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

Clase 11
Pruebas Unitarias

IIC3745 – Testing

Rodrigo Saffie

rasaffie@uc.cl

30 de septiembre de 2020

1. Cobertura en base a lógica
2. Pruebas unitarias

ICC: Cobertura de cláusula inactiva *(Inactive Clause Coverage)*

Por cada $p \in P$ y cada cláusula mayor $c_i \in C_p$, escoja las cláusulas menores c_j con $i \neq j$ de modo que c_i no determina p. ***TR*** contiene cuatro requisitos por cada c_i:

1. c_i se evalúa como verdadero con p verdadero
2. c_i se evalúa como falso con p verdadero
3. c_i se evalúa como verdadero con p falso
4. c_i se evalúa como falso con p falso

Estos cuatro requisitos permiten demostrar que c_i no tiene incidencia alguna sobre p.

GICC: Cobertura de cláusula inactiva general *(General Inactive Clause Coverage)*

Por cada $p \in P$ y cada cláusula mayor $c_i \in C_p$, escoja las cláusulas menores c_j con $i \neq j$ de modo que c_i no determina p. ***TR*** contiene cuatro requisitos por cada c_i:

1. c_i se evalúa como verdadero con p verdadero
2. c_i se evalúa como falso con p verdadero
3. c_i se evalúa como verdadero con p falso
4. c_i se evalúa como falso con p falso

Los valores de las cláusulas menores c_j no necesitan ser los mismos cuando c_i es verdadero y cuando c_i es falso.

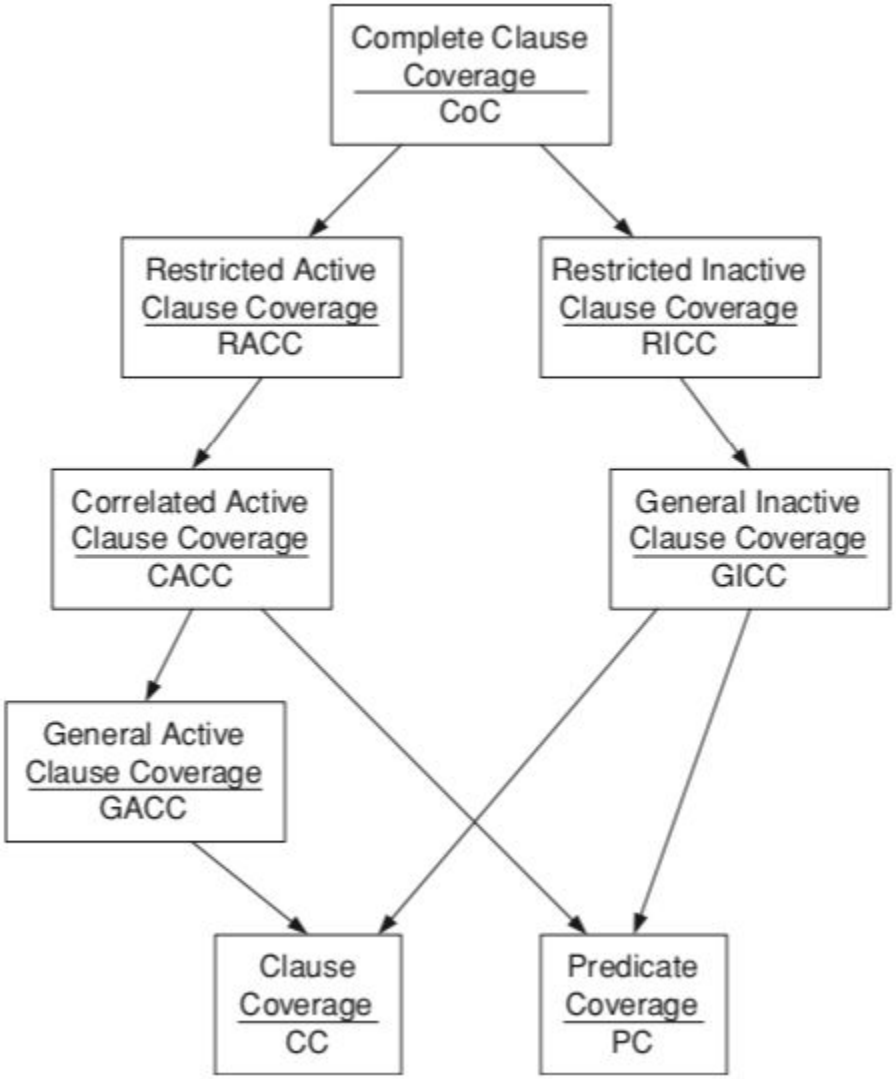
RICC: Cobertura de cláusula inactiva restrictiva *(Restrictive Inactive Clause Coverage)*

Por cada $p \in P$ y cada cláusula mayor $c_i \in C_p$, escoja las cláusulas menores c_j con $i \neq j$ de modo que c_i no determina p. ***TR*** contiene cuatro requisitos por cada c_i:

1. c_i se evalúa como verdadero con p verdadero
2. c_i se evalúa como falso con p verdadero
3. c_i se evalúa como verdadero con p falso
4. c_i se evalúa como falso con p falso

Los valores de las cláusulas menores c_j deben ser los mismos para (1) y (2) / (3) y (4).

