# Ingeniería de Software I
# Testing

Jocelyn Simmonds

Departamento de Ciencias de la Computación

# Verificación y Validación

En las palabras de Barry Boehm:

Validación: *Are we building the right product?*

Verificación: *Are we building the product right?*

# Verificación y Validación

En las palabras de Barry Boehm:

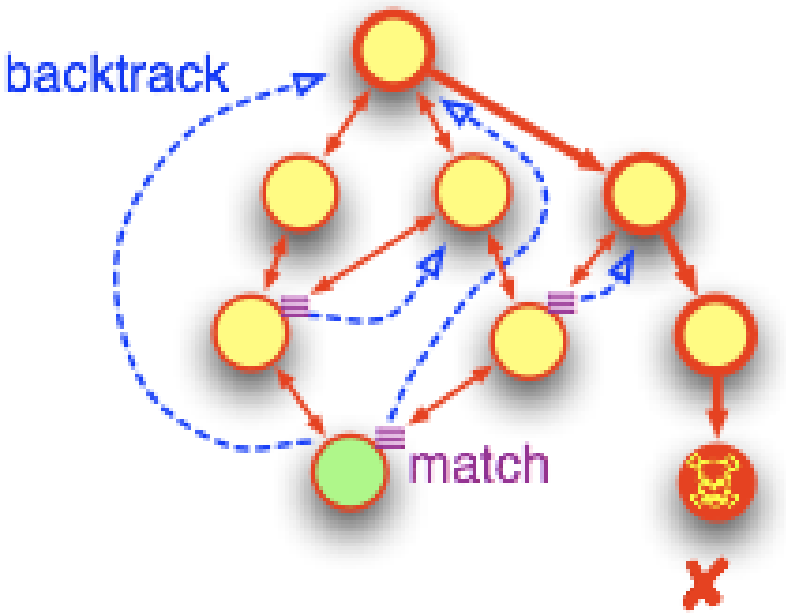
Validación: *Are we building the right product?*

Verificación: *Are we building the product right?*

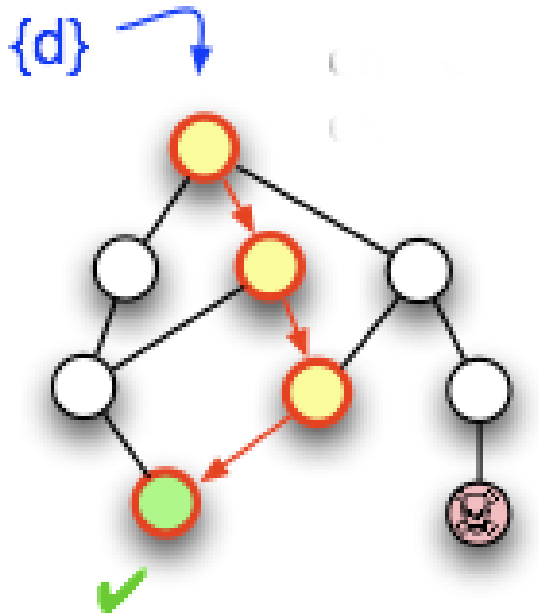
(tanto para los requisitos funcionales como los no-funcionales)

# Hoy hablaremos de testing ...

Análisis: construir un modelo del sistema y analizar si cumple ciertas propiedades




Testing: ejecutar algunas trazas para ver si se produce un error




En la practica, no podemos ejecutar el programa sobre todas sus entradas posibles ...

# ¿Garantía de correctitud?

Vamos a tener que elegir que casos de prueba ejecutar:

- ¿Como sabemos si hemos hecho suficiente testing?

# ¿Garantía de correctitud?

Vamos a tener que elegir que casos de prueba ejecutar:

- ¿Como sabemos si hemos hecho suficiente testing?
- Testing solo puede indicarnos que existen defectos . . .
- . . . pero nosotros tenemos que deducir la causa:
  - falto un requisito
  - algo quedo mal especificado
  - algún requisito no era factible
  - hay algún problema con el diseño del sistema
  - usamos el algoritmo equivocado
  - hay errores en la implementación
  - . . .

