



Pontificia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación

# Clase 3

# Introducción: Conceptos

**IIC3745 – Testing**

Rodrigo Saffie

[rasaffie@uc.cl](mailto:rasaffie@uc.cl)

19 de agosto de 2020

# Encuestas

Jueves 20 de agosto:

- Conocimientos generales
- Grupos proyectos semestral

## 1. Introducción

- Motivación
- Conceptos
- Tipos de *tests*

## 2. Criterios de cobertura

# ¿Por qué falla el software?

En los requisitos:

- Faltan requisitos
- Requisitos mal definidos
- Requisitos no realizables
- Diseño de software defectuoso

En la implementación:

- Algoritmos incorrectos
- Implementación defectuosa

# Errores, Defectos, Fallas

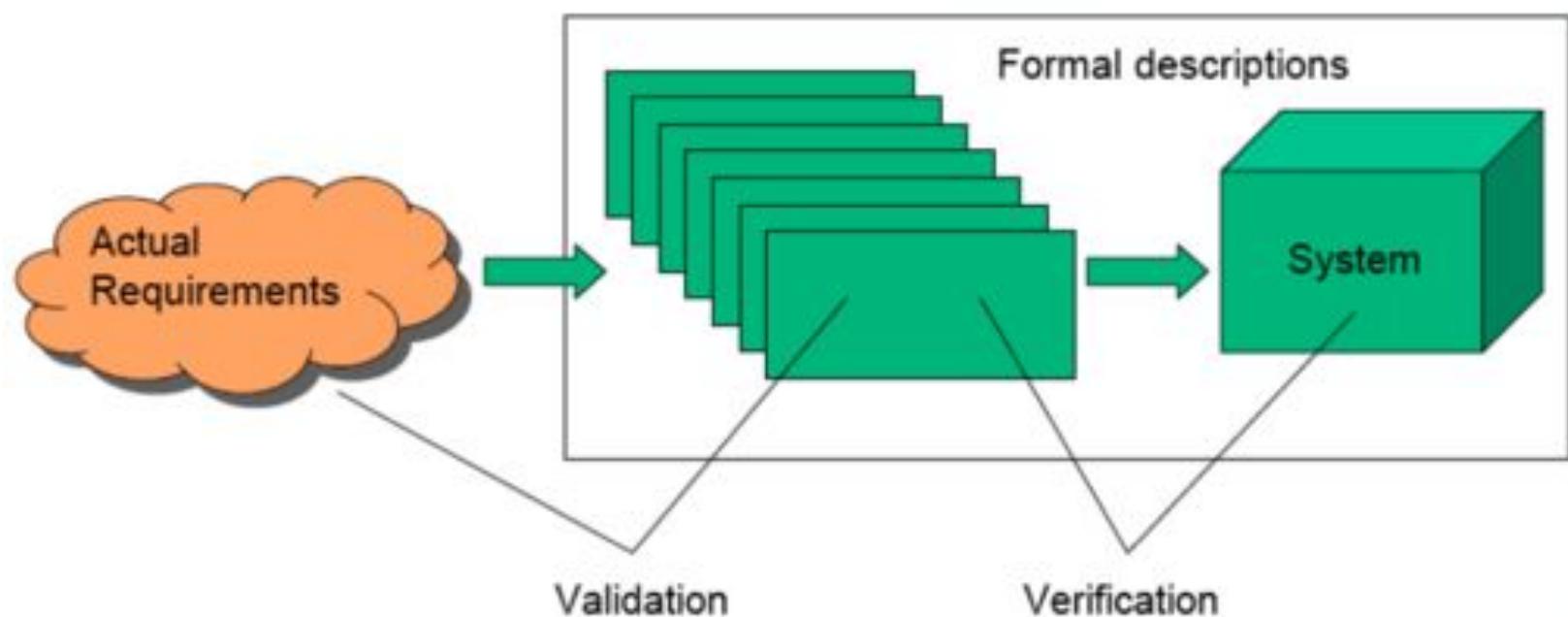
- **Error:** acción humana que produce un resultado incorrecto.
- **Defecto:** presencia de una imperfección que puede generar fallas.
- **Falla:** comportamiento observable incorrecto con respecto a los requisitos.

Un **error** introduce un **defecto** en el software que se manifiesta a través de una **falla** en las pruebas.

# ¿Qué es un *bug*?

- Concepto que se utiliza informalmente para representar un defecto, un error y/o una falla
- Hace alusión a que “algo no anda bien”
- Origen:
  - ~1880: [Thomas Edison](#) para referirse a problemas de diseño
  - 1947: Reportado por [Grace Hopper](#) en proyecto Mark I de la universidad de Harvard

# Validación y Verificación



# Hacia ausencia de defectos

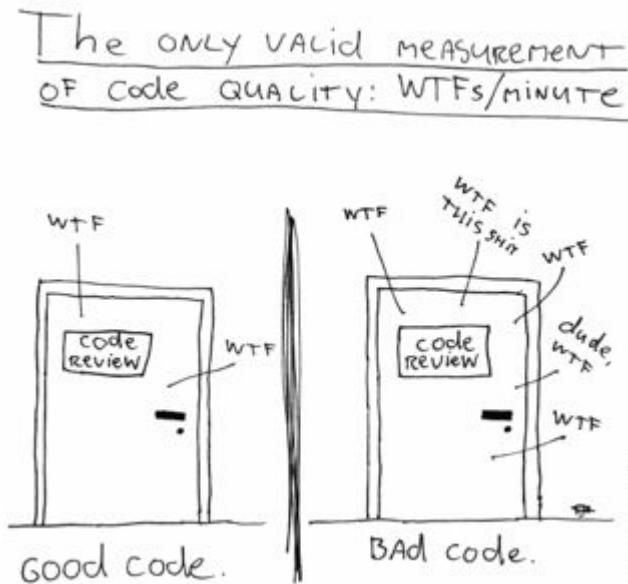
- No incorporarlos al construir software (muy difícil)
- Análisis estático
- Inspección formal de código
- *Testing*

