



# Introducción al Pensamiento Computacional - Semana 5



© Todos los derechos reservados Universidad Rafael Landívar URL.

---

## DESEMPEÑOS ESPERADOS

---

- ☰ Desempeños esperados

---

## DESARROLLO DE CONOCIMIENTOS

---

- ☰ Marco de referencia de los defectos y errores
- ☰ Tipos de defectos
- ☰ Mitigación de errores
- ☰ Pruebas de calidad (testing)
- ☰ Seguimiento paso a paso (debugging)
- ☰ No se puede atender todo: decidir qué errores se corregirán

☰ A manera de síntesis

☰ Recursos complementarios

#### APLICANDO LO APRENDIDO

---

☰ Actividad 1

☰ Actividad 2

☰ Actividad 3

☰ Actividad 4

☰ Actividad 5

#### 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 Explica las técnicas para prevenir errores en las soluciones planteadas.
- 2 Discute cómo mitigar el efecto de los errores.
- 3 Utiliza un enfoque sistemático para realizar pruebas de calidad por medio de estrategias top-down y bottom-up.
- 4 Muestra cómo realizar pruebas unitarias de las partes de la solución de manera aislada.

5

Identifica el número de estrategias de seguimiento paso a paso (debugging).

6

Explica cómo manejar los defectos encontrados en una solución.

## Marco de referencia de los defectos y errores

---



# Marco de referencia de los defectos y errores

---

“Me he dado cuenta de que gran parte de mi vida la dedicaré a encontrar errores en mis programas”

- Wilkes (1995)

---

Las personas deben hacerse a la idea que las soluciones que crean no funcionarán de manera infalible. Contendrán defectos y errores, aún con su mejor esfuerzo. Aún los mejores ingenieros de software y científicos de computación producen software que contiene errores. Un error puede ser desde una falta de ortografía hasta un malentendido de la definición original del problema.

**Defecto:** es una falla en la solución que causa un comportamiento equivocado.

Los defectos pueden ser detectados y eliminados en cualquier etapa del proceso de desarrollo, pero cuanto antes se encuentren y arreglen, será mejor.

Se recomienda cambiar algo que esté solo en idea que cuando ya se haya implementado. Imagine el costo de corregir el error en un smartphone luego que millones de ellos ya se vendieron.

---

**Error:** es un comportamiento observado cuando se ejecuta una solución que no coincide con el comportamiento esperado.

Algunos defectos aparecerán solo cuando la solución se ha implementado. En estos casos se requieren pruebas (testing) y seguimiento paso a paso (debugging).



## Tipos de defectos

---

De escritura





Durante las etapas de planeación la solución existe en diferentes formatos de diseño: documentos, notas, diagramas, algoritmos, etc. La mejor forma de prevenir defectos en etapa es buscar las equivocaciones elementales. Los errores de escritura incluyen:

- 1 Errores de ortografía.
- 2 Mayúsculas incorrectas.
- 3 Palabras faltantes.
- 4 Palabras equivocadas, por ejemplo: fase en lugar frase.
- 5 Listas mal numeradas, etc.

Algunas equivocaciones son triviales, pero cualquiera de ellas causa problemas.



## Mala gramática y ambigüedades

Al igual que los defectos de escritura, los gramaticales son fáciles de ubicar y corregir. Es fácil identificar la equivocación en una oración, pero algunos de ellos son muy sutiles. Por ejemplo:

Después de tener la moneda recibida, el usuario debe poder empujar la puerta.

¿Qué o quién recibe la moneda? Las oraciones deben ser muy explícitas para ser interpretadas solo de una manera.

También algunas ambigüedades pueden aparecer en oraciones bien escritas gramaticalmente. Por ejemplo:

Después que el proceso de autenticación falle un número de veces seguidas, el incidente debe ser reportado por correo al administrador de seguridad del sistema.

La intención de la oración es buena, pero ¿qué significa un número de veces? ¿Dos, cinco, cien? Debe ser permitido especificar el número exacto de veces, dando el mayor detalle posible.

Otro caso de ambigüedad que puede causar problemas es cuando el mismo concepto es referenciado utilizando términos diferentes. Por ejemplo:

Utilizar de manera indistinta los términos smartphone, Tablet y handheld.

Esto debería cuestionar si se refiere el mismo tipo de dispositivo y pueden en realidad ser "intercambiados".

Es importante mantener un glosario de términos utilizados en la solución y apegarse a ellos.

## Inconsistencias

El eliminar ambigüedades ayuda a encontrar otro tipo de defectos llamados **inconsistencias**. Una inconsistencia emerge cuando diferentes partes de la solución realizan enunciados que no tienen sentido cuando están juntos. Los defectos serán difíciles de localizar porque pueden aparecer en diferentes contextos.

Considere el ejemplo del proceso de autenticación reescrito de la siguiente manera:

Después que el proceso de autenticación falle **cinco** veces seguidas, este incidente debe ser reportado por correo al administrador de seguridad del sistema.

Además, imagine que otra parte de la solución contiene esta oración.

Después que el proceso de autenticación falle **tres** veces seguidas, la pantalla de autenticación debe bloquearse automáticamente y debe evitar más intentos.

Aquí tenemos una inconsistencia. Si un intento de autenticación falla tres veces seguidas, entonces el usuario no podrá realizar más intentos. Por consiguiente, nunca se podrá alcanzar los cinco intentos seguidos de autenticación, así que el primer enunciado resulta redundante.

## Defectos lógicos y matemáticos

Defectos simples pueden ser encontrados en expresiones lógicas. Por ejemplo, el siguiente algoritmo debe imprimir el valor máximo de dos números  $x$ ,  $y$ .

