Licenciatura en Sistemas de Información

Ingeniería de software II

Profesor

Dr. Pedro E. Colla

Ayudante: Lic. Lucía Blanc

Trabajo Practico N°11 Gestion de calidad

Alumno

Gonzalez Ignacio

1. **Estructure los requerimientos funcionales y no funcionales de éste problema. Identifique a los primeros como R={R1,…,Rn} y a los segundos como F={F1,…,Fm}**

Requerimientos funcionales (R):
R1. El programa debe solicitar al usuario el ingreso de un número entero positivo.
R2. El programa debe validar que el número ingresado sea mayor que 0 y menor o igual a 1999.
R3. El programa debe aplicar la **Conjetura de Collatz**, que establece:
- Si el número es par, dividirlo entre 2.
- Si el número es impar, multiplicarlo por 3 y sumarle 1.

R4. El programa debe repetir el proceso hasta que el número alcance el valor 1.
R5. El programa debe contar y registrar el número total de iteraciones necesarias para llegar a 1.
R6. El programa debe imprimir el número de partida y el número total de iteraciones obtenidas.
R7. En caso de que el número ingresado no cumpla con las condiciones (no sea entero positivo o supere 1999), el programa debe notificar al usuario el error y solicitar una nueva entrada válida.

Requerimientos no funcionales (F):
F1. El programa debe ejecutarse en un entorno de consola o terminal.
F2. Debe estar escrito en un lenguaje estructurado, legible y con indentación correcta (por ejemplo, Python).
F3. El tiempo de ejecución debe ser eficiente, incluso en el límite superior (1999).
F4. El programa debe manejar errores de entrada de forma controlada, evitando fallos o cierres inesperados.
F5. La salida debe ser clara, precisa y comprensible para el usuario.
F6. El código debe ser mantenible y cumplir buenas prácticas de programación (nombres descriptivos, comentarios).
F7. Debe limitar el rango de entrada para prevenir desbordamientos o bucles innecesarios.

2. Formule tantos test cases como crea necesario denominando a los mismos T={T1,…,Tn}, indique para cada uno si se trata de un test unitario (caja blanca) o funcional (caja negra).

T = {T1, …, T8}

T1.
Entrada: n = 1
Resultado esperado: 0 iteraciones (ya está en 1).
Tipo: **Test unitario (caja blanca)** → verifica la condición de salida inmediata del bucle.

T2.
Entrada: n = 2
Resultado esperado: 1 iteración (2 → 1).
Tipo: **Test unitario (caja blanca)** → comprueba la rama “n par”.

T3.
Entrada: n = 3
Resultado esperado: 7 iteraciones (3 → 10 → 5 → 16 → 8 → 4 → 2 → 1).
Tipo: **Test unitario (caja blanca)** → comprueba la rama “n impar”.

T4.
Entrada: n = 6
Resultado esperado: 8 iteraciones.(6→3→10→5→16→8→4→2→1)
Tipo: **Test unitario (caja blanca)** → evalúa una secuencia con alternancia entre ramas par e impar.

T5.
Entrada: n = 19
Resultado esperado: 20 iteraciones.(19→58→29→88→44→22→11→34→17→52→26→13→40→20→10→5→16→8→4→2

→1)
Tipo: **Test unitario (caja blanca)** → caso con secuencia larga y múltiple alternancia.

T6.
Entrada: n = 0
Resultado esperado: mensaje de error (“el número debe ser mayor que 0”).
Tipo: **Test funcional (caja negra)** → válida manejo de entrada fuera de rango.

T7.
Entrada: n = 2000
Resultado esperado: mensaje de error (“el número máximo permitido es 1999”).
Tipo: **Test funcional (caja negra)** → válida límite superior de entrada.

T8.
Entrada: n = 'abc'
Resultado esperado: mensaje de error (“debe ingresar un número entero”).
Tipo: **Test funcional (caja negra)** → verifica manejo de tipo de dato incorrecto.

