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

IIC3745 — Testing
2023 - 1

Tarea 3

Fecha de entrega: Martes 11 de Abril del 2023 a las 23:59

Información General

La siguiente tarea contempla como objetivo principal entender la lógica de una técnica de análisis dinámico usada en la actualidad: code instrumentation y tracer. Al implementar un instrumentador y un tracer sencillo el estudiante podrá ver las ventajas, el alcance y las desventajas de las técnicas en general. No solo cuando uno implementa un instrumentador o un tracer sino cuando uno usa alguno existente.

Objetivos

- Entender en detalle como funciona la técnica code instrumentation y tracer.
- Entender las ventajas, desventajas, y limitaciones de las técnicas.
- Ser capas de adaptar un instrumentador y tracer para reportar cierta información de un programa analizado.

Contexto

En clase se desarrolló un **Profiler** y un **Tracer**, para detectar las funciones que fueron ejecutadas durante la ejecución de un programa. En la tarea 3 se pide diseñar e implementar un `ClassProfiler` y un `CoverageTracer`.

ClassProfiler

Este profiler detecta los métodos de alguna clase que se ejecutaron y enlista los métodos llamados por alguna función ejecutada. Uno de los análisis dinámicos más usados es recolectar información sobre las llamadas entre funciones, ya que, llamar a una función puede implicar la ejecución de una o multiples funciones. Por lo tanto, este profiler se podría usar para detectar si un conjunto de funciones (tests) contempla todos los métodos definidos en una clase dada.

```python
class Rectangle:

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height

    def get_perimeter(self):
        return self.width * 2 + self.height * 2

    def set_width(self, width):
        self.width = width

    def set_height(self, height):
        self.height = height

    def __eq__(self, other):
        return self.width == other.width and self.height == other.height


def test_area():
    rectangle = Rectangle(2, 3)
    assert rectangle.get_area() == 6, 'Incorrect area'

def test_perimeter():
    rectangle = Rectangle(5, 7)
    assert rectangle.get_perimeter() == 24, 'Incorrect perimeter'


test_area()
test_perimeter()
```

Figura 1: Código a ser analizado para `ClassProfiler`

Considere las funcionalidades de `ClassProfiler` mostradas en la Tabla 1 y los resultados esperados al analizar el código ilustrado en la Figura 1.

Nombre y descripción	Verificación de resultado esperado

`report_executed_methods`: Este método retorna una lista de tuplas que representan los métodos que fueron ejecutados (excluye las funciones que fueron definidas fuera de alguna clase). La lista esta ordenada descendentemente en base al segundo elemento de la tupla (t[1]) y cada tupla tiene el formato:
`(<method-name>,<codeline-methodDef>, <method-class>)`
	`myProfiler = ClassProfiler.getInstance()`
	`result = myProfiler.report_executed_methods()`
	
	`expected = [`
	`('__init__', 3, 'Rectangle'),`
	`('get_area', 7, 'Rectangle'),`
	`('get_perimeter', 10, 'Rectangle')]`
	
	`assert result == expected`

`report_executed_by`: Este método recibe el nombre de una función ejecutada (no pertenece a una clase) y retorna una lista de tuplas que representan los métodos que fueron llamados por esa función. La lista retornada esta ordenada descendentemente en base al segundo elemento de cada tupla (`t[1]`), donde cada tupla sigue el formato:
`(<method-name>,<codeline-methodDef>, <method-class>)`
	`myProfiler = ClassProfiler.getInstance()`
	`result = myProfiler.report_executed_by('test_area')`
	
	`expected = [('__init__', 3, 'Rectangle'),`
	`('get_area', 7, 'Rectangle')]`
	
	`assert result == expected`

Table 1: Descripción de los métodos de `ClassProfiler` que reportan información para el Ejercicio 1.

CoverageTracer

Este tracer identifica que líneas de código fueron ejecutadas y cuántas veces. Este tipo de análisis dinámico juega un papel importante en el testeo, ya que, con los tests se busca cubrir la mayor cantidad de diferentes aspectos posibles (código). Por lo tanto, este tracer se podría usar para localizar líneas de código que no fueron ejecutadas por un conjunto de funciones (tests).

```python
def remove_html_tags(s):
    tag = False
    quote = False
    out = ""

    for c in s:
        if c == '<' and not quote:
            tag = True
        elif c == '>' and not quote:
            tag = False
        elif c == '"' or c == "'" and tag:
            quote = not quote
        elif not tag:
            out = out + c

    return out
```

Figura 2: Código a ser analizado para `CoverageTracer`

Considere las funcionalidades de `CoverageTracer` mostradas en la Tabla 2 y los resultados esperados al analizar el código ilustrado en la Figura 2.

