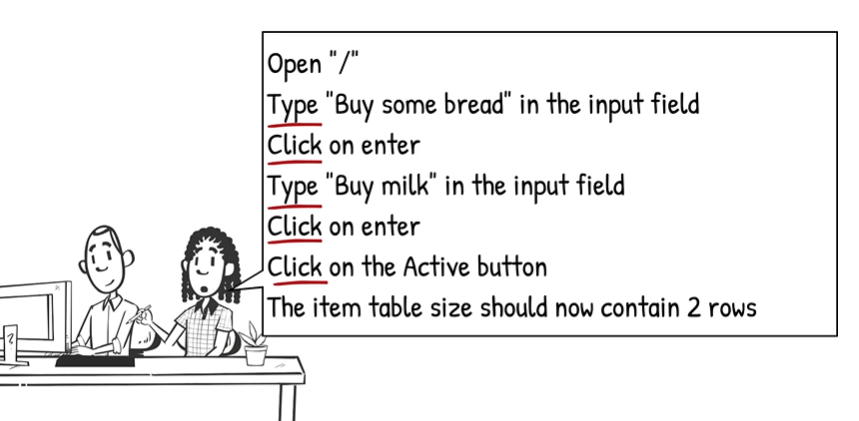
El código de la testing Logic suele llamarse Scripting. Pero puede ser un término engañoso. El código de esa capa debe ser simple para una aplicación simple, pero para una aplicación complicada, puede volverse bastante enmarañado. En este último caso, el código debe estar bien diseñado y organizado; justo como el código productivo.

Supongamos que tenemos el siguiente “script”: Nos dice **como** se hace el test, pero no nos dice **que** es lo que estamos testeando.



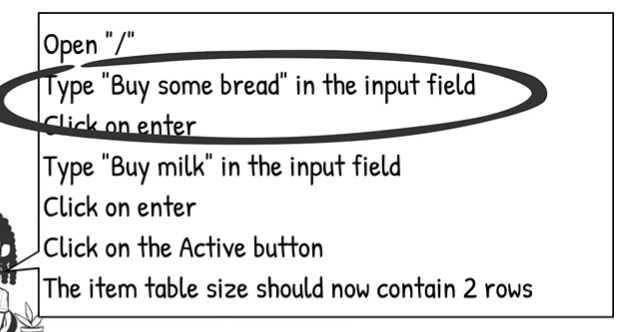
Solo se trata de ingresar datos en algún campo y de hacer clicks en algún botón. Está muy bien si nuestro fin es entender la mecánica de cómo funciona la prueba. Pero no nos dice nada acerca de lo que la prueba quiere demostrar. Si se rompe, será difícil saber si es una bug de la aplicación o si la UI ha cambiado.

Aun en este simple y pequeño ejemplo tenemos **mucho código duplicado** -> tenemos la acción de ingresar un valor y de clickear en Enter dos veces. La duplicación de código hace que nuestra aplicación sea difícil de mantener porque significa que tenemos que hacer cambios en más de un lugar en el caqso que la UI cambie. Asi se escriben muchos test, pero no es la forma si queremos pruebas fáciles de escribir, de leer y de mantener.

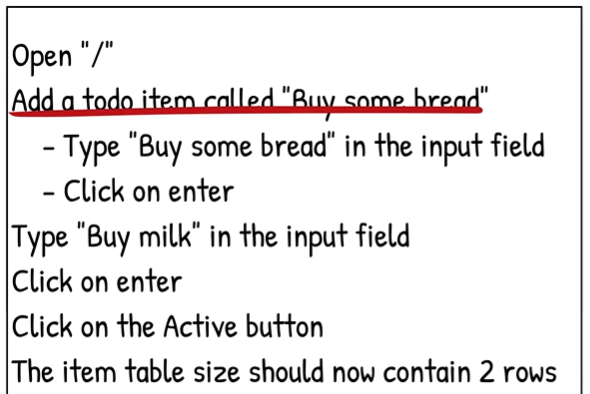
Una buena prueba:

* no solo te dice como se prueba en términos de interacción con la aplicación,
* sino que también te dice que es lo que está tratando de demostrar en términos de negocio.

