Q.1: Classify and compare the “Dynamic and Static verifications techniques” you are aware of.

Dynamic testing differs fundamentally and almost completely from static verification in many ways. To discuss them in comparison, the only point they share is a relation to the source code: static analysis takes source code in as it’s input, while dynamic often operates on a compiled form of the source code.

The first form that I am familiar with is a style of static verification. This is compilation - the process of compiling source code into executable format. A good, strict, compiler (i.e. gcc with -WError) will catch quite a few bugs in the program and offer a good first level of static verification - they can often eliminate many bugs and misspellings, programming mistakes and missing characters. Though compilation may not typically be considered to be a form of static analysis, I consider it one, as it essentially takes source code in, and improves it, demonstrating where there may be bugs or are typos.

Next I have experience with another form of static verification, called linting. Linting is static verification in its purest, automated form: it takes a look over the source code, and inspects it against style rules, and recommends changes to make the code more readable. This is a form of static analysis / verification, as it does not require running the source code.

I am familiar with a form of Dynamic Verification: CI Testing. What CI testing does is run code, and expect a result, for example an exit code 0, or a result printed. This ensures that changes in the software have not violated conditions of satisfaction for the project. Further, this can serve as a method of managing milestones - each CI test complete is a milestone.

Another form of dynamic testing I know of is Integration Testing. Easily confused by me with CI testing, this differs in that its goal is to test how parts of a program work together. For example, if you have a large, refactored project, the Integration Tests would compile those parts individually and assert that they are unchanged, to be able to assure the function of each module. For a dynamic array module, for example, this would entail compiling it with a test suite that tries all the operations for that module, and tests the edge cases associated with those operations.

Q.2: Explain and provide an example for each of the following concepts of “Failure”, “Fault”, and “Error” in the context of Software Testing.

A failure is essentially described to be a state where the system cannot perform as it is required - it operates in a manner that is not consistent with the specification of the software. For example, it would be a failure if software designed to compute the altitude of a plane could not produce an accurate enough altitude, or produced the speed instead.

A fault is a problem in the code that results in a failure. Given the example above, this failure could result from a programming mistake in the code such that the altitude is converted to int from float, reducing the necessary accuracy, or in the latter failure case listed above, this could be caused by mislabeled or unclear variable names, such as ‘a’ or ‘input’. ‘Input’ could contain the data for the wind speed, and the programmer made the program have a fault in it by considering this value to be the altitude. Either of these cases, the thing that caused the fault is likely an error (see below).

An error is a mistake caused by the programmer or human designer that results in a fault. This does not have to be human caused, but it likely is. An example of this is the human programmer’s decision to name variables with non-descriptive names like ‘test’, ‘input’.

Q.3: Explain the limits of the software inspection technique?

Software inspection is a limited process as a result of the fact that it requires human participation to work. It is based on the human(s) conducting the tests, and it’s success is directly tied to the focus, expertise, memory and knowledge of the inspector. The inspectors skills in these areas can prove to make the inspection less effective, as I will illustrate using two scenarios below.

The first scenario is one where the inspectors have not fully prepared. As an inspector, you are required to know the source materials, such as designs or deliverables that this code is supposed to follow. Suppose that the inspectors are new to their job, and they leave this to the night before, and are tired from a week of work. They will skim read this information and not internalize it as perfectly as a computer might (in formal verification). Therefore, their background on the program’s desired specifications might be dull the next day. Consider a different scenario where the project is massive. In this case, the inspector(s) memories will be tested as there will be many, many specification parameters and conditions of satisfaction, tests and goals to remember, or have on hand. This is almost impossible to use without slowing down the project and inspection greatly.

The other scenario is one where the inspection is rendered less useful as a result of the inspector’s credentials. For example, asking someone with a BS in CS to be on a team reviewing microcontroller drivers might be quite surprising to that individual, as drivers and low level C are programmed very differently than they learned in school. This could extend to several members of the inspection. These members will likely not speak up about not having the credentials to review this as they have ‘C Programming’ on their resumes, and do not want to disappoint their manager / get fired. So they sit there, and cannot meaningfully contribute to the code inspection, as they are struggling to comprehend the use of, say, binary operators.

Q.4: Compare the following different Dynamic Testing techniques: Unit Testing, Integration Testing, System Testing, Release Testing.

Unit testing is a type of testing where individual parts, “units” are tested separately to ensure that each of these units behaves and performs as expected. This type of test relies on the assumption that if the units all function as designed, the whole will work. The goal of Unit Tests is to call these units with as many input parameters as possible to get the best overview of function.

Unit tests are great for checking small pieces, but Integration Testing is crucial to the utility of unit testing. Integration testing takes the units (possibly checked with unit testing) and combines them, to test the functioning of the whole system. This can sometimes entail stubbing certain parts of the software so that all units work right. Essentially, this tests that the units work together as expected.

System testing is one level above integration tests. Instead of there being the possibility of using sub-assemblies to test (where integration testing can test collections of a subset of the units of a program, system testing cannot), system testing tests the entire system together. This test is the most deterministic way of ascertaining whether a program or suite of software meets the requirements or not.

Release testing is a subset of system testing where the engineers that wrote the code have no role in the process of testing the software. Essentially, this team checks as well as they can for bugs and other defects, checks that the system meets the requirements, and most importantly, that it is ready to be released. Typically these releases are rolling, and are assigned a version once they are validated to be working and meeting criteria.

Q.5: Illustrate the verification and validation concepts.

Verification and Validation are two relatively disparate ideas that work together. Together, they can help guide a project to completion and the goal of meeting requirements. They enable this approach by discovering not only bugs in the code, but also places where the software does not meet the specification, and needs additional features.

Verification takes the role of ensuring that the software functions right - it eliminates the ‘bugs’ we discussed earlier, like errors, faults and failures. These ensure that the product is in a good functioning state, and can eliminate the errors that cause the faults causing failures. In a way, this overlaps with validation, as the elimination of failures can actually meet requirements.

Validation is the process of checking off requirements, and making sure that the project meets the requirements it must. This process typically takes place at the end of the Software Development Life Cycle, and is responsible for making sure that the project satisfies the needs for which it was designed.