Nombre y descripción	Verificación de resultado esperado

`report_executed_lines`: Este método retorna una lista de tuplas que representan las líneas de código que fueron ejecutados y pertenecen a la función bajo análisis. La lista esta ordenada descendentemente en base al segundo elemento de la tupla (t[1]) y cada tupla tiene el formato:
 	
`(<fun-name>, <codeline>)`
	`with CoverageTracer() as coverage:`
	`  remove_html_tags('abc')`
	`result = coverage.report_executed_lines()`
	
	`expected = [`
	`('remove_html_tags' , 2),`
	`('remove_html_tags' , 3),`
	`('remove_html_tags' , 4),`
	`('remove_html_tags' , 6),`
	`('remove_html_tags' , 7),`
	`('remove_html_tags' , 9),`
	`('remove_html_tags' , 11),`
	`('remove_html_tags' , 13),`
	`('remove_html_tags' , 14),`
	`('remove_html_tags' , 16)]`
	
	`assert result == expected`

`report_execution_count`: Este método retorna una lista de tuplas que representan las líneas ejecutadas de la función bajo análisis y cuantas veces esas líneas fueron ejecutadas. La lista retornada esta ordenada descendentemente en base al segundo elemento de cada tupla (`t[1]`), donde cada tupla sigue el formato:
 	
`(<fun-name>,<codeline>,`
`<number-executions-codeline>)`
	`with CoverageTracer() as coverage:`
	`  remove_html_tags('abc')`
	`result = coverage.report_execution_count()`
	
	`expected = [`
	`('remove_html_tags', 2, 1),`
	`('remove_html_tags', 3, 1),`
	`('remove_html_tags', 4, 1),`
	`('remove_html_tags', 6, 4),`
	`('remove_html_tags', 7, 3),`
	`('remove_html_tags', 9, 3),`
	`('remove_html_tags', 11, 3),`
	`('remove_html_tags', 13, 3),`
	`('remove_html_tags', 14, 3),`
	`('remove_html_tags', 16, 1)]`
	
	`assert result == expected`

Table 2: Descripción de los métodos de `CoverageTracer` que reportan información para el Ejercicio 2.

Ejercicio 1

Implemente el código necesario para un profiler de clases:

- **(a)** Implemente la clase `ClassProfiler` que hereda de la clase `Profiler` en un archivo llamado `classInstrumentor.py`. También implemente los métodos de clase necesarios en `ClassProfiler` para usarlos para inyectar código (por ejemplo, `record`). Además implemente el método `instrument` en el archivo `classInstrumentor.py`, este método recibe un AST y devuelve otro AST con el código inyectado. Si tiene dudas, vea el código de clases de `FunctionProfiler` y `Profiler`.

- **(b)** Implemente los métodos `report_executed_methods` y `report_executed_by` en `ClassProfiler`. Note que las descripciones de estos métodos estan ilustrados en la tabla 1.
- **(c)** Implemente la clase `ClassInstrumentor` que hereda de la clase `NodeTransformer`. Implemente los métodos de visita necesarios para inyectar el código usando los métodos de clase de `ClassProfiler` y recolectar la información requerida para esta tarea.
- **(d)** Verifique que los test del archivo `test_tarea.py` pasen. Tenga en cuenta que el código no debe estar hardcodeado. Note que asignará un puntaje entre 0 y 3, en base al número de tests que pasen (3 puntos).

Ejercicio 2

Implemente el código necesario para un coverage tracer:

- **(a)** Implemente la clase `CoverageTracer` que hereda de la clase `StackInspector` en un archivo llamado `coverageTracer.py`. También implemente los métodos `__init__` y `traceit` (función de rastreo) en la clase `CoverageTracer`. Si tiene dudas, vea el código de las clases de `FunctionTracer` y `Tracer`.
- **(b)** Implemente los métodos `report_executed_lines` y `report_execution_count` en `CoverageTracer`. Note que las descripciones de estos métodos estan ilustrados en la tabla 2.
- **(c)** Verifique que los tests del archivo `test_tarea.py` pasen. Tenga en cuenta que el código no debe estar hardcodeado. Note que asignará un puntaje entre 0 y 3, en base al número de tests que pasen (3 puntos).

