2015

Hot swapping

Implementation in three languages

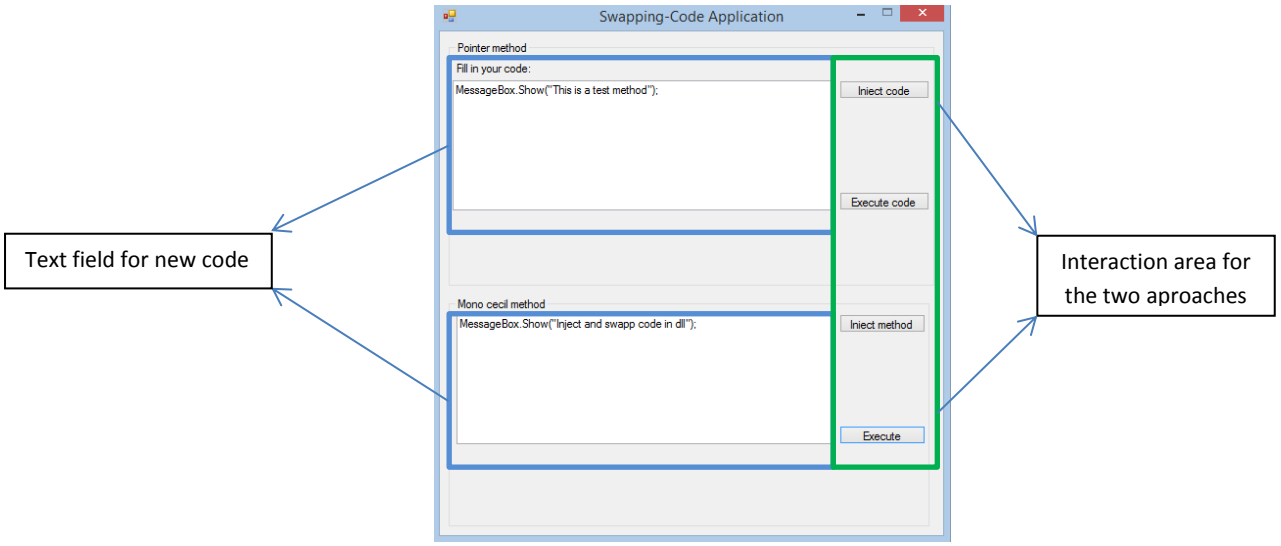
Florian Grill
FH Wiener Neustadt
20.05.2015

Inhalt

Hot swapping in C# 1
Pointer approach 2
Mono Cecil approach 3
Hot swapping in Java 4
JavaAssist approach 5
ByteBuddy approach 6
Python 8

Hot swapping in C#

For this project two methods for hot swapping C# code was evaluated. The First method tries to swap a method by changing its pointer in the method table of the specific class. The second method uses the well-known API Mono.Cecil. The second approach will try to swap a method in a DLL at runtime. Below you can find an image of the current project and some explanations. The project was developed in Visual Studio 2013 and use .Net Framework 4.0. Start the application by double clicking the "exe" in the application folder.



Pointer approach

This approach has been found during researches on the Internet at codeproject[1]. It was developed by Ziad Elmalki. The following description will summarize the information in this article. All .Net Applications are compiled in CLI (Common Language Runtime). In general processors are not able to run this Intermediate Code directly. This is where the JIT (Just in time compiler) coming into force. The JIT will turn the CLI Code to machine code which the processor can work with. Important is that every method will only be compiled once by the JIT because of performance issues. The JIT will cache the machine code for future calls. Every method have a so called JITStub, it is a chunk of machine code that will invoke the JIT for the specific method. After a method is JIT'ed the code in the stub is replaced by code which calls the machine code directly.

Every class has a so called method table which contains the addresses of all JITStubs of his methods. It is used when as long at the specific method has not already been JIT'ed, because otherwise the machine code which is created by the JIT will call the STUB address directly. This approach of swapping methods will try to change the addresses of the method table so it will point on another method. But because of the described limitations this approach will only work once. After a user clicks the "Execute code" button in the Pointer method section, the code gets JIT'ed and his pointer cannot be changed anymore because he will no longer refer to the method table address. A solution was found by calling the specific method by reflection. It seems that

[1] http://www.codeproject.com/Articles/37549/CLR-Injection-Runtime-Method-Replacer

then there are no limitations of swapping the code but the practical use of this approach is limited.

