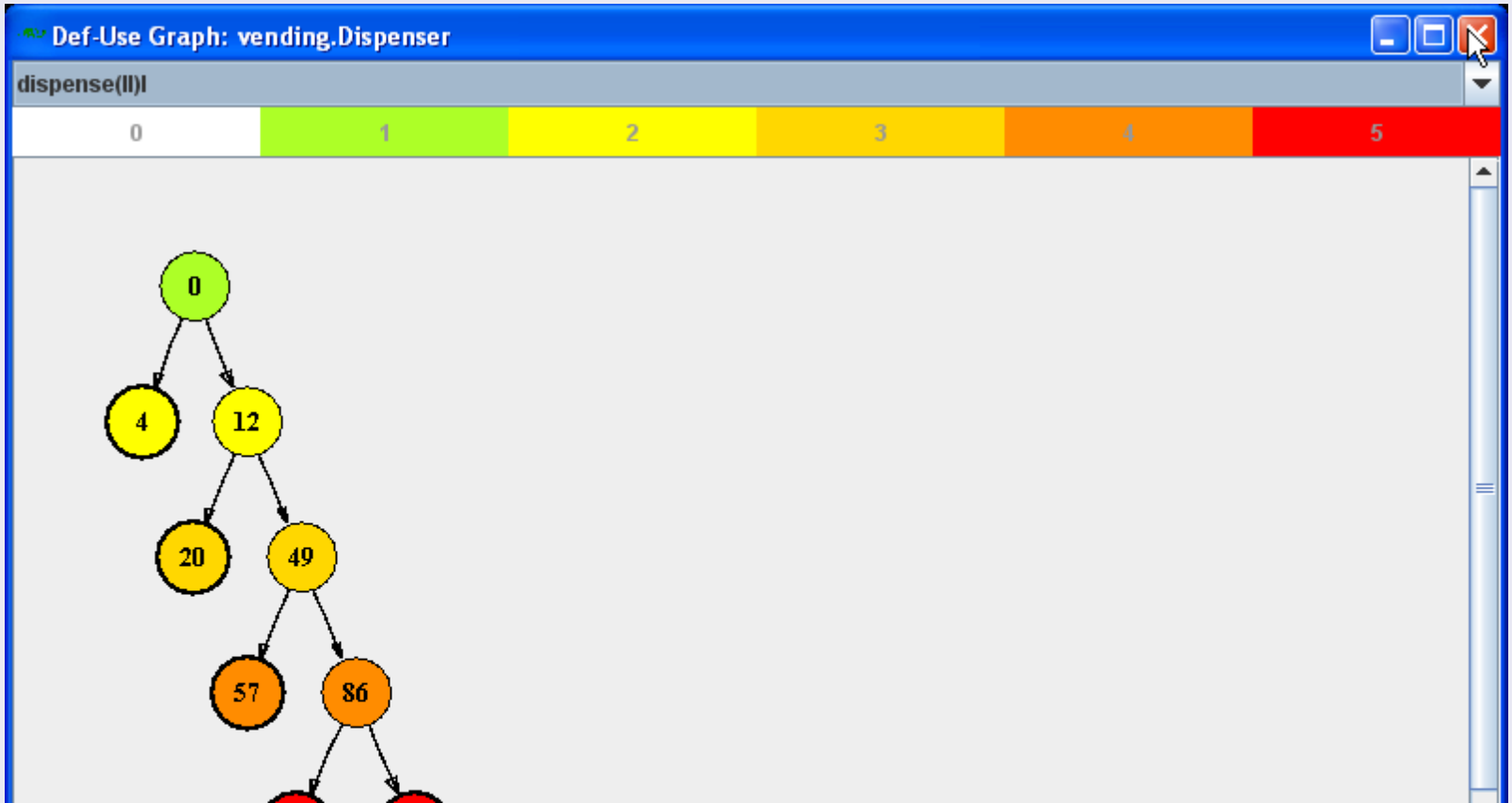
Demo



Def-use-graph

Def-Use Graph option shows the definition-use graph of a given method of the current class.

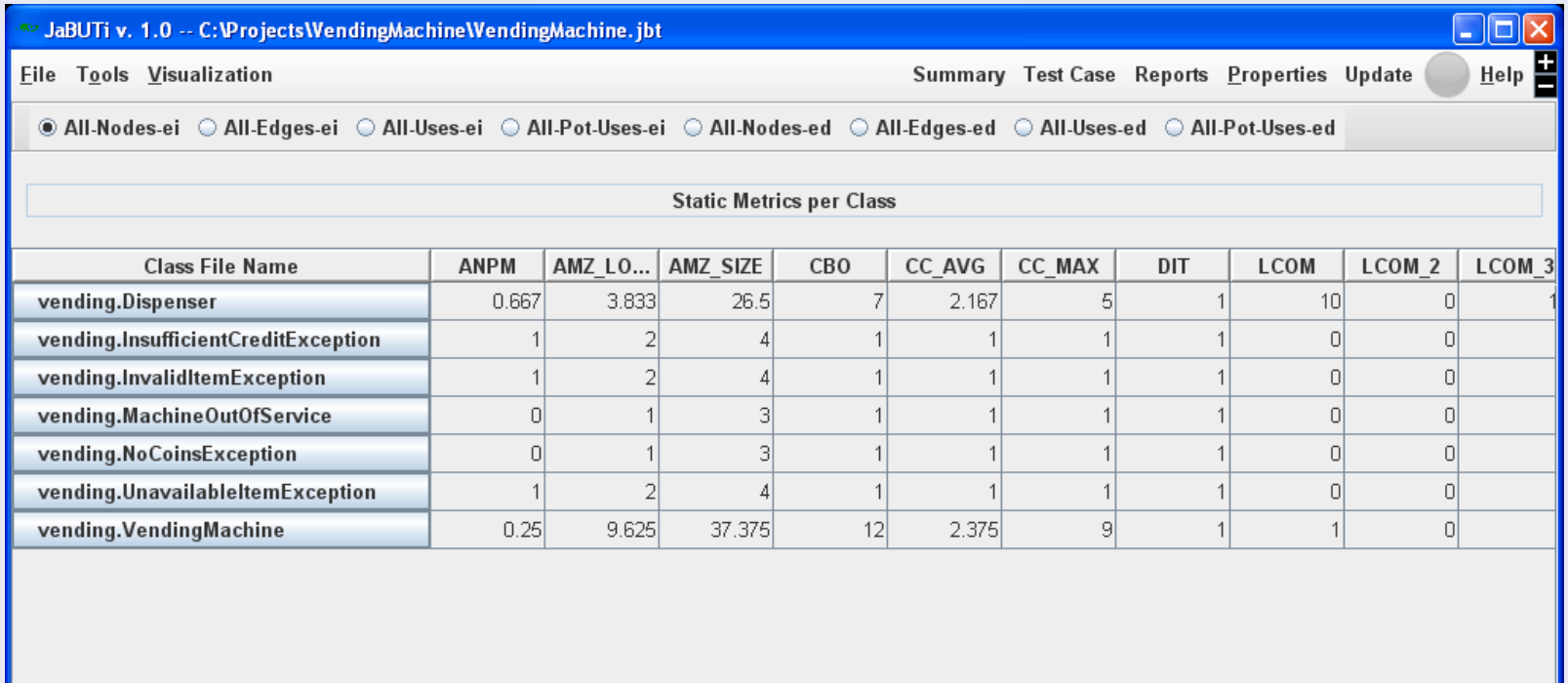
Demo



Complexity Metrics

Complexity Metrics option shows the resultant value of the set of complexity metrics implemented in JaBUTi for the complete set of user classes obtained from the base class (classes under testing).

Demo



Class File Name	ANPM	AMZ_LO...	AMZ_SIZE	CBO	CC_AVG	CC_MAX	DIT	LCOM	LCOM_2	LCOM_3
vending.Dispenser	0.667	3.833	26.5	7	2.167	5	1	10	0	1
vending.InsufficientCreditException	1	2	4	1	1	1	1	0	0	
vending.InvalidItemException	1	2	4	1	1	1	1	0	0	
vending.MachineOutOfService	0	1	3	1	1	1	1	0	0	
vending.NoCoinsException	0	1	3	1	1	1	1	0	0	
vending.UnavailableItemException	1	2	4	1	1	1	1	0	0	
vending.VendingMachine	0.25	9.625	37.375	12	2.375	9	1	1	0	

Summary

The **Summary** provides personalized coverage information in different levels of abstraction.

Levels of abstraction

- **By Criterion** shows the cumulative coverage information for each test criterion, considering all classes under testing.
- **By Class** shows the coverage information with respect to the current selected criterion for each individual class under testing.
- **By Method** shows the coverage information with respect to the current selected criterion for each individual method of each class under testing.

Summary

The **Summary** provides personalized coverage information in different levels of abstraction.

Levels of abstraction

- **By Criterion** shows the cumulative coverage information for each test criterion, considering all classes under testing.
- **By Class** shows the coverage information with respect to the current selected criterion for each individual class under testing.
- **By Method** shows the coverage information with respect to the current selected criterion for each individual method of each class under testing.

Summary

The **Summary** provides personalized coverage information in different levels of abstraction.

Levels of abstraction

- **By Criterion** shows the cumulative coverage information for each test criterion, considering all classes under testing.
- **By Class** shows the coverage information with respect to the current selected criterion for each individual class under testing.
- **By Method** shows the coverage information with respect to the current selected criterion for each individual method of each class under testing.

Summary

The **Summary** provides personalized coverage information in different levels of abstraction.

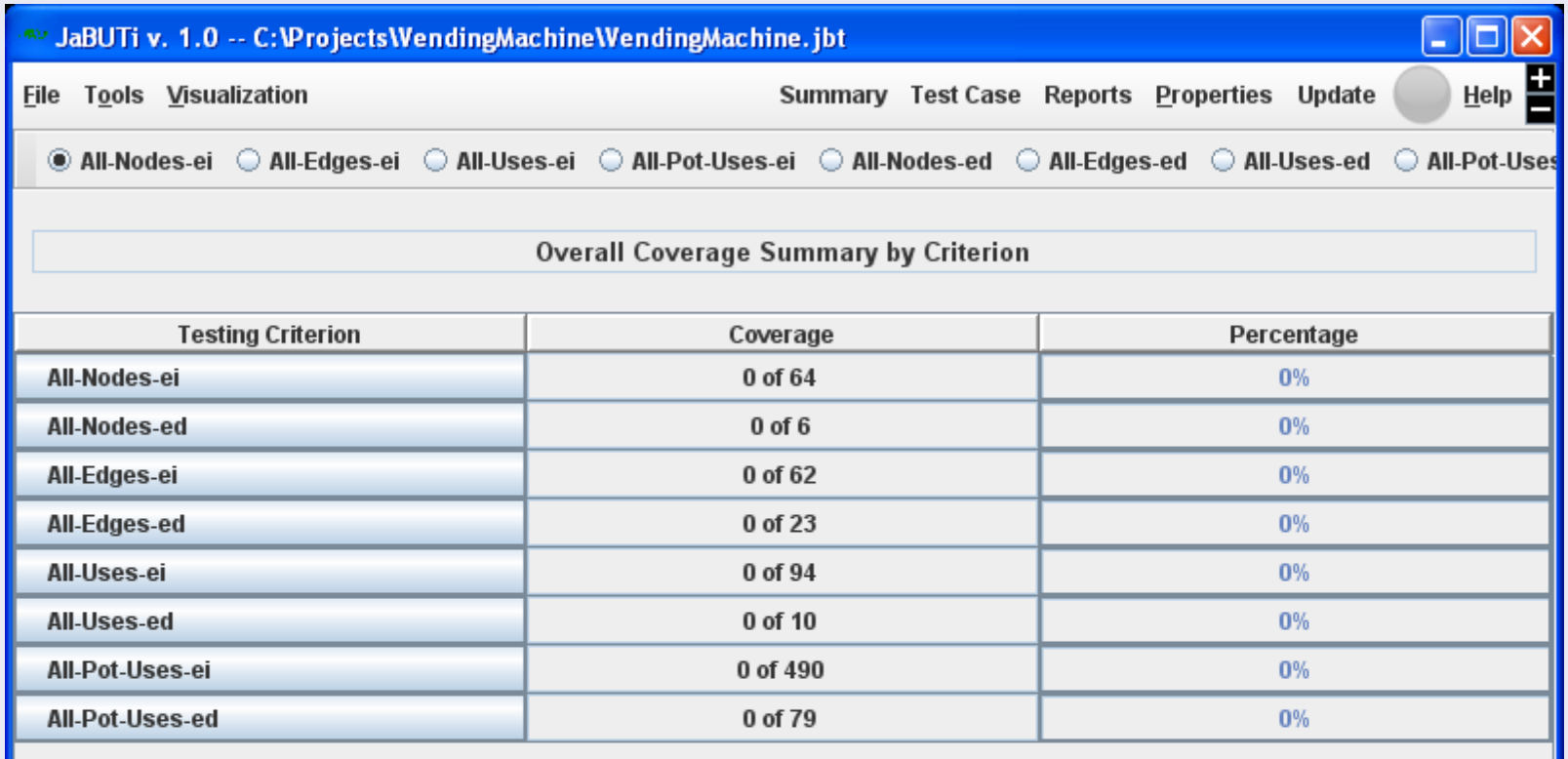
Levels of abstraction

- **By Criterion** shows the cumulative coverage information for each test criterion, considering all classes under testing.
- **By Class** shows the coverage information with respect to the current selected criterion for each individual class under testing.
- **By Method** shows the coverage information with respect to the current selected criterion for each individual method of each class under testing.

Summary By Criterion

Summary By Criterion shows the cumulative coverage information for each test criterion, considering all classes under testing.

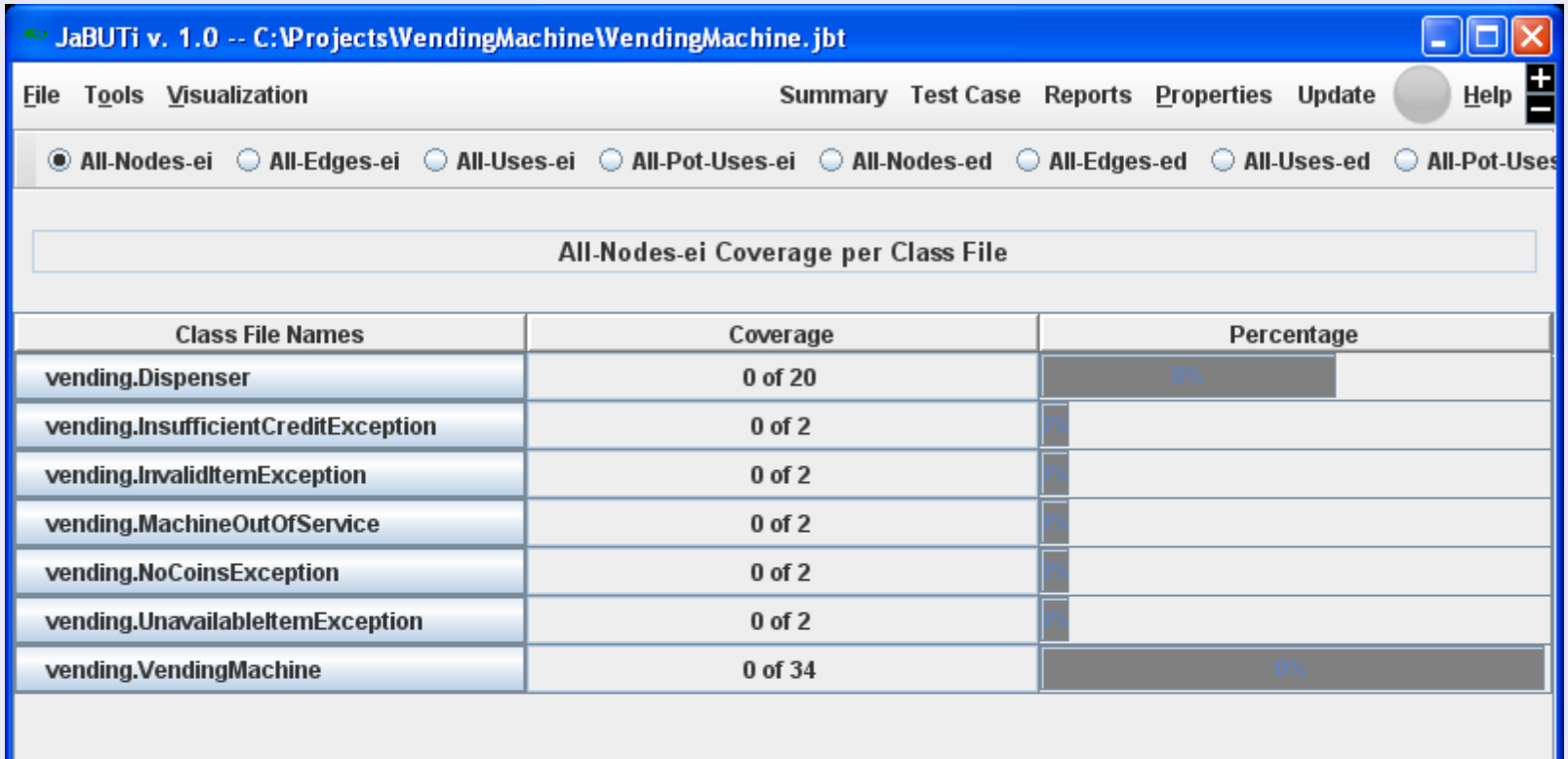
Demo



Summary By Class

Summary By Class shows the coverage information with respect to the current selected criterion for each individual class under testing.

