PyconAr 2016
Bahía Blanca

Code-cropper: código que genera código más simple

Ing. Gervasio Calderón (Jampp)

Breve historia del proyecto

- 2009-2011: Proyecto de Research en Core Security Technologies (tutor: Lic. Aureliano Calvo).
- 2011: Presentación en la PyCon Argentina (Junín) con el viejo nombre “Bug-reproducer Assistant”.
- 2011: Presentado como Trabajo Profesional Final en FIUBA para mi título de Ingeniero en Informática (tutor: Lic. Andrés Veiga)
- 2015: Subido a GitHub bajo el nuevo nombre de “Code Cropper”
- 2016: Presentación en la PyconAr (Bahía Blanca)

Técnicas tradicionales de detección de bugs (problemas)

- Sistema completo:
  - Complejidad.
  - Lentitud.
  - Se pierde la “Historia” del programa.
- Armar unit test desde cero:
  - No es fácil reproducir el bug.
  - No siempre el test representa al sistema.

Una nueva idea: Code-cropper

- Para evitar estos problemas de las técnicas tradicionales se me ocurrió:
  - Simplificar el programa original, reduciéndolo a las clases y funciones de interés.
  - Reducción dinámica, basada en las llamadas → se conserva la “Historia”.
  - Esto también permite generar Unit Tests de forma más fácil.
- La dificultad técnica que surge a la vista:
  - Cambia el código (metaprogramación).

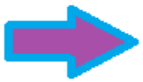
Ejemplo de la idea

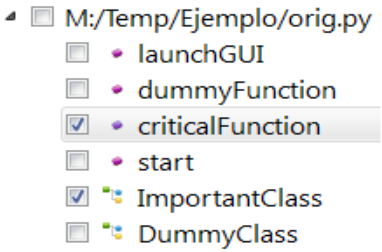
PythonWin - [orig.py]

File Edit View Tools Window Help

```
def launchGUI():
    print 'Please wait. Launching GUI.'
    print 'This may take several minutes.'

def dummyFunction():
    print 'JUAS'

def criticalFunction():
    print 'WARNING!'

class ImportantClass:
    def __init__(self):
        print 'This class is important'

    def importantFunction(self, i):
        print 'This function is really important'

class DummyClass:
    def __init__(self):
        print 'This class is just a joke'

    def dummyFunction(self, i):
        print 'LOL :D'

def start():
    i = 1
    a = ImportantClass()
    dummyFunction()
    dummyFunction()
    launchGUI()
    i += 4
    criticalFunction()
    b = DummyClass()
    criticalFunction()
    b.dummyFunction(i)
    a.importantFunction(i)
```

Ready





result.py

```
import orig

var0 = orig.ImportantClass()
orig.criticalFunction()
orig.criticalFunction()
var0.importantFunction(5)
```

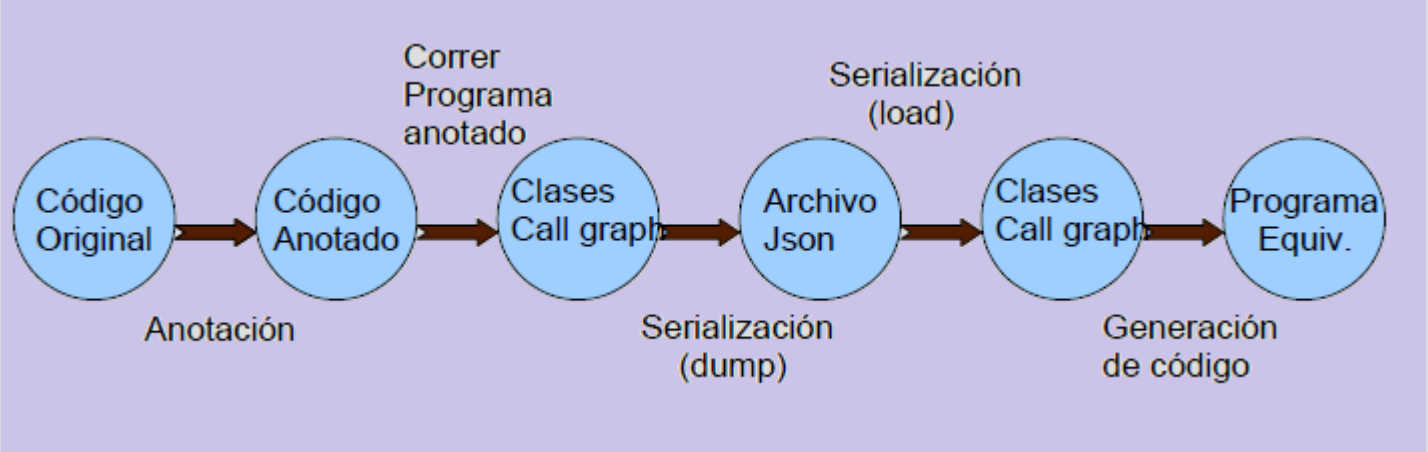
Implementación inicial (C++)

- Enfoque inicial para C++:
  - Parsing en Python de fuentes C++ (sencillo, con expresiones regulares).
  - El parser reemplaza las funciones elegidas (archivo de configuración) por otras que “anotan” al entrar a la función.
  - Modelo mínimo de clases en C++ para representar anotaciones (hacer print a un archivo ***main.cpp***).