If  $x > y$ , then print  $y$

Else if  $y > x$ , then print  $x$

## Tiene dos defectos:

### Primero

El primero es obvio, el algoritmo imprime el mínimo en lugar del máximo.

### Segundo

El segundo es más sutil, ¿qué pasa si los dos números son iguales? Si esto sucede, no se imprimirá nada.

Esto demuestra que los defectos pueden surgir de expresiones válidas, pero cuando la semántica (significado) es incorrecta.

Asimismo, los errores matemáticos pueden resultar de expresiones válidas. Si la eficiencia de un carro se evalúa con la ecuación:

$$e = \frac{m}{g}$$

Donde  $e$  es la eficiencia,  $m$  las millas viajadas y  $g$  los galones consumidos de combustible. Esto funcionará bien, excepto cuando el contador del viaje se reinicie, en este caso los galones consumidos se convierten en cero y dividir por cero es inválido matemáticamente. La solución deberá tener especial cuidado en consideración.



## Ser exhaustivo



Recordemos que las computadoras no tienen sentido común, no tratarán de llenar los vacíos en las instrucciones y no podrán dialogar con el usuario cuando se ingrese algo incorrecto.

El camino que una computadora toma a través de un algoritmo varía en ocasiones. Depende de los parámetros que se le envíen, así como de otros factores del ambiente. Aún así, los algoritmos deberían ser exhaustivos y contener instrucciones para todas las salidas posibles.

Este tipo de defectos son más comunes con ingenieros junior, pero también algunos senior los cometan. Escribir un algoritmo robusto, que considere todos los caminos (deseados o no deseados) requiere experiencia y aprender principios generales:

### **Pensar críticamente (lo vimos en el módulo 2).**

Una vez que haya escrito una secuencia de instrucciones, debe analizar qué puede salir mal. Examine cada paso y pregúntese si hay aspectos que puedan salir de otra forma.

### **Revise con especial atención a los condicionales.**

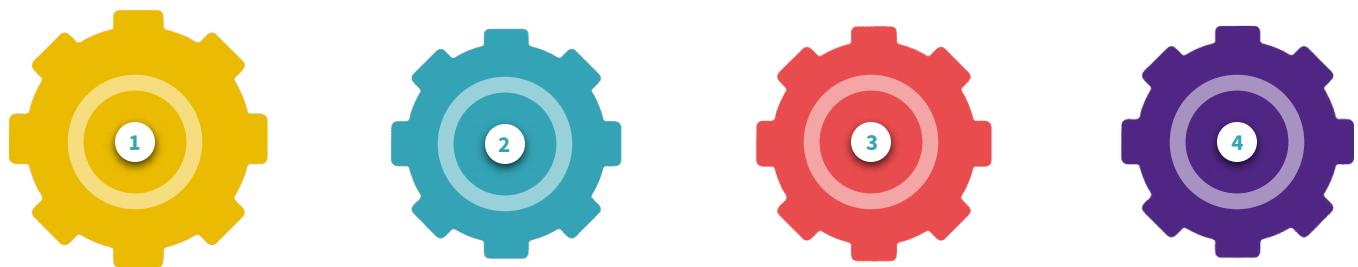
- Si tiene una instrucción de la forma “Si x, entonces haga y”, piense ¿qué pasa si no se cumple? Siempre considere “Si x, entonces haga y, si no, haga z”.
- Los condicionales compuestos pueden dar problemas. Por ejemplo: “Si a y no B o C”. Piense en la cantidad de combinaciones posibles.
- Un consejo antiguo y confiable de Beizer (1990) dice: “los defectos asechan los rincones y se agrupan en los límites”. Tome este ejemplo: “si nota < 50, entonces el rendimiento es ok; si nota > 50, entonces el rendimiento es bueno”. ¿Qué falta? ¿Qué pasa si la nota es exactamente (=) 50? Tome especial cuidado cuando evalúe rangos y límites.

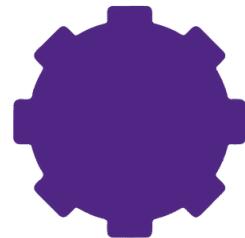
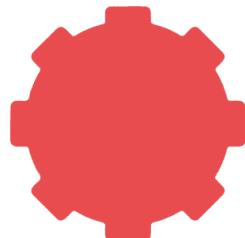
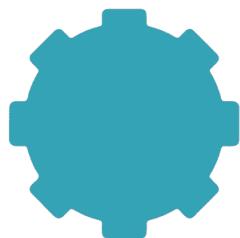
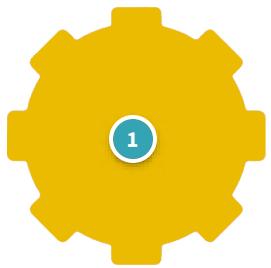
### **Revise con especial atención a los ciclos.**

- Si un ciclo se puede terminar antes, inserte una instrucción que lo permita.
- El resto del ciclo puede ser obviado y la computadora se moverá a la siguiente iteración.

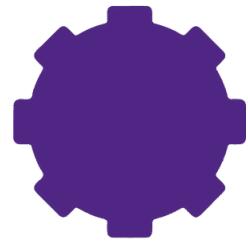
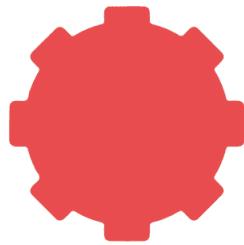
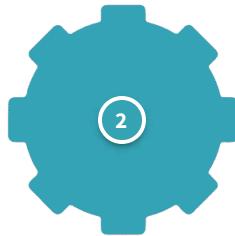
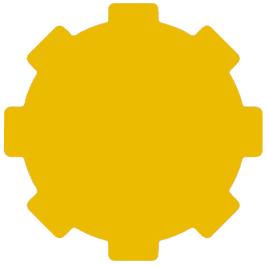
- Asegúrese **por completo** que el ciclo no se repita infinitamente.

