Re-familiarizing Yourself with Your Own Programs

This is the typical motivation for documenting code through design documents and comments in code. But the issue with those is that after a few change cycles, the documentation just does not cover everything anymore. That's where tools answering questions about the as-build dependencies in your code become important.

Who hasn't seen this scenario? Initially, a module (or class or otherwise encapsulated part of the code) has a clean structure, well-defined interface, and is easy to understand. Sometime later, there is a need for an enhancement; with some tweaks, this module now fits these requirements as well. What a great design! While the tweaks required a few compromises and the data and interfaces are spread out a little more, but we are certain to remember those few gotchas next time we need to make changes. And then in middle of some unrelated project, an emergency comes up with this code and quick action is required. Fortunately, we know everything about this code and are the best resource to fix these performance concerns. Surprise, things in the previous enhancements start to break.

The difficulty with fixing or enhancing our own code is that we could fall in the trap believing we know everything about 'our' code (much less any unknown changes other authors might have introduced in the meantime). It is natural that we might remember the concepts well but lose sight of details.

This is where lookup tools are essential. There is the basic search for all uses of a function, class, or variable. While these can be done textually, preferable are semantic driven lookups that know about the meaning of identifiers and avoid flooding us with irrelevant lines of code.

Higher level abstractions are helpful too. Even if our interest is a single function, seeing the use of that function in architectural layers can remind us that there are some uses beyond the original design. It will also show us that we used the function across layers once because it was an emergency fix and that was the quickest way to solve it.

Now typical changes involve more than a single function, variable or class. What we want to see in this case is the set of all primary entities of interest and all areas potentially affected by them. Then we can explore the areas of use, drilling down to make sure we understand the change impact.

Another important piece of information in re-familiarizing with the code is understanding recent changes. The basic tool for this is textual compares of source files. Again, in most cases we are back to too much information; working through all the changes is too cumbersome. If we can compare the different versions at a higher level and then drill down, reviewing changes becomes manageable.

We have all seen the numbers showing that more 50% of development effort goes into follow-on work on our software. But while initial development is well supported with tools and budgets, there is often resistance to invest also in useful tools to maintain the software over time. These investments save time and reduce unintended side effects of changes when we go back to code which we might

have known very well originally. It is just impossible to remember all details of real-life software.

Further information is available at https://www.verifysoft.com/fr_imagix4d.html.

Translation of a blog post, source: https://www.imagix.com/blog/re-familiarizing-yourself-with-your-own-programs/