Siguiente paso: versión simple para Python

- Enfoque sencillo para Python:
  - En Python se puede reemplazar una función por otra en tiempo de ejecución (funciones ***setattr()*** y ***getattr()***).
  - En el main del programa, se crea una instancia del Annotator que reemplaza las funciones por otras que “anotan” al principio y luego hacen lo mismo que la función original.
  - Al igual que en C++ → Modelo simple de clases en Python, para hacer “print” a un archivo ***main.py***.

Finalmente: modelo unificado con persistencia



- (Idea del Lic. Aureliano Calvo) Modelo de datos para el grafo de llamadas → flexibilidad para los programas equivalentes:
  - Niveles de profundidad.
  - Unit tests.

Finalmente: modelo unificado con persistencia (II)

- Persistencia: archivos Json.
  - Fácil de leer.
  - Se pueden comparar en modo texto.

- Unificación de código Python y C++:
  - Sólo difieren en la “parte inicial” del diagrama (Annotator, call graph, guardado en Json)
  - Mismo código para la “parte final” (está en Python): cargar Json, call graph y ***generación de código.***

Ejemplo: “Demo simple”

- Programa Original:
  - Scripts:
    - prueba.py
    - prueba_anotado.py
    - **MyCode.py:**
      - *launch_GUI()*
      - *dummy_function()*
      - ***critical_function()***
      - ***ImportantClass***
      - *DummyClass*

Ejemplo: Sistema “Cohetes Argento”

- Programa Original:
  - Scripts:
    - cohetes_argento.py
    - cat.py
    - logger.py
    - **main.py**
    - **mathematics.py:**
      - *MathematicalError*
      - *Vector*
      - *Matrix*

Primera pasada de Code-cropper

- Anotando Matrix:
  - Genera un programa equivalente con:
    - Tipos básicos (list, integer)
    - Matrix
    - Dummy('mathematics.Vector')
    - Dummy('logger.Logger')
  - No se reproduce el error (parece ser necesario agregar Vector).

Refinando las anotaciones

- Anotando Matrix y Vector:
  - Genera un programa equivalente con:
    - Tipos básicos (list, integer)
    - Matriz
    - Vector
    - Dummy('logger.Logger')
  - ¿Se producirá el error?

Programa equivalente con Matriz y Vector

```
from code_cropper import dummy
import mathematics
import logger
var1 = [1, 2]
var0 = mathematics.Vector(var1)
var5 = [2, 4]
var4 = mathematics.Vector(var5)
var8 = [var0, var4]
var7 = mathematics.Matrix(var8, dummy.Dummy('logger.Logger'))
var4 = mathematics.Vector(var1)
var10 = [2, 5]
var0 = mathematics.Vector(var10)
var13 = [var4, var0]
var12 = mathematics.Matrix(var13, dummy.Dummy('logger.Logger'))
var14 = mathematics.Matrix.getMatrizInversa(var12)
mathematics.Matrix.printMatriz(var14)
mathematics.Matrix.getMatrizInversa(var7)
```

¡Unit Test generado automáticamente!

```
import unittest
from code_cropper import dummy
import mathematics
import logger

class CohetesArgentoTest(unittest.TestCase):
    def setUp(self):
        unittest.TestCase.setUp(self)

    def tearDown(self):
        unittest.TestCase.tearDown(self)

    def test_main(self):
        var1 = [1, 2]
        var0 = mathematics.Vector(var1)
        var6 = [2, 4]
        var5 = mathematics.Vector(var6)
        var9 = [var0, var5]
        var8 = mathematics.Matrix(var9, dummy.Dummy('logger.Logger'))
        var5 = mathematics.Vector(var1)
        var11 = [2, 5]
        var0 = mathematics.Vector(var11)
        var14 = [var5, var0]
        var13 = mathematics.Matrix(var14, dummy.Dummy('logger.Logger'))
        var15 = mathematics.Matrix.getMatrizInversa(var13)
        self.assertEquals(None, mathematics.Matrix.printMatriz(var15))
        self.assertRaises(mathematics.MathematicalError, mathematics.Matrix.getMatrizInversa, var8)

if __name__ == '__main__':
    unittest.main()
```

¡También podemos entender los race conditions!