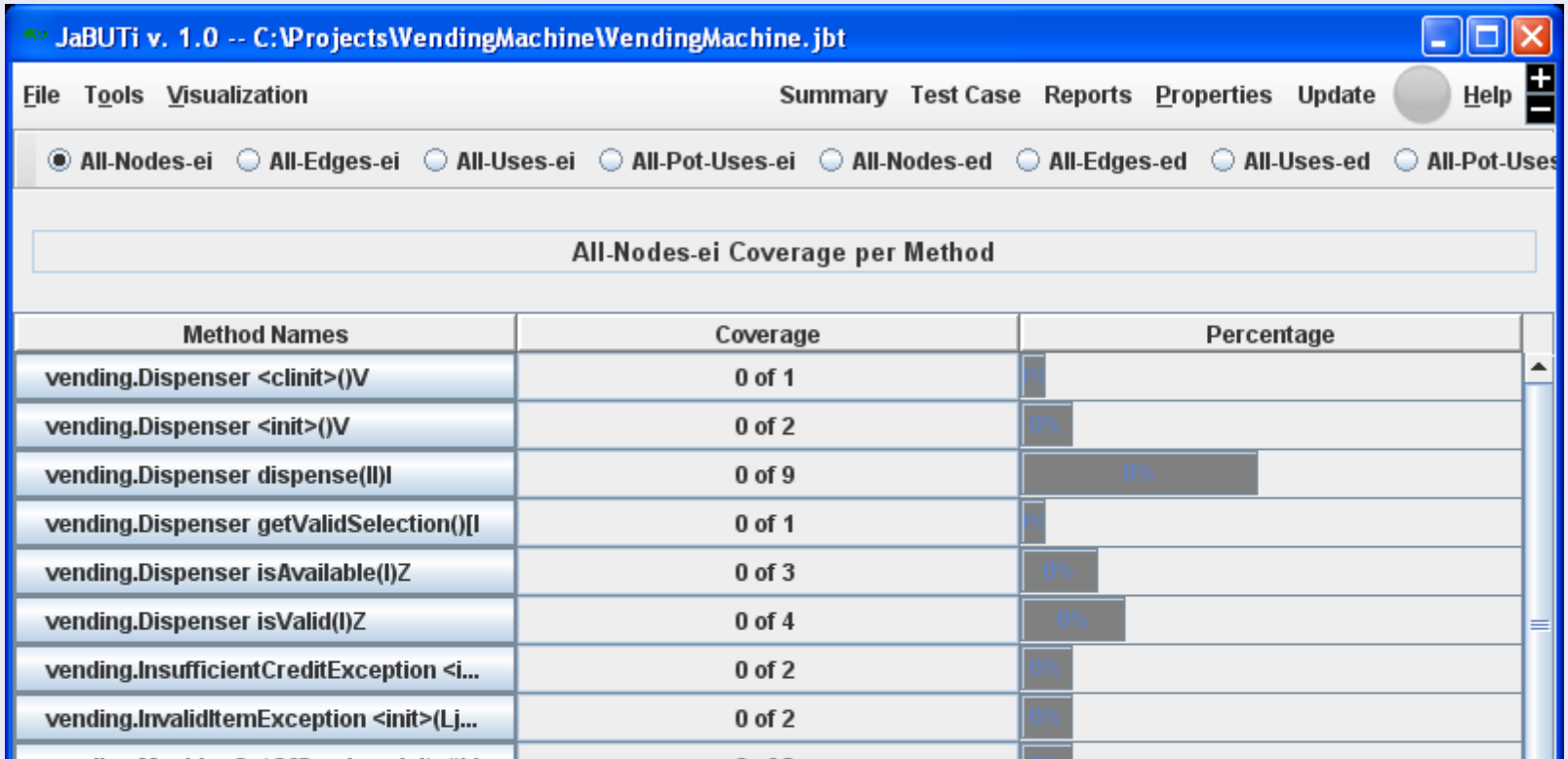
Demo



Summary By Method

Summary By Method shows the coverage information with respect to the current selected criterion for each individual method of each class under testing.

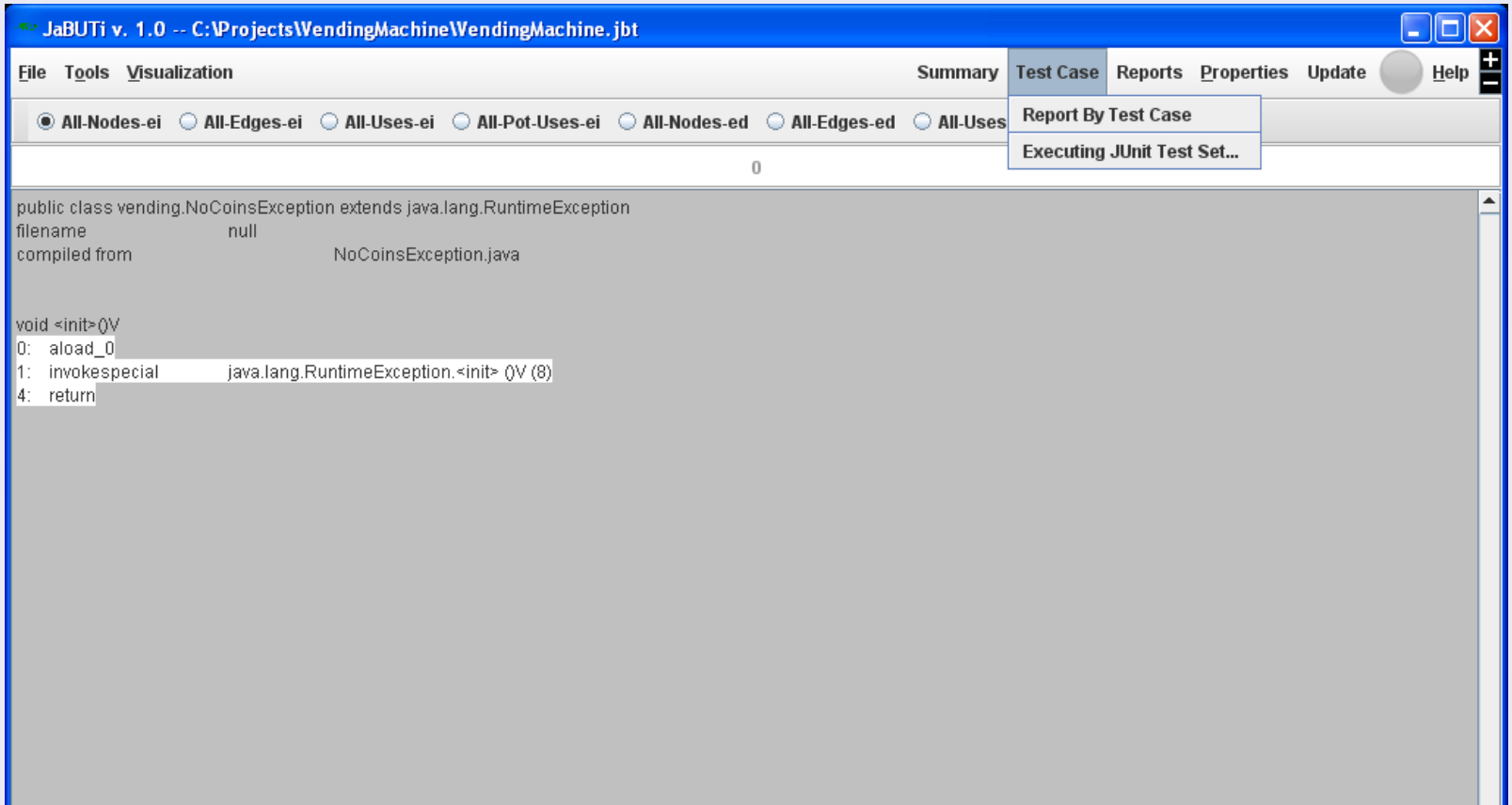
Demo



Test case

The **Test Case** menu provides options for test set manipulation and report generation.

Demo



Report By Test Case

Report By Test Case option shows the coverage information with respect to the current selected test criterion, for each individual test case, considering all class under testing, and also allows to enable/disable and delete/undelete test cases.

Demo

Importing from JUnit

Importing from JUnit option allows to import a test set generated according to the JUnit framework.

Demo

Reports

The **Reports** menu provides options to save JaBUTi's reports in HTML format.

Report types

- **Custom Reports** option allows to generate a custom HTML report from the current testing project considering different levels of granularity.
- **Summary to HTML** option allows to generate a HTML from any tabled style report provided by the JaBUTi graphical interface.

Reports

The **Reports** menu provides options to save JaBUTi's reports in HTML format.

Report types

- **Custom Reports** option allows to generate a custom HTML report from the current testing project considering different levels of granularity.
- **Summary to HTML** option allows to generate a HTML from any tabled style report provided by the JaBUTi graphical interface.

Custom Reports

Custom Reports option allows to generate a custom HTML report from the current testing project considering different levels of granularity.

Demo

Summary To HTML

Summary to HTML option allows to generate a HTML from any tabled style report provided by the JaBUTi graphical interface.

Demo

Properties

The **Properties** menu provides general configuration options.

Available properties

- **Look and Feel**: allows to change the look and feel style considering different options: Metal (default), Motif, and Windows.
- **Project Manager**: allows to verify and change the current set of classes under testing in the current project.
- **Test Server**: handles the configuration of the test server.
- **Available Memory Dialog**: shows current available memory on system.

Properties

The **Properties** menu provides general configuration options.

Available properties

- **Look and Feel**: allows to change the look and feel style considering different options: Metal (default), Motif, and Windows.
- **Project Manager**: allows to verify and change the current set of classes under testing in the current project.
- **Test Server**: handles the configuration of the test server.
- **Available Memory Dialog**: shows current available memory on system.

Properties

The **Properties** menu provides general configuration options.

Available properties

- **Look and Feel**: allows to change the look and feel style considering different options: Metal (default), Motif, and Windows.
- **Project Manager**: allows to verify and change the current set of classes under testing in the current project.
- **Test Server**: handles the configuration of the test server.
- **Available Memory Dialog**: shows current available memory on system.

Properties

The **Properties** menu provides general configuration options.

Available properties

- **Look and Feel**: allows to change the look and feel style considering different options: Metal (default), Motif, and Windows.
- **Project Manager**: allows to verify and change the current set of classes under testing in the current project.
- **Test Server**: handles the configuration of the test server.
- **Available Memory Dialog**: shows current available memory on system.

Look and Feel

Look and Feel allows to change the look and feel style considering different options: Metal (default), Motif, and Windows.

Demo

Project Manager

Project Manager allows to verify and change the current set of classes under testing in the current project.

Demo

Test Server

Test Server handles the configuration of the test server.

Demo

Available Memory Dialog

Available Memory Dialog shows current available memory on system.

Demo

Update

The **Update** menu provides a visual information every time an event that affect the coverage occurs.

- The **Update** button becomes red in case additional test cases are imported or appended in the end of the trace file.
- New test cases execution are an event that affects the coverage information.
- As soon as it is clicked, its color changes to gray, and the coverage information is updated.

Update

The **Update** menu provides a visual information every time an event that affect the coverage occurs.

- The **Update** button becomes red in case additional test cases are imported or appended in the end of the trace file.
- New test cases execution are an event that affects the coverage information.
- As soon as it is clicked, its color changes to gray, and the coverage information is updated.

Update

The **Update** menu provides a visual information every time an event that affect the coverage occurs.

- The **Update** button becomes red in case additional test cases are imported or appended in the end of the trace file.
- New test cases execution are an event that affects the coverage information.
- As soon as it is clicked, its color changes to gray, and the coverage information is updated.

Update

The **Update** option provides a visual information every time an event that affect the coverage occurs.

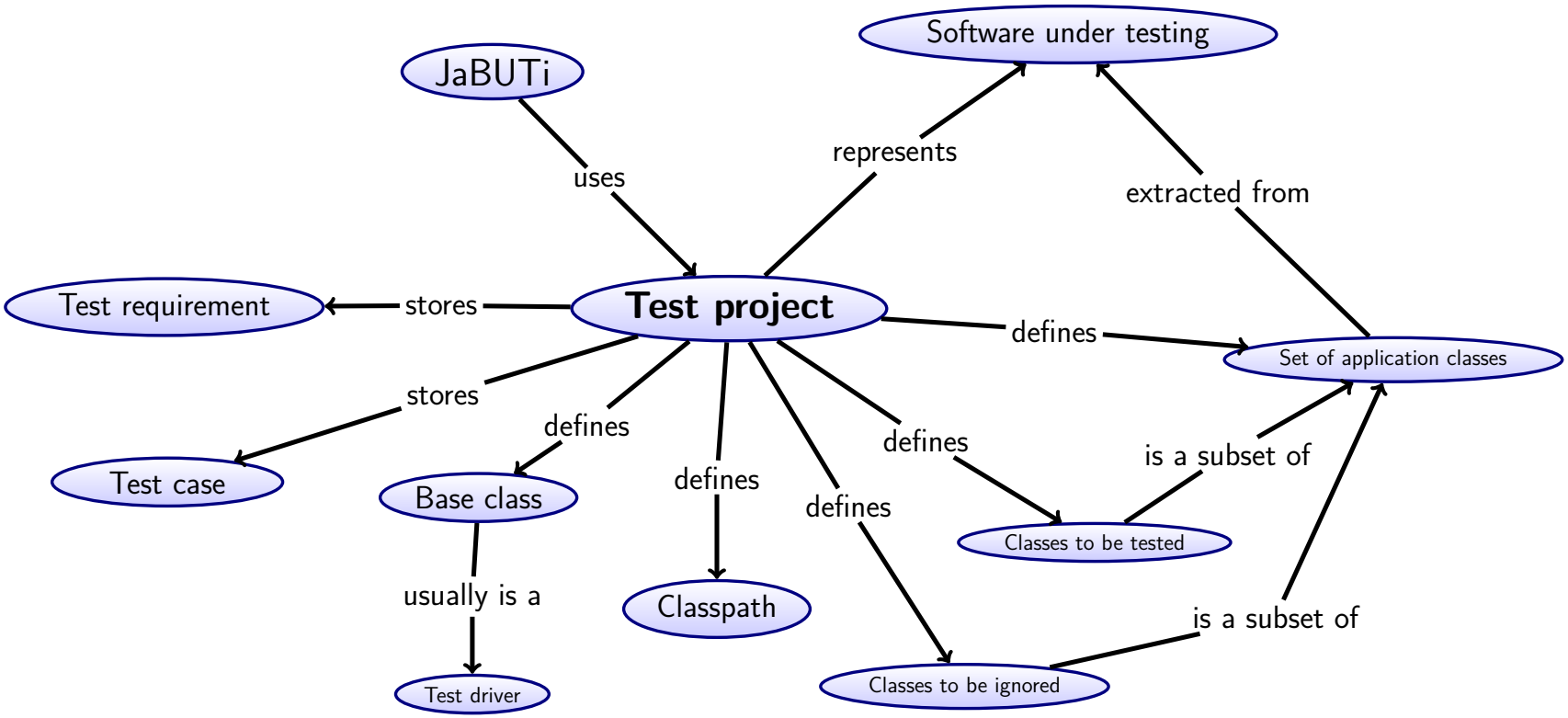
Demo

Help

The **Help** menu provides only one option to show information about the authors and developers of JaBUTi.

