Feature files

Some common formats suggested for use between OUP projects using BDD.

Terminology

- **BDD**: Behaviour Driven Development.
- **Feature File**: a text file with the .feature extension that defines a set of rules and examples for a specific feature in a system. They are written in natural language but contain elements that can be tested automatically
- **Rules**: a list of rules that define a feature. Similar to Acceptance Criteria. e.g. "Aces are high / Playing an ace wins you the last played card"
- **Scenarios**: examples of the rules in the real world. Can be positive or negative. e.g. "Given Player 1 played Queen / When Player 2 plays Ace / Then Player 1 wins the trick / And Player 1 takes the Queen" - not all rules can necessarily have an example.
- **Gherkin**: the name for the syntax used in feature files (Feature/Scenario/Given/When/Then etc)
- **Cucumber**: a tool used to run feature files as automated tests
- **Tags**: identifiers starting with an ampersand that allows some Scenarios to be tested only in a certain context (e.g. manually, automatically in a browser)
- **Contexts**: a testing context e.g. in a browser. Some contexts all for automated testing (e.g. @browser using Selenium or @api using JMeter)

Converting User Stories

Here is an excellent introduction to breaking down user stories into Rules and Examples:

https://cucumber.io/blog/2015/12/08/example-mapping-introduction

Tagging

Any Scenario can have one or more tags. Please use the tags below when working on OUP projects if possible:

- @wip - not fully defined. Will be excluded from all testing. Overrides all other tags
- @manual - must be manually tested. Not included in automatic tests.
- @browser - must be tested in a browser context. Tests will fail if it does not have matching BrowserContext tests.
- @api - must be run in a service context. Tests will fail if it does not have matching APIContext tests.
- @javascript - must be run in a JavaScript-aware context. Tests will fail if it does not have matching JavascriptContext tests.
- @critical - identifies tests that must be run on every release (AKA smoke tests) - useful if the full manual test suite is too long to be run but a level of assurance is required.
- @pending - Pending sign-off
- @agreed - Signed off
- @release

Process

The process of creating feature files will depend on the project. Here are some suggestions:

Setup

1. Create a new folder in \\Gboxfdfs01\OUP\OUP Group\Group-Net\feature-files for your project and push it to GitHub (ask the Platforms SDC if you need help). Use the format "features-yourprojectname"
2. Identify the key user types in your system and make a folder for each one
3. You can create feature files inside each of these folders (or even top level for features affecting all users)
4. If it helps navigation, create a subfolder by user for any top level concepts in your system for that user (e.g. "login", "dashboard", "import")

Tests can be run across all folders, a subfolder or for a single feature file. Tests will also consider tags (e.g. manual tests will not be included in browser context tests.)

Defining a feature

Use whatever file format your team are most comfortable with. If your team favours Word, create a document with the name of the feature and write the various rules and add comments etc. Similarly you could define in Excel and think of the various tests required in separate columns. If working in Google Docs, you may prefer just to have a feature file that links to that document.

You may want to start writing some examples of system behaviours in the Given/When/Then format.

Getting feedback

When you have defined your basic Rules, you should share those ideas more broadly, ideally involving at least a developer, a tester and an architect if appropriate. Questions should be added to the feature file (or feature document). These must be answered before the feature can go into development or progress further.

It is especially important that dependencies between different features are understood and removed/accounted for. Ideally the feature should be testable in isolation.

Writing some examples

Before they can be tested, rules must have some positive and negative examples. When the basic feature is understood, the product owner and/or QA can write some examples that are likely to start as @wip (untested), become @manual tests and then hopefully also @browser so the test can be run automatically.

Development

Depending on whether you are working in a test-first (TDD) manner, you can now start building your solution. Or you may ask your developers to write the tests for the various rules as they go. This will depend on your team structure and there is no "right" way to do it, even if someone tells you there is. Develop first if it helps. In some cases, the feature may already exist in which case you are definitely going to have to test-second.

Testing

Ideally you will build your automated tests into a continuous integration (CI) system so that you get immediate feedback on issues. In practice, this is hard. At very least, associated tests for a feature should be run when the feature is complete (even if only manually) and during regression tests. In an ideal state, all tests are run on every commit. At least once per release is the bare minumum. Smoke tests can be a way of assuring the major Scenarios work still and we can use the @critical tag to identify them.

