BDD Pointers

**Process**

Create acceptance tests for scenarios which are considered valuable when repeatedly tested.

(e.g.: Happy path, Error scenarios).

Consider creating and adding points for automation test when the PBI has an action to perform.

·        Action performed**should** be validating errors, review details, success message, error message etc.

·        Action performed **should not** be draw input fields, test created link, help icon works etc..

**Feature File**

**Organization**

1.     **Related functionality should be placed in the same feature file.**

2.     **Some thought and research should be done researching existing Step Definitions before creating new steps.**  
Re-using existing steps we reduce the amount of low value Step Definitions that are created and maintenance costs.

3.     **Order of scenarios in feature file should follow workflow steps.**  
Ordering by workflow steps is a logical and easily understood method to lay out feature file.

4.     **Folders are used to represent epic level features.**  
Use organization by epic (Appointments, Parts, etc.).

5.     **Feature file naming should use business domain words in the format *VerbNoun*.*feature*.**  
We need a consistent naming format of feature files to aid in organizing and locating feature functionality.

6.     **Distinct, yet related functionality should be placed in a separate feature file.**  
To promote manageable feature file sizes, related functionality should be placed in separate files.

7.     **Do not create technical feature files.**  
Feature files should be user oriented and not technical such as dropdown look & feel that are common across multiple pages.

**Scenario Patterns**

1.     **Hide navigation logic inside steps when navigation validation is not the goal of a test.**   
Explicitly stating steps for each part of the navigation path to functionality under test causes scenarios to be verbose and brittle when change occurs.  Instead rely on compositing multiple smaller steps by calling them in a higher level step.

2.     **Creating steps that are composite of multiple other Step Definitions is preferred when you need to traverse a workflow.**   
Step definition "I submit a appointment request" should call into multiple other step definitions of "When I fill in the appointment form with:," "I review the appointment," and "I confirm the appointment." This promotes clear and concise scenarios.

3.     **Keep default data population in WebExtensions and not in feature files or steps.**   
Contributes to concise scenarios in the feature file and consolidates where default data population should occur.

**When scenario validation is based on state of the backend, do not worry about the details of the data being passed in, but that the UI responds correctly when the failure condition occurs.**   
Don't specify non-valuable data details when conclusions do not depend on them.  For instance if you know the platform is going to give you a failure response to creating an appointment, do not worry about the appointment details.  Errors returned from the platform will most likely be along these lines.

**Syntax**

1.     **Proper and consistent English grammar, spelling, spacing, and punctuation are to be followed.**   
Even small differences in verbiage can cause a step statement to not match or confuse the intended step meaning.

2.     ***Then* statements should reflect testing what the scenario defines.**   
Be specific that *Scenario* and *Then* statements are in agreement to what is being tested.

3.     ***Given/When/Then* statements with a following table data shall end with a colon ":".**   
All other statements shall end with no punctuation.

4.     **Table data should not have quotes**

5.     **Use consistent business language when writing the feature file.**

6.     **Feature and step definition files should not perform UI mapping.**   
UI details should be performed in the WebExtensions.

7.     **Use *Scenario Outline* when possible.**   
The permutations of the scenario can more easily be tested.

**Limitations and Restrictions**

1.     **Do not attempt to write overly generic scenarios that cover a large amount of test validation.**

2.     Concentrate on testing specific aspects to reduce complexity. Break large scenarios into small scenarios.

3.     **Do not write longer scenarios with multiple "When Then"-"When Then" patterns.**   
Concentrate on testing specific aspects to reduce complexity. Break large scenarios into small scenarios.

4.     **Only put *Given* statements in the *Background* section.**  
Starting a *Scenario* with *And* (to continue the set of *Given* statements in the *Background*) is natural and predictable. If the *And* is actually a chained *When* statement it can be confusing to maintain. Don’t refactor common *When* setup into the *Background* – leave it in each test.

**Step Definitions**

1.     **Proper namespaces to be used in step definition files to help ease test development.**  
Aids in finding the appropriate steps by preventing an overwhelming list of options.

2.     **Step definitions do not need XML comments.**  
Steps should use descriptive business language to describe their functionality.  XML documentation does not add value in this instance.