## Consejos generales

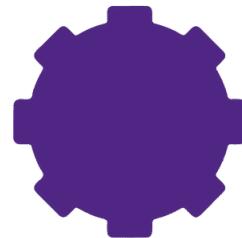
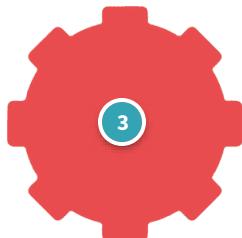
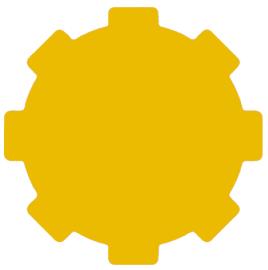




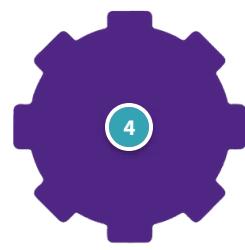
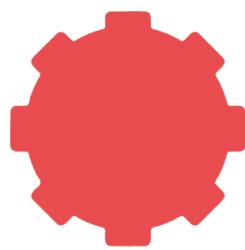
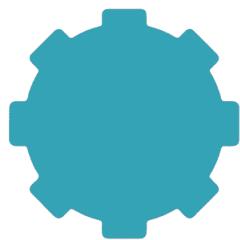
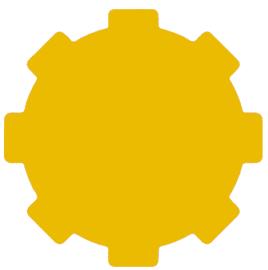
Como el autor de la solución, Ud. no es un observador imparcial. Le será difícil encontrar los problemas que existen, ya que está muy inmerso en los detalles de cómo debe funcionar y que existe cierto sesgo por ver a su solución como infalible. Además, se pueden haber realizado supuestos implícitos sin pruebas. Es una buena idea encontrar a un colega que revise la propuesta de manera imparcial.



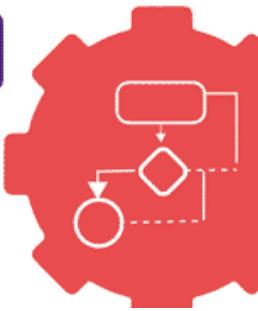
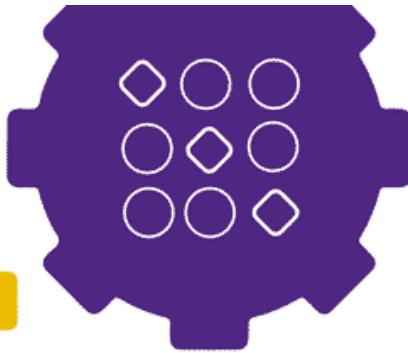
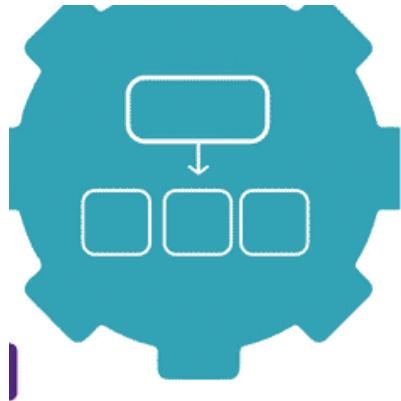
Además, según lo visto en el módulo 3, el enfoque de resolución de problemas es iterativo. Este es un buen momento para retomar ese consejo. Cuando se diseñe una solución, realizarlo en pasos pequeños. Trabaje la solución por etapas y aplique puntos de control.



Haga lo posible que cada parte de la solución sea autónoma y minimice las relaciones con otras partes. Esto debería permitir que, al localizar los defectos o errores, se prevenga el efecto en cascada con otras partes de la solución.



Finalmente, si la definición del problema proviene de un tercero, se debe involucrar a esa persona en la verificación del diseño de la solución. Una de las principales fuentes de defectos se debe a los malentendidos entre el diseñador de la solución y quien definió el problema.



