Doxygen Flavor for Structure 101g

By Marcio Marchini (marcio.marchini@gmail.com)

2012/01/05

1) What is the Doxygen Flavor for Structure101g?

This is a sort of a plugin for Structure 101g (called a flavor). It allows you to get XML files produced by Doxygen (from a software project's source code) and load that representation into Structure 101g. This flavor will convert the Doxygen XMLs into a single XML file that can be loaded into Structure 101g. The conversion is done from a command-line tool, *Doxygen2Structure101g*. It's as simple as that. But, as they say, the devil is in the datails...

2) Motivation to Create the Doxygen Flavor for Structure101g

This flavor was created due to a need to look at large legacy C++ code bases, and figure out their overall architecture and plan strategies to refactor & improve their designs. More specifically, we were interested in DSM support at an affordable price (think budgets for small teams in small companies or companies in BRIC countries as opposed to North America). Both Lattix/LDM and Structure101/C++ are excellent tools, but for a different league, out of reach of the solo developer or of small start-ups in BRIC countries and similar.

3) Languages Supported

In theory this flavor should support all the languages that Doxygen supports. In practice, we have been focusing on C/C++, Java, and .NET. We run Doxygen on Open Source projects in these languages and make sure we are properly parsing all tags used by Doxygen. If you hit upon any limitation in our converter, please contact us and we will make sure it properly parses whatever it may be that we are missing.

As for Python, we ran it quickly on the source code for TRAC and some dependencies are shown, but we aren't sure if Doxygen can capture them all. Feel free to contact us if you are interested in Python analysis and can help Doxygen improve its Python parsing capabilities.

The same goes for PHP. Many dependencies are not captured by Doxygen's fuzzy parser, unfortunately.

4) Limitations of The Flavor

First and foremost, this flavor is limited to the same precision as you get from Doxygen itself. Generate HML output from Doxygen and make sure that Doxygen itself is correctly picking up your dependencies and elements. Keep in mind that Doxygen is a sort of a fuzzy parser and not a real compiler. It will miss things, it will mix things, and it will drop things. An interesting example we had was a project that was using SQLite and also PostgreSQL. Both libraries have two completely different structs with the same name (Trigger). Doxygen's output generated a single struct that was a mix of the two. Needless to say this caused all sorts of false dependencies. For this reason alone we have added the capability of dropping particular dependencies in your dependency graph. It allows you to override Doxygen's limitations when you hit them (and believe me, you will hit them for any real project). That's why we added *-blacklist*.You can also drop entities or dependency types, allowing you to keep just file includes, for example. This will generate a fairly accurate initial view of the C++ system, for example.

5) Why Three Flavors, and Not Just One?

This flavor started as a single one, *com.sglebs.doxygen*. But soon we hit upon an interesting limitation: in Structure 101g, a sub-module (compound in Doxygen parlance) can only "belong to" a single parent, via nesting of XML tags. But in Doxygen's representation a class belongs to both a file (which in turn belongs to a directory) and to a namespace (think a C++ namespace or a Java package) which may span across multiple directories. When generating the output, we had to choose a single parent container for a class – either the namespace or the file. But which one? Since we did not want to hardcode the decision in *Doxygen2Structure101g*, we made it configurable via a command-line parameter: *-rootOrder*. This is a comma-separated list of regexes (CSV of regexes) which defines the order of root containers to traverse when generating the output XML. Here one can give priority to a namespace view or to a filesystem view of the project. For example, passing *namespace,dir,.** means that first the converter will give preference to output namespaces and classes they contain, then directories and elements they contain, and only then "the rest" (.* is a regex that matches anything). Therefore, if a Class belongs to both a file and to a namespace (in Doxygen's model), we will put it inside the namespace only (in the Structure 101g model that we generate). On the other hand, passing *dir,namespace,.** will first generate directories and the elements they contain (files etc), and only then namespaces, and then the rest (.*). This means that if a Class belongs to both a file and to a namespace, it will appear as belonging only to the file in Structure 101g. In other words, the regex is a way to drive the mechanism that chooses a single parent/container for elements with multiple parents/containers, based on which one appears first in the CSV of regexes. The first parent to dump/traverse a child, becomes its sole parent in the Structure 101g's model.

Although this solution was fairly simple and good, it still generated some bogus circular dependencies. There were references from an element in a file pointing into an element in a namespace, and vice-versa. Also, the metadata.xml file in a Doxygen 101g pretty much defines what sorts of elements can be containers or not. We also needed custom

metadata.xml files for the different views of a system (filesystem-based and namesace-based). This is why the 2 custom flavors were added: *com.sglebs.doxygen.filesystem* and *com.sglebs.doxygen.namespace*. In practice, you should use one of the newer two. We still leave the original *com.sglebs.doxygen* flavor in for investigation purposes in odd cases.

5.1) com.sglebs.doxygen.filesystem

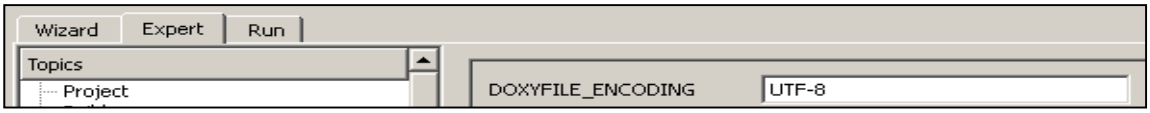
Use this flavor when you want to look at a system using a filesystem-based view. Directories represent modules, and the files are containers for classes. Lots of C and C++ projects fall into this category, as well as lots of PHP projects.

5.2) com.sglebs.doxygen.namespace

Use this flavor when you look at a system/language that relies heavily on namespaces. Examples are Java (packages are the namespaces) and C#. Although C++ also has namespaces, they are not used as THE mechanism for modularity and *componentization*. We have found the namespace perspective mostly useless for C++, but you may find it useful for some specific C++ analysis.