Entregables

- Debe subir el código de su tarea.
- Debe entregar un archivo README que contiene los nombres de quienes realizaron la tarea y su aporte (Declaración de la tarea).

Archivos iniciales

En canvas se encuentra el código desarrollado en clases con los siguientes folders:

- *instrumentor (folder)* – contiene los archivos para el instrumentador.
  - *input_code (folder)* – que contiene a ejecutar para probar el profiler.
  - *profiler.py* – código de la clase Profiler hecha en clases.
  - *functionInstrumentor.py* – código de las clases que permiten inyectar código y reportar las métricas.
  - *test_clase.py* – contiene el código de prueba desarrollado en clase.
  - *test_tarea.py* – contiene el código de prueba para el ejercicio 1.
- *tracer (folder)* – contiene los archivos para el tracer.
  - *input_code (folder)* – que contiene a ejecutar para probar el FunctionTracer.
  - *stackInspector.py* – código de la clase StackInspector hecha en clases.
  - *tracking.py* – código de la clase Tracer hecha en clases.

- *functionTracer.py* – código de la clase FunctionTracer hecha en clases.
- *tracking.py* – código de la clase Tracer hecha en clases.
- *code_task.py* – código usado para probar el CoverageTracer.
- *test_clase.py* – contiene el código de prueba desarrollado en clase.
- *test_tarea.py* – contiene el código de prueba para el ejercicio 2.

Para la tarea usted debe modificar o agregar código a los archivos anteriormente mencionados según corresponda.

Reportar problemas en el equipo

En el caso de que algún integrante no aportara como fue esperado en la tarea, podrán reportarlo enviando un correo con asunto **Problema Equipo {NumeroGrupo} Testing** a *juanandresarriagada@uc.cl* con copia a *afernandb@uc.cl* explicando en detalle lo ocurrido. Posterior a eso se revisara el caso en detalle con los involucrados y se analizara si corresponde aplicar algún descuento. Instamos a todas las parejas que mantengan una buena comunicación y sean responsables con el resto de su equipo para evitar problemas de este estilo.
Advertencia: Si algun integrante del grupo no aporta en las tareas puede implicar recibir nota mínima en esa entrega.

Restricciones y alcances

- Su programa debe ser desarrollado en **Python 3.9 o superior**.
- Los archivos de código entregados deben terminar con la extensión **.py**.
- En caso de dudas con respecto al enunciado deben realizarlas en un foro relacionado a la tarea que se encontrara disponible en canvas.
- Si no se encuentra especificado en el enunciado o en el código realizado en clases, supón que el uso de cualquier librería de Python adicional a las utilizadas en el código base se encuentran prohibidas. En caso de que estimes necesario podrás preguntar en el foro de la tarea por el uso de alguna librería adicional.

Entrega

Código: Deberán entregar el código que utilizaron para su tarea. La entrega se realizara por medio de un buzón de canvas habilitado para esta tarea.

Declaración de tarea: Deberán entregar un archivo README que contiene los nombres de quienes realizaron la tarea y su aporte. La entrega de este archivo se realizara junto con el código por medio de un buzón de canvas habilitado para esta tarea.

Atraso: Se efectuara un descuento por entregar tareas atrasadas. Se descontara 0.5 si la tarea se entrega con menos de una hora de retraso. El puntaje final en caso de atraso seria calculado mediante la siguiente formula:

$$PuntajeFinal = PuntajeObtenido - (0,5 + 0,05 \cdot k)$$

, donde k es el numero de horas de retraso menos uno.

Integridad académica

Este curso se adscribe al Código de Honor establecido por la Escuela de Ingeniería. Todo trabajo evaluado en este curso debe ser hecho **individualmente** o en **los grupos asignados** según sea definido en la evaluación y **sin apoyo de terceros**. Se espera que los alumnos mantengan altos estándares de honestidad académica, acorde al Código de Honor de la Universidad. Cualquier acto deshonesto o fraude académico está prohibido; los alumnos que incurran en este tipo de acciones se exponen a un Procedimiento Sumario. Es responsabilidad de cada alumno conocer y respetar el documento sobre Integridad Académica publicado por la Dirección de Pregrado de la Escuela de Ingeniería.
¡Éxito con la Tarea! :D