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

Clase 5
Cobertura de grafos aplicada

IIC3745 – Testing

Rodrigo Saffie

rasaffie@uc.cl

28 de agosto de 2019

1. Clase pasada
    - Cobertura basada en grafos: flujo de información
2. Criterios de cobertura
    - Cobertura basada en grafos en software

Cobertura de grafos aplicada

- Código fuente
- Elementos de diseño
- Especificación de diseño
- Casos de uso

Cobertura para elementos del diseño

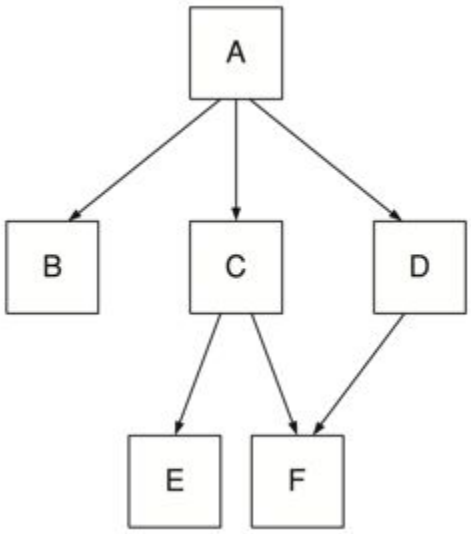
- Modularidad y reuso de componentes genera complejidad en las relaciones de estos
- Si bien esto permite probar componentes de manera independiente, también se debe probar las relaciones entre ellos

Cobertura para elementos del diseño Estructural

- Los grafos estructurales suelen representar el acoplamiento entre componentes
- No son muy útiles para encontrar defectos
- A través de *grafos de llamadas* se puede representar:
  - Dependencia entre modulos
  - Herencia y Polimorfismo

Grafo de llamadas

- **Nodos**: unidades de software (ej: métodos)
- **Aristas**: llamadas

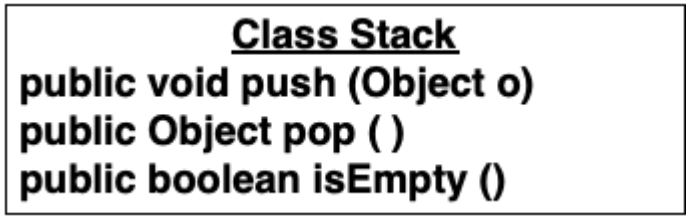


- **Cobertura de nodos**: visitar cada unidad al menos una vez
- **Cobertura de aristas**: ejecutar cada llamada al menos una vez

Grafo de llamadas: clases / módulos

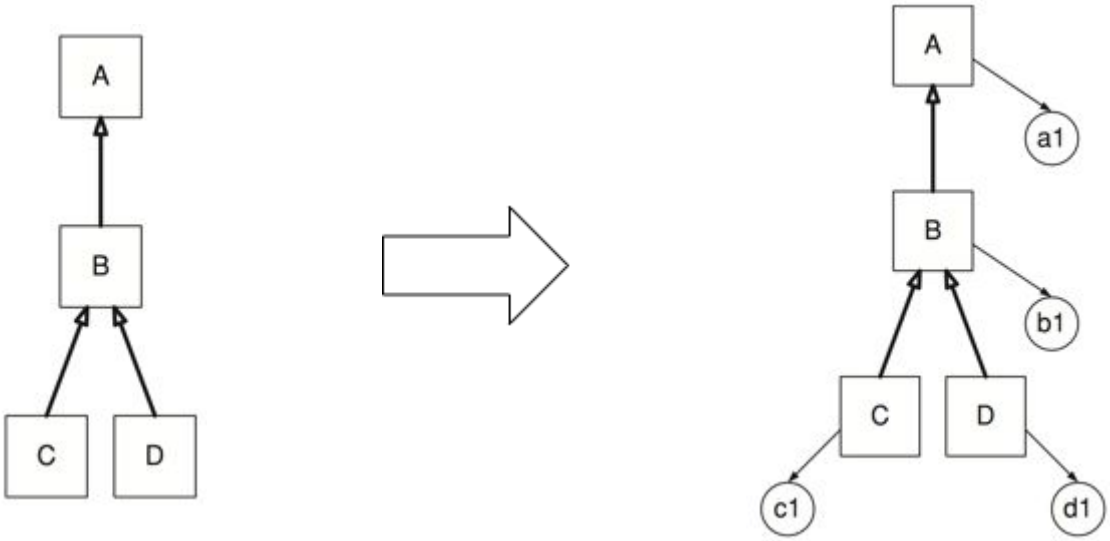
- Útil para representar relaciones de métodos dentro de una clase

- Generalmente entre clases puede no ser útil si existe bajo acoplamiento
  - Se generan grafos disconexos



Herencia y Polimorfismo

- No existe consenso en la mejor forma de probarlo estructuralmente
- Las clases no son *testeables*, por lo que se agregan nodos de instancias al grafo de herencia.



