



# Introducción al Pensamiento Computacional - Semana 12



© Todos los derechos reservados Universidad Rafael Landívar URL.

---

## DESEMPEÑOS ESPERADOS

- ☰ Desempeños esperados

---

## DESARROLLO DE CONOCIMIENTOS

- ☰ Introducción
- ☰ Anticipar defectos
- ☰ Verificación y validación
- ☰ Probar las partes
- ☰ Probar el todo
- ☰ Debugging

## APLICANDO LO APRENDIDO

---

☰ Actividad 1

☰ Actividad 2

☰ Actividad 3

☰ Actividad 4

## DE LA TEORÍA A LA PRÁCTICA Y REFLEXIÓN

---

☰ Recursos

## CRITERIOS DE EVALUACIÓN

---

☰ Rúbrica de evaluación

☰ Diario de experiencias de laboratorio

## FUENTES DE REFERENCIAS

---

☰ Referencias

## CRÉDITOS

---

☰ Créditos

# Desempeños esperados

---



## El estudiante:

- 1 Identifica los tipos de errores que pueden ocurrir al programar.
- 2 Utiliza excepciones para capturar un defecto de comportamiento del sistema.
- 3 Explica cómo probar las partes individuales de una solución por medio de pruebas unitarias.
- 4 Conoce aspectos emergentes de las soluciones y cómo probarlos.

5

Explica cómo probar una solución completa de un sistema y su aceptación.

6

Utiliza métodos para localizar errores en las soluciones utilizando bitácoras y debugging.

# Introducción

---

## Introducción



Existen muchos aspectos para evaluar programas. Principalmente el comprobar si una solución cumple con resolver el problema original sin cometer errores. Se realiza por medio de las pruebas.

Las respuestas que los equipos de pruebas deben contestar son: ¿es estable? ¿Lo realiza de manera efectiva? ¿Previene accesos no esperados?

En todos los casos, como se trata con programación, la evaluación puede ser automatizada, es decir, se puede escribir código que permita probar otro código.



## Anticipar defectos

---



## Anticipar defectos

Algunos defectos entran en los programas antes de realizar las pruebas o de codificar. Es decir, cuando se realiza el diseño. Identificarlos a tiempo es clave. Adicionalmente, hay otros defectos que se realizan en la etapa de codificación. Algunas estrategias para anticiparlos son las siguientes.

### Errores sintácticos y semánticos

Los **errores sintácticos** ocurren cuando un enunciado está mal escrito y no puede ser comprendido por la computadora. La sintaxis del lenguaje de programación describe las reglas que los enunciados deben seguir. Se debe respetar la sintaxis del lenguaje de programación utilizado. En la siguiente figura, cada línea de código presenta un error de sintaxis en Python.

```
class VendingMachine
# Error: Missing colon after class name

machine_name = SuperVendor VX-9000'
# Error: Missing opening quote mark

def __init__(self
# Error: Missing closing parenthesis

3rd_compartiment = get_compartiment(3)
# Error: Variable names cannot begin with a number
```

Cuando la computadora trata de correr un programa que contiene errores de sintaxis, automáticamente parará y reportará el problema. Por ejemplo:

```
File 'VendingMachine.py', line 1
  class VendingMachine
                         ^
SyntaxError: invalid syntax
```

Afortunadamente estos errores se corrigen rápidamente y se evitan cometer conforme se gana experiencia en el lenguaje de programación.

Los errores semánticos ocurren cuando se escriben enunciados que siguen las reglas del lenguaje, pero que son inválidos. Por ejemplo:  $x = y / z$

Es sintácticamente correcta, pero con ciertos valores puede producir errores. En el caso que  $z$  sea igual a 0, no tiene sentido matemáticamente, ya que la división dentro de 0 es indefinida.

Los errores de sintaxis se pueden detectar antes de que el código es ejecutado, los errores semánticos son detectados cuando se está ejecutando el programa y colapsa en cierto punto. En el ejemplo anterior, si se programa en Python, daría el siguiente mensaje al momento de ejecutar:

```
>>> 42 / 0
      File '<stdin>', line 1, in <module>
ZeroDivisionError: division by zero
```

Los errores de la naturaleza semántica con los que pueden consumir más tiempo.