## Mitigación de errores

---



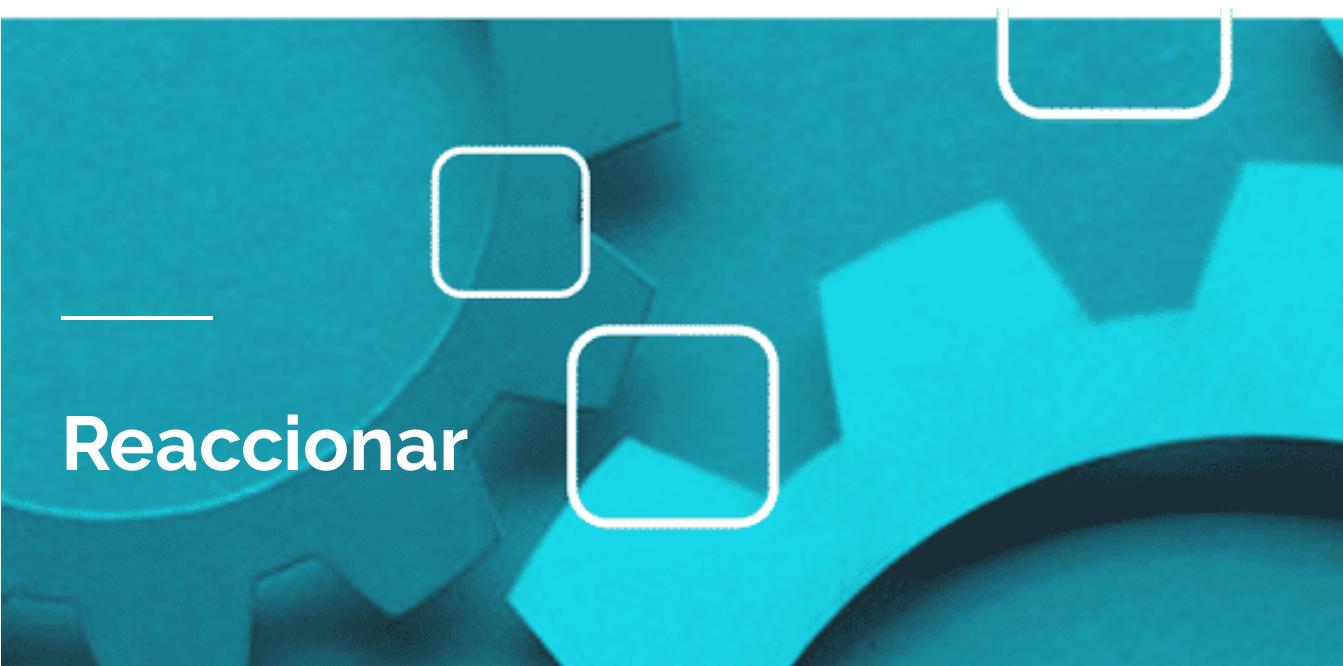
Hasta ahora, se han analizado las acciones preventivas para parar los defectos que pueda tener una solución. Sin embargo, hay altas probabilidades que la solución final contenga errores, a pesar de los esfuerzos. En esta sección algunas estrategias para mitigarlos.





## Defender

Una forma de minimizar los efectos de los errores es colocar barreras defensivas. En ciencias de la computación se le llama programación defensiva. Por ejemplo: para una operación que divide dos números, si el divisor es cero ocasionará un error, así que se puede crear una precondition para realizar la validación que el divisor no sea cero.



## Reaccionar



Qué hacer en caso de que falle una precondición, depende del contexto. Si la severidad de la consecuencia es baja, la acción puede ser el despliegue de un mensaje. Si la severidad es alta y se causa un daño mayor, el sistema podría entrar en modo de emergencia y apagarse.

Por ejemplo, una máquina que despache bebidas lee el valor de dinero insertado y despacha al cliente la bebida de su elección. Pero muchas cosas podrían salir mal:

- 1 El cliente podría ingresar dinero de otro país. La máquina puede recuperarse, devolviendo el dinero, idealmente acompañado de un mensaje.
- 2 Si el vaso o taza se ha caído de su posición, la máquina debiera detectarlo y no despachar la bebida.
- 3 Si se diera una filtración del sistema que provee el agua a la máquina pondría en riesgo el sistema eléctrico. Si se detecta la filtración, la máquina debiera apagarse.

---

**Otras consideraciones para reaccionar a errores incluyen:**

1

2

Si el sistema es interactivo, se puede identificar qué solicitarle al usuario.

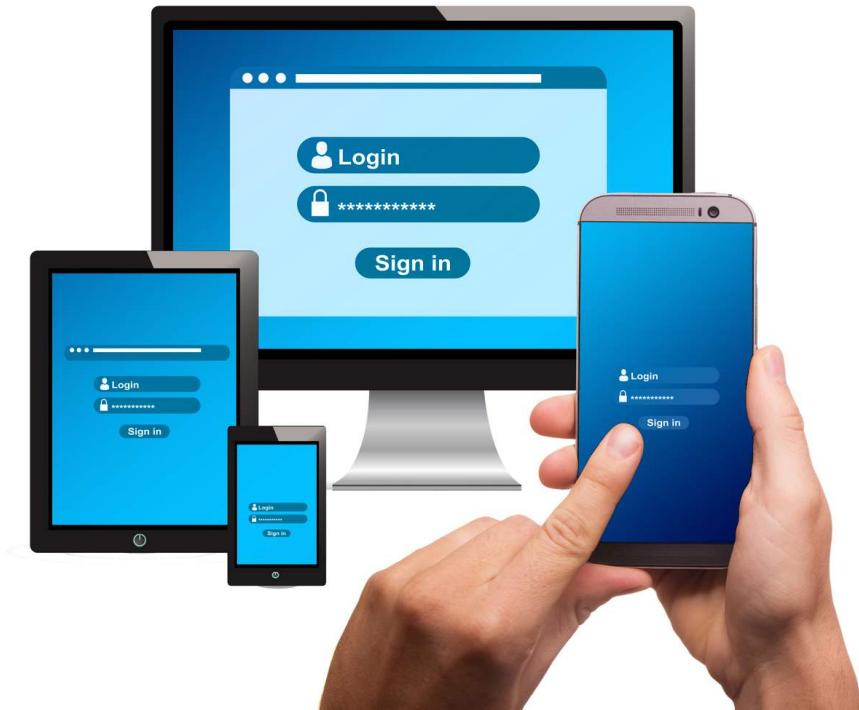
1

2

Si el sistema es transaccional, debe permitir revertir los cambios durante un paso defectuoso. Una transacción debe validar que se ejecute “todo” o “nada”. Por ejemplo una transferencia bancaria entre cuentas: si el débito se realiza exitosamente, pero el crédito a otra cuenta falla, debe revertirse toda la operación, incluyendo el débito realizado.