Herencia y Polimorfismo: Cobertura

- Cobertura de Nodos
  - Crear un objeto de cada clase
  - Poco confiable ya que no incluye ejecución
- Cobertura de Aristas
  - Cobertura agregada: cobertura de llamadas por lo menos para una instancia de cada clase en la jerarquía
  - Cobertura total: cobertura de llamadas para cada instancia de cada clase en la jerarquía

Cobertura para elementos del diseño
Flujo de datos

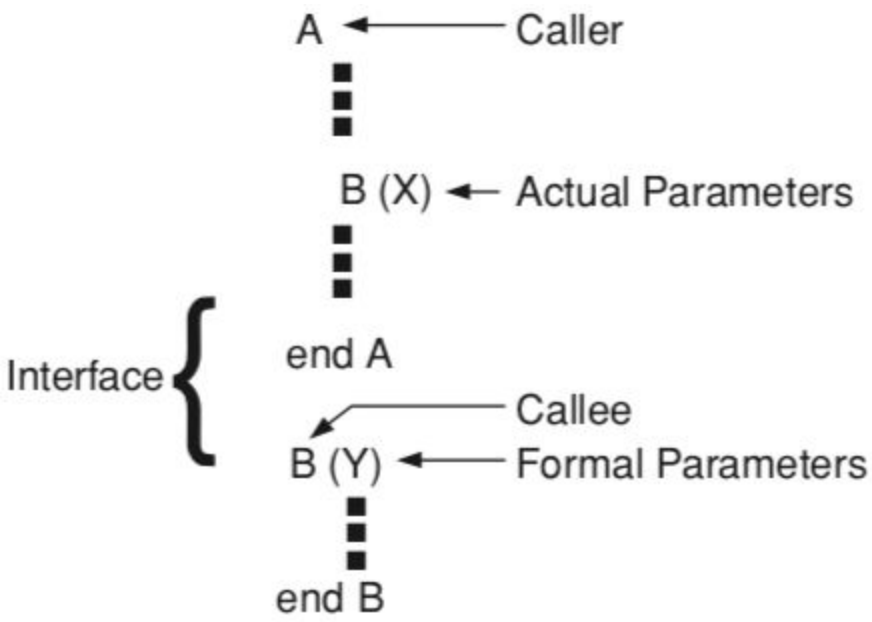
- Son grafos complejos y difíciles de analizar
  - Los parámetros pueden cambiar de nombre entre llamadas
  - Hay múltiples formas de compartir información
  - Encontrar *def* y *uses* es difícil

- Son útiles al momento de diseñar *tests* de integración

Cobertura para elementos del diseño
Flujo de datos

- Definiciones
  - ***Caller***: unidad que llama a otra
  - ***Callee***: unidad que es llamada
  - ***Call site***: lugar donde ocurre la llamada
  - ***Actual parameter***: valor en el *caller*
  - ***Formal parameter***: valor en el *callee*
  - ***Interface***: Mapeo de parámetro actual a formal

Cobertura para elementos del diseño
Flujo de datos



- Probar todo es muy costoso, pero probar la interfaz entrega un grado de seguridad razonable

Pares-DU entre unidades

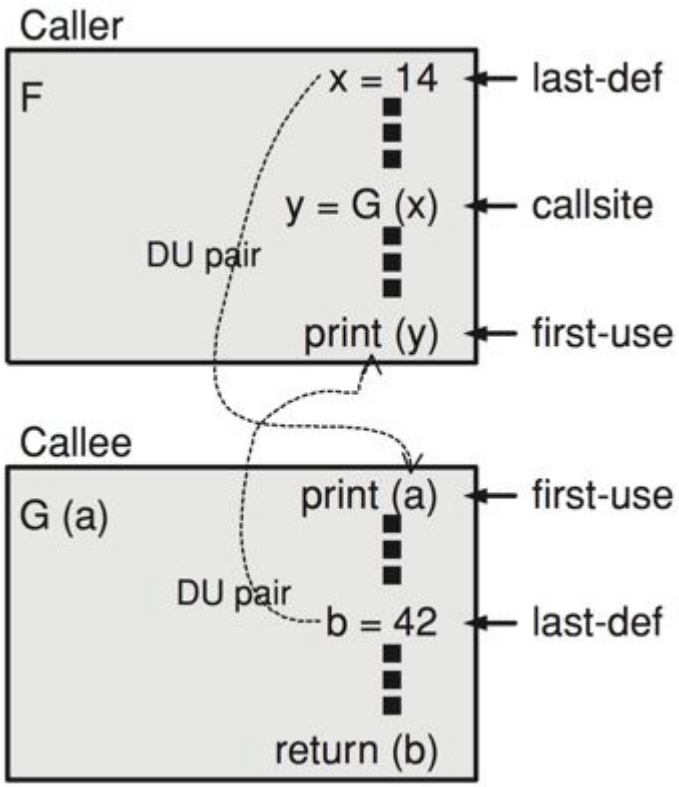
- Si nos enfocamos en la interfaz sólo debemos considerar
  - el último ***def*** antes de una llamada
  - el primer ***use*** luego de una llamada