## Evitando defectos

Ya que los errores semánticos se pueden reconocer solo en la ejecución, es importante realizar previsiones de cómo tratarlos, agregando código extra al programa. Ese código le indica a la computadora qué hacer en caso de que ocurra un error. Es una forma segura que no contribuye a la solución, pero permite prevenir defectos. Algunas estrategias que se pueden seguir son las siguientes:

## Capturar errores potenciales

---

Utilizando excepciones, que son detecciones de error durante la ejecución de un programa. En C# se utiliza el bloque try-catch.

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch>

Esta es un enfoque **optimista**. Quien programa dice “este pedazo de código puede dar problemas, pero tratemos de todas formas y si uno sucede lo manejamos”.

```
namespace ErrorHandlingApplication {
    class DivNumbers {
        int result;

        DivNumbers() {
            result = 0;
        }
        public void division(int num1, int num2) {
            try {
                result = num1 / num2;
            } catch (DivideByZeroException e) {
                Console.WriteLine("Exception caught: {0}", e);
            } finally {
                Console.WriteLine("Result: {0}", result);
            }
        }
        static void Main(string[] args) {
            DivNumbers d = new DivNumbers();
            d.division(25, 0);
            Console.ReadKey();
        }
    }
}
```

## Programación defensiva

---

La motivación detrás de la programación defensiva es la mitigación de errores, como se estudió en el módulo 5. Mientras que los bloques try-catch aplican un enfoque optimista, algunos casos requieren un enfoque **pesimista**. Esto implica validar ciertas condiciones antes de intentar realizar acciones erróneas. Por ejemplo, esta función en Python convierte grados Celsius a Farenheit.

```
def celsius_to_fahrenheit(celsius):
    if celsius < -273:
        raise ValueError('Temperature less than absolute zero was
reported.')

    return celsius * 1.8 + 32
```

## La Física indica

La Física indica que el cero absoluto se da a -273 Celsius aproximadamente, por lo que la medida de temperatura no puede ser leída más bajo que eso. La función puede levantar su propia excepción.

Cómo crear las propias excepciones en C#:

<https://docs.microsoft.com/en-us/dotnet/standard/exceptions/how-to-create-localized-exception-messages>

```
>>> celsius_to_fahrenheit(-1000)
Traceback (most recent call last):
  File '<stdin>', line 1, in <module>
  File '<stdin>', line 3, in celsius_to_fahrenheit
ValueError: Temperature less than absolute zero was reported.
```



## Verificación y validación

---



Evaluar una solución de software debe responder dos preguntas.

1

¿Hemos construido el producto correctamente? Se comprueba el proceso. A esto se llama verificación.

2

¿Hemos construido el producto correcto? Se comprueba el producto. A esto se la llama validación.

