Introducción a la Programación

Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2023

Introducción a Validación & Verificaciónl

Problema, especificación, algoritmo, programa

Dado un problema a resolver (de la vida real), queremos:

- Poder describir de una manera clara y unívoca (especificación)
  - Esta descripción debería poder ser validada contra el problema real
- Poder diseñar una solución acorde a dicha especificación
  - Este diseño debería poder ser verificado con respecto a la especificación
- Poder implementar un programa acorde a dicho diseño
  - Este programa debería poder ser verificado con respecto a su especificación y su diseño
  - Este programa debería ser la solución al problema planteado

Validación y Verificación

Según wikipedia...
En el contexto de la ingeniería de software, verificación y validación (V&V) es el proceso de comprobar que un sistema de software cumple con sus especificaciones y que cumple su propósito previsto. También puede ser denominado como el control de la calidad del software

Calidad en Software

Uno de los objetivos principales en el desarrollo de sotware es obtener productos de alta calidad

Generalmente, se mide en atributos de calidad...	
Confiabilidad	Usabilidad
Corrección	Robustez
Facilidad de Mantenimiento	Seguridad (en datos, acceso, ...)
Reusabilidad	Funcionalidad
Verificabilidad + Claridad	Interoperabilidad
Etc..	

Asegurar la calidad vs Controlar la calidad

Una vez definidos los requerimientos de calidad, tengo que tener en cuenta que:

- Las calidad **no puede inyectarse al final**
- La calidad del producto depende de tareas realizadas durante **todo el proceso**
- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos

Validación y Verificación

Son procesos que ayudan a mostrar que el software cubre las expectativas para las cuales fue construido: contribuyen a garantizar calidad.

- Validación
  - ¿Estamos haciendo el producto correcto?
  - El software debería hacer lo que el usuario requiere de él.
- Verificación
  - ¿Estamos haciendo el producto correctamente?
  - El software debería realizar lo que su especificación indica.

Nociones básicas

- Falla
  - Diferencia entre los resultados esperados y reales
- Defecto
  - Desperfecto en algún componente del sistema (en el texto del programa, una especificación, un diseño, etc), que origina una o más fallas
- Error
  - Equivocación humana
    - Un **error** lleva a uno o más **defectos**, que están presentes en un producto de software
    - Un **defecto** lleva a cero, una o más **fallas**
    - Una **falla** es la manifestación del **defecto**

El proceso de V&V

- V&V debería aplicarse en cada instancia del proceso de desarrollo
  - En rigor no sólo el código debe ser sometido a actividades de V&V sino también todos los subproductos generados durante el desarrollo del software
- Objetivos principales
  - Descubrir **defectos** en el sistema
  - Asegurar que el software **respeta su especificación**
  - Determinar si satisface las **necesidades** de sus usuarios

Metas de la V&V

- La verificación y la validación deberían **establecer** la **confianza** de que el software es adecuado a su propósito
- Esto **NO** significa que esté completamente **libre de defectos**
- Sino que debe ser lo **suficientemente bueno** para su uso previsto y el tipo de uso determinará el grado de confianza que se necesita

Verificación estática y dinámica

- Una forma de realizar tareas de V&V es a través de análisis (de programas, modelos, especificaciones, documentos, etc.). En particular para el *código*, tenemos análisis estático y análisis dinámico
- **Dinámica**: trata con *ejecutar* y observar el *comportamiento* de un producto
- **Estática**: trata con el *análisis* de una *representación estática* del sistema para descubrir problemas

Verificación estática y dinámica

Técnicas de Verificación Estática

- Inspecciones, Revisiones
- Análisis de reglas sintácticas sobre código
- Análisis Data Flow sobre código
- Model checking
- Prueba de Teoremas
- Entre otras…

Técnicas de Verificación Dinámica

- Testing
- Run-Time Monitoring. (pérdida de memoria, performance)
- Run-Time Verification
- Entre otras…

