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

IIC3745 — Testing
2024 - 1

Tarea 2

Fecha de entrega: Martes 7 de Mayo del 2024 a las 23:59

Información General

La siguiente tarea contempla como objetivo principal el entender la lógica de técnicas de análisis dinámico usadas en la actualidad: code instrumentation, tracer y sampler. Al implementar un instrumentador, un tracer y un sampler sencillo se podrá ver las ventajas, el alcance y las desventajas de las técnicas en general. Esto será beneficioso al momento de implementar alguna de estas técnicas así como al momento de usar una herramienta que aplica alguna técnica existente.

Objetivos

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

Contexto

En clase se vieron tres técnicas de análisis dinámico: instrumentación, rastreo y muestreo (sampling). Esta tarea trata de utilizar estas tres muestras para recolectar información de la ejecución de programas en Python.

Instrumentación

Modifique el código fuente de la herramienta de instrumentación (`instrumentor`) vista en clases para calcular las siguientes métricas:

Frecuencia (10 pts, básico) – Esta métrica reporta el número de veces que una función fue llamado al ejecutar un programa.

Callers (15 pts, intermedio) – Una función puede ser invocada de diferentes partes dentro un programa, en particular, desde otras funciones. El reporte en consola debe reportar desde que otras funciones la función bajo análisis fue invocada. **Pista:** Para implementar esta funcionalidad, se debe guardar la información de la última función que fue llamada antes de ejecutar cada función.

Cacheable (20 pts, avanzado) – Normalmente, si una función devuelve consistentemente el mismo resultado cuando se le pasan los mismos argumentos, podría ser apropiado implementar un caché para esa función. La herramienta debe identificar las funciones que podrían ser candidatas para aplicar un caché. Desde la primera vez que se llama a una función, se considera como una posible candidata para el caché. Sin embargo, si la función es llamada múltiples veces y en al menos una de esas llamadas el resultado difiere en términos de argumentos o valor de retorno, se descarta como candidata para el caché.

Consideraciones

Valores de retorno Note que implementar la ultima métrica (*Cacheable*) no es posible con la forma actual con la que instrumentamos el código. Actualmente, agregamos una expresión al inicio y al final de la función. Por ejemplo, considere la siguiente función:

```
1 def foo():
2     for _ in range(10):
3         bar()
4         time.sleep(3)
```

Después de aplicar la instrumentación queda de la siguiente manera:

```
1 def foo():
2     Profiler.record start('foo', [])
3     for _ in range(10):
4         bar()
5         time.sleep(3)
6     Profiler.record end('foo', None)
```

Cuando existe una función que retorna un valor como en la función factorial, la instrumentación se ve de la siguiente forma:

```
1 def factorial(n):
2     Profiler.record_start('factorial', [n])
3     answer = 1
4     time.sleep(3)
```

```
    if n > 0:
        answer = n * factorial(n - 1)
    Profiler.record_end('factorial', None)
    return answer
```

Considerando lo anterior, note que no es posible rastrear exactamente que devolvió la función. En este sentido no podemos monitorear si una función devuelve el mismo valor siempre.
Para implementar la ultima métrica (*Cacheable*), es necesario modificar la forma en que instrumentamos. Deben modificar el instrumentador para que después de instrumentar quede de la siguiente forma:

```
def factorial(n):
    Profiler.record_start('factorial', [n])
    answer = 1
    time.sleep(3)
    if n > 0:
        answer = n * factorial(n - 1)
    return Profiler.record_end('factorial', answer)
```

Si se implementa bien, esta modificación no afectaria la extracción de las anteriores métricas. Si no logran realizar esta modificación, dejen la instrumentación antigua y obtendrán los puntajes de las primeras métricas.

¿Cómo reporto la información recolectada? Se espera que el método `report_executed_functions` retorne una lista de instancias de `FunctionRecord`. Por lo tanto considere la clase `FunctionRecord` para guardar la información recolectada.

¿Como lo pruebo?

Para probar su implementación deben ejecutar el siguiente comando en consola:

```
instrumentor> python3 profile.py code1
```

Noten que `code1`, es el nombre del archivo a ser analizado usando su herramienta de instrumentación. La carpeta *input_code* tiene varios archivos para que realicen pruebas. El comando anterior debe imprimir el siguiente reporte en consola cuando termine de ejecutar el código a analizar:

```
fun     freq    cache       callers
main    1          1           []
foo     1          1           ['main']
bar     2          0           ['foo']
```

El código base cuenta con 2 códigos para probar, ustedes pueden agregar más. El equipo docente tendrá sus propios códigos privados de prueba. Los códigos privados serán parecidos a los proporcionados en el código base.