Demo

Test project



Test session → Test project

- A test session in JaBUTi is configured by creating a test project.

Test project

A test project is characterized by a file with the necessary information about the application under testing:

- base class file,
- complete set of classes required by the base class,
- set of classes to be instrumented (and tested),
- set of classes that should not be instrumented (and tested),
- `CLASSPATH` environment variable necessary to run the base class.

Test session → Test project

- A test session in JaBUTi is configured by creating a test project.

Test project

A test project is characterized by a file with the necessary information about the application under testing:

- base class file,
- complete set of classes required by the base class,
- set of classes to be instrumented (and tested),
- set of classes that should not be instrumented (and tested),
- `CLASSPATH` environment variable necessary to run the base class.

Test session → Test project

- A test session in JaBUTi is configured by creating a test project.

Test project

A test project is characterized by a file with the necessary information about the application under testing:

- base class file,
- complete set of classes required by the base class,
- set of classes to be instrumented (and tested),
- set of classes that should not be instrumented (and tested),
- `CLASSPATH` environment variable necessary to run the base class.

Test session → Test project

- A test session in JaBUTi is configured by creating a test project.

Test project

A test project is characterized by a file with the necessary information about the application under testing:

- base class file,
- complete set of classes required by the base class,
- set of classes to be instrumented (and tested),
- set of classes that should not be instrumented (and tested),
- `CLASSPATH` environment variable necessary to run the base class.

Test session → Test project

- A test session in JaBUTi is configured by creating a test project.

Test project

A test project is characterized by a file with the necessary information about the application under testing:

- base class file,
- complete set of classes required by the base class,
- set of classes to be instrumented (and tested),
- set of classes that should not be instrumented (and tested),
- `CLASSPATH` environment variable necessary to run the base class.

Test session → Test project

- A test session in JaBUTi is configured by creating a test project.

Test project

A test project is characterized by a file with the necessary information about the application under testing:

- base class file,
- complete set of classes required by the base class,
- set of classes to be instrumented (and tested),
- set of classes that should not be instrumented (and tested),
- `CLASSPATH` environment variable necessary to run the base class.

Test session → Test project

- A test session in JaBUTi is configured by creating a test project.

Test project

A test project is characterized by a file with the necessary information about the application under testing:

- base class file,
- complete set of classes required by the base class,
- set of classes to be instrumented (and tested),
- set of classes that should not be instrumented (and tested),
- `CLASSPATH` environment variable necessary to run the base class.

Test project

- It also stores some information generated by the tool:
  - test requirements (for every criteria supported by JaBUTi),
  - test cases execution results (test case name, and status).
- Those information are saved to a file which extension is `.jbt`.
  - This file is an XML document.

Test project

- It also stores some information generated by the tool:
  - test requirements (for every criteria supported by JaBUTi),
  - test cases execution results (test case name, and status).
- Those information are saved to a file which extension is `.jbt`.
  - This file is an XML document.

Test project and JaBUTi

- JaBUTi requires the bytecode of application under testing. If they are not available, the source code must be compiled beforehand.

Compilation procedure

1. Consider the Vending Machine example. To compile it, the following command can be used:

```
$ javac -g -d example example/vending/*.java
```

2. Observe that the debug option is activated (-g). JaBUTi can make use of the debug information to ease the navigation between graph and code.

Test project and JaBUTi

- JaBUTi requires the bytecode of application under testing. If they are not available, the source code must be compiled beforehand.

Compilation procedure

1. Consider the Vending Machine example. To compile it, the following command can be used:

```
$ javac -g -d example example/vending/*.java
```

2. Observe that the debug option is activated (-g). JaBUTi can make use of the debug information to ease the navigation between graph and code.

Test project and JaBUTi

- JaBUTi requires the bytecode of application under testing. If they are not available, the source code must be compiled beforehand.

Compilation procedure

1. Consider the Vending Machine example. To compile it, the following command can be used:

```
$ javac -g -d example example/vending/*.java
```

2. Observe that the debug option is activated (-g). JaBUTi can make use of the debug information to ease the navigation between graph and code.

Test project

- From the generated `.class` files, the user can create a test project using JaBUTi.

Execute JaBUTi

1. Invoke JaBUTi's graphical interface.
   - Double-click `Jabuti-bin.jar`.
   - Supposing that JaBUTi is installed on `/opt/JaBUTi`, it is possible to start the application from the command line:

```
$ java -jar /opt/JaBUTi/Jabuti-bin.jar
```

Create new project

1. Select a base `.class` file from `File/OpenClass` menu.
2. Select the directory where the base class file is located, and then select the base class file itself.
3. Check the `Package` field.
   - Once the base class file is selected, the tool automatically identifies the package that it belongs to and fills out the `Package` field with the package's name.
4. Set the `Classpath` field.
   - It should contain only the path necessary to run the selected base class.

Create new project

6. Click the `Open` button. The `ProjectManager` windows will be displayed.
   - From the selected base class file, the tool identifies the complete set of classes necessary to its execution.
7. From the `ProjectManager` window (left side), the user can select the class files that will be tested.
   - At least one class file must be selected.
   - The base class, if it is the driver, should not be selected.

Create new project

6. Click the `Open` button. The `ProjectManager` windows will be displayed.
   - From the selected base class file, the tool identifies the complete set of classes necessary to its execution.
7. From the `ProjectManager` window (left side), the user can select the class files that will be tested.
   - At least one class file must be selected.
   - The base class, if it is the driver, should not be selected.

Create new project

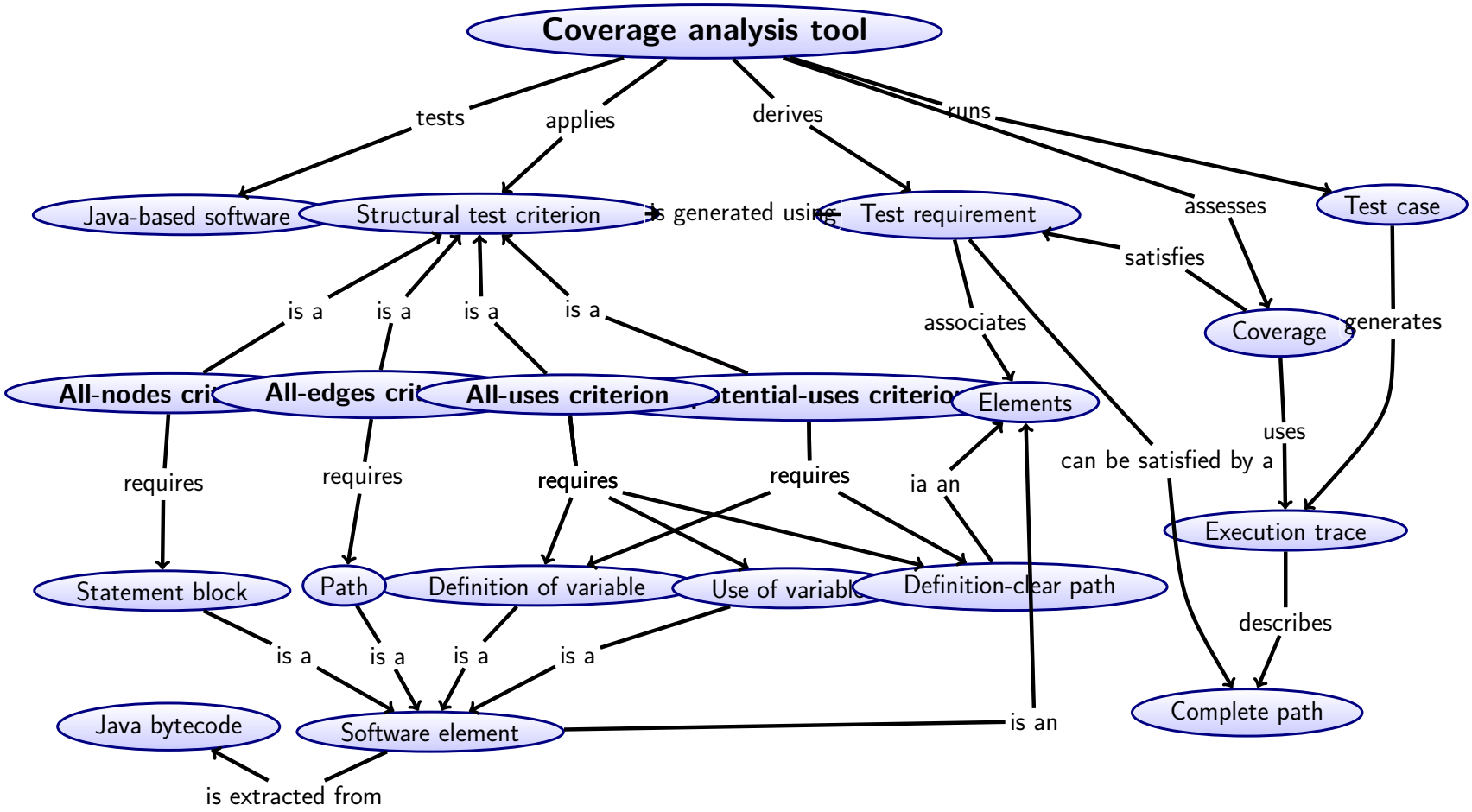
8. A name must be given to the project being created by clicking on the `Select` button.
9. Click the `Ok` button. After this action, JaBUTi will:
   - create a new project,
   - construct the DUG for each method of each class under testing,
   - derive the complete set of test requirements for each criterion,
   - calculate the weight of each test requirement,
   - and show the bytecode of a class under testing.

Create new project

8. A name must be given to the project being created by clicking on the `Select` button.
9. Click the `Ok` button. After this action, JaBUTi will:
   - create a new project,
   - construct the DUG for each method of each class under testing,
   - derive the complete set of test requirements for each criterion,
   - calculate the weight of each test requirement,
   - and show the bytecode of a class under testing.

Test project demonstration

Coverage analysis tool



Coverage analysis tool

JaBUTi's main tool is the coverage analysis tool. It uses test criteria from the structural test technique to create test requirements that a given test set must satisfy.

Basic workflow for coverage analysis

1. selection of classes to be tested;
2. test requirements creation;
3. visualization of test requirements, definition-use graph, and related bytecode and source code;
4. instrumentation of classes to be tested;
5. specification and execution of test cases;
6. trace collection and coverage calculation;
7. test case management;
8. identification of infeasible test cases (test requirement management).

Coverage analysis tool
Selection of classes to be tested

1. Create a test project for every set of classes that must be tested.
   - For small size software (e.g., less than 100 classes), all the classes of the application under testing can be selected.
   - For bigger software, it is recommended to select just a subset of classes (e.g., those of a single package).

Example

How JaBUTi create test requriements

1. JaBUTi will create test requirements for all the test criteria it supports.
2. As no test case has been imported yet, the coverage will be zero.

Example

Visualization

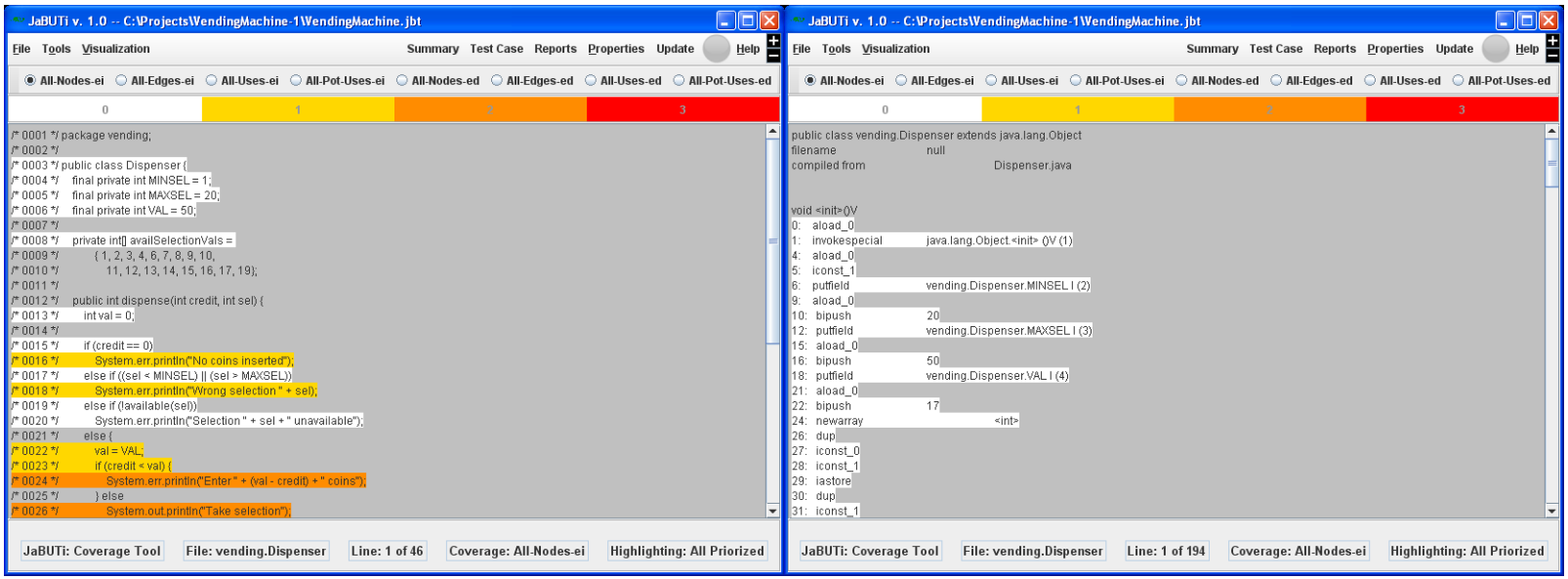
- JaBUTi generates, for every method, a definition-use graph, as well as the visualization of the bytecode and of the source code (when available).

Available information

- For each node in the graph, the following information is available:
  - set of variables used,
  - set of variables defined,
  - corresponding source code lines,
  - corresponding bytecode “line” (PC).

Source code and bytecode

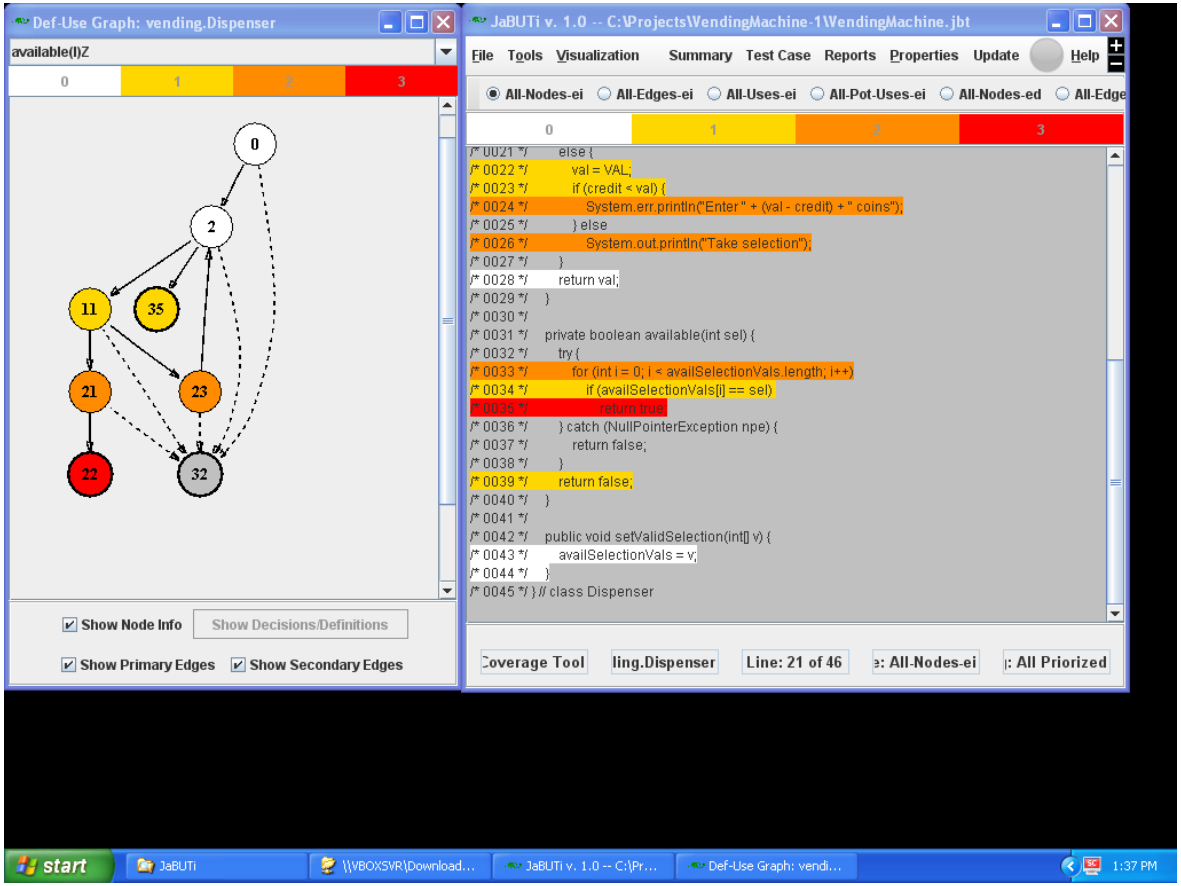
- The corresponding source code of the current class is displayed in colors, mapped back from the bytecode, facilitating the identification of which part of the source code should be covered first.