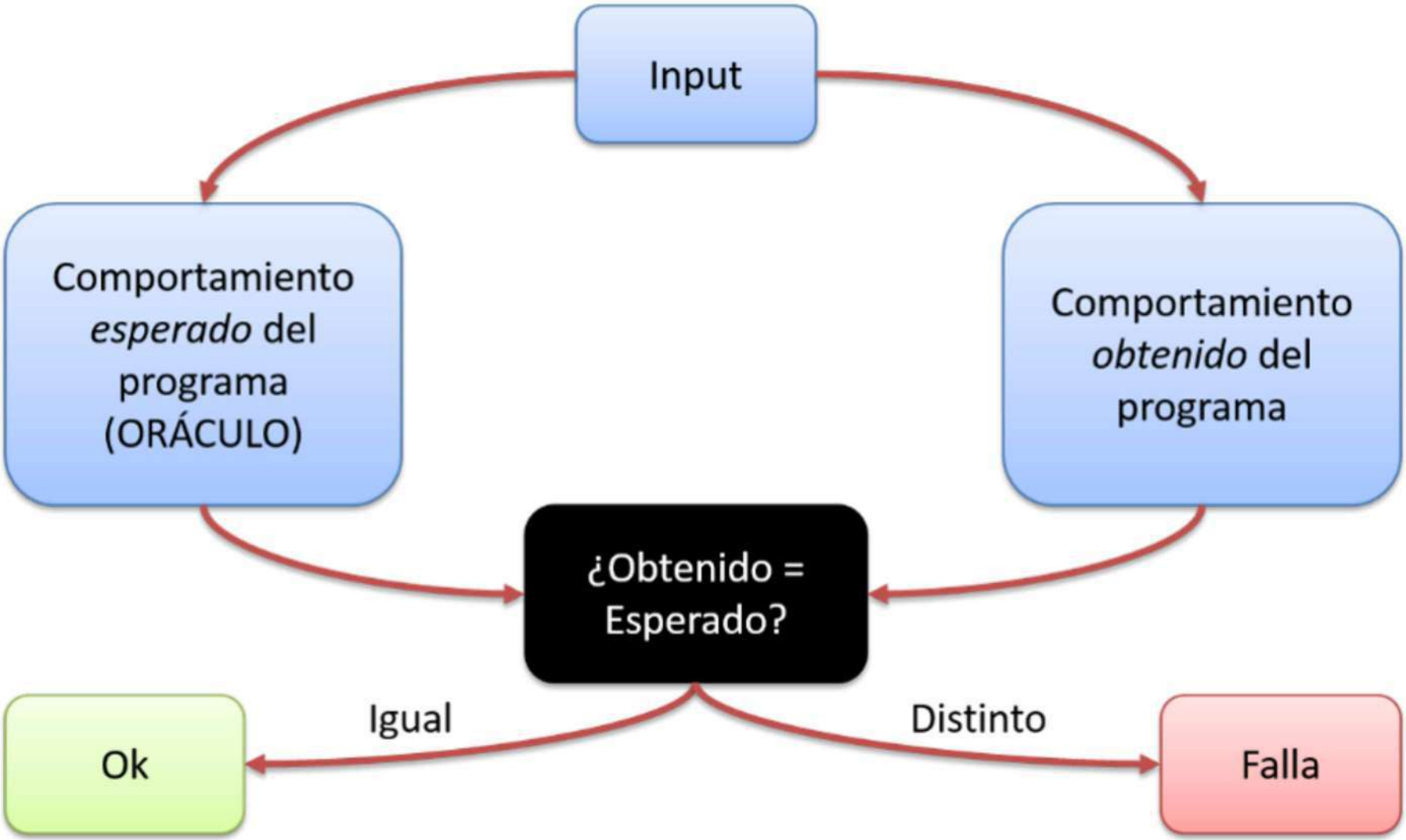
Recap: ¿Por qué escribir la especificación del problema?

- Nos ayuda a entender mejor el problema
- Nos ayuda a construir el programa
  - Derivación (Automática) de Programas
- Nos ayuda a prevenir errores en el programa
  - Testing
  - Verificación (Automática) de Programas

¿Qué es hacer testing?

