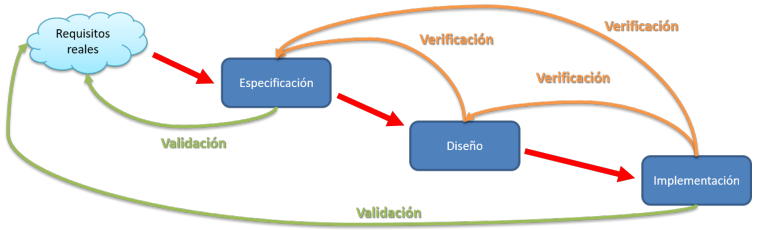
Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2023

Departamento de Computación - FCEyN - UBA

Introducción a Validación & Verificación

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 (también llamado Bug)
  - 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…

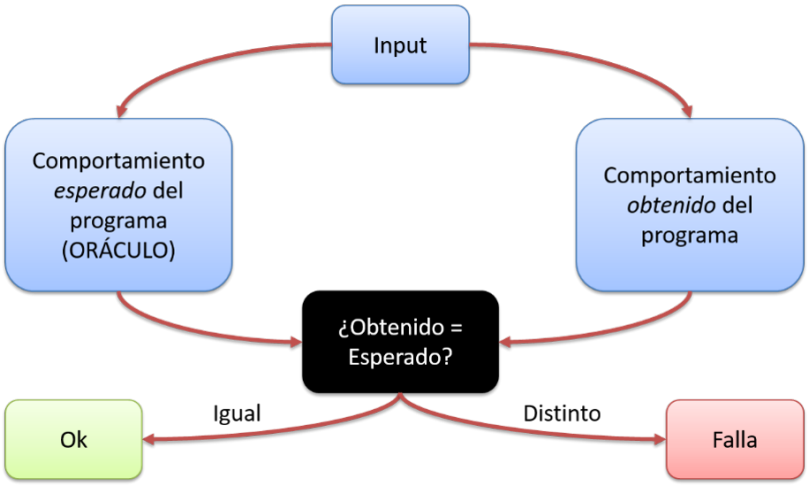
Recap: ¿Por qué escribir la especificación del problema?

- 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:
  - `valorAbsoluto :: Int -> Int`
- 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: $result = 1$

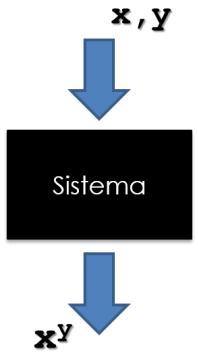
Retomando... ¿Qué casos de test elegir?

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: $\{res = x^y)\}$
}



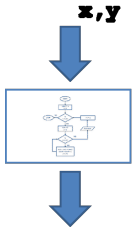
Criterios de **caja blanca** o estructurales

- 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?

¿Se puede automatizar el testing?

El diseño de casos de test, no sólo permite organizar u optimizar el trabajo de la persona que ejecutará los casos buscando fallas: existen herramientas que permiten *programar* estos casos de pruebas.

- Vimos que el nivel más básico del testing se denomina TEST UNITARIO
- La gran mayoría de los lenguajes de programación tienen herramientas que permiten programar casos de prueba.
- En el caso de Haskell: HUnit

HUnit

```
import Test.HUnit

-- La funcion que queremos probar
valorAbsoluto :: Int → Int
valorAbsoluto x | x ≥ 0 = x
                | otherwise = -x


-- Las pruebas unitarias
testSuiteValorAbs = test [
    "casoPositivo" ~: 1 ~?= (valorAbsoluto 1),
    "casoNegativo" ~: 5 ~?= (valorAbsoluto (-5)),
    "casoCero" ~: 0 ~?= (valorAbsoluto 0)
    ]

-- Corre todas las pruebas
correrTests = runTestTT testSuiteValorAbs
```

Método de Partición de Categorías

- Consiste en una técnica que permite generar casos de prueba de una manera metódica.
- Es aplicable a especificaciones formales, semiformales e inclusive, informales.

El método se puede resumir en los siguientes pasos:

1. Listar todos los problemas que queremos testear
2. Elegir uno en particular
3. Identificar sus parámetros o las relaciones entre ellos que condicionan su comportamiento. Los llamaremos genéricamente factores.
4. Determinar las características relevantes (categorías) de cada factor.
5. Determinar elecciones (choices) para cada característica de cada factor.
6. Clasificar las elecciones: errores, únicos, restricciones, etc
7. Armado de casos, combinando las distintas elecciones determinadas para cada categoría, y detallando el resultado esperado en cada caso.
8. Volver al paso 2 hasta completar todas las unidades funcionales.