# Tipos de errores

- Errores de sintaxis
  - detectar al compilar/interpretar
- Errores en tiempo de ejecución
  - usuario le quita permisos a una app
  - se cae la red ...
- Errores lógicos
  - variable mal inicializada
  - testeamos la condición incorrecta ...

Un error puede introducir uno o mas defectos ... los se manifiestan a través de fallas.

# Tipos de errores

- Errores de sintaxis
  - detectar al compilar/interpretar
- Errores en tiempo de ejecución
  - usuario le quita permisos a una app
  - se cae la red ...
- Errores lógicos
  - variable mal inicializada
  - testeamos la condición incorrecta ...

Un error puede introducir uno o mas defectos ... los se manifiestan a través de fallas.

$$\text{Falla} = \text{desviación entre el resultado esperado y el real}$$

# Algunas cualidades deseables de casos de pruebas

- No solo deben alertarnos de la existencia de un error . . . ojala nos ayuden a detectar donde esta
- Deberían ser fáciles de repetir, para que los podamos ejecutar una vez corregido el error
- Que encuentren problemas de verdad y no casos hipotéticos
- Que no sean redundantes: es mejor pocos tests complejos, que muchos similares
- Que ejerciten una "buena" parte del producto (cobertura)
- Que sean mantenibles . . . los casos de prueba deben evolucionar con el código

# Existen distintos tipos de testing . . .

Testing dentro del equipo de desarrollo:

- Pruebas unitarias: probar una componente a la vez (método, clase)
- Pruebas de integración: probar una colección de componentes que dependen unos de otros
- Pruebas de sistema: probar el sistema completo
- Pruebas de regresión: enfocarse en probar lo que cambio entre dos versiones
- Pruebas de humo: pruebas rapidas para descubrir fallas simples pero severas antes de hacer un release
- De volumen de procesamiento, de performance, de estrés, que sea instalable, usable, seguro, . . .

# Existen distintos tipos de testing . . .

Testing afuera del equipo de desarrollo:

- Pruebas de aceptación:
  - los casos de prueba se extraen solo de los requerimientos
  - deben ser diseñados por el cliente y/o usuario . . .
  - . . . y el cliente y/o usuario debe ejecutarlos
  - son un paso importante para lograr la aceptación formal del producto entregado
- Pruebas tipo alfa y beta:
  - entregar una versión preliminar a usuarios "amigos" para que nos ayuden a encontrar fallas y detectar posibles mejoras
  - alfa = usuario interno ("amigo"), beta = usuario externo y real

# Testing funcional

Cobertura = que porcentaje de los requisitos fueron testeados:

- Ejecutando escenarios de uso
  - en el caso de GUI, ojala usar algún framework que permita "grabar" las pruebas para volver a ejecutarlas
  - pero es frágil ante cambios en la interfaz gráfica
- En base a tablas de decisiones
  - "si el usuario ingresa las credenciales incorrecta, entonces se debe mostrar una pagina de error"
- En base a grafos de causas y efectos
  - "si el usuario borra un pedido, este pedido debe quedar archivado, hay que cancelar el envío, ..."

# Especificando un caso de pruebas

Se puede documentar los casos de prueba funcionales, creando una ficha por cada prueba. Esta ficha usualmente tiene tres partes:

- Introducción/visión general
  - información general acerca del caso de prueba
- Cuerpo del caso de prueba
  - actividades a realizar para ejecutar el caso de prueba
- Resultados esperados
  - cual es la salida esperada, cual es el impacto del defecto si esta prueba falla, etc.

# Ficha: introducción

- Identificador: un identificador único como `T001`
- Dueño/creador: nombre de la persona que diseño el caso de prueba, o el responsable de su desarrollo
- Versión: 1.1 $\rightarrow$ 1.2, etc.
- Nombre: un nombre descriptivo del caso de prueba, que indique el propósito de este y su campo esperado de aplicación
- Requerimientos testeados: indicar los requisitos que están siendo probados
- Propósito: breve descripción del propósito de la prueba, y la funcionalidad que chequea
- Dependencias: indicar dependencias a otros subsistemas
  - "se valida el RUT con el Registro Civil, la prueba no funciona si ese sistema no esta accesible"

# Ficha: cuerpo (1/2)

- Inicialización: las acciones que deben ser ejecutadas ANTES de comenzar a ejecutar el caso de prueba
  - "el usuario debe estar logeado como administrador del sistema"
- Descripción de los datos de entrada
- Acciones: lista paso a paso de las acciones a realizar para completar la prueba
  1. Ir al menú "Configuración"
  2. Seleccionar la opción "Cargar configuración"
  3. Subir el archivo de configuración al sistema
  4. ...