## Comprobar las entradas del usuario



En caso de que la solución solicite la entrada de un usuario, debe existir una “paranoia saludable”. Es decir, saber que los humanos cometén errores todo el tiempo (algunos lo hacen de manera intencionada), así que es mejor asumir que el usuario ingresará entradas defectuosas.

Algunos ejemplos de cosas que pueden salir mal con el ingreso de entradas de usuarios son:

- 1 Si la solución espera un número, se debe validar que en realidad sea un número.
- 2 Si la solución requiere una entrada dentro de un rango, se debe chequear que el valor ingresado lo cumpla.
- 3 Formatos incorrectos de números de teléfono, sitios web, direcciones de correo electrónico, fechas y horas. En todos los casos, se debe validar que la entrada cumpla el formato requerido.

Se debe ser proactivo en cuanto al comportamiento no esperado, aún así algunos defectos podrán filtrarse.



# Pruebas de calidad (testing)

---

## Enfoques



La meta de las pruebas de calidad es usar el sistema que se ha desarrollado para verificar si se comporta como se espera e identificar áreas específicas de fallo. Estas pruebas deben esperar hasta que la solución alcance un estado para ser operable o funcional. Pueden realizarse pruebas de calidad de partes funcionales o de toda la solución.

Para realizar las pruebas de manera efectiva, se debe tener claridad de la meta de la solución, como se indicó en el módulo 3. Si no se sabe qué esperar, no se pueden realizar buenas pruebas. La meta de la solución sirve de brújula a las pruebas.

# Las pruebas de calidad deben ser sistemáticas, para identificar dónde está el problema y por qué.

Se puede elegir dos enfoques de pruebas de calidad, según lo que se quiera lograr.

TOP-DOWN

BOTTOM-UP

Se prueba la solución como un todo para asegurarse que funciona. Es la estrategia más efectiva para detectar fallas de diseño y validar que el sistema se comporte bien de manera integrada.

TOP-DOWN

BOTTOM-UP

Requiere probar pequeñas partes de la solución de manera individual. Permite verificar que cada parte cumple su función y objetivo y permite comprobar que el sistema se ha desarrollado con bases sólidas.

Estrategia de pruebas	Ventanas	Desventajas
Top-down	<ul style="list-style-type: none"><li>Efectiva para detectar fallas de diseño.</li><li>Puede ser alentador probar un sistema completo, aunque esté incompleto o tenga fallas.</li></ul>	<ul style="list-style-type: none"><li>Se dificulta aplicar si el sistema está incompleto.</li><li>Requiere utilizar sustitutos o simuladores para las partes del sistema que están incompletas.</li><li>Es más difícil encontrar un defecto cuando sucede un error.</li></ul>
Bottom-up	<ul style="list-style-type: none"><li>Es más fácil de realizar en las primeras etapas de desarrollo.</li><li>Es efectiva para localizar problemas.</li></ul>	<ul style="list-style-type: none"><li>Requiere simular las partes de control de la solución.</li><li>No expone las fallas de diseño.</li></ul>

Se deben combinar estos enfoques al realizar pruebas, para tener lo mejor de los dos mundos.

## Pruebas de partes individuales



El realizar pruebas a las partes individuales permite localizar los errores de manera más sencilla. Las pruebas que se realizan a las partes individuales se llaman **pruebas unitarias**.

**Considere el juego FizzBuzz, conozca más en este video.**

 YOUTUBE



## What is Fizz Buzz?

Fizz Buzz is becoming a popular interview question for Computer Science jobs. Let's take a look, from the non-programmer's perspective, of what the fuss is a...

**VER EN YOUTUBE >**

---

¿Qué pasaría si en lugar de tener un ciclo que clasifique cada número del 1 al 100 se le pidiera al usuario que ingrese un número? En este caso, deberían considerarse las siguientes posibilidades de entradas:

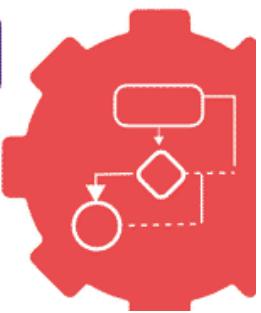
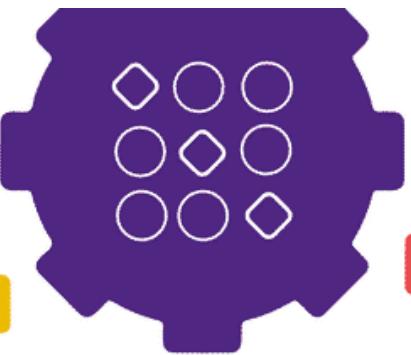
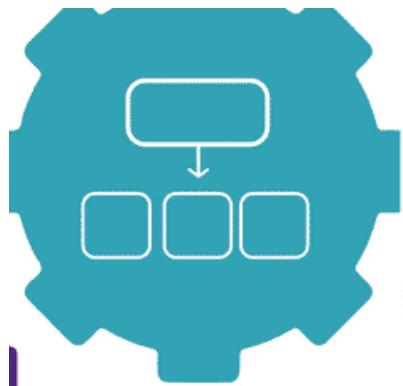
- 1 Números aceptables (por ejemplo, el 8).
- 2 Fizzes (por ejemplo, el 3): representa los números divisibles por 3.
- 3 Buzzes (por ejemplo, el 10): representa los números divisibles por 5.
- 4FizzBuzzes (por ejemplo, el 30): represente los números divisibles por 3 y por 5.
- 5 Números no aceptados (por ejemplo el -10): números fuera del rango de [1,100] deben ser rechazados.
- 6 Caracteres no numéricos (por ejemplo, el VIII): solo se aceptan números arábigos. Si se ingresa un VIII en romanos, debe rechazarse.