## VERIFICACIÓN

## VALIDACIÓN

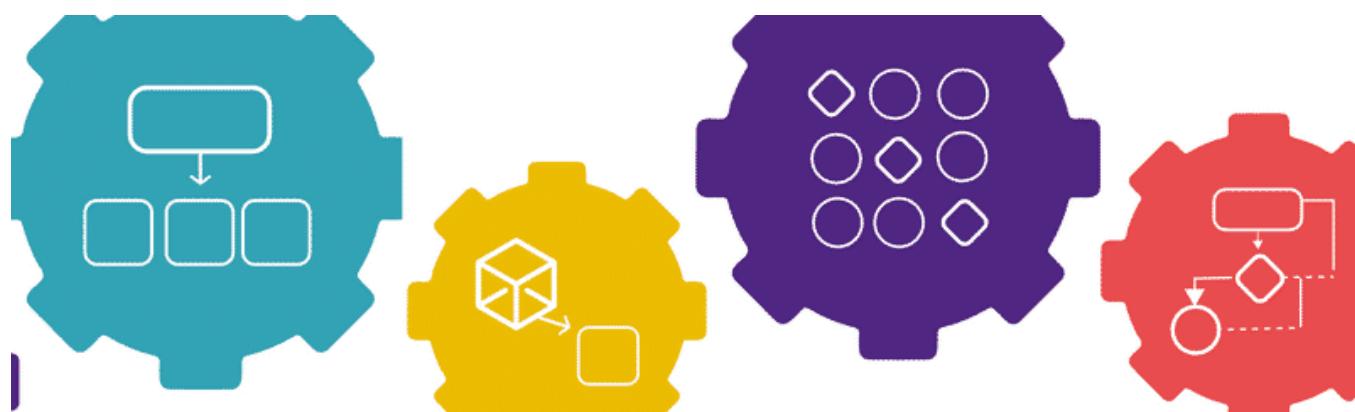
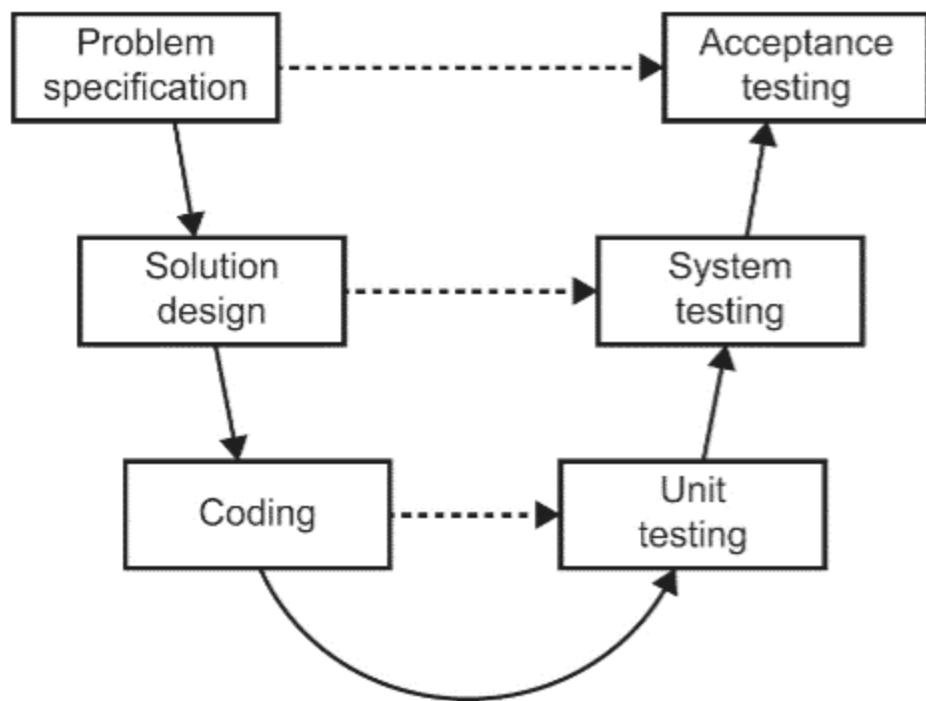
La **verificación** cubre consideraciones técnicas, es decir, si se ha construido una solución de calidad en sus componentes. Responde a preguntas como: ¿está libre de errores? ¿Es confiable? ¿Es seguro? Es importante identificar los atributos de calidad a probar para cada solución. Por ejemplo, para un videojuego es más importante el rendimiento que la seguridad. Estos atributos de calidad deben estar considerados desde la definición de la solución.

## VERIFICACIÓN

## VALIDACIÓN

La **validación** implica comprobar si la solución resuelve el problema original en cooperación con el usuario. Al final de cuentas, es el usuario quien aplicará la solución al problema, y son los usuarios quienes comprueban el resultado. Si un ingeniero diseña un bello puente para caminar a pie, pero el cliente había pedido rieles para tren, fallará en la validación.

Ambas operaciones se realizan a medida que se desarrolla la solución, como muestra la siguiente figura.



## Probar las partes

---



## Probar las partes

En esta sección se describe cómo realizar los casos de pruebas para probar partes de la solución. Aplicando el enfoque bottom-up discutido en el módulo 5, se comprueban las unidades más pequeñas de funcionalidad antes de moverse a piezas más grandes. Esto permite establecer qué partes funcionan por sí mismas, como se esperaba. La clave de este enfoque es localizar un problema rápidamente e implica crear emulaciones de partes de la solución que aún no existen.

## **Pruebas unitarias**

Para crear pruebas unitarias, se recomiendan los siguientes pasos.

## Paso 1

Establecer los objetos de soporte que requiere la unidad de código que se probará.

## Paso 2

Establezca los resultados deseados de la unidad de código.

### Paso 3

Ejecute la unidad para que realice su función.

#### Paso 4

Verifique que el resultado de la función coincida con los resultados esperados.

## Paso 5

Elimine cualquier objeto de soporte que haya creado.

