Lineamiento Técnico Patrón de Diseño Screenplay
Q-Vision Technologies

Q-Vision Technologies S.A.

Colombia

LAS COPIAS IMPRESAS NO ESTÁN CONTROLADAS
Uso INTERNO

TABLA DE CONTENIDO

HISTORIAL DE VERSIONES Y REVISIONES

Versión	Fecha	Acción	Estado	Descripción	Responsable
1.0	26/06/2020	C	Creación	Creación del documento y la definición de su contenido	Carlos Andrés Guerrero Daybirth Zapata
1.0	30/06/2020	A	Aprobado	Aprobación Documento	Daybirth Zapata

Descripción de valores para el campo Acción

C – Creación del documento
D – Distribución del documento
M – Modificaciones del documento
R – Revisión del documento
A – Aprobación del documento.

INTRODUCCIÓN.

Descripción del Lineamiento Técnico.

En la construcción de proyectos de automatización es importante seguir lineamientos técnicos y buenas prácticas de desarrollo, que permitan tener un proyecto con altos estándares de codificación y a su vez fácil de mantener. Esto se logra con la implementación del patrón de diseño Screenplay.

Alcance.

El siguiente es un documento guía en el que se establecen los lineamientos y estándares definidos por QVision Technologies, para la creación de proyectos de automatización bajo el patrón de diseño Screenplay, entre ellos se encuentran: Creación del arquetipo base, estándares de nombramiento y buenas prácticas a nivel general.

Los lineamientos contenidos en este documento, pueden ser aplicados a automatizaciones de tipo web, mobile, consumo de API's, AS/400, automatización de escritorio en conjunto con otras herramientas como Winium, WinAppDriver o AutoIT.

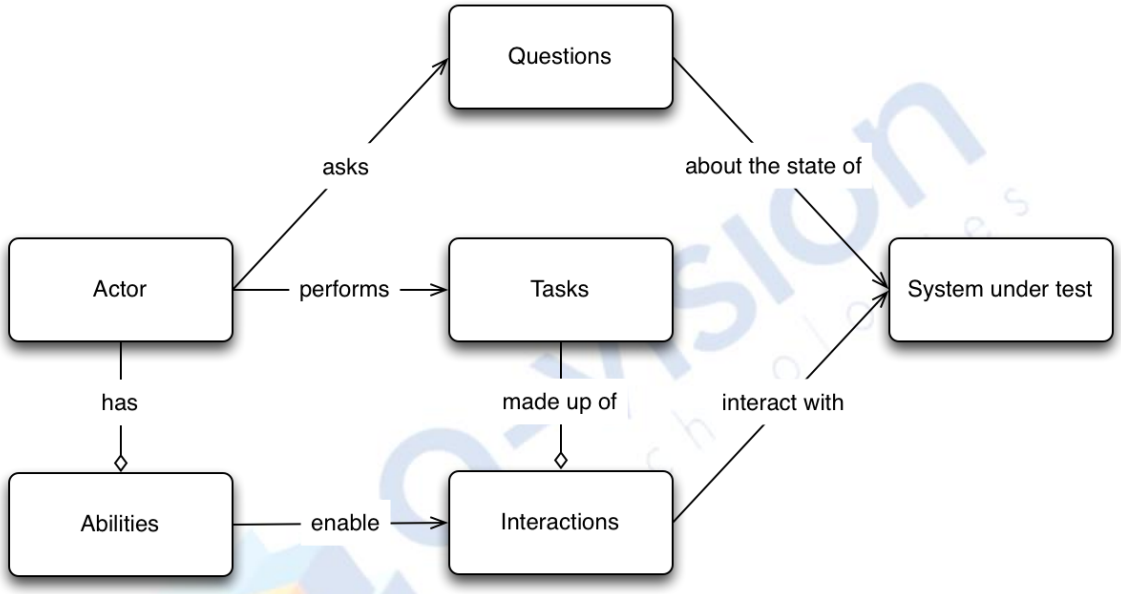
Restricciones y/o Limitaciones.

Los estándares y lineamientos técnicos descritos en este documento no serán aplicables en aquellos casos en los que el cliente ya cuente con su propia definición, en caso de no tenerlos, el analista podrá recomendar los lineamientos definidos en este documento.

1. DEFINICIÓN E IMPLEMENTACIÓN DEL PROYECTO.

1.1 Patrón de diseño Screenplay.

Es un patrón de diseño con un enfoque de desarrollo guiado por comportamiento (BDD). Brinda gran facilidad en la lectura y reutilización de artefactos. Está orientado al cumplimiento de buenas prácticas de desarrollo aplicando principios SOLID de la programación orientada a objetos.



1.2 Descripción de las herramientas.

Para la construcción del proyecto se debe hacer uso de las siguientes herramientas: Cucumber, Junit, Gradle, Sonarlink, SonarQube Serenity BDD*.*

- **Cucumber**: Es una herramienta que permite interpretar historias de usuario escritas en lenguaje Gherkin y convertirlas en código ejecutable en la automatización. Para esto se utilizan ficheros con extensión “Feature”.

- **Junit**: Es un conjunto de librerías para la creación de pruebas unitarias para aplicaciones JAVA.

- **Sonarlint**: Es una extensión para IDEs que permite detectar y corregir errores (de escritura, estándares de nombramiento, complejidad del código, entre otras) mientras se escribe código.

- **SonarQube:** Es una plataforma para evaluar código fuente, usa diversas herramientas de análisis estático de código fuente como Checkstyle, PMD o FindBugs para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.

- **Serenity BDD**: Serenity BDD es un framework para la automatización de pruebas de regresión y aceptación, que permite usar los resultados de éstas para producir informes narrativos ilustrados que documentan y describen lo que hace la aplicación y cómo funciona.

- **IDE**: Entorno de programación a elección del analista, se recomienda Eclipse e IntelliJ IDEA.

1.3 Arquetipo del proyecto.

Un arquetipo expresa un modelo sobre el que se pueden desarrollar tareas que son de un mismo tipo. Puede decirse que son plantillas parametrizadas o configuradas para utilizar determinadas tecnologías que los desarrolladores utilizan como base para escribir y organizar el código de la aplicación.

Que todos los proyectos que involucren ciertas tecnologías partan de una base (arquetipo, plantilla o esqueleto configurado) común, tiene ventajas evidentes:

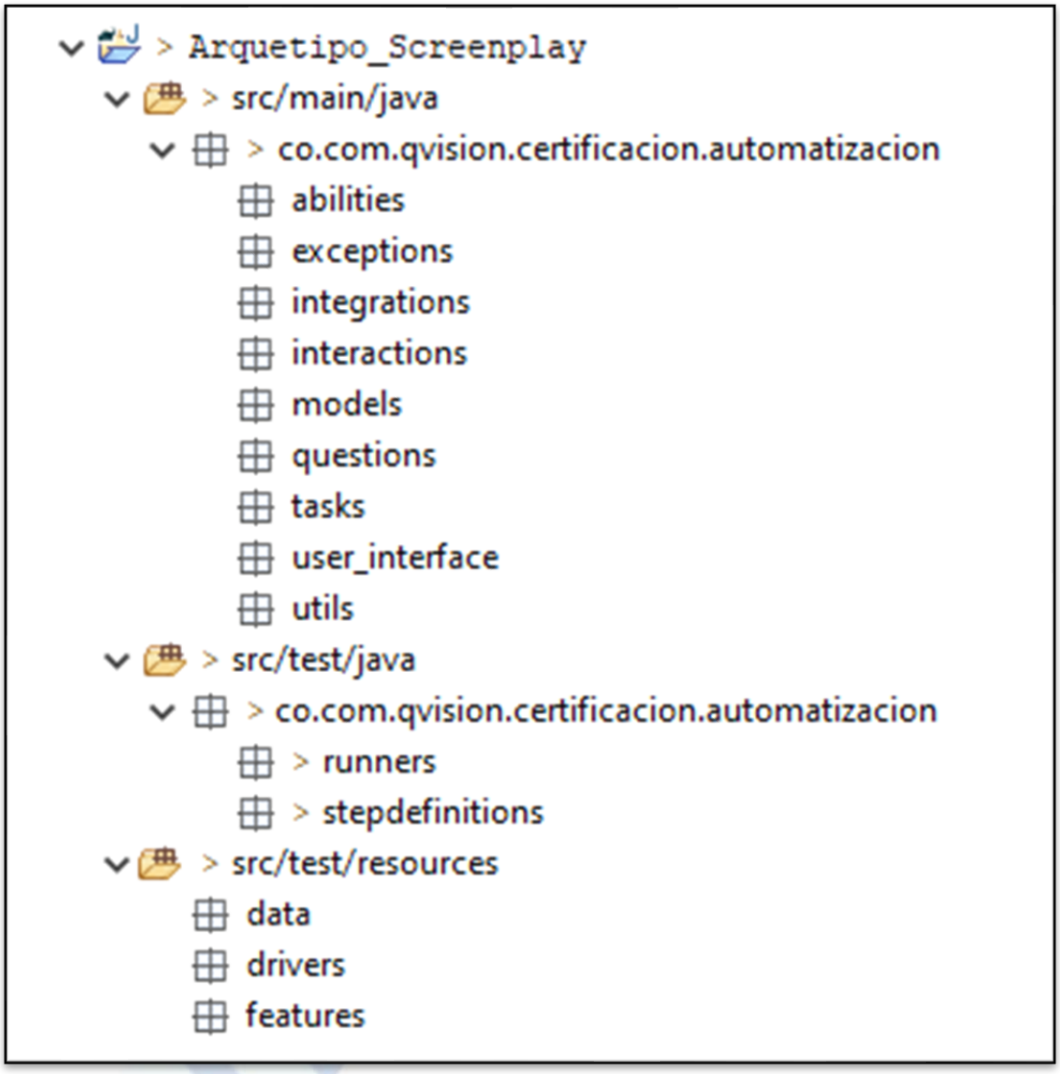
- **Homogeneidad** entre distintos desarrollos que utilizan las mismas tecnologías.
- **Reutilización** y construcción de unos arquetipos como suma de otros.
- **Estandarización** de los proyectos dentro de una organización. La homogeneidad en la estructura del proyecto facilita las tareas de desarrollar, desplegar y mantener el proyecto generado.
- **Reducción del tiempo** necesario para la construcción de los diversos servicios.

Se definirá a continuación el arquetipo mediante el cual se estructurará el patrón de diseño Screnplay mediante paquetes:

src/main/java	**src/test/java**	**src/test/resources**
Exceptions Interactions Models Questions Tasks Userinterface Utils Integrations Abilities	Runners Stepdefinitions	Drivers Features Data

1 Arquetipo screenplay.

Ejemplo: ver página 30. Estándares generales.



1. Arquetipo screenplay.

1.4 Implementación del patrón screenplay:

- **src/main/java:** En términos del proyecto de automatización de pruebas, en estos paquetes se alojan las clases que hacen referencia a las distintas capas del patrón de diseño. Son clases que apoyan o sirven para la ejecución de las pruebas, a continuación:

  - **Abilities:** en este paquete se deben crean clases que contienen las habilidades que puede tener un actor, tales como la habilidad de conectarse a una base de datos entre otras.

    Estas deben ser nombradas en PascalCase y el nombre debe ser diciente y respetar el principio de interfaz fluida. Ejemplo: `actor.can(ConnectToTheDatabase)`

```java
package com.qvision.screenplay.abilities;

import net.serenitybdd.screenplay.Ability;

public class QueryDatabase implements Ability {
    private final String urlDatabase;
    private final String username;
    private final String password;

    private QueryDatabase(String urlDatabase, String username, String password) {
        this.urlDatabase = urlDatabase;
        this.username = username;
        this.password = password;
    }

    public static QueryDatabase configDatabase(String urlDatabase, String username, String password) {
        return new QueryDatabase(urlDatabase, username, password);
    }

}
```

Ver tema 3.4 del curso Screenplay en QVUniversity.

- **Exceptions:** En esta capa se crearán excepciones específicas de dominio que permitirán tener reportes legibles, debido a que, estas le dan significado a los fallos del aplicativo que se esté automatizando. Normalmente las excepciones generadas por el framework de pruebas junto con los componentes que lo conforman son de carácter general y ambigua.

  El uso de las excepciones permitirá tener un reporte con errores específicos y correctamente descritos y ahorro en tiempo de depuración de código.

  El estándar de nombramiento de estas clases sugiere que deben ser dicientes para identificar el tipo de error que se quiere controlar, las palabras deben ser escritas con forma PascalCase y al final del nombramiento debe

tener la palabra Exception. Ejemplo: NoSeEncuentraTipoDocumentoException.

En el framework de pruebas se encontrarán los siguientes tipos de excepciones:

- **Fallas:** indican que la aplicación no se ha comportado como se esperaba, estas son arrojadas cuando se realizan validaciones de negocio, estas deben heredar de AssertionError. Son marcadas en el reporte de pruebas en color rojo indicando un posible bug.

  Ejemplo:

```
public class CodigoRespuestaServicioError extends AssertionError {

    public static final String CODIGO_RESPUESTA_SERVICIO = "El código de respuesta obtenido no es igual al esperado";

    public CodigoRespuestaServicioError(String mensaje, Throwable causa) {
        super(mensaje, causa);
    }
}
```

  Implementación:

```
theActorInTheSpotlight()
.should(
    seeThat(CodigoRespuestaServicio.obtenido(), equalTo(HttpStatus.SC_OK))
        .orComplainWith(CodigoRespuestaServicioError.class, CODIGO_RESPUESTA_SERVICIO));
```

- **Excepciones semánticas:** estas pueden indicar un mal comportamiento en la implementación de la prueba automatizada, generalmente son las excepciones arrojadas por el webDriver. Se debe tener en cuenta que estas también pueden detectar fallas en el aplicativo, por lo tanto, se debe realizar un análisis que permita identificar donde estas excepciones ocurren con frecuencia para controlarlas mediante excepciones de dominio que hereden de la clase Exeption, estas deben ser utilizadas en la implementación de las tareas. Son marcadas en el reporte de color anaranjado.

```
public class ActorNoPuedeConectarseBaseDatosException extends RuntimeException {

    public ActorNoPuedeConectarseBaseDatosException() {

        super("El actor no tiene la habilidad para conectarse a la base de datos");
    }
}
```

- **Comprometidos (compromised):** Estas excepciones indican que un elemento externo que no pertenece al alcance de la prueba afecto el comportamiento de la misma. Son utilizadas para indicar fallos o inestabilidad del ambiente en el cual están ejecutándose las pruebas, por ejemplo: cuando la conexión a la base de datos no está disponible. Son marcadas en el reporte de color purpura y se deben configurar en el serenity.conf o en el serenity.properties separadas por comas de la siguiente manera:

  serenity.compromised.on= java.sql.SQLException, [otra excepción].