# Análisis estático

- Análisis del código sin ejecutarlo, mediante herramientas que permiten detectar elementos sospechosos
  - Ejemplos: [rubocop](#), [brakeman](#), [reek](#), [flay](#), [bundler-audit](#), [dependabot](#)
- Se detecta gran cantidad de falsos positivos
- No se pueden detectar defectos importantes

# Inspección formal de código

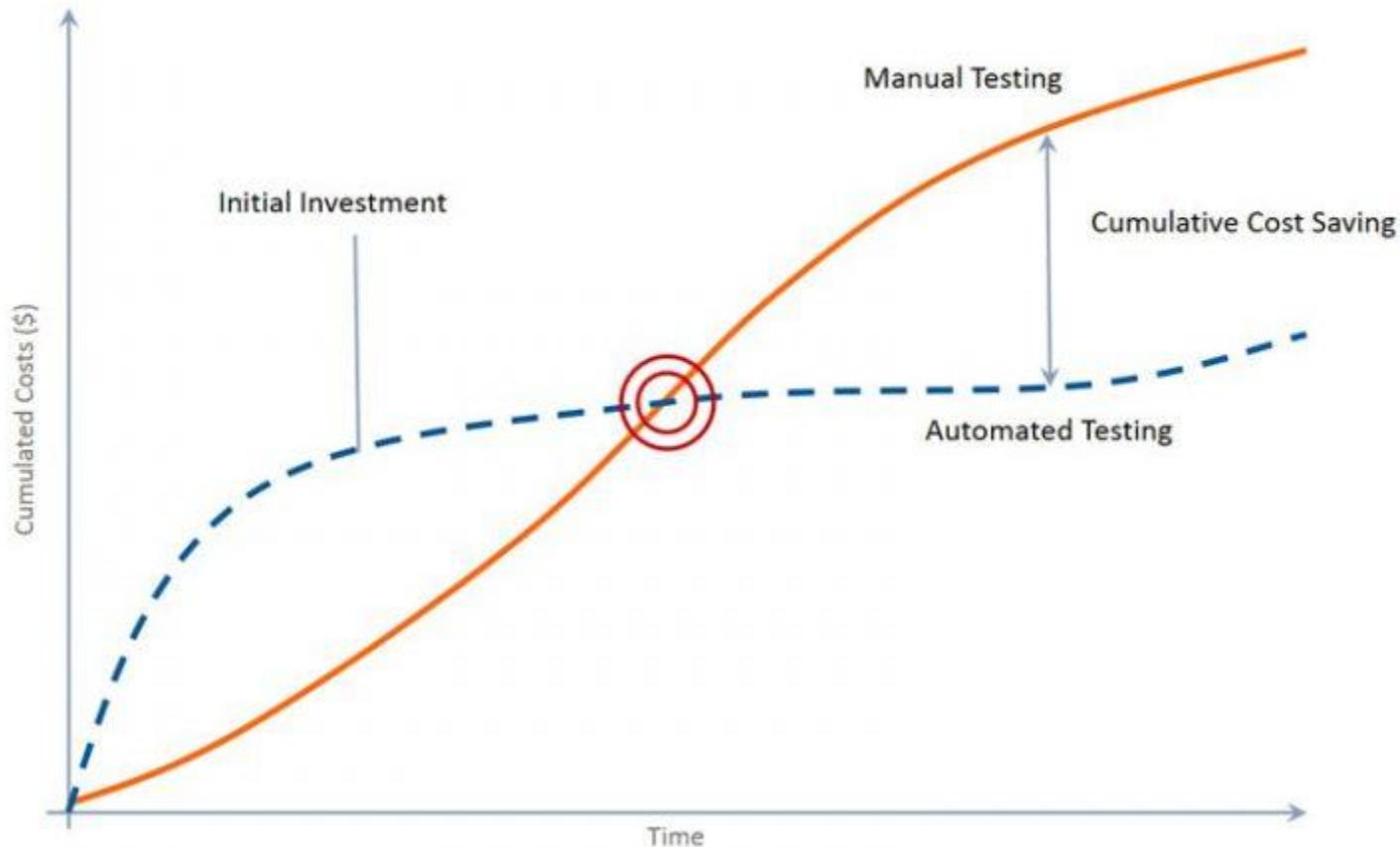
- Código es revisado por equipo de pares con el objetivo de detectar la mayor cantidad de defectos
- Forma efectiva para producir código de calidad
- Ejemplos: *Pull Requests* o sesiones de inspección