- Demo Multithreading:
  - Hay dos threads y una función compartida, que generan un race condition.
  - Anotamos “global_fun()”
  - Funciona porque trabajamos con colas multithreading.
  - El output es muy claro (rompe cuando deja de alternar pares e impares:

```
def test_main(self):
    self.assertEquals(None, MyCode.global_fun(1))
    self.assertEquals(None, MyCode.global_fun(0))
    self.assertRaises(exceptions.AssertionError, MyCode.global_fun, 2)
    self.assertEquals(None, MyCode.global_fun(3))
    self.assertRaises(exceptions.AssertionError, MyCode.global_fun, 5)
```

Formato del Json generado

- El Json es un formato:
  - Jerárquico (módulo → clase → instancia → función)
  - Normalizado (no se repiten las declaraciones)
- Ejemplo de keywords:
  - “callGraph”
    - “arguments”
    - “calleeId”
    - “returnedObject”
  - “language”
  - “languageObjects”
  - “languageTypes”

¿Cómo se logra este efecto en el código?

- Componentes:
  - Modelado de entidades de un programa (*call_graph.py*)
  - Anotación (reemplazo de funciones por otras con *getattr()* y *setattr()*)
  - Uso de colas (*callGraphQueue_*) para desacoplar.
  - Persistencia (lectura y escritura del Json, *serialization.py*)
  - Generación de código (leyendo del Json, *code_generator.py*).

Conclusiones

- Se simplifica mucho la tarea de seguir un programa.
- No sólo sirve para reproducir bugs:
  - Unit Test automáticos.
  - Profiling.
  - Entender cómo funciona un sistema.
  - Descubrir race conditions.

Conclusiones (II)

- No sólo sirve para reproducir bugs:
  - Hacer clases “Mock” fácilmente (leyendo información de los archivos Json).
  - Es un paso adelante para “grabar” ejecuciones de programas como:
    - » Wireshark
    - » Telemetría:
  - IMHO, el estado actual del arte de “grabar ejecuciones de programas” sigue siendo muy primitivo (logging, tracing, etc.)

Oportunidades de desarrollo

- Arreglar problemas conocidos:
  - No funciona con derivados de Object.
  - No soporta pruebas de stress.
- Usar una Base de Datos más potente (para aplicar en sistemas de tiempo real).
- Generar código “multi-threading” (deducir las clases derivadas de Thread del programa original) → hacer reproducibles ciertos “race condition”.
- Implementar el “Mockifier”.

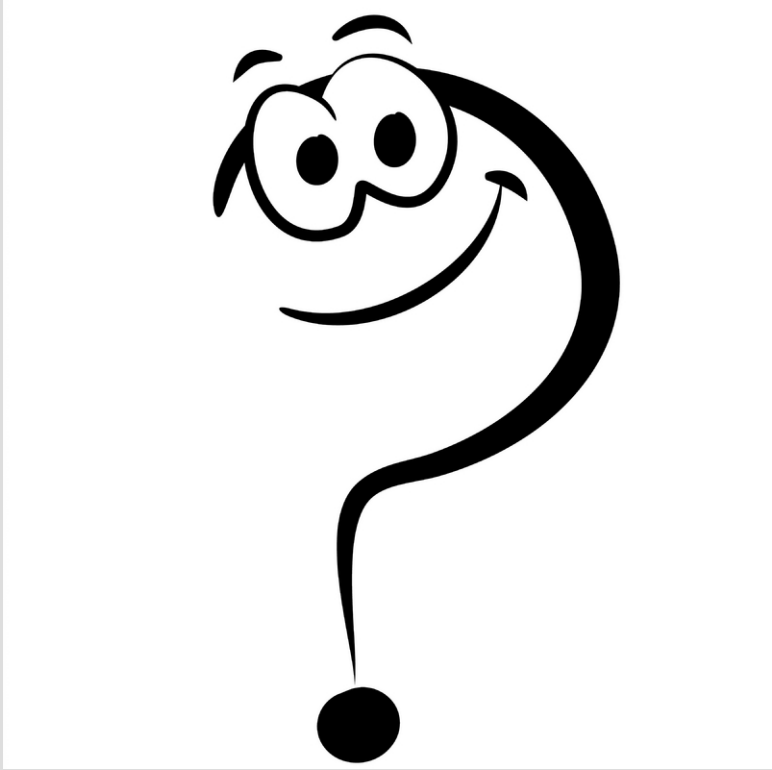
Mockifier

- Idea del Lic. Aurelano Calvo durante una exposición, usar Code-cropper para mockear objetos:
  - Para testing
  - Para abaratar costos o correr más rápido
- Pasos a seguir:
  - Anotar las clases a mockear (modo “escritura”)
  - Correr el programa (escribe en el Json)
  - Anotar las clases a mockear (modo “lectura”)
  - Correr el programa (lee del Json)
- Es como grabar una película y reproducirla después.

A volar la imaginación

- ¿Ideas? Por ejemplo:
  - ¿qué pasaría si anoto “TODO” el programa?
  - ¿cómo se podría reducir el impacto del tiempo en las ejecuciones, para poder grabar la corrida y que el sistema igual haga lo mismo?
  - ¿se podría en algún momento “estandarizar” una ejecución de un programa?
  - ¿Otras ideas?

¿Preguntas?



Si les gustó el proyecto… :-)

- ***GitHub***: https://github.com/GervasioCalderon/code-cropper
- ***Mail***: gervicalder@gmail.com
- ¡Eso es todo, amigos!