Método de Partición de Categorías

Paso 1: Descomponer la solución informática en unidades funcionales

Consiste en enumerar todas las operaciones, funciones, funcionalidades, problemas que se probarán.

problema $esPermutacion(s1, s2 : seq\langle T\rangle) : Bool$ {
  asegura: $\{res = true \leftrightarrow ((\forall e : T)(cantidadDeApariciones(s1, e) = cantidadDeApariciones(s2, e)))\}$
}

problema $cantidadDeApariciones(s : seq\langle T\rangle, e : T) : \mathbb{Z}$ {
  requiere: $\{e \in s\}$
  requiere: $\{|s| > 0\}$
  asegura: $\{res = \sum_{i=0}^{|s|-1}(\text{if } s[i] = e \text{ then } 1 \text{ else } 0 \text{ fi})\}$
}

En este caso:

- esPermutacion
- cantidadDeApariciones

Método de Partición de Categorías

Paso 2: Elegir una unidad funcional

Lo ideal es llegar a testear todas las unidades funcionales. Un buen criterio, es empezar por aquellas que *son utilizadas* por otras.

En este caso:

- esPermutacion
- cantidadDeApariciones

Método de Partición de Categorías

Paso 3: Identificar factores

Esto pueden ser los parámetros del problema a testear (si el sistema es más complejo... podrían ser otros factores).

problema $cantidadDeApariciones(s : seq\langle T\rangle, e : T) : \mathbb{Z}$ {
  requiere: $\{|s| > 0\}$
  asegura: $\{res = \sum_{i=0}^{|s|-1}(\text{if } s[i] = e \text{ then } 1 \text{ else } 0 \text{ fi})\}$
}

En este caso:

- Tomamos T como $\mathbb{Z}$ (porque vamos a buscar datos concretos para probar)
- $s : seq\langle\mathbb{Z}\rangle$
- $e : \mathbb{Z}$

Método de Partición de Categorías

Paso 4: Determinar categorías

Las categorías son distintas características de cada factor, o características que relacionan diferentes factores, y que tienen influencia en los resultados. Son el resultado del análisis de toda la información disponible sobre la funcionalidad a testear.

En nuestro ejemplo, para cada parámetro podemos determinar las siguientes características:

- $s : seq\langle\mathbb{Z}\rangle$
  - ¿Tiene elementos?
- $e : \mathbb{Z}$
  - En este caso, para e no se distingue ninguna característica interesante
- Relación entre s y e (esta relación puede ser interesante)
  - ¿Pertenece e a s?

Método de Partición de Categorías

Paso 5: Determinar elecciones

Se trata de buscar los conjuntos de valores donde se espera un comportamiento similar. Se basa en las especificaciones, la experiencia, el conocimiento de errores.

En nuestro ejemplo, para cada categoría, determinamos sus elecciones o choices:

- $s : seq\langle\mathbb{Z}\rangle$
  - ¿Tiene elementos?
    - Si
    - No
- $e : \mathbb{Z}$
- Relación entre s y e
  - ¿Pertenece e a s?
    - Si
    - No

Método de Partición de Categorías

Paso 6: Clasificar las elecciones

Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional.
Las clasificaciones más comunes son:

- Error: Se clasificarán como error aquellas elecciones que por sí mismas determinen que como resultado de la ejecución el sistema debe detectar un error o que no está definido su comportamiento.
- Otros posibles valores: Único, Restricción, etc. (más adelante ampliaremos).

En nuestro ejemplo, para cada elecciones o choices, analizamos si debemos clasificarlo especialmente:

- $s : seq\langle\mathbb{Z}\rangle$
  - ¿Tiene elementos?
    - Si
    - No [ERROR]
- $e : \mathbb{Z}$
- Relación entre s y e
  - ¿Pertenece e a s?
    - Si
    - No

Método de Partición de Categorías

Paso 7: Armar los casos de test

Finalmente, se combinarán las distintas elecciones de las categorías consideradas generando los distintos casos de test. Cada combinación es un caso de test, el cual deberá tener identificado con claridad su resultado esperado.
En nuestro ejemplo, deberemos combinar:

- ¿s tiene elementos?: Si
- ¿s tiene elementos?: No
- ¿e pertenece a s?: Si
- ¿e pertenece a s?: No

Tenemos 4 elecciones posibles a combinar

- ¿Nos interesan todas las combinaciones?
- ¿Cuántos casos de test tenemos?
- Las elecciones marcadas como ERROR y UNICO, suelen no ser combinables (recortando así la cantidad de combinaciones a realizar).
- Las elecciones RESTRICCION, condicionan las combinaciones posibles.
- Tip: estas elecciones son las primeras a considerar en el armado de casos.

