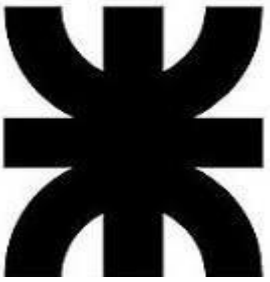
Universidad Tecnológica Nacional
Facultad Regional Córdoba



Ingeniería en Sistemas de Información

Trabajo Práctico

Cátedra: Ingeniería y Calidad de Software.

Tema Entrega: Requerimientos Ágiles – User Stories.

Curso: 4K3

Número de Entrega: 6

Profesores:

- Ing. Covaro Laura.
- Ing. Massano, Cecilia.
- Ávila, Pilar.

Nº Grupo: 6

Integrantes:

- Balmaceda Uema, Florencia — Legajo: 67636 (balmacedauema@gmail.com)
- Campos, Nelson Ramiro — Legajo: 58816 (nelsonramirocampos@gmail.com)
- Capovilla, Luisina — Legajo: 67999 (luisicapovilla@gmail.com)
- Marc, Florencia — Legajo: 60060 (florenciamarc46@gmail.com)
- Luzara Quiroga, Jorge Ezequiel — Legajo: 67788 (ezequielluzara@gmail.com)

Fecha de Entrega: 13 de Septiembre de 2023

ENUNCIADO:

Unidad:	**Unidad Nro. 3: Gestión del Software como producto**
Consigna:	Implementar una User Story determinada usando un lenguaje de programación elegido por el grupo respetando un documento de reglas de estilo.
Objetivo:	Que el estudiante comprenda la implementación de una User Story como una porción transversal de funcionalidad que requiere la colaboración de un equipo multidisciplinario.
Propósito:	Familiarizarse con los conceptos de requerimientos ágiles y en particular con User Stories en conjunto con la aplicación de las actividades de SCM correspondientes.
Entradas:	Conceptos teóricos sobre el tema desarrollados en clase. Definición completa de las User Stories correspondientes al Trabajo Práctico 2 "Requerimientos Ágiles – User Stories y Estimaciones"
Salida:	Implementación de la User Story correspondiente en un programa ejecutable Documento de estilo de código
Instrucciones:	- Seleccionar una User Story a implementar de entre las siguientes opciones: o *Realizar Pedido a Comercio adherido* **(grupos pares)** o *Realizar un Pedido de "lo que sea"* **(grupos impares)** - Seleccionar el conjunto de tecnologías para implementar la funcionalidad elegida. - Buscar y seleccionar un documento de buenas prácticas y/o reglas de estilo de código para el lenguaje de programación a utilizar. - Implementar la US siguiendo las reglas de estilo determinadas.

User stories:

Realizar Pedido a un Comercio adherido **Como Solicitante quiero realizar un Pedido a un comercio adherido para recibir los Productos en mi domicilio** **Criterios de Aceptación** • Se debe indicar la dirección (calle, número, ciudad y referencia opcional en formato de texto). • Se debe poder seleccionar la ciudad de un listado de ciudades disponibles. • Se debe seleccionar la forma de pago: Efectivo o Tarjeta de Débito/Crédito. • Si paga en efectivo debe indicar el monto con el que va a pagar. • Si paga con tarjeta debe ingresar el número de la tarjeta, nombre y apellido del Titular, fecha de vencimiento (MM/AAAA) y CVC. • Debe ingresar cuando quiere recibirlo: "Lo antes posible" o una fecha/hora de recepción. • El Carrito debe contener al menos un Producto del Comercio adherido.	**8**
• Probar realizar un Pedido a un Comercio adherido en efectivo "lo antes posible" (pasa) • Probar realizar un Pedido a un Comercio adherido con tarjeta "lo antes posible" (pasa) • Probar realizar un Pedido a un Comercio adherido programando la fecha/hora de entrega (pasa) • Probar realizar un Pedido a un Comercio adherido con el carrito vacío (falla) • Probar realizar un Pedido a un Comercio adherido con una tarjeta inválida (falla) • Probar realizar un Pedido a un Comercio adherido con una tarjeta de crédito MasterCard (pasa) • Probar realizar un Pedido a un Comercio adherido en efectivo sin indicar el monto a pagar (falla) • Probar realizar un Pedido a un Comercio adherido programando una fecha/hora de entrega no válida (falla)	

Guía de estilo, convenciones y buenas prácticas de desarrollo conJavaScript.