- **Integrations**: Esta es una capa opcional, que se debe utilizar para crear integraciones con sistemas externos y subsistemas del aplicativo que se esté automatizando, como manejo y administración de base de datos y conexiones de otra índole. Se debe tener en cuenta que para la creación de estas clases se debe realizar una abstracción tal de la implementación que al momento de utilizarlas sea muy sencilla su manipulación.

- **Interactions:** En este paquete se crean clases que describen como interactúa el usuario con la aplicación para realizar una acción específica. Como dar Click, Seleccionar una lista, Scroll a una página, realizar switch to frame etc. En estas clases se permite la utilización de lógica de programación y la utilización de interacciones de screenplay para resolver interacciones complejas. También es posible convertir Targets en WebElementFacade para más interacciones con los objetos.

  Estas clases deben implementar de la interfaz `Interaction` o heredar de clases derivadas de interacciones de screenplay.

  Para el nombramiento de estas clases se utiliza el estándar general de JAVA para clases, teniendo en cuenta que se debe respetar el principio de interfaz fluida en su implementación.

Ejemplo:

```
public class Cargar implements Interaction {

    @Override
    public <T extends Actor> void performAs(T actor) {
        do {
            actor.attemptsTo(Click.on(BOTON_MOSTRAR_MAS_VUELOS));
        } while (WebElementQuestion.the(BOTON_MOSTRAR_MAS_VUELOS).answeredBy(actor).isVisible());
    }

    public static Cargar losVuelosDisponibles() {
        return instrumented(Cargar.class);
    }
}
```

Esta clase llamada Cargar implementa de la interfaz Interaction, en la cual se realiza la interacción de realizar clic en un botón hasta que este mismo deje de ser visible permitiendo cargar todos los vuelos disponibles en una página de reservas. Se observa también que respeta el principio de interfaz fluida, ya que al implementarla se llamara de la siguiente manera Cargar.*losVuelosDisponibles*().

- **Models:** Se crean clases que representan abstracciones de objetos que hacen parte de la aplicación o del negocio. Por ejemplo: Usuario, Cuenta, Inmueble, etc. Estos objetos poseen atributos donde se almacena la información que es utilizada para completar tareas e incluso validaciones.

  Estas clases pueden ser construida solo con Getters ya que pueden ser seteadas desde el stepDefinitions por referencia, también es muy útil construirlas mediante el patrón builder.

  Para el nombramiento utilizaran el estándar general de Java, los nombres deberán ser nemotécnicos de tal forma que se indique con el nombre que tipo de información contendrá la clase.

  Ejemplo:

```java
public class InformacionPersonal {

    private String nombres;
    private String apellidos;
    private String fechaDeNacimiento;
    private String tipoDeIdentificacion;
    private String identificacion;
    private String movil;

    public String getNombres() {
        return nombres;
    }

    public String getApellidos() {
        return apellidos;
    }

    public String getFechaDeNacimiento() {
        return fechaDeNacimiento;
    }
    ⋮
```

A simple vista se observa que esta clase hace referencia a que manejará información personal de algún individuo, tales como lo son sus atributos como el nombre, apellidos entre otros. Accedemos a ellos mediante los métodos get.

Esta clase puede ser seteada con los datos enviados desde el feature, teniendo en cuenta que el nombre de los encabezados del data table deben tener el mismo orden y nombre de los atributos de la clase:

```
Cuando el usuario se registra con us informacion personal
  | <nombres> | <apellidos> | <fechaDeNacimiento> | <tipoDeIdentificacion> | <identificacion> | <movil> |
```

```java
@Cuando("^el usuario se registra con us informacion personal$")
public void elUsuarioSeRegistraConSuInformacionPersonal(List<InformacionPersonal> informacionPersonal) {

}
```

Ejemplo implementando una clase Model con el patrón builder:

```java
public class Ingresa {

    private String origen;
    private String destino;

    public Ingresa ciudadOrigen(String origen) {
        this.origen = origen;
        return this;
    }

    public Ingresa ciudadDestino(String destino) {
        this.destino = destino;
        return this;
    }

    public static Ingresa aDondeViaja() {
        return new Ingresa();
    }

    public DondeViaja completaElItinerario() {
        return instrumented(DondeViaja.class, origen, destino);
    }

}
```

En su utilización se vería así:

```java
theActorInTheSpotlight().attemptsTo(Ingresar.aDondeViaja().ciudadOrigen(origen).ciudadDestino(destino).completaElItinerario());
```

El actor en escena intenta ingresar a donde viaja, ciudad de origen, ciudad de destino y completa el itinerario.

- **Questions:** En este paquete se crean clases donde se responde a una pregunta sobre el estado de la aplicación o el resultado de una operación realizada anteriormente en una funcionalidad. Con el fin de poder validar el resultado obtenido con el resultado esperado de la prueba.

  Esta clase implementa de la interface Question<ANSWER> y se debe sobrescribir el método answeredBy(actor) en la cual se debe escribir la lógica que permita obtener el estado o el resultado de un tipo específico que se ha de comparar con el valor esperado.

  Como buena práctica se ha de tener en cuenta que en esta clase no se realizan aserciones

Ejemplo:

```
package co.com.qvision.reto.avianca.questions;

import net.serenitybdd.screenplay.Actor;
import net.serenitybdd.screenplay.Question;
import net.serenitybdd.screenplay.annotations.Subject;
import net.serenitybdd.screenplay.questions.Visibility;

import static co.com.qvision.reto.avianca.ui.ListaDeVuelosComponent.LISTA_DE_VUELOS;
import static net.serenitybdd.screenplay.questions.ValueOf.the;

@Subject("Vista de Resultados de la búsqueda")
public class ResultadoDeLaConsulta implements Question<ListaDeVueloDisponible> {

    @Override
    public ListaDeVueloDisponible answeredBy(Actor actor) {

        return ListaDeVueloDisponible.from(the(Visibility.of(LISTA_DE_VUELOS).viewedBy(actor)));
    }

}
```

Como se observa, se implementa de la interfaz Question y se sobrescribe el método answereBy, en este método se implementará toda la lógica necesaria para devolver los datos u los objetos a validar. En este ejemplo se pregunta sobre la visibilidad de una lista de vuelos que debe visualizar el actor.

Se debe utilizar la interfaz **@Subject;** Esta se coloca encima del nombre de la clase con una breve descripción de la intención de la misma.