6) How To Generate Doxygen XML Files for this Flavor

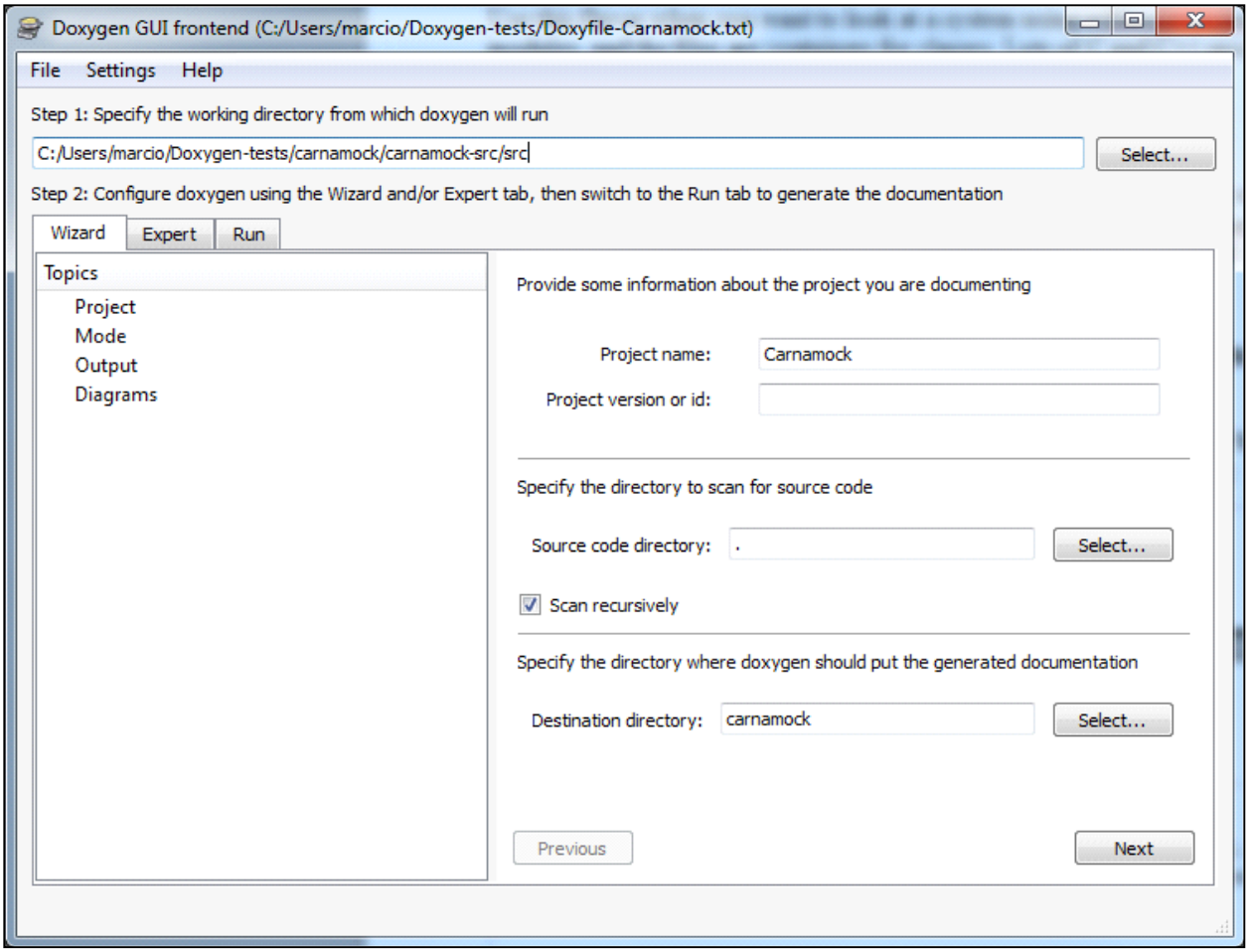
First you should download and then run Doxywizard on your sources. You should make sure you run a version with proper support for UTF-8. We recommend running 1.7 or above. Proper support for UTF-8 can be seen in Expert, on the Project item on the left. When selected, you should see DOXYFILE_ENCODING as UTF-8, as in the screenshot below:



We have identified a few extra “tricks” that will make you produce a better output for Structure 101g. The are:

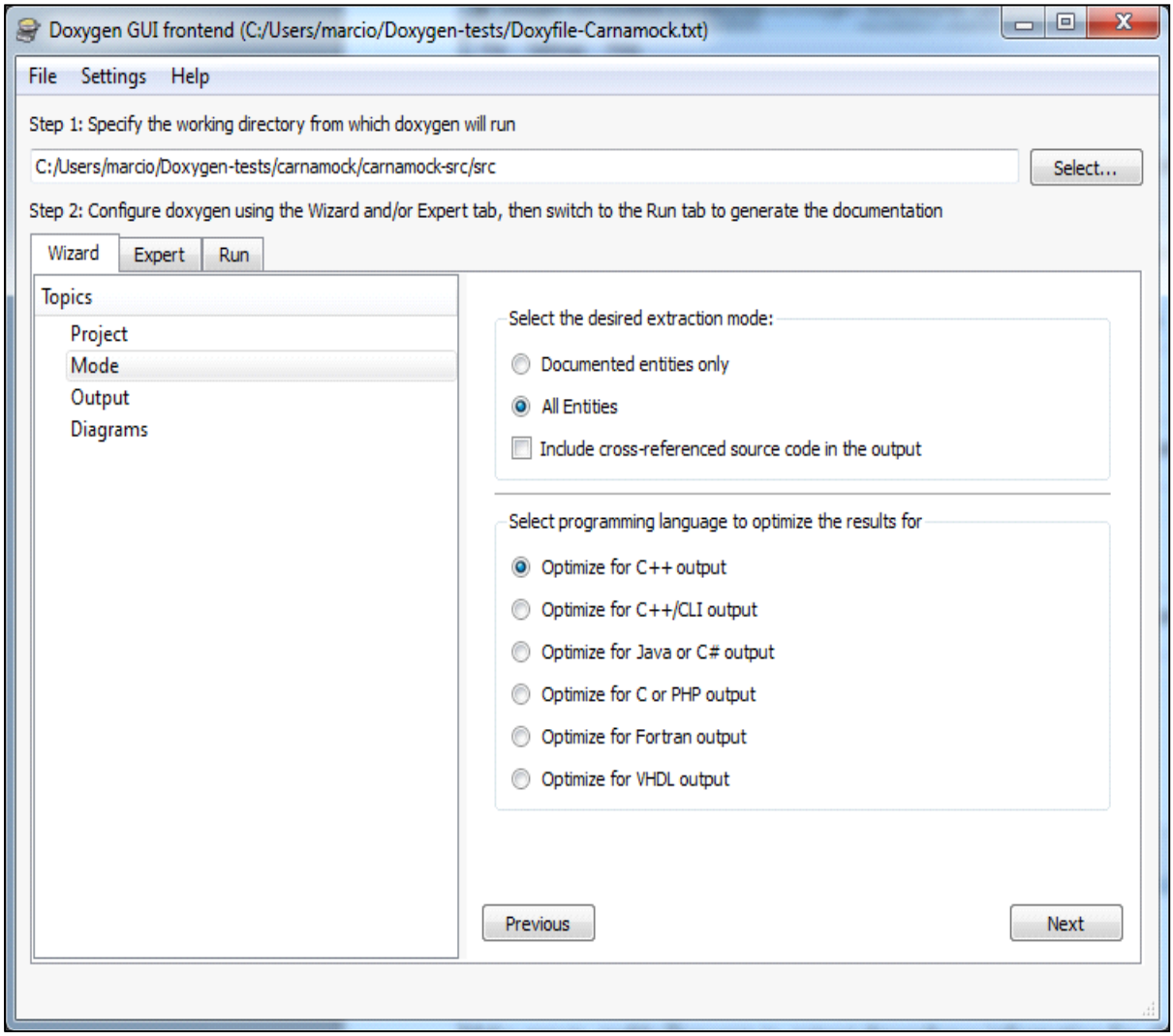
6.1) Generate output with relative paths

Set the working directory to be the directory where the source files are, and set the source file directory to be “.”. Also, enable the “scan recursively” flag. The screenshot below shows an example, running it on carnamock.