Rastreo

El código desarrollado en clase (`tracer`) permite rastrear las funciones que fueron llamadas durante la ejecución de un programa, así como los eventos de la pila de ejecución.
Modifique el archivo `coverage_tracer.py` para recolectar la siguiente información:

Frecuencia de líneas ejecutadas (15 pts, básico) – Por cada línea ejecutada es necesario obtener: nombre de la función a la cuál pertenece la línea de código ejecutada, número de línea de código que fue ejecutada y cuántas veces fue ejecutada. Note que este tipo de análisis dinámico juega un papel importante al calcular la cobertura de código en Testing; ya que, con los tests se busca cubrir la mayor cantidad de líneas de código posibles. Por lo tanto, esta información se podría usar para localizar líneas de código que no fueron ejecutadas por un conjunto de funciones (tests).

Consideraciones

¿Cómo reporto la información recolectada? Se espera que el método `report_executed_lines` retorne una lista de instancias de `LineRecord`. Por lo tanto considere la clase `LineRecord` para guardar la información recolectada.
Nota: La lista retornada por el método `report_executed_lines` debe estar ordenada ascendentemente en base al número de línea.

¿Cómo lo pruebo?

Para probar su implementación deben ejecutar el siguiente comando en consola:

```
tracer> python3 tracer.py code2
```

Noten que `code2`, es el nombre del archivo a ser analizado usando su herramienta de instrumentación. La carpeta *input_code* tiene varios archivos para que realicen pruebas. El comando anterior debe imprimir el siguiente reporte en consola cuando termine de ejecutar el código a analizar:

```
fun                line        freq
factorial            2           1
factorial            4           1
factorial            5           4
factorial            6           3
factorial            7           3
factorial            8           1
```

El código base cuenta con 2 códigos para probar, ustedes pueden agregar más. El equipo docente tendrá sus propios códigos privados de prueba. Los códigos privados serán parecidos a los proporcionados en el código base.

Muestreo

Actualmente, nuestro `sampler` solo muestra las funciones que se encuentran dentro de la pila de ejecución cada segundo. Deben modificar el código implementado en clases para reportar el tiempo de ejecución de cada método considerando el contexto en el que fue invocado.
Por ejemplo, considere el siguiente código:

```
def main():
    foo()
    bar()
    zoo()

def foo():
    for _ in range(10):
        bar()
        time.sleep(1)

def zoo():
```

```
12      foo()
13
14  def bar():
15      time.sleep(1)
```

En el código anterior la función `bar()` se la llama desde tres contextos diferentes:

1. *main, foo, bar* – Donde `main` llama a `foo` y `foo` llama a `bar`.
2. *main, bar* – Donde `main` llama a `bar`.
3. *main, zoo, bar* – Donde `main` llama a `zoo` y `zoo` a `bar`.

Call Context Tree (25 pts, intermedio/avanzado) – Reporte el tiempo de ejecución que tiene cada método en diferente contexto. El reporte debe mostrarse en base a la estructura de un árbol. A esta estructura se la conoce como Call Context Tree. Por ejemplo, analizando el código anterior, el reporte debe verse similar al siguiente reporte:

```
1  sampler> python3 profile.py code2
2  total (41 seconds)
3    _bootstrap(41 seconds)
4      _bootstrap_inner(41 seconds)
5        run(41 seconds)
6          execute_script(41 seconds)
7            <module>(40 seconds)
8              main(40 seconds)
9                foo(19 seconds)
10                 bar(10 seconds)
11               bar(1 seconds)
12               zoo(20 seconds)
13                 foo(20 seconds)
14                   bar(10 seconds)
```

Pista. Recuerde que la técnica de muestreo actual puede imprimir los métodos que se encuentran en la pila de ejecución cada segundo. Los elementos dentro de la pila ya están ordenados por contexto. Usted debe armar el árbol de llamadas utilizando la información que obtiene de la pila cada segundo. Recuerde que como se saca muestras cada segundo, el numero de veces que aparece un método en la pila es "equivalente" a su tiempo de ejecución.