Ejemplo:

```
@Subject("Visualiza el usuario logueado")
public class ComprobacionUsuarioEnPaginaPrincipal implements Question<String>{
```

Estas clases deben ser llamadas de acuerdo al objeto o tipo de objeto que devolverá los datos a evaluar, se debe mantener el principio de interfaz fluida; en el siguiente ejemplo se observa que la clase cuestión es llamada “TheUserName”:

```
sam.should(seeThat(TheUserName.value(), equalTo("sam")));
```

El ejemplo anterior diría: **(sam debería ver que el valor del nombre de usuario es igual a "sam").**

Tener en cuenta que hay muchos Matchers que ofrece Serenity BDD para realizar estas validaciones, como, por ejemplo:

- isVisible()
- isNotVisible()
- isCurrentlyVisible()
- isNotCurrentlyVisible()
- isEnabled()
- isNotEnabled()
- isCurrentlyEnabled()
- isNotCurrentlyEnabled()
- isPresent()
- isNotPresent()
- isSelected()
- isNotSelected()
- containsText()
- containsOnlyText()
- containsSelectOption()
- hasValue()

- **Tasks:** En este paquete se crean clases compuestas por un conjunto de acciones o interacciones con el fin de documentar lo que hace un Actor en el sistema.

  Estas clases deben ser limpias y legibles de tal forma que solamente se utilicen interacciones de Screenplay o personalizadas, no se deben realizar implementaciones lógicas de programación para interactuar con la aplicación en estas clases.

  El nombre de las clases debe ir acorde a la tarea específica que realiza en su más alto nivel respetando siempre el principio de interfaz fluida.

  Ejemplo:

```
package co.com.qvision.reto.avianca.tasks;

import static co.com.qvision.reto.avianca.ui.ListaDeVuelosComponent.*;

public class AplicarFiltro implements Task {

    public static final String ORDENAR_POR_NUMERO_DE_PARADAS = "N";

    @Override
    public <T extends Actor> void performAs(T actor) {
        actor.attemptsTo(WaitUntil.the(FRAME, isPresent()),
                Switch.toFrame(FRAME_NOMBRE),
                WaitUntil.the(IMAGEN_DE_ESPERA, isNotVisible()),
                SelectFromOptions.byValue(ORDENAR_POR_NUMERO_DE_PARADAS).from(CAMPO_ORDENAR_POR),
                Click.on(CAMPO_FILTRAR_VUELOS),
                WaitUntil.the(BOTON_APLICAR_FILTRAR, isPresent()),
                Click.on(BOTON_APLICAR_FILTRAR));

    }

    public static AplicarFiltro PorNumeroDeParadas() {
        return instrumented(AplicarFiltro.class);
    }
}
```

Se puede observar que esta clase está compuesta por acciones atómicas pertenecientes a Screenplay, tales como, Switch.toFrame, WaitUntil.the, SelecFromOptions, Click.on, etc. En su conjunto realizan la tarea de aplicar un filtro sobre una lista de datos.

Es posible la utilización de la interfaz **@Step**; esta se utiliza para que determinado método que hace referencia a un paso en la automatización, pueda ser documentado para que aparezca en el reporte de Serenity. Esta se coloca encima del método y se le puede adicionar una breve descripción a esta anotación.

Ejemplo:

```
private String codigo;

public SearchBankAccount(String codigo) { this.codigo=codigo; }


@Override
@Step("{0} ingresa el '#codigo' de la cuenta bancaria ")
public <T extends Actor> void performAs(T actor) {
```

En la descripción de la anotación se pueden recibir parámetros utilizando una notación indexada que comienza con cero: {0} para el primer parámetro, {1}

para el segundo, etc. En el ejemplo anterior podemos observar que recibe un parámetro, en este caso recibe el nombre del actor. En caso de que se desee mostrar el valor de una variable que no recibe el método, pero que hace parte de la clase, se hace colocando '#<nombre de la variable>', como se observa en el ejemplo recibe el valor de la variable código.

- **Userinterface** En estas clases se capturan todos los elementos pertenecientes a la interfaz gráfica, con los cuales él actor interactuará durante la automatización. Estas clases no llevan implementación lógica, solo se crean variables de clase compuesta por los localizadores que identifican donde está ubicado el objeto que se interactuará mediante las acciones de Screenplay. Estas deben estar acompañadas de un constructor privado vacío.

  Las clases se deben nombrar según la pantalla a la que pertenezca en la interfaz del aplicativo, se debe tener en cuenta que, si una interfaz tiene muchos objetos, estos se pueden segregar en diferentes clases de forma lógica.

  Se debe evitar tener los mismos elementos declarados en varias clases, ya que puede generar deuda técnica por duplicidad de código.

  Solo se debe extender de la clase PageObjects cuando se hace uso de la interfaz **@DefaultUrl,** se utiliza para indicar la dirección web donde se ejecutarán las pruebas del proyecto. Usualmente esta interfaz se utiliza en la clase perteneciente al home de la aplicación.

  Ejemplo:

```
@DefaultUrl("http://ec2-52-45-228-254.compute-1.amazonaws.com:8080/ModuloAFC")
public class MenuPrincipalPageAFC extends PageObject {
```

  Para identificar los objetos de una aplicación es necesario identificar sus atributos, con el fin de interactuar con ellos. Serenity nos facilita el mapeo de los atributos HTML, dichos atributos deben ser utilizados de acuerdo al siguiente orden de prioridad.

  1. Id
  2. Name

3. ClassName
4. cssSelector
5. linkText
6. partialLinkText
7. tagName
8. xpath

Como estrategia para capturar los elementos de la interfaz, se debe tener en cuenta cuales son los elementos estáticos de esta, como los menús y otras opciones, que independientemente de donde estén ubicados siempre estarán presentes y disponibles para la interacción con el usuario. Estos elementos deberían tener su propia clase definida con el fin de utilizarlos en cualquier tarea o interacción donde se necesite hacer uso de estos. La clase recibiría el nombre de MenuPage (Siempre al final del nombre se le adiciona la palabra Page).

Las variables de clase deben ser públicas y final, deben seguir el estándar de nombramiento de constantes. Estas constantes al inicio del nombramiento deben tener un prefijo el cual indique el tipo de objeto, lo cual ayuda a conocer qué tipo de acciones se van a realizar sobre este, haciendo que el código sea más legible.

A continuación, se muestra la tabla para identificar el nombre de los prefijos que se pueden utilizar.

Prefijo	Tipo y Significado	Ejemplos
lbl	Label	lblNombre
lnk	LinkLabel	lnkNombre
btn	Button	btnNombre
txt	TextBox	txtNombre
mnm	MainMenu	mnmNombre
chk	CheckBox	chkNombre
rdb	RadioButton	rdbNombre
gbx	GroupBox	gbxNombre
pct	PictureBox	pctNombre
pnl	Panel	pnlNombre
dtg	DataGrid	dtgNombre
lbx	ListBox	lbxNombre
cbx	ComboBox	cbxNombre
lsv	ListView	lsvNombre
tvw	TreeView	tvwNombre
gvw	GridView	gvwNombre
tbc	TabControl	tbcNombre
prg	ProgressBar	prgNombre
rtb	RichTextBox	rtbNombre
img	ImageList	imgNombre
stb	StatusBar	stbNombre
dtp	DateTimePicker	dtpNombre
ddl	DropDownList	ddlNombre
hdf	HiddenField	hdfNombre
cld	Calendar	cldNombre
rep	Repeater	repNombre

¿Cuándo hacer el mapeo de un objeto con id y xpath?

El id debe de usarse cuando sea único. No se debe usar cuando pueda cambiar su valor de forma dinámica cada vez que se inicie la interacción con el objeto de la aplicación. Una forma de saber si el id es dinámico es mapeándolo y luego refrescar la aplicación.

