**ESTÁNDAR PARA LA ARQUITECTURA DE PROYECTOS DE AUTOMATIZACIÓN DE PRUEBAS FUNCIONALES SERENITY BDD - Área de certificación**

**Contexto**

Este documento se construye con el fin de establecer una arquitectura estándar para la construcción de proyectos de automatización de pruebas funcionales con Serenity BDD. Es indispensable que la arquitectura, tecnologías, prácticas y patrones sean aplicados correctamente según los lineamientos aquí presentados para garantizar principalmente la mantenibilidad, reusabilidad y confiabilidad de los suite de pruebas que serán creados por los diferentes equipos.

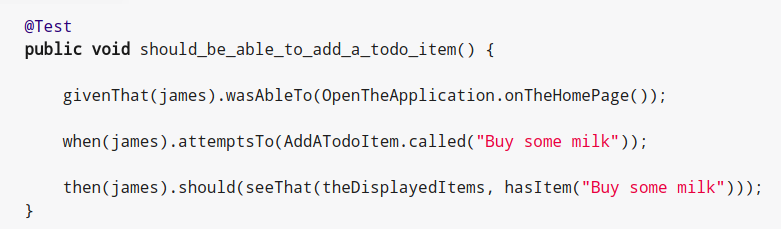
**Antes de leer este documento**

Se recomienda que el lector se documente sobre Serenity BDD en el [manual de referencia](http://thucydides.info/docs/serenity-staging/) especialmente la sección del patrón screenplay y el uso de Serenity BDD + Cucumber. Adicional a la documentación de referencia de [cucumber](https://cucumber.io/docs/reference/browser-automation#serenity) y los principios S.O.L.I.D para la construcción de buen software.

**Patrón screenplay**

Screenplay se presenta como una mejora al patrón Page Object principalmente porque usando page object, el automatizador construye código basado en la interfaz de usuario y no en las interacciones que tiene el usuario con el sistema. Este patrón, **centrado en el usuario y orientado a tareas** es la aplicación de los principios S.O.L.I.D en los proyectos de automatización, especialmente el principio de responsabilidad única y abierto-cerrado. Este patrón habilita al automatizador a diseñar dirigido en el dominio (DDD), lo que se espera se traduzca en un mayor acercamiento al negocio dado que siempre se habla y desarrolla en términos de negocio (lenguaje ubiquo). Para mayor información acerca de DDD puede dirigirse al libro de Erick Evans: *Domain-Driven Design: Tackling Complexity in the Heart of Software* o al siguiente enlace donde se resume este [enfoque](https://airbrake.io/blog/software-design/domain-driven-design).

Implementar correctamente screenplay permitirá además, escribir código en un lenguaje mucho más cercano al lenguaje natural, esto es reproducido en los reportes, por tanto se resalta la importancia de esto. Un ejemplo presente en la documentación oficial sobre la legibilidad esperada en el código se muestra en la figura 1.

****

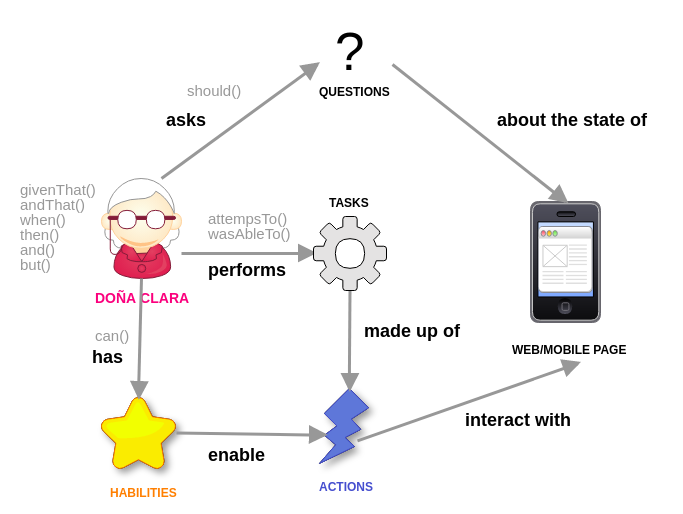
**Figura 1. Ejemplo de alta legibilidad del código**

La importancia de la alta legibilidad del código facilita principalmente la mantenibilidad y la reusabilidad dentro del proyecto, debido a que se favorece enormemente la descripción de lo que el sistema hace y lo que se está tratando de probar. Se resalta que esto hace parte de la **documentación de las pruebas automatizadas**, de hecho es la documentación más valiosa de las mismas.

Screenplay es “framework agnostic”, eso quiere decir que no tiene dependencias con la tecnología, procesos o sistemas, sin embargo, para Bancolombia será aplicado usando Serenity BDD.

Lo que se pretende con la arquitectura de referencia definida en este documento es habilitar a los equipos de automatización a construir software limpio, con buenas prácticas y que cumpla con los requisitos no funcionales de usabilidad y mantenibilidad mínimamente. También se espera poder incrementar las habilidades técnicas de los automatizadores y actualizar a nivel de tecnología y técnicas de desarrollo las pruebas funcionales que se realizan en el banco.