DUG representation

- JaBUTi represents the definition-use graph using three different types of nodes – common nodes, exit nodes, and call nodes – and two different types of edges – primary edges, and secundary edges.
- JaBUTi represents:
  - common nodes with single-lined circles,
  - exit nodes with bold single-lined circles,
  - call nodes with double-lined circles,
  - primary edges with continuous lines,
  - secondary edges with dashed lines.

Coverage analysis tool
Visualization of definition-use graph



Test requirement weight

- Depending on which test criterion is active, the bytecode, source code and definition-use graph is colored in a different way in JaBUTi.
- The colors represents the weight of the test requirements.
  - A test requirement with weight zero (a covered requirement) is painted in white.

Example

Instrumentation

1. Before running the test cases, JaBUTi need to instrument the classes under testing.
   - Instrumentation is required to insert probe instructions (that will generate the trace file).
2. Instrumentation can done either by select `File/Saveinstrumentedclasses` or when importing test cases.
   - If the set of selected or ignored classes were changed after the generation of a package of instrumented classes, you must use the `Saveinstrumentedclasses` menu.
3. A package (.jar) will be created with the instrumented classes.

Example

9. Test cases must be imported to the test project.
    - JaBUTi supports test classes defined with JUnit.
10. Another option is to just run the tests using the instrumented Jar package.

Example

Trace collection and coverage analysis

11. The trace file, generated by the instrumented classes, must be collected.
    - The trace information with respect to the current execution is appended in a trace file with the same name of the testing project but with the extension `.trc` instead of `.jbt`.
12. Trace information is used to update the coverage of the test set with respect to the test criteria supported by JaBUTi.
    - Every time the size of the trace file increase, the Update button in the JaBUTi's graphical interface becomes red, indicating that the coverage information can be updated.

Example

13. Test cases can be disabled or enabled at any time.
14. They can also be deleted from the test set.

Example

15. JaBUTi does not detect infeasible requirements.
    - Actually this is a non-computable problems.
16. Thus, it is necessary to manually define such requirements as infeasible.
17. JaBUTi provides a test requirement management feature to accomplish that.

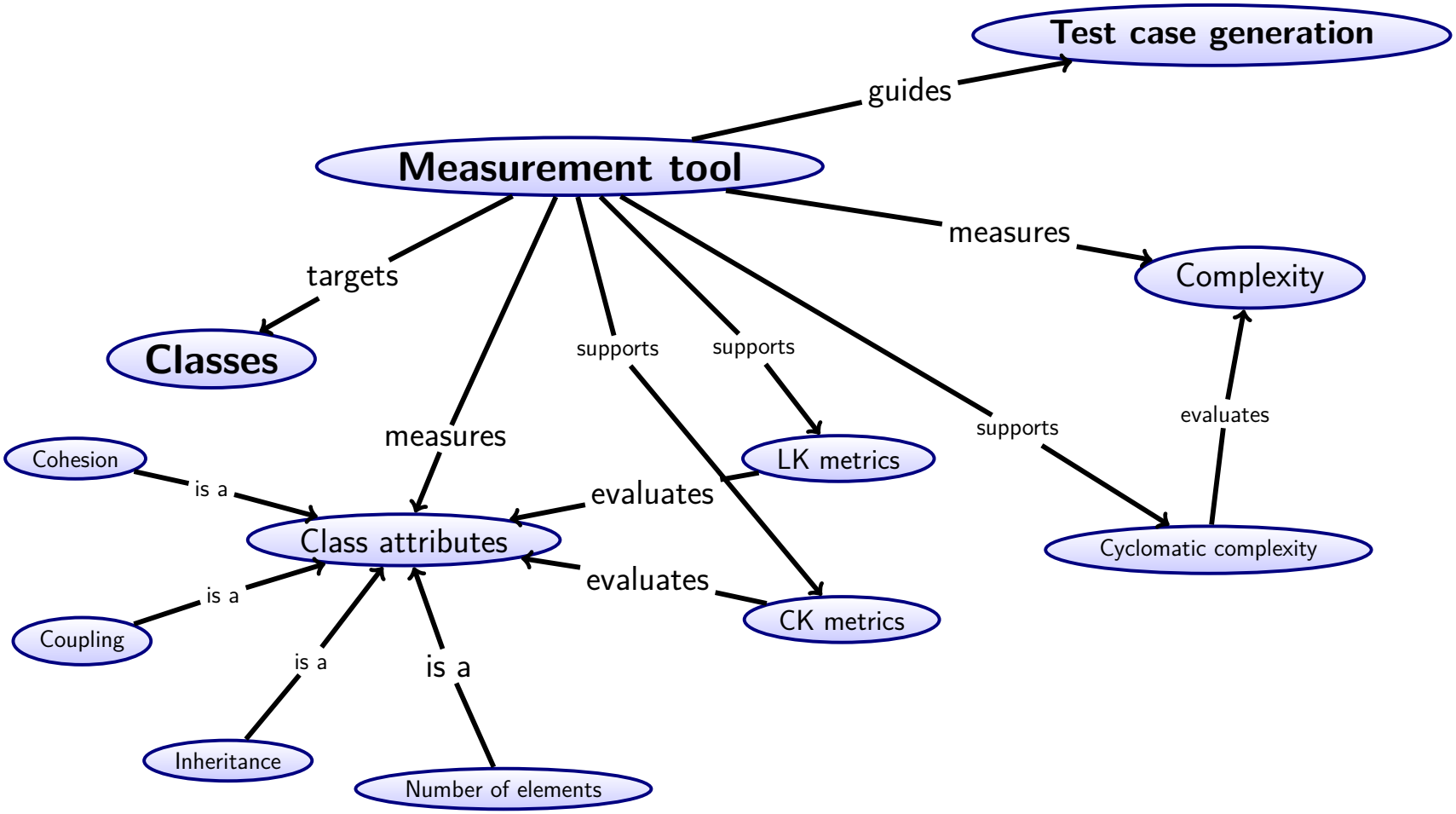
Example

Reporting test session results

- To evaluate the coverage obtained, the tool provides personalized tabled style testing reports that can be accessed from the Summary and Test Case menus.
- The tool provides reports with respect to each test criterion, class file or method.
  - When showing the summary by class or by method, the tester can choose, among the available testing criteria, which one he wants to evaluate.

Example

Measurement tool



Software measurement

- Measurements can be used to define a test strategy:
  - Which classes, methods and test requirements should be tested first?

Measurement tool

- JaBUTi implements several metrics to aid the tester in the definition of the test strategy:
  - Test requirement weight based on dominator and super-block analysis.
  - Static metrics for classes and methods.

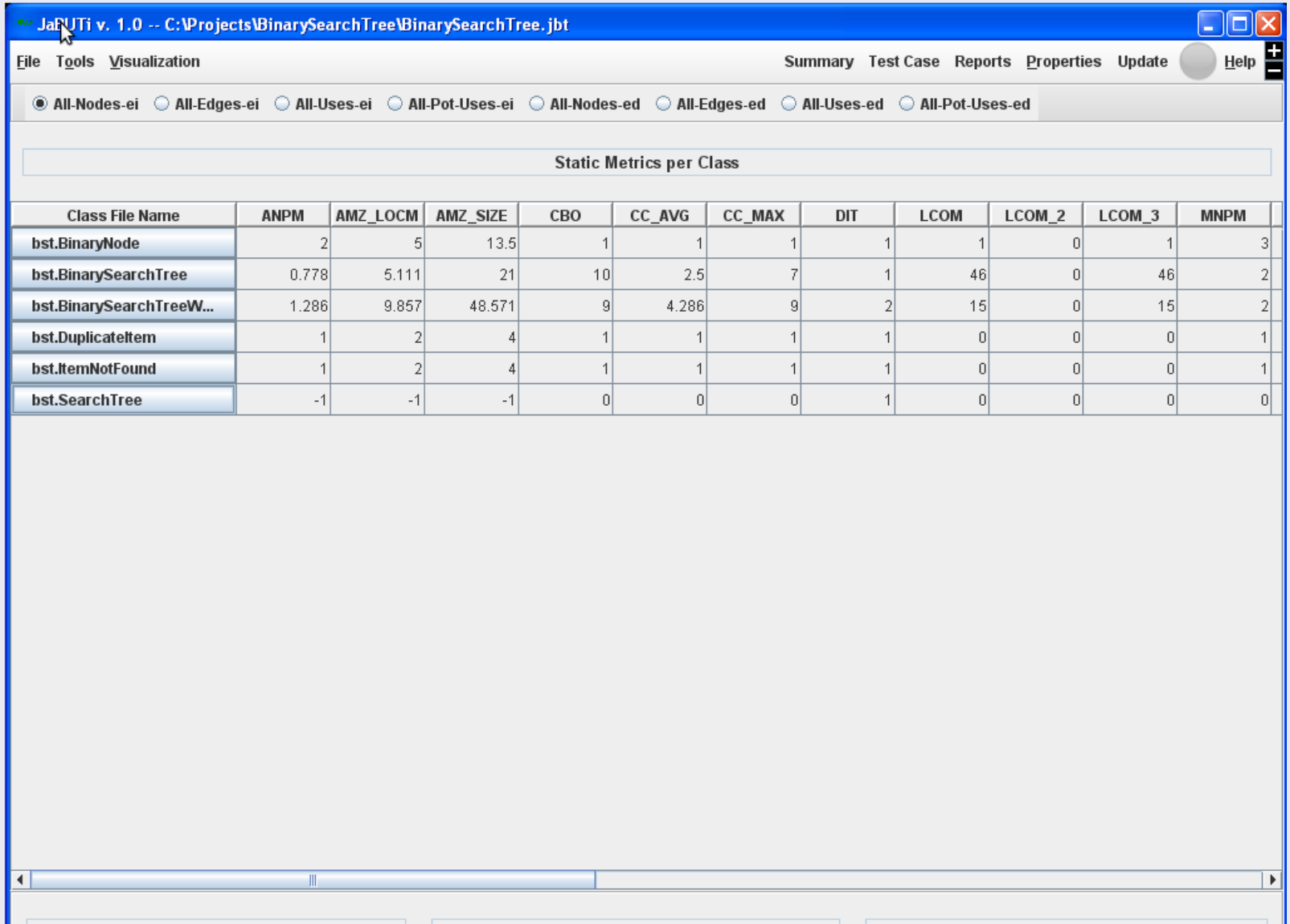
Software measurement

- Measurements can be used to define a test strategy:
  - Which classes, methods and test requirements should be tested first?

Measurement tool

- JaBUTi implements several metrics to aid the tester in the definition of the test strategy:
  - Test requirement weight based on dominator and super-block analysis.
  - Static metrics for classes and methods.

Example



Class File Name	ANPM	AMZ_LOCM	AMZ_SIZE	CBO	CC_AVG	CC_MAX	DIT	LCOM	LCOM_2	LCOM_3	MNPM
bst.BinaryNode	2	5	13.5	1	1	1	1	1	0	1	3
bst.BinarySearchTree	0.778	5.111	21	10	2.5	7	1	46	0	46	2
bst.BinarySearchTreeW...	1.286	9.857	48.571	9	4.286	9	2	15	0	15	2
bst.DuplicateItem	1	2	4	1	1	1	1	0	0	0	1
bst.ItemNotFound	1	2	4	1	1	1	1	0	0	0	1
bst.SearchTree	-1	-1	-1	0	0	0	1	0	0	0	0

Metrics

- Three types of measures are provided by JaBUTi:
  - Lorenz and Kidd metrics (LK) [1],
  - Chidamber and Kemerer metrics (CK) [2], and
  - Complexity based metrics [3].

LK and CK metrics

- LK and CK metrics are used for classes:
  - number of methods of a class,
  - average of number of parameters of the method of a class,
  - number of attributes of a class,
  - average size of the methods of a class,
  - number of implemented interfaces,
  - lack of cohesion in methods.

Example

Class	AMZ_LOCM	CBO	DIT
bst.BinaryNode	5.000	01	1
bst.BinarySearchTree	5.111	10	1
bst.BinarySearchTreeWithRank	9.857	09	2

- Complexity metrics are used for methods:
  - Average value of the complexity metric of the methods of a class.
  - Maximum value of the complexity metric of the methods of a class.

Example

Class	CC_AVG	CC_MAX
bst.BinaryNode	1.000	1
bst.BinarySearchTree	2.500	7
bst.BinarySearchTreeWithRank	4.286	9

Test strategy

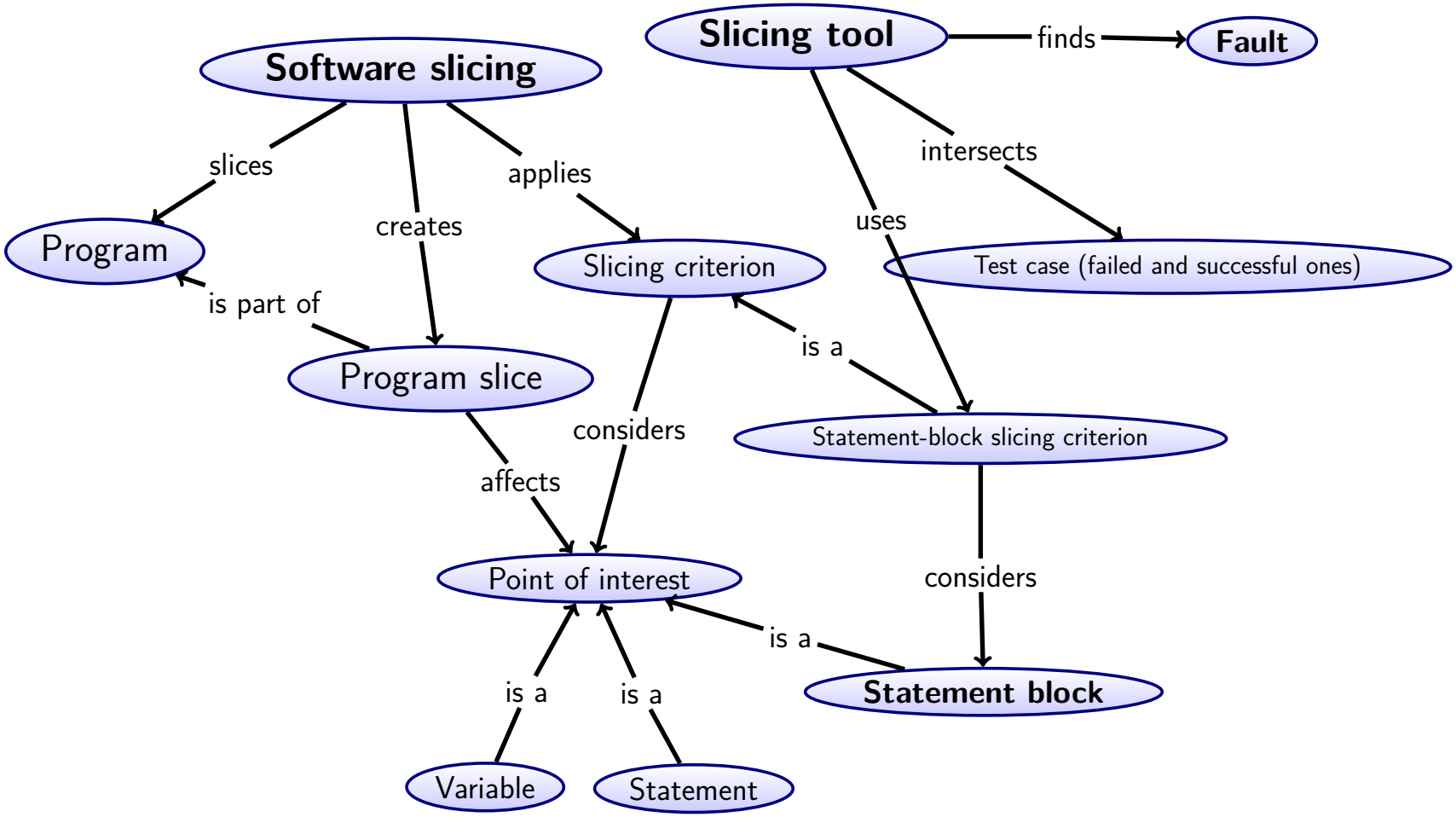
- JaBUTi does not establish a test strategy, nor suggest a test requirement weight using the measurements herein described.
- It is the tester duty to use them and define himself the test requirements which should be satisfied first.

