Understanding Other People's Programs

The combined effect of large scale of today's software, the wide-spread use of open source components, and frequent project and job changes cause the software engineer to deal with unknown code. While his/her education and experience might help him to build up domain and framework knowledge, he still has to map large sets of source code into models he can work with.

Obviously, any existing documentation and knowledgeable staff will be a great introduction to a large new world. But eventually the engineer needs to work with the source code and find his way around it.
Source code has a natural structure impressed by directories and files. Drilling further down, functions and global variables — and in newer languages, classes and namespaces/packages — show the next level of structure. Use relationships like calls and data use link up these structural elements. All of these structures can be called the physical architecture of the software.
Architectural diagrams expressing the physical architecture are essentially the maps that help the software engineer navigate and comprehend the elements and their interdependencies. For these architectural diagrams to become useful, a key element is that they be interactive and allow the user to move freely through levels of abstraction. But unlike zooming in on geographical maps where the rest disappears except for the area of interest, architectural diagrams can still show the surrounding areas at higher levels of abstraction while drilling down in one area. This way they accommodate the behavior of programmers who work at all levels of abstraction.
The traditional call graph as an abstraction of the statement by statement execution order of course still has its place in the comprehension process. A place to start might be well be the entry point of the program, collapsing non-interesting parts into subsystems as it grows out. Again, following the observation that programmers' work at all levels of abstraction, the tools they use need to provide these various levels of showing structure and control flow, and easy ways of combining and switching between them. Ideally, such levels are down to the statement block level, displayed in the form of flow charts.
Depending on the purpose of the program exploration, a data flow approach might lead to a quicker understanding of the relevant parts. Data flow in its simplest form starts with the data variable and looks at all functions that directly or indirectly manipulate it. A more evolved approach is to track how the data values are calculated from other data values. This data-driven slice through the program can in theory show how each data variable is calculated from the set of input values.

The third dimension of program understanding is concurrency. Of course, some programs are purely linear in control flow and therefore the notions of control described above are sufficient. But the use of tasks in embedded systems or threads in hosted applications is more common now and requires additional models for understanding.
The first step is to identify the tasks in the program. Typically, tasks get started through a call to some system routine and can therefore be determined with basic call graph searches. The next step is to determine how the tasks interact with other each either through shared data variables or through events or signals. These require additional graphical models in the form of task flow or collaboration diagrams.
Overall, the process of understanding unknown code is about building models in the programmer's mind. Models are about abstracting and seeing the concepts that are used in the program. To help the programmer with that, various levels of abstraction need to be combined and it needs to be easy to switch between them.

Tools such as Imagix 4D https://www.verifysoft.com/en_imagix4d.html are available to the programmer for convenient source code analysis.

Source: https://www.imagix.com/blog/understanding-other-peoples-programs/