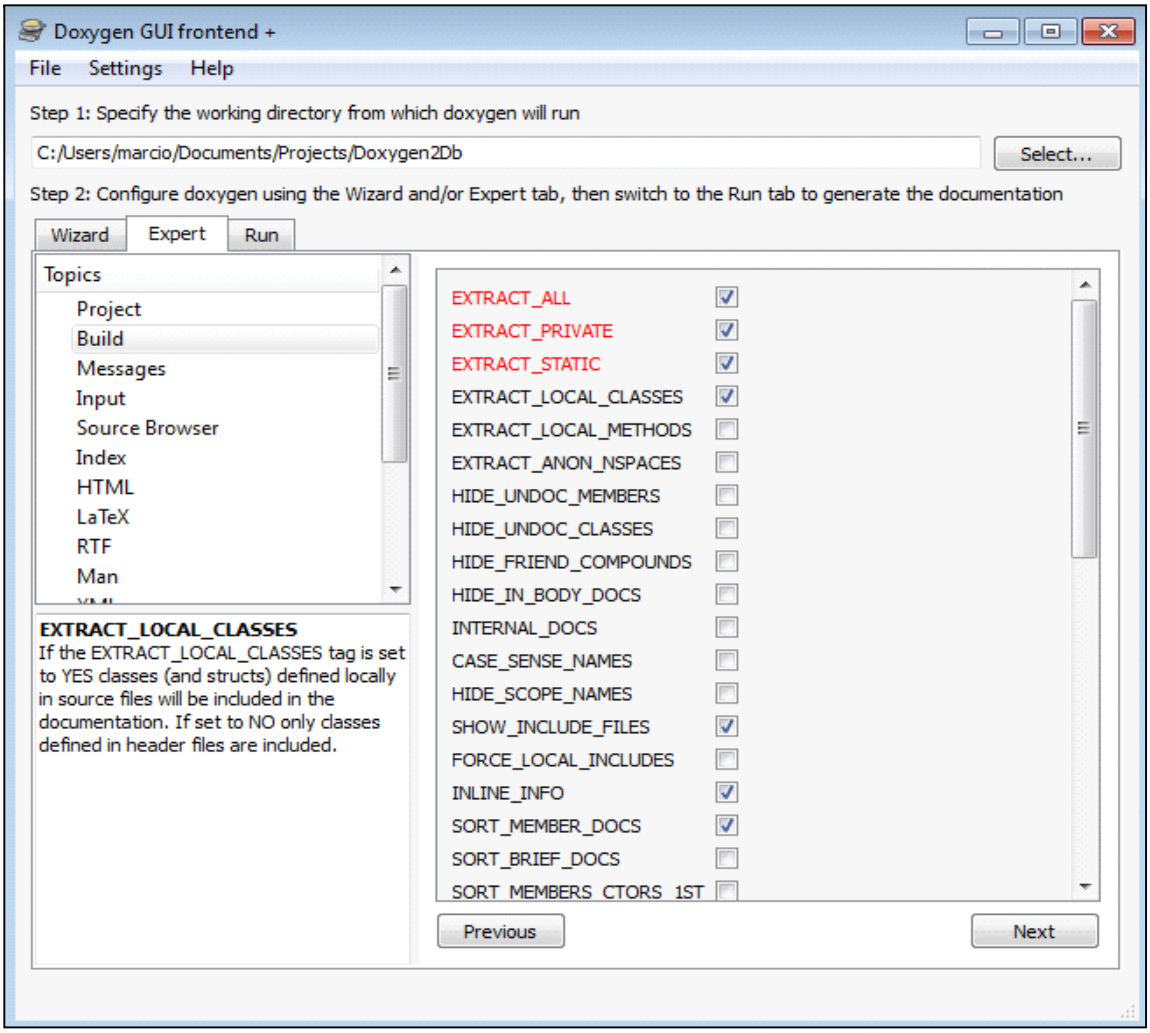
6.2) All Entities, not just Documented Entities

Make sure to enable Doxygen to extract dependency information for all elements in your source code, and not just the elements that have been documented with Doxygen-supported comments. The screenshot below shows how.



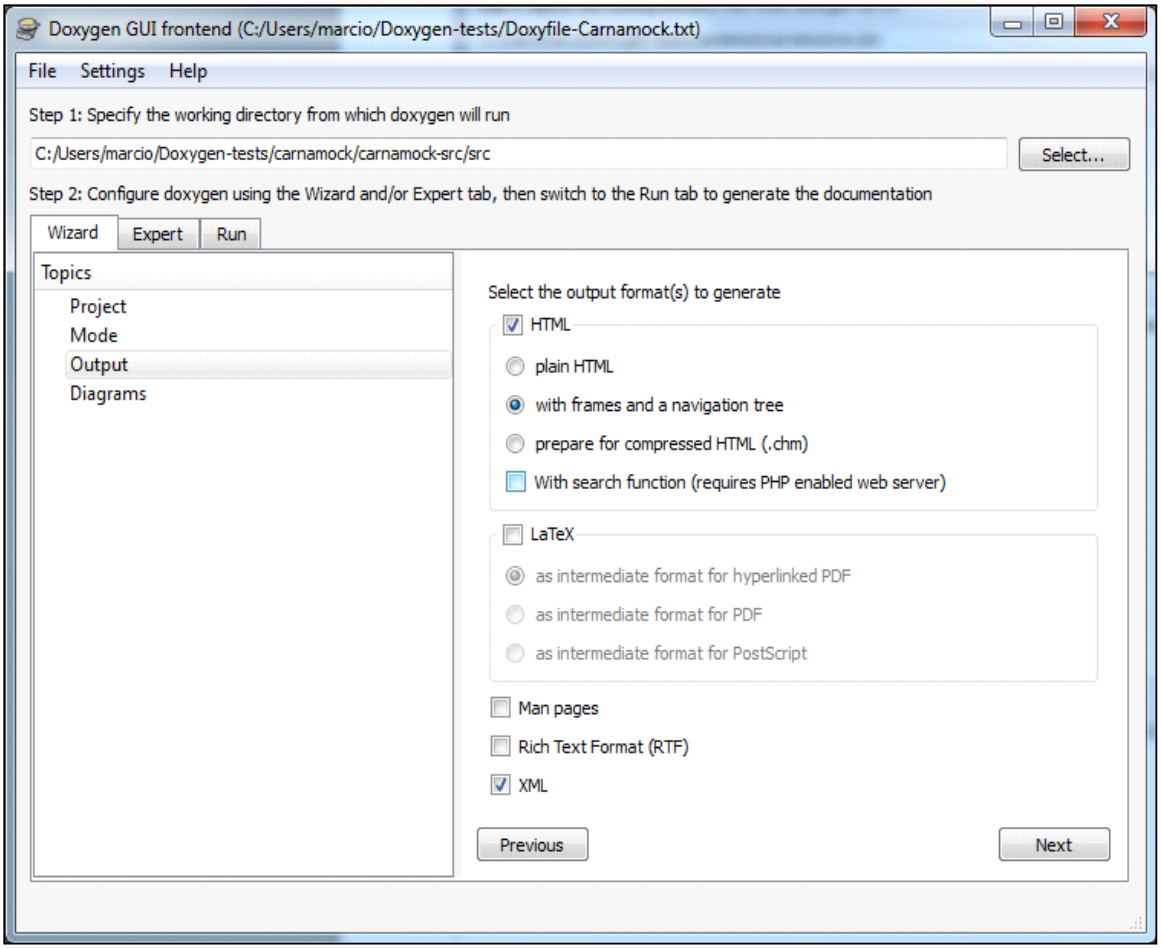
Extra Dependencies

Note that for even more advanced, in-depth dependency capturing, you should also enable Doxygen's EXTRACT_PRIVATE and EXTRACT_STATIC in the Expert tab. This can be seen in red in the next screenshot.

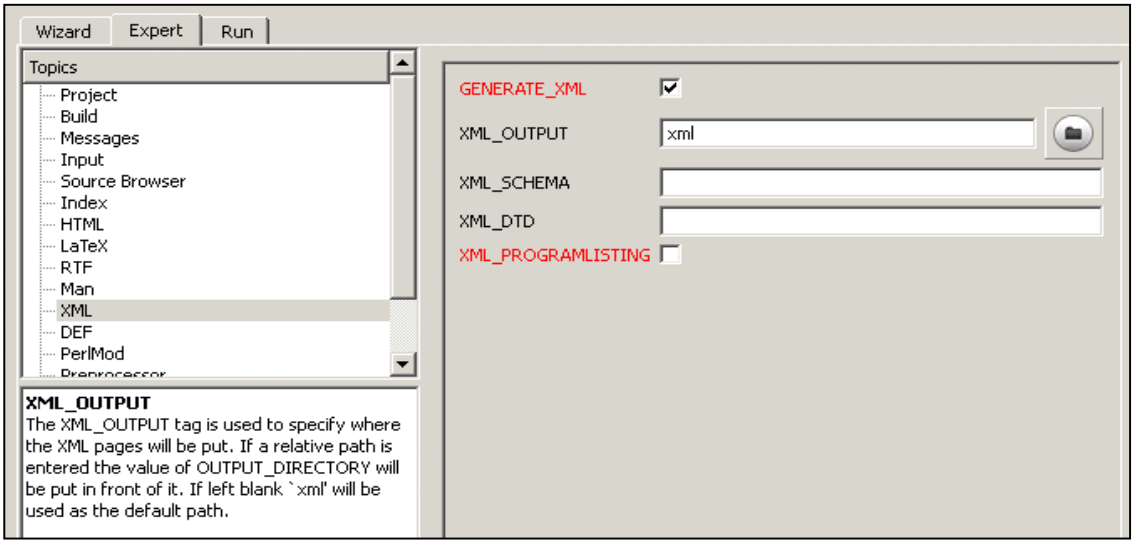


6.3) XML output (HTML is not needed, but useful for sanity checking)

You need to generate XML output. The other formats are not needed. In cases where you want to double check if Doxygen is generating bogus dependencies, you should check the HTML output. The screenshot below shows how to generate the XMLs (and also enables the optional HTML output).