Example

- `bst.BinarySearchTreeWithRank` and `bst.BinarySearchTree` have the highest complexity (CC_AVG, CC_MAX), larger methods (AMZ_LOCM, AMZ_SIZE), and coupling (CBO).
- The absolute size of `bst.BinarySearchTreeWithRank` is smaller than `bst.BinarySearchTree` (LCOM).
- `bst.BinarySearchTreeWithRank` is higher in the hierarchy than `bst.BinasrySearchTree` (DIT).
- Thus, `bst.BinarySearchTreeWithRank` is a good candidate to start the testing activity with.

Example

Slicing tool



Slicing tool

- Slicing tool uses program slicing to highlight, for a given set of successful and failure test cases, the parts of the program under testing that have a higher probability of having a fault.

Program slicing

- Program slicing can be used to help engineers to understand code.
  - A backward slice from a point in the program identifies all parts of the code that contribute to that point.
  - A forward slice identifies all parts of the code that can be affected by the modification to the code at the slice point.

Example

- Suppose a proposed program modification only changes the value of variable v at program point p.
- If the forward slice with respect to v and p is disjoint from the coverage of a test set t, then test set t does not have to be rerun.
- Suppose a coverage tool reveals that a use of variable v at program point p has not been tested.
- The input date required to cover p can be found in the backward slice of v with respect to p.

Static and dynamic slice

- An important distinction exists between a static and a dynamic slice.
- Static slices are computed without making assumptions regarding a program's input, which provides the set of all statements that might affect the value of a given variable.
- Dynamic slices relies on a the execution trace information of the software, providing all statements that actually affect the value of a variable.

- Using dynamic tracing, JaBUTi can be used to find the fault triggered by a failed test case.

Slicing tool

- Slicing tool is available through the `Tools/SlicingTool` menu option.
- By changing to the slicing tool, the tester has to choose, among the test cases:
  - the ones that cause the fault;
  - and the ones that do not reveal the fault.
- Based on the execution path of the failed and successful test cases, the tool highlights the part of the code that have a higher probability of containing the fault.

Slicing tool

- JaBUTi uses a simple dynamic slice criterion, based on control-flow information, to identify a subset of statements that probably contain the fault.
- The idea is to compute:
  - the failed set F_S of BG nodes (the execution path) of a failed test case (which includes all statements executed by the failed test case)
  - the successful set S_S of BG nodes considering successful test cases,
  - the difference and the intersection of these sets to establish to prioritize the statements executed by the failed test case that are candidate to trigger the failure.

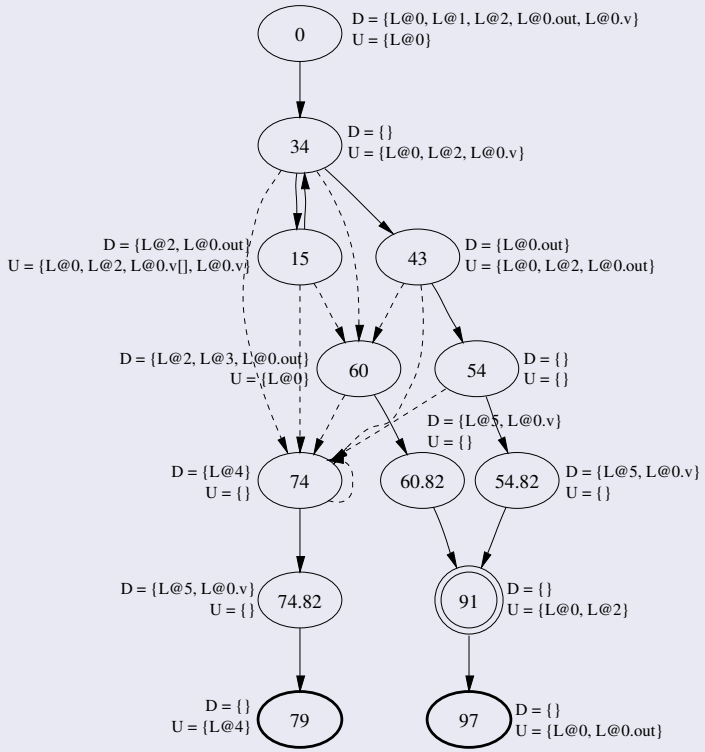
Slicing tool

- Using such approach, instead of the complete set of BG nodes N (which represents all statements of a method), the tester has only to consider the subset of BG nodes present in F_S
  - The other BG nodes contains the statements not executed by the failed test case and that cannot contain the fault.
- Moreover, considering the subset of nodes executed by the successful test cases, the most probably location of the fault is in the statements executed by the failed test case but not executed by the successful test cases, *i.e.*, the subset $F_S \setminus S_S$.

Example

Consider the BG presented below. It represents a program that outputs the average of the numbers in an array.

Average



Example

1. N is the complete set of BG nodes ($N = \{0, 15, 34, 43, 54, 54.82, 60, 60.82, 74, 74.82, 79, 91, 97\}$).
2. Suppose a failed test case that goes though BG nodes $F_S = \{0, 34, 15, 34, 43, 54, 54.82, 91, 97\}$ and a successful test case that goes through BG nodes $S_S = \{0, 34, 43, 60, 60.82, 91, 97\}$.
3. The most probably locations for the fault are in the statements in nodes 15, 54 or 54.82, since they are only executed by the faulty test case ($F_S \setminus S_S$).
4. If the fault is not located on such statements, it will be found in the other statements that compose the BG nodes 0, 34, 43, 91 and 97 ($F_S \cap S_S$). All the other BG nodes have not to be analyzed.

Example

References


LORENZ, M.; KIDD, J. *Object-Oriented Software Metrics*. 1. ed. New Jersey, USA: Prentice Hall, 1994. 146 p. (Prentice-Hall Object-Oriented Series).


CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions of Software Engineering*, v. 20, n. 6, p. 476–493, jun. 1994. ISSN 0098-5589.


MCCABE, T. J. A software complexity measure. *IEEE Transactions of Software Engineering*, v. 2, n. 4, p. 308–320, dez. 1976.


AMMANN, P.; OFFUTT, J. *Introduction to software testing*. Cambridge, UK: Cambridge University Press, 2008. Disponível em: <http://cs.gmu.edu/ offutt/softwaretest/>.

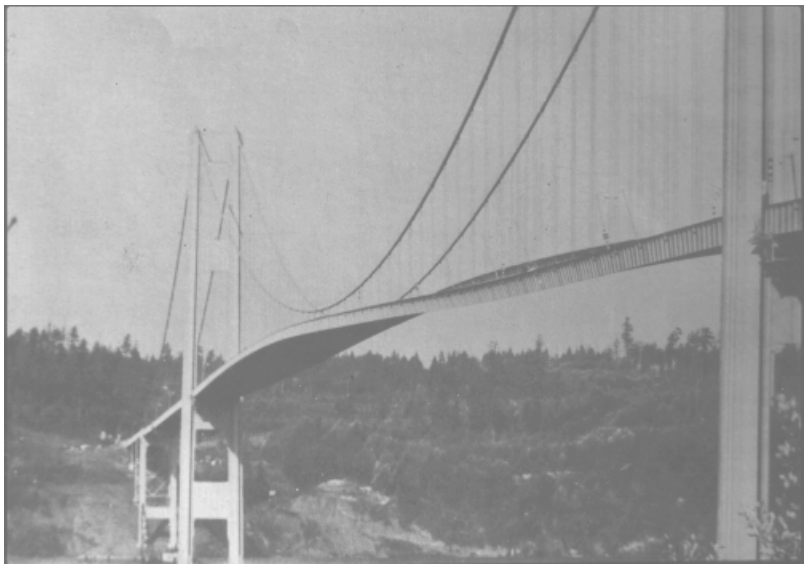
Credits

- Reviewers:
  - Fabiano Cutigi Ferrari
  - Otávio Augusto Lazzarini Lemos

Product failures

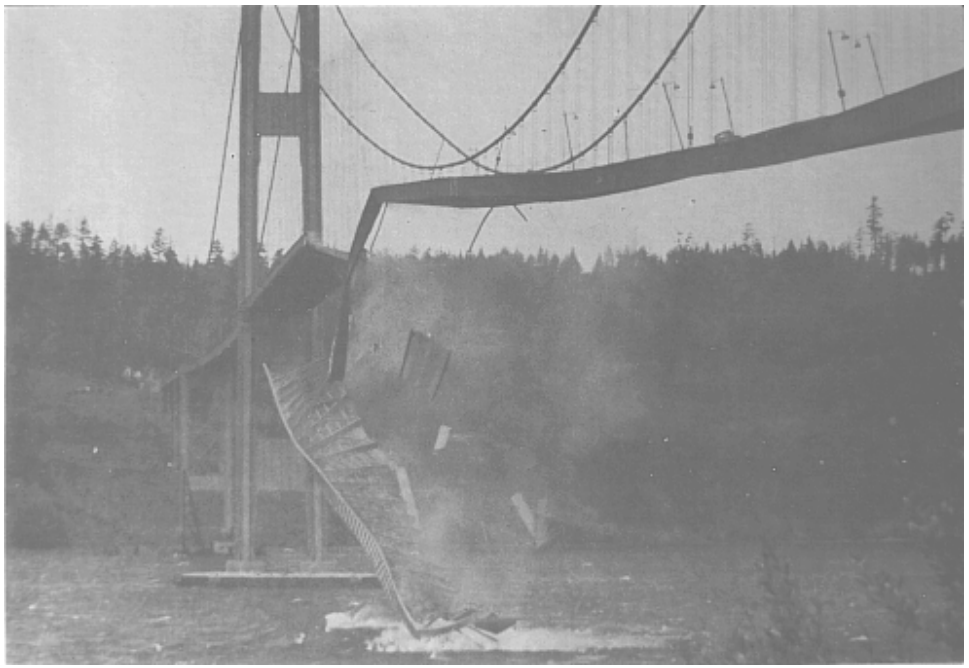
Tacoma Narrows Bridge
Description

- The Tacoma Narrows Bridge is a pair of mile-long suspension bridges in the U.S. state of Washington, across the Tacoma Narrows, between Tacoma and the Kitsap Peninsula.
- It was opened to traffic on July 1, 1940.



Tacoma Narrows Bridge
Problem

- It collapsed four months later on November 7, 1940, at 11:00 AM (Pacific time) due to a physical phenomenon known as aeroelastic flutter caused by a 67 kilometers per hour (42 mph) wind.



- Due to financing issues, shallower supports-girders 2.4 m deep were used.
  - This approach meant a slimmer, more elegant design and reduced construction costs compared to the original design.
- The decision to use shallow and narrow girders proved to be the first bridge's undoing.
  - With such girders, the roadbed was insufficiently rigid and was easily moved about by winds.
  - Bridge nicknamed as "Galloping Gertie".

Tacoma Narrows Bridge
Diagnostic

- The failure of the bridge occurred when a never-before-seen twisting mode occurred, from winds at a mild 40 miles per hour (64 km/h).
- The amplitude of the motion produced by the fluttering increased beyond the strength of a vital part, in this case the suspender cables.
- Once several cables failed, the weight of the deck transferred to the adjacent cables that broke in turn until almost all of the central deck fell into the water below the span.

- Two solutions were devised:
  - Drill some holes in the lateral girders and along the deck so that the air flow could circulate through them (in this way reducing lift forces).
  - Give a more aerodynamic shape to the transverse section of the deck by adding fairings or deflector vanes along the deck, attached to the girder fascia.
- The second option was the chosen one; but it was not carried out, because the bridge collapsed five days after the solution was proposed.

New York Subway

- In 1995, a train crashed into another train, killing the driver and hurting another 54 people.
- The distance between the signals (projected in 1918) was smaller than the distance required to stop the current trains (that are bigger, heavier and faster).
- The trains were updated without modification in the control system.
- Failure cause:
  - They updated part of the system, but did not validated the system as a whole.

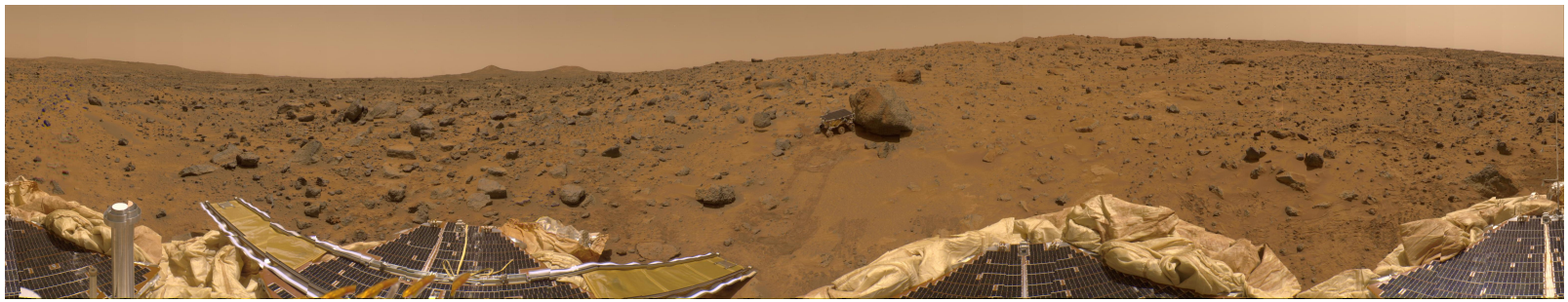
Challenger mishap

- In 1986, 73 seconds after taking off, a explosion destroyed the space shuttle Challenger, killing 7 astronauts.
- Investigations arrived at the conclusion that some rocket joints were not projected for the temperature and pressure they were suffering.
- Failure cause:
  - The pressure specification was not correct for the system requirement.
  - The tests were insufficient to detect the failure.