Método de Partición de Categorías

Paso 7: Armar los casos de test

En este caso, sólo nos quedan 3 casos interesantes (nos ahorramos 1)

- Por cada caso, debemos describir su resultado esperado: es importante indicar si el resultado será un posible resultado correcto u esperable o un error o comportamiento indefinido.
- Recordar que los casos de prueba definido serán una herramienta que eventualmente otra persona pueda ejecutar los test: eligiendo datos concretos y comparando el resultado obtenido con el esperado.

Característica	¿s tiene elementos?	¿e pertenece a s?	Resultado esperado	Comentario
Caso 1: S sin elementos	No	-	ERROR: no está espeficado que sucede en este caso.	Como la elección No es un ERROR, no importa el valor
Caso 2: E no pertenece	Si	No	OK: 0	
Caso 3: S tiene elementos y e Pertenece	Si	Si	OK: el resultado obtenido debe ser igual a la cantidad de	

- La tabla es una representación gráfica y práctica de los casos.
- Suele ocurrir, que las primeras columnas son siempre de aquellas elecciones que tienen errores, únicos o restricciones entre sus posibles valores: porque descartan casos hacia la derecha.

Método de Partición de Categorías

Paso 6: Clasificar las elecciones - Volviendo un paso atrás

Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional.
Las clasificaciones más comunes son:

- Único: Son aquellas elecciones que no necesitan combinarse con ninguna otra elección para determinar el resultado esperado.

problema *siElPrimeroEsCinco*($x : \mathbb{Z}, y : \mathbb{Z}$) : $\mathbb{Z}$ {
  asegura: $\{x = 5 \longrightarrow res = 5\}$
  asegura: $\{x \neq 5 \longrightarrow res = x + y\}$
}

- Factor: x
  - Característica: valor
    - Elección: Igual a 5 [ÚNICO]
    - Elección: Distinto que 5
- Factor: y

Método de Partición de Categorías

Paso 6: Clasificar las elecciones - Volviendo un paso atrás

Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional.
Las clasificaciones más comunes son:

- Único: Son aquellas elecciones que no necesitan combinarse con ninguna otra elección para determinar el resultado esperado.

problema *siElPrimeroEsCincoHaceOtraCosa*($x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z}$) : $\mathbb{Z}$ {
  requiere: $\{x = 5 \longrightarrow z \neq 0\}$
  asegura: $\{x = 5 \longrightarrow res = x + y/z\}$
  asegura: $\{x \neq 5 \longrightarrow res = x + y\}$
}

- Factor: x
  - Característica: valor
    - Elección: Igual a 5 [RESTRICCIÓN]
    - Elección: Distinto que 5
- Factor: y
- Factor: z
  - Característica: valor
    - Elección: Igual a 0
    - Elección: Distinto de 0 Sólo si $x \neq 5$ por RESTRICCIÓN

Método de Partición de Categorías

Si hay otra funcionalidad a testear $\longrightarrow$ Paso 2: Elegir una unidad funcional

Si probamos *esPermutacion* luego de haber testeado *cantidadDeApariciones*, podemos asumir que *cantidadDeApariciones* funciona correctamente.

En este caso:

- esPermutacion
- cantidadDeApariciones

Y repetimos el proceso

HUnit

```
import Test.HUnit

-- La funcion que queremos probar
cantidadDeApariciones :: [Int] → Int → Int
cantidadDeApariciones [] _ = 0
cantidadDeApariciones (x:xs) e
 | x == e = 1 + pasoRecursivo
 | otherwise = pasoRecursivo
 where pasoRecursivo = cantidadDeApariciones xs e

-- Las pruebas unitarias
testSuiteCantidadApariciones = test [
    "eNoPertenece" ~: 0 ~?= (cantidadDeApariciones [1,2,3] 4),
    "ePertenece" ~: 3 ~?= (cantidadDeApariciones [1,2,(-2),4,(-2),1,(-2)] (-2))
    ]

-- Correr todas las pruebas
correrTests = runTestTT testSuiteCantidadApariciones
```


Otro ejemplo

Diseñar los casos de test de caja negra utilizando el método de partición por categorías para el siguiente problema:

problema *multiplosDeN*($n : \mathbb{Z}, s : seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z}\rangle$ {
  requiere: {No hay elementos repetidos en s}

asegura: {*res* contiene los elementos de s múltiplos de n, respetendo el orden }
}

Método de Partición de Categorías

Paso 1: Descomponer la solución informática en unidades funcionales

Consiste en enumerar todas las operaciones, funciones, funcionalidades, problemas que se probarán.