# Ficha: cuerpo (2/2)

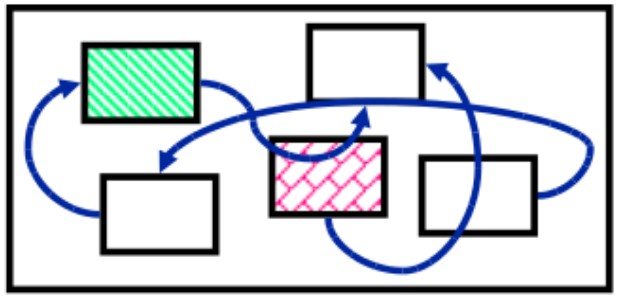
- Finalización: las acciones que deben ser ejecutadas DESPUÉS de terminar el caso de prueba
  - usualmente acciones de "limpieza", para que el resto de las pruebas puedan ser ejecutadas correctamente
- Ambiente de prueba/configuración:
  - información acerca de la configuración del hardware o software en el cuál se ejecutará el caso de prueba
  - levanten un ambiente de testing, ¡no hagan pruebas en el ambiente de producción!

# Ficha: resultados

- Salida esperada: una descripción de lo se debería ver tras haber completado todos los pasos de la prueba
- Salida obtenida: breve descripción de lo que se obtuvo al ejecutar los pasos de la prueba
- Resultado: pass/fail
- Severidad: el impacto del defecto en el sistema
  - grave, mayor, normal, menor
- Evidencia: un screenshot, un stack trace, una parte de un log, etc. de la salida obtenida
- Seguimiento: si la prueba fallo, referencia al defecto asociado
  - ojala el ID en un issue tracker
- Estado: no iniciado, en curso, terminado