Software failures

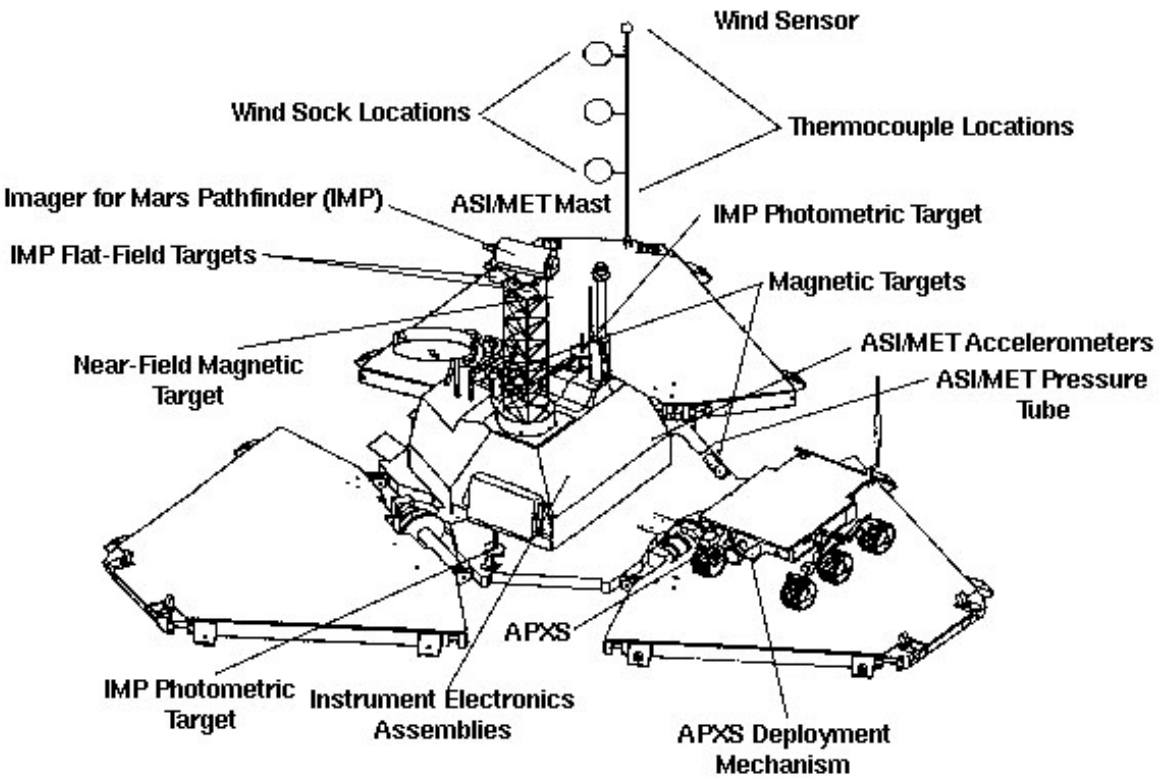
Mars Pathfinder
Description

- Mars Pathfinder was a low-cost planetary mission to Mars
  - Landed on Martian surface at July 4th, 1997.
- It demonstrated several new technologies for Mars exploration:
  - Airbag-based landing mechanism.
  - Use of rovers to collect and analyse soil and rock samples.
- The spacecraft had two parts: the lander and the rover.



Mars Pathfinder
Problem

- After collecting data for a long period, the lander would reset itself and all the data was lost.



- The lander software was concurrent and employed preemptive scheduling.
- Each thread had a priority and exchanged information between each other using an information bus.
  - Information bus = shared memory which access was controlled by a mutex.
- The information bus management system itself was a thread that run frequently, with a high priority.
- Another applications that run in the system was:
  - meteorological system, run much less frequently and with lower priority, and
  - communication thread: medium priority, but run frequently.

- This combination of threads **usually** worked correctly.
- However, the information bus management system could be blocked in the mutex in the following situation:
  1. Communication thread is scheduled and uses the processor, as it has higher priority than the meteorological thread.
  2. Communication thread can take as much time as it needs to run.
  3. However, a timer is expired whenever the information bus thread is not executed for a long time.
  4. The corrective measure taken by the timer is to reset the system.

- Change the value of a constant in the software, enabling priority inheritance.
  - When the information bus was blocked in the mutex, the meteorological thread would heritage the information bus thread priority, thus avoiding the communication thread to run for a too long time.

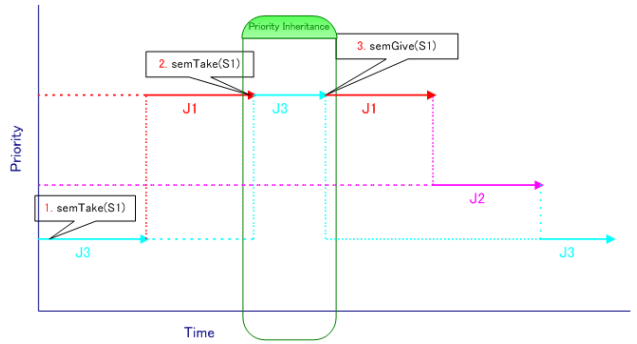


Figure: Priority inheritance example.

Ariane 5
Description

- Ariane 5 is an expendable launch system used to deliver payloads into geostationary transfer orbit or low Earth orbit.
- The rocket took a decade to be developed and required 7 billions dollars.

Ariane 5
Problem

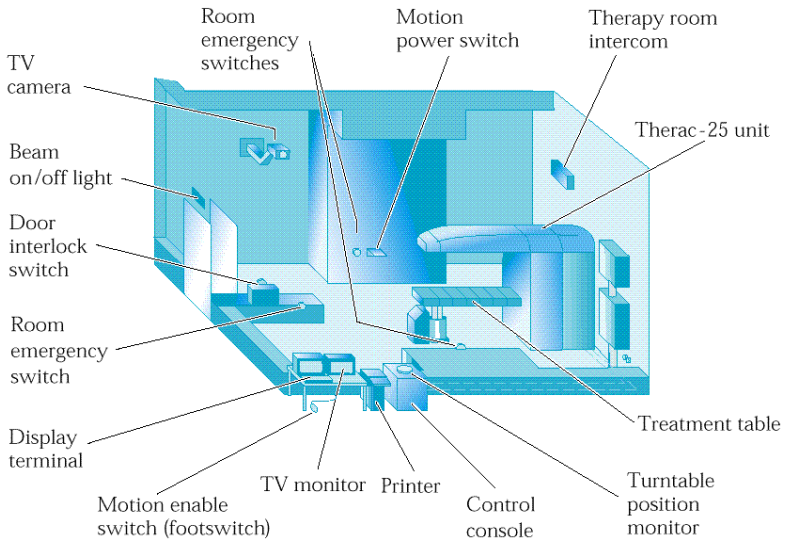
- On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure.
- Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.

[hasprev=true,hasnext=false]

- The back-up Inertial Reference System failed, followed immediately by the failure of the active Inertial Reference System.
- The failure cause was a fault in the software that calculated the horizontal velocity of the rocket.
  - The variable that stored the value was 64 bit wide (floating point) and was incorrectly changed to 16 bits (signed integer).
  - The value was bigger than 32,767 (biggest value a signed integer can represent), causing a conversion failure.
- Insufficient testing for components reused from Ariane 4 were the cause of the failure.

Therac-25
Description

- Therac-25 was a computerized radiation therapy machine produced after the Therac-20 units.
- The machine offered two modes of radiation therapy:
  - direct electron-beam therapy,
  - megavolt X-ray therapy, which delivered X-rays produced by colliding high-energy (25 MeV) electrons into a target.



Therac-25
Description

- Therac-20 employed independent protective circuits and mechanical interlocks to protect against overdose.
- Therac-25 relied more heavily on software.
- FDA approved Therac-25 as pre-market equivalence to Therac-20 (even though the safety mechanisms were moved into the software, a major change from previous version of the machine.)

Therac-25
Problem

- The machine massively overdosed patients at least six times between June 1985 and January 1987.
  - Each overdose was several times the normal therapeutic dose.
  - The overdose resulted in the patient's severe injury or even death.

Therac-25
Diagnostic

- The accidents occurred when:
  1. **High-power** electron beam was activated instead of the intended **low power** beam, and
  2. beam spreader plate **not** rotated into place.
- The machine's software did not detect that this had occurred, and therefore did not prevent the patient from receiving a potentially lethal dose of radiation.

- The failure only occurred when a particular nonstandard sequence of keystrokes was entered on the VT-100 terminal which controlled Therac-25.

```
PATIENT NAME        : TEST
TREATMENT MODE      : FIX          BEAM TYPE: X      ENERGY (MeV): 25

                                   ACTUAL          PRESCRIBED
          UNIT RATE/MINUTE           0                200
          MONITOR UNITS            50  50             200
          TIME (MIN)                0.27              1.00

GANTRY ROTATION (DEG)               0.0               0      VERIFIED
COLLIMATOR ROTATION (DEG)         359.2             359      VERIFIED
COLLIMATOR X (CM)                  14.2              14.3    VERIFIED
COLLIMATOR Y (CM)                  27.2              27.3    VERIFIED
WEDGE NUMBER                          1                 1    VERIFIED
ACCESSORY NUMBER                      0                 0    VERIFIED

DATE    : 84-OCT-26     SYSTEM   : BEAM READY     OP. MODE  : TREAT   AUTO
TIME    : 12:55: 8      TREAT    : TREAT PAUSE               X-RAY   173777
OPR ID  : T25V02-R03    REASON   : OPERATOR        COMMAND:
```

- The primary reason should be attributed to the bad software design and development practices and not explicitly to several coding errors that were found.
  - The equipment control task did not properly synchronize with the operator interface task, so that race conditions occurred if the operator changed the setup too quickly.
  - AECL had never tested the Therac-25 with the combination of software and hardware until it was assembled at the hospital.

Therac-25
Solution

- First suggested solution (proposed by the manufacturer):
  - Change in operating procedures:
    - The key used for moving the cursor back through the prescription sequence (i.e., cursor "UP" inscribed with an upward pointing arrow) must not be used for editing or any other purpose.
    - To avoid accidental use of this key, the key cap must be removed and the switch contacts fixed in the open position with electrical tape or other insulating material.
- After several years of troubleshooting, the final solution was comprised of numerous software fixes, the installation of independent, mechanical safety interlocks, and a variety of other safety related changes.

Thunder Horse
Description

- Thunder Horse is a semi-submersible deep-water platform:
  - Displacement of about 130,000 tons.
  - Largest and, reportedly, the most technologically advance deep-water platform ever built.

Thunder Horse
Problem

- In mid July 2005, Hurricane Dennis swirled in the Gulf of Mexico.
- The platform was evacuated as a precaution.
- Upon return to the platform on July 12 2005, it was found precariously listing 20 to 30 degrees.
  - The lower deck of the platform was at sea level.

Thunder Horse
Diagnostic

- Thunder Horse oilfield listing after an hurricane due to a ballast system error.

La Tosca at San Diego
Description

- The opera La Tosca (Giacomo Puccini) debuted just over one hundred years ago, at the Teatro Constanzi in Rome on January 14, 1900.
- Soon after its premiere, it became one of the most popular operas in the repertoire, and it remains so to this day.
- It was the candelabra that played a prominent role in a San Diego performance of Tosca in 1956.
  - The script called for Tosca to blow out the four candles in the candelabra before dramatically placing a candle on either side of Scarpia and a crucifix on his breast and exiting the stage.

La Tosca at San Diego
Problem

- In San Diego, the candles were electric, and the order of their going out was fixed on a computer tape along with all the rest of the lighting cues.
- The tape obeyed the stage manager's signal and snuffed the candles exactly as Tosca blew them out.
  - Except that, on this occasion, the programming was wrong and it blew them out in a different order from hers.
  - She blew to the right, the candle on the left went out, she blew the back one, the one in front went out!

Incorrect statement

- Consider the following statement, as defined in the software requirements specification.
  - The z dimension value of the vortex is the sum of its x and y dimension values.
- By an unknown reason (the cat walked on the keyboard, insomnia, keyboard malfunctioning), the programmer wrote a statement that does not correctly implements the requirement:

```
z = x - y.
```

- This programmer incurred into a mistake.

Incorrect statement
Description

- Consider the following software requirement: "The value of `y` should be a function of `x` and `y`, as defined by the expression `y=(x-z)/2`".
- The programmer, when implementing the requirement, wrote:

```
y = x - z / 2
```

Incorrect statement
Diagnostic

- The programmer wrote an incorrect data definition in the software! That's a fault!
- For `x`=10 and `z`=8, the expected output is `y`=1, but the incorrect version's output is `y`=6.

Incorrect statement

- Consider the following statement, as defined in the software requirements specification.
  - The z dimension value of the vortex is the sum of its x and y dimension values.
- By an unknown reason (the cat walked on the keyboard, insomnia, keyboard malfunctioning), the programmer wrote a statement that does not correctly implements the requirement:

```
z = x - y.
```

- This programmer incurred into a mistake, which characterized a fault.

Incorrect statement

- If such a fault is activated (executed) with `x=0`, regardless of the value of `y`, no incorrect output is produced.
  - Although the fault is activated, it does not lead to an error and no failure occurs.
- For any other value different from `x=0`, the fault activation causes an error on the variable `z`.
  - Such an error, when propagated until the product output, will result in a failure.

Physician analogy for defect taxonomy

- Consider a physician making a diagnosis for a patient.
- The patient enters the physician's office with a list of **failures** (that is, symptoms).
- The physician then must discover the **fault**, or root cause of the symptom.
- To aid in the diagnosis, the physician may order tests that look for anomalous internal conditions, such as high blood pressure, an irregular heartbeat, high levels of blood glucose, or high cholesterol.
    - In our terminology, these anomalous internal conditions correspond to **errors**.

numZero
Description

- Is there any fault in this program?

```
class numZero
{
    /**
     * If arr is null throw NullPointerException, else return the
     * number of occurrences of zero in arr
     */
    public static int numZero (int[] arr)
    {
        int count = 0;
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] == 0) {
                count++;
            }
        }
        return count;
    }
}
```

numZero
Diagnostic

- The fault in this program is that it starts looking for zeroes at index 1 instead of index 0, as is necessary for arrays in Java.
  - For example, numZero([2, 7, 0]) correctly evaluates to 1, while numZero([0, 7, 2]) incorrectly evaluates to 0.
    - In both of these cases the fault is executed.
    - Although both of these cases result in an error, only the second case results in failure.

```
for (int i = 1; i < arr.length; i++) {
    if (arr[i] == 0) {
        count++;
    }
}
return count;
```

numZero
Diagnostic

- For the first example given above, the state at the if statement on the very first iteration of the loop is (x = [2, 7, 0], count = 0, i = 1, PC = if).
- This state is in error precisely because the value of i should be zero on the first iteration.
- However, since the value of count is coincidentally correct, the error state does not propagate to the output, and hence the software does not fail.

```
for (int i = 1; i < arr.length; i++) {
    if (arr[i] == 0) {
        count++;
    }
}
return count;
```

numZero
Diagnostic

- For the second example given above, the corresponding error state is (x = [0, 7, 2], count = 0, i = 1, PC = if).
- In this case, the error propagates to the variable count and is present in the return value of the method. Hence a failure results.

```
for (int i = 1; i < arr.length; i++) {
    if (arr[i] == 0) {
        count++;
    }
}
return count;
```

Sort test cases

- Consider a sort implementation for a array of integers.
- The input of the sort application is an array, set as the argument of the application, and the result is printed to the console.
- For example, for the input array '3 1 7 4', the result would be:

```
$ sort 3 1 7 4
1 3 7 4
```

Sort test cases

- A possible set of test cases is:
  - $< [3, 1, 7, 4], [1, 3, 7, 4] >$
  - $< [1, 2, 3, 4], [1, 2, 3, 4] >$
  - $< [0, 1, 0, 2], [0, 0, 1, 2] >$
  - $< [], [] >$
  - $< [a, b, 0, 1], errormessageprintedtoconsole >$

numZero test cases
Description

- Consider the following code:

```
public static int numZero(int[] arr) {
    int count = 0;
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] == 0) {
            count++;
        }
    }
    return count;
}
```

- Some test cases that could be designed for this code are:
  - ([2, 7, 0], 1)
  - ([0, 7, 2], 1)
  - ([], 0)
  - ([0, 0, 0], 3)

Cascading test cases

- The first test case exercises a particular feature of the software and then leaves the system in a state such that the second test case can be executed.
- While testing a database consider these test cases:
  1. create a record,
  2. read the record,
  3. update the record,
  4. read the record,
  5. delete the record, and
  6. read the deleted record.