El xpath se utiliza cuando se tiene una estructura compleja de la aplicación para el mapeo del objeto, por ejemplo, cuando el objeto no cuenta con atributos propios, sino que los atributos los contienen etiquetas, padres o hermanos.

Cuando se realice el mapeo con xpath no se debe hacer uso de “contains”, ya que genera dependencias sobre el texto que debe de encontrar. Tampoco se debe hacer uso de esta estructura “//*” en la construcción de un xpath, porque esto hace que recorra todos los nodos de la aplicación para encontrar el objeto, por lo tanto, no es recomendable por temas de rendimiento en la ejecución de las pruebas.

Ejemplo:

```
package co.com.qvision.reto.avianca.ui;

import static net.serenitybdd.core.annotations.findby.By.id;
import net.serenitybdd.screenplay.targets.Target;

public class ReservaTuVueloComponent {

    public static final Target GBX_RESERVA_TU_VUELO = Target.the("Pestaña reserva tu vuelo")
            .located(id("reservatuvuelo-home"));

    public static final Target TXT_DESDE_DONDE_VIAJAS = Target.the("Campo desde donde viajas")
            .locatedBy("//input[starts-with(@id,'pbOrigen_1')]");

    public static final Target LBX_CIUDAD_ORIGEN_SUGERIDA = Target.the("Selecciona la ciudad origen sugerida")
            .locatedBy("//li[@data-terminal='MDE']");

    private ReservaTuVueloComponent() {}

}
```

Los Target también pueden ser dinámicos permitiendo apuntar a diferentes objetos cuando los xpath son semejantes al recibir parámetros mediante un string format ‘{0}’ de la siguiente manera:

```
public static final Target BTN_RESERVAR = Target.the("reservar")
        .locatedBy("//p[@class='rate-number' and contains(text(),'{0}')]");
```

Su implementación se realizaría de la siguiente manera en una interacción de screenplay:

```
actor.attemptsTo(Click.on(BTN_RESERVAR.of(menorValor)));
```

Como se observa se envía una variable llamada menorValor como parámetro al Target BTN_RESERVAR.

- **Utilis:** En la automatización se puede requerir el uso de métodos que no hacen parte del objeto de prueba, pero ayudan a garantizar la funcionalidad de la misma. Para esto, se crean clases en esta capa. Métodos como, por ejemplo: lectura de archivos, formato de fechas, constantes, etc. Teniendo en cuenta que dichas clases deben de cumplir con el principio de única responsabilidad.

src/test/java: esta ruta contiene las pruebas que se ejecutaran.

- **Runners:** En este paquete se crean clases para ejecutar los features a través de tags y el glue code (los métodos que se encuentran en las clases StepDefinitions).

  Las clases runners se deben escribir en PascalCase y deben llamarse igual que el feature, donde se encuentran los escenarios que va a ejecutar. El runner está compuesto por:

  - **Features:** se indica la ruta del feature que va a ser ejecutado.
  - **Glue:** se indica la ruta del StepDefinitions que contiene los métodos asociados a los escenarios de prueba del feature.
  - **Snippets:** se indica que los métodos asociados a los escenarios de pruebas del feature en el StepDefinitions, van a ser construidos con un estilo de escritura de tipo “Camelcase”.
  - **Tags:** Se indican escenarios específicos que van a ser ejecutados mediante tags que son colocados al inicio de la construcción del escenario y son mencionados en esta opción de parametrización. Esto debe colocarse en el runner si es estrictamente necesario probar un escenario específico de lo contrario no debe de colocarse.

  Ejemplo:

```
import cucumber.api.CucumberOptions;
import cucumber.api.SnippetType;
import net.serenitybdd.cucumber.CucumberWithSerenity;
import org.junit.runner.RunWith;

@RunWith(CucumberWithSerenity.class)
@CucumberOptions(features = "src/test/resources/features/modulares/RegistrarInmuebleAFC.feature",
                 glue ="co.com.qvision.poc.sistemaAFC.stepDefinitions.modulares",
                 tags = "@RegistrarInmuebleExitoso",
                snippets = SnippetType.CAMELCASE
)
public class RegistrarInmuebleAFC  {
}
```

- **Stepdefinitons:** Se crean las clases que representan la traducción de los steps de los features a código ejecutable.

  Estas clases deben ser llamadas igual a los features asociados y adicionar la palabra “StepDefinitions”. Ejemplo: RegistrarInmuebleAFCStepDefinitions.

  En la clase no debe de ir lógica de programación, solo debe de ser llamadas las clases que realizan las acciones requeridas para la ejecución de las pruebas.

  En el método **Given** siempre debe estar asociada la acción “wasAbleTo” que hace referencia a una precondición. Se especifica que el actor pudo realizar determinada acción.

  Ejemplo:

```
@Dado("^el usuario ingresa a la pagina del sistema AFC$")
public void elUsuarioIngresaALaPaginaDelSistemaAFC() {

    theActorCalled( requiredActor: "AFC").wasAbleTo(NavegarApaginaLogueoSistemaAFC.paginaLogueo(OpcionUrl.URLINDEX));
}
```

  En el método When siempre debe estar asociada la acción “attemptsTo” que hace referencia a las tareas o acciones que va a realizar el actor. Las clases que se utilizan en este método son de las capas Tasks, Intereactions, Utils y Models.

  Ejemplo:

```
@Cuando("^guarda la informacion del inmueble registrado$")
public void guardaLaInformacionDelInmuebleRegistrado() {

    theActorInTheSpotlight().attemptsTo(GuardarInfoInmueble.guardarInfo());
}
```

  En el método Then siempre debe estar asociada la acción “should” que hace referencia a que debería hacer la comparación del resultado de la prueba. Las clases que se utilizan en este método son de las capas Questions, Utils y Exceptions.

Ejemplo:

```
@Entonces("^visualiza el mensaje de login incorrecto$")
public void visualizaElMensajeDeLoginIncorrecto() {

    theActorInTheSpotlight().should(seeThat(ComprobacionMsjLoginIncorrecto.msjLoginIncorrecto(),
            equalTo(Constantes.MENSAJECREDENCIALESINCORRECTAS)));
}
```

Para la generación del reporte de evidencias es importante hacer uso de etiquetas para que se pueda visualizar de una forma limpia, ordenada, que entreguen información legible y demás detalles de la ejecución de las pruebas. Para esto se hará uso de las siguientes etiquetas en el proyecto de automatización en la clase StepDefinitions.

- **@Managed:** Se utiliza para declarar una instancia del WebDriver que será administrada por serenity. Esta etiqueta siempre debe de ir en cada StepDefinions y se puede hacer uso de los siguientes parámetros:

    - **Driver:** Definir en qué controlador WebDirver desea ejecutar las pruebas. Ejemplo: @Managed (Driver=”Chrome”).

    - **UniqueSession:** Permite que Serenity abra el navegador al comienzo de las pruebas y lo deje abierto hasta que se hayan ejecutado todas las pruebas. Ejemplo: @Managed (uniqueSession=true).

    - **ClearCookies:** Permite que Serenity borre las cookies para cada prueba o nunca se borren las cookies, los posibles valores son BeforeEachTest y Never. Ejemplo: @Managed (clearCookies=BeforeEachTest).

    Los controladores compatibles con esta anotación son: Firefox, Chrome, Opera, HtmlUnit, PhamtonJS, IExplorer, Edge, Safari, Appium.