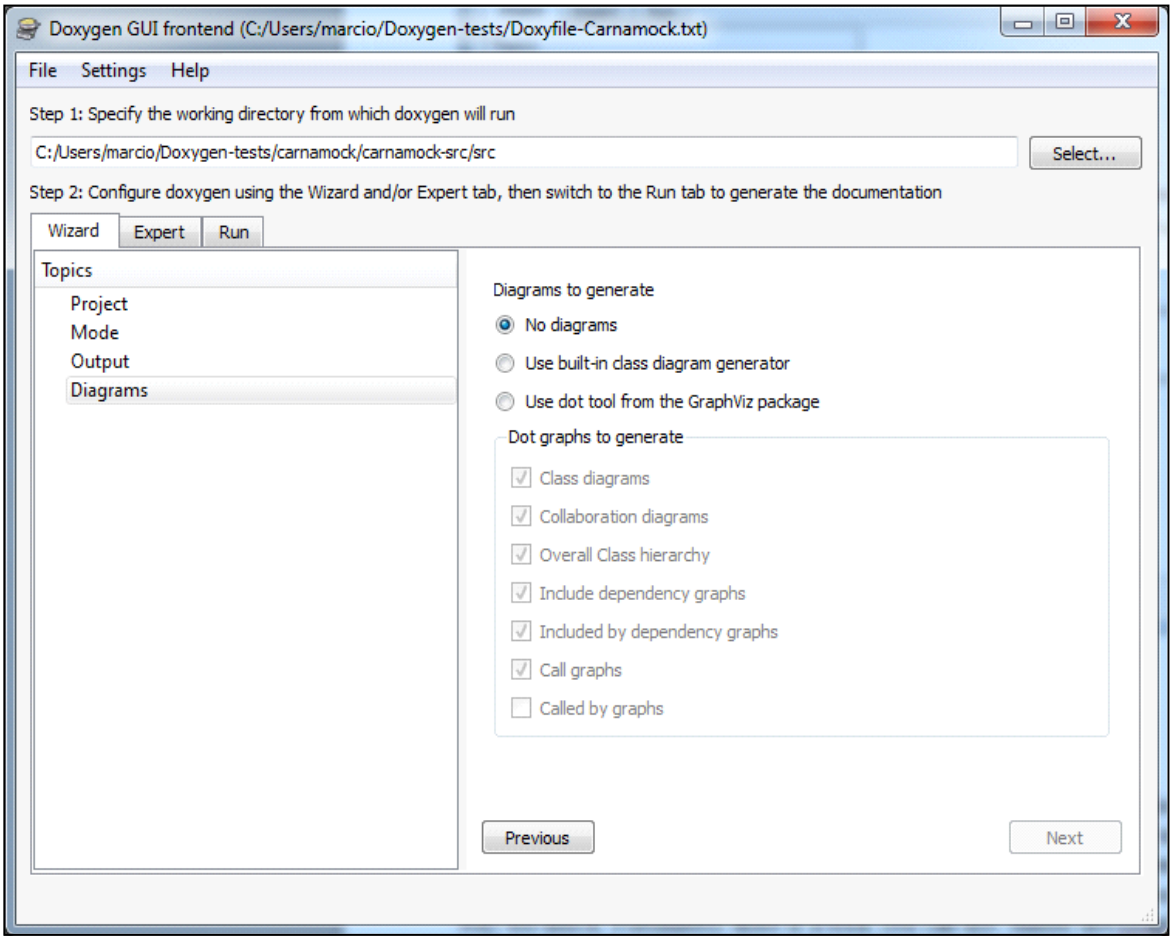


Also, make sure you exclude Program Listing. Sometimes the source code will have binary characters (HEX FF) in the markup, making the XML Parser trip and we end up losing important semantic information. To avoid hours of chasing a bogus output, please disable program listing as below (Expert Tab, under XML):



6.4) You want Call Graphs, etc

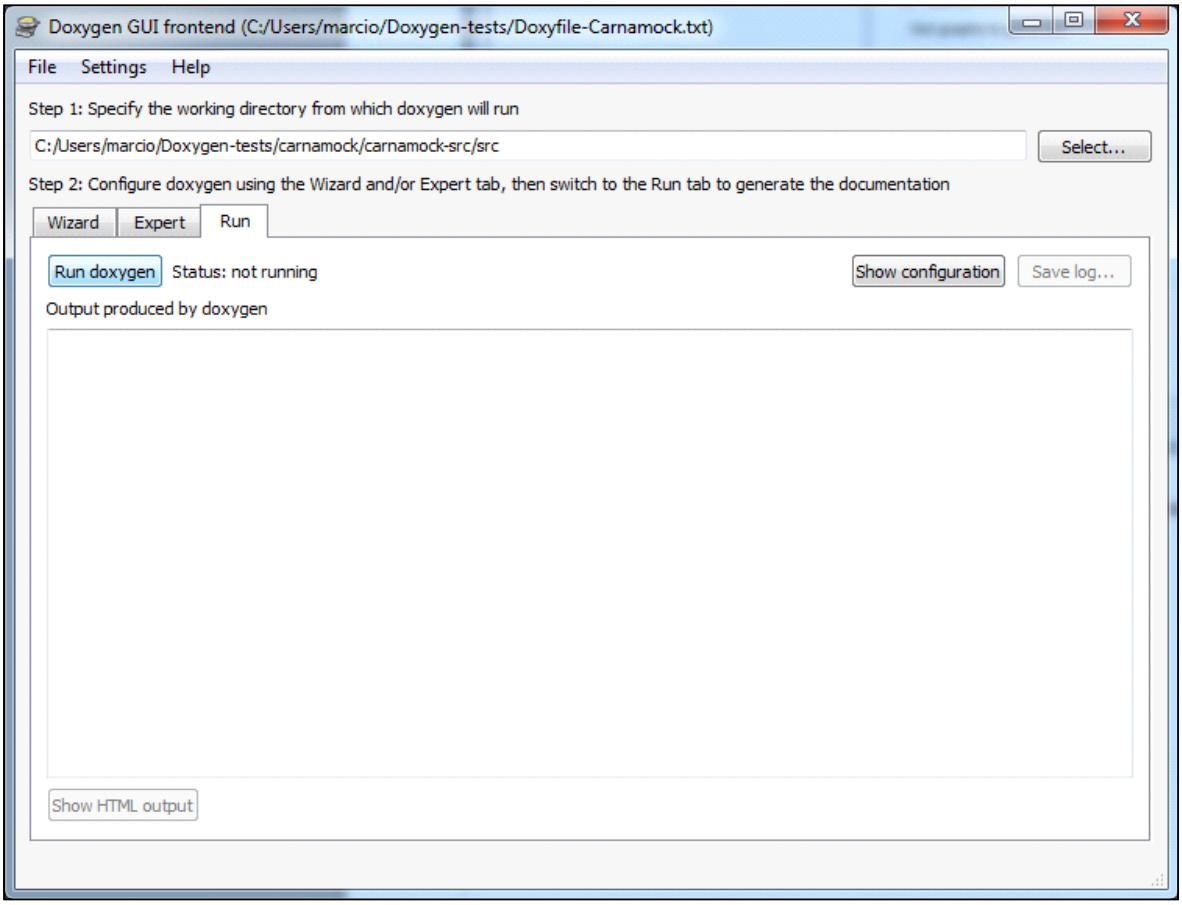
The bodies of functions and methods will call (depend on) other functions, methods and classes. This dependency only appears in the output if you explicitly ask Doxygen to generate these dependencies. Unfortunately Doxywizard only allows you to generate these extra dependency tags if you enable the use of Graphviz's dot tool. The problem is that using Graphviz will slow down the Doxygen XML generation way too much. Fortunately there is a trick you can use: enable dot/Graphviz, set the flags, and disable it again. Doxygen will still generate the dependency tags. The screenshot below shows the trick after the fact.