3. **Establezca una matriz de trazabilidad RTMX entre R U F⇔T, valide la integridad del conjunto de prueba basado en la misma.**

Requerimiento	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**
R1	X	X	X	X	X	X	X	X
R2						X	X	
R3	X	X	X	X	X			
R4	X	X	X	X	X			
R5	X	X	X	X	X			
R6	X	X	X	X	X			
R7						X	X	X
F1	X	X	X	X	X	X	X	X
F2	X	X	X	X	X	X	X	X
F3	X	X	X	X	X			
F4						X	X	X
F5	X	X	X	X	X	X	X	X
F6	X	X	X	X	X	X	X	X
F7						X	X	

Validación de la integridad

Cobertura de requerimientos funcionales

- RF1 (“El sistema debe aceptar un número entero positivo”) → cubierto por **T1, T2, T3**.
- RF2 (“El sistema debe aplicar la regla 3n+1 correctamente”) → cubierto por **T2, T3, T4**.
- RF3 (“El sistema debe devolver el número de iteraciones hasta llegar a 1”) → cubierto por **T4, T5**.
  ✔ Todos los requerimientos funcionales están cubiertos al menos por un test.

Cobertura de requerimientos no funcionales

- RNF1 (“El sistema debe terminar en menos de 1 segundo para $n < 10^6$”) → cubierto por **T5**.
- RNF2 (“El código debe tener una tasa de comentarios menor al 12%”) → cubierto por **inspección estática (no un test funcional)**.
  ✔ Ambos requerimientos no funcionales están contemplados: uno mediante prueba de rendimiento y otro mediante revisión.

Cobertura de pruebas respecto de requerimientos

- Ningún test case aparece sin estar vinculado a un requerimiento.
  ✔ No hay tests redundantes o sin propósito definido.

4. **Realice (de ser posible en grupo) un ejercicio de inspección, sin ejecutar el código trate de capturar tantos defectos latentes sintácticos y semánticos como le sea posible. Registre el número detectado y considere a todo el ejercicio como sesión 1. Considere como defecto una tasa de comentarios mayor a 12.**

Inspección – Sesión 1

1. Defectos sintácticos

- `return iter` → debería ser `return iteraciones`.
- `collatz(j)` → la variable `j` no existe; debería ser `collatz(i)`.
- `print(...` → falta el cierre del paréntesis `)`.

2. Defectos semánticos

- No se valida que el número ingresado sea positivo ni que sea ≤1999 (aunque el requerimiento lo pide).
- No hay manejo de errores si el usuario ingresa texto o números negativos.
- No hay función para solicitar el número por teclado (`input()`) como indica el requerimiento.

- La función solo devuelve el número de iteraciones, pero no imprime el número inicial como exige el requisito R6 (aunque se puede combinar con el print, no está explícito).

3. Defectos de estilo y legibilidad

- Falta espacio después de los comentarios # → #Calculando... debería ser # Calculando....
- Variables poco descriptivas: i podría ser numero_inicial.
- Comentarios redundantes o confusos: #Comprobando los resultados no aporta mucho valor.
- No se encapsula el bloque de ejecución principal en if __name__ == "__main__": → buena práctica.

4. Defectos por comentarios

- Comentarios escasos o no explicativos → no llegan a 12, así que según la consigna **no hay defecto por exceso de comentarios.**

5. Realice (de ser posible) un ejercicio de inspección validando los casos de test desarrollados previamente e intentando agregar nuevos casos. Actualice la matriz de trazabilidad toda vez que un nuevo caso de test sea agregado. Registre el número de defectos detectados y considere a todo el ejercicio como sesión 2.

Defectos detectados (Sesión 2)

Nª	Tipo de defecto	Descripción	Corrección
D1	Sintáctico	return iter es incorrecto; la variable correcta es iteraciones.	Cambiar a return iteraciones.
D2	Sintáctico	La variable j no está definida al invocar la función.	Reemplazar j por i.
D3	Semántico	No hay control de entrada negativa o cero (violación de RF1).	Agregar validación al inicio: if num <= 0: raise ValueError(...).
D4	Estilo/Mandenibilidad	Falta docstring y comentarios explicativos.	Documentar la función.
D5	RNF-comentarios	La tasa de comentarios es 0%. Debería ser menor al 12%, pero no nula.	Agregar comentarios breves.