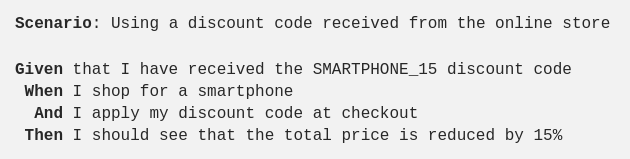
Usar screenplay obliga a pensar en términos de roles, objetivos y acciones para lograr esos objetivos. Aquello que se automatiza está en términos del proceso de negocio y no en como llegar a el (clicks y selects). El modelo se presenta en la siguiente imagen:



**Figura 2. Modelo conceptual de screenplay**

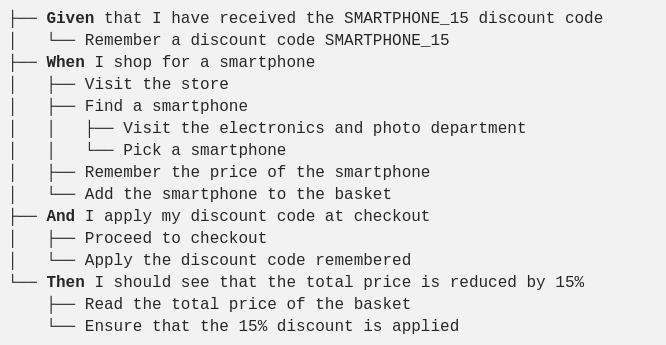
¿Cómo leer ese modelo? : Un usuario (DOÑA CLARA), habilitado para abrir el navegador ejecuta una serie de tareas (procesos de negocio como transferir saldo) a través de una serie de acciones directamente relacionadas con la aplicación (como ingresando a la aplicación, consultando el saldo, ingresando la cuenta destino y el valor a transferir) y realiza un conjunto de preguntas acerca del estado de esa aplicación (como consultar que se haya realizado el débito a su cuenta)

Consideremos un escenario como el siguiente:



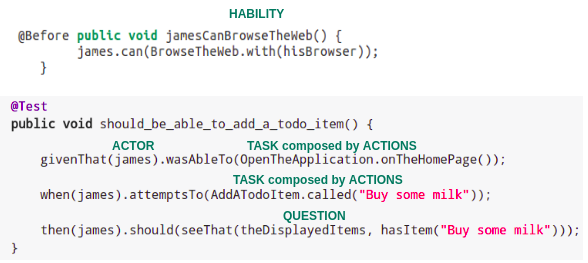
**Figura 3. Escenario de ejemplo**

Descomponiendo mentalmente ese escenario, se encuentra que ese usuario está habilitado o intenta llegar a un fin (then) a través de la ejecución de tareas (tasks) y acciones (actions)



**Figura 4. Descomposición en tareas y acciones para el escenario de ejemplo**

A nivel de código fuente, tomando el ejemplo de la figura 1, se tiene que siempre que se aplique el patrón se logrará tener un código legible en términos de negocio y no de cómo funciona la aplicación a través de clics. A esto se le llama “Layers of abstraction”.



**Figura 5. Patrón screenplay aplicado**

Para mayor información se proponen los siguientes documentos:

* [Serenity and the Screenplay pattern](http://thucydides.info/docs/serenity-staging/#_serenity_and_the_screenplay_pattern)
* [Beyond Page Object: Liberate yourself from the chains of UI-Think!](https://serenitydojo.teachable.com/blog/1017196/beyond-page-objects)
* [User-centric and task-driven: A better way to automate](https://serenitydojo.teachable.com/blog/1107150/user-centric-and-task-driven-automation)
* [User-Centred Design: How a 50 year old technique became the key to scalable test automation](https://janmolak.com/user-centred-design-how-a-50-year-old-technique-became-the-key-to-scalable-test-automation-66a658a36555)
* [Beyond Page Objects: Next Generation Test Automation with Serenity and the Screenplay Pattern](https://www.infoq.com/articles/Beyond-Page-Objects-Test-Automation-Serenity-Screenplay)

Y los siguientes videos:

* [Serenity BDD Screenplay Demonstration - Part 1](https://www.youtube.com/watch?v=MiMdiPApfDw&t=237s)
* [Serenity BDD Screenplay Demonstration - Part 2](https://www.youtube.com/watch?v=CtZB6AkggF8)
* [TestingAR XIX - Investing in Testing - Luz, cámara, Screenplay! por Rodrigo Martin](https://www.youtube.com/watch?v=79a1PdcQLBM)

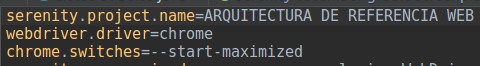
**Arquitectura de referencia**

La elección de tecnologías se hizo basado en diferentes criterios claves para la compañía, entre ellos la facilidad de uso de las herramientas de reporting, el lenguaje de programación, las capacidades para paralelización que provee y la documentación.

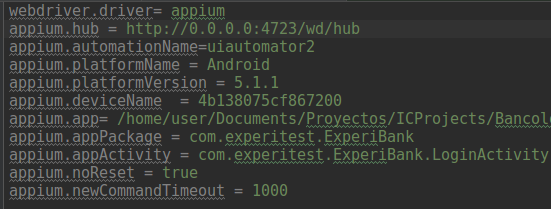
Esta arquitectura es aplicable a proyectos tanto web como mobile. Las diferencias entre cada una son las siguientes:

* El driver: Para web se usará SELENIUM, para mobile APPIUM (Cliente y servidor)
* Los capabilities:

Un ejemplo para web se presenta en la siguiente imagen:



Un ejemplo para mobile es el siguiente:



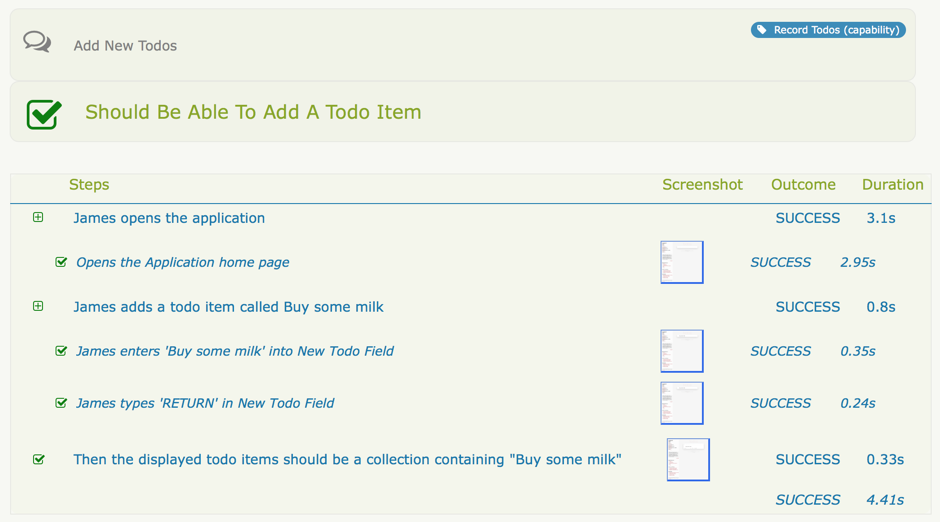
Para mayor información diríjase al siguiente link: [Appium Capabilities](http://appium.io/docs/en/writing-running-appium/caps/)

* Los recursos: Las automatizaciones para móviles requieren un recurso adicional que es la aplicación a probar (.apk o .ipa). Una práctica recomendada es que no se agregue la aplicación como recurso, sino que en el capability la ruta que se ingrese, sea una del artefacto versionado en artifactory y descargado al servidor de pruebas, de tal manera que se garantice la integridad de ese paquete (esto aplicable a automatizaciones que ya estén corriendo en un pipeline).

*Introduciendo el concepto de LIVING DOCUMENTATION*

Se le llama “living documentation” a un concepto que viene directamente de la técnica BDD y se refiere a la documentación que es creada por la herramienta de automatización basado en los suites de pruebas que se hayan automatizado cada vez que se corren. Serenity BDD es una de las herramientas más poderosas y robustas para la generación de este tipo de documentación. La documentación presenta los resultados de los tests, la descripción de los mismos con datos de entrada y el paso a paso para obtener los resultados deseados, el tiempo de ejecución de los tests, entre otras. Es generada por la herramienta según las buenas prácticas de desarrollo implementadas en las automatizaciones, por tanto se hace énfasis en que el seguimiento y la aplicación de patrones, prácticas y técnicas dentro de los proyectos es esencial para garantizar una construcción de pruebas completa y eficaz.

Un ejemplo de ello se detalla en la siguiente imagen donde se grafica cada capa de la arquitectura según el escenario que se revisó en la figura 1. Los roles, proceso, capturas de pantalla y paso a paso son agregados al reporte.



**Figura 6. Ejemplo de Living Documentation.**

Para mayor información por favor dirigirse a los siguientes enlaces:

[Living Documentation: it’s not just about test reports](https://serenitydojo.teachable.com/blog/1338817/living-documentation)

*Serenity BDD*

Es una herramienta para facilitar la codificación y ejecución de pruebas automatizadas, además de servir como mecanismo para generar el Living Documentation basado en los tests que se ejecutan. Serenity además, ayuda a construir relaciones entre personas técnicas y no técnicas a través del uso de BDD al integrarse con herramientas como Cucumber o Jbehave. Comúnmente utilizada para test automatizados y de regresión, Serenity apoya la construcción de software ágil y colaborativo entre los miembros de un equipo facilitando la comunicación entre los mismos, una de esas técnicas es a través de la ejecución de escenarios de pruebas como ejemplos de lo que haría un usuario o un cliente en el mundo real y no con generalizaciones de procesos y procedimientos. Es por esto que la herramienta elegida para la construcción de pruebas será Serenity.

Más información puede encontrar en:

* [Automated Acceptance Testing with Style](https://johnfergusonsmart.com/serenity-bdd/)
* [Introducing Serenity](http://thucydides.info/docs/serenity-staging/#introduction)

*Gherkin*

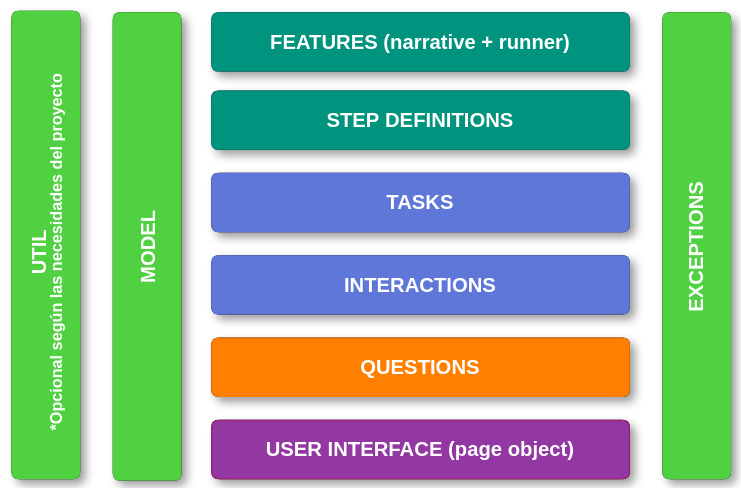
Dada la necesidad de aplicar BDD dentro de los proyectos y la intención de querer acercar más a los equipos de desarrollo, certificación y negocio, se hace necesario escoger una herramienta capaz de integrarse con Serenity BDD y que facilite la creación de escenarios de prueba mediante la construcción de los mismos en lenguaje natural e independiente de la tecnología con la cual se desarrollen los proyectos. Dos herramientas Jbehave y Cucumber hacen posible eso, sin embargo para facilitar los ejemplos que se presentarán se escoge Cucumber, que es adicional la más recomendada. Sin embargo será del libre albedrío de cada equipo el uso de una o la otra. Se recomienda Cucumber, debido a que es una herramienta más nueva que Jbehave y a pesar de eso tiene una excelente documentación, además soporta Java 8 y su comunidad es más extensa y más activa que la de Jbehave.