En nuestro caso, este paso ya está listo:

problema *multiplosDeN*($n : \mathbb{Z}, s : seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z}\rangle$ {
  requiere: {No hay elementos repetidos en s}

asegura: {*res* contiene los elementos de s múltiplos de n, respetendo el orden }
}

Método de Partición de Categorías

Paso 2: Elegir una unidad funcional

Este paso también ya lo tenemos listo:

problema *multiplosDeN*($n : \mathbb{Z}, s : seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z}\rangle$ {
  requiere: {No hay elementos repetidos en s}

asegura: {*res* contiene los elementos de s múltiplos de n, respetendo el orden }
}

Método de Partición de Categorías

Paso 3: Identificar factores

Estos son los parámetros del problema a testear.

problema *multiplosDeN*($n : \mathbb{Z}, s : seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z}\rangle$ {
  requiere: {No hay elementos repetidos en s}

asegura: {*res* contiene los elementos de s múltiplos de n, respetendo el orden }
}

En este caso:

- $n : \mathbb{Z}$
- $s : seq\langle\mathbb{Z}\rangle$

Método de Partición de Categorías

Paso 4: Determinar categorías

Las categorías son distintas características de cada factor, o características que relacionan diferentes factores, y que tienen influencia en los resultados. Son el resultado del análisis de toda la información disponible sobre la funcionalidad a testear.

En nuestro ejemplo, para cada parámetro podemos determinar las siguientes características:

- $n : \mathbb{Z}$
  - valor
- $s : seq\langle\mathbb{Z}\rangle$
  - ¿Tiene elementos? ¿Tiene elementos repetidos?
- Relación entre n y s
  - Cantidad de múltiplos de n en s

Método de Partición de Categorías

Paso 5: Determinar elecciones

Se trata de buscar los conjuntos de valores donde se espera un comportamiento similar. Debería ser una partición sin dejar valores afuera.

En nuestro ejemplo, para cada categoría, determinamos sus elecciones o choices:

- $n : \mathbb{Z}$
  - valor
    - < 0
    - $= 0$
    - > 0
- $s : seq\langle\mathbb{Z}\rangle$
  - ¿Tiene elementos? ¿Tiene elementos repetidos?
    - No tiene elementos
    - Sí tiene elementos, pero no repetidos
    - Sí tiene elementos repetidos
- Relación entre n y s
  - Cantidad de múltiplos de n en s
    - 0
    - 1
    - > 1

Método de Partición de Categorías

Paso 6: Clasificar las elecciones

Se trata de identificar algunas propiedades o restricciones de las elecciones en el marco de la unidad funcional.

Las clasificaciones más comunes son:

- **Error**: Se clasificarán como error aquellas elecciones que por sí mismas determinen que como resultado de la ejecución el sistema debe detectar un error o que no está definido su comportamiento.
- **Único**: Nos libra de realizar todas las combinaciones con esta elección
- **Restricción**: Nos permite indicar una condición que se debe cumplir para combinar con esta elección

Método de Partición de Categorías

Paso 6: Clasificar las elecciones

- ¿Tenemos casos de ERROR?

  Sí. El requiere del problema exige que s no tenga elementos repetidos.

- ¿Tenemos casos de ÚNICO?

  Sí. Cuando la lista s es vacía.

- ¿Tenemos casos de RESTRICCIÓN?

  Sí. Cuando valor es 0 no puede haber más de un múltiplo.

- ¿Nos interesan todas las combinaciones?

  No.