Note how all but the “called by graph” are checked. This is how you want it.

6.5) Ready to Run

Now you are ready to click the “Run” tab and click the “Run doxygen” button, as seen below.



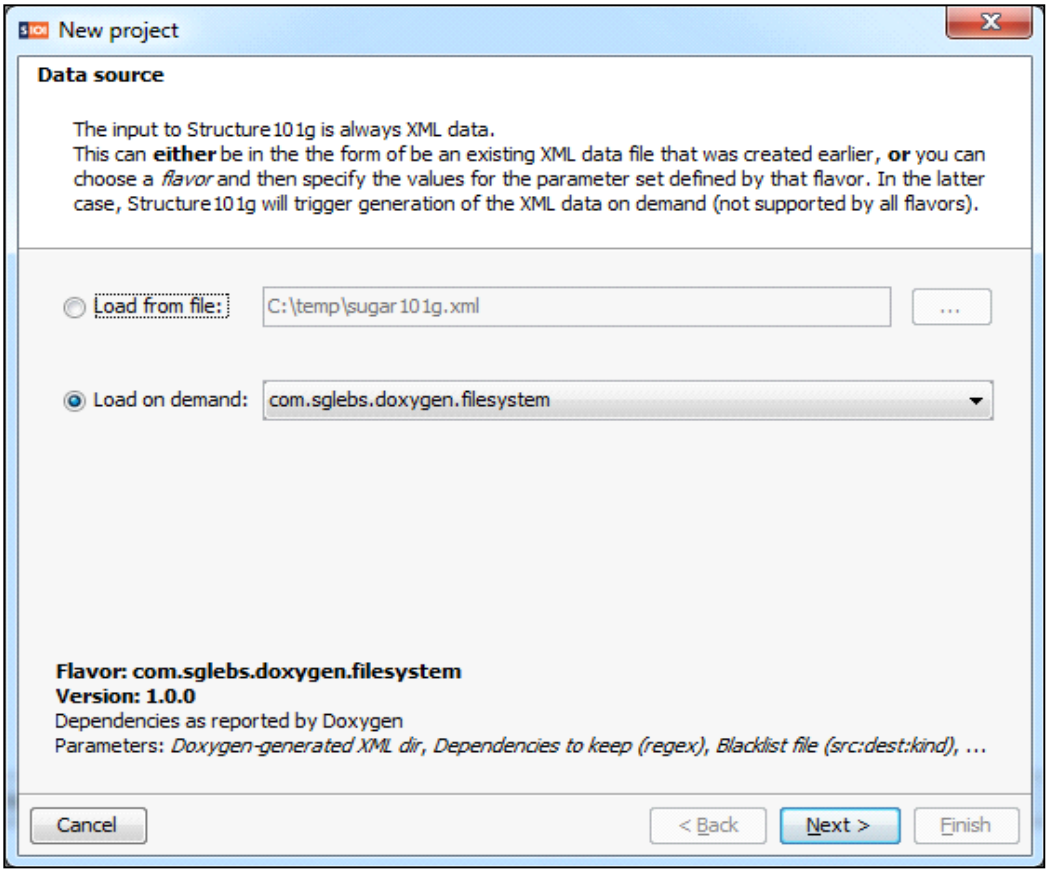
Note that you can save a Doxyfile from the Doxywizard and use that for subsequent runs. You can also run Doxygen from the command-line using this Doxyfile as parameter, possibly from a Continuous Build System like Hudson etc.

7) Running the Flavor(s) on the XMLs from Doxygen

There are 2 ways to run the converter – from the Structure 101g's GUI or from the command-line. We recommend that you always run from the GUI, as running from the command-line is fairly advanced.

7.1) Running the com.sglebs.doxygen.filesystem Flavor from the GUI

Initially you want to invoke File->New in Structure 101g, and choose “Load on demand:” as in the screenshot below:

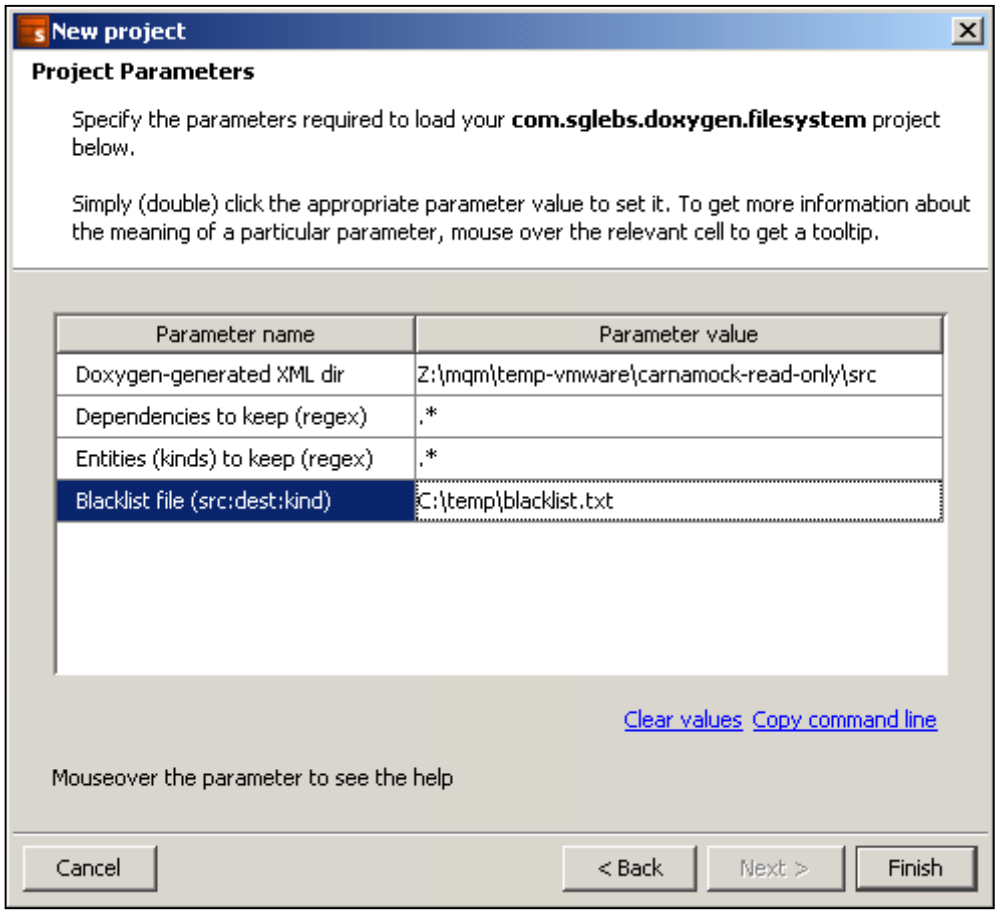


By clicking next, you will be able to set some input values on the next screen. These are:

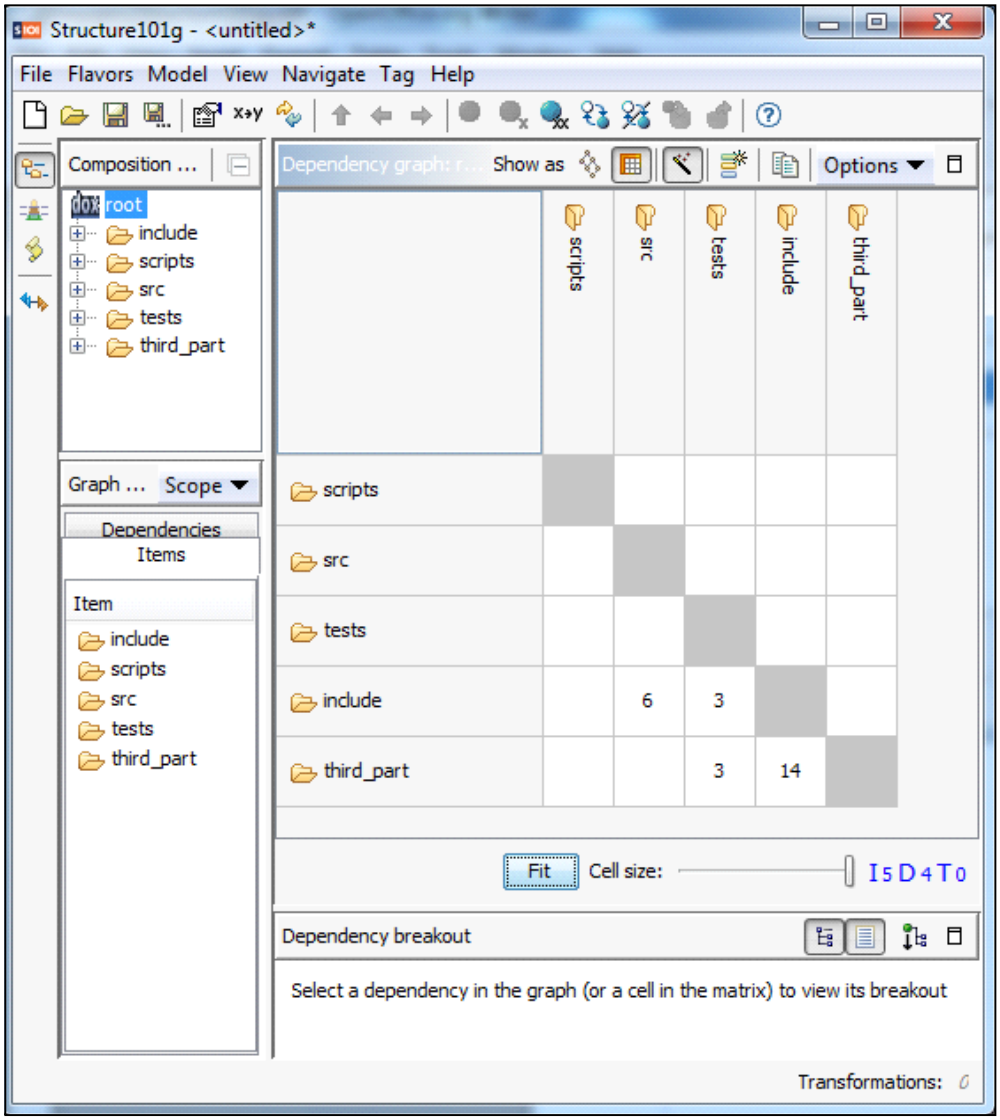
- Directory where you generated the XML files. There should be an index.xml in this dir.
- Dependencies to keep: leave the default value .* until you become an advanced user. This is a regex that filters what kind of dependencies you really want to capture from Doxygen. The types of dependencies are listed in metadata.xml of this flavor. Some values are: includes, extends, hasParam, etc. So, for example, if you just want to capture the dependencies of include files in your C++ project, you want to use the value *includes*. If you want to capture includes and also subclass relationships, you want to use *includes|extends*. Use any regular expression to filter the possible values that you want to filter in.
- Entities to keep: leave the default value .* until you become an advanced user. This is a regex that filters what kind of entities you really want to capture from Doxygen. The types of entities are listed in metadata.xml of this flavor. So, for example, if you just want to capture the dependencies of include files in your C++ project, you want to capture just the file entities, and therefore you want to use the value *file*. Entities like classes, structs etc will be discarded in this case (producing a more compact DSM). Use any regular expression to filter the possible values that you want to filter in.
- Blacklist file: This should be a text file, containing multiple lines, each line defining a dependency you want to discard (filter out). Each line should have the format: SRCID:DESTID:DEPTYPE. In other words, it is a triple where the first value is the ID

of the element that is source of the dependency. The second value is the ID of the element that is the target of the dependency. The last value in the triple is the dependency type (the same ones we saw previously – includes, extends, hasParam, etc).

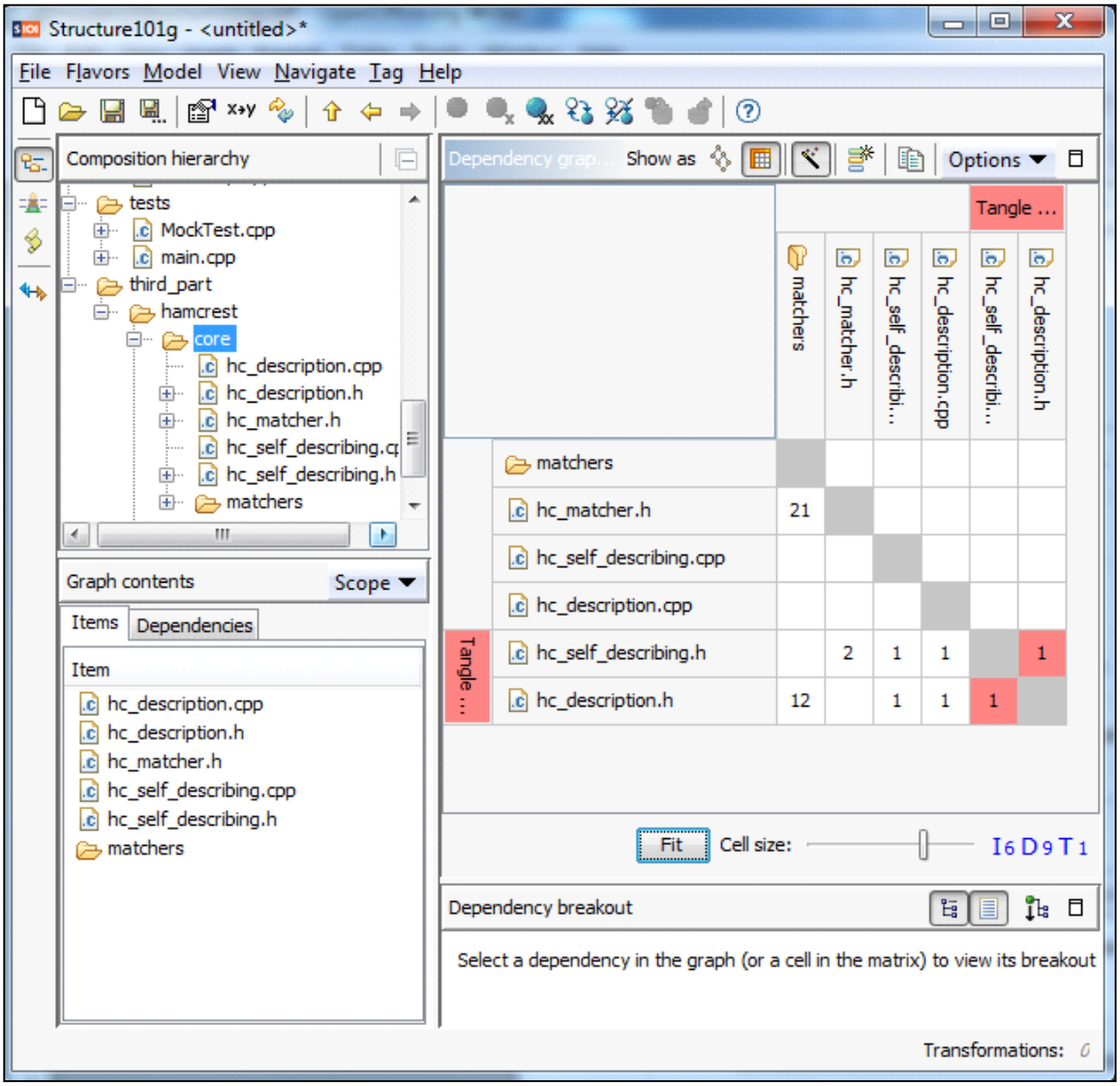
The screenshot below illustrates the values for loading the XML files produced for Carnamock.