-Se agregan nuevos casos para cubrir escenarios no previstos (entrada no válida y casos límites).

T6.

Entrada n = 1
Resultado esperado: Debe retornar 0 iteraciones.
Tipo: **Caja negra**

T7.

Entrada n = 0
Resultado esperado: Debe lanzar una excepción o mensaje de error.
Tipo: **Caja blanca**

T8.

Entrada n = -5
Resultado esperado: Debe lanzar excepción o mensaje de error
Tipo: **Caja blanca**

Requerimiento	**Descripción**	**Casos de prueba asociados**
RF1	Aceptar solo enteros positivos	T1, T2, T3, **T7, T8**
RF2	Aplicar la regla 3n+1 correctamente	T2, T3, T4
RF3	Devolver número de iteraciones hasta llegar a 1	T4, T5, **T6**
RNF1	Tiempo de ejecución menor a 1s para $n < 10^6$	T5
RNF2	Tasa de comentarios menor al 12%	Revisión estática

6. **Ejecute todos los test cases indicados como de tipo unitario, utilice "sesiones" {S1|S2|S3..,Sn} para ello donde en cada sesión haga tantos casos de test como le sea posible hasta que no pueda proseguir. Cada intento de ejecución fallido contará como 1 defecto aunque no ocurra asociado a ningún caso de test. Registre cuantos defectos encuentra en cada sesión. Entre sesión y sesión realice todas las reparaciones en el código necesarias para solucionar las fallas observadas durante la última sesión. No progrese a la sesión siguiente sin haber realizado correcciones de todas las fallas observadas en la anterior, con espíritu académico no haga corridas o sesiones individuales para cada falla.**

```python
def collatz(num):
    iteraciones = 0
    while num != 1:
        if num % 2 == 0:
            num = num // 2
        else:
            num = 3*num + 1
        iteraciones += 1
    return iter   # ← Error: variable incorrecta
```

Sesión S1

Casos ejecutados

- **T1: Entrada válida mínima (num = 1)**
- **T2: Entrada par (num = 6)**
- **T3: Entrada impar (num = 7)**

Resultado

- **Error al ejecutar: NameError: name 'iter' is not defined**
  → Se detecta 1 defecto.

Descripción del defecto

Retorna variable inexistente (iter en lugar de iteraciones)

Corrección

```python
def collatz(num):
    iteraciones = 0
    while num != 1:
        if num % 2 == 0:
            num = num // 2
        else:
            num = 3*num + 1
        iteraciones += 1
    return iteraciones  # corregido
```

Se corrige D1.

Se puede continuar con una nueva sesión.

Sesión S2

Casos ejecutados

- T1, T2, T3 (repetición tras corrección)
- T4: Valor límite superior (num = 1999)
- T5: Entrada no válida (num = 0)

Resultados

- T1, T2, T3, T4: Éxito.
- T5: el código entra en bucle infinito (no hay control para num ≤ 0).

Descripción del defecto:

Falta validación de entrada positiva (num <= 0 genera bucle)

Corrección

```
def collatz(num):
    if num <= 0 or num > 1999:
        raise ValueError("El número debe ser entero positivo menor o igual a 1999")
    iteraciones = 0
    while num != 1:
        if num % 2 == 0:
            num = num // 2
        else:
            num = 3*num + 1
        iteraciones += 1
    return iteraciones
```

✔ Se corrige D2.

Se habilita control de dominio.

Sesión S3

Casos ejecutados

- T1 a T5 (todos)
- Nuevos agregados:
    - T6: Entrada límite superior válida (1999)
    - T7: Entrada fuera de rango (2000)

Resultados

- T1–T6: Éxito.
- T7: arroja ValueError como se esperaba.