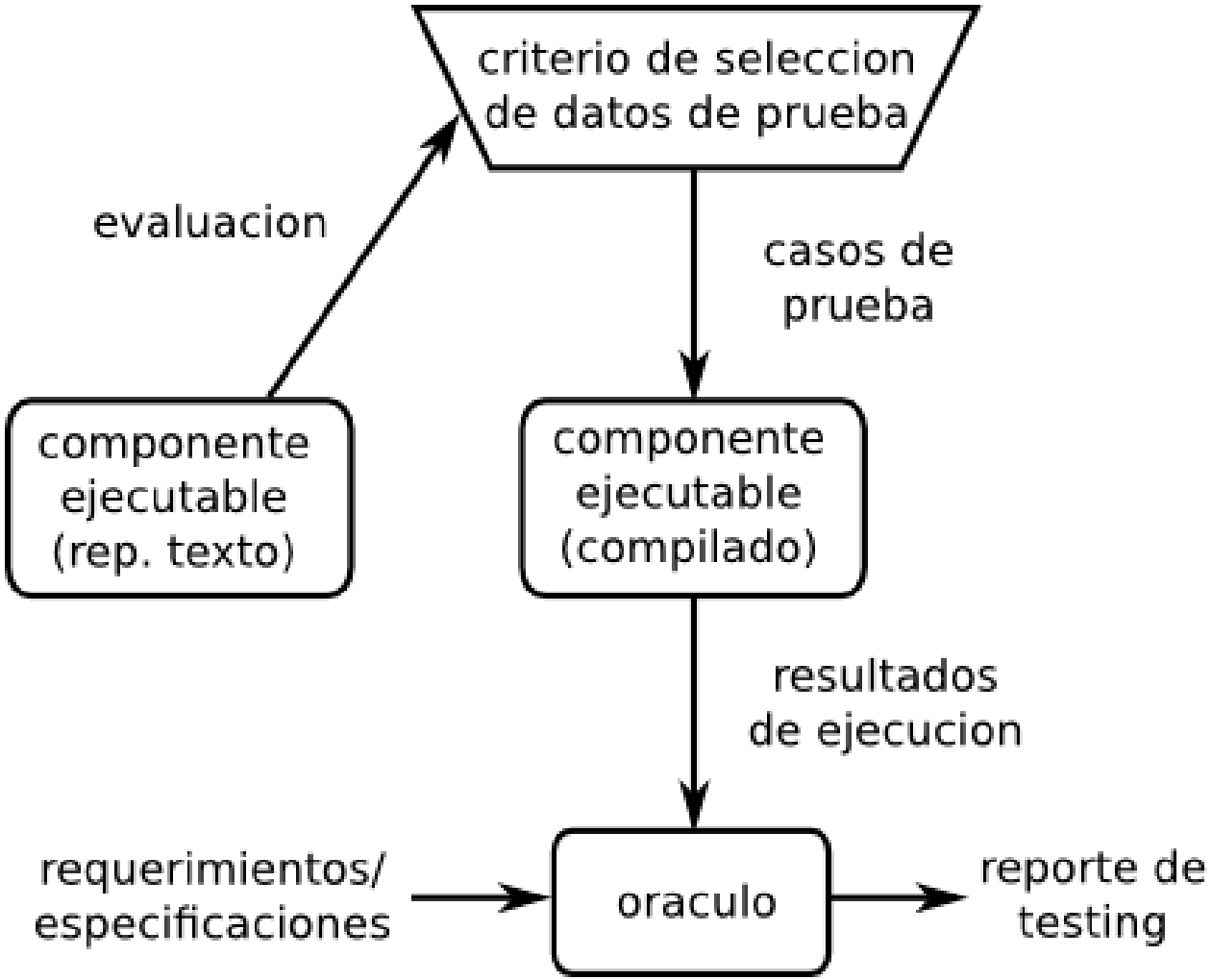
# Distintos enfoques para diseñar casos de prueba

- Caja blanca – en base al código
  - caminos básicos
  - estructuras de control
  - ...