    Ejemplo:

```
public class LogueoSistemaAFCStepDefinitions {

    @Managed (driver= "Chrome")
    WebDriver driver;

    @Steps
    NavegarApaginaLogueoSistemaAFC navegarApaginaLogueoSistemaAFC;

    @Before
    public void doSomethingBefore() {
        OnStage.setTheStage(new OnlineCast());
    }
```

- **@Steps:** la cual se utiliza para agrupar todos @Step colocados en los métodos de una clase, esto permite que en la generación del reporte de las evidencias salga de forma ordenada.

  Esta etiqueta siempre se debe colocar en cada StepDefinitions con cada declaración de una clase que contenta @Steps.

  Ejemplo:

```
public class LogueoSistemaAFCStepDefinitions {

    @Steps
    NavegarApaginaLogueoSistemaAFC navegarApaginaLogueoSistemaAFC;

    @Before
    public void doSomethingBefore() {
        OnStage.setTheStage(new OnlineCast());
    }
```

src/test/ Resources: dentro de test existe un paquete llamado resources donde se obtienen todos los recursos definidos para su uso dentro de la automatización, este paquete está compuesto de lo siguiente:

- **Data:** Se crean los archivos que contienen los datos de los escenarios de pruebas. Los archivos pueden ser de diferentes tipos como: .txt, .json, .xml., .properties, etc.

- **Drivers:** En este paquete se guarda el driver que se utiliza para hacer uso de la aplicación donde se van realizar las pruebas como el driver de Chrome el cual es llamado ChromeDriver.

- **Features:** Se crean los features con los escenarios de pruebas definidos en lenguaje Gherkin y se hace uso de la herramienta Cucumber.

  Para el nombramiento de estas **l**as palabras deben ser en minúsculas y se debe usar un verbo en infinitivo. Si contiene más de una palabra debe ser separada por un guion bajo. Ejemplo: registrar_inmueble.feature.

  El feature como tal no es una clase, es un archivo de texto con el cual podemos interpretar una historia de usuario con unos criterios de aceptación, los cuales se convierten en un conjunto de acciones que se realizan para verificar una funcionalidad determinada.

  Estos escenarios describen un punto de partida, una acción y un resultado esperado con el apoyo de expresiones regulares para nombrar variables de entrada que se usarán durante la ejecución de las pruebas.

  Los escenarios de prueba deben de estar redactados en tercera persona de forma declarativa o funcional para que puedan ser comprendidos por los interesados en el proyecto o negocio. Por lo tanto, deben estar en un lenguaje de alto nivel donde no se mencione el detalle de la implementación, es decir, no deben redactarse de forma imperativa, es decir, a un nivel técnico indicando acciones sobre los elementos, por ejemplo: dar clic sobre determinada opción, escribir en un campo específico, seleccionar un checkbox etc.

El lenguaje en el que estarán escritos los steps será el que utiliza el cliente en las historias de usuario y este debe regir el lenguaje descrito en las clases.

La estructura de un archivo feature está dado por lo siguiente:

- **Given (Dado):** Indica la precondición del negocio o el contexto inicial del sistema. Que se requiere tener realizado para continuar con la prueba y hacer las acciones determinadas.

- **When (Cuando):** Indica las acciones o tareas que se ejecutan con el fin de llegar a un resultado verificable.

- **Then (Entonces**): Indica los pasos que se utilizan para comparar un resultado esperado con el resultado obtenido.

- **And – But (Y – Pero):** Permite unir acciones, es decir, se puede utilizar después de un Given (Dado), When (Cuando) o un Then (Entonces). Se recomienda que se utilicen cuando sea estrictamente necesario, ya que se requiere dar más entendimiento acerca del escenario.

- **Background (Antecedentes):** Es utilizado para definir una serie de acciones que son comunes en cada escenario del feature. Se coloca antes de los escenarios definidos y las acciones del background ya no se colocarían en los escenarios.

- **Examples:** Es una tabla separada por (| -> pipes), donde cada columna está compuesta por el título o variable que hace referencia al dato que tiene asociado. Son definidos después de un escenario, y este debe ser construido como **“Scenario Outline (Esquema del escenario)”,** allí se colocan los datos de entrada a utilizar en métodos de la clase StepDefinitions que hacen referencia a los escenarios del feature.

El When (Cuando) y Then (Entonces) son obligatorios en la construcción de los escenarios del feature, ya que, siempre se debe indicar las acciones a ser ejecutadas por el actor, y luego hacer la validación del resultado de las acciones realizadas en la ejecución de las pruebas.

Ejemplo:

```
Característica: Regristrar inmueble
  Yo como usuario del sistema AFC
  Necesito ingresar al modulo registrar inmueble
  Para ingresar los inmuebles de las cuentasAFC

@RegistrarInmuebleExitoso
  Esquema del escenario: Realizar el registro de un inmueble
    Dado el usuario ingresa a la pagina del sistema AFC
    Cuando ingresa sus credenciales correctas
      | contrasenaCorrecta |
      | glgg               |
    Y ingresa al menu retiros a registrar inmueble
    Y diligencia los datos para registrar el inmueble con sus beneficiarios
      | nroCuenta   | vlrInmueble   | estadoInscripcion   | nombreInmueble   | observaciones   | tipoIdBeneficiario
      | <nroCuenta> | <vlrInmueble> | <estadoInscripcion> | <nombreInmueble> | <observaciones> | <tipoIdBeneficiario>
    Y guarda la informacion del inmueble registrado
    Entonces puede observar el registro exitoso del inmueble
    Y puede salir de la aplicación
    Ejemplos:

      | nroCuenta   | vlrInmueble | estadoInscripcion | nombreInmueble | observaciones        | tipoIdBeneficiario |
      | 10000000237 | 10000000    | Activo            | Finca          | Registro apartamento | Cédula Ciudadanía  |
```

En el feature (Característica), debe ser nombrada la funcionalidad que se va a probar mediante los escenarios, y después se coloca la necesidad u objetivo de la prueba en forma de historia de usuario como se encuentra en el ejemplo anterior.

Cuando los datos de entrada que se envían en el feature al StepDefinitions van a ser utilizados a través de una clase de la capa model, las variables de cada columna de la tabla deben ser nombradas igual los atributos de la clase.