# Testing

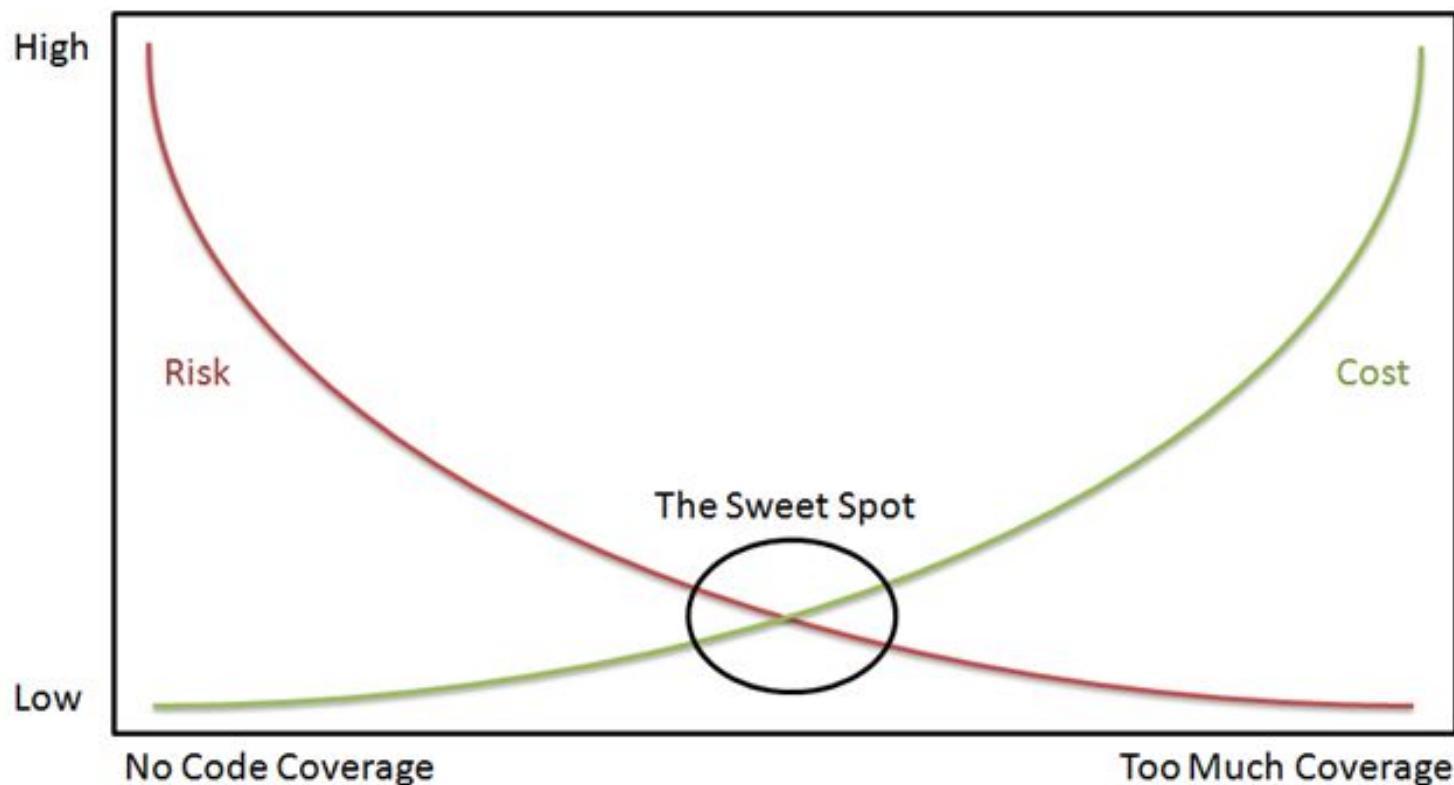
- Proceso de ejecutar un programa con el objetivo de encontrar un error
- Se diseñan casos de prueba y se somete el *software* a ellas
- Un buen caso de prueba es uno con una alta probabilidad de encontrar un error oculto
- No necesariamente es automatizado: *debugging* o uso de plantillas

# Testing: manual vs automatizado



# Costo del *Testing*

- ¿Se debe *testear* todo el *software*?

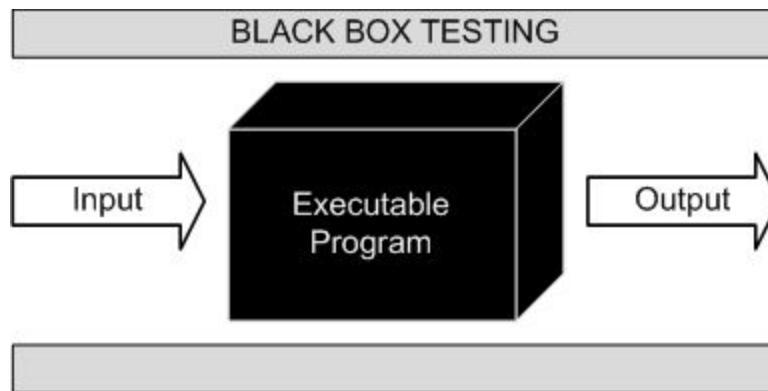


# ¿Por qué hacer pruebas?

- Mejorar calidad / disminuir riesgo de defectos
- Reducir costos a causa de errores no detectados
- Garantizar nivel de servicio
- Generar confianza en el *software*