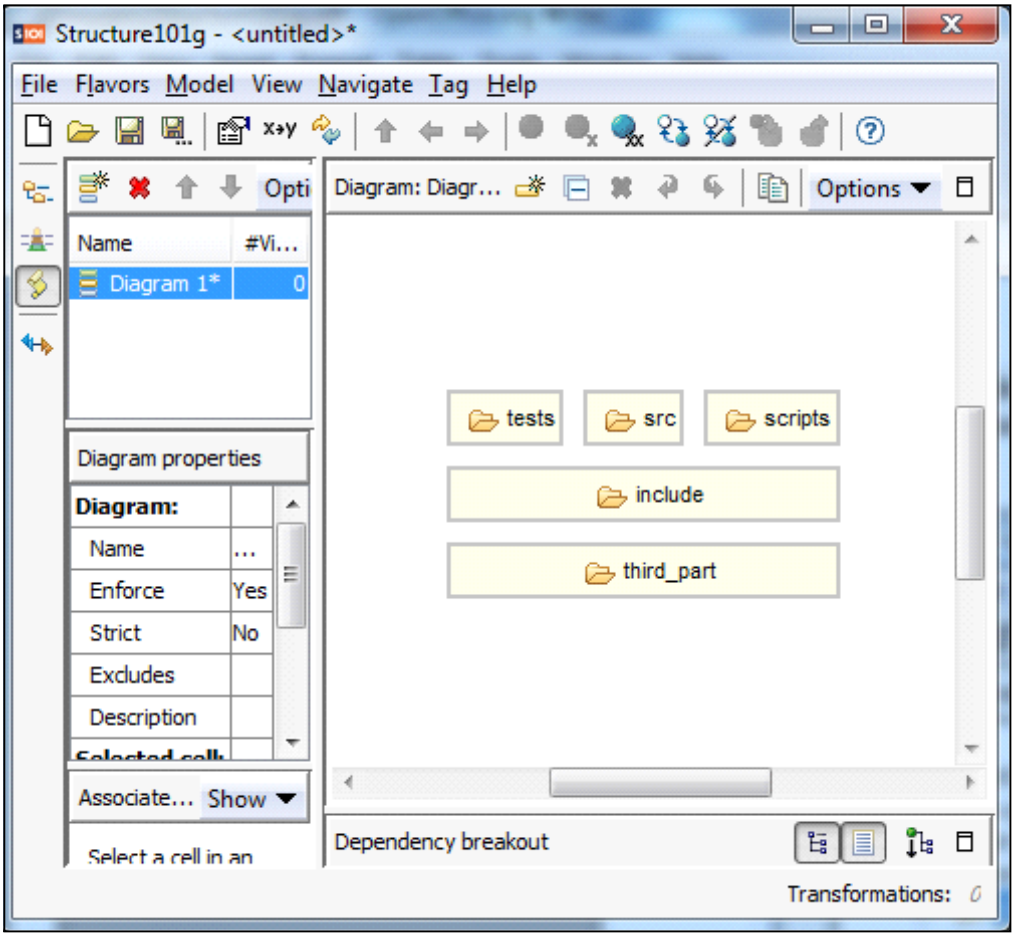
Click Finish. The conversion will happen and at the end you should be able, for example, to switch to the DSM perspective and see something like this:



As usual, you can drill down with Structure 101g and analyze individual "modules"/"components" (directories containing files, in a filesystem perspective). Here's an example inside Carnamock:

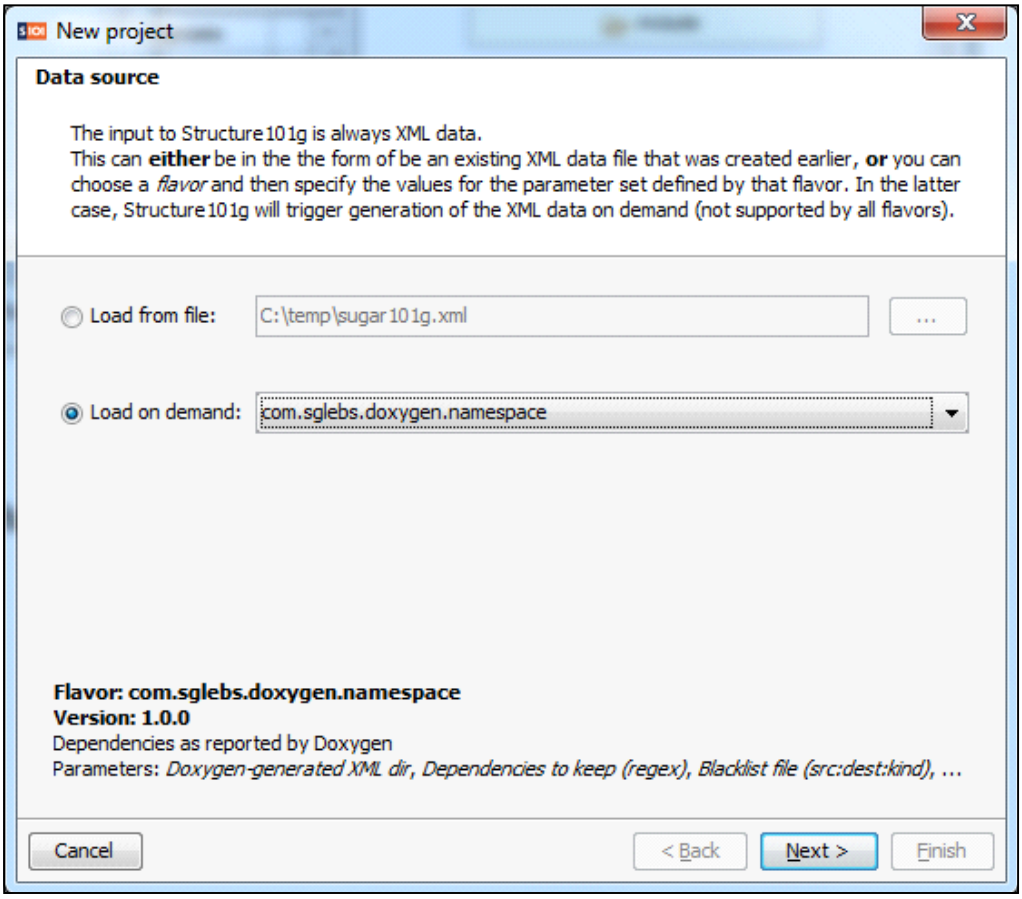


The architecture diagram should also work, as can be seen below (for Carnamock):

