# [GDP Test Automation Framework Rules and Guidelines](https://confluence.mastercard.int/display/GDP/GDP+Test+Automation+Framework+Rules+and+Guidelines)

Given below is the list of "Standards" (Required), and "Guidelines" (Recommended), for the GDP automation Code base, that the team is following, and is expected from future Pull-Requests.

**Behavior Driven Development (BDD) Standards**

**Java Coding Standards**

**Source Code Management Standards**

Behavior Driven Development (BDD) Standards:

**What is BDD?**

*According to Dan North who introduced BDD:*

Behaviour-driven development is an “outside-in” methodology. It starts at the outside by identifying business outcomes, and then drills down into the feature set that will achieve those outcomes. Each feature is captured as a “story”, which defines the scope of the feature along with its acceptance criteria.

**Structure of a story**

Title - one sentence describing the story

Narrative:

As a [role]

I want [feature]

So that [benefit]

Or

In order to <meet some goal>

As a <type of user>

I want <a feature>

Scenario: Title of first scenario

Given [context]

And [some more context]

When  [event]

Then  [outcome]

And [another outcome]

Scenario: Title of second scenario

……………………..

**An example story:**

Story: Account Holder withdraws cash from ATM

Narrative:

As an Account Holder

I want to withdraw cash from an ATM

So that I can buy groceries

Scenario: Successful withdrawal from account with sufficient balance

Given I have $100 in my account

And the card is valid

And the ATM has more than $100

When I request $50

Then $50 should be dispensed

And the account balance should be $50

Scenario: Unsuccessful withdrawal from account with insufficient balance

Given I have $100 in my account

And the card is valid

And the ATM has more than $100

When I request $150

Then no money should be dispensed

And insufficient balance message should be displayed

**Characteristics of good stories:**

1. The title of the story should describe a feature/capability of the product/application.

2. The narrative should include the user/role, the capability/feature and the benefit/business value.

3. The scenario title should clearly indicate the purpose and should be different than other scenario titles in the story.

4. All contexts should be specified with ‘Given’ clause, action should be specified with ‘When’ clause and expected outcome in ‘Then’ clause

Given -->  Use past/present tense, passive voice

When  --> Use present tense, active voice

Then  -->  Use future/present tense, passive voice

5. Consistency in using either "first person" or "third person" in all the scenarios

**Style and Structure:**

While style often takes a backseat during [code review](https://automationpanda.com/2017/05/08/10-gotchas-for-automation-code-reviews/), it is a factor that differentiates good feature files from great feature files. In a truly behavior-driven team, non-technical stakeholders will rely upon feature files just as much as the engineers. Good writing style improves communication,  and  good communication skills are more than just resume fluff.

Below are a number of tidbits for good style and structure:

1. Focus a feature on customer needs.
2. Limit one feature/story per story file. This makes it easy to find features.
3. Limit the number of scenarios per feature. Nobody wants a thousand-line feature file. A good measure is a dozen scenarios per feature.
4. Limit the number of steps per scenario to less than ten.
5. Limit the character length of each step. Common limits are 80-120 characters.
6. Use proper spelling.
7. Use proper grammar.
8. Capitalize Gherkin keywords.
9. Capitalize the first word in titles.
10. Do not capitalize words in the step phrases unless they are proper nouns.
11. Do not use punctuation (specifically periods and commas) at the end of step phrases.
12. Use single spaces between words.
13. Indent the content beneath every section header.
14. Separate features and scenarios by two blank lines.
15. Separate examples tables by 1 blank line.
16. Do not separate steps within a scenario by blank lines.
17. Space table delimiter pipes (“|”) evenly.
18. Adopt a standard set of tag names. Avoid duplicates.
19. Write all tag names in lowercase, and use hyphens (“-“) to separate words.
20. Limit the length of tag names.
21. Remove unused imports.

**Java Coding Standards :**

1. Limit methods to 1 return statement.
2. For void methods, include a simple return statement (not all IDE's allow for setting brake-points at scope-level).
3. Follow current maven conventions.  Namely:
   1. use /src/main/java              // Main source code root.
   2. use /src/main/resources    // Main resources root.
   3. use /src/test/java               // Main test code root.
   4. use /src/test/resources      // Main test resources root.
4. Styling:
   1. Open Curly-Braces are Right-Side.
5. Use Constants with meaningful names, instead of "magic numbers".   
   <http://www.techrepublic.com/article/avoid-using-magic-numbers-and-string-literals-in-your-code/>
6. Catch Blocks:
   1. Catch something more specific than just "Exception".
   2. Do NOT SWALLOW Exceptions (empty catch blocks).
   3. If logging is needed within a catch block, it should be .error level.
7. Use sufficient logging to avoid use of .printStackTrace().   
   <https://stackoverflow.com/questions/7469316/why-is-exception-printstacktrace-considered-bad-practice>
8. Keep variable assignments to a single line.  Avoid use of C++ style assignment statements within an if-statement, Like:

if((v = someMethod()) != 0) return true;