Subsumición cobertura lógica



Infactibilidad

- En la práctica existen varias complicaciones para aplicar estos criterios.
- Generalmente aparecen combinaciones de valores imposibles dado que las cláusulas están relacionadas.

```
while (i < n && a[i] != 0) {do something to a[i]}
```

- Por esta razón se busca satisfacer únicamente los requisitos de pruebas factibles.
- Además, se priorizan criterios con la mayor cantidad de opciones posibles (*CACC* sobre *RACC*).

Infactibilidad

$$(a > b \wedge b > c) \vee c > a$$

- No es factible que:
  - $a > b = true$
  - $b > c = true$
  - $c > a = true$
- Los requisitos de pruebas que no son factibles deben ser **identificados** e **ignorados**.

Definición cláusulas activas

- En predicados simples es fácil encontrar valores para cláusulas menores.
- Para encontrar los valores de cláusulas menores que definen una cláusula mayor se debe resolver:

$$p_c = p_{c=true} \oplus p_{c=false}$$

- Luego de simplificar p_c describe exactamente los valores necesarios para que ***c*** determine a ***p***.
- Asimismo, ¬***p*** describe los valores necesarios para que ***c*** no determine a ***p***.

Evaluación disyunción exclusiva

$$p \oplus q$$

$$= (p \vee q) \wedge \neg(p \wedge q)$$

$$= (p \wedge \neg q) \vee (\neg p \wedge q)$$

Ejemplos

$p = a \vee b$

$p_a = p_{a=true} \oplus p_{a=false}$

$= (true \vee b) \oplus (false \vee b)$

$= true \oplus b$

$= \neg b$

$p = a \wedge b$

$p_a = p_{a=true} \oplus p_{a=false}$

$= (true \wedge b) \oplus (false \wedge b)$

$= b \oplus false$

$= b$

$p = a \vee (b \wedge c)$

$p_a = p_{a=true} \oplus p_{a=false}$

$= (true \vee (b \wedge c)) \oplus (false \vee (b \wedge c))$

$= true \oplus (b \wedge c)$

$= \neg(b \wedge c)$

$= \neg b \vee \neg c$

Variables repetidas

$$(a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$$

- Si bien hay 6 cláusulas, solamente son 3 únicas
- Existen 8 pruebas posibles (no 64)

- Conviene probar predicados simples
  - Se evitan casos de pruebas redundantes

Variables repetidas

$$p = a \wedge b \vee a \wedge \neg b$$

$p_a = p_{a=true} \oplus p_{a=false}$

$= true \wedge b \vee true \wedge \neg b \oplus false \wedge b \vee false \wedge \neg b$

$= b \vee \neg b \oplus false$

$= true \oplus false$

$= true$

$p_b = p_{b=true} \oplus p_{b=false}$

$= a \wedge true \vee a \wedge \neg true \oplus a \wedge false \vee a \wedge \neg false$

$= a \vee false \oplus false \vee a$

$= false$

Variables repetidas

$$p = a \wedge b \vee a \wedge \neg b$$

- ***a*** siempre determina a ***p***
- ***b*** nunca determina a ***p***

$$p = a$$

- Error conceptual que se debe detectar al momento de diseñar pruebas

Aplicación en artefactos de *software*

- Código fuente
- Especificación de requisitos
- Máquinas de estados
- Forma Normal Disyuntiva (***DNF***)

Código fuente

```
if (a && b)
   S1;
else
   S2;
```

```
if (a)
{
   if (b)
      S1;
   else
      S2;
}
else
   S2;
```

Código fuente

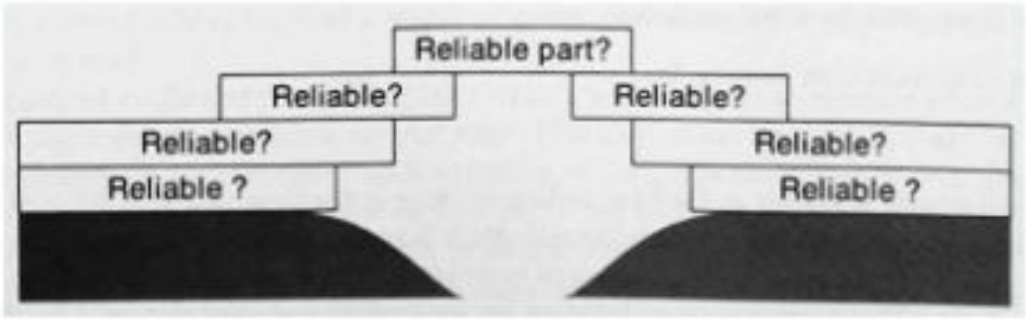
```
if ((a && b) || c)
   S1;
else
   S2;
```

```
if (a)
   if (b)
      S1;
   else
      if (c)
         S1;
      else
         S2;
else
   if (c)
      S1;
   else
      S2;
```