### Recordando que los defectos se esconden en los límites:

- Límite aceptable (por ejemplo, el 1): debería funcionar.
- Límite no aceptable (por ejemplo, el 0): debería ser rechazado.



Las pruebas son una oportunidad para que nuestra solución tenga la mirada de otra persona. Todas las pruebas deberían ser registradas en un documento (plan de pruebas). Deben ser fáciles de comprender y muy claras. Luego, entregar el plan a otra persona y pedirle que pruebe la solución. La perspectiva de un tercero permitirá que se encuentren problemas que la persona que diseñó la solución pasaría por alto.



## Seguimiento paso a paso (debugging)

---

# Seguimiento paso a paso (debugging)

Inevitablemente, en algún punto de la solución, cuando ésta ya esté en marcha, ocurrirá un error. El evento será poco claro sobre qué parte de la solución causó el error, pero no hay que angustiarse. Nuevamente, se puede ser sistemáticos al encontrar y corregir el error.

Para esto se puede utilizar el debugging o seguimiento paso a paso de la solución mientras se ejecuta.

Para realizar el debugging, además del conocimiento de herramientas, se requiere creatividad e imaginación. Algunas estrategias pueden ser:

o 1

## **Darles la oportunidad a las coronadas y rastrearlas**

Si resulta incorrecta, se descarta.

## Paso 2

### Aplique el principio de Occam

Entre dos hipótesis, elija la más simple. Resístase a las explicaciones complicadas. Para ampliar este principio se puede visualizar este video:

<https://www.youtube.com/watch?v=9GI0EJyBxlg>

## Divide y vencerás

Si no se sabe dónde está el problema, será más efectivo reducir el número de lugares dónde buscar. Las partes en las que se descompuso el problema pueden ser beneficios al “debuggear”.

## Cambiar una cosa a la vez

El buen debugging es como la buena experimentación. Cuando un científico lleva a cabo un experimento, solo varían un factor a la vez. Si se piensa que un cambio resolverá el problema, debe aplicarse y probar. Si no funciona, deshacer el cambio y llevar al sistema a su estado original antes de probar otro escenario.

## Dejar bitácoras

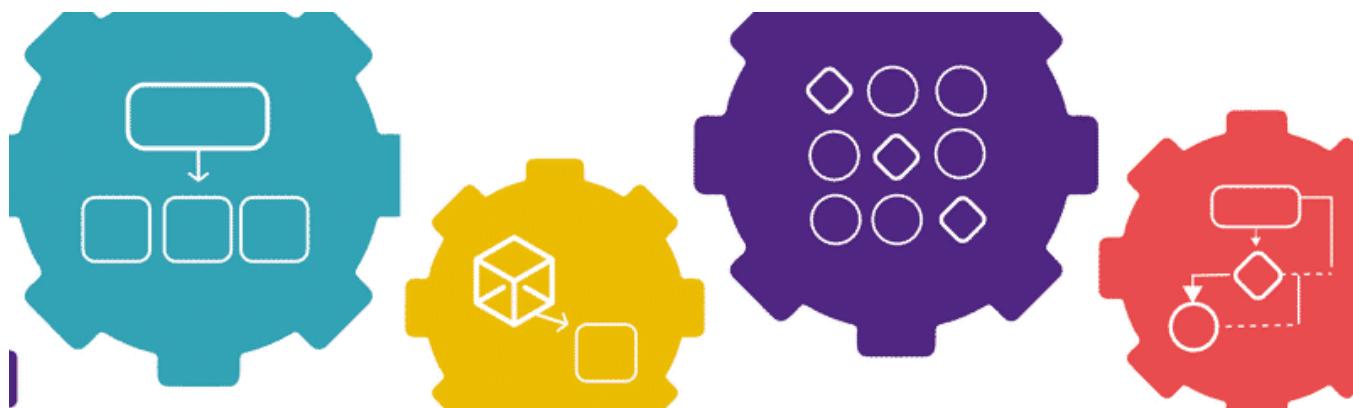
Si se requiere rastrear las operaciones internas que realiza la computadora, se pueden agregar instrucciones de registro de bitácoras en el algoritmo, para que se deje un rastro de las operaciones y sus valores.

## Seguimiento

Se puede realizar paso a paso en tiempo real para conocer cómo se ejecuta ciertas partes del algoritmo implementado en código.

## Explicar el problema en voz alta

El explicitar el problema en voz alta permite colocarlo en palabras y mencionarlo a alguien más. Las preguntas que realicen otras personas sobre la explicación recibida, permitirán identificar cómo resolver el problema.



## No se puede atender todo: decidir qué errores se corregirán

---



Todo software siempre tendrá fallas, es importante entregar la mejor solución posible, esto implica identificar a qué errores se les dará prioridad para ser corregidos. Para ellos es útil clasificar los errores en ciertas categorías:

## **Severidad**

Describe si el error detiene el uso del sistema. Puede ser menor, moderada, mayor o crítica.

## **Prioridad**

Considera otros factores para definirse.

- Frecuencia: si el error ocurre muy seguido su prioridad es alta.
- Funcionalidad: si el error afecta una funcionalidad no importante, puede tener baja prioridad. Pero, si comprometa una funcionalidad esencial, tendrá una alta prioridad.
- Calendarización: estimar cuánto tomará corregir el error y si vale la pena realizarlo. Si es poco probable que ocurra y se estima que tarde una semana entera o bien si el esfuerzo es mínimo y es más probable que ocurra.