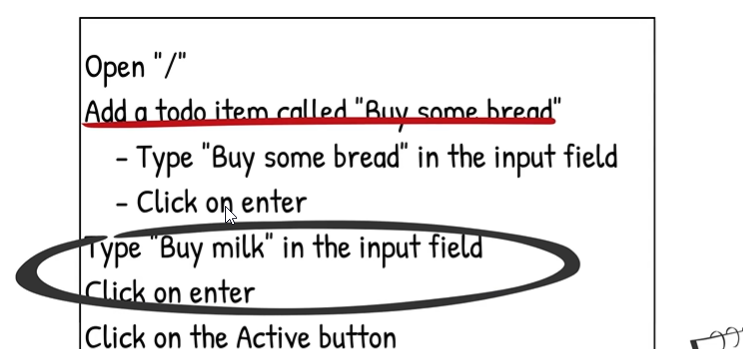
Esta prueba se trata sobre agregar ítems en una ToDo list.



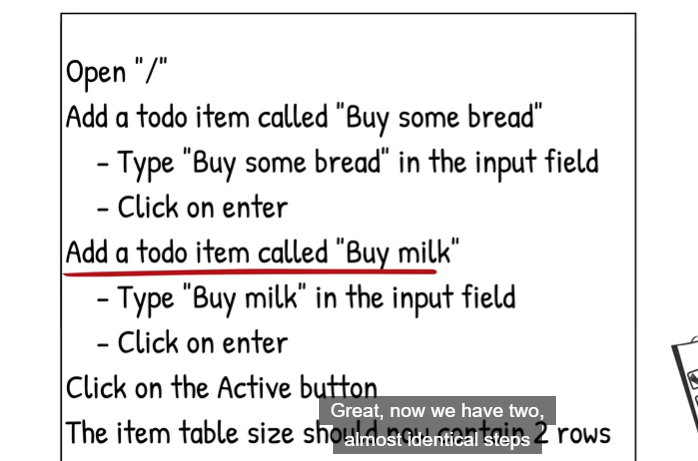
Pongamos el texto de una forma más obvia. Las dos acciones señaladas abajo se refieren a ingresar un ítem dentro de la ToDo list. Vamos a poner un título y agrupar juntas las dos acciones.



Si miramos las interacciones siguientes, son muy similares: Solo se tratan de agregar un ítem ToDO diferente.



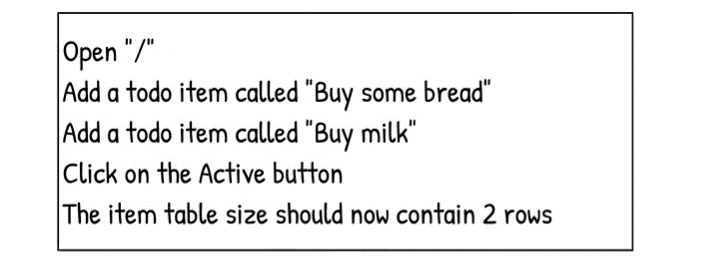
Entonces podemos agruparlas de la misma manera. Y así obtenemos dos steps casi idénticos que pueden ser implementados con el mismo método (diferenciándose únicamente en el o los parámetros).



Mientras tanto, podemos ocultar los detalles y dejamos visibles solo los títulos:

Vemos que esa parte del test se lee mas claramente en relación al QUE o propósito de la prueba.

Pero, que pasa que el primer paso: Open “/” ? -> se refiere a abrir un browser desde un URL.



¿Y que pasa con el cuarto paso que es sobre clickear cierto botón? ¿Qué hacemos para transformarlo más en una descripción de lo que el usuario está tratando de lograr y no tanto en la interacción del usuario con el UI?