## Paso 6

### Ejemplo

A continuación, un ejemplo de pruebas unitarias en C# <https://docs.microsoft.com/es-es/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022>

## Algunas recomendaciones:

- Antes de ejecutar la prueba, el sistema debe colocarse en estado inicial.
- Después de cada corrida se deben limpiar los datos que se hayan generado, para prevenir que los resultados de una prueba afecten a otra.



## Probar el todo

---



## Probar el todo

Además de las pruebas unitarias, es posible probar la solución como un todo. Recordando la verificación y la validación:

- La verificación se asegura que se haya construido una solución de calidad, con los pasos correctos.
- La validación se asegura que la solución creada en realidad resuelva el problema original.

## **Verificación (pruebas de sistema)**

Las pruebas unitarias son valiosas y necesarias, pero solo muestran como los elementos individuales de una solución funcionan como se espera. Cuando se prueba todo el sistema, se debe asegurar que todas las unidades de código cooperen y entreguen la solución exitosamente. A esto se le llama pruebas de sistema.

Para las pruebas de sistema es posible generar unidades de pruebas, como las realizadas para las unitarias, con la diferencia que se invocan a componentes intermedios o finales de la solución y se valida el resultado de la orquestación de llamadas internas. También son posibles de realizar manualmente.

## **Validación (aceptación del sistema)**

La validación es la otra cara de las pruebas a nivel de sistema. Responde a la pregunta si la solución resuelve el problema original. Se le llaman pruebas de aceptación. Un sistema puede fallar las pruebas de aceptación porque el problema original no fue comprendido y la solución resultante no lo representa.

Las pruebas de validación pueden ser realizadas de manera automática o manual. Es más importante para los usuarios validar un sistema manualmente por ellos mismos para asegurarse que cumple sus necesidades. Puede ser realizado de manera personalizada, reuniendo a los usuarios y permitiéndoles que interactúen con el sistema. También puede ser más formal, entregando a los usuarios un guion similar al plan de pruebas.

Prueba No.	Aspecto para probar	Descripción de la prueba	Salida esperada	Salida actual
1	Depósito de monedas	Insertar una moneda de 50 centavos.	La moneda es aceptada. 0.5 aparece en la pantalla.	Se comportó como se esperaba
2	Depósito de monedas	Insertar una moneda extranjera.	La moneda es rechazada. Despegar el mensaje "La moneda no es reconocida".	Falló. La moneda fue rechazada, pero no desplegó el mensaje.
3	Teclado	Ingresar una selección válida del No. 15.	Desplegar el mensaje: "Ud. ha elegido chocolate".	Se comportó como se esperaba.
4	Teclado	Ingresar una selección inválida del No. 99.	Desplegar el mensaje: "Selección inválida, pruebe de nuevo".	Se comportó como se esperaba.

No se debe esperar hasta que el sistema esté finalizado para realizar las pruebas de aceptación. En caso la solución completa no sea aceptada, tendría que retrabajar mucho en

poco tiempo. Puede comprobar versiones incompletas, pero funcionales tempranamente (pruebas alfa y beta).



# Debugging

---



En el módulo 5 se describieron estrategias para localizar defectos. En este módulo se desea poner en práctica algunas de estas estrategias.

## Bitácoras

---

Dejar un registro en bitácoras permite registrar mensajes en ciertos puntos de la ejecución de la programación. Si se dejan instrucciones de registro en puntos sensibles del código, la computadora dejará un rastro de mensajes, es decir, una bitácora. **La bitácora se realiza para los ojos de quien programa, no de los usuarios, ya que puede tener detalles técnicos.**

Las bitácoras pueden indicar lo que ha pasado y permitir a quien programa, seguir pistas como un detective para reconstruir las condiciones de la falla y resolver el misterio.

Las bitácoras son muy útiles para:

- Registrar valores de variables en ciertos puntos de la ejecución.
- Informar si una instrucción se ejecutó o no.
- Reportar información sobre una excepción ampliando detalles en un bloque de excepción.
- Registrar detalles de eventos significativos, como actualizaciones a la base de datos o fallos en intentos de login.

Evite imprimir en pantalla los registros de la bitácora, utilice una base de datos, un sistema de archivos o los registros del sistema operativo. Recuerde grabar la fecha y hora en que se registra cada evento.

Los tipos de registro en la bitácora varían según el caso y su uso, como lo muestra la siguiente tabla.

