**How do you review code?**

**While reviewing code at first it try to figure out what is the code doing and what’s the purpose behind it?**

If I don’t have the idea about the purpose of the code then I’ll not be able to add productive comments

In this case I ask 3 question myself that are given below

* Do I have the right background info?
* Do I fully understand why this code is needed?
* Do I understand how this code fits into the project?

**Test the code**

* Does the code generate errors/ warnings?
* Dose the code take longer than normal to build and run?
* Do the changes requires any additional setup steps

**Inspect the code?**

* Are the naming conventions followed?
* Does the code follow dry principle?
* Are new file and folder named correctly?
* Does the code have error handling?
* Is the code written in a similar style to another code in the project?

**Compile my review**

* By adding comments to specific lines of code
* Asking question in a common thread
* Sometimes create a new brunch with suggested changes

**While reviewing code I always try to be aware of the following things**

* Be respectful
* Be specific and descriptive
* Be open to follow up
* Try not to review more than 300 line code at a time

**How do you enforce coding standards?**

* Try to use write data structure for a given problem
* Try to write compact code
* Try to avoid unnecessary variables
* Avoid abbreviation as much as I can
* Use descriptive functions and variable name over code comments.
* Try to keep the code simple
* Too much nested code kills code readability
* Consistent coding style and naming convention
* Try to use the right structure of file and folder
* Using OOP
* Use design pattern

**How do you plan what kind of approach you take for test automation - what libraries to use, how does it work in couple of years, how to make it easy to maintain, etc? What are the main points to consider?**

The test automation framework is a set of detailed guidelines that one needs to follow while writing and running tests. For example, coding standards, procedures, test-data reporting, etc. There are six test automation framework that you can be used for automation testing

* Data-driven Test Framework
* Modular Based Testing Framework
* Keyword-driven Test Framework
* Hybrid Testing Framework
* Linear Scripting- Record & Playback
* Library Architecture Testing Framework

While preparing strategy about automation I also keep in mind the following things

* Test Automation Scope

### Test automation Approach

### Risk analysis

### Test automation environment

### Execution plan

### Review & analysis

At first I try to understand the context of the project and try to understand what type of testing is required then I decide what libraries to choose. In recent times I’ve used mostly Keyword driven Test Framework (RF). I used data driven approach in this framework also. And I found it very easy.

In the project I also could have applied data driven approach but did not due to time contrarians.

I’ve always been advised to remain focused on maintaining the effectiveness of regression tests. This can be achieved by periodically cleaning older test cases and categorizing the current ones. You should remain clear on your requirements and whether old test cases are effective to keep.

Test scripts should be minimal with more outcomes. If you can make smaller test cases, automation becomes easier.

**Robot framework libraries I used**

* Selenium
* Browser
* Appium
* Request
* AutoRecorder
* Robotframework-faker
* allure-robotframework

**Code testability, how do you enforce it?**

* Testability is a leading characteristics of clean code.
* I try to break code in to pure functions, where input -> output with no side effect.
* Try to follow Single Responsibility Principle. This principle can be applied at both class and method levels and to some extent at the package level. The Single Responsibility Principle does not state that a module should only do one thing. It states it should have one and only one reason to change.
* Dependency Inversion Principle (DIP) ->High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.
* Law of Demeter (LoD) Another “law” which is useful for keeping the code decoupled and testable is the Law of Demeter. This principle states the following: Each unit should have only limited knowledge about other units: only units “closely” related to the current unit. Each unit should only talk to its friends; don’t talk to strangers. Only talk to your immediate friends.
* Interface Segregation Principle (ISP) -> No client should be forced to depend on methods it does not use.

**How do you make sure that the product is testable?**

* Requirements are clear
* Has enough sample data for testing
* Has a test environment that matches production in terms of features and infrastructure