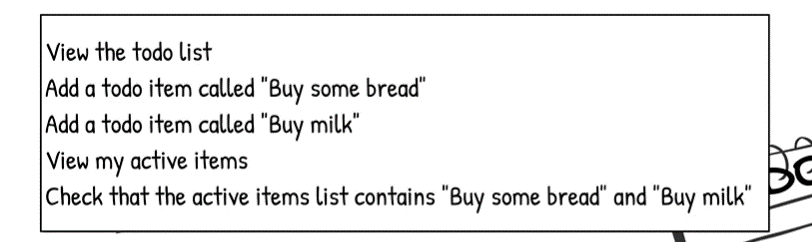
El primero es solamente relativo a lo que el usuario hace para abrir la ToDo App y ver la ToDo list. -> Entonces, hagamos que diga eso.

La cuarta línea es relativa a ver los ítems que no han sido completados aun en la ToDo list (Items activos) -> Entonces, solo digamos: Ver ítems (actividades) activos.

Finalmente, respecto a la última línea, no está muy claro que es lo que se trata de demostrar. Si el UI cambia, vamos a tener problemas para actualizar el texto. Una vez más, necesitamos saber el porqué…

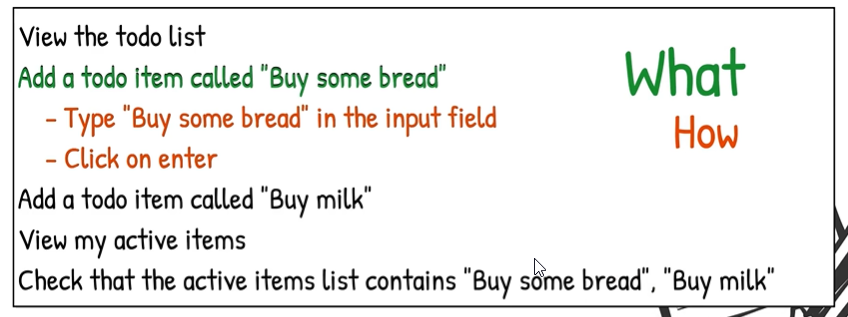
La comprobación real no es que ahora hay 2 rows en la tabla, sino que los 2 ítems que agregamos sean visibles como activos (pendientes)

De esta manera, el script fue transformado a los siguiente: Donde no se captan solo las acciones del usuario si no sus intenciones (lo que quiere lograr):



Y siempre podemos ir adentro de los detalles de los métodos si queremos saber el COMO se ejecutan las acciones.

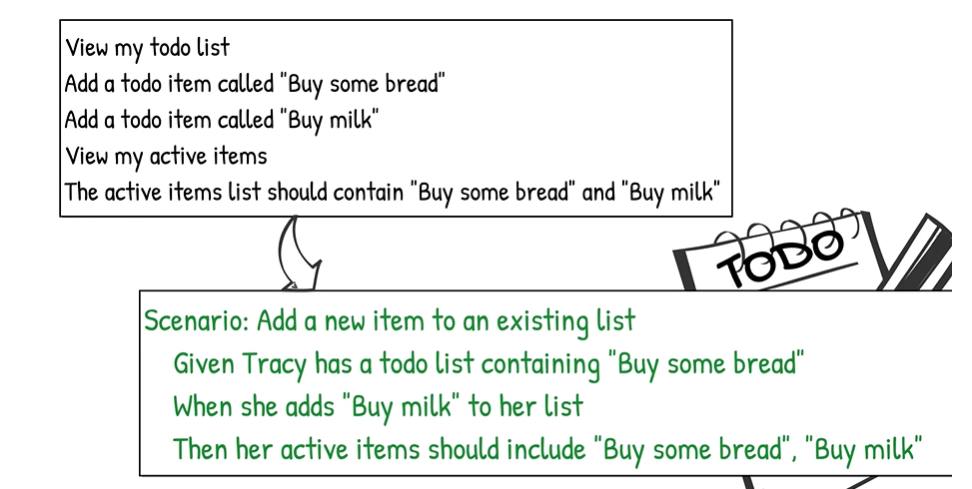
Pero aún hay una cosa que falta. Empezamos con el COMO de los detalles en como interactúa el usuario con la UI. Después nos movimos al QUE. Que es lo que el usuario intenta hacer o demostrar.