Ejemplo de un escenario implementado con un scenario outline:

```
@Caso:Adicionar_Medicamentos_A_RAF_Vigente
Esquema del escenario: Adicionar Medicamentos a un RAF vigente.
  Y Se realiza la busqueda del paciente que se ingresará y se elige el plan de atención
    | <Id paciente> | <Plan> |
  Y Se inicia el proceso de transcripción
  Y Se ingresa el diagnóstico
    | <Diagnostico> |
  Cuando Se transcriben los medicamentos
    | Medicamento   | Dosis   | Periodo   | Duracion   | Cantidad   | PeriodoRAF   | Tratamiento   | Adicionado   |
    | <Medicamento> | <Dosis> | <Periodo> | <Duracion> | <Cantidad> | <PeriodoRAF> | <Tratamiento> | <Adicionado> |
  Entonces Se valida el medicamento Transcrito por el usuario
  Y Se finaliza la Transcripción sin imprimir la Orden
  Y Se valida la transcripción de los medicamentos añadidos al RAF vigente
    | <Id paciente> | <Diagnostico> | <Medicamento> | <Dosis> | <Periodo> | <Duracion> | <Cantidad> | <Adicionado> |

  @StartIt
  Ejemplos: Se crea un RAF Vigente y se Valida.
    | Id paciente | Plan                    | Diagnostico | Medicamento | Dosis | Periodo | Duracion | Cantidad | PeriodoRAF | Tratamiento | Adicionado |
    |    40402036 | POS (POS SALUD EN CASA) | Z010        |        7036 |     1 |      24 |       30 |       30 |          6 |             |            |

  Ejemplos: Se adiciona un medicamento al RAF Vigente Al finalizar la consulta y se valida.
    | Id paciente | Plan                    | Diagnostico | Medicamento | Dosis | Periodo | Duracion | Cantidad | PeriodoRAF | Tratamiento | Adicionado |
    |    40402036 | POS (POS SALUD EN CASA) | Z010        |       14102 |     1 |      24 |       30 |        1 |          6 | A           | No         |
```

1.5 Estándares generales.

Estándar de paquetes:

Estándar tomado de la guía para las convenciones de nomenclatura en groupId, artifactId y versión de Apache Maven Project.

Groupid: Es el dominio de una página web de la empresa (en orden contrario) seguido de [empresa.departamento.aplicativo]. Por ejemplo:

- co.com.empresa.departamento.aplicativo
- co.com.qvision.certificacion.automatizacion

Estándar de nombramiento; tomado de especificaciones técnicas de paquetes de Oracle:

- El nombre de los identificadores de los paquetes se escribe en minúsculas.
- Si el nombre de dominio contiene un guion, o cualquier otro carácter especial no permitido en un identificador se debe remplazar por un guion bajo "_".
- Si alguno de los componentes del nombre del paquete resultante son palabras reservadas del lenguaje, entonces agregue un guión bajo a ellas.
- Si alguno de los componentes del nombre comienza por un dígito o cualquier carácter no permitido debe reemplazarse por un guion bajo.
- Los nombres de los paquetes deben ser únicos.

Nota: Los proyectos de automatización tienden a crecer con el tiempo, entre más escenarios se cubran con las pruebas funcionales automáticas se irán construyendo más clases, runners, features y demás, por lo tanto, dentro de los paquetes antes mencionados se deben crear subpaquetes que ayuden a organizar el proyecto de forma lógica, esta tarea se puede realizar por funcionalidades cubiertas del aplicativo probado.

- **Clases**: Las palabras deben ser escritas en forma PascalCase, es decir, todas las palabras deben tener la primera letra en mayúscula. Ejemplo: RegistrarInmueble.

- **Métodos y atributos:** Las palabras deben ser escritas en forma camelCase, la primera palabra debe ser en minúsculas y las demás palabras deben ser con la primera letra mayúscula.

  Ejemplo:

  -Método: crearInmueble();
  -Atributo: public String tipoInmueble;

- **Constantes**: Las palabras deben ser escritas en mayúsculas, si contiene más de una palabra debe ser separadas con un guion bajo. Ejemplo: NOMBRE_USUARIO.

  Encondig: cuando se va a trabajar en español se deben configurar diferentes opciones de encoding a nivel de: IDE, Feature, Serenity.properties y del Build.gradle.

1.6 Configuración de serenity BDD.

La configuración se puede realizar en dos archivos dependiendo la versión de cucumber que se esté utilizando:

- ***Serenity.properties***

Es un archivo que se crea en la raíz del proyecto y se utiliza para especificar por medio de las propiedades de serenity la forma en que se va a realizar la ejecución de las pruebas. Las parametrizaciones mínimas necesarias que se deben de implementar en el serenity.properties son las siguientes:

- **Serenity.project.name:** Se coloca el nombre del proyecto para que salga en el reporte de evidencias.

- **Webdriver.driver:** En que navegador desea que se ejecuten las pruebas. Por ejemplo: Firefox, Chrome, Internet Explorer, etc.

- **Webdriver.{type}.driver:** Se la ruta del controlador que se va utilizar. El tipo de controlador debe especificarse en la propiedad del sistema. Por ejemplo: webdriver.chrome.driver.

- **Webdriver.wait.for.timeout:** Se especifica el tiempo de espera para elementos específicos de la página web antes de ser lanzada una excepción como por ejemplo “ElementNotVisibleException”.

- **Webdriver.timeouts.implicitlywait:** Se especifica el tiempo de espera para que aparezcan todos los elementos de la aplicación.

- **Serenity.timeout:** Se especifica cuanto tiempo debe esperar el controlador para que los elementos sean visibles.

- **Serenity.logging:** Propiedad para proporcionar el nivel de acciones de serenity. Se debe asignar el valor VERBOSE para que registre el inicio, el fin y los pasos de pruebas ejecutados.

- **Serenity.report.encondig:** Se coloca el encondig que va utilizar para el manejo de caracteres el proyecto para la generación del reporte.

- **Serenity.verbose.steps:** Se utiliza para dar un registro más detallado de los pasos de WebElementFacade cuando se ejecutan las pruebas.

- **Serenity.reports.show.step.details:** Se utiliza para mostrar los resultados de las pruebas. Si el valor asignado es true el reporte de evidencias muestra de forma detallada los resultados de las pruebas, pero si valor asignado es false se muestran los resultados por defecto.

- **serenity.compromised.on** = se utiliza para configurar las excepciones por las cuales los test pueden fallar por temas externos a los mismos. Se configura colocando las excepciones separadas por comas.

- **Chrome.switches:** Se asignan parámetros de configuración al controlador separado por comas para indicar de qué forma se desea ejecutar las pruebas en una aplicación web. Por ejemplo: En modo incógnito, deshabilitar notificaciones, maximizar el navegador, etc.

  Ejemplo:

```
serenity.project.name = AutomatizaciónSistemaAFC
webdriver.chrome.driver= src/test/resources/drivers/chromedriver.exe
webdriver.driver= chrome
webdriver.wait.for.timeout = 20000
webdriver.timeouts.implicitlywait = 20000
serenity.timeout = 20000
serenity.logging =VERBOSE
serenity.verbose.steps =true
serenity.reports.show.step.details = true
thucycides.step.delay = 300000
chrome.switches = --incognito --lang=es,--disable-popup-blocking,--disable-download-notification,--start-maximized,--headless,--window-size=900,768
```

  En la parametrización de **chrome.swiches** es importante hacer el uso del parámetro “--headless”, porque permite mejorar el rendimiento de la ejecución de las pruebas en cuanto a tiempo y recursos del sistema. Se debe utilizar cuando las pruebas tienen un tiempo muy largo de ejecución, ya que las pruebas son ejecutadas en memoria. Se recomienda ejecutar primero las pruebas sin el uso de este parámetro visualizando todo el proceso de ejecución de las pruebas para garantizar el correcto funcionamiento de la automatización.

  También, se debe especificar la resolución de la pantalla, esto permite que todos los objetos de la aplicación sean visibles durante la ejecución de las

pruebas. La resolución de la pantalla se especifica con el parámetro "- -window-size".

Ejemplo:
serenity.browser.width=1400
serenity.browser.height=800
chrome.switches =--disable-print-preview, --disable-gpu, --headless

- **Serenity.conf**

Es un archivo que se crea en src/test/resources, se utiliza para especificar a través de las propiedades de serenity las configuraciones para utilizar más de un controlador, así poder acceder a navegadores y ambientes diferentes, generalidades de los capabilities, entre otras opciones de configuración.