Kiddie oracle example

- Consider the factorial of a number.
- For a given input number, if the result looks correct, then it is correct.
- For small input numbers, like 2, 3, it is easy to verify that:
  - $2! = 2$
  - $3! = 6$
- But what about bigger numbers?
  - $11! = 39916800$
  - $21! = 51090942171709440000$
- Actually, $21! = 51090\mathbf{9}42171709440000$ (instead of $51090\mathbf{8}42171709440000$)

Regression test suite oracle example

- Mozilla runs build computers that continually build the latest source code.
- A tool, called Thinderbox, collects the test results and compare to the previous version of the product under testing.
- It uses tables to describe status of the source tree for the platform, product, and code branch:
  - 1
  - The green bar means the latest code compiles and passes the tests that are run on the box.
  - Red means the build failed during compilation.
  - Orange means the binary was built successfully, but failed some of the tests.
  - Yellow means the build and testing are in progress.
- Tinderbox for Firefox:
  - `http://tinderbox.mozilla.org/showbuilds.cgi?tree=Firefox`

Java test suite oracle

- Any Java implementation (Java Virtual Machine and standard libraries) can be tested against the Java Compatibility Test Tools (Java CTT), available at `http://www.jcp.org/en/resources/tdk`.
- However, the Java CTT is not freely available (it requires you to be JCP member).
- As an alternative, you can use the Mauve, which is available at `http://sources.redhat.com/mauve/`.

Pentium FDIV bug
Description

- In 1994, Intel introduced its Pentium microprocessor, and a few months later, a mathematician found that the chip gave incorrect answers to certain floating-point division calculations.
  - The chip was slightly inaccurate for a few pairs of numbers.
- Example: A number multiplied and then divided by the same number should result in the original number)
  - Expected result: $4195835 * 3145727/3145727 = 4195835$
  - Flawed Pentium result: $4195835 * 3145727/3145727 = 4195579$

Pentium FDIV bug
Diagnostic

- The fault was the omission of five entries in a table of 1,066 values (part of the chip's circuitry) used by a division algorithm.
  - The five entries should have contained the constant $+2$, but the entries were not initialized and contained zero instead.

Pentium FDIV bug
Solution

- The mistake that caused the fault was very difficult to find during system testing.
  - Intel claimed to have run millions of test cases using this table.
- The table entries were left empty because a loop termination condition was incorrect
  - The loop stopped storing numbers before it was finished.
- This turns out to be a very simple fault to find during unit testing
  - Analysis showed that almost any unit level coverage criterion would have found this multi-million dollar mistake.

Mars Climate Orbiter

- Mars Climate Orbiter was a NASA spacecraft that would study the Martian weather, climate, water and carbon dioxide budget.
- It was intended to enter orbit at an altitude of 140.5 - 150 km above Mars.
- However, a navigation error caused the spacecraft to reach as low as 57 km.
- The spacecraft was destroyed by atmospheric stresses and friction at this low altitude.

Mars Climate Orbiter
Diagnostic

- The navigation error arose because of a software programming error.
  - The thruster control software did not properly interpret the data fed to it.
- The modules of the thruster control were created by separate software groups.
- One module computed thruster data in English units and forwarded the data to a module that expected data in the International System units (meters).
- This is a very typical integration fault (but in this case enormously expensive, both in terms of money and prestige).

Ghost Train
Description

- In 1995, failures were registered by the sensors in the rails situated in tunnels within England channels.
  - The trains stopped due to suspicion that another train was in the rail.
- The cause was that the salty water fog was confusing the sensors.
  - The system requirements probably didn't considered the salty water fog as a possible event.
  - Adequate system testing and validation could have detected the problem.

Test criterion example

- Suppose we are given the enviable task of testing bags of jelly beans. We need to come up with ways to sample from the bags.
- Suppose these jelly beans have the following six flavors and come in four colors: Lemon (colored Yellow), Pistachio (Green), Cantaloupe (Orange), Pear (White), Tangerine (also Orange), and Apricot (also Yellow).
- A simple approach to testing might be to test one jelly bean of each flavor. Then we have six test requirements, one for each flavor.
- We satisfy the test requirement "Lemon" by selecting and, of course, tasting a Lemon jelly bean from a bag of jelly beans.

Test criterion example

- The “flavor criterion” yields a simple strategy for selecting jelly beans.
- In this case, the set of test requirements, TR, can be formally written out as $TR = flavor = Lemon, flavor = Pistachio, flavor = Cantaloupe, flavo$

Exhaustive testing
Blech example

- Consider the function `blech`, implemented as follows:

```
int blech(int j) {
    j = j - 1; // should be j = j + 1;
    j = j / 30000;
    return j;
}
```

- Input domain:
  - Consider an integer type of 16 bits (2 bytes).
  - The lowest possible input value is -32,768 and the highest is 32,767.
  - Thus there are 65,536 possible inputs into this small software.
- Which test cases can detect the fault?

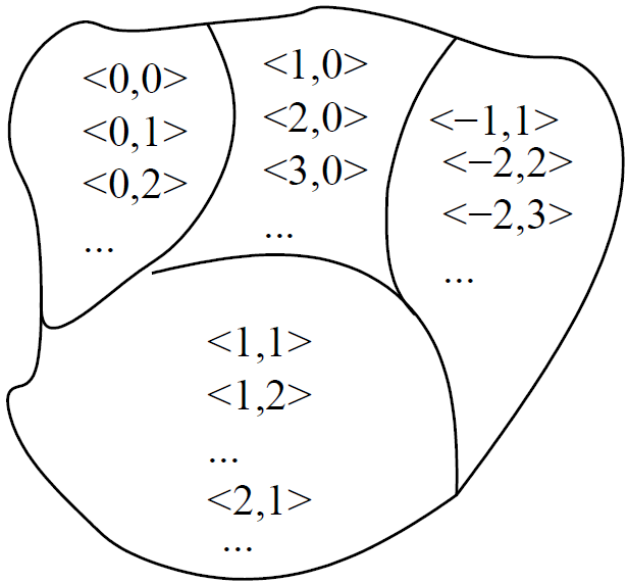
- Only four out of the possible 65,536 input values will find this fault:

Test Cases Input(j)	Expected Output	Actual Result
-30000	0	-1
-29999	0	-1
30000	1	0
29999	1	0

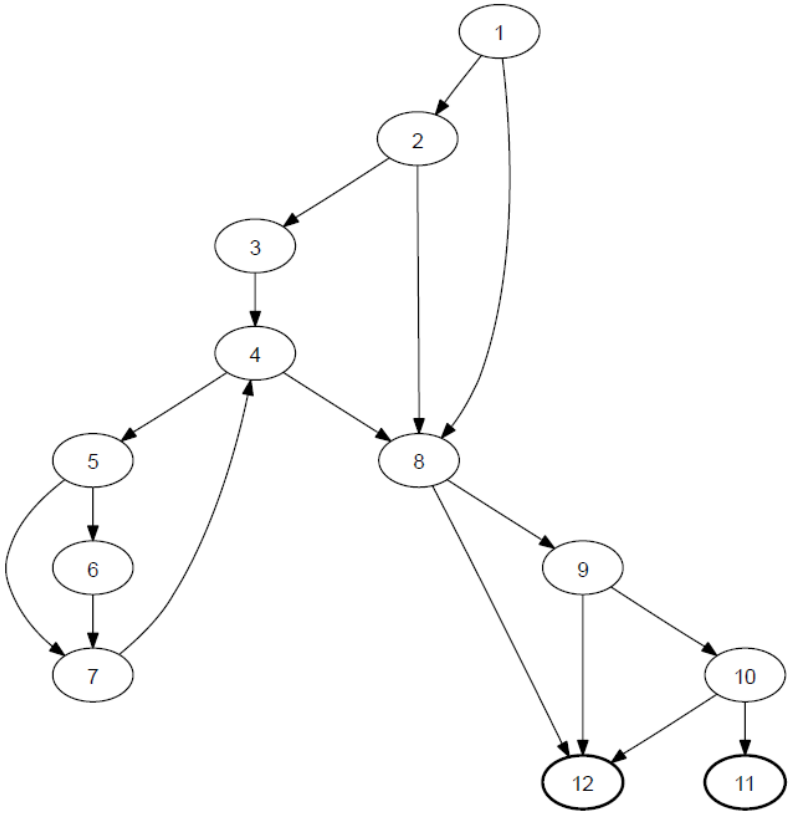
```
int blech(int j) {
    j = j - 1; // should be j = j + 1;
    j = j / 30000;
    return j;
}
```

Functional testing example

Consider the function x^y, where x is an integer and y is a non-negative integer. The input domain is: for every tuple (x,y), with $y >= 0$. The input domain can be partitioned as follows:



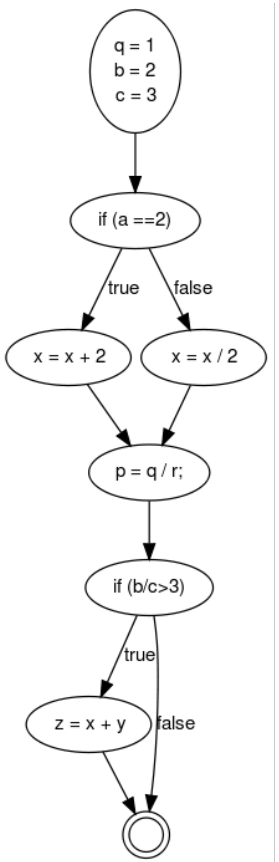
Identifier control-flow graph



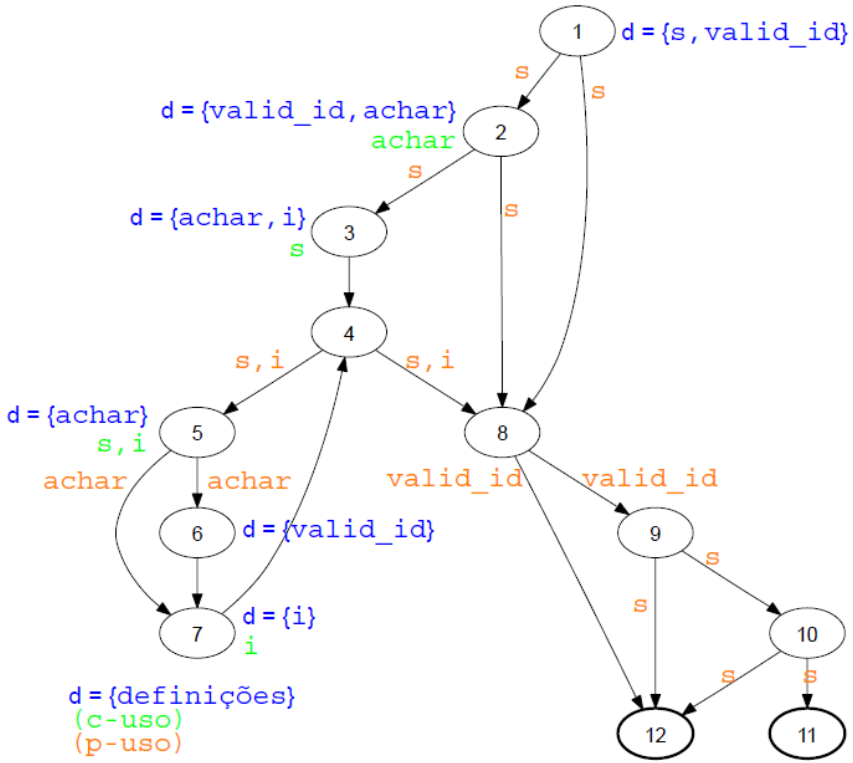
Program graph


```
q = 1;
b = 2;
c = 3;
if (a ==2) {
    x = x + 2;
}  else {
    x = x / 2;
}
p = q / r;
if (b/c>3) {
    z = x + y;
}
```

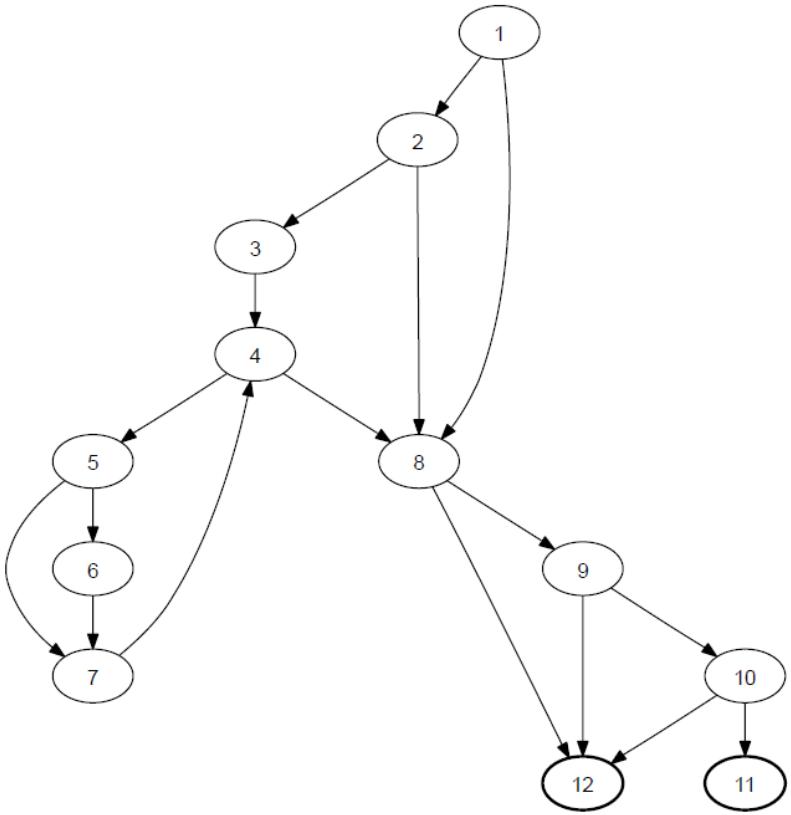
Program graph



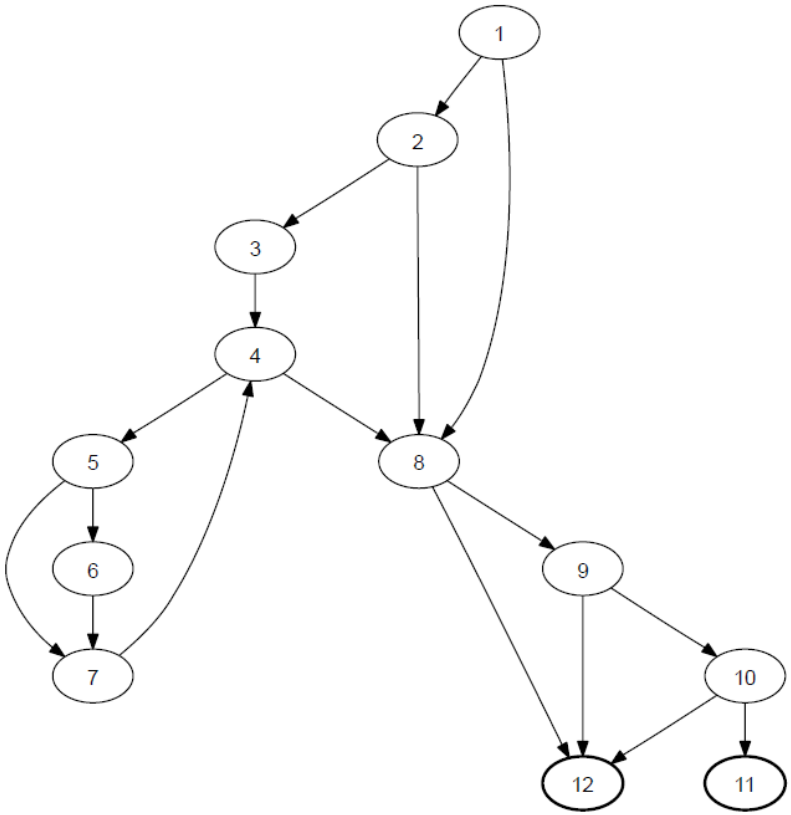
Identifier definition-use graph



The following path is infeasible: (5, 6, 7, 4, 8, 9, 10, 11)