Aun así, el test sigue algo desordenado y no es tan fácil deducir cual es la historia que el test nos quiere contar.

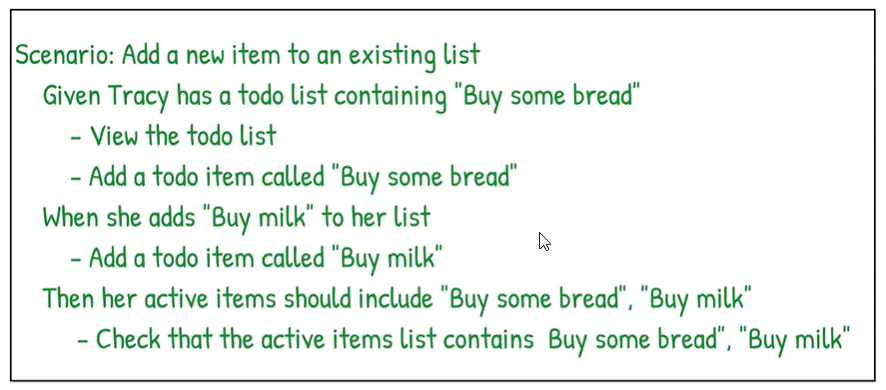
Estamos viendo que sucede cuando dos items son agregados a una lista vacía? O Estamos viendo que sucede cuando se agrega un ítem a una lista que ya contenía un ítem?

Aquí es cuando Cucumber entra en juego: Aquí, podemos escribir escenarios de alto nivel que nos digan cuales son los resultados que esperamos:

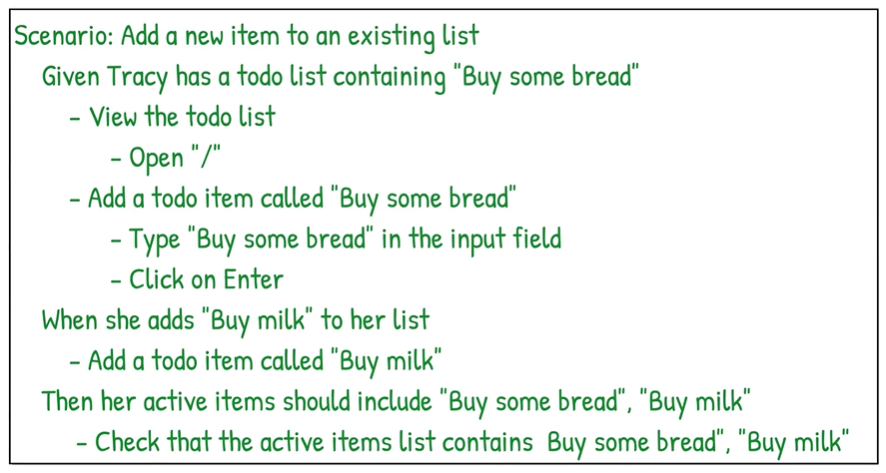


Este escenario describe que es lo que Tracy está tratando de hacer con los pasos a alto nivel que ella necesita llevar a cabo para alcanzar el objetivo. Y también nos dice si hemos sido exitosos en alcanzar ese objetivo -> y todo en un inglés muy legible en términos de negocio.