Esta es una guía sobre reglas, convenciones y buenas prácticas usados en el lenguaje de programación Javascript. Está inspirado en varios documentos de autores reconocidos en el ámbito Javascript, y que han sido adoptados por muchas personas, inclusive por grandes entidades tales como PayPal, entre otros.
Además, se tiene una fuerte referencia de los libros “Código Limpio” y “Javascript: The Good Parts”.
En nuestro proyecto usamos framework React Js aplicando este documento.

Convenciones

Para lograr un código limpio, a continuación, se describen las convenciones que se utilizarán en Javascript, independientemente si es en front o back.
Es importante mencionar que dichas convenciones son tomadas de varias referencias, e inclusive algunas convenciones han sido tomadas de Frameworks Javascript.

Espacios en blanco (Identación):

Normalmente se puede trabajar con identación y espacios en blanco en el código y no es importante para el intérprete, puesto que no los toma en cuenta, pero para el programador, esto provee una mejor legibilidad del código.

Entonces la recomendación es que:

- Nunca se debe mezclar espacios y tabulaciones.
- De preferencia, se debe identar con 4 espacios.
- No uses tabulaciones

Por lo general tu editor de texto o IDE, te permite configurar este parámetro y por defecto suele estar configurado con una identación de 4 espacios, pero en el caso de que no sea así, hay que revisar la configuración del IDE o editor de texto y configurarlo.
Para archivos .js que serán servidos públicamente, hay que recordar que el interprete de Javascript no toma en cuenta espacios en blanco o identación, y no hay que preocuparse por el tamaño del archivo inicial, si sos prudente debes realizar procesos de mitificación para crear archivos optimizados en su tamaño.

Longitud de una línea

El límite de las líneas de código deberán ser de 80 caracteres, esto ayudará a una mejor lectura del código. Es seguro que tu editor de texto o IDE soporta esta característica.

Comillas

Aunque Javascript permite usar comillas simples o comillas dobles indistintamente, generalmente es recomendable, escribir con comillas simples y en otros con comillas dobles.
En ciertos casos se desea tener comillas dentro de un texto, y tendrán otras como delimitadores externos del texto.

Es válido:

```
const foo ='"Bienvenidos"'; console.log(foo); //
"Bienvenidos"
//
const foo1 ="'Bienvenidos'"; console.log(foo1); //
'Bienvenidos'
//Otro ejemplo
const foo2 = 'Bienvenidos,';
const foo3 = 'Hola';
console.log(foo2 + foo3 + ' Mundo.'); //Bievenidos, HolaMundo.
```

Como recomendación, se utiliza comillas simples como delimitadores externos.
Solo si estás escribiendo JSON, utiliza comillas dobles, en los demás casos simples.

Recomendado:

```
const  foo  =  'bar';
console.log('bar');
function(foo, bar){
const x = 'hi' + foo + ';' + 'bar';}
```

Llaves

La llave de apertura deberá ir en la misma línea de la sentencia. Si se coloca en la siguiente línea, en algunos casos muy particulares se puede producir algún error. Más adelante se explica dicho error en el uso de punto y coma en las sentencias.

No recomendado:

```
// control flow stament
if ( true )
{
    //codes goes here
}
//Anonymous function declaration
function ( args )
{
      return true;
}
//Named function declaration
function foo()
{
    return true;
}
//Anonymous function expression
const bar = function ( args )
{
    return true;
}
//Arrow function
const bar = (arg) =>
{
return true,
};
const bar = ( args ) =>
{ true };
```

Recomendado:

```
// control flow stament
if ( true ) {
//codes goes here
}


//Anonymous function declaration
function ( args ) {
    return true;
}
//Named function declaration
function foo() {
    return true;
}
//Anonymous function expression
const bar = function ( args ) {
    return true;
}
//Arrow function
const bar = (arg) => {
    return true,
};
const bar = ( args ) => { true };
const bar = ( args ) => true;
```

Punto y coma

Javascript utiliza ASI (Automatic Semicolon Insertion) cuando se trata de insertar un "punto y coma" que ha sido omitido encada instrucción, y que es separada en una línea diferente.

