**CGLIB introduction**

Today I would like to briefly discuss the bytecode generation framework, CGLIB.  
  
There are a lot of these frameworks, each one works at the different level of abstraction.  
Recently I was looking for a high-level framework that would let me to dynamically change my classes providing its proxy and substituting the functionality of some methods.  
  
While the most obvious jdk proxies can do the job (java.lang.reflect.Proxy), I've figured out, that when I don't have both an interface and implementation of my to-be-proxified class, it just doesn't work. So I've found another solution, a library called [CGLIB](http://cglib.sourceforge.net/)  
  
The only significant drawback for me was a **lack of comprehensive documentation**, in fact I've found only one decent tutorial [here](http://jnb.ociweb.com/jnb/jnbNov2005.html) (It could be great if someone could point me on more tutorials about this tool).  
Anyway, I think that a beginner's level introduction can't harm so I fill the gap and share the experience :)  
  
So CGLIB is a bytecode generation library, that relies on low-level  
[ASM](http://forge.ow2.org/projects/asm) framework.  
So in order to create a working example we'll need to open a regular java project and add two jars as a dependency (the latest versions available at the moment):  
  
- cglib-2.2.jar  
- asm-all-3.2.jar  
  
We will 'proxify' the mock Algorithm class which is supposed to implement some long-running algorithm. We would like to measure its execution time.  
  
So we create our algorithm like this:

public class Algorithm {

public void runAlgorithm() {

System.out.println("running the algorithm");

try {

// do something here -

// simulate some real time-consuming operation here

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Now the most interesting part of the program:  
We'll create a class that adds the 'measurements'. This class will be used by CGLIB to proxify our algorithm, so it should implement *net.sf.cglib.proxy.MethodInterceptor*  
  
The class looks like this:

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class MyInterceptor implements MethodInterceptor {

// the real object

private Object realObj;

// constructor - the supplied parameter is an

// object whose proxy we would like to create

public MyInterceptor(Object obj) {

this.realObj = obj;

}

// this method will be called each time

// when the object proxy calls any of its methods

public Object intercept(Object o,

Method method,

Object[] objects,

MethodProxy methodProxy) throws Throwable {

// just print that we're about to execute the method

System.out.println("Before");

// measure the current time

long time1 = System.currentTimeMillis();

// invoke the method on the real object with the given params

Object res = method.invoke(realObj, objects);

// print that the method is finished

System.out.println("After");

// print how long it took to execute the method on the proxified object

System.out.println("Took: " + (System.currentTimeMillis() - time1) + " ms");

// return the result

return res;

}

}

The last class is the main class. Here we will actually create the proxy so here we'll see some CGLIB related code:

import net.sf.cglib.proxy.Enhancer;

public class Main {

public static void main(String[] args) {

// 1. create the 'real' object

Algorithm alg = new Algorithm();

// 2. create the proxy

Algorithm proxifiedAlgorithm = createProxy(alg);

// 3. execute the proxy - as we see it has the same API as the real object

proxifiedAlgorithm.runAlgorithm();

}

// given the obj, creates its proxy

// the method is generified - just to avoid downcasting...

public static <T> T createProxy(T obj) {

// this is the main cglib api entry-point

// this object will 'enhance' (in terms of CGLIB) with new capabilities

// one can treat this class as a 'Builder' for the dynamic proxy

Enhancer e = new Enhancer();

// the class will extend from the real class

e.setSuperclass(obj.getClass());

// we have to declare the interceptor - the class whose 'intercept'

// will be called when any method of the proxified object is called.

e.setCallback(new MyInterceptor(obj));

// now the enhancer is configured and we'll create the proxified object

T proxifiedObj = (T) e.create();

// the object is ready to be used - return it

return proxifiedObj;

}

}

The output of this program is predictable :)  
  
*Before*  
*running the algorithm*  
*After*  
*Took: 500 ms*

http://java.dzone.com/articles/cglib-missing-manual