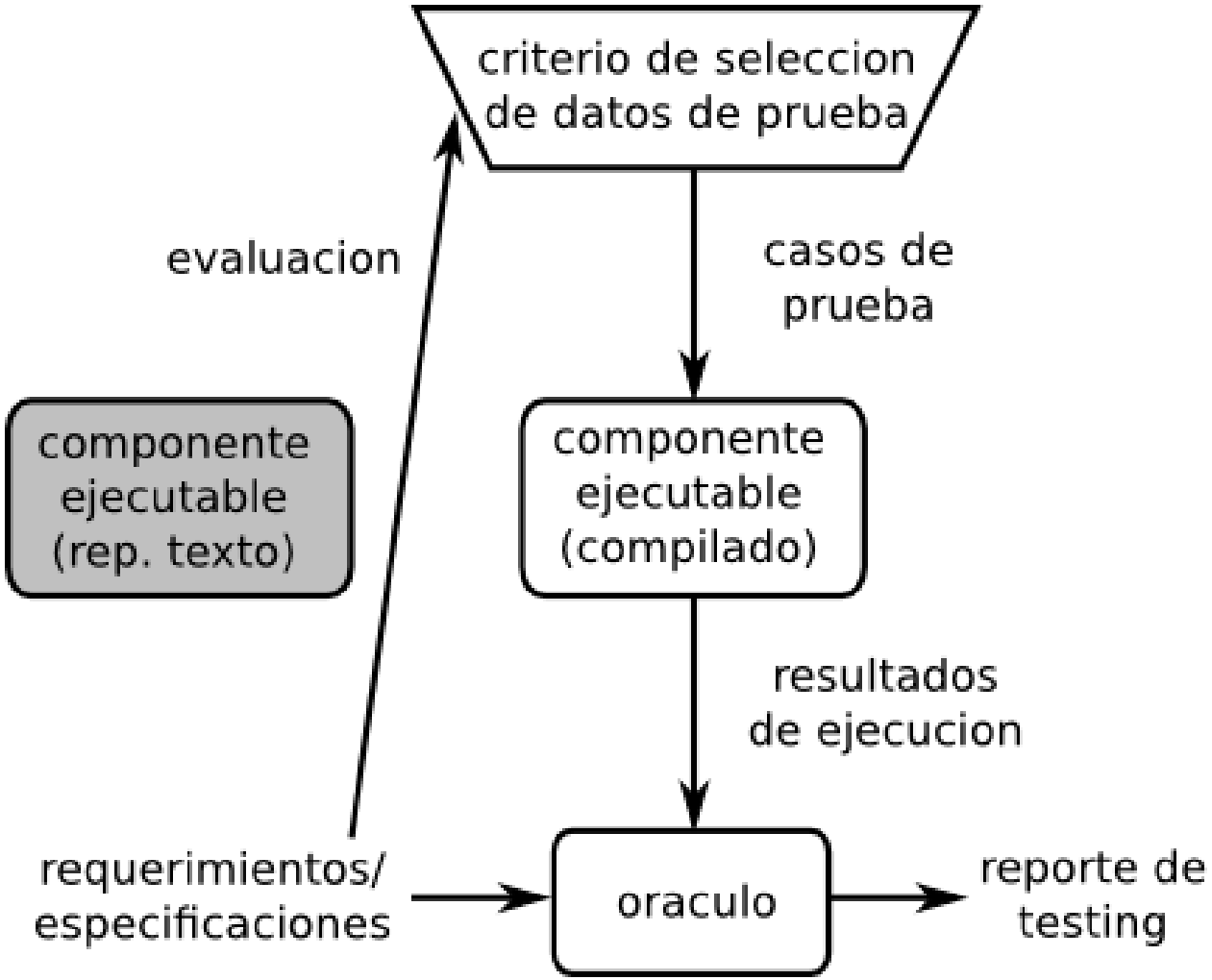


- Caja negra – en base a requerimientos
  - clases de equivalencia
  - análisis de fronteras
  - ...

# Caja blanca


criterio de seleccion
de datos de prueba
evaluacion
casos de
prueba
componente
ejecutable
(rep. texto)
componente
ejecutable
(compilado)
resultados
de ejecucion
requerimientos/
especificaciones
oraculo
reporte de
testing

# Caja negra


criterio de seleccion
de datos de prueba
evaluacion
casos de
prueba
componente
ejecutable
(rep. texto)
componente
ejecutable
(compilado)
resultados
de ejecucion
requerimientos/
especificaciones
oraculo
reporte de
testing

# Caja blanca Y caja negra?

- Caja negra
  - A veces no tenemos acceso al código
  - A veces, no nos importa como esta implementado, solo como funciona (o alguna cualidad no funcional, como performance, seguridad, etc.)
  - Nos ayuda a detectar que features faltan
- Caja blanca
  - La idea es usar toda la información a nuestra disposición para generar casos de prueba
  - Podemos mejor identificar las debilidades del sistema si conocemos su estructura

# Formalicemos un poco

Dado un programa $P$, su dominio de entradas $D(P)$ y su dominio de salidas $R(P)$:

- Un programa es una función parcial $P : D(P) \to R(P)$
- Digamos que tenemos un oráculo $OR \subseteq D(P) \times R(P)$, que para cualquier entrada nos dice el resultado esperado
- $P(d)$ es correcto si $(d, P(d)) \in OR$
- Un programa es correcto si $\forall d \in D(P), P(d)$ es correcto

# Definición de test suite

De forma similar:

- Un caso de prueba $t$ es un elemento de $D(P)$
- Un conjunto de pruebas $T$ (test set o test suite) es un subconjunto finito de $D(P)$
- Un caso de prueba es exitoso (o "pasa") si $P(t)$ es correcto
- Un conjunto de pruebas es exitoso si $\forall t \in T, P(t)$ es correcto

# Criterio para seleccionar casos de prueba

- Un criterio de selección de pruebas $C(T, P)$ es un predicado que especifica si un test suite satisface un cierto criterio con respecto a un programa P
- Podemos definir el test suite $T$ que satisface el criterio $C$ como
  $\{t \in T \mid T \subseteq D \wedge C(T, P)\}$