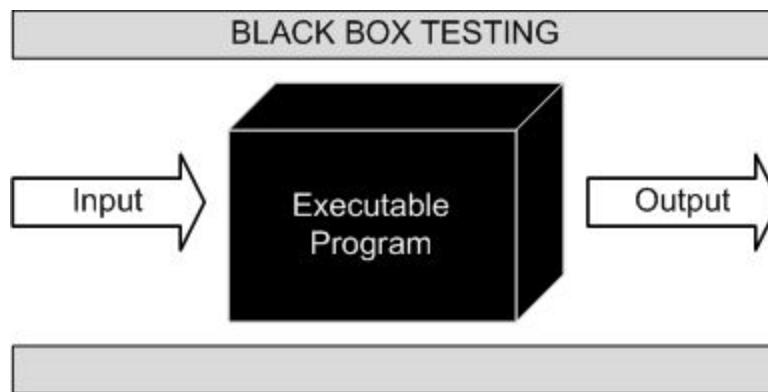
# *Black-box testing*

- No se conocen los detalles del *software*, solamente se considera *input* y *output*.



# Black-box testing

- No se conocen los detalles del *software*, solamente se considera *input* y *output*.



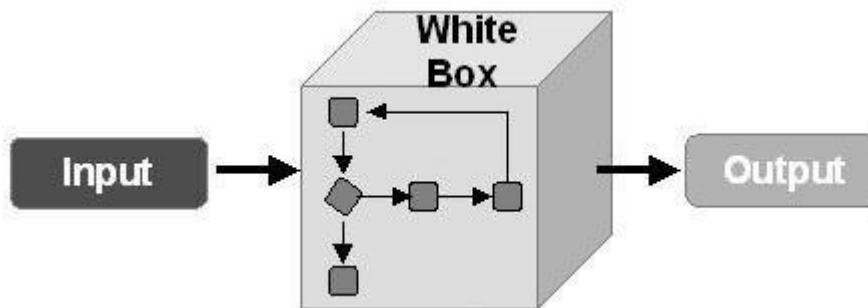
- Ejemplos
  - Pruebas de usuarios
  - Pruebas de seguridad

# *Black-box testing*

- No se conocen los detalles del *software*, solamente se considera *input* y *output*.
- Ventajas:
  - El tester no necesariamente debe saber programar o haber participado en el desarrollo
  - Se pueden diseñar junto con el levantamiento de requisitos
- Desventajas:
  - Se omiten muchos casos de pruebas
  - Alta probabilidad de pruebas redundantes

# *White-box testing*

- Se conocen los detalles del software y se puede utilizar al diseñar los *tests*



# *White-box testing*

- Se conocen los detalles del software y se puede utilizar al diseñar los *tests*
- Ventajas:
  - Los casos de pruebas abordados son más completos
- Desventajas:
  - Son más costosos de diseñar y mantener

# *Black-box vs White-box*

- *Black-box*

```
// Retorna la cantidad de ceros contenidos en un arreglo  
function countZeros(array) {
```

# *Black-box vs White-box*

- *Black-box*

```
// Retorna la cantidad de ceros contenidos en un arreglo  
function countZeros(array) {
```

- *White-box*

# *Black-box vs White-box*

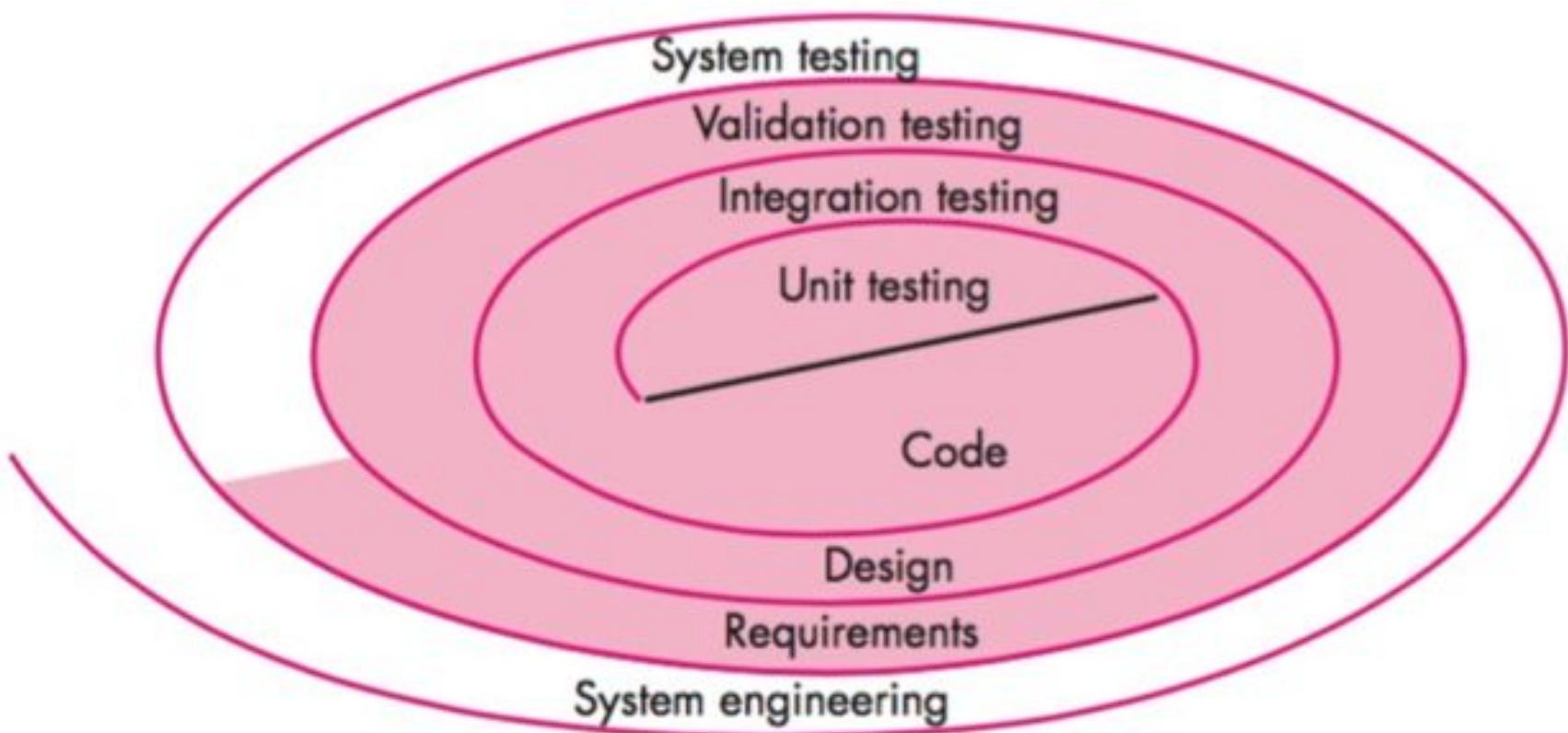
- *Black-box*

```
// Retorna la cantidad de ceros contenidos en un arreglo
function countZeros(array) {
```