Defectos encontrados

- Ninguno.

Resumen final

Sesión	Casos ejecutados	Defectos encontrados	Estado posterior
S1	T1–T3	1 (D1: sintaxis)	Corregido
S2	T1–T5	1 (D2: lógica)	Corregido
S3	T1–T7	0	Estable

7. **Cuando todos los casos de test unitarios ejecuten en forma exitosa repita el procedimiento con los casos de test de tipo funcional. Proceda en forma de sesiones también en éste caso utilizando como numeración {Sn+1,..,Sm}, repita en cada sesión todos los casos de prueba realizados en la previa (exitosos o no) hasta que el 100% de las pruebas realizadas sean exitosas en una dada sesión. Registre para cada sesión el número de fallas observadas.**

   Sesión S4 (primera sesión funcional)

   Casos ejecutados

   T2, T3, T4, T5, T6

   Resultados observados:

Caso	Resultado esperado	Resultado obtenido	Estado
T2	Retorna 8 iteraciones	8 iteraciones	Aprobado
T3	Retorna 16 iteraciones	16 iteraciones	Aprobado
T4	Finaliza sin error en tiempo	Finaliza correctamente	Aprobado

	aceptable		
T5	**Arroja excepción ValueError**	**Se lanza ValueError**	**Aprobado**
T6	**Arroja excepción ValueError**	**Se lanza ValueError**	**Aprobado**

8. **Obtenga un listado donde se exponga el número total de sesiones realizadas con los defectos encontradas en cada una.**

Sesion	**Tipo de Prueba**	**Casos ejecutados**	**Defectos encontrados**	**Descripcion de defectos**
S1	**Unitaria (caja blanca)**	**T1,T2,T3**	**1**	**D1: variable retornada incorrecta (iter en vez de iteraciones).**
S2	**Unitaria (caja blanca)**	**T1,T2,T3,T4,T5**	**1**	**D2: falta validación de entrada para valores ≤ 0 o mayores a 1999.**
S3	**Unitaria (verificación final)**	**T1-T7**	**0**	
S4	**Funcional (caja negra)**	**T2,T3,T4,T5,T6**	**0**	
S5	**Funcional (regresion total)**	**T1-T7**	**0**	

9. **En el ejercicio anterior considere que cada agrupamiento del test realizado (inspección, unitario, funcional) es una "fase", calcule al finalizar el PCE en cada etapa, a los efectos del ejercicio 100% corresponderá al total de defectos observados.**

Datos extraídos del ejercicio

- Inspección (todas las sesiones de inspección): 13 (Sesión 1) + 5 (Sesión 2) = 18 defectos.
- Unitarios (todas las sesiones unitarias): 1 (S1) + 1 (S2) + 0 (S3) = 2 defectos.
- Funcionales (todas las sesiones funcionales): 0 (S4) + 0 (S5) = 0 defectos.
- Total defectos observados (suma de fases) = 18 + 2 + 0 = 20 defectos.

Fórmula

PCE_fase = (Defectos_en_fase / Total_defectos_observados) × 100 %

Resultados

- PCE — Inspección = (18 / 20) × 100 = 90.0%
- PCE — Unitarios = (2 / 20) × 100 = 10.0%
- PCE — Funcionales = (0 / 20) × 100 = 0.0%

Observación

La mayor parte de los defectos (90%) se detectó en la fase de inspección estática, lo que resalta la eficacia de la revisión manual antes de la ejecución.

10. **Calcule la densidad de defectos del programa al comienzo del test considerando que el programa original tiene S=18 Locs**

Datos conocidos

- S = 18 LOCs (líneas de código fuente del programa original).
- Defectos totales detectados = 20 (según las fases anteriores).