# Criterio para seleccionar casos de prueba

- Un criterio de selección de pruebas $C(T, P)$ es un predicado que especifica si un test suite satisface un cierto criterio con respecto a un programa P
- Podemos definir el test suite $T$ que satisface el criterio $C$ como
  $\{t \in T \mid T \subseteq D \wedge C(T, P)\}$
- Un criterio de selección es ideal si:
  para cualquier programa $P$ y todo $T \subseteq D(P)$ que satisface $C(T, P)$,
  si $P(T)$ es correcto, entonces $P$ es correcto
- Ojala $|T| << |D(P)|$ ... pero en general, $T = D(P)$ es el único criterio de selección que cumple la definición de ideal

# Efectividad vs correctitud

Diremos que un criterio de selección $C$ es efectivo si para cualquier programa $P$ y todo $T \subseteq D(P)$ que satisface $C(T, P)$:

- si $P(T)$ es correcto, entonces tenemos un alto nivel de confianza de que $P$ es correcto . . . o
- si $P(T)$ es correcto, podemos garantizar (o es muy probable) que $P$ no tenga fallas de cierto tipo

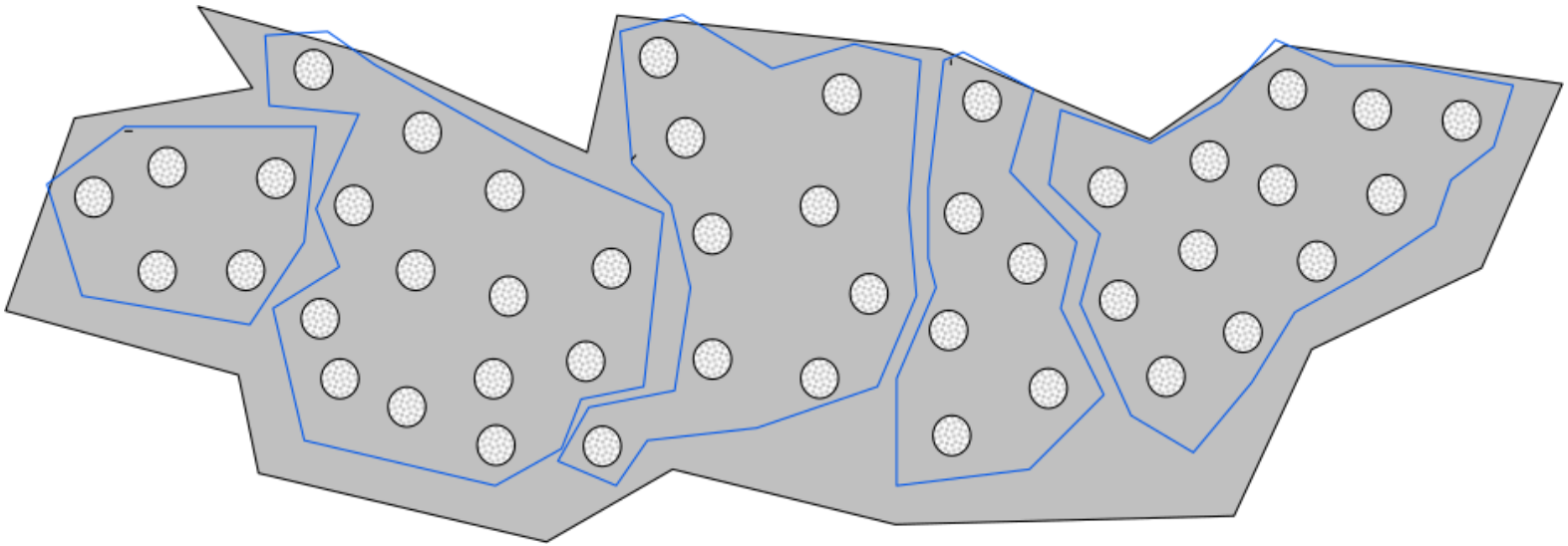
En realidad, no podemos hacer esto muy bien en la practica . . .