<b>Nivel de severidad</b>	<b>Caso en el cual se recomienda utilizarlo</b>
<b>Debug</b>	Información muy detallada, de interés para quien diagnostica el problema.
<b>Información</b>	Eventos que confirman que una operación trabaja como se esperaba.
<b>Advertencia</b>	La solución funciona como se esperaba, pero sucedió algo que puede causar error.
<b>Error</b>	Un problema ha ocurrido y el programa fue incapaz de ejecutar una función o pieza de código.
<b>Crítico</b>	Un problema serio ha ocurrido, lo que implica que el programa no pueda seguir ejecutándose.

## Usar un seguimiento paso a paso (debugger)

Las bitácoras van dejando pistas, pero el debugging permite conocer la naturaleza del error en tiempo real. Leer una bitácora es como leer la narración de un partido de football en el periódico. Debuggearla, es como visualizar nuevamente el juego de manera interactiva. Debugging permite visualizar qué sucede, paso a paso, desde el punto de su elección.

Para debuggear se debe conocer la forma en que se realiza en cada entorno del lenguaje de programación, enfocándose en:

- Empezar el debugger dentro de su programa.
- Ejecutar el programa paso a paso.
- Desplegar los valores de las variables en cierto punto de la ejecución.

Para conocer cómo se hace en C# ingresar a: <https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-debugger?view=vs-2022>



## Actividad 1

---



The background of the slide features a close-up photograph of two interlocking gears. One gear is a solid blue color, and the other is a solid yellow color. They are shown from a perspective angle, with light reflecting off their metallic surfaces.

## Actividad 1



## Instrucciones

Marque los siguientes enunciados si son verdaderos o falsos.

Un error de sintaxis no será reportado hasta que la línea que lo contiene se ejecute.

Verdadero



Falso

SUBMIT

El código en el bloque "finally" se ejecutará siempre después de try.

---



Verdadero



Falso

SUBMIT

La validación determina si una solución resuelve el problema original.

---



Verdadero



Falso

SUBMIT

Es posible realizar pruebas unitarias de manera automática.

---



Verdadero



Falso

SUBMIT

Los registros de las bitácoras deben desplegarse en pantalla.

---



Verdadero



Falso

SUBMIT

## Actividad 2

---



Actividad 2



## Instrucciones

Recuerda el ejercicio 5, del módulo 5: Escriba un algoritmo para el juego FizzBuzz que analice los números ingresados y devuelva una respuesta correcta. Si el número no está en el rango esperado, el algoritmo debe devolver un mensaje de error.

Convierta su algoritmo en un programa en C#.

Diríjase a la plataforma para subir su actividad en el espacio correspondiente.

