Portable Semantic Alterations In Java: A Load-Time, Class Based, Macro System.

RDT

Marc Ségura-Devillechaise

msegura@emn.fr

Ecole des Mines de Nantes

Introduction: context

- OOP —> code reuse
- but difficulties to combine new components
  - a super interface may be missing etc..
- one solution:
  - shift some decisions from components adaptation time to component integration time
- needs to be able to alter binary packaged components

Introduction: purposes

- provide alterations capabilities to Java
  - most 00 languages are based on bytecodes (to abstract the execution environment.
  - components are provided in bytecode without source code
- alterations should be performed on bytecodes

Introduction: plan

- I State of the art: position towards existing tools
- II Naive: our proposition
- III An example: demonstration of use

State of the art

- **I State of the art**
- II Naive
- III An example

How to compare ?

- granularity of enabled alterations
- ease of use ˜ abstractions levels of the different entities offered to the programmer

How to enumerate ?

- bytecode level abstractions
  - typically direct reifications of bytecode instructions
  - explicitly with assembly like entities
- source level abstractions
  - closer to the entities manipulated by a programmer mind while writing source code
  - typically reifications of class, methods..

Bytecode level abstractions

- non general purpose
  - BIT, BCA, Hyper/J, Kimera
  - very specific, sharp reduction of expressiveness
- general purpose
  - BCEL, JOIE, JikesBT
  - very expressive —> allow fine grained alterations
  - bytecode level oriented —> not easy to use

Source code level abstractions

- model built by the reflection community
- non portable reflective extensions of Java
  - source code rewriting: OpenJava, Kava
  - dedicated virtual machine: MetaXa, Guarana
  - non portable —> useless in our case
- portable reflective extensions of Java
  - Kava and Javassist
  - interesting but grain of alterations: class or instance member (not ability to go within a method body)
  - grain of alterations —> not very fine grained alterations
  - source level oriented —> easy to use

Naive

- I State of the art
- **II Naive**
- III An example

Naive goals

- still be easy to use
  - source code level entities
- still be portable
  - alterations performed on bytecodes
- be fine grained
  - ability to alter things within a method body