In the application the user can enter some code in the Textbox which will get compiled by clicking at the button “Inject code”. The program uses the `CodeCompileUtil` class which surrounds the entered text in the Textbox by a small application skeleton. After compilation it returns the compiler results and in case of successful compilation an assembly object. After that the program reads the method information of the source and destination method which is stored in a `MethodBase` object. See above code example.

```
Type program = assembly.GetType("HotSwapping.Container");
MethodBase swapMethod = program.GetMethod("containerMethod", BindingFlags.Instance |
BindingFlags.Public);
MethodBase originMethod = typeof(SwapContainer).GetMethod("swapMe",
BindingFlags.Instance | BindingFlags.Public);
```

Now the program has to find the Pointer to these methods. It is important that the two methods must have the same method signature otherwise it is not possible to swap the code. In our example we use .Net 4.0, it is not proved that this method will also work in further releases. The code from Ziad Elmalki shows different solutions for getting the addresses of a specific method. At the time of writing it will support all .Net Framework versions (newest version .Net 4.5). If the program uses .Net Framework 2.0 or higher there is an easier solution to get a method pointer than the solution of Ziad Elmalki. See code below.

```
private static IntPtr GetMethodAddress20SP2(MethodBase method)
{
       unsafe
       {
              return new IntPtr(((int*)method.MethodHandle.Value.ToPointer() + 2));
       }
}
```

When we have the two pointers we can simple change it and swap the methods during runtime by calling this piece of code.

```
IntPtr destAdr = GetMethodAddress(dest);
unsafe
{
    if (IntPtr.Size == 8)
    {
        ulong* d = (ulong*)destAdr.ToPointer();
        *d = *((ulong*)srcAdr.ToPointer());
    }
    else
    {
        uint* d = (uint*)destAdr.ToPointer();
        *d = *((uint*)srcAdr.ToPointer());
    }
}
```

My conclusion of this approach is that the practical use is limited. First, this approach works only before the method gets JIT'ed and second it is not guaranteed that the code will work in further .Net Frameworks.

Mono Cecil approach

This approach tries to use the Library Mono Cecil[2] to modify a loaded DLL during runtime and swap a method and populate its method body with new instructions. After successfully compilation of the entered Text in the Textbox by the `CodeCompileUtil` class, the compiled assembly is saved in a temporary DLL file in the root folder of the program. With Mono Cecil we will read the temporary DLL and the specific method we want to swap. After successful swap we write the new DLL back to Disk. Below you can find the code of the swapping mechanism.

```
public void swappMethods(Assembly assembly)
{
	AssemblyDefinition tempAssembly =
		AssemblyDefinition.ReadAssembly("tempfile.dll");
	TypeDefinition tempClass = tempAssembly.MainModule.Types.FirstOrDefault(x =>
		x.Name == "Container");

	MethodDefinition injectMethod = tempClass.Methods.FirstOrDefault(x => x.Name ==
		"containerMethod");

	// Get the SwappingClass from the dll
	TypeDefinition swappingClass =
		this.swappingAssembly.MainModule.Types.FirstOrDefault(x => x.Name ==
			"SwappingClass");

	// Get the swappingMethod from the SwappingLibrary
	MethodDefinition swappMethod = swappingClass.Methods.FirstOrDefault(x => x.Name
		== "swappingMethod");

	// We inject "containerMethod" in the SwappingLibrary.dll
	if (swappMethod != null && injectMethod != null)
	{
		// clear the old method body
		swappMethod.Body.Instructions.Clear();
		// Get the ILProcessor of the method body
		var ilProcessor = swappMethod.Body.GetILProcessor();

		// Get the MethodReference of the new method body
		MethodReference methodReference = this.importMethod(injectMethod);
		Instruction inst = ilProcessor.Create(OpCodes.Call, methodReference);
		ilProcessor.Append(inst);

		// Write new library back to disk
		this.swappingAssembly.Write("SwappingLibrary.dll");
}
	 }

private MethodReference importMethod(MethodDefinition injectMethod)
{
    return this.swappingAssembly.MainModule.Import(injectMethod);
}
```