- ***Last-def***: El conjunto de nodos que define una variable ***x*** y tiene un camino ***def-clear*** desde el nodo a través de un *call site* a un ***use*** en otra unidad.

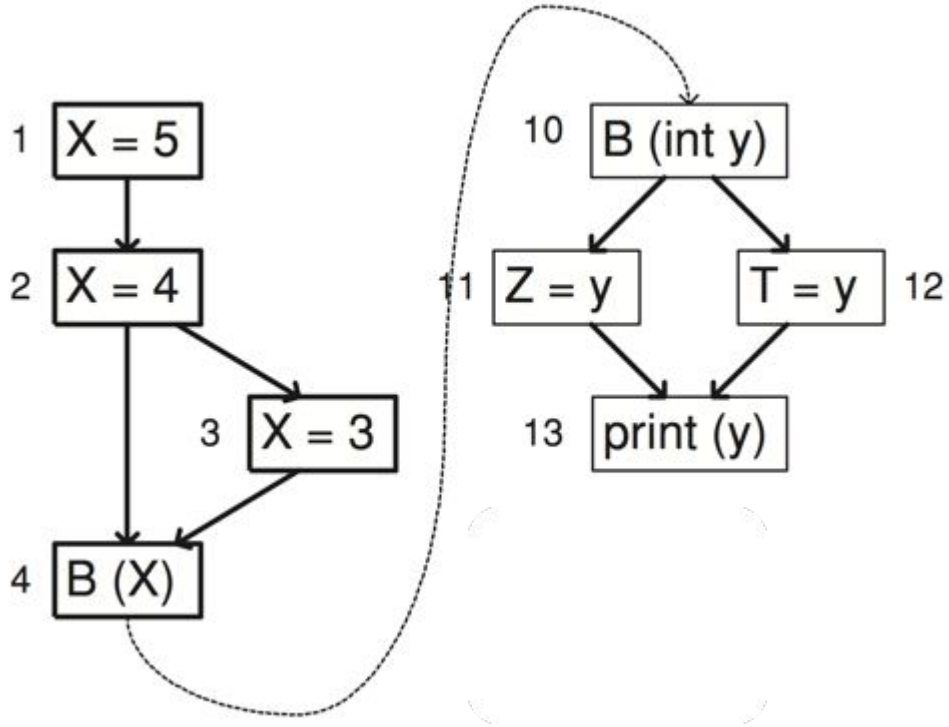
- ***First-use***: El conjunto de nodos que tienen usos de una variable x y para la cual existe un camino ***def-clear*** y ***use-clear*** desde el punto de ingreso hasta los nodos.

- Pueden ser en ambas direcciones:
  - *Caller* a *Callee* a través de un parámetro
  - *Callee* a *Caller* a través de un valor de retorno