Para más información de comparativos y movimiento de la comunidad se puede dirigir a los siguientes enlaces:

* [Repositorio Jbehave](https://github.com/jbehave)
* [Repositorio Cucumber](https://github.com/cucumber)
* [JBehave Vs Cucumber JVM: Comparison and Experience Sharing](https://medium.com/agile-vision/jbehave-vs-cucumber-jvm-comparison-and-experience-sharing-439dfdf5922d)

*Capas de la arquitectura*

Teniendo en cuenta lo documentado en las secciones anteriores, en la siguiente imagen se presentan las capas que conformarán la arquitectura de referencia de los proyectos de automatización del área de certificación.

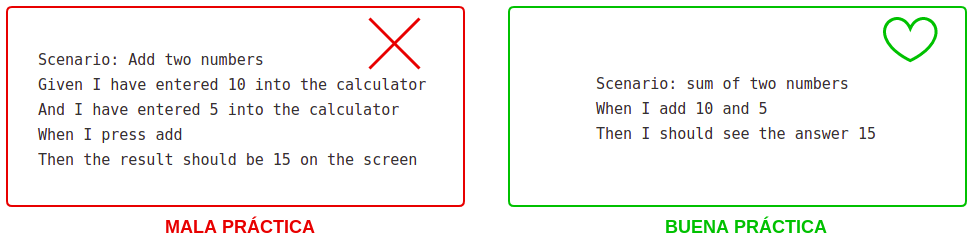


**Figura 7. Arquitectura de referencia**

Cada capa tiene una responsabilidad específica:

**Features:** Son las narrativas o .feature que equivalen a los escenarios de cucumber diseñados aplicando BDD. Esos archivos son nombrados en minúscula, separados por “\_” y pueden contener uno o más escenarios dependiendo del caso de negocio de prueba. *Ejm:* ***filtering\_client\_bank\_accounts.feature***

Se espera que los escenarios sean especificados usando lenguaje Gherkin y que estén dados en términos de lo que el usuario espera que pase más no del cómo debe hacerlo. Es decir, escribir los escenarios como un proceso de negocio de alto nivel, no en términos de clicks y selects. Para aclararlo, se presenta la siguiente imagen:



**Figura 8. Malas y buenas prácticas cuando se construye un escenario**

Más información la puede obtener en la siguiente documentación:

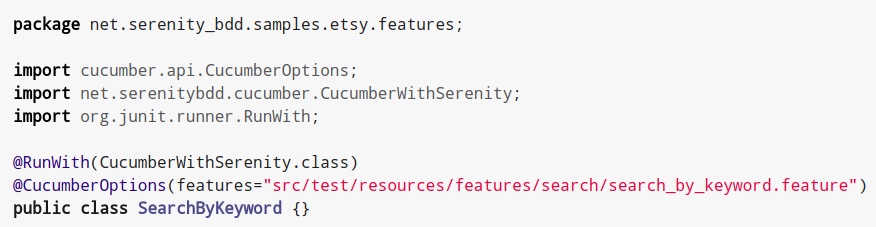
* [A BIT OF UCD FOR BDD & ATDD: GOALS -> TASKS -> ACTIONS](http://antonymarcano.com/blog/2011/03/goals-tasks-action/)
* [Browser Automation - Serenity](https://cucumber.io/docs/reference/browser-automation#serenity)
* [Serenity with Cucumber](http://thucydides.info/docs/serenity-staging/#_serenity_with_cucumber)

**Algo muy importante** y un caso muy frecuente es el de reuso de escenarios, para esto se propone no reusar a nivel de narrativas sino de tareas. Para hacer una mayor aclaración por favor diríjase a la siguiente documentación:

[How not to prepare test data in JBehave and Cucumber](https://serenitydojo.teachable.com/blog/1104192/preparing-test-data-in-jbehave-and-cucumber)

Adicional a esto representan los runners o archivos que serán ejecutados por Serenity cuando corran la pruebas. Estos archivos mapean a las narrativas y por cuestiones de legibilidad y mantenibilidad se aconseja que vayan nombrados de la misma manera en la que se nombraron las narrativas a las que mapean pero en Camel case. Para el ejemplo anterior el runner se nombraría así: ***FilteringClientBankAccounts.java***

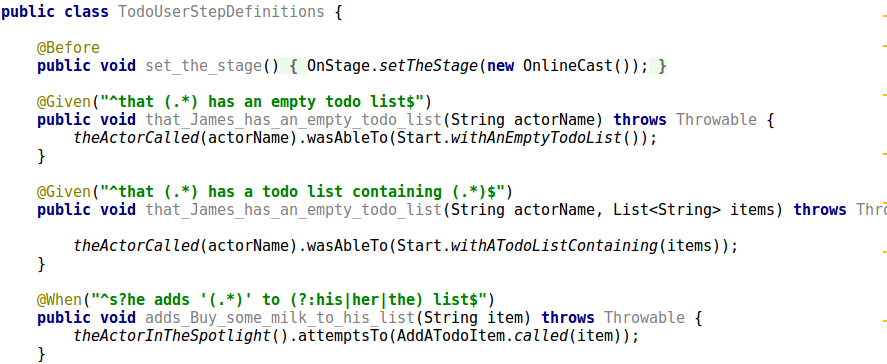
Un ejemplo de un runner se presenta en la siguiente imagen:



**Figura 9. Clase ejecutable de suite de pruebas.**

**Step Definitions:** Son clases que mapean cada línea de los escenarios definidos en lenguaje Gherkin a métodos java. Por cuestiones de legibilidad y mantenibilidad se aconseja que vayan nombrados de la misma manera en la que se nombraron los runner agregando StepDefinitions al final. *Ejm:* ***FilteringClientBankAccountsStepDefinitions.java***