[2] http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/

The problem with the method is the behaviour how .Net loads and handles DLLs. Once a DLL is referenced it is loaded in the current AppDomain and gets locked. This means it is not possible to modify or update this DLL during your program runs. Unfortunately C# doesn't provide any possibility to unload a used DLL from the current AppDomain. One workaround is to create a new temporary AppDomain load the DLL and unload it after using. This will also release all used DLLs in this AppDomain so it can modified again. This means in a strict sense that with this approach we are not able to hot swap code because the DLL we use for swapping mechanism is not really loaded during runtime. We only load it for executing and unload it afterwards. Nevertheless for demonstration purpose I integrated this solution in the project. To not locking the DLL when executing its method the program loads the DLL in a temporary AppDomain and calls the function via reflection. Below you can a find short example of using a DLL without locking it.

```
public void doWorkWithShadow()
{
	AppDomainSetup setup = new AppDomainSetup
	{
		ApplicationBase =
		AppDomain.CurrentDomain.SetupInformation.ApplicationBase,
		ShadowCopyFiles = "true",
		ShadowCopyDirectories = PathHelper.getAssemblyPath(),
	};
	AppDomain newDomain = AppDomain.CreateDomain("tempDomain2", null, setup);
	//Create an instance of loader class in new appdomain
	var swappingLibrary = newDomain.CreateInstanceAndUnwrap("SwappingLibrary",
		"SwappingLibrary.SwappingClass");
	Type type = swappingLibrary.GetType();
	MethodInfo methodInfo = type.GetMethod("swappingMethod", Type.EmptyTypes);
	object instance = Activator.CreateInstance(type);
	methodInfo.Invoke(instance, null);

	// Unload the application domain and its resources
	AppDomain.Unload(newDomain);
}
```

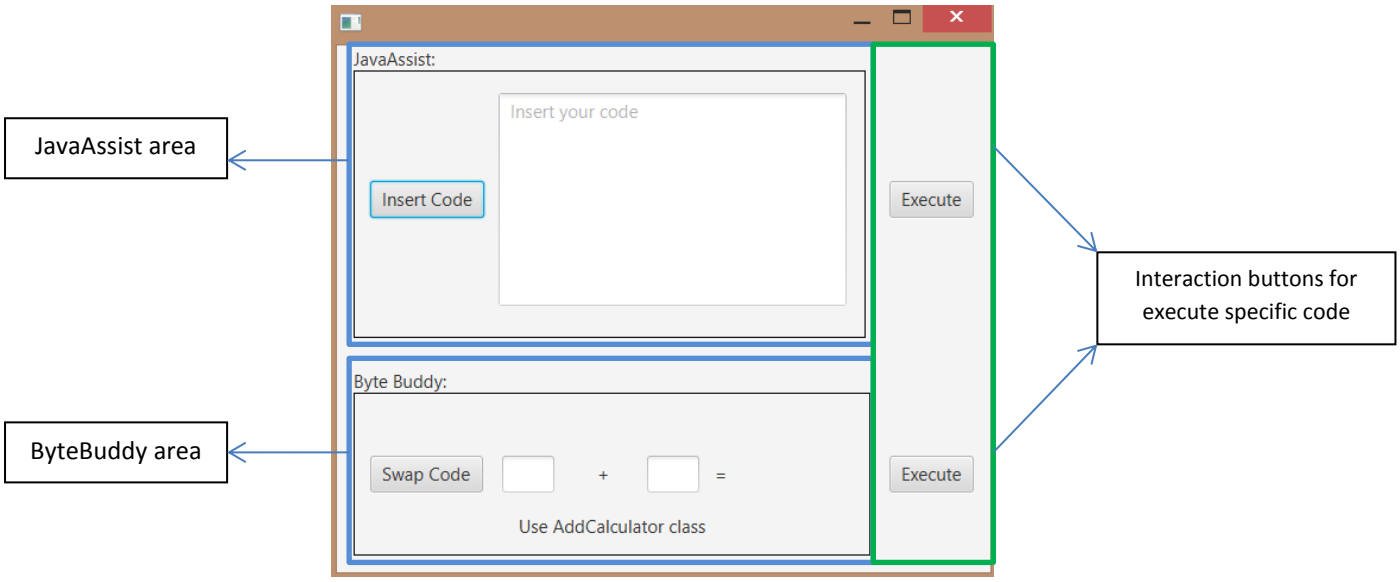
As described above this approach works only with a workaround. Nevertheless the practical use of this method is also limited for hot swapping. If it is only necessary to modify a compiled DLL without completely recompile it Mono Cecil is one of the best approaches.

