Agile Test Strategy

https://devqa.io/agile/agile-test-strategy-example-template

Agile Test Strategy

In an agile environment, where we work in short sprints or iterations, each sprint is focused on only a few requirements or user stories, so it is natural that documentation may not be as extensive, in terms of both number and content.

The purpose of the agile test strategy document is to list best practices and some form of structure that the teams can follow. Remember, agile does not mean unstructured.

Here, we take a look at a sample Agile Test Strategy and what to include in the document.

A test strategy usually has a mission statement which could be related to the wider business goals and objectives.

A typical mission statement could be:

To Constantly Deliver Working Software that Meets Customer's Requirements _by means of _Providing Fast Feedback _and _Defect Prevention, rather than Defect Detection.

Supported by:

- No code may be written for a story until we first define its acceptance criteria/tests
- A story may not be considered complete until all its acceptance tests pass

In the Agile Test Strategy document, I would also include a reminder to everyone about Quality Assurance

- QA is a set of activities intended to ensure that products satisfy customer requirements in a systematic, reliable fashion.
- In SCRUM (agile) QA is the responsibility of everyone, not only the testers. QA is all the activities we do to ensure correct quality during the development of new products.

Test Levels

Unit Testing

WHY: To ensure code is developed correctly

WHO: Developers / Technical Architects

WHAT: All new code + re-factoring of legacy code as well as Javascript unit Testing

WHEN: As soon as new code is written

WHERE: Local Dev + CI (part of the build)

HOW: Automated, Junit, TestNG, PHPUnit

API / Service Testing

WHY: To ensure communication between components are working

WHO: Developers / Technical Architects

WHAT: New web services, components, controllers, etc

WHEN: As soon as new API is developed and ready

WHERE: Local Dev + CI (part of the build)

HOW: Automated, Soap UI, Rest Client

Acceptance Testing

WHY: To ensure customer's expectations are met

WHO: Developer / SDET / Manual QA

WHAT: Verifying acceptance tests on the stories, verification of features

WHEN: When the feature is ready and unit tested

WHERE: CI / Test Environment

HOW: Automated (Cucumber)

System Testing / Regression Testing / UAT

WHY: To ensure the whole system works when integrated

WHO: SDET / Manual QA / Business Analyst / Product Owner

WHAT: Scenario Testing, User flows and typical User Journeys, Performance and security testing

WHEN: When Acceptance Testing is completed

WHERE: Staging Environment

HOW: Automated (Webdriver) Exploratory Testing

Product Backlog

Most common cause of software development failure is due to unclear requirements and different interpretation of requirements by different members of the team.

User stories should be simple, concise and unambiguous. As a good guideline, it is best to follow the INVEST model for writing user stories.

A good user story should be:

Independent (of all others)

Negotiable (not a specific contract for features)

Valuable (or vertical)

Estimable (to a good approximation)

Small (so as to fit within an iteration)

Testable (in principle, even if there isn't a test for it yet)

The following format should be used to write user stories

```
As a [role]
I want [feature]
So that [benefit]
```

It is important not to forget the "Benefit" part, as everyone should be aware of what value they are adding by developing the story.

Acceptance Criteria

Each of the User stories must contain acceptance criteria. This is possibly the most important element which encourages communication with different members of the team.

Acceptance criteria should be written at the same time the user story is created and should be embedded within the body of the story. All acceptance criteria should be testable.

Each Acceptance Criteria should have a number of Acceptance Tests presented as scenarios written in Gherkin format, e.g.

```
Scenario 1: Title

Given [context]
And [some more context]...
When  [event]
Then  [outcome]
And [another outcome]...
```

Story Workshops / Sprint Planning

In each story workshop, everyone in the team learns about the details of the stories so developers and QA know the scope of the work. Everybody should have the same understanding of what the story is about.

Developers should have a good understanding of the technical details that are involved in delivering the story, and QA should know how the story will be tested and if there are any impediments to test the stories.