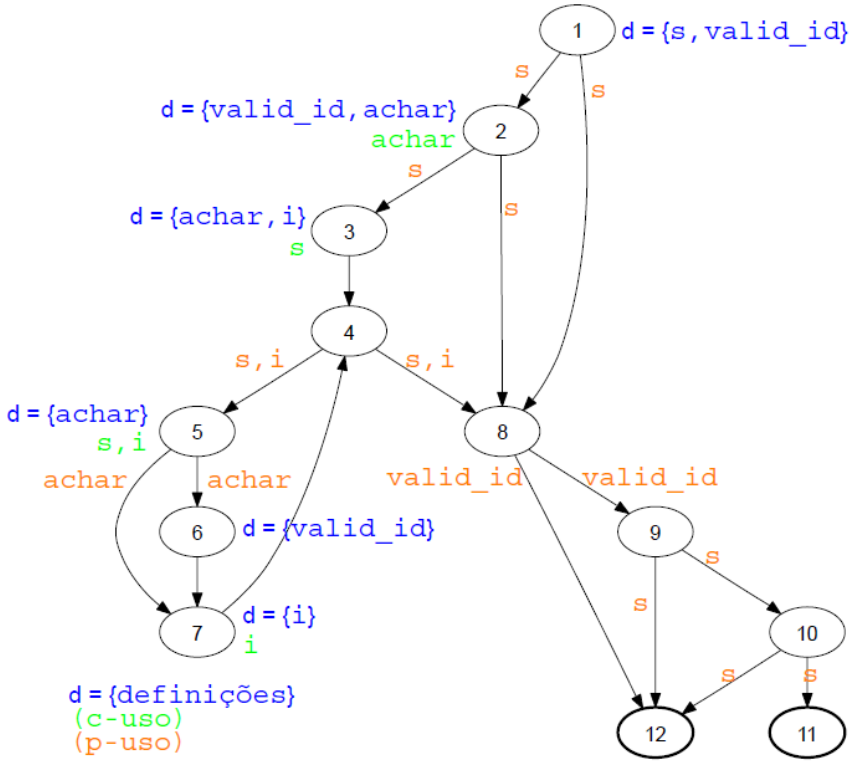


The following path is complete: (1, 2, 3, 4, 5, 6, 4, 8, 12)



Definition-clear paths example

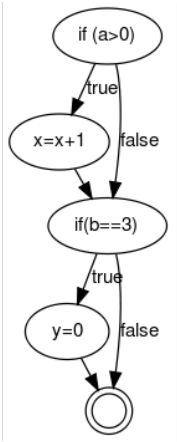
Consider the following graph:



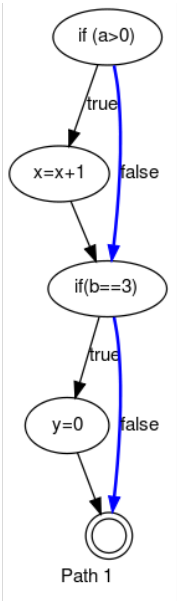
Definition-clear paths example

- Path (1,8,12) is a definition-clear path with respect to `valid_id` defined at node 1.
- Path (1,2,8,12) is not a definition-clear path with respect to `valid_id` defined at node 1, because `valid_id` is redefined at node 2.

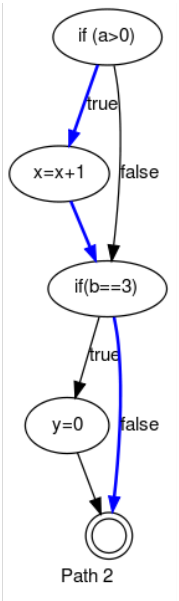
All-nodes



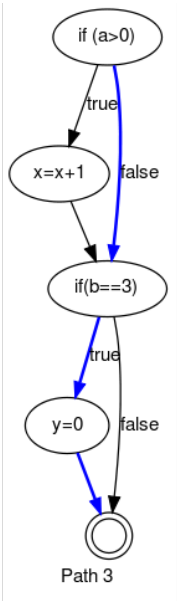
All-nodes
Path 1



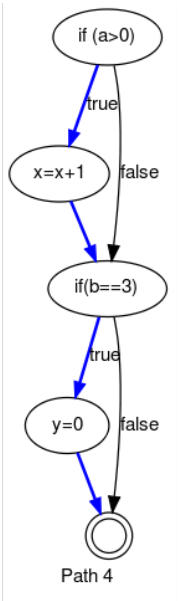
All-nodes
Path 2



All-nodes
Path 3



All-nodes
Path 4



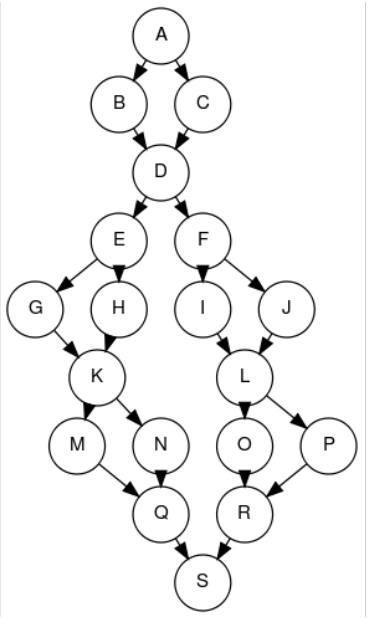
All-paths infeasibility example

The block below executes `doSomethingWith()` one billion times (1000 x 1000 x 1000).

```
for (i =1; i <=1000; i ++)
for (j =1; j <=1000; j ++)
for ( k =1; k <=1000; k++)
doSomethingWith(i, j, k);
```

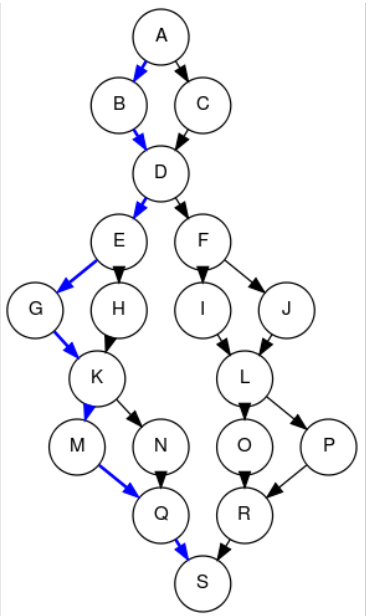
McCabe example

Consider the graph below:

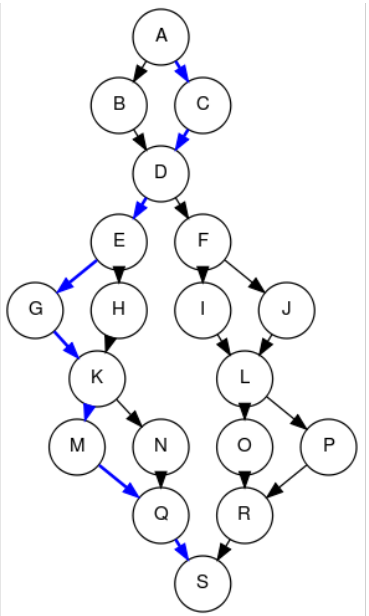


The cyclomatic complexity of the graph is 7. So, seven test requirements, thus seven complete paths, must be devised for the graph.

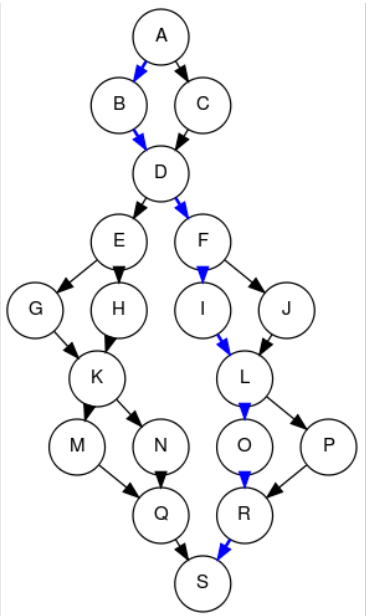
McCabe example
Path 1



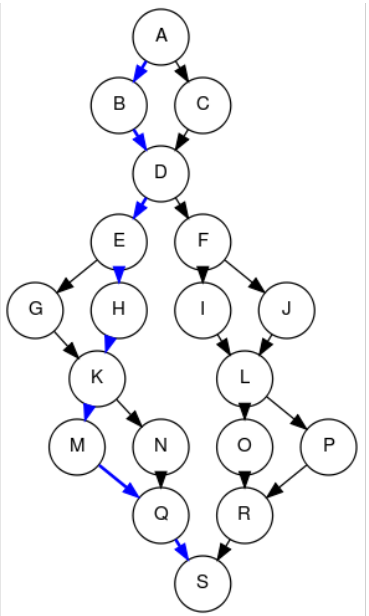
McCabe example
Path 2



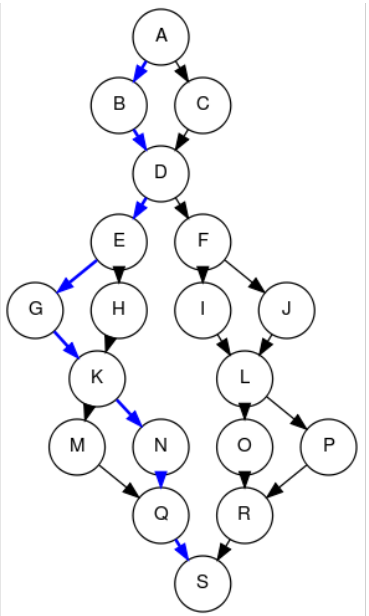
McCabe example
Path 3



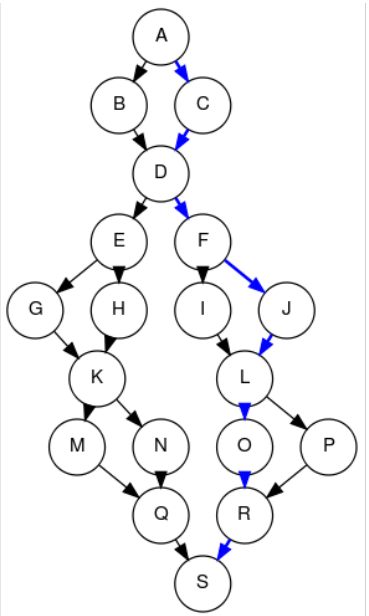
McCabe example
Path 4



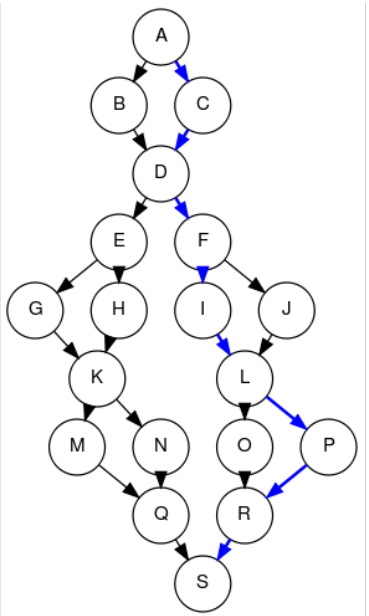
McCabe example
Path 5



McCabe example
Path 6



McCabe example
Path 7

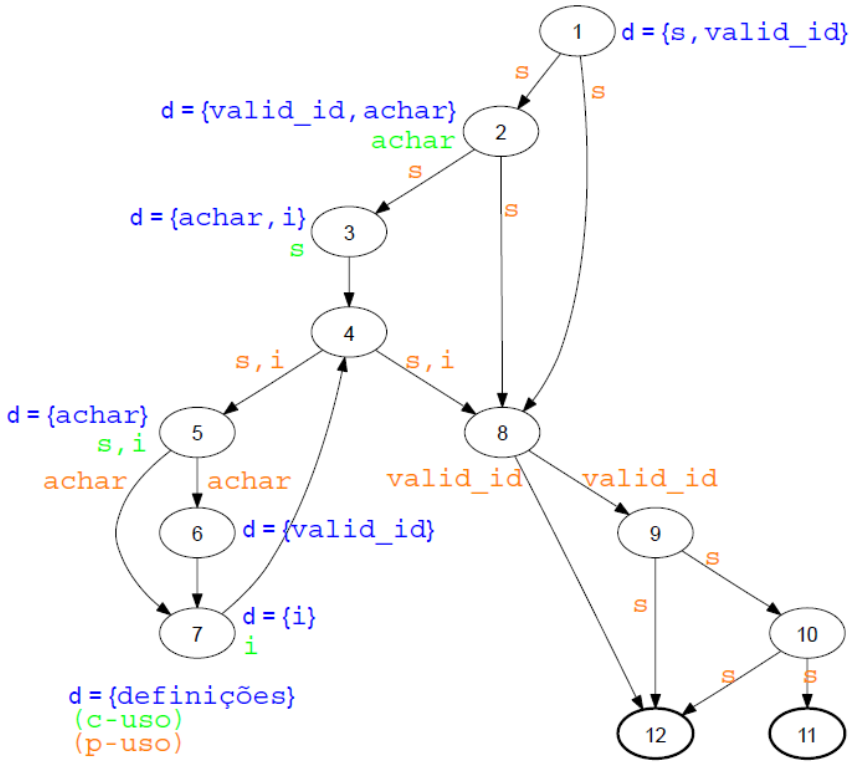


All-Uses for Identifier

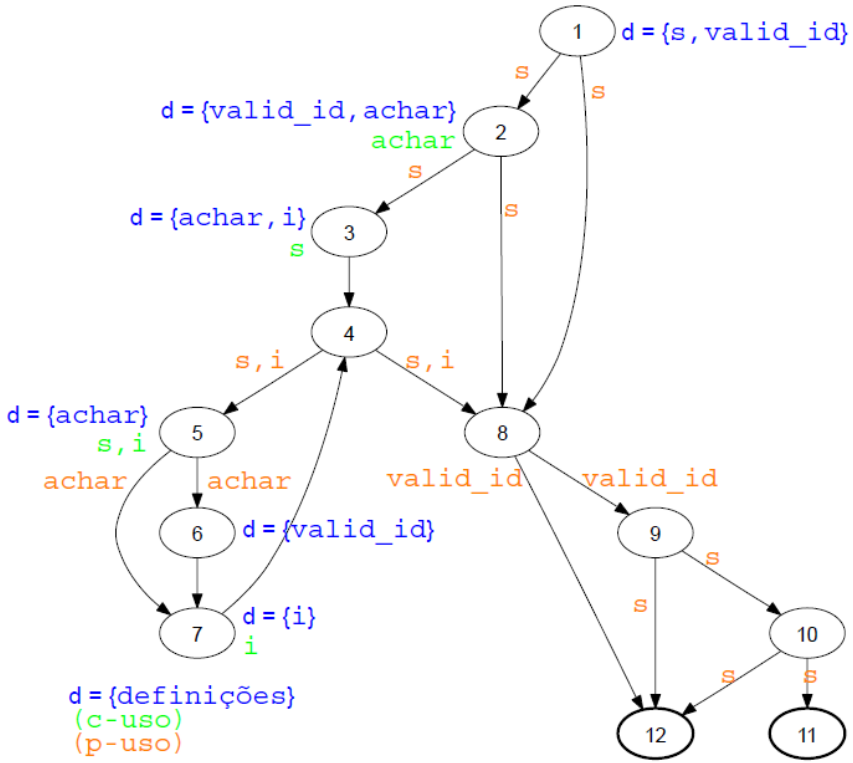
Some test requirements for Identifier considering the All-uses criterion:

- (length, 1, 2)
- (achar , 1, 3)
- (valid id, 1, (1, 3))

All-Uses for Identifier



All-Potential-Uses for Identifier



All-Potential-Uses for Identifier

Some test requirements for Identifier considering the All-Pot-uses criterion:

- (length, 1, 2)
- (achar , 1, 3)
- (valid id, 1, (1, 3))
- (length, 2, (8, 10))
- (achar , 3, (8, 10))

JaBUTi new project

JaBUTi import

1. Consider a program P, which belongs to a critical system.
2. The correct operation of the system depends upon the correctness of the program P.
3. Thus, the software tester will test P as much as possible, using several test criteria and evaluating the adequacy of the developed test cases.
4. Initially, a test set $C_1 - adequate$ is created. Now, a question is arisen: **Given a test set $C_1 - adequate$ and a test criterion C_2, is is possible to improve the current test set?**
5. Such issue is recurrent when deciding whether a program has been sufficiently tested.
6. Experimental studies can evaluate test criteria properties, providing evidences that aids the software tester decision making regarding such issues.

Subsume relation