In some projects, developers will implement their own automated tests. In others, a dedicated test team will write them. The main thing is that you get some coverage if possible. But writing features clearly is a good start even if they are all @manual to start with.

Migrating

If you've already defined features in another system, here are some ways you could approach the transition:

New project with ACs written and available in Jira

1. For each Jira ticket, extract the ACs and create a feature file for each.
2. Extract the ACs from the Jira ticket to prevent duplication of rules
3. Reference the feature file as a link in the Jira ticket (e.g. \\Gboxfdfs01\OUP\OUP Group\Group-Net\feature-files\features-myproject\teacher\login\login-page.feature)
4. Make sure the feature file only covers one feature.
5. Split the ACs across multiple feature files if required and link each one to the associated Jira ticket

If this process sounds onerous, start small and just work through the items already prioritised in the backlog.

Existing project with limited documentation

1. For any feature you are changing, retro-fit the feature file to describe current behaviour
2. Still ensure you ask around to ensure that is really how it works and should work
3. Write enough Scenarios that you have some basic test coverage. You may be able to source these from existing test scripts.
4. Create a Jira ticket for the change, referencing the precise Rules and Examples that changed and why
5. Develop the change
6. Adapt (or create) new tests

Storing and Change Tracking Feature Files

Feature files are usually executed by platforms as part of their test suites. So, there needs to be one source of truth. They are also, in a very real sense, code. They just happen to be more human-readable than, say, Perl.

So, for any feature that is being developed, we will commit the feature file to source control which for us is Git. Our chosen storage for our repositaries is GitHub.

For each project we will create a new repo in GitHub. For example, this is the repo for Digital Reading:

https://github.com/OUP/features-digitalreading

This is a similar format to the equivalent "design-digitalreading" repo. We should follow this same format for clarity and consistency.

For users who do not know Git

One major aim of feature files is that they are readable without special technical skills and owned by the business. For this reasons, a low barrier to entry is critical. Those unfamiliar with source control can access feature files on the OUP network:

[\Gboxfdfs01\OUP\OUP Group\Group-Net\feature-files](\Gboxfdfs01\OUP\OUP Group\Group-Net\feature-files)

Each repo inside this network folder will be overseen by a member of the project team who will review changes on a daily basis and commit as appropriate. A PO or QA lead is a good choice for this role but in the short term, Platforms SDC can assist with managing changes.

If people do not want or need to engage with BDD at all, Word or similar assets can be placed in the appropriate context in the feature-files network folder and can be converted to feature files as required by the team.

For users who wish to learn Git

In time, users will be encouraged to commit their changes via the VS Code source control tab. This is very simple and will encourage committing changes to features clearly and with a description e.g. "Changes to Login functionality after meeting with UX 2017/09/10" (this might span several feature files.)

For users who are familiar with Git

If you know how to use Git, you can clone the repo you are interested in (e.g. OUP/features-digitalreading) via your favourite tool.

VS Code has excellent support for Git so this will be the choice suggested for most editors. If you need setup assistance, please contact the Platforms SDC. Install rights are required so IT must be contacted, after which the "Cucumber (Gherkin) syntax" extension should be installed.

It is assumed that all developers and testers working on the project will use a local copy of the appropriate feature-file repo. This will also prevent non-signed off features from being worked on. For example, it is not expected that developers should start building features that are still being discussed in a Word document but they might contribute comments.

FAQs

> Is it enough to refer to the feature files in the acceptance criteria section and update the features files as the stories are generated to make the changes

Yes. The exception would be if you need to make several tickets for a single feature file or one ticket that refers to parts of many (this isn't common but it does happen). In those cases, just copy/paste the relevant Rules into the ticket (and provide a link to the original file so there's some context).

Note that there may be ACs for the ticket that aren't in the feature file. For example those that relate to how the task should be carried out or tested (perhaps you are making a feature that imports users and you want 10 users to be created on Dev with that feature to prove it works then for a performance metric to be

supplied). In those cases, put them in the ticket as normal. They can then be signed off by QA separately.