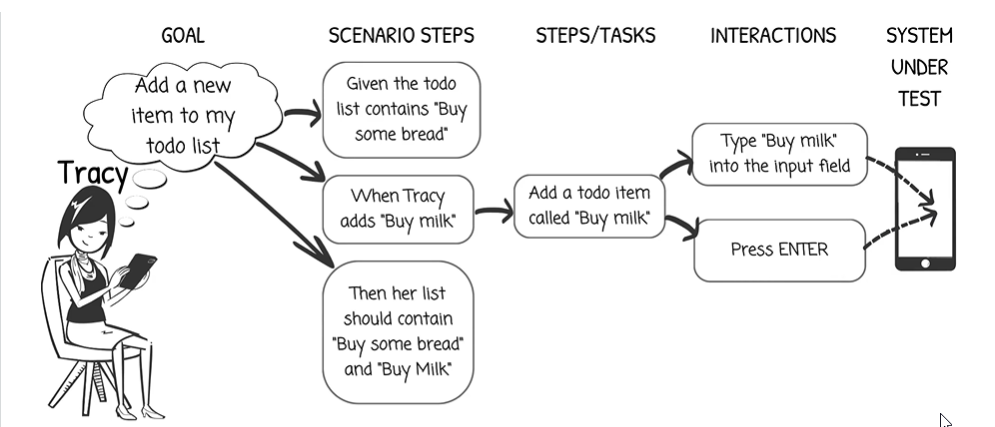
Visto de otra manera, se puede mapear la lista de actividades con el escenario gherkin de la siguiente manera:



Pero no olvidemos que tenemos 3 capas; debajo de cada una de las tareas hay una interacción con el UI.



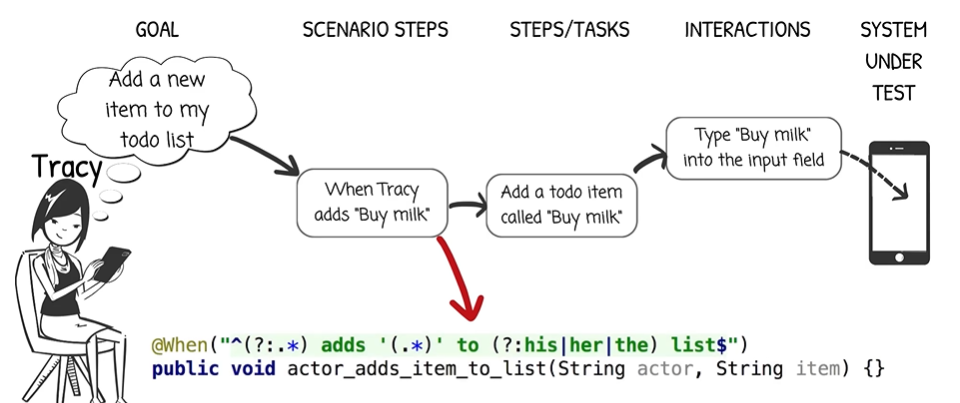
Entonces, vayamos a través de las 3 capas: Tenemos a Tracy que es usuaria de la ToDo App, ella tiene un objetivo: agregar un nuevo ToDo ítem a su ToDo list.



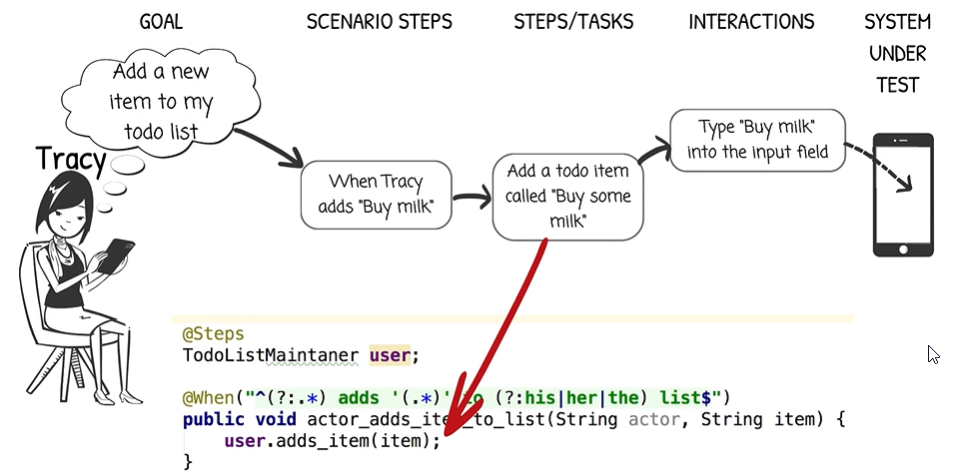
Un ejemplo de ella consiguiendo el objetivo podría involucrar 3 steps. Esos steps pueden descomponerse en tareas más pequeñas como “Add a todo ítem called “Buy milk”” o “view the todo list”. Pero, aún seguimos muy enfocados en el negocio, no estamos preocupados en **como** interactuamos con el sistema. Podemos entonces descomponer en **interacciones** como “Type “Buy milk” into the input field” y “Press Enter” Y esto si interactúa con el sistema bajo testeo.