A continuación, se especifica que se debe implementar en el serenity.conf como lineamiento para la construcción del proyecto de automatización:

Configurar la URL base para diferentes ambientes: En el desarrollo de la automatización es muy común usar @DefaultUrl ("<dirección url>") para abrir el navegador con una página web determinada, pero con este archivo se puede indicar a través de la propiedad **webdriver.base.url** la URL a la que se va a direccionar, la ventaja del serenity.conf es que se puede configurar más de un entorno para la ejecución de las pruebas. Esta función es disponible a partir de la versión de serenity 2.0.30.

En el archivo se implementa una sección llamada **environments**, allí se pueden definir más secciones, una de ellas siempre debe ir con el nombre **default** donde se especifica la URL base por defecto, y las demás secciones que se deseen colocar reciben el nombre de los ambientes o entornos a los que se necesite acceder con su respectiva URL base.

Ejemplo:

```
environments {
    default {
        webdriver.base.url = "http://localhost:8080/myapp"
    }
    dev {
        webdriver.base.url = "http://dev.myapp.myorg.com"
    }
    staging {
        webdriver.base.url = "http://staging.myapp.myorg.com"
    }
    prod {
        webdriver.base.url = "http://myapp.myorg.com"
    }
}
```

Para acceder a la URL base de un ambiente determinado, se hace mediante el comando gradel: gradle clean test -Denviroment=[nombre_ambiente], si en el comando no es especificado el ambiente, es decir, se hace uso del comando: gradle clean test -Denviroment, tomaría la URL por defecto de la sección default, ya que no fue especificado el ambiente en el comando ejecutado.

Es posible agregar Driver proveedor de WebDriver personalizado implementando la interfaz DriverSource. Para esto, se debe configurar las siguientes propiedades del sistema:

- webdriver.driver = provided; se indica que se proporcionará un driver personalizado.
- webdriver.provided.type = “NombeDelDriver”
- webdriver.provided.[NombeDelDriver] = [“Ruta del driver en el src”]
- thucydides.driver.capabilities = “NombeDelDriver”

Ejemplo:

```
webdriver.driver = provided

webdriver.provided.type = WiniumDriver

webdriver.provided.WiniumDriver = "starter.winium.util.CustomWiniumDriver"

thucydides.driver.capabilities = WiniumDriver
```

1.7 Ejecución del proyecto.

Para la ejecución de las pruebas del proyecto se pueden utilizar los siguientes comandos:

- **Comando para ejecutar todo el proyecto:**
  gradle build

- **Comando para ejecutar un grupo de escenarios:**
  gradle test -Dcucumber.options = “- -tags @debug1, @debug2”

- **Comando para todos los escenarios de un feature específico**:
  gradle clean test -Dtest.single=”<nombre_runner>” --info

1.8 Generación del Reporte.

Para la generación del reporte se pueden utilizar los siguientes comandos:

- **Comando para ejecutar todo el proyecto y generar el reporte:**
  gradle clean test aggregate

- **Comando para ejecutar todos los escenarios de un feature específico:**

  Se recomienda hacer uso de **clean** en el comando para limpiar reportes que se hayan generado anteriormente, y al ejecutar el comando se pueda generar uno nuevamente. El uso de **aggregate** en el comando es para la generación del reporte de evidencias.

1.9 Gestión de dependencias.

Como herramienta para la gestión de dependencia se utilizará Grade desde la versión 2.5 en adelante.

En el archivo build.gradle se realiza la gestión de dependencias del proyecto entre otras configuraciones.

```
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'net.serenity-bdd.aggregator'

sourceCompatibility = 1.8

repositories {
    mavenLocal()
    jcenter()
}

buildscript {
    repositories {
        mavenLocal()
        jcenter()
    }
    dependencies {
        classpath("net.serenity-bdd:serenity-gradle-plugin:2.2.0")
    }
}
ext {
    slf4jVersion = '1.7.7'
    serenityCoreVersion = '2.2.0'
    serenityCucumberVersion = '2.2.0'
    junitVersion = '4.12'
    assertJVersion = '3.8.0'
    logbackVersion = '1.2.3'
}

dependencies {
    implementation "ch.qos.logback:logback-classic:${logbackVersion}"

    testImplementation "net.serenity-bdd:serenity-core:${serenityCoreVersion}",
            "net.serenity-bdd:serenity-cucumber5:${serenityCucumberVersion}",
            "net.serenity-bdd:serenity-screenplay:${serenityCoreVersion}",
            "net.serenity-bdd:serenity-screenplay-webdriver:${serenityCoreVersion}",
            "junit:junit:${junitVersion}",
            "org.assertj:assertj-core:${assertJVersion}"
}

test {
    testLogging.showStandardStreams = true
    systemProperties System.getProperties()
}

tasks.withType(JavaCompile) {
    options.encoding = 'UTF-8'
}

gradle.startParameter.continueOnFailure = true
test.finalizedBy(aggregate)
```

Plugin: La aplicación de un complemento a un proyecto permite que el complemento amplíe las capacidades del proyecto:

- **Plugin** 'java': el complemento de Java agrega compilación de Java junto con capacidades de prueba y agrupación a un proyecto.
- **Plugin** 'eclipse': el complemento de Eclipse genera archivos que son utilizados por el IDE de Eclipse, lo que permite importar los proyectos.
- **Plugin** 'idea': el complemento de intellij IDEA genera archivos que son utilizados por el IDE de intellij IDEA, lo que permite importar los proyectos.
- **Plugin** 'net.serenity-bdd.aggregator': genera el reporte de pruebas luego de la compilación de los test.

SourceCompatibility: especifica que se utilizará una versión del lenguaje de programación Java para compilar archivos **.java**

Repositories: en este apartado se especifica los repositorios en los cuales se realizará la descarga de los complementos. Gradle puede resolver dependencias de uno o varios repositorios basados en Maven, Ivy o formatos de directorio plano.

Buildscript: en este bloque se declararán dependencias externas para el build.gradle en sí mismo, con el fin de que pueda realizar la compilación. Este determina qué complementos, clases y tareas están disponibles para su uso en el resto del script de compilación.

Ext: (ExtraPropertiesExtension) Las extensiones de propiedades adicionales permiten agregar nuevas propiedades a los objetos de dominio existentes. Actúan como mapas, permitiendo el almacenamiento de valores. Estos valores pueden ser modificados en tiempo de ejecución si es necesario.

Dependencias: En este apartado se declaran las dependencias que se necesitan para el proyecto. La sentencia compile/implementation se utiliza para las dependencias que utilizaran en src/main/java, y la sentencia testCompile/ TestImplementation se utiliza para las dependencias de los test src/test/java.

Test: esta tarea se utiliza para configurar variedad de opciones que se utilizan en tiempo de ejecución de los test.

gradle.startParameter.continueOnFailure: configuración que permite.
test.finalizedBy(aggregate): esta opción permite que al finalizar la construcción del proyecto o la ejecución de las pruebas se genere el reporte de pruebas de Serenity BDD.

Se debe indicar el enconding para el manejo de caracteres en el proyecto:

```
tasks.withType(JavaCompile) {
    options.encoding = "UTF-8"
}
```