Técnicamente es posible, sin embargo, esta no es una buena práctica, porque en ciertas ocasiones se puede generar un error que será un dolor de cabeza poder solucionarlo.

Por ejemplo, la función getName únicamente nos devolverá una estructura de datos sencilla.

```
console.log(getName());
function getName(){
    return
    {    name: '@davidenq'
    }
}
```

Ejecutando el código se obtendrá el resultado será: "undefined"Cuendo el resultado esperado es {name: '@davidenq'}

Analizando el código

En el instante de ejecutar el código, el intérprete de Javascript detecta donde es necesario colocar un ";" (puntp y coma) y simplemente lo inserta. En este caso el intérprete consideró que después de un return termina la instrucción porque a continuación hay un salto de línea por lo que inserta el ";".

Luego se lee las siguientes líneas y coloca otro ";" al final de la llave.

Al ejecutar el código ya sabemos cuál es resultado: undefined

```
//Esto es lo que ASI interpreta
function getName(){
    return; //mistake
    {
        name: “Bootcampio”
    };
}
```

El error se encuentra después de un return porque la llave de apertura debería colocarse a continuación, y no en la línea siguiente.
Para evitar este tipo de problemas es importante adoptar de manera estricta la convención de apertura de llaves y procurar finalizar una instrucción con un punto y coma cuando sea necesario.

¿Dónde está permitido colocar punto y coma en las instrucciones?

Cuando se trata de algunos flujos de control:

```
//control flow stament
do {
   //codes goes here
} while ( condition ); //necesary semicolon
```

Cuando se trata de expresiones de funciones o funcionesflecha

```
//Anonymous function expression
const bar = function ( args ) {
    //codes goes here
};
//Named function expression
const bar = function foo( args ) {
    //codes goes here
};
//Autoload function
(function( args ){
    //codes goes here
})( args );
//Arrow function
const bar = () => {
    return foo;
};
const bar = () => { foo };
const bar = () => foo;
```

Luego de notación de objetos

```
const params = {
    'a': 1,
    'b': 2,
    'c':'string'
};
```

Luego de un arreglo:

```
const fruits = ['Apple', 'Orange', 'Pear'];
```

Cuando retornas una expresión o resultado:

```
…
return { … };
…
…
return ( … );
…
…
return expression;
…
```

Ámbito de las variables (Scope) y Hoisting

Cuando se habla del ámbito de las variables (scope), hay que tener en cuenta si es un ámbito global o local. Normalmente hablamos de un ámbito local cuando dichas variables están contenidas dentro de una función. Entonces dicha función sería el contenedor de la variable, siendo este su scope. Esta variable no tiene ningún valor fuera de su contenedor.

Veamos con ejemplos más claros acerca de estos conceptos.

```
var day = ‘Saturday’;
function showDay() {

console.log(day); //
      var day = ‘Monday’;
      console.log(day); //
}
showDay();
console.log(day);
```

Al ejecutar el código, automáticamente el intérprete de Javascript moverá la variable x a la parte superior del ámbito contenedor (Esto es lo que se conoce cómo Hoisting).

```
var day = ‘Saturday’;//Declaración uno
function showDay(){
      var day;//Fue movida por el interprete Javascript. Esto nose ve
                  //reflejado en el editor de texto.
      console.log(day);//primer console.log
      var day = ‘Monday’;//Declaración dos
      console.log(day);//segundo console.log
}
showDay();
console.log(day)
```

El resultado obtenido en consola será:

```
>_ undefined
>_ Monday
>_ Saturday
```

Aunque se haya declarado la variable day antes de la función y seteado con un valor, el primer console.log muestra el resultado undefined y no Saturday como se esperaba. Esto se debe a que la declaración uno es realizaba en el ámbito global, y la declaración dos en el ámbito de su contenedor, que es la función misma. Por lo tanto, la declaración dos es movida por Javascript al inicio de su ámbito contenedor, haciendo que esta variable se setee a undefined, posteriormente es seteada a Monday y finalmente termina su ámbito contenedor y day y vuelve a tener el valor Saturday seteado inicialmente.
Obviamente, este tipo de declaraciones no suele llevarse a cabo cuando se está escribiendo código. Es decir, declarar dos veces la misma variable en dos ámbitos diferentes; normalmente no suele ocurrir, pero podría darse el caso. No obstante, a modo de ejemplo, vemos que Javascript lo permite sin que saltee algún error por declarar repetidas veces una misma variable. Esto cambia en ECMASCRIPT6 con las palabras reservadas let y const.