- Es el proceso de ejecutar un producto para ...
  - Verificar que satisface los requerimientos (en nuestro caso, la especificación)
  - Identificar diferencias entre el comportamiento **real** y el comportamiento **esperado** (IEEE Standard for Software Test Documentation, 1983).
- Objetivo: encontrar defectos en el software.
- Representa entre el 30 % al 50 % del costo de un software confiable.

¿Cómo se hace testing?



Niveles de Test

- Test de Sistema
  - Comprende todo el sistema. Por lo general constituye el test de aceptación.

- Test de Integración
  - Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto
  - Testeamos la interacción, la comunicación entre partes

- Test de Unidad
  - Se realiza sobre una unidad de código pequeña, claramente definida.
    - ¿Qué es una unidad? Depende...

Test Input, Test Case y Test Suite

- **Programa bajo test**: Es el programa que queremos saber si funciona bien o no.
- **Test Input** (o dato de prueba): Es una asignación concreta de valores a los parámetros de entrada para ejecutar el programa bajo test.
- **Test Case:** Caso de Test (o caso de prueba). Es un programa que ejecuta el programa bajo test usando un dato de test, y chequea (automáticamente) si se cumple la condición de aceptación sobre la salida del programa bajo test.
- **Test Suite**: Es un conjunto de casos de Test (o de conjunto de casos de prueba).

Hagamos Testing

- ¿Cuál es el programa de test?
  - Es la implementación de una especificación.
- ¿Entre qué datos de prueba puedo elegir?
  - Aquellos que cumplen la **precondición (requieres)** en la especificación
- ¿Qué condición de aceptación tengo que chequar?
  - La condición que me indica la **postcondición (aseguras)** en la especificación.
- ¿Qué pasa si el dato de prueba no satisface la precondición de la especificación?
  - Entonces no tenemos ninguna condición de aceptación

Hagamos Testing

¿Cómo testeamos un programa que resuelva el siguiente problema?

problema $valorAbsoluto(n : \mathbb{Z}) : \mathbb{Z}\{$

requiere: $\{True\}$
asegura: $\{res = \|n\|\}$

$\}$

- Probar valorAbsoluto con 0, chequear que result=0
- Probar valorAbsoluto con -1, chequear que result=1
- Probar valorAbsoluto con 1, chequear que result=1
- Probar valorAbsoluto con -2, chequear que result=2
- Probar valorAbsoluto con 2, chequear que result=2
- ...etc.
- ¿Cuántas entradas tengo que probar?

Probando (Testeando) programas

- Si los enteros se representan con 32 bits, necesitaríamos probar 2^{32} datos de test.
- Necesito escribir un test suite de 4,294,967,296 test cases.
- Incluso si lo escribo automáticamente, cada test tarda 1 milisegundo, necesitaríamos $1193,04$ horas (49 días) para ejecutar el test suite.
- Cuanto más complicada la entrada (ej: secuencias), más tiempo lleva hacer testing.
- La mayoría de las veces, el testing exhaustivo no es práctico.

Limitaciones del testing

- Al no ser exhaustivo, el testing NO puede probar (demostrar) que el software funciona correctamente.

"El testing puede demostrar la presencia de errores nunca su ausencia" (Dijkstra)

- Una de las mayores dificultades es encontrar un conjunto de tests adecuado:
  - **Suficientemente grande** para abarcar el dominio y maximizar la probabilidad de encontrar errores.
  - **Suficientemente pequeño** para poder ejecutar el proceso con cada elemento del conjunto y minimizar el costo del testing.

¿Con qué datos probar?