Prioridad	Alta	Baja	Alta
	Baja	Severidad	
Alta	<p>La falla se da en una funcionalidad importante, pero no detiene el uso del sistema.</p> <p><b>Ejemplo:</b> el logo de la página principal se ve con baja definición.</p>	<p>Causa una falla en la funcionalidad básica y detiene el buen uso del sistema.</p> <p><b>Ejemplo:</b> falla el método de autenticación.</p>	<p>Un error esporádico que causa falla de menor importancia.</p> <p><b>Ejemplo:</b> al cargar la imagen de un avatar ocasionalmente se tarda mucho y expira el tiempo de subida.</p>



## A manera de síntesis

---



## A manera de síntesis

Los defectos son equivocaciones que causan errores en una solución. Existen diferentes técnicas para prevenir los defectos o tratar con los errores una vez la solución ha sido implementada.

Prevenir que los defectos lleguen a la solución implica un esfuerzo de dedicación en el diseño. Cuanto más pronto sean detectados los defectos, se vuelven menos costosos de reparar.

```
isVideo = ( type == "image" ) || ( type == "video" );
isUrl  = ( source.indexOf("youtube.com/watch") > -1 );
isElement = ( type == "url" ) || ( type == "image" ) || ( type == "video" );
isObject = ( type == "element" ) || ( type == "image" ) || ( type == "video" );

// Check if boxer is already active, if yes, return
if ($("#boxer").length > 1) {
    return;
}

// Kill event
_killEvent(e);

// Cache internal data
data = $.extend({}, {
    $window: $(window),
    $body: $("body"),
    $target: $target,
    $object: $object,
    visible: false,
    resizeTimer: null,
    touchTimer: null,
    gallery: {
        active: false
    }
});
```

Mitigar los errores significa anticiparse y controlar sus consecuencias potenciales. La programación defensiva permite prevenir que los defectos ocasiones estragos.

La cacería de defectos y su remoción se realiza por medio de pruebas de calidad y debugging. En las pruebas de calidad se buscan errores con estrategias top-down y bottom-up. El uso de bitácoras y seguimiento paso a paso permite identificar las causas de los errores.



¶

—

—

## Recursos complementarios

---

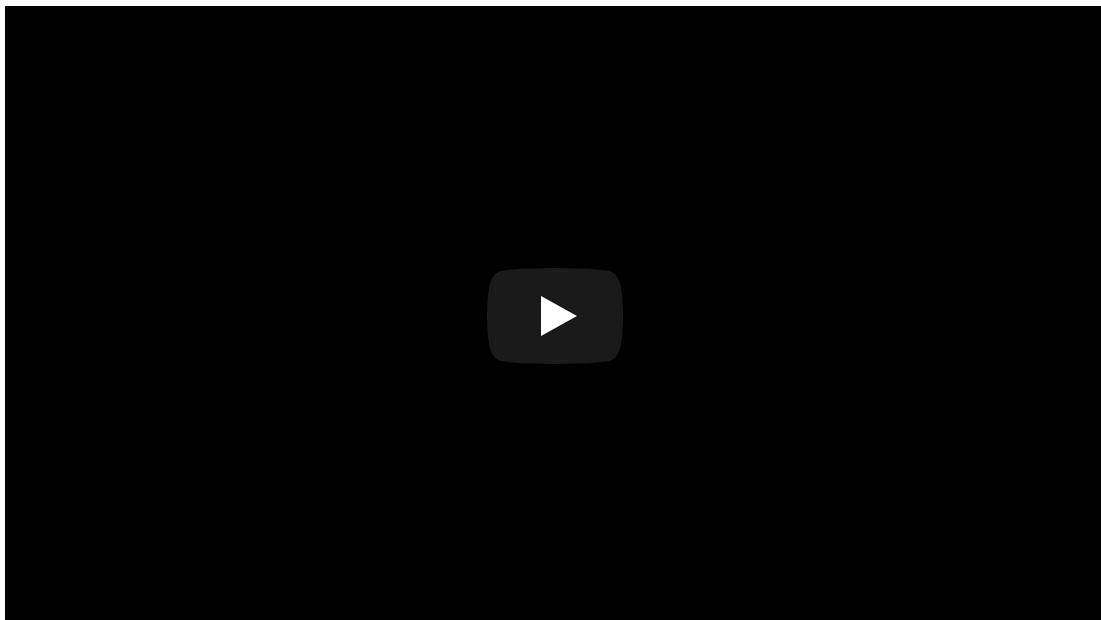


Recursos complementarios

---

## NASA's first software engineer: Margaret Hamilton - Matt Porter & Margaret Hamilton

 YOUTUBE



## NASA's first software engineer: Margaret Hamilton - Matt Porter & Margaret Hamilton

Learn how Margaret Hamilton and her team of engineers built the software for the Apollo 11 mission that landed mankind on the Moon. --The Apollo 11 moon land...

[VER EN YOUTUBE >](#)

¿Qué característica de la programación ayudó a cumplir la misión de la llegada a la Luna?

Escriba su respuesta aquí

SUBMIT



## Actividad 1

---



Actividad 1



## Instrucciones

Marque si los siguientes enunciados son verdaderos o falsos.

Los errores pueden surgir de enunciados que son válidos gramaticalmente.

Verdadero



Falso

SUBMIT

La condición de salida de un ciclo se evalúa después de cada enunciado dentro del ciclo.

---



Verdadero



Falso

SUBMIT

La programación defensiva asume que el código es seguro hasta que se pruebe lo contrario.

---



Verdadero



Falso

SUBMIT

Las pruebas bottom-up son una forma efectiva de encontrar errores de diseño.

---



Verdadero



Falso

SUBMIT

## Actividad 2

---



Actividad 2



## Instrucciones

¿Cuál es la diferencia entre defecto y error? Explique su respuesta.