## Actividad 3

---



Actividad 3

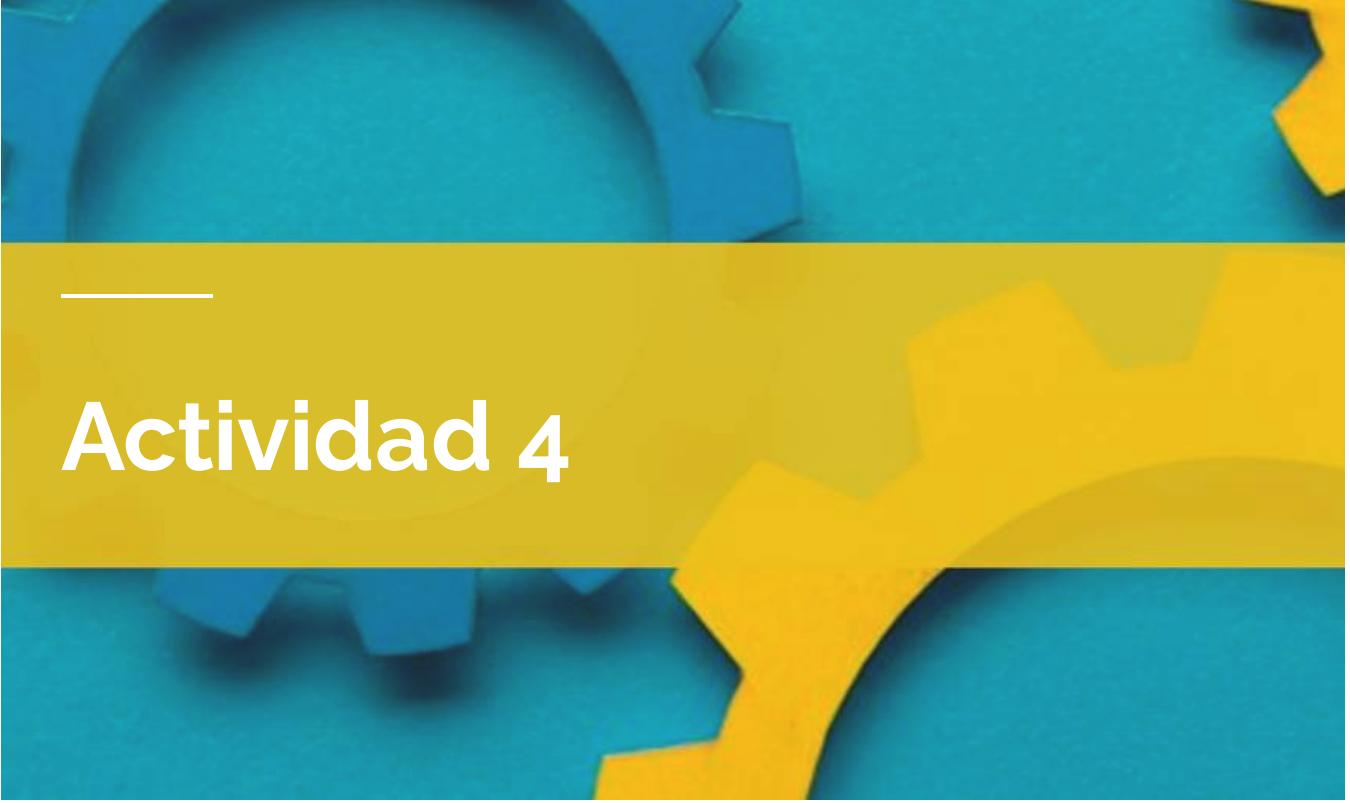


Realice una prueba unitaria para el programa FizzBuzz. Asegúrese de probar todas las posibilidades.

Diríjase a la plataforma para subir su actividad en el espacio correspondiente.

## Actividad 4

---



Actividad 4



## Instrucciones

El siguiente código implementa el juego de Ahorcado en Python.

```
word = 'underutilise'

guesses = []
user_input = ''

while user_input != '0':
    user_input = input('Enter a letter, or 0 to give up:')
    guesses.append(user_input)
    output = ''
    for letter in range(1, len(word)):
        if word[letter] in guesses:
            output = output + word[letter]
        else:
            output = output + '_'
    print(output)
    if output == word:
        print('You win!')
        break

print('Game over!')
```

Al ejecutarlo debería verse así:

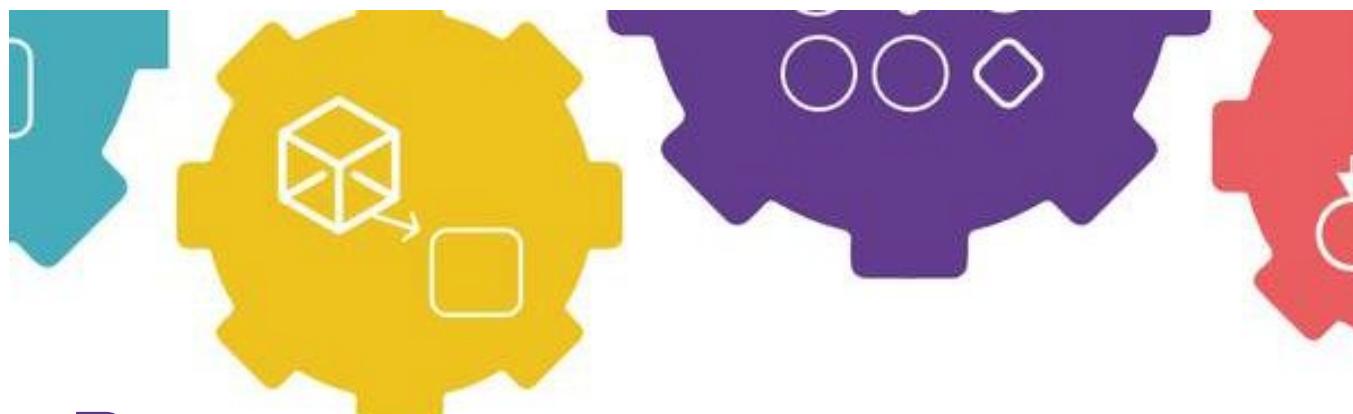
```
Enter a letter, or 0 to give up: u
u__u__
Enter a letter, or 0 to give up: e
u_e_u___e
Enter a letter, or 0 to give up: t
u_e_ut___e
Enter a letter, or 0 to give up:
```

Sin embargo, el código tiene unos defectos. Pruebe el programa en C# y usando el debugger y/o bitácoras, encuentre y corrija los problemas.

