**Other questions to answer about testing and testability, from your point of view**

1. How do you review code?

The main focus while I review code is to evaluate the impact of the change request to the overall quality of the code base in terms of continuous improvements. I also ensure that the code is easy to maintain, easy to understand and that the tests are valid and do not introduce false positives. I read and analyze the code to see if it can introduce bugs or race conditions or breakages in the other parts of the overall system. I use unbiased professional tone in ensuring that coding guidelines, test implementation and annotation guidelines are adhered. Validate test tags management is followed in the desired order.

1. How do you enforce coding standards?

Define and document the standards. Introduce and Communicate the standards across the teams through various channels like Community meetings, emails, and presentations. During code review, remind reviewers to evaluate that the coding standards are met by providing direct links for the documentation in the review page. Automated code style profile tools can also be helpful for enforcing coding guidelines through config files or IDE settings.

1. 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?

First, I will try to understand the architecture of the whole system under test. Then, I’ll analyze the full tech stack being adopted across different systems. Based on the gathered information, there are three possible ways to choose automation approach:

1. Picking test libraries or frameworks that are in close relation to the available tech stack results in creating an ecosystem of software code and test code that can co-exist in the world of continuous improvements and continuous delivery. For example, in case of Mobile automation for iOS: choosing XCUITest that comes with the SDK of iOS tech stack ensures that any frequent changes in the core iOS platform or the Swift language is well-taken care of in XCUITest so there is very little turbulence we will encounter during maintenance phase.
2. Choosing test libraries or frameworks that are more widely used by several different developers/testers across the world but that are slightly different to the chosen SUT architecture will ensure that there is plenty of open source tools and support to maintain the automation suite. This also ensures that more testers/developers are well versed in the technology and the need to provide additional training is reduced. They are more probably using client-server architecture for frontend testing. Good examples are Selenium, Appium, Robot Framework, etc.
3. Writing own custom in-built software for writing test code. Though this approach will be very flexible to write tests, but it is however very expensive and very hard to maintain.

The matrix below illustrates the advantages and disadvantages of these various approaches which will guide in choosing the viable solution:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Approach** | **Maintainability** | **Long running** | **Need for training** | **Resource allocation** | **Other factors** |
| Approach 1- Native | Very easy to maintain. | Insulates against future possible changes. | Additional training may be needed for testers. | Developers can pitch in during critical phases as they are already well-versed in this tech stack. | Optimal solution; Write once and never look after unless fundamentals change.  Cross-platform support is hard. |
| Approach 2 – 3rd party | Easy to maintain though continuous monitoring is needed. | Long running though there is need to manage flakiness, changes in the ecosystem, etc. | Very less training required. | Additional resources where developers can assist are less as they need some level of training. | Feasible solution; Take-off is easy  Cross-platform support is very easy. |
| Approach 3 - Custom | Hard to maintain | With rapid tech changes, this will hardly survive. | Newcomers will have to be trained heavily. | Might be flexible to scale between dev and testers. | Not optimal solution |

There are advantages and disadvantages on all these approaches. I’ll choose one of the approaches based on the current status of the test project and several other factors. Other main points to consider in choosing a strategy for test automation:

* TDD, ATDD or BDD approach towards testing will improve the maintainability of the code.
* Scalability or extensibility to run large numbers of tests without much overhead.
* The right product fit between the architectures of SUT (System Under Test) and the TAS (Test Automation Solution). Mismatch between these can result in overhead.
* The right fit between the existing skill set of test automation engineers and the selected tools.
* Data-driven or keyword-driven or behavior-driven strategies.
* Metrics, logging and reporting capabilities of the tools.
* How long is the tool in market, how frequent are the releases, how good is their support system in terms of training, bug fix, problem resolution, licensing etc.

1. Code testability, how do you enforce it?
   * 1. By evaluating the proposed software blueprint during the architectural design or planning phase and by offering related suggestions to include solutions for testability can increase code testability.
     2. Test Driven Development where automated tests are developed hand in hand with or before the development of the SUT (System Under Test) will enforce code testability.
     3. Making testability as a non-functional requirement to be included in the Definition of Readiness checklist.
     4. Mimicking or mocking stubs to generate simulated solutions to invoke corner scenarios.
     5. Testability can also be improved by including test-automation IDs in the code base so that the automation suite can detect the objects easily.

1. How do you make sure that the product is testable?

In addition to the suggestions provided in the above section, these will help to make the product testable:

* + 1. By enforcing that the requirements are well defined with clear acceptance criteria will clarify what is the expected behavior and ensures that the product is testable.
    2. Designing additional test modules for some functionalities such as a feature that is presented only once during its lifetime, by providing interfaces to turn them ON as required for repeated testability.
    3. To control the states of the application under test by using software interfaces with the help of feature flags for testing purposes can also make the code testable.
    4. If the product has met the guidelines of Definition of Readiness, then we can be sure that the product is testable.
    5. Requirements mapping, code coverage metrics along with unit and integration tests pass criteria will ensure that the product is testable.