Preventing Defects

In story workshops, **PO, BA, Dev, and QA must be involved.**

Scenarios (valid, invalid and edge cases) should be thought of (QA can add huge value here by thinking abstractly about the story) and written down in feature files.

It is important to note that it is the scenarios (more than anything else) that will reveal defects when testing the product, so the more effort and time spent on this activity, the best results at the end.

Because the majority of defects are due to unclear and vague requirements, this activity will also help prevent implementation of incorrect behavior as everyone should have the same understanding of the story.

Likewise, in the sprint planning meetings, the estimates given for a story should include the testing effort as well and not just coding effort. **QA (manual and automation) must also be present** in the sprint planning meetings to provide an estimate for testing of the story.

Development

When development starts, new production code and/or modification to legacy code should be backed by **unit tests written by developers** and peer-reviewed by another developer or a skilled SDET.

Any commit to the code repository should trigger an execution of the unit tests from the CI server. This provides a fast feedback mechanism to the development team.

Unit tests ensure that the system works at a technical level and that there are no errors in the logic.

Developer Testing

As a developer, behave as if you don't have any QA in the team or organization. It is true that QAs have different mindset but you should test to the best of your ability.

You think you are saving time by quickly moving on to the next story, but in reality, when a defect is found and reported, it takes longer to rectify the issue than spending few minutes making sure the feature works well.

Any new code and/or refactoring of legacy code should have appropriate unit tests that will be part of the unit regression test.

Automated Acceptance Tests and Non-functional Testing

The automated acceptance tests include Integration Tests and Service Tests and UI tests which aim to prove the software works at a functional level and that it meets user's requirements and specifications.

Automated acceptance tests are usually written in Gherkin language and executed using a BDD tool such as cucumber.

Remember: Not all tests need to be automated!

Because these tests typically require communication over `HTTP`, they need to be executed on a deployed application, rather than run as part of the build.

Non-functional tests such as Performance and Security tests are as equally important as functional tests, therefore need to be executed on each deploy.

Performance Tests should check performance metrics on each deploy to ensure no performance degradation.

Security Tests should check for basic security vulnerabilities derived from OWASP

It is vital that this should be a completely automated process with very little maintenance to get the most benefit out of automated deployments. This means there should be no intermittent test failures, test script issues, and broken environment.

Failures should only be due to genuine code defects rather than script issues, therefore any failing test which is not due to genuine failures should be fixed immediately or removed from the automation pack, to be able to get consistent results.

Regression Testing

Not expecting to find many defects. Their purpose is only to provide feedback that we haven't broken major functionality. There should be a very little amount of manual regression testing.

Smoke pack – Should be no more than 15 mins

This pack contains only high-level functionality to make sure the application is stable enough for further development or testing.

For example, for an eCommerce website, tests included in this pack could be:

- Product Search,
- Product Review
- Purchase Item
- Account Creation / Account Login

Full regression pack – should be no more than 1 hour

This pack contains the full regression suite of tests and contains everything else which is not included in the smoke pack.

Here, the goal is to get a quick feedback with a larger set of tests. If the feedback takes more than 1 hour, it is not quick. Either reduce the number of tests by using pairwise test technique, create test packs based on risk or run the tests in parallel.

UAT and Exploratory Testing

There is no reason why UAT and exploratory testing cannot run in parallel with the automated acceptance tests. After all, they are different activities and aim to find different issues. The aim of UAT is to ensure that the developed features make business sense and helpful to customers.

PO (Product Owner) should run User Acceptance Tests or Business Acceptance Tests to confirm the built product is what was expected and that it meets user's expectations.

Exploratory testing should focus on user scenarios and should find bugs that automation misses. Exploratory testing should not find trivial bugs, rather it should find subtle issues.

Done Criteria

Once all the above activities are completed and no issues found, the story is **Done!**

The above are some guidelines on what can be included in an Agile Test Strategy Document. Obviously this needs to be tailored to your organization's needs, but hopefully, this template would assist you in creating your own Agile Test Strategy document.