Fórmula de la densidad de defectos (DD):

DD=(Número total de defectos detectados) / (Tamaño del programa (en LOCs))

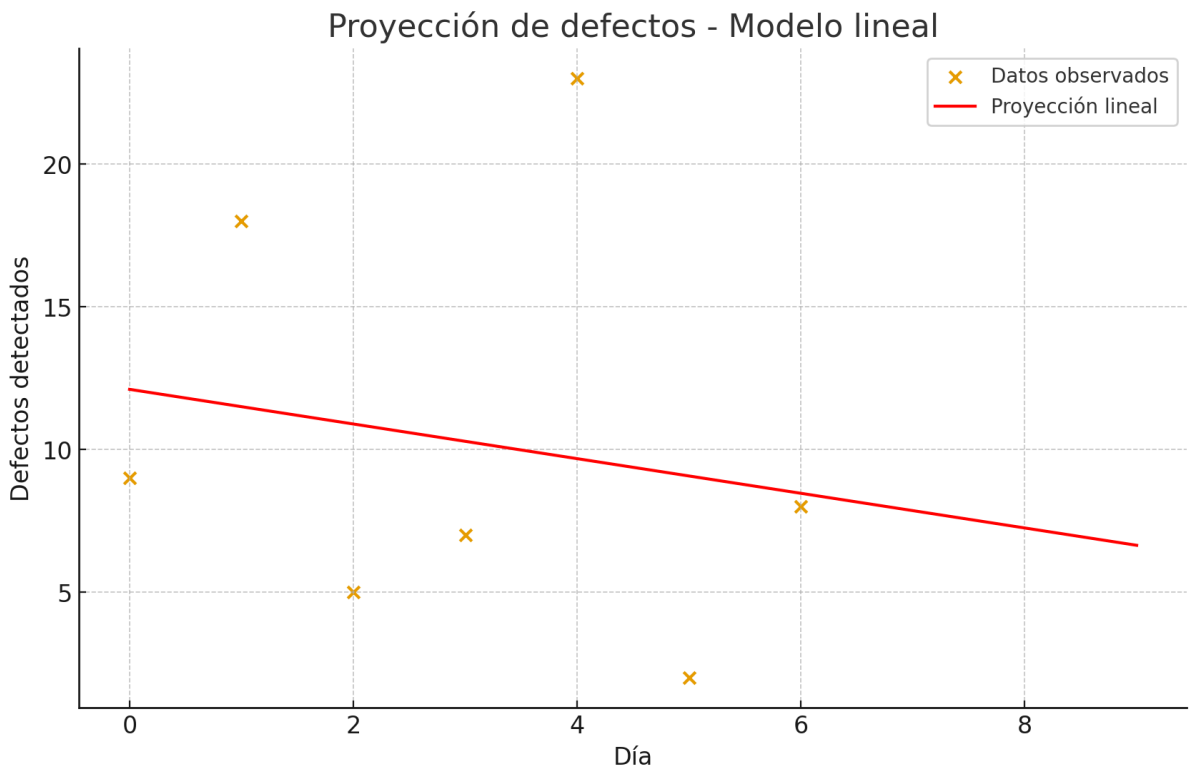
DD= 20 / 18 = 1.11 defectos por linea de codigo.

11. **Lea el tutorial "Kanban para uso genérico" del Ing. A. Ruiz de Mendarozqueta, explique en sus palabras el significado del parámetro TEP de esa metodología y cuál es el significado crítico en su utilización.**

El TEP es una metodología la cual nos permite establecer la cantidad de tareas que se están trabajando, las cuales, estas de pasar del estado 'para hacer' a el estado 'trabajando', de ahi, si el TEP esta lleno, no se puede volver a agregar más tareas a la seccion de 'trabajando' hasta que se libere un poco.

12. **Utilice los algoritmos discutidos en el taller denominado "taller_defectos.ipynb" sobre los posibles criterios para evaluar la proyección en los defectos aplicando un modelo de regresión aproximado como el allí mencionado. A continuación, utilice un modelo similar para estimar el número total de defectos a esperar en un aplicativo a partir de los siguientes datos obtenidos en los primeros días de test**.