Método de Partición de Categorías

Paso 6: Clasificar las elecciones

- $n : \mathbb{Z}$
  - valor
    - < 0
    - $= 0$ [RESTRICCIÓN]
    - > 0
- $s : seq\langle\mathbb{Z}\rangle$
  - ¿Tiene elementos? ¿Tiene elementos repetidos?
    - No tiene elementos [ÚNICO]
    - Sí tiene elementos, pero no repetidos
    - Sí tiene elementos repetidos [ERROR]
- Relación entre n y s
  - Cantidad de múltiplos de n en s
    - 0
    - 1
    - > 1 Sólo si $n \neq 0$ por RESTRICCIÓN

¿Cuántos casos nos quedaron? **1 (error) + 1 (único) + 1*1*2 (restricción) + 2*1*3 = 10 casos**

Método de Partición de Categorías

Paso 7: Armar los casos de test

- Por cada caso, debemos describir su **resultado esperado**: es importante indicar si el resultado será un posible resultado correcto u esperable o un error o comportamiento indefinido.
- Los casos de prueba definidos serán una herramienta para que eventualmente otra persona pueda ejecutar los test: eligiendo datos concretos y comparando el resultado obtenido con el esperado.

Método de Partición de Categorías

Paso 7: Armar los casos de test

Analizamos cada caso Caso 1: s tiene elementos repetidos Caso 2: s vacía Caso 3: *valor* $= 0$, cantidad de múltiplos $= 0$ Caso 4: *valor* $= 0$, cantidad de múltiplos $= 1$ Caso 5: *valor* < 0, cantidad de múltiplos $= 0$ Caso 6: *valor* < 0, cantidad de múltiplos $= 1$ Caso 7: *valor* < 0, cantidad de múltiplos > 1 Caso 8: *valor* > 0, cantidad de múltiplos $= 0$ Caso 9: *valor* < 0, cantidad de múltiplos $= 1$ Caso 10: *valor* < 0, cantidad de múltiplos > 1

caso	descripción	¿tiene elem? ¿repetidos?	valor	cant. de múlt	resultado esperado
1	repetidos	sí (repe)	-	-	no especificado
2	lista vacía	no	-	-	lista vacía
3	valor 0, múlt 0	sí	0	0	lista vacía
4	valor 0, múlt 1	sí	0	1	[0]
5	valor < 0, múlt 0	sí	< 0	0	lista vacía
6	valor < 0, múlt 1	sí	< 0	1	lista con el múlt
7	valor < 0, múlt > 1	sí	< 0	> 1	lista con los múlt
8	valor > 0, múlt 0	sí	> 0	0	lista vacía
9	valor > 0, múlt 1	sí	> 0	1	lista con el múlt
10	valor > 0, múlt > 1	sí	> 0	> 1	lista con los múlt

Método de Partición de Categorías

Comentarios finales

- Este es sólo un método más (de otros tantos que existen) para encarar el problema de generar casos de prueba.
- No se evaluará su uso de manera rigurosa en la materia:
  - La intención es que cuenten con alguna herramienta en caso de que se encuentren frente a la situación de no saber cómo testear sus problemas.
  - No existe una única forma de generar casos de prueba!
  - Lo importante, es que sus casos de prueba abarquen, en la medida de lo posible, todas las casuísticas posibles con respecto a loss parámetros de entrada.

HUnit

```
module MultiplosDeN where

-- La funcion que queremos probar
multiplosDeN :: Int → [Int] → [Int]
multiplosDeN _ [] = []
multiplosDeN n (x:xs)
 | n == 0 && x == 0 = [0]
 | n /= 0 && mod x n == 0 = x : pasoRecursivo
 | otherwise = pasoRecursivo
 where pasoRecursivo = multiplosDeN n xs
```


HUnit

```
import MultiplosDeN
import Test.HUnit


-- Las pruebas unitarias
testSuiteMultiplosDeN = test [
  "lista vacia" ~: [] ~?= (multiplosDeN 4 []),
  "valor 0, mult 0" ~: [] ~?= (multiplosDeN 0 []),
  "valor 0, mult 1" ~: [0] ~?= (multiplosDeN 0 [-1,0,9]),
  "valor < 0, mult 0" ~: [] ~?= (multiplosDeN (-3) [20,13,-4]),
  "valor < 0, mult 1" ~: [-16] ~?= (multiplosDeN (-8) [9,-16,7]),
  "valor < 0, mult > 1" ~: [0,-14] ~?= (multiplosDeN (-7) [0,-14,15]),
  "valor > 0, mult 0" ~: [] ~?= (multiplosDeN 5 [4,-7,9]),
  "valor > 0, mult 1" ~: [7] ~?= (multiplosDeN 7 [7,8,-9]),
  "valor > 0, mult > 1" ~: [-22,33] ~?= (multiplosDeN 11 [-22,10,33])
  ]

-- Correr todas las pruebas
correrTests = runTestTT testSuiteMultiplosDeN
```