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

## Actividad 3

---



Actividad 3



Una instrucción “if-then”, le dice a la computadora que realice una acción si una condición se cumple (es verdadera). ¿Cómo cambiaría esta instrucción para que se ejecute si la condición **NO** se cumple (es falsa)?

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

## Actividad 4

---



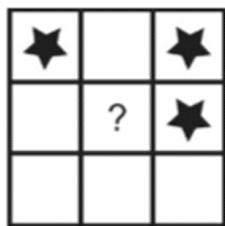
Actividad 4



## Instrucciones

En este algoritmo hay un error. Encuéntrelo y corríjalo.

## Partway through tracing a Minesweeper algorithm



	1	2	3	4	5	6	7	8
square	*		*	*				

```
let mines = 0
let n = 1
start loop
    if squaren has a mine
        then mines = mines + 1
    n = n + 1
loop again if n < 8
```

mines	n
0	1
1	2
2	3
3	4
4	5
5	6
6	7

Tal y como están configurados los cuadros en el ejemplo no necesariamente exponen el error. Pruebe “debuggeando” con las minas en diferentes posiciones.

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

## Actividad 5

---



Actividad 5



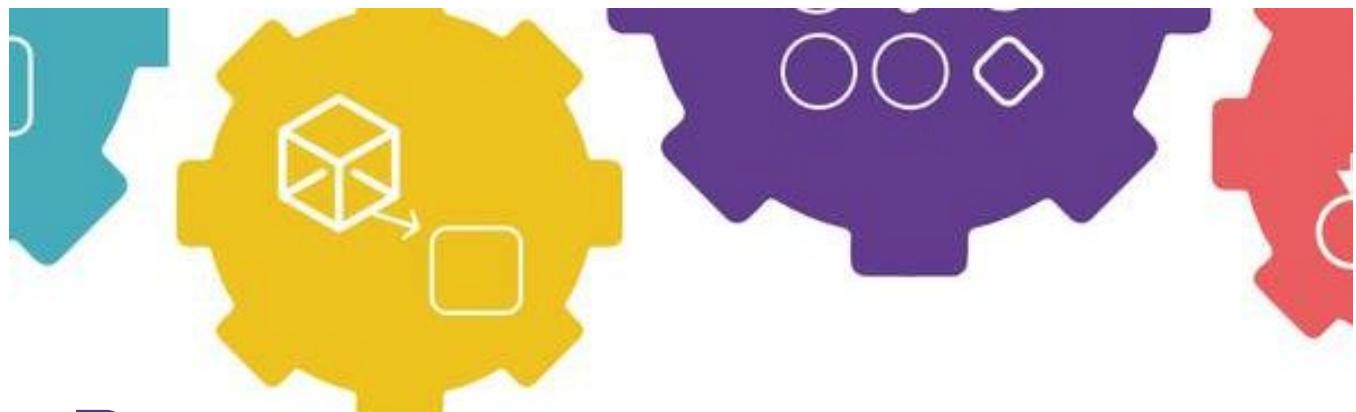
## Instrucciones

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.

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

## Recursos

---



## Recursos

### Instrucciones:

Consultar con su profesor sobre los retos a realizar en el laboratorio y los materiales que se deben llevar.



## Rúbrica de evaluación

---



Rúbrica



Descargue la siguiente rúbrica de evaluación.



**Rúbrica de Evaluación.pdf**

142.3 KB



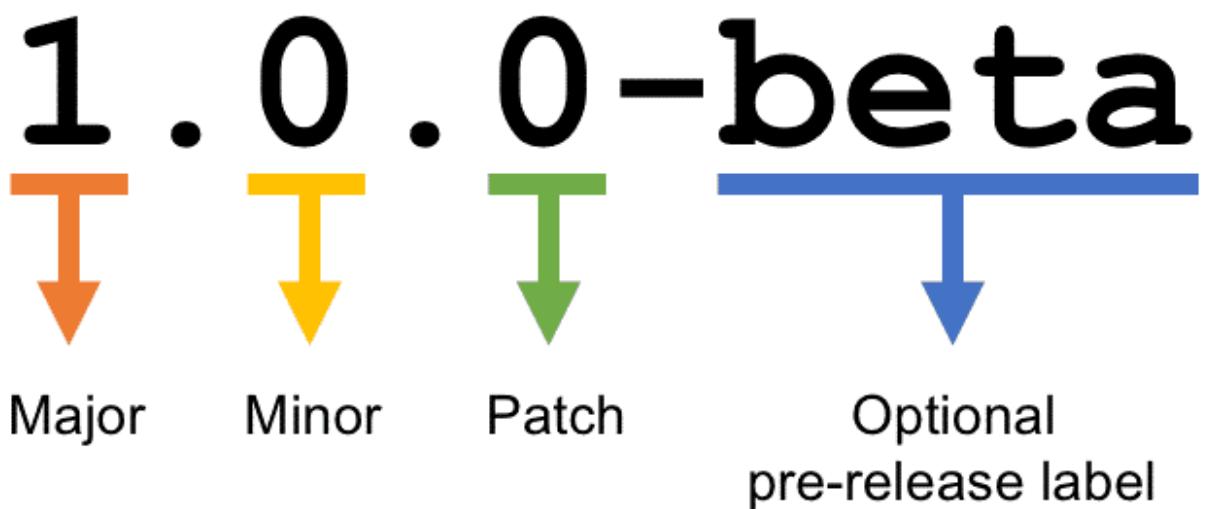
## Diario de experiencias de laboratorio

---



# 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

Bordigon, 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.

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

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

## Créditos

---