- *White-box*

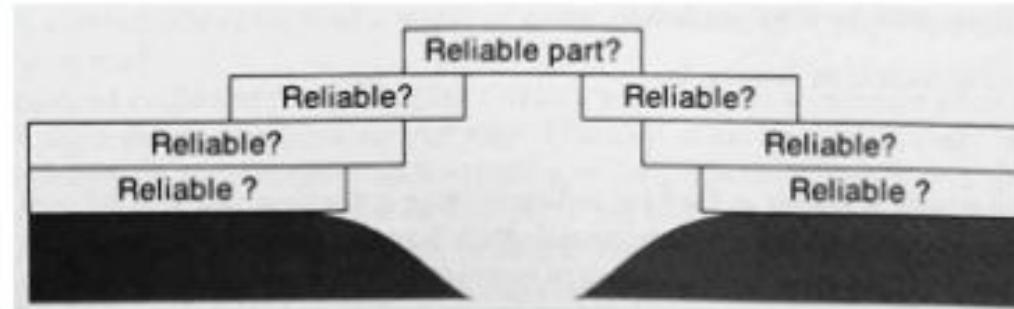
```
// Retorna la cantidad de ceros contenidos en un arreglo
function countZeros(array) {
    var count = 0;
    for (var i = 1; i < array.length; i++) {
        if (array[i] === 0) {
            count++;
        }
    }
    return count;
}
```

# Niveles tradicionales de *Testing*



# Tests unitarios

- Se centran en verificar las unidades más pequeñas del *software* (componentes y módulos)
- Se pueden realizar antes, durante o después de la codificación
- Se debe tener un control de los resultados esperados (*inputs* y *outputs*)