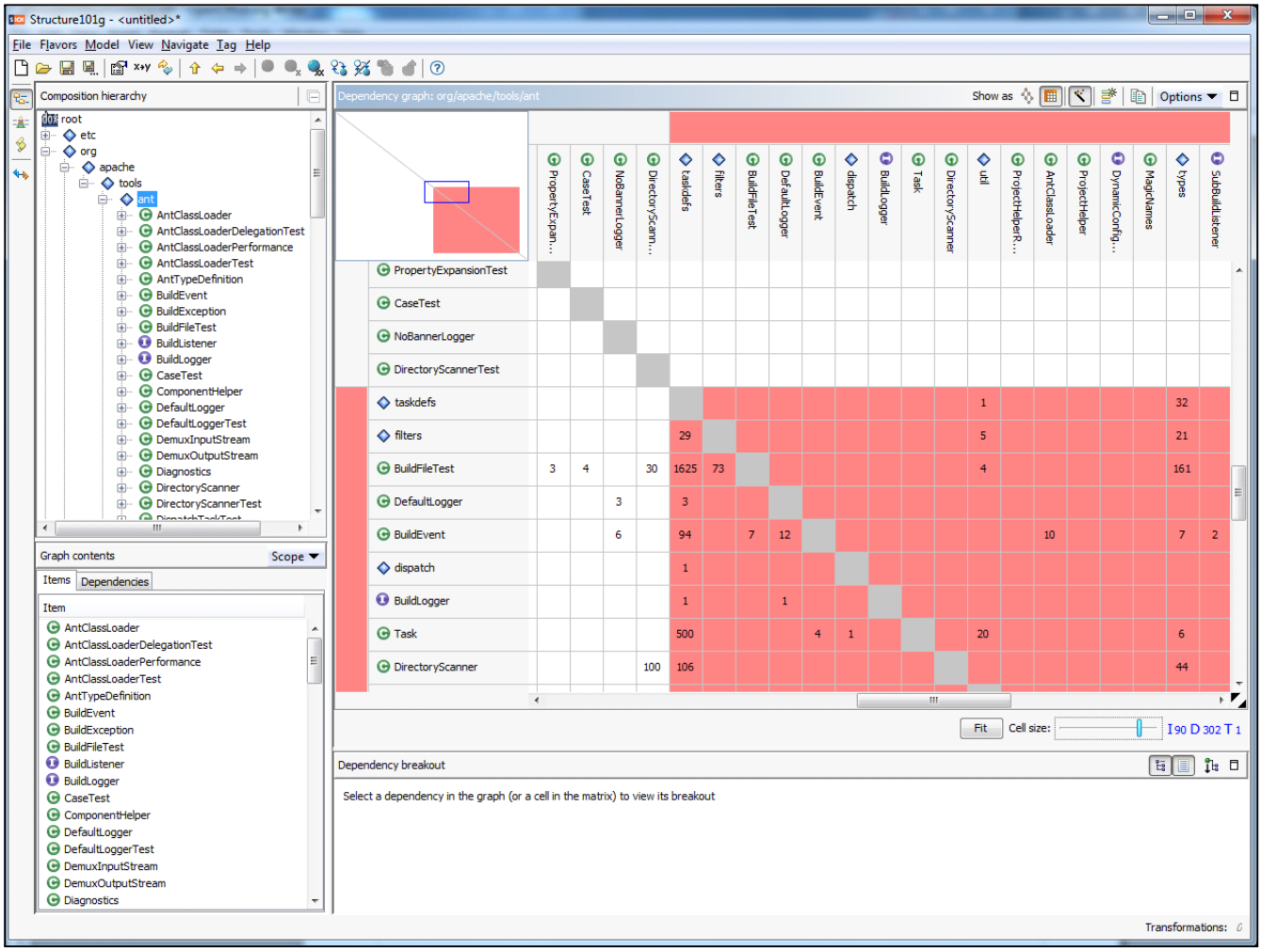


7.2) Running the com.sglebs.doxygen.namespace Flavor from the GUI

Running the namespace flavor is very similar to running the filesystem flavor. First make sure you choose it appropriately:

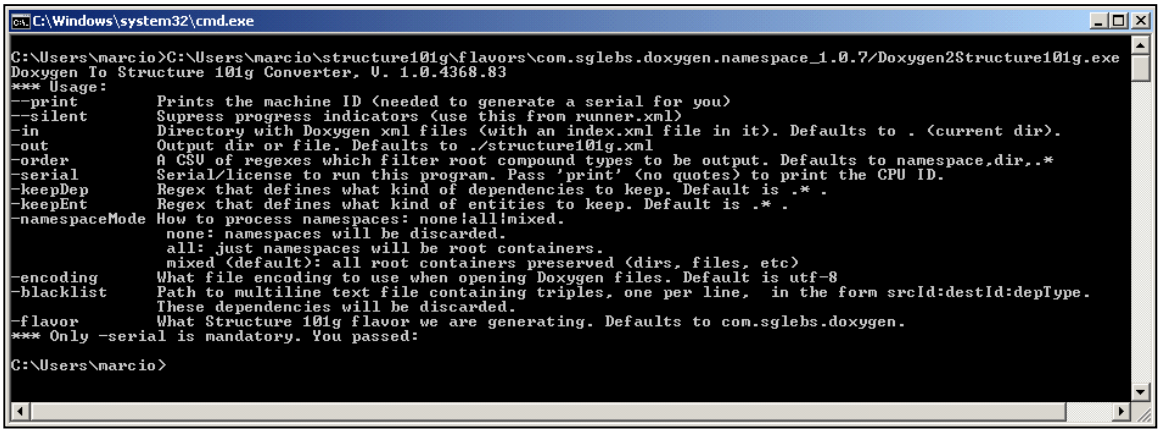


Then you provide the input values just like with the filesystem flavor seen in 7.1. Here's an example of what we get when running it on the XML files produced by Doxygen on the Java source files for Ant:



7.3) Running any flavor from the console

Each flavor directory comes with a copy of the *Doxygen2Structure101g.exe* file. If you run it without parameters, you will get something like this:



We highly recommend that you use the GUI wizard and use the little “Copy command line” link at the bottom/right of the Dialog. It will copy to the clipboard the full command-line used by the wizard behind the scenes. For instance, here's an example of what we used to run it on the Java Ant source files:

```
C:\Users\marcio\structure101g\flavors\com.sglebs.doxygen.namespace_1.0.7/Doxygen2Structure101g.exe -in "C:\Users\marcio\Doxygen-tests\apache-ant-1.8.1\src" -namespaceMode pure -order namespace,class,dir,file,.* -out "C:\Users\marcio\AppData\Local\Temp\s101g_7373410421013554521.tmp" -serial "C:\Users\marcio\structure101g\flavors\com.sglebs.doxygen.namespace_1.0.7\dox2101g.lic" -keepDep .* -keepEnt .* -flavor com.sglebs.doxygen.namespace -silent -blacklist "C:\Users\marcio\Doxygen-tests\empty.txt"
```

Try this approach different ways until you become familiar with the command-line options. Then you can invoke the executable on your own customizing the parameters as you want.

- Note that the serial value can be either theserial/license itself or a valid text file path containing the actual serial/license value. This is a key (serial) that you must obtain from us so you can run our converter on your PC. It is tied to your specific machine, unless we provided you with a TRIAL. Therefore, if the converter fails to run we ask that you send us your Machine ID. To obtain your Machine ID, run this from the console:
  Doxygen2Structure101g –print

This should print your machine. You should get an output similar to this:

Machne ID: ***48996***
Invalid '-in' parameter - no index.xml found in inout dir passed (.)