¿Cómo lo pruebo? Para probar su implementación deben ejecutar el siguiente comando en consola:

```
1  sampler>python3 profile.py code1
```

En este caso, *code1*, es el nombre del archivo a ejecutar. La carpeta *input_code* tiene varios archivos de prueba. El comando anterior debe mostrar algo similar al siguiente reporte en consola:

```
1  total (7 seconds)
2    _bootstrap(7 seconds)
3      _bootstrap_inner(7 seconds)
4        run(7 seconds)
5          execute_script(7 seconds)
6            <module>(6 seconds)
7              main(6 seconds)
8                foo(6 seconds)
9                  bar(1 seconds)
10                 zoo(3 seconds)
11                   bar(1 seconds)
```

El código base cuenta con 2 códigos para probar, ustedes pueden agregar más. El equipo docente tendrá sus propios códigos privados de prueba. Los códigos privados serán parecidos a los proporcionados en el código base.

Tarea

Desarrolle los siguientes ejercicios:

- ***Instrumentación*** – Implemente el calculo de las siguientes métricas en el instrumentor:
  - `Frecuencia`
  - `Callers`
  - `Cacheable`

  Para implementar las métricas deberá modificar el código contenido en la carpeta `instrumentor`. Para obtener el puntaje para cada métrica considere que es necesario (i) generar e imprimir el reporte completo con `print_fun_report` y (ii) retornar el diccionario de instancias `FunctionRecord`. Si no es posible observar el calculo correcto de las métricas considerando los 2 puntos anteriores, no se asignará el puntaje asociado a la métrica correspondiente.

- ***Rastreo*** – Implemente el código necesario para obtener la *Frecuencia de líneas ejecutadas*. Para lograr este ejercicio deberá modificar el código contenido en el archivo `coverage_tracer.py` e implementar el método `report_executed_lines` en la clase `CoverageTracer`.

- ***Muestreo*** – Implemente el código necesario para obtener el *Call Context Tree*. Para lograr este ejercicio deberá modificar el código contenido en la carpeta `sampler` y implementar el método `print_report` en la clase `Sampler`, tal que este imprima el reporte del call context tree.

Importante

- El puntaje a cada ejercicio será asignado basado en la cantidad de pruebas que pasen exitosamente. Se consideraran tanto los codigos de prueba entregados en los archivos iniciales, como los codigos secretos para verificar más casos.

- Se espera que todos los ejercicios sean resueltos extendiendo o modificando en codigo base que se vio en clases.

- Cualquier solución hardcodeada no recibirá puntaje. Nos referimos a soluciones que solo funcionen para ciertos casos, por ejemplo, que solo apunten a resolver 2 o más tests proporcionados en el enunciado de forma intencionada. Las soluciones deben apuntar a resolver el problema para cualquier código que se le envié de input.

Consejos

- Recuerden que pueden utilizar print en su código o usar el debugger proporcionado por su IDE si algo no les esta funcionando como esperaban.

- Si tienen dudas del código base les recomendamos jugar un poco con el código al agregar prints para que puedan ir viendo lo que va pasando con los eventos, la pila de ejecución o como se van visitando los nodos. Si a pesar de eso hay algo que no les quede claro pueden recurrir al foro de dudas de la tarea y preguntar.

Archivos iniciales

En canvas se encuentra el código desarrollado en clase y los archivos relacionados a los casos de prueba para la tarea:

- *instrumentor (folder)* – contiene el codigo relacionado a la implementación de la instrumentación.

- *tracer (folder)* – contiene el código relacionado a la implementación de los rastreadores.
- *sampler (folder)* – contiene el código relacionado a la implementación del muestreo.

Para la tarea usted debe modificar o agregar código a las ubicaciones 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 *nicolas.olmos@uc.cl* explicando en detalle lo ocurrido. Posterior a eso se revisara en detalle la contribución del equipo 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 algún integrante del grupo no aporta en las tareas recibirá la nota mínima en esa entrega.

Restricciones y alcances

- Su programa debe ser desarrollado en **Python 3.10**.
- No debe modificar los nombres de los archivos y clases entregados ya que de lo contrario los tests de la corrección podrían fallar. Pueden crear nuevas clases o archivos adicionales a los entregados para usar dentro de las clases pedidas.
- 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, supón que el uso de cualquier librería de Python adicional a las utilizadas en el código base se encuentran prohibidas. Esta permitido ocupar librerías nativas que hemos usado en clases por ejemplo operator, collections, functools, entre otras. 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 todo el código 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: Cada grupo cuenta con un cupón de atraso. Si el grupo decide activarlo en una tarea podrán realizar la entrega dos días después de la fecha original de entrega. Para ocuparlo solo deben realizar una entrega pasado la fecha señalada en el enunciado. Realizar la entrega atrasada sin contar con un cupón para usar implicara obtener nota mínima en la entrega.

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! :)