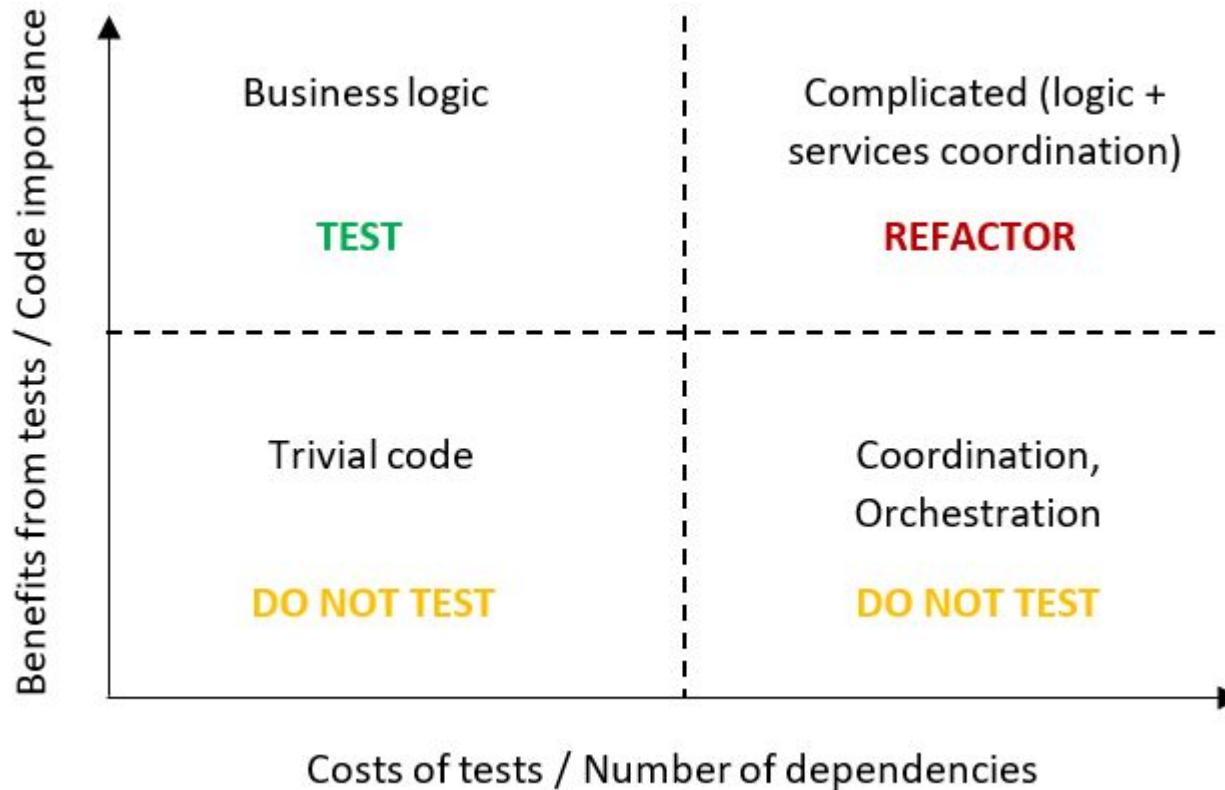
# *Tests unitarios: Beneficios*

- Permiten hacer cambios al código de manera segura
- Ayudan a entender el diseño y funcionalidades a desarrollar
- Sirven como apoyo a la documentación (son ejemplos de uso)

# *Tests unitarios: Costos*

- Consumen más tiempo en el corto plazo
  - Diseñarlos
  - Implementarlos
  - Mantenerlos
- No todas las pruebas agregan el mismo valor
- No representan ni garantizan la calidad del *software*

# Tests unitarios: Costos



# Tests de integración

Cada uno de los componentes puede tener pruebas unitarias, pero aún así el sistema necesita pruebas



# Tests de aceptación

- Validación efectuada por usuarios con el objetivo de aceptar o rechazar el producto desde una mirada funcional



# Tests de sistemas

- Pruebas de la aplicación en su ambiente de ejecución
- Validan los requisitos no funcionales del sistema:
  - Rendimiento: que cumpla con los requerimientos de desempeño
  - Carga: cómo se comporta el sistema bajo ciertas condiciones de uso
  - Stress: cómo responde el sistema bajo una carga mayor para la cual fue diseñado
  - Seguridad: ataques simulados para encontrar fortalezas y debilidades

# Tests de humo

- Subconjunto de pruebas enfocadas en garantizar las funcionalidades más importantes
- Se ejecutan de manera rápida y barata antes de cada implementación



# Tests de mutación

- Se introducen pequeñas variaciones en el código con el objetivo de cuantificar cuántos *tests* fallan
- Evalúan la calidad de los *tests* existentes

```
# Original
if x > y:
    z = x - y
else:
    z = 2 * x
```

```
# Mutación 1
if x >= y:
    z = x - y
else:
    z = 2 * x
```

```
# Mutación 2
if x > y:
    z = x + y
else:
    z = 2 * x
```

```
# Mutación 3
if x > y:
    z = x - y
else:
    z = 2 * y
```

# Tests de regresión

- *Tests* para garantizar que luego de modificar un software las funcionalidades originales siguen respetando las especificaciones
- Pueden ser un subconjunto de las pruebas o la batería completa
- Pueden ser útiles al momento de versionar código según el criterio [SemVer](#)

# Criterios de cobertura de pruebas

- Las pruebas son caras y consumen esfuerzo
- Los criterios de cobertura sirven para decidir qué entradas de prueba usar
- Los criterios hacen que las pruebas sean más eficientes y efectivas:
  - Es más probable encontrar problemas
  - Generan mayor confianza en la calidad del código
  - Se responde al por qué de cada prueba

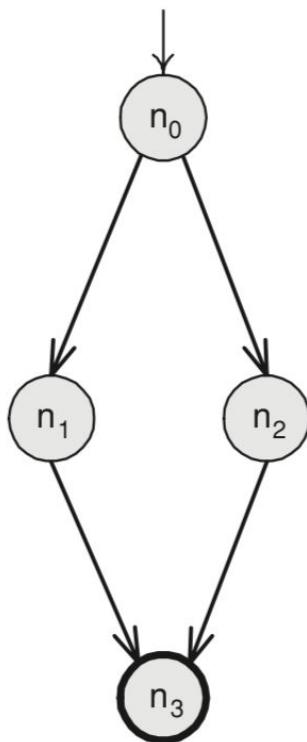
# Criterios de cobertura de pruebas

- Basados en grafos
- Expresiones lógicas
- Dominio de parámetros de entrada
- Estructuras sintácticas

# Grafo

- $\mathbf{N}$ : Set de nodos
- $\mathbf{N}_0$ : Set de nodos iniciales, subconjunto de  $\mathbf{N}$
- $\mathbf{N}_f$ : Set de nodos finales, subconjunto de  $\mathbf{N}$
- $\mathbf{E}$ : Set de aristas, subconjunto de  $\mathbf{N} \times \mathbf{N}$