Ahora veamos un caso más práctico. Supongamos que vamos a recorrer un array:

```
for(var i = 0; i < 3; i++){
    var msg = 'Hi';
    console.log(msg + i);
}
function foo(){
    var bar = 'bar';
}

console.log(i);
console.log(msg);
console.log(bar);
```

El resultado obtenido en consola será:

```
>_ Hi0
>_ Hi1
>_ Hi2
>_ 3
>_ Hi
>_ bar is not defined
```

Se supone que console.log(i) y console.log(msg) no deberían mostrar resultado alguno, es más debería arrojar algún error en consola como console.log(i) y console.log(msg). Por otro lado, console.log(bar) si arroja un error.
Esto en otros lenguajes de programación fuertemente tipados, no estaría permitido; arrojando siempre una excepción.
En el caso de for, debería haber arrojado como resultado undefined puesto que suponemos el ámbito de la variable declarada i es la sentencia for(), y no transciende más allá de su contenedor.
Sin embargo, el resultado fue 3. Lo mismo sucede con la variable msg.
Esto se debe a lo ya mencionado; Javascript mueve todas las variables a la parte superior del ámbito. Pero aquí está la diferencia; las funciones for, if, while, switch; ellas mismas no delimitan un ámbito, por lo tanto, las variables que se declaren dentro formarán parte de un ámbito mayor; global o a nivel de función.

Entonces la solución a esto sería la siguiente:

```
for(let i = 0; i < 3; i++){
    const msg = 'Hi';
    console.log(msg + i);
}
function foo(){
    const bar = 'bar';
}
console.log(i); //ReferenceError: i is not defined console.log(msg);
//ReferenceError: msg is not definedconsole.log(bar); //ReferenceError: bar is
not defined
```

Declaración de variables

Teniendo en cuenta todo lo mencionado anteriormente en el ámbito de las variables y hoisting, ahora es recomendable usar const y let en lugar de var.

Se podría optar por declarar las variables en la parte superior de su ámbito aunque Javascript no requiera esto. Tratando de que su ámbito sea local y no global. Lo que se obtiene con esto es lo siguiente:

- Código más limpio
- Proporciona un único punto de búsqueda de las variables que son utilizadas en un ámbito.
- Ayuda a conocer que variables están involucradas en el ámbito contenedor.

Formato de declaración

Esto nuevamente, es un punto discutible y cuestión de gustos de cada uno, pero sirve como buena referencia dado que muchos optan por la opción recomendada, y otros prefieren las opciones no recomendadas.

En fin, si estás siguiendo esta guía, mi sugerencia y no una obligación, es que adoptes por la opción recomendada. Sino es así, no hay problema alguno, en este caso es solo cuestión de gustos.

No recomendado:

```
// separar por comas
const day = 'Monday',
      count = 10;

//Alinear
const day       = 'Monday',
      count = 10;

//En la misma línea
const day = 'Monday', count = 10;//En la misma línea
const day = 'Monday', count = 10;
```

Recomendado:

```
const day = 1; const
count = 10;
const keys = ['foo', 'bar'];const values =
[1, 2];
```

Declaración de variables fuera de las sentencias

No recomendado:

```
/*
No hagas que el ciclo for lea repetidas veces la longitud delarray. Solo y solo si estás
utilizando colas y/o pilas en el mismo array sería recomendable dado que el array va
disminuyendo y/o aumentando su tamaño. Caso contrario no lo hagas.
*/
for ( var i = 0; i < fruits.length; i++ ) {
      var type = //something do;
      console.log(type);
}
```

Recomendado

```
const sizeArray = fruits.length;
let type;

for ( i = 0; i < sizeArray ; i++ ) {
      type = //something do;
}
```

Declaración de variables dentro de un contenedor(Reduciendo variables globales)