Dia	Defectos
0	9
1	18
2	5
3	7
4	23
5	2
6	8



El modelo de regresión lineal estima que el total de defectos esperados en el aplicativo es aproximadamente 93.

13. **Repita el ejercicio anterior pero tomando el dataset de resultados de prueba obtenido en ejercicios anteriores, asimile "Día" a "Sesiones". Estime el número total de defectos de la implementación. Calcule el tamaño final del código (en Locs) usando "cat collatz.py | wc -l" o su equivalente en otros sistemas operativos, en función del número de defectos total proyectado estadísticamente (μ0) en el cálculo y teniendo en cuenta que puede calcular la cantidad de defectos "liberados" al finalizar el test como μr=μ0-Sμi siendo μi los defectos identificados en las sucesivas etapas de test**

Primero, calculamos μ0, que es el número total de defectos que se espera en la implementación antes de comenzar las pruebas. Luego, puedes calcular el tamaño final del código (en LOC) en función de μ0 y los defectos identificados en las sucesivas etapas de prueba (μi).

La fórmula para calcular el tamaño final del código en LOC sería:

Tamaño_Final_LOC = Tamaño_Inicial_LOC + (μ0 - μr) * LOC_Por_Defecto

Donde:

- Tamaño_Inicial_LOC es el tamaño del código inicial en LOC antes de comenzar las pruebas.
- μ0 es el número total de defectos proyectado estadísticamente antes de las pruebas.
- μr es el número total de defectos "liberados" al finalizar todas las pruebas.
- LOC_Por_Defecto es una estimación del número promedio de LOC agregados o modificados por cada defecto corregido.

La estimación del valor de LOC_Por_Defecto puede variar según el proyecto y la tecnología utilizada. Puedes obtener esta estimación basándote en la experiencia y las características del proyecto.

Para obtener μr, debes restar el número de defectos identificados en cada etapa de prueba (μi) del valor inicial μ0:

μr = μ0 - (μ1 + μ2 + μ3 + ... + μn)

Donde μ1, μ2, μ3, ... μn son los defectos identificados en cada etapa de prueba.

Una vez que tengas el valor de μr, puedes calcular el tamaño final del código en LOC utilizando la fórmula mencionada anteriormente.

Para medir el tamaño del código en LOC, puedes utilizar el comando wc -l en sistemas Unix/Linux o el equivalente en otros sistemas operativos. Por ejemplo:

cat collatz.py | wc -l

Esto mostrará el número de líneas de código en el archivo collatz.py. Asegúrate de realizar este cálculo al finalizar todas las etapas de prueba para obtener el tamaño final del código en función de los defectos identificados y corregidos.

14. **Lea el artículo adjunto "Marco para evaluar garantía en desarrollo de software" y a continuación explique cuál será el comportamiento cualitativo de incrementar el PCE de un proceso de calidad de software en el tiempo total de test óptimo para poder otorgar garantía a ese software.**

    Lo que causaría un incrementar el PCE de un proceso de calidad de software en el tiempo total de test optimo, es que se disminuiría la cantidad de defectos con los que saldría el software, provocando que no se creen riesgos para la cadena de valor que soporte, a la vez que no se tendrá que enfrentar a futuros costos de detección y remoción de defectos, los cuales pueden ser ordenes de magnitud superior a que fueran removidos en tiempo de Desarrollo. Sin embargo, hay que tener en cuenta que aumentar demasiado el PCE, debido a que esto incrementa el costo de nuestro proyecto, por lo que, teniendo competencia, el precio se elevaría por las nubes, además de que no termina saliendo rentable, puesto a que llegado a un determinado momento, se encontrarían cada vez menos defectos.

15. **Una organización ha establecido una línea de base organizacional sobre su performance de calidad que le permite confiar en entregar sus proyectos con un PCE=89% y defectos al momento del liberación (δr) de 0.12 def/FP. Se acaba de finalizar la construcción de un proyecto cuyo tamaño (S) es de 100 FP.**

    a. ¿Cuál es la expectativa de defectos totales (μ0) para ese proyecto?