# Grafo: Ejemplos

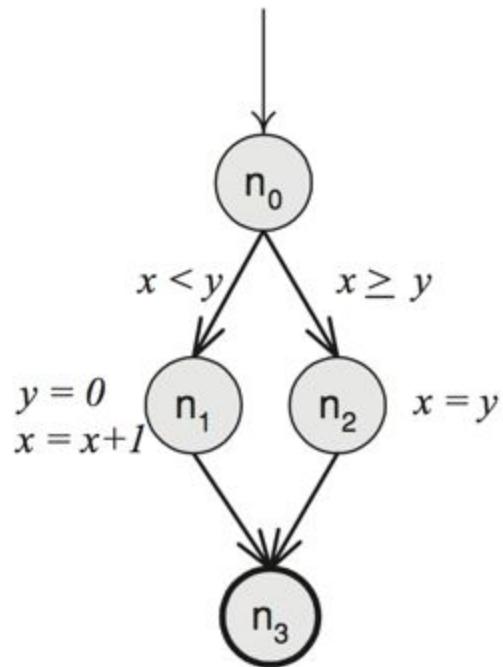


$$N = \{ n_0, n_1, n_2, n_3 \}$$

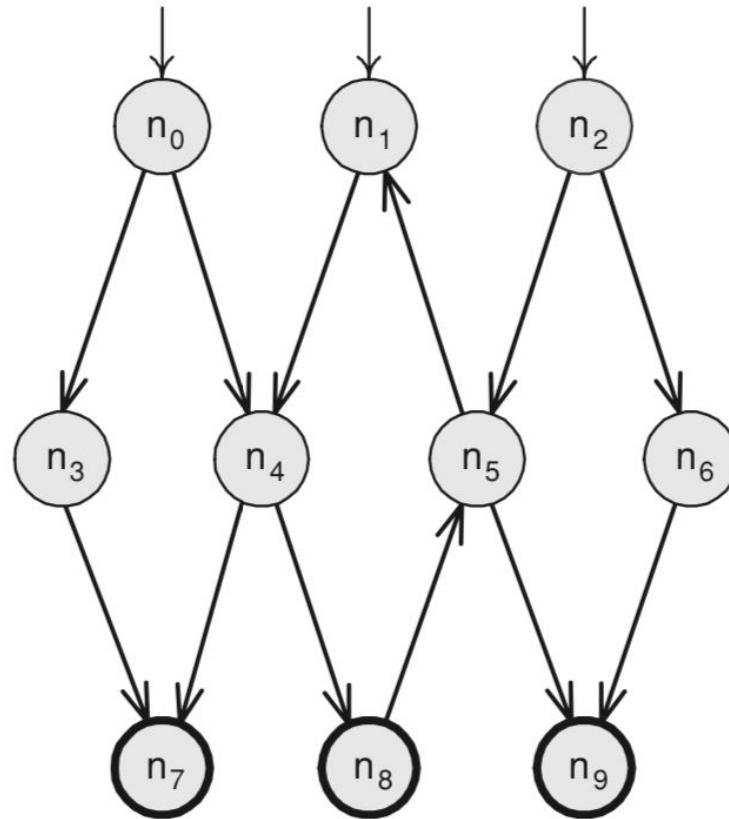
$$N_0 = \{ n_0 \}$$

$$E = \{ (n_0, n_1), (n_0, n_2), (n_1, n_3), (n_2, n_3) \}$$

```
if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}
```



# Grafo: Ejemplos



$$N = \{ n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9 \}$$

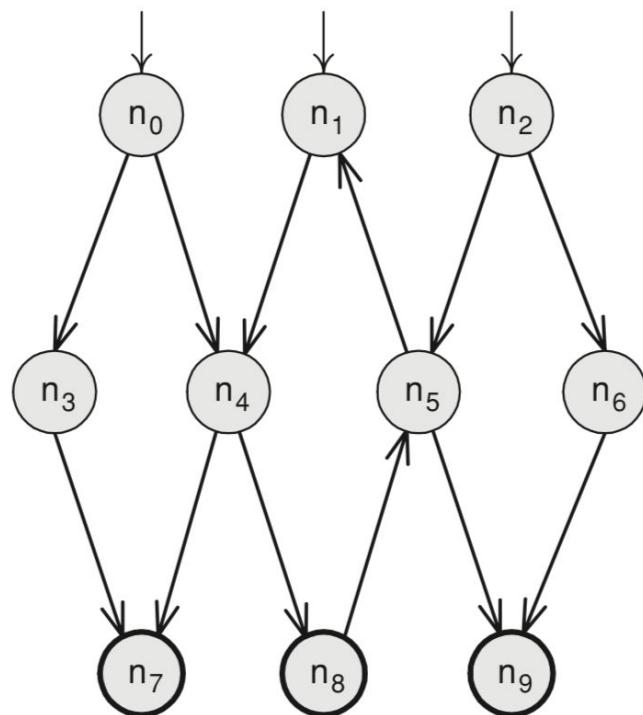
$$N_0 = \{ n_0, n_1, n_2 \}$$

$$|E| = 12$$

# Grafo: Definiciones

Camino:

- secuencia de nodos donde cada par de nodos adyacentes ( $n_i, n_{i+1}$ ) está contenido en el set **E** de aristas.