No recomendado:

```
const basket = ['orange', 'apple', 'apple', ….];const fruit = 'apple';
const numberApples = 0;
console.log(countApples(basket));
unction countApples ( fruits ) {
for ( const i = 0; i < fruits.length; i++ ) {
            if( fruit === fruits[i] ){
                  numberApples++;
            }
      }
return numberApples;
}
```

Recomendado:

```
const basket = ['orange', 'apple', 'apple', ….];
console.log(countApples(basket));

function countApples ( fruits ) {
const fruit = 'apple';
      let numberApples = 0;
      const sizeArray = fruits.length;

for ( const i = 0; i < sizeArray ; i++ ) {
            if( fruit === fruits[i] ){
                  numberApples++;
            }
      }
      return numberApples;
}
```

Comparación de variables
Si de comparar true o false se trata, procura seguir la siguienteconvención.

No recomendado:

```
if ( x === true ) {
      //codes goes here
}

//or

if ( x === false ) {
      //codes goes here
}
```

Recomendado

```
if ( x ) {
      //codes goes here
}
//or
if ( !x ) {
      //codes goes here
}
```

Otras comparaciones

```
const x = 10;
```

Operador	Descripción	Comparación	Resultado
==	igual valor	x == 10	true
==	igual valor	x == 15	false
==	igual valor	x == "10"	true
===	igual valor e igual tipo	x === 10	true
===	igual valor e igual tipo	x === 15	false
===	igual valor e igual tipo	x === "10"	false

Procura utilizar una comparación estricta de valor y tipo con eloperador según corresponda=== or !==

Evaluación condicional

No recomendado:

```
// Evaluando si el array tiene elementos
if ( array.length > 0) {
    //codes goes here
}
// Evaluando si el array no tiene elementos (está vacío)
if ( array.length === 0 ) {
    //codes goes here
}
// Evaluando que un string no es vacío
if( string !== “” ) {
    //codes goes here
}
```

Recomendado: Evaluar por el valor true o false de la expresión

```
// Evaluando si el array tiene elementos
if ( array.length ) {
    //codes goes here
}
// Evaluando si el array no tiene elementos (está vacío)
if ( !array.length ) {
    //codes goes here
}
// Evaluando que un string no es vacío
if ( string ) {
    //codes goes here
}
// Evaluando que un string es vacío
if ( !string ) {
    //codes goes here
}
```

Notaciones cortas

No declares las variables primitivas como objetos, puesto que relentizan la ejecución del código y produce efectos no deseados.

Ejemplos tomados de www.w3schools.comPor ejemplo:

```
const x = “John”;
const y = new String(“John”);
(x === y) // is false because x is a string and y is anobject.

const x = new String(“John”);
const y = new String(“John”);
(x == y) // is false because you cannot compare objects.
```

No recomendado

```
const name = new String(“John”);
```

Recomendado

```
const name = “John”;
```

No recomendado

```
const lunch = new Array();lunch[0]='Dosa';
lunch[1]='Roti';
lunch[2]='Rice';
lunch[3]='what the heck is this?';
```

Recomendado

```
const lunch = [
    'Dosa',
    'Roti',
    'Rice',
    'what the heck is this?'
];
```

No recomendado

```
const o = new Object(); o.name = 'Jeffrey';
o.lastName = 'Way'; o.someFunction = function()
{
    console.log(this.name);
}
```

Recomendado

```
const o = {
    name: 'Jeffrey',
    lastName = 'Way',
    someFunction : function() {
        console.log(this.name);
    }
};
```

Formato de nombres de variables yfunciones

Utiliza camelCase para nombrar funciones, declaración de variables,instancias, etc.

Por ejemplo:

- functionNamesLikeThis;
- variableNamesLikeThis;
- methodNamesLikeThis;

Utiliza PascalCase para nombrar constructores, prototypes, clases, etc

- ConstructorNamesLikeThis;
- EnumNamesLikeThis;

En variables

Para declarar valores constantes utiliza Mayusculas y separado por unguión bajo cada palabra.

