Transparent Proxy Objects

- Background (Proxy Objects) - 4
- Problem statement (A Java case analysis) - 6
- Implementation for Java - 4

Background (Proxy Objects) 1/4

- In most programming languages there is the concept of memory reading and memory writing.
- In object oriented programming languages this is modeled as object-field reading and writing.
- Methods can be considered ”callable” fields of an object.

Background (Proxy Objects) 2/4

- Memory reading and writing is supported mainly by a plain/simple ”memory contents copy” model: *assignment*.
    - x = y : copy contents of memory segment indicated by ”y” to memory segment indicated by ”x”
- Sometimes this simple notion of assignment is not enough.

Background (Proxy Objects) 3/4

- A *Proxy Object* will mediate between every memory access requested to it and another object.
- It can, however, *wrap* the assignment notion, additing further logic to it.
  - For example, counting field accesses.

Background (Proxy Objects) 4/4

- In Object-Oriented-Programming terms, a Proxy Object will run its own statements
  - *right before* an operation (method call) on an object,
  - *right after* an operation (method call) on an object.
- This happens transparently, as the programmer is not aware whether he is using a Proxy Object or the original object.

Problem statement (A Java case analysis) 1/6

- Java, being an object oriented language, can adopt the TPO-pattern as described above.
    - An object can act as a proxy to another object, transparently (”the programmer doesn't have to know about it”).

Problem Statement (A Java case analysis) 2/6

- How to implement the TPO pattern in Java?
- A straighfoward way:
    - Before every ”field access” (field access or method call), call a ”getInstance()” method on the proxy object.
    - ”getInstance()” performs the extra proxy logic and then returns the original object instance.

Problem Statement (A Java case analysis) 3/6

- This way, however, ignores the ”transparent” characteristic: the programmer has to be aware of whether she is using a proxy object in order to call the ”getInstance()” method before working on the actual object.
- Solution: post-processing byte code and inserting the ”getInstance()” part of the code transparently.

Problem Statement (A Java case analysis) 4/6

- The problems statement is:

Given a java program and some proxy-types (ProxyT), transform the program's bytecode so that proxy objects are used transparently, without the programmer having to write any extra code for it.

Problem Statement (A Java case analysis) 5/6

- For example:

  ```
  public class Main {
    public static void main(String[] args) {
        Foo    f  = fooBuilder.getAFoo();
        Proxy  p  = new Proxy(f);
        Foo    pf = (Foo) (Object) p;

        f.m1(); pf.m1();
    }
    }
  ```

- ”f” points to a Foo object, ”pf” points to a Proxy, but they are both used as Foo-s: transparency.

Problem Statement (A Java case analysis) 6/6

- The code from the example should be automatically converted to:

```
public class Main {
  public static void main(String[] args) {
     Foo    f  = fooBuilder.getAFoo();
     Proxy  p  = new Proxy(f);
     Foo    pf = (Foo) (Object) p;
     // replacement for ”f.m1()”
     Foo tmp;
     if (f instanceof Proxy)
        tmp = (Foo) ((Proxy)f).getInstance();
     else
        tmp = f;
     tmp.m1();
     // … same for ”pf”
  }
  }
```

Implementation for Java 1/4

- The Soot library is used to load and manipulate Java bytecode.
- Soot provides a simplified java-bytecode view, called *Jimple*.
- Jimple transforms every kind of complex statement to some very basic forms, so it is easy to handle many cases of proxy-object-accesses in java with only 1-2 cases in Jimple code.

Implementation for Java 2/4

- Soot also provides a Points-to analysis.
- A Points-to analysis is used to determine which variable *might* be pointing to a proxy object. Those variables get *tagged.*
- Going through the jimple-statements, we have to whether proxy objects are referenced in:
    - Invokation statements
    - Assignment statements

Implementation for Java 3/4

- In Invokation Statements:
  - If the variable on which the method is invoked is tagged, then perform the code transformation described before (if o instaceof proxy …).
- In Assignment Statements:
  - If the rhs is a field access or a method call on a tagged variable, transform it.
  - Same for the lhs.

Implementation for Java 4/4

- These two types of transformations are enough, as Jimple transforms everything to simple three-field statements.
- Also, only on statement is executed per Jimple triad. So we know that we have to perform at most one transformation per Jimple statement.