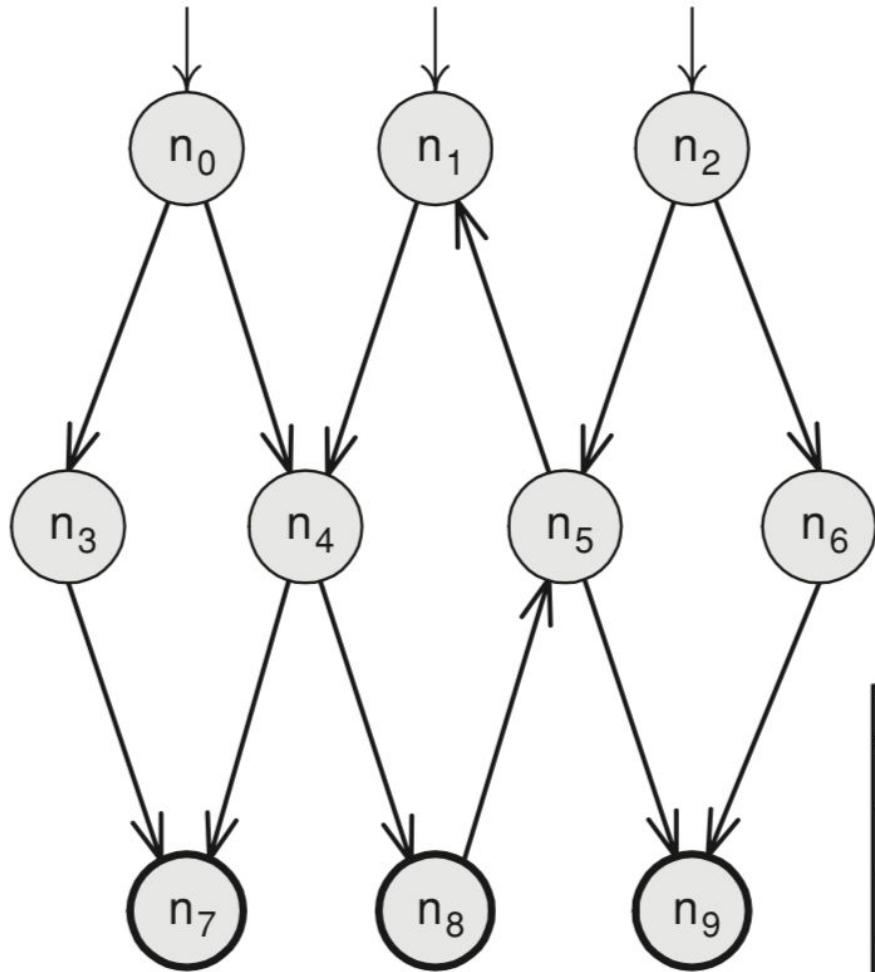
Path Examples	
1	$n_0, n_3, n_7$
2	$n_1, n_4, n_8, n_5, n_1$
3	$n_2, n_6, n_9$

Invalid Path Examples	
1	$n_0, n_7$
2	$n_3, n_4$
3	$n_2, n_6, n_8$

# Grafo: Definiciones

- *Camino*: secuencia de nodos donde cada par de nodos adyacentes ( $n_i, n_{i+1}$ ) está contenido en el set  $E$  de aristas.
- *Sub-camino*: subsecuencia de nodos de un camino  $p$ .  
 $[n_0, n_3]$  es subcamino de  $[n_0, n_3, n_7]$
- *Largo de camino*: cantidad de aristas contenidas en el camino.

# Alcance Sintáctico y Semántico



¿Es  $n_2$  alcanzable desde  $n_0$ ?

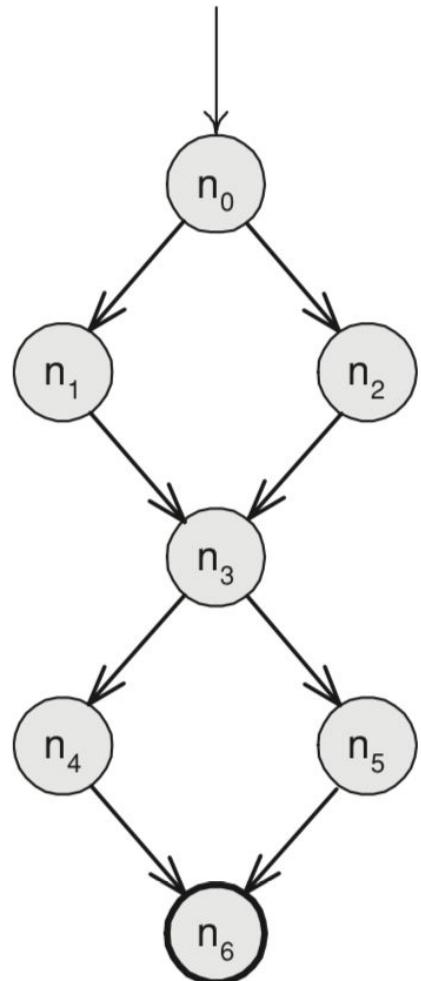
¿Es  $n_7$  alcanzable desde  $n_1$ ?

**Sintáctico:** Depende de la estructura del grafo.

**Semántico:** Depende la semántica del software.

Reachability Examples	
1	$\text{reach}(n_0) = N - \{n_2, n_6\}$
2	$\text{reach}(n_0, n_1, n_2) = N$
3	$\text{reach}(n_4) = \{n_1, n_4, n_5, n_7, n_8, n_9\}$
4	$\text{reach}([n_6, n_9]) = \{n_9\}$

# Grafo Single Entry Single Exit (SESE)



- $|N_0| = 1$
- $|N_f| = 1$
- $\text{reach}(n_0) = G$
- $\text{reach}(n_f) = [n_f]$



Pontifícia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación

# Clase 3

# Introducción: Conceptos

**IIC3745 – Testing**

Rodrigo Saffie

[rasaffie@uc.cl](mailto:rasaffie@uc.cl)

19 de agosto de 2020