Un ejemplo de un step definition se presenta en la siguiente imagen:

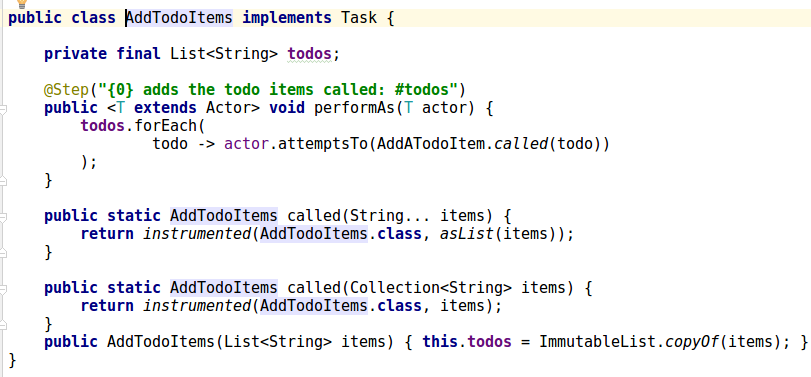


**Figura 10. Ejemplo de step definition.**

En el @Before se especifican las habilidades (abilities) del actor, lo que en términos técnicos indica que las habilidades representan el arrange de una prueba o la preparación de la misma. Dentro de cada mapeo de líneas Gherkin, se hace referencia a las tareas (tasks) no a las acciones (actions) dado que las tareas tal y como se especifica en el modelo de la figura 2 están en un nivel de abstracción más alto que las acciones.

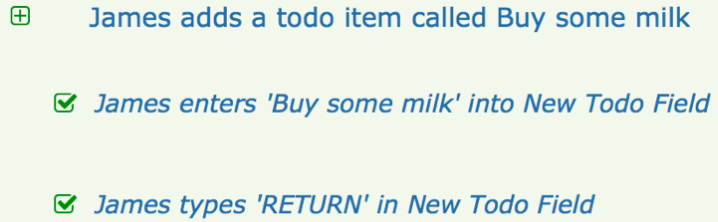
El uso de expresiones regulares es importante cuando se quiere establecer un reuso de sentencias que varían en parámetros o algunas palabras. Por tanto se recomienda su uso.

**Tasks:** Son tareas a un nivel de granularidad más alto al de clicks y selects. Representan operaciones importantes para llegar a cumplir una meta (lo que se está probando). Ejm consultar saldo, hacer transferencia, completar un pago, matricular cuentas. Las tareas, al ser operaciones, son verbos y así mismo serán nombrados, además implementan de Task. Es necesario aquí, aplicar el principio de única responsabilidad, es decir, una tarea es representada por una clase, una clase no debe ser usada para representar 2 o más tareas. Una tareas puede tener uno o más métodos nombrados basados en el patrón objectBuilder para facilitar la legibilidad de los tests cuando éstos sean llamados. Una tarea puede estar compuesta por otras tareas o por acciones. Un ejemplo de una tarea se presenta a continuación:



**Figura 11. Ejemplo de una tarea.**

Cuando una tarea es compuesta por otras tareas o por acciones, en el reporte, ese paso a paso se verá reflejado.



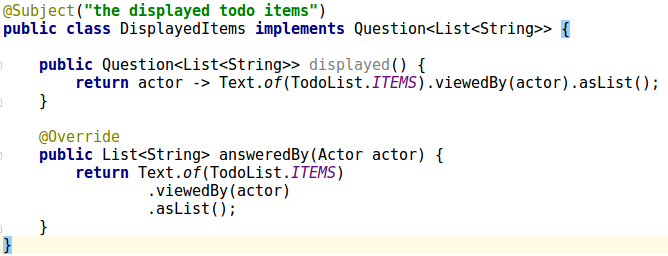
**Figura 12. Los reportes muestran las relaciones entre tareas y acciones.**

**Interactions:** Una interacción representa una acción directa del usuario con la interfaz como ingresar datos en campos o dar clics en botones. Serenity screenplay trae por defecto muchas interacciones sobre la interfaz, por tanto, no siempre es necesario crear nuevas interacciones. Un ejemplo de una interacción se representa en la siguiente imagen:

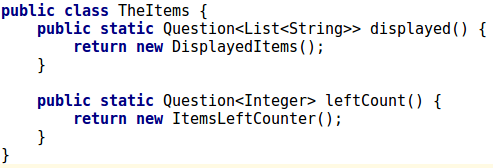


**Figura 13. Ejemplo de una acción.**

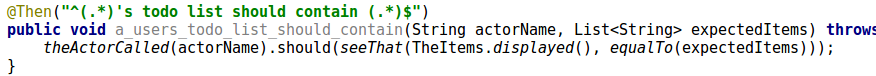
**Questions:** Es la capa donde se especifica el assert de las pruebas. Donde se verifican los resultados de las operaciones realizadas en las capas anteriores. Serenity usa una librería que se llama Hamcrest para la construcción de assertions fluidos, esta librería trae muchos por defecto, en caso de que haya un assertion que sea muy específico del negocio entonces se crearía en la capa questions. Un ejemplo se presenta en la siguiente imagen:



**Figura 14. Ejemplo de una clase question.**



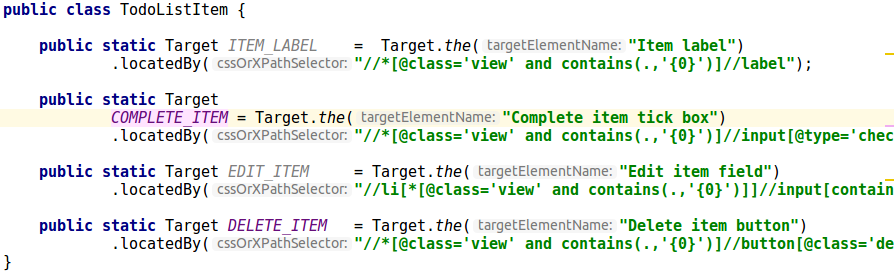
**Figura 15. Clase que servirá como Factory para crear Questions de objetos de negocio llamados Items.**



**Figura 16. Uso del question.**

Es válido el uso de patrones para mantener la legibilidad y simplicidad de las oraciones dentro de los steps definitions.