Diríjase a la plataforma para subir su actividad en el espacio correspondiente.

## Recursos

---



## Recursos

### Instrucciones:

Haber instalado el lenguaje de programación en las computadoras que utilizará para realizar las prácticas.



**Calculadora.txt**

3 KB



## Rúbrica de evaluación

---



Rúbrica



Descargue la siguiente rúbrica de evaluación.



**Rúbrica de Evaluación.pdf**

162.6 KB



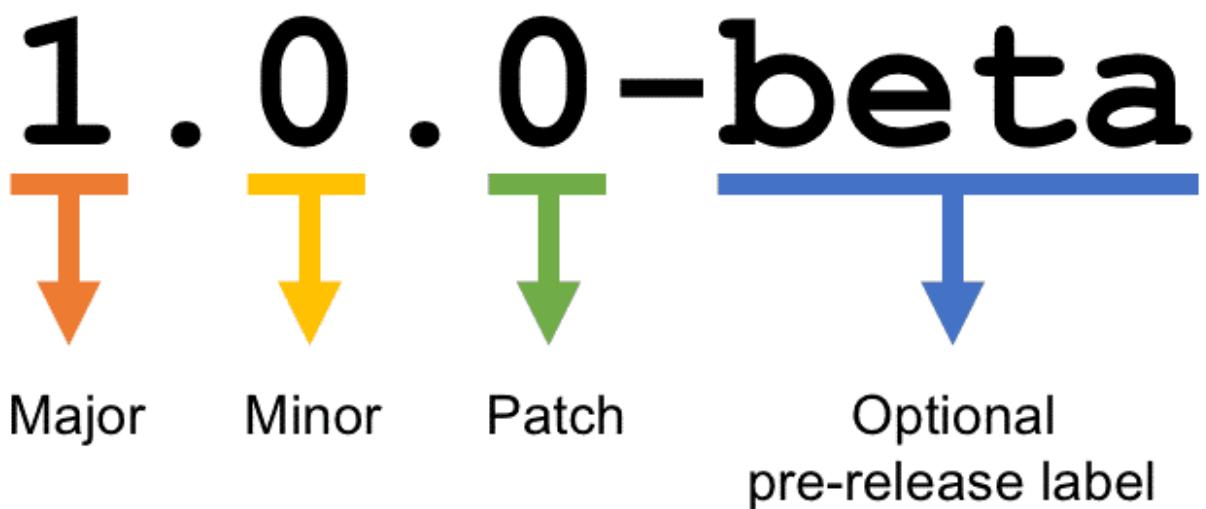
## Diario de experiencias de laboratorio

---



# Diario de experiencias del laboratorio

Cada práctica de laboratorio deberá publicarse en una carpeta en el diario de experiencias de laboratorio, bajo la versión 1.0.0.



# Referencias

---

## Referencias

Bordigón, F. e Iglesias, A. (2020). Introducción al pensamiento computacional. Universidad Pedagógica Nacional, Educar Sociedad del Estado y el Ministerio de Educación Argentina.

Beecher, K. (2017). Computational Thinking. A beginner's guide to problem-solving and programming.

ISTE (2018). Computational Thinking. Meets Students Learning. International Society for Technology in Education.

Microsoft (2022). Documentación de C#. <https://docs.microsoft.com/es-es/dotnet/csharp/>

MIT (2022). Computational Thinking, a live online Julia/Pluto textbook. Julia: A Fresh Approach to Computing. computationalthinking.mit.edu. Massachusetts Institute of Technology

Pourbahrami & Tritty (2018). Computational Thinking: How Computer Science Is Revolutionizing Science and Engineering. ENGenious (15) 8-11. <https://resolver.caltech.edu/CaltechCampusPubs:20181025-110029157>.

## Créditos

---