No recomendado

```
const symbolic_constants;const
```

symbolicConstants;

Recomendado
const SYMBOLIC_CONSTANTS_LIKE_THIS;

Para declarar valores variables utiliza camelCas

No recomendado
const *admin_user;*
const *days_since_creation;*

Recomendado
const adminUser;
const daysSinceCreation;

En funciones

Sean estas anónimas o con nombre.Por ejemplo:

```
//Anonymous function expression
const foo = function () {
      return "anonymous function";
}
console.log(foo()); // 'result anonymous function'

//Named function expression
const foo = function bar(){
      return "named function";
}

//Arrow function

const foo = () => {
      return "arrow function";
};

console.log(bar()); //'result bar is not defined'console.log(foo()); //'result named
function' console.log(foo); //'result [Function:bar]'
```

Nombres y comentarios en ingles

En general, los lenguajes de programación basan su vocabulario en el idioma inglés, puesto que es el lenguaje másutilizado en el mundo. Esta también es una recomendación discutible. Generalmente la recomendación es que se escriba tanto comentarios, como nombres de variables, de funciones y demás, en inglés. Sin embargo si te sientes mejor escribiendo en el idioma que más lo utilizas, hazlo, pero ten en cuenta que si compartes un módulo o librería probablemente no tenga un alcance global.

Nombres con sentido

La intención de asignar un nombre con sentido a carpetas, archivos, variables, funciones, etc, es indicar cuál es su cometido, de tal manera que quede claro el propósito por el cuál fue creado, y es que esta tarea se realiza constantemente, y toma su tiempo porque muchas de las veces cuesta mucho saber que nombre se le puede asignar para indicar de forma directa su intención y

cometido. Pero también tiene sus beneficios asignar nombres coherentes, puesto que más adelante será menor el esfuerzo cuando se vuelva a leer el código pues facilita su compresión.

Por ejemplo:

```
const d; //Elapsed time in days
//or
const day;//Elapsed time
```

Las variables d o day no refleja absolutamente nada, puesto que no sabemos cuál es la intención de querer saber los díastranscurridos.

```
const daysSinceCreation;
int daysSinceModification;
```

Comentarios

Procura asignar comentarios solo cuando sea necesario y queayuden a entender mejor el código.

no recomendado:

```
//day of the monthconst
dayOfMonth;

/*
*    Return
*    @return: day of the month
*
* /
function getDayOfMonth(){
…
}
```

Es claro que muchas de las declaraciones son obvias. Sin embargo, es lo que se suele encontrar en el código, no olvidar que declarar nombres con sentido ayudan a la legibilidad del código y por ende a evitar comentarios innecesarios.

Recomendado:

```
const dayOfMonth;
function getDayOfMonth(){
        …
}
```

En el ejemplo anterior son obvias las intenciones solo con leer el nombre y no falta asignar algún comentario para dar a entender el cometido. No hay que redundar.

Asignaciones y menciones

No agregues menciones en el código que has modificado (eliminado o agregado). Deja que el sistema de control de versiones mantenga esta información fuera del código.

Si desarrollas una librería para el sistema y que pudiera ser liberado y utilizado por otros sistemas, agrega tu información. No hay un esquema básico, sin embargo, a continuación se muestra un ejemplo.

```
/**
```

```
* Descripción general de lo que hace la librería
*
* @author:
* @email:
* @param: {Tipo} Descripción
* @return:{Tipo} Descripción
* @module: nombre del módulo o libreria
* @licencia:
*
*/
```

Referencias bibliográficas

Enlaces web

- https://github.com/rwaldron/idiomatic.js/tree/master/translations/es_ES
- https://mhdev.readthedocs.org/es/latest/js-style.html
- http://www.w3schools.com/js/js_scope.asp
- http://code.tutsplus.com/tutorials/24-javascript-best-practices-for-beginners–net-5399
- https://www.thinkful.com/learn/javascript-best-practices-1
- https://www.thinkful.com/learn/javascript-best-practices-2

Finalmente, para facilitar el control de buenas prácticas se utiliza un analizador de código estático muy utilizado llamadoESLint. Este software es fácilmente instalable y configurable, además se integra con una amplia gama de IDEs y ya viene con varias configuraciones disponibles que cumplen con las convenciones y reglas antes mencionadas.

Para más información acerca de ESLint te dejamos la url:
https://eslint.org/docs/user-guide/getting-started