1. Cobertura en base a lógica
2. Pruebas unitarias

Pruebas unitarias

- 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 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*

Mocks y *Stubs*

Mocks:
Son imitaciones de objetos, de las cuales se espera que ciertos métodos sean invocados durante un *test*. De no ser así el *test* falla.

Stubs:
Son imitaciones de objetos que proveen resultados predefinidos para ciertas invocaciones sobre ellos. Son útiles para probar de forma aislada los componentes.

También existen conceptos similares:

- *doubles*, *dummies*, *fakes*

Test unitarios: estructura

1. Nombre del *test*
2. *Setup* general de las pruebas
3. *Setup* particular de un *test*
4. Ejecución del método a probar
5. Validación de resultados a través de *asserts*
6. *Teardown*

Coverage aplicado

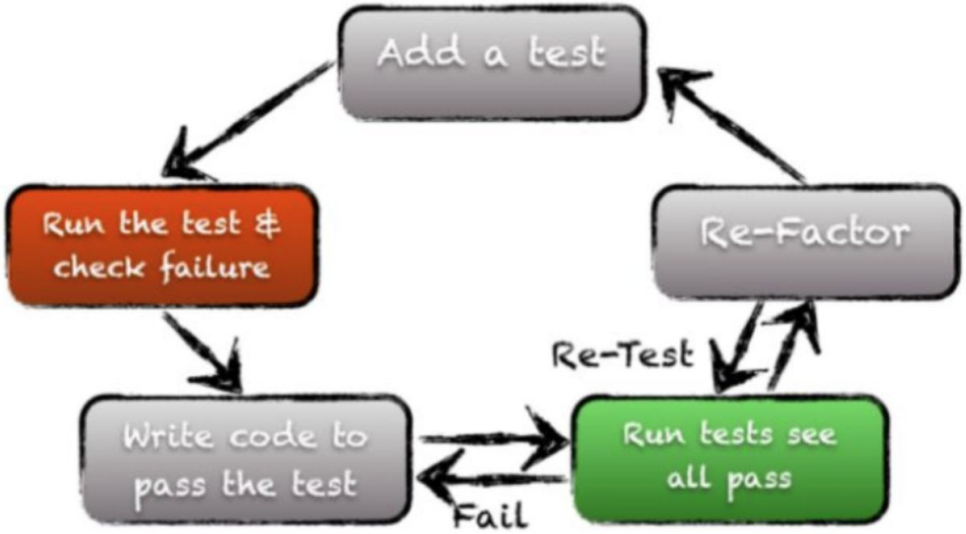
Distintos criterios de medición computables:

- ***Function coverage***: proporción de métodos que son invocados.
- ***Statement coverage***: proporción de instrucciones ejecutadas.
- ***Branch coverage***: proporción de caminos independientes recorridos.
- ***Condition coverage***: proporción de predicados/cláusulas probados.

Test Driven Development

Metodología basada en desarrollar código en pequeños ciclos iterativos que incluyen:

- Diseñar *tests* para un requerimiento
- Desarrollar código hasta que los *tests* pasen
- Mejorar implementación



Test Driven Development

Beneficios:

- Garantiza que toda línea de código tenga *tests* asociados.
- Implica un análisis del diseño del código al momento de crear los *tests*.

Desventajas:

- Genera muchos *tests* que quedan obsoletos rápidamente.
- Puede que no se justifique probar todo el código exhaustivamente.

Recomendaciones al *testear*

- Nombres descriptivos para *tests* y variables
  - Apoyo a la documentación y más fáciles de mantener
- Valores definidos explícitamente al momento de comparar *outcomes*
- Un objetivo claro y definido por *test*
  - Conocido como "un *assert*" (aunque no necesariamente)
- Evitar pruebas redundantes y operaciones costosas (como escribir en base de datos)
- No depender de condiciones externas al código
  - Por ejemplo: API externa, valores de *DateTime.now*
- Si un código es difícil de *testear* es mejor rediseñarlo

Lectura complementaria

- Martin Fowler - UnitTest

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

Clase 11
Pruebas Unitarias

IIC3745 – Testing

Rodrigo Saffie

rasaffie@uc.cl

30 de septiembre de 2020