Vayamos a través de un solo camino para ver cómo encaja con el código de automatización de Cucumber y Serenity

* El GOAL corresponde al título del escenario
* Los SCENARIOS STEPS se mapean con las clausulas Given, When, Then
* Los Cucumber Given, When, Then mapean con steps definition methods (ver imagen de más abajo) que actúan de traductor entre los datos, el escenario cucumber y la test logic que es la que realmente hace el trabajo.

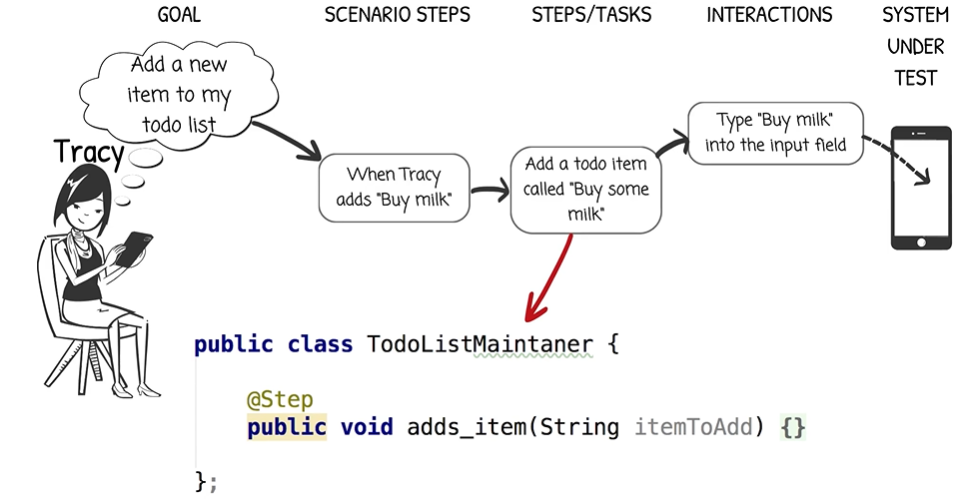


* STEPS/TASKS: Luego, necesitamos implementar los step definition methods. Estos métodos se denominan business level steps (pasos de negocio) or taks (tareas) que implementamos ya sea como pequeños **métodos** o como **objetos** si es que estamos usando el **patrón screenplay**. Por ahora, representaremos pasos de negocio o tareas como métodos como el método add\_item de la imagen de abajo.