- **Intuición:** hay inputs que son "parecidos entre sí" (por el tratamiento que reciben)
- Entonces probar el programa con uno de estos inputs, ¿equivaldría a probarlo con cualquier otro de estos parecidos entre sí?
- Esto es la base de la mayor parte de las técnicas
- ¿Cómo definimos cuándo dos inputs son "parecidos"?
  - Si únicamente disponemos de la especificación, nos valemos de nuestra *experiencia*

Hagamos Testing

¿Cómo testeamos un programa que resuelva el siguiente problema?

problema $valorAbsoluto(inn : \mathbb{Z}) : \mathbb{Z}\{$

requiere: $\{True\}$
asegura: $\{res = \|n\|\}$

$\}$

Ejemplo:

- Probar valorAbsoluto con 0, chequear que result=0
- Probar valorAbsoluto con un valor negativo x, chequear que result=-x
- Probar valorAbsoluto con un valor positivo x, chequear que result=x

Ejemplo: valorAbsoluto

- Programa a testear:
  - `int valorAbsoluto(int x)`
- Test Suite:
  - Test Case #1 (cero):
    - Entrada: $(x = 0)$
    - Salida Esperada: $result = 0$
  - Test Case #2 (positivos):
    - Entrada: $(x = 1)$
    - Salida Esperada: $result = 1$
  - Test Case #3 (negativos):
    - Entrada: $(x = -1)$
    - Salida Esperada: $result = 1$

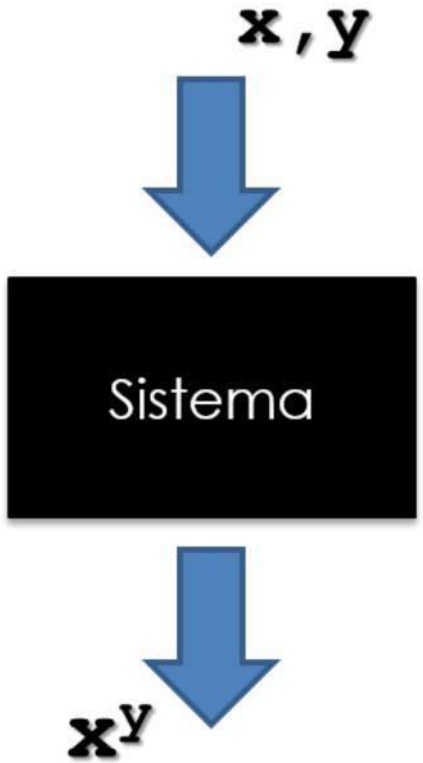
Retomando... ¿Qué casos de test elegir?

1. No hay un algoritmo que proponga casos tales que encuentren todos los errores en cualquier programa.
2. Ninguna técnica puede ser efectiva para detectar todos los errores en un programa arbitrario
3. En ese contexto, veremos dos tipos de criterios para seleccionar datos de test:
   - **Test de Caja Negra**: los casos de test se generan analizando la especificación sin considerar la implementación.
   - **Test de Caja Blanca**: los casos de test se generan analizando la implementación para determinar los casos de test.

Criterios de **caja negra** o funcionales

- Los datos de test se derivan a partir de la descripción del programa sin conocer su implementación.

problema *fastexp*($x : \mathbb{Z}, y : \mathbb{Z}$) : $\mathbb{Z}$ {
  requiere: $\{(0 \leq x \wedge 0 \leq y)\}$
  asegura: $\{res = x^y)\}$
}



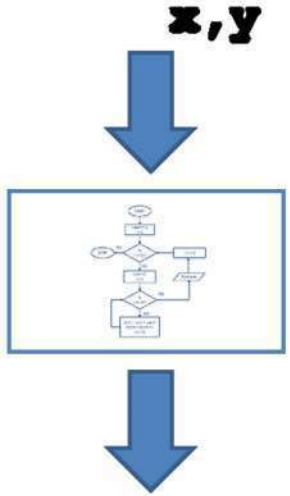
Criterios de **caja blanca** o estructurales

- Los datos de test se derivan a partir de la estructura interna del programa.

```
def fastexp(x: int, y: int) -> int:
    z: int = 1
    while(y != 0):
        if(esImpar(y)):
            z = z * x
            y = y - 1

        x = x * x
        y = y / 2

    return z
```



¿Qué pasa si y es potencia de 2?

¿Qué pasa si y = 2n -1?