Hot swapping in Java

Also in the Java application two approaches of hot swapping was evaluated. In Java it was much easier to find some libraries and explanations for hot swapping byte code. In this project two libraries were used, the first one is called Javassist[3] which makes bytecode manipulation simple and the second one is called ByteBuddy[4]. The program is a JavaFX (Java 8) application and it was developed in Netbeans. Below you can find an image of the final application. You can start the application with the batch file in application folder.

[3] http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/

[4] http://bytebuddy.net/



JavaAssist approach

The UI of the JavaAssist area is similar to the C# example. The User can enter simple Java code in the Textbox and replace a specific method body by this. The JavaAssist library is very user friendly and easy to use. The code to be injected is represented as strings and can inject to any existing method during runtime. In order to use JavaAssist in the program you have start the VM of your application with following parameters:

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=8000 Test
```

Java is compiled into bytecode and it's stored in a binary file called class file. Each class file represents one Java class or interface. JavaAssist provides a class which is called CtClass. It provides you the possibility to deal with class files. Below is a short example of the used method in the project.

```
private void swapMethod(String swapCode) {
	this.swap = new HotSwapper("8000");
	if (this.swap != null) {
		ClassPool pool = ClassPool.getDefault();
		try {
		CtClass target = pool.get("hotswapping.swappingClass.SwappingClass");
		target.defrost();
		CtMethod targetMethod = target.getDeclaredMethod("swappingMethod");
		targetMethod.setBody("{" + swapCode + "}");
		swap.reload("hotswapping.swappingClass.SwappingClass",
			target.toBytecode());
		} catch (CannotCompileException ex) {
		} catch (NotFoundException ex) {
		} catch (SecurityException ex) {
		} catch (IllegalArgumentException ex) {
		} catch (IOException ex) {}
	}
}
```

Java Assist provides class for reloading a class at runtime. This class is called `HotSwapper`. To use this class the program has to be launched with the JPDA (Java Platform Debugger Architecture) enabled. At first we will get the current available class pool which is possible by calling `ClassPool.getDefault()`. The ClassPool object is a hashtable of CtClass objects which use the class name as key. So we can get the specific "SwappingClass" simply by name. JavaAssist freezes the CtClass object to protect it for further modification. We have to explicitly call the defrost method so that the CtClass object can be modified again. We can then search for the specific method and replace its body by the `swapCode` variable (has to be a valid java code, otherwise this line will throw an exception). After that we can call the reload function of the `HotSwapper` class. This method will first unload the specific class and load a new version which is represented by a byte array which contains the new content of the class. My Personal opinion of this approach is that JavaAssist is very easy to use and provides good code documentation. Nevertheless hot swapping only works if the application has JPDA enabled, otherwise the JVM does not allow to dynamically reloading a class.

ByteBuddy approach

For the second approach I used the library ByteBuddy[5] version 0.5.6. It is relative new library and it may have some problems as I will describe below. ByteBuddy is a code genearation library which can create classes during runtime. The application shows the features of ByteBuddy in a simple example. In the lower section of the application you can enter two values in the textboxes and perform a simple calculation. The default class which will perform the calculation looks as shown below.

[5] http://bytebuddy.net/#/

```
package hotswapping.swappingClass;

/**
 *
 * @author Flo
 */
public class SubstractCalculator {

    public int calc(int a, int b){
        int result = a - b;
        System.out.println("SubstractCalculator result is: " + result);
        return result;
    }
}
```

With ByteBuddy we will try to replace this whole class by another one. The new class is called `AddCalculator` class and is shown in the next code snippet.