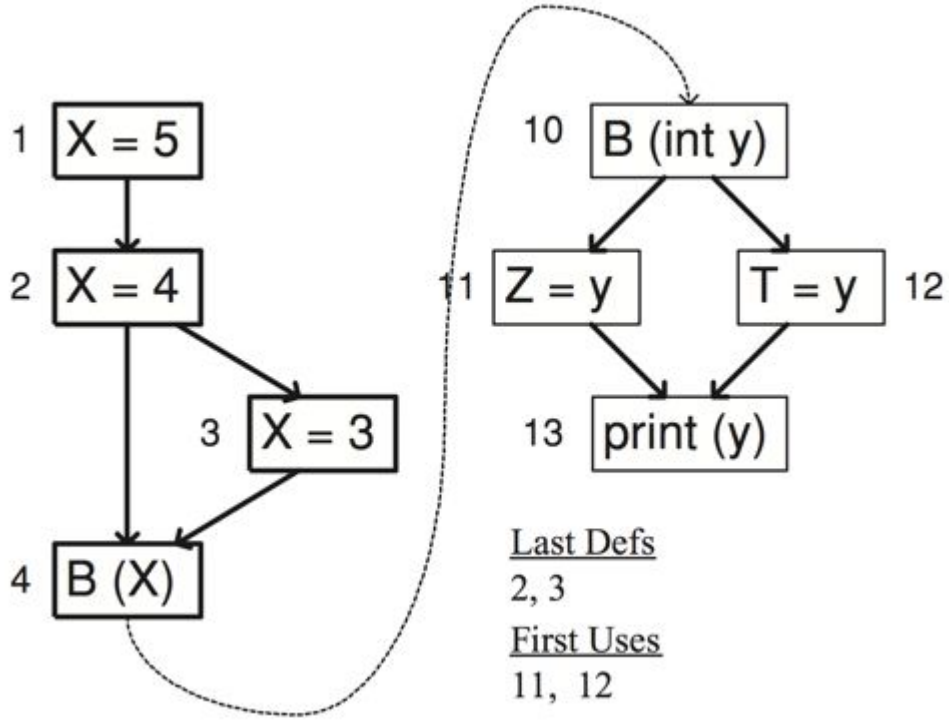
Pares-DU entre unidades: ejemplo



Pares-DU entre unidades: ejemplo



Pares-DU entre unidades: ejemplo



Cobertura de grafos aplicada

- Código fuente
- Elementos de diseño
- Especificación de diseño
- Casos de uso

Cobertura para especificación de diseño

- Describe el comportamiento que debe exhibir un elemento de *software*
- La implementación puede (o no) reflejar la especificación
- Veremos 2 formas de describir comportamiento:
  - Restricciones de secuencia
  - Según estado

Restricciones de secuencia

- Son reglas que imponen restricciones en el orden en que se ejecutan los métodos.
- Buenos ejemplos son estructuras de datos:

Class Queue
void EnQueue(int e)
int DeQueue()

```
public int DeQueue ()
{
// Pre: At least one element must be on the queue.
.
:
public EnQueue (int e)
{
// Post: e is on the end of the queue.
```

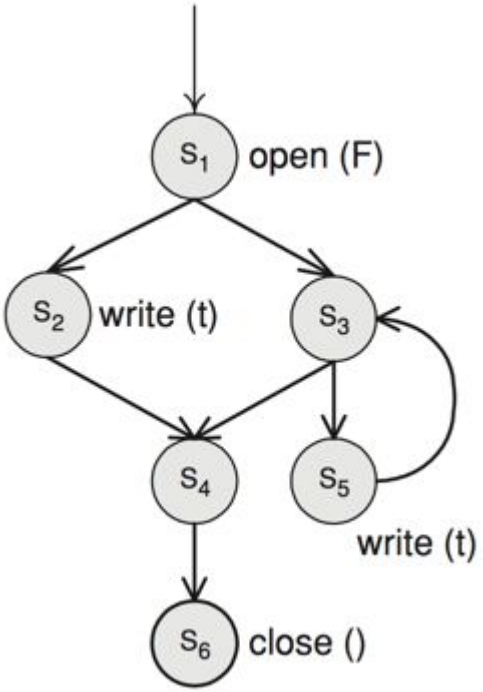
Ejemplo: Escritura de archivo

- Existen los siguientes métodos:
  - *open(String fName)* // abre el archivo *fName*
  - *close(String fName)* // cierra el archivo *fName*
  - *write(String textLine)* // escribe una línea de texto
- Restricciones:
  - Se debe abrir el archivo antes de escribir
  - Se debe abrir el archivo antes de cerrarlo
  - No se puede escribir luego de cerrar un archivo (a menos que se vuelva a abrir)
  - Se debe escribir antes de cerrar el archivo

Análisis estático

¿Existe algún camino que viole alguna de las restricciones?

R4 => [1, 3, 4, 6]



Análisis dinámico

- Puede ser que el programa exija entrar por lo menos una vez al *loop*.
- Analizar de forma estática es insuficiente.
- Análisis dinámico:
  - Generar todos los requisitos de pruebas que violen las restricciones de secuencia.
  - Verificar que las pruebas para estos requisitos sean infactibles.

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

Clase 5
Cobertura de grafos aplicada

IIC3745 – Testing

Rodrigo Saffie

rasaffie@uc.cl

28 de agosto de 2019