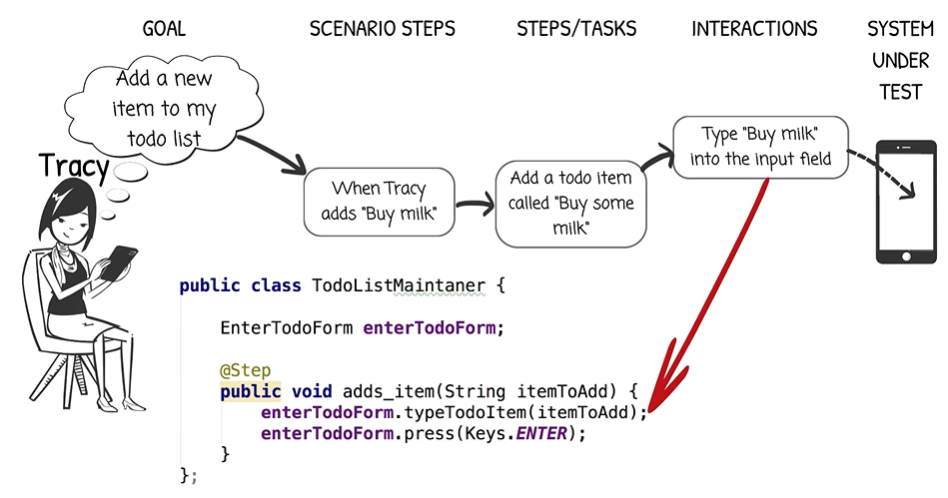


Notemos la clase TodoListMaintaner y el objeto user. La step anotation “@Step” significa que esta clase es una serenity step library, una colección de pasos de negocio o tareas que llamamos desde los steps definitions. La anotation nos dice que serenity monitoreara todas las llamadas que surgen en los steps methods de la clase para que figuren en el reporte de testing.

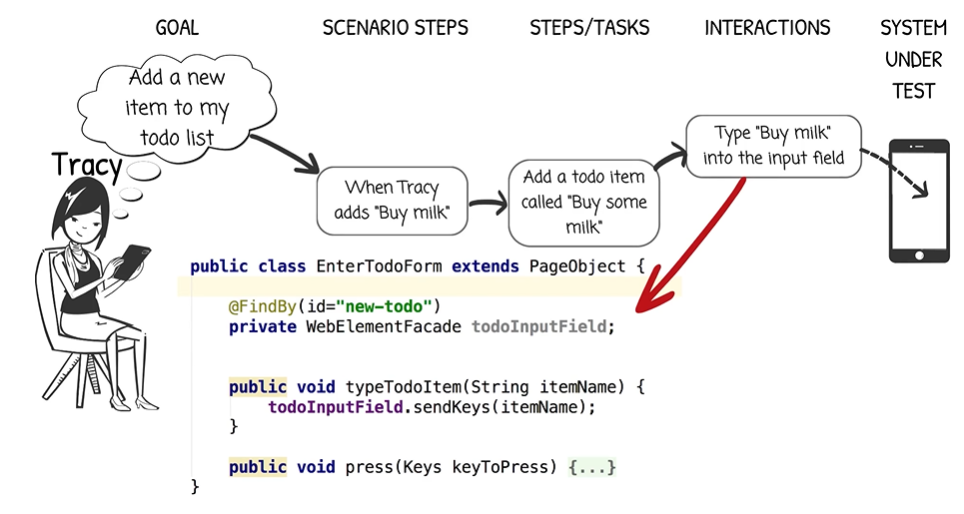
Step library son solo simples clases con la antacion Step en cada método que representa un business step o task.



El rol de un método SET es interactuar con la aplicación. A menudo interactúan con otras



step libraries. En este, queremos tipear un valor dentro de un campo input. En otras palabras, estamos llevando a cabo una interacción con la UI.



Vamos a usar WebDriver para interactuar con la UI pero no lo llamaremos directamente desde este método lo cual nos llevaría a la **duplicación de código** si muchos métodos necesitan llevar a cabo la misma interacción.

Entonces, necesitamos una manera de escribir el código de interacción con la UI en un solo lugar. En la porción de código, hemos usado un patrón llamado **PageObjet**.

PageObjet encapsula las interacciones con un componente particular de la pantalla/página y mantiene el código del webdriver para ese componente en un solo lugar.

Aquí tenemos un pageObjet que representa un campo input de la ToDO list. **Serenity** inyectará el pageObject dentro de la step library por nosotros. El pageObjetc por si mismo utiliza la API del webdriver para identificar e interactuar con elementos de la pagina web. **Defined By** anotation le dice al webdriber donde buscar un elemento en la página. Y en esta clase, el método **typeTodoItem** se encarga precisamente de eso: tipear el valor que nosotros ingresamos en el campo input.