```
package hotswapping.swappingClass;

/**
 *
 * @author Flo
 */
public class AddCalculator{

    public int calc(int a, int b){
        int result = a + b;
        System.out.println("AddCalculator result is: " + result);
        return result;
    }
}
```

To enable the HotSwap feature you have to specify the Java agent on the startup of the JVM by using the -javaagent parameter. ByteBuddy even offers the possibility to load a Java agent during runtime when the Java application is run form a JDK version of the JVM. You only have to perform the `ByteBuddyAgent.installOnOpenJDK()` method. This could be a very convenient alternative to the first solution. In the version 0.5.6 of ByteBuddy there is a bug in the code which prevents you from starting an application with -javaagent parameters. I have contact the developer of ByteBuddy and he promised to fix this error in the next releases of ByteBuddy[6]. Below you can find the code snippet which will swap the classes.

```
new ByteBuddy()
        .redefine(SubstractCalculator.class)
        .name(AddCalculator.class.getName())
        .make()
        .load(
                AddCalculator.class.getClassLoader(),
                ClassReloadingStrategy.fromInstalledAgent()
        );
```

[6] http://stackoverflow.com/questions/29537813/byte-buddy-hotswap-with-bytebuddyagent

After this we will now have redefined the class `AddCalculator` to become `SubstractCalculator`. This redefinition will even apply for pre-existing instances. There is just one limitation of Java's HotSwap feature. Current implementations require that both classes apply the same class schema before redefinition. There is no way to add new methods or new fields to a class when reloading. In general the ByteBuddy Library is a nice little tool which is easy to use and it is worth to try it. According to the official website there are plans to extend the HotSwap feature in the future.

Python

In Python two approaches of hot swapping have been evaluated. For this program I used Python version 3.4.3. Python provides the feature of dynamically load modules and reloads already loaded modules. For demonstration purpose a simple test program which loads a module and executes a function was developed. You can start the application by executing the hotSwapping.py file from the command line. The file is located in the application folder. The skeleton from the program is shown below.

```
from importedModule import ModuleClass

def main():
   while True:
      inputValue = input("Want to execute program? (y/ n): ")
      if inputValue == "n":
          break;
      # dynamically load module
      dynamicExecute()
      # reload loaded module
      reloadExecute()

def dynamicExecute():


def reloadExecute():


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

In the first line the module ModuleClass will be imported. This module provides a "doSomething" function. The function `dynamicExecute` loads the module every time before the "doSomething" gets called. This means the module is loaded not only once but several times. So every modification in the ModuleClass which was made during runtime comes into effect. Below you can see the function `dynamicExecute`.

```
def dynamicExecute():
      # Dynamically read source before executing and assign it to variable module
      module = imp.load_source("importedModule", "./importedModule.py")
      # Execute method from loaded source
      module.ModuleClass.doSomething()
```

The second approach tries to reload the already loaded ModuleClass. Look code section below.

```
def reloadExecute ():
      global ModuleClass
      # Reload already loaded ModuleClass
      ModuleClass = importlib.reload(sys.modules['importedModule']).ModuleClass
      # Execute method from imported module
      ModuleClass.doSomething()
```

For testing the program start the command line and navigate to the “hotSwapping.py” file. Start the program by writing “python hotSwapping.py”. You will get asked if you want to run the program. Type “y” and the method `dynamicExecute` will get called twice (Once for each approach). If you want to modify the ModuleClass open the “importedModule.py” file and change their content. The next time the `dynamicExecute` will get called you will see the changes you have made. To improve the program a method, which automatically reloads a module every time its content changes, could be implemented. This could be very useful in big projects. A small change does not require a completely restart of the program. There is already a project which does exactly this[7]. It is open source and everyone can enhance and add features.

7 https://pypi.python.org/pypi/hotswap/0.1