<https://stackoverflow.com/questions/16148580/assign-variable-value-inside-if-statement>

1. Any new classes added, MUST HAVE a corresponding Unit Test (\_UT) that provides a minimum of 50% coverage, and exercises ALL public methods.
2. Use Constants & Enums instead of String Literals.
3. Use String.format for concatenation instead of the "+" plus-operator; Especially for heavily used methods.
4. Import Statements:
   1. Use FULLY-QUALIFIED package-class names.  Do not use wild card import statements, Like:

import static com.google.common.\*;

* 1. Do not use static imports, Like:

import static com.google.common.base.Preconditions.checkNotNull;

<https://stackoverflow.com/questions/420791/what-is-a-good-use-case-for-static-import-of-methods>  
<http://docs.oracle.com/javase/1.5.0/docs/guide/language/static-import.html>

1. Logging:
   1. Correct Import should be: import org.apache.log4j.Logger;
   2. Use Loggers & Appenders properly for file based, and console logging.  Do NOT USE System.out.println.
   3. For now (30-Jun-2017), define only 1 logger at class-level, we plan to implement AOP-Based Logging.

**Branch Naming**

1. Create a new branch for your changes for an iteration under: PIXX/IterationXX

2. Branch name should follow below convention:

             ALMStoryOrDefectNo\_ShortDescription

             Examples: S403975\_B2B\_Holistic\_Record\_View

                              DE77557\_AOB\_Issue\_With\_Handling\_PhoneNumbers

3. Create pull request providing details of changes to merge your branch to "development" branch.

**GDP automation technology stack :**

These are the technologies available in MTAF out of the box:

**Selenium WebDriver** - Selenium is an open source library with bindings in multiple languages (Java, C#, Python, etc) that allows an engineer to write code that is then translated into human-like interactions with various browsers and mobile devices.  At it's core Selenium spins up a lightweight server on a machine that sends commands in the JSON format to a browser or device.  These JSON commands typically include information such as the action to be performed (click, enter text, submit form, etc.) as well as information about how to identify the element for the action to be performed on.  These pieces of identifying information are based on the Document Object Model (DOM) or a web page or app and, for all intents and purposes, can be though of as the HTML of a page.

**JBehave** - JBehave is an open source BDD (Behavior Driven Development) library that allows users to write their test cases in plain English and have them automatically translated into chunks of Java code to be executed.  JBehave allows someone like a product owner or scrum master to write test cases, hand them off to automation engineers and have those engineers write the automation scripts.  JBehave also creates easily digestible and human readable reports after execution, including information such as what test cases were run, how many test cases passed/failed and provides screenshots for any failed test cases.  Everything in JBehave is customizable and flexible, giving each team the power to define their own test runs and even create custom reports.

**BrowserStack** - BrowserStack is a third party cloud provider that gives MTAF users access to thousands of OS/Browser/Version combinations.  When a user makes a request to BrowserStack they provide an OS (Windows XP - 10, OSX Snow Leopard - El Capitan), browser (Chrome, Firefox, Safari, Opera, Yandex, IE, Edge) and a browser version.  BrowserStack spins up a Virtual Machine (VM) in their private cloud with these specifications and the automation scripts are executed against them as if they were the users local desktop.  This allows Mac users to test on IE, Windows 10 users to test old versions of Chrome, etc.  BrowserStack is a paid service.  Teams can request licenses and pricing information through a Software Request Purchase in the GSM Self Service Center.

**Page Object Model** - The Page Object Model is an industry standard best practice for writing test automation scripts that allows the user to represent a page's functionality in a highly intuitive, readable format.  Page Objects allow engineers to define an application's behavior in single place and reuse it throughout their tests.  The model allows for easy and efficient updating of automation scripts as the application UI changes.  MTAF provides utilities that assist in writing these objects and take care of many of their commonly pain points users experience with them.

**Report Portal**- ReportPortal is a service, that provides increased capabilities to speed up results analysis and reporting through the use of built-in analytic features. Report Portal is an open-source tool that allows testers and key project stakeholders to manage and view all of their testing results and reports in one place. Using the tool, testers can:

* Speed product time-to-market with collaborative test automation
* View test cases with all related data in one place, right where you need it, with logs, screenshots and binary data
* Associate particular test cases with a product bug, automation issue or system issue. and so much more!

**Test Automation Project Structure:**

When you create a test project in Eclipse or IntelliJ IDEA, it is good to maintain a project structure.  Given below the GDP Test Automation project structure (UI Project):

— your-test-auto-projectName

--- **POM.xml**

--- **src/main/java**

            ---testing.gdp.ui.tests

                --aspect  
                --configuration  
                --constants  
                --dbconnection  
                --domain  
                --exception  
                --helper  
                --pages  
                --reporters  
                --steps  
                --stories  
                --utils

 ---**src/main/resources**

            ---config

                --env (dev/qa/stage)

                      --jsontestdata

                      --sqltestdata

                      --application.properties (portal url & access details, database access details, etc)

             ---stories

                --portal wise JBehave story folders

                      --Jbehave story files