**User interface:** Son las clases que mapean los componentes de una interfaz de usuario. Sólo deben realizar ese mapeo, no tienen comportamiento. Se nombran basados en el contenedor de lo que se está mapeando en mayúscula separado por guión bajo. Ejemplo:



**Figura 17. Ejemplo del mapeo de una interfaz de usuario.**

**Capas transversales o de uso general en el proyecto:**

**Util:** Una capa opcional que será creada a criterio de cada equipo en caso de que hayan utilidades que consideren se puedan reusar.

**Model:** Una capa donde se encontrarán todos los objetos complejos de negocio, como personas o cuentas bancarias para ser usados dentro de cualquier capa del proyecto.

**Exceptions:** Una capa donde se crearán las excepciones específicas que permitirán la legibilidad de los reportes cuando las pruebas fallen. La razón de ser de esta capa es porque normalmente las excepciones del web driver son ambiguas y generalizadas, esto hace que no entreguen mucha información acerca del estado real de la aplicación en el momento de la ejecución de los test. Un correcto manejo de las excepciones asegurará un reporte con errores específicos y bien descritos (no técnicos) para ser presentado como documentación ante entidades reguladoras.

Dentro de los tipos de excepciones, se categorizan específicamente tres:

* Fallas (failures): Indican que la aplicación no se está comportando como es esperado. Excepciones de tipo AssertionError son categorizados como fallas. Se presentan en color rojo en la documentación viva.
* Errores (errors): Indican un error potencial de la implementación del test en si mismo. Normalmente las excepciones del webDriver se categorizan como errores. Se presentan en color naranja en la documentación viva.

Excepciones como ElementNotVisibleException son excepciones del webDriver que serán tomadas como errores. Ese tipo de excepciones no siempre son errores, pueden ser lanzadas como resultado de una falla en la aplicación. En este caso, para que esa excepción sea lanzada como falla, es necesario construir una excepción propia de dominio que extienda de una que si sea una falla, como por ejemplo de AssertionError y controlarla en los posibles puntos en los que pueda ser lanzada. Eso además de mejorar los mensajes de error que genera Serenity tran las ejecución de las pruebas, también permitirá a las automatizaciones ser más asertivas en términos del tipo de error lanzado.

* Comprometidos (compromised): Indican que una excepción particular presentada durante la ejecución de la prueba está relacionada a un problema con la estabilidad del ambiente o algo que vaya más allá del alcance de los tests. Se presentan en color púrpura en la documentación viva. Para identificar que tipos de excepciones generan que las pruebas queden con estado comprometido, se debe agregar al serenity.conf una propiedad con las excepciones, de la siguiente manera:

serenity.compromised.on=org.openqa.selenium.NoSuchFrameException

Dentro de screenplay hay un método llamado orComplainWith() que se puede usar para manejar excepciones de una manera personalizada y que respete el principio de que las pruebas deben ser documentación y su uso es como el siguiente:



Para conocer más sobre el manejo de excepciones por favor dirigirse a la siguiente documentación: [Domain-Specific Exceptions with Serenity BDD](http://thucydides.info/docs/articles/semantic-exceptions.html)**.**

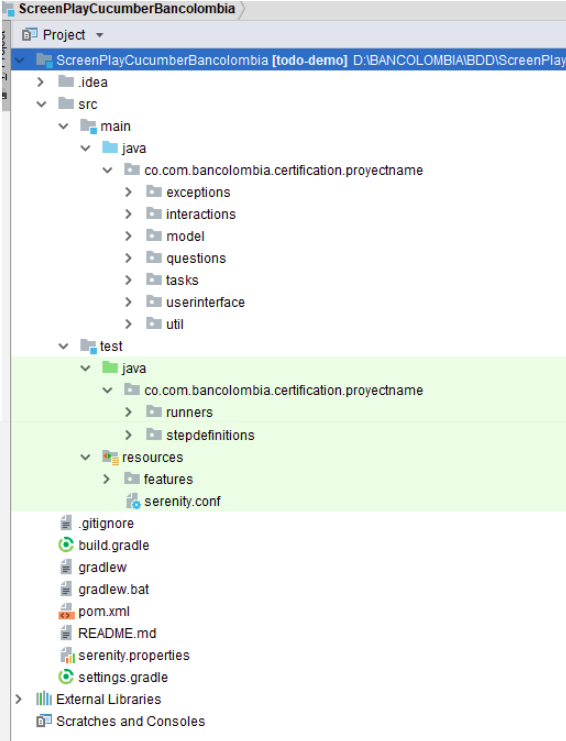
***Nota*: El nombramiento de objetos, clases, métodos, etc debe obedecer a la búsqueda del mejoramiento de la legibilidad de los suites de pruebas y por tanto del reporte. Un recomendación especial es realizar los desarrollos en inglés, ya que screenplay está desarrollado en inglés y los métodos y clases que de el se usen aparecerán en inglés en el reporte, esta recomendación se hace con el fin de estandarizar el lenguaje.**

**Recomendaciones para interactuar con componentes o puntos de control que no tengan interfaz de usuario.**

Se recomienda que para este tipo de componentes, si no tienen las debidas pruebas unitarias y/o de integración, crear un jar con las validaciones y acciones que se realizan con los mismos y desde una o varias interacciones o questions (cuando sea requerido) y según su responsabilidad, llamar ese jar y así lograr abstraerse de la implementación de componentes de bajo nivel. Se sugiere que el .jar esté versionado en artifactory, sea agregado como dependencia en maven o gradle y versionado en .git en un proyecto a parte.