3.     **Only one *Given/When/Then* attribute may be given to a method.**  
Multiple attributes confuse the meaning of the intended usage of a method and could cause maintenance.

4.     ***Given* statements will prepare needed pre-conditions such as data and or/response setup.**  
*Given* statements have a clear job to prepare the system and data before other steps execute such as TestData.Create(). *Login as a user is the exception to this rule*.

5.     ***When* statements shall be the actor's (user, admin) actions in the UI.**  
After *Given* statements have been executed the *When* statements have the clear job of interacting with the UI's links, buttons, and other control objects just as if a user was interacting with the UI.

6.     **There will be no empty *Given/When/Then* statements except in very limited circumstances.**  
There may be very limited occurrences of this happening where an empty step is needed.  This may usually be seen as *When* steps where no input by the user to the page or is specifying a pre-condition that naturally exists is critical to the scenario.

7.     ***Then* statements are the only place assertion statements reside for validating the scenario's goal.**  
After the *When* statements have been executed validation of expected outcomes should be performed.

8.     ***Then* statements may be atomic in validation performed or composite (more than one validation point) in regards to the amount of validation performed in a step.**  
Not all scenarios will test the exact same amount of data.  Re-use should be explored if possible when creating *Then* statements to see if it makes sense to have a composite *Then* statement call into multiple single validation *Then* steps.

9.     ***Then* statements should read data from the page using the application driver extensions.**  
To ensure consistent abstraction of the UI *Then* statements should not directly access Selenium WebDriver, but should use extensions on the application driver abstraction around WebDriver.

10.  **Any Selenium technical interaction should not be present in the Step Definitions.**  
Don't access WebDriver directly. Instead, use the application driver to restrict access to the WebDriver and enforce usage of the extension methods.

11.  **Order Step Definition files by *Given* -> *When* -> *Then* sections.**  
To balance entropy and logic.

12.  **Place application login, logout, and other common steps used by all areas in a separate Step Definition file**

13.  **Abstract out waiting into application driver extensions.**  
Create application driver extensions that make use of WaitForElement() for waiting. Don't put waiting explicitly in the step definitions.

**WebDriver/Application Driver Extensions([http://docs.seleniumhq.org/](http://docs.seleniumhq.org/" \t "_blank))**

To avoid incorrect usage of WebDriver inside step definitions, the use of Selenium WebDriver should be abstracted by wrapping it with an "application driver" - effectively the same thing as the WebDriver, but without the methods for locating elements directly on the page. Doing this enforces proper abstraction in the step definitions (no direct WebDriver access) and makes it easy to add nice extensions that are business-specific.

Guidelines here are for writing extensions to the "application driver" and the use of WebDriver inside those extensions.

1.     **The order of preference for selecting page elements is to use ID if available, followed by CSS, and finally XPath.**  
To promote consistent reliable selection of controls and avoid the chaos that is an index. *Really try to avoid XPath if you can* - it's extremely slow compared to the other two mechanisms. (Think "minutes" instead of "seconds.")

2.     **Avoid Hungarian naming of methods.**  
To be prepared for future changes and to ease maintenance avoid using Dropbox, RadioButton or other suffixes that may need updating if a control is changed.

3.     **WebDriver and application driver extension methods should have proper and concise XML documentation.**  
This is to aid using the existing library of methods and attempt to reduce the amount of duplicated code.

**Keep it Clean and Refactor**

It's easy to want to add a new step definition that's similar to an existing one by copy/pasting the existing one and just "tweaking" code a bit. The problem is, this can sometimes lead to difficult maintenance problems and cause challenges when more new steps get added later on. It also makes the existing steps sometimes difficult to follow - How is that step different from this one? Why wasn't the step reused?

Where possible, consider code reuse - if there are some common things happening in multiple steps, refactor to a method that all the steps can call.

**Design patterns**

As we are using Page Objects to align with best practice in selenium webdrivers.

Source : <https://code.google.com/p/selenium/wiki/PageObjects>

**Page Objects**

Within your web app's UI there are areas that your tests interact with. A Page Object simply models these as objects within the test code. This reduces the amount of duplicated code and means that if the UI changes, the fix need only be applied in one place.

**The PageFactory**

In order to support the PageObject pattern, WebDriver's support library contains a factory class.

<https://code.google.com/p/selenium/wiki/PageFactory>