# Criterios de selección: caja negra

- Casos típicos
- Condiciones/valores de borde
- Condiciones excepcionales
- Condiciones ilegales (para revisar robustez)
- Casos que podrían revelar fallas
  - basado en la intuición de que podría "quebrar" el sistema
- Casos especiales

# Particionar el dominio de entrada

Para cada partición, solo agregar un elemento a $T$



# Ejemplo

Tenemos una función que calcula el factorial de $n$

- Si $n$ es menor que 0, o mayor que 200, entonces debe imprimir un mensaje de error
- Si $0 \leq n < 20$, entonces debe retornar el valor exacto de $n!$
- Si $20 \leq n \leq 200$, entonces debe retornar un valor aproximado de $n!$, con un máximo error de 0,1%

# Ejemplo

Tenemos una función que calcula el factorial de $n$

- Si $n$ es menor que 0, o mayor que 200, entonces debe imprimir un mensaje de error
- Si $0 \leq n < 20$, entonces debe retornar el valor exacto de $n!$
- Si $20 \leq n \leq 200$, entonces debe retornar un valor aproximado de $n!$, con un máximo error de 0,1%

Posibles clases de equivalencia del dominio de entrada:

- $D_1 = \{n < 0\}, D_2 = \{0 \leq n < 20\}, D_3 = \{20 \leq n \leq 200\}$, $D_4 = \{n > 200\}$
- Seleccionamos solo un caso de prueba por cada clase de equivalencia

# Clases de equivalencia

Si podemos identificar clases de equivalencia, entonces definen una partición del dominio de entrada

- ¿y si no tenemos clases de equivalencia?

# Clases de equivalencia

Si podemos identificar clases de equivalencia, entonces definen una partición del dominio de entrada

- ¿y si no tenemos clases de equivalencia?
- podemos tratar de minimizar el numero de casos de prueba al elegir los casos de prueba
- ejemplo:
  $D_1 = \{x \text{ es par}\}, D_2 = \{x \text{ es impar}\}, D_3 = \{x \leq 0\}, D_4 = \{x > 0\}$

# Clases de equivalencia

Si podemos identificar clases de equivalencia, entonces definen una partición del dominio de entrada

- ¿y si no tenemos clases de equivalencia?
- podemos tratar de minimizar el numero de casos de prueba al elegir los casos de prueba
- ejemplo:
  $D_1 = \{x \text{ es par}\}, D_2 = \{x \text{ es impar}\}, D_3 = \{x \leq 0\}, D_4 = \{x > 0\}$
- . . . el conjunto de pruebas $\{x = 48, x = -23\}$ es suficiente para testear este dominio

# Casos extremos

Aun cuando tengamos clases de equivalencias, hay algunos casos extremos:

- cada clase de equivalencia tiene un solo elemento $\rightarrow$ testing exhaustivo
- el dominio de entrada es su propia clase de equivalencia . . . solo tenemos un caso de prueba

# Mas criterios para seleccionar casos de prueba

- Para cada rango de valores $[R_1, R_2]$ de entrada o salida, seleccionar cinco casos de prueba:
  - algún $x_1$ donde $x_1 < R_1$
  - $x_2 = R_1$
  - algún $x_3$ donde $R_1 < x_3 < R_2$
  - $x_4 = R_2$
  - algún $x_5$ donde $x_5 > R_2$
- Para cada (conjunto | lista) S, seleccionar dos casos de prueba:
  - $y_1 \in S$
  - $y_2 \notin S$
- Para cada test de igualdad, seleccionar dos valores tal que:
  - son iguales
  - no son iguales

# Testing de condiciones de borde

Para el ejemplo de la función que calcula el factorial, los rangos de la variable $n$ de entrada son:

- $[-\infty, 0], [0, 20], [20, 200], [200, \infty]$
- Un posible conjunto de pruebas:
  $\{n = -5, n = 0, n = 11, n = 20, n = 25, n = 200, n = 3000\}$
- Si conocemos el valor mínimo y máximo de $n$, podemos agregarlos al conjunto de pruebas

# Caja blanca

Algunos criterios de cobertura de código

- Cobertura de lineas de código (statement coverage)
- Cobertura de condicionales (branch coverage)
- Condiciones escondidas
- Cobertura de caminos (path coverage)
- Cobertura de loops

Esto queda para la siguiente clase :-)