**Arquetipo de referencia**

Los proyectos que cumplan con esta arquitectura, deberán seguir las prácticas y técnicas en este documento relacionadas, adicional al arquetipo de referencia que se grafica en la siguiente imagen:



**Figura 18. Arquetipo de referencia.**

Para conocer el funcionamiento a nivel de código fuente del patrón, puede interactuar y estudiar el ejemplo presente en el siguiente repositorio: [Screenplay Pattern with Serenity BDD](https://github.com/serenity-bdd/screenplay-pattern-todomvc)

**Data Driven**

La construcción de datos se recomienda sea directamente en los archivos .feature cuando la proporción de estos es pequeña (hasta 6 filas y/o 6 columnas de datos).

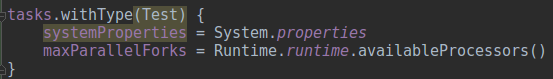
Cuando el número de datos es mayor, se aconseja usar un external data, puede ser cualquier formato (xml, json, xls, etc)

Se aconseja realizar un análisis previo de los datos que se ingresan a la automatización de tal manera que sólo vayan los estrictamente necesarios en el feature o external data (cuando se requiera). Ejemplo:

*Puede existir un escenario de una transacción que requiera la creación de una persona natural. Datos como el nombre, el peso, la estatura, la dirección y el cónyuge importan pero no deben tener un valor estricto enviado desde el escenario. Sin embargo, el tipo y número de documento si. Entonces para evitar enviar a la automatización todos los datos desde el .feature, los que no son tan vitales, se aconseja cargarlos desde un builder, pojo o cargados desde una base de datos de tal manera que el .feature sólo contenga la información que se requiera obligatoriamente para la ejecución de los casos.*

**Paralelizando la ejecución de pruebas**

El proceso de paralelización en la ejecución de pruebas puede solventar los costos en la ejecución de las mismas. Esto requiere la intervención de un buen manejo de tareas y asignación de recursos dentro de los pipelines en VSTS, apoyo que será dado o delegado por el área de devops. Adicional a la intervención del automatizador en la definición de cuantos hilos por JVM serán levantados para la ejecución de pruebas. Lo único que se debe hacer es agregar al build.gradle las siguientes líneas:



Se recomienda que cada proyecto de automatización las contenga y la cantidad de hilos levantada dependerá de los recursos disponibles de las máquinas en donde se corran las pruebas y es asignado automáticamente.

Si se está usando maven en el proyecto, la configuración es la siguiente dentro del plugin surefire. Parece ser que no hay una forma sencilla de encontrar los procesadores disponibles en maven así como en gradle, sin embargo es posible usar el valor “1C” que es exactamente lo mismo que realiza la función de availableProcessors().

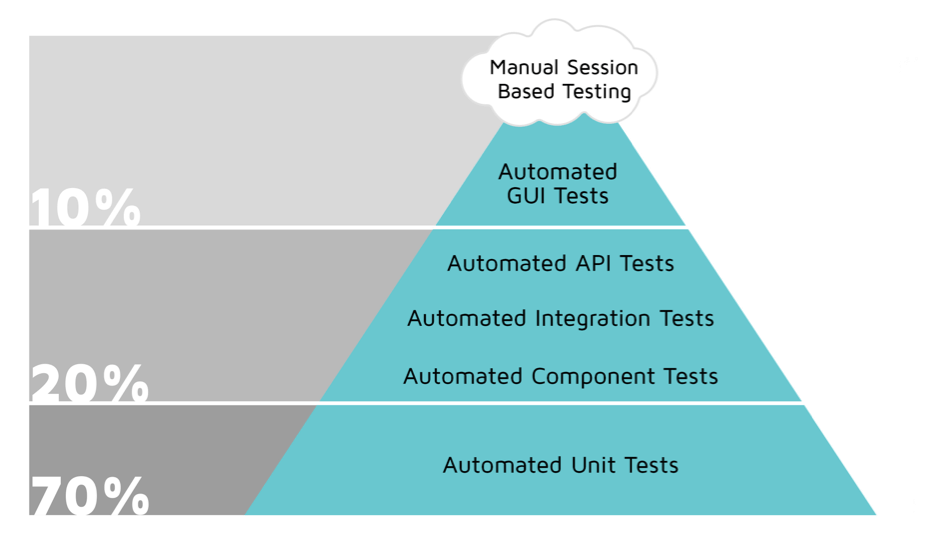


Para más información del paralelismo con maven, por favor diríjase a la siguiente documentación: [Fork Options and Parallel Test Execution](http://maven.apache.org/surefire-archives/maven-surefire-2.15/maven-surefire-plugin/examples/fork-options-and-parallel-execution.html)

**Buenas prácticas**

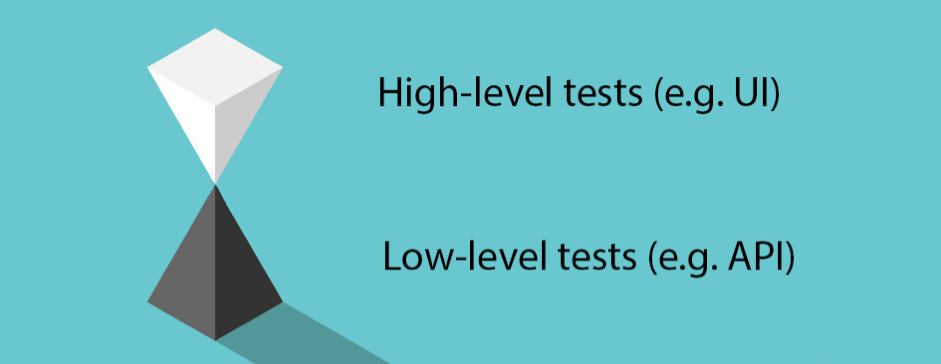
Se recomienda tener siempre en cuenta:

* No recargar las automatizaciones solo en las de interfaz de usuario. Siempre se debe intentar mantener una proporción entre tests y una buena manera de hacerlo es teniendo en cuenta la pirámide de pruebas de Cohn:

****

**Figura 19. Proporción de pruebas automatizadas según la pirámide de pruebas de Cohn.**

* Las pruebas unitarias y de integración por naturaleza son más estables entre los tipos de pruebas de la pirámide, debido a que son más granulares y exhaustivas, además se ejecutan con mayor periodicidad. Así que como certificador, es importante apoyar a los equipos de desarrollo y alentarlos en la construcción de más pruebas unitarias y de integración. En la siguiente imagen se ilustra la proporción de estabilidad de pruebas de alto nivel (blanco) comparado con las de bajo nivel (negro)

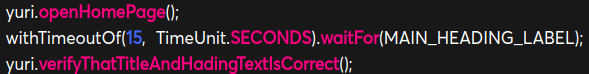
****

**Figura 20. Proporción de estabilidad de pruebas basado en el tipo de prueba.**

* Evitar completamente el uso de la función Thread.sleep(): Por naturaleza una prueba funcional es costosa en términos de tiempo, agregar líneas como Thread.sleep() lo único que hará es alargar esos tiempos de ejecución e ir en contra de mantener la estabilidad del suite de pruebas, Las herramientas mencionadas en este documento ejecutan en milisegundos las pruebas, por tanto no es necesario hacer “una pausa” del set. Por defecto, Screenplay espera a que la página termine de cargar completamente, por lo que el uso de waits no es muy necesario. Las opciones son las siguientes, se organizan según el valor recomendado que tengan siendo la primera la más recomendada y la última la menos recomendada:
  + Evitar el uso de wait hacia algún estado particular de un evento, reemplazarlo por WebElementStateMatchers de la siguiente manera: *then*(dana).should(*seeThat*(*the*(ProfilePage.*NAME*), WebElementStateMatchers.isVisible()));
  + Si se trata de algún cambio asíncrono en la página, para el cual no se sabe cuando puede aparecer, existe un método llamado “eventually” que es una manera discreta de alcanzar un wait. *then*(dana).should(eventually(*seeThat*(*the*(ProfilePage.*NAME*), WebElementStateMatchers.isVisible())));
  + Usar los métodos expuestos por screenplay para esperar hasta que cierta condición se cumpla: WaitUntil.the(LOGIN\_BUTTON, expectedState)
  + Uso de waits implícitos y explícitos. Esta es la menos recomendada porque si no se controlan los tiempos correctamente se puede incurrir en problemas de lentitudes o envolver problemas técnicos de la aplicación, además de que la forma en la que se construyen las sentencias no va de acuerdo a lo planteado según el uso de screen play. El wait implícito se usa para ordenar la espera hasta que aparezca cierto elemento en la pantalla.



Un wait explícito se usa para especificar una condición de espera.



Para mayor información se puede dirigir a la siguiente documentación: [The serenity manual - Working with timeouts](http://thucydides.info/docs/serenity-staging/#_working_with_timeouts) y [Comentarios de Ferguson ante el uso de waits](https://groups.google.com/forum/#!topic/thucydides-users/zxvX6TZCbNs)

* Siempre nombrar mnemotécnicamente variables, clases, métodos y especialmente los tests. Los tests sólo por su nombre deben mostrar que hacen y cual es el resultado. Un ejemplo de un nombramiento incorrecto es **flightSearchTest.** Un correcto nombramiento debería ser **userShouldBeAbleToFindFlightsFromBostonToNewYork** o **user\_should\_be\_able\_to\_find\_flights\_from\_Boston\_to\_New\_York.**
* No agregar comentarios a ninguna sección de código. Cada desarrollo debe ser lo suficientemente simple y claro como para que un nuevo automatizador sepa que hace y que se espera que resulte de ahí. Agregar comentarios es una mala práctica y lo único que dice es que es tan poco entendible el código que es necesario explicarlo.
* La duplicidad de código no está permitida y es considerada siempre una mala práctica de desarrollo. El uso de técnicas básicas de refactoring es apropiado para evitar caer en este tipo de errores.
* Los tests siempre deben ser independientes, un test no debe afectar la ejecución de otro. El uso del @Before es recomendado para la preparación de cada test.

**Tips**

Algunos tips que se consideran importantes para ser aplicados dentro de los proyectos están relacionados a:

* Paralelismo: Visite adicional a la información dada en este documento el siguiente post: [Running parallel tests in Cucumber with Serenity BDD](https://johnfergusonsmart.com/running-parallel-tests-serenity-bdd-cucumber/)
* Interacciones comunes con las páginas web: [Interacting with Web Pages using the Serenity BDD Journey Pattern](http://thucydides.info/docs/articles/ui-interactions.html)
* Assertions múltiples en un test (se recomienda tener cuidado con esto, porque no es una buena práctica, sin embargo en algunas ocasiones es necesario): [Soft Asserts using the Serenity BDD Journey Pattern](http://thucydides.info/docs/articles/soft-asserts.html)
* Mobile automation: [Running tests on Mobile Devices with Appium](http://serenity-bdd.info/docs/serenity/#_running_tests_on_mobile_devices_with_appium)