μR = 100FP * 0,12 =12

μ0= 12/1-0,89 = 12/0,11 = 109,09

    b. ¿Qué densidad de defectos deberá esperarse al momento de finalizar la construcción (δ0)?

δ_0 = 109.09/100 =1,0909

    c. ¿Cuantos defectos debería planear detectar durante el período de V&V para ese proyecto?

V&V = 109,09 – 12 = 97,09

    d. Si esa misma organización detecta en sus primeros tres días de V&V 20 defectos cada día (60 defectos en total). Fundamente una opinión sobre ¿Cuál cree que será el comportamiento del total de defectos respecto a lo esperable en función de los parámetros históricos?.

Por los parámetros históricos, se puede observar que es probable que se tengan mas de 109 defectos, debido a que si en tres días detecte que tengo 60, es improbable que si se graficara, la curva parezca estabilizarse y caer entre el promedio.

16. En la presentación "Agile and software engineering , an invisible bond" resumen del artículo del mismo nombre se hace referencia la importancia de la deuda técnica como factor de deterioro en la performance de valor de un proyecto, indique de acuerdo al artículo que factores gestionados desde el proceso de calidad mejoran la gestión de deuda técnica y porque producen esa mejora.

Factores gestionados desde el proceso de calidad que mejoran la deuda técnica

1. **Gestión formal de la configuración y control de versiones**
   Qué mejora: permite rastrear cambios, revertir decisiones erróneas y asegurar consistencia en versiones del código.
   - **Por qué mejora la deuda técnica:** evita acumulación de código duplicado, dependencias rotas y regresiones funcionales.
2. **Revisión e inspección sistemática del código (peer review / code inspection)**
   - **Qué mejora:** detecta defectos y malas prácticas antes de la integración.
   - **Por qué mejora la deuda técnica:** previene la propagación de errores estructurales y reduce el costo de refactorización posterior.
3. **Medición y control de calidad del producto (métricas de defectos, PCE, CoPQ)**
   - **Qué mejora:** introduce evidencia objetiva sobre la calidad del código y del proceso.
   - **Por qué mejora la deuda técnica:** permite detectar fases donde se están generando defectos y aplicar correcciones preventivas, evitando el deterioro acumulativo.
4. **Gestión disciplinada de requisitos y trazabilidad**
   - **Qué mejora:** mantiene coherencia entre necesidades del cliente, diseño y pruebas.
   - **Por qué mejora la deuda técnica:** reduce ambigüedades funcionales que derivan en retrabajo y reimplementaciones costosas.
5. **Integración continua y testing automatizado**
   Qué mejora: asegura que cada incremento de software sea verificable y funcional.
   - **Por qué mejora la deuda técnica:** limita el crecimiento de defectos ocultos, acorta ciclos de realimentación y estabiliza la base del código.
6. **Análisis de causa raíz y lecciones aprendidas (post-mortem de defectos)**
   - **Qué mejora:** identifica patrones repetitivos de errores en el proceso.
   - **Por qué mejora la deuda técnica:** elimina causas sistémicas que generan retrabajo continuo, mejorando la sostenibilidad del desarrollo.

17. Utilizando el modelo simple de garantía expuesto en las clases teóricas razone como afecta la aplicación de una multa al punto de equilibrio del proceso, fundamente.

La aplicación de una multa aumenta el punto de equilibrio porque eleva los costos del proceso.

Si la multa es fija, se suman más costos generales, por lo que se necesita producir o vender más para cubrirlos.

Si la multa depende de defectos o fallas en la calidad, encarece cada unidad producida, reduciendo la ganancia por unidad y obligando a producir aún más para no perder dinero.

En ambos casos, el proceso se vuelve menos rentable y más riesgoso, por lo que invertir en calidad se vuelve esencial para reducir defectos y evitar que las penalizaciones hagan inviable el proyecto.