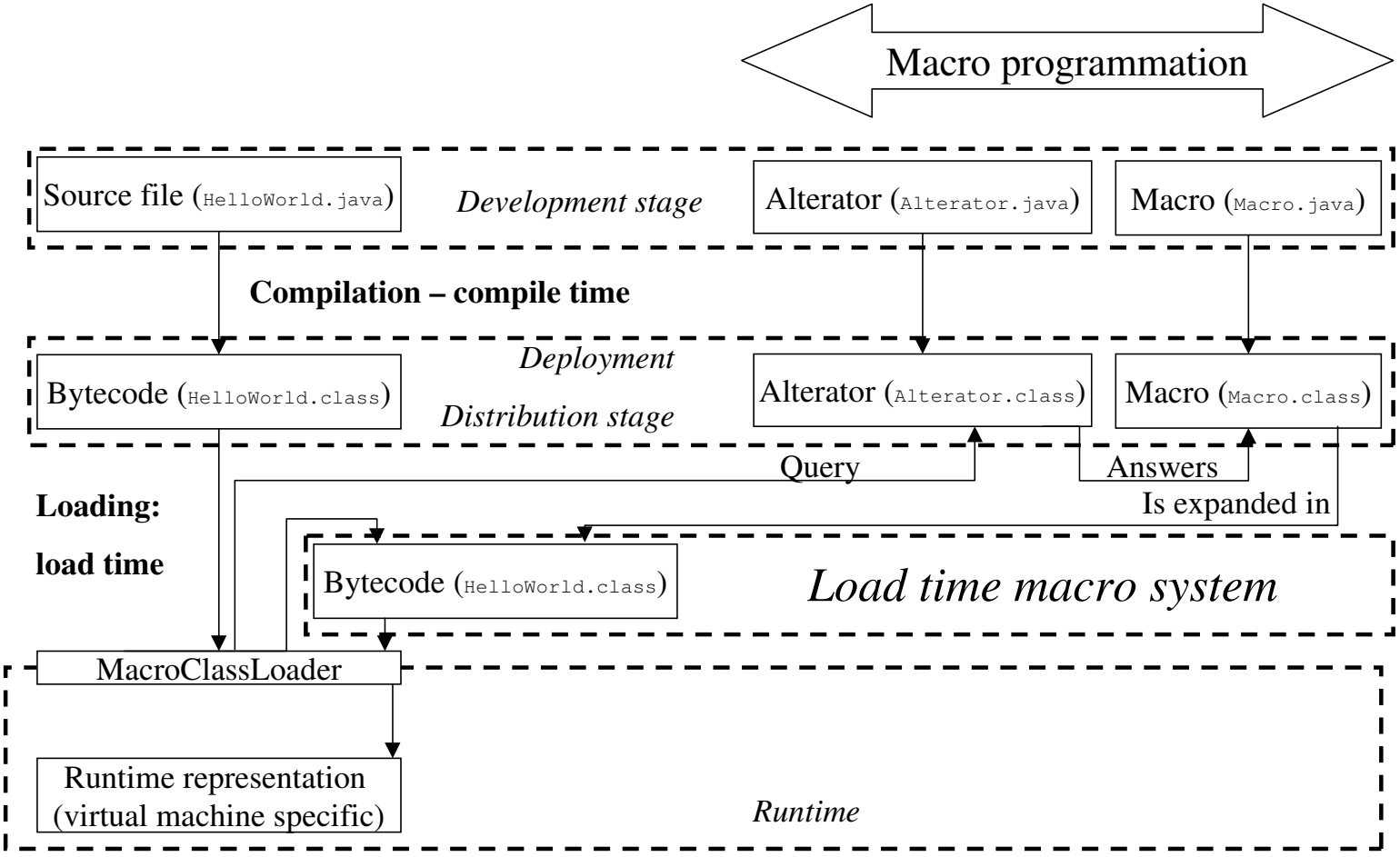
Merging or not?

- Question: should the altered code be merged at runtime with the altering code ?
  - in the case of merging, it is still possible to apply a hook insertions strategy
  - merging increases performance
- Naive merges the altering and the altered code at runtime
- altered code ˜ a macro
- Naive ˜ a macro system at the bytecode level

Macro expansion

- the user locates the method in which the expansion will take place using usual `Class`, `Method` entities
  - rich logical and contextual information
  - easy and unexpensive analysis to perform on the bytecode level
- within a method a visitor pattern is applied
- macro expansion are triggered on specific source code language constructs appearing inside a method body
  - limited logical and contextual information: "static" information
  - based on the ideas that some bytecode instructions directly maps to some source code level contructs

Architecture



Constructions triggering a macro expansion

	Pseudo source code examples	Bytecode instructions
Messages sent	`object.`*toString()*	`INVOKEVIRTUAL` `INVOKESTATIC` `INVOKESPECIAL` `INVOKEINTERFACE`
Constructor	*new* `Object()`	`INVOKESPECIAL`
Exception throwing	*throw* `exception`	`ATHROW`
Field read	`point.`*x*	`GETFIELD` `GETSTATIC`
Field written	`point.`*x*`=0`	`PUTFIELD` `PUTSTATIC`
Cast	*(Object)*`point`	`CHECKCAST`
Run-Time Type Identification	`object` *instanceof* `String`	`INSTANCEOF`
Local variable read	`int var=0;` `var=`*var*`+1;`	`ALOAD, DLOAD` `FLOAD, ILOAD` `LLOAD`
Local variable write	`int var;` *var*`=var+1;`	`ASTORE, DSTORE` `FSTORE, ISTORE` `LSTORE`
Get value from array	`array[index]=` *array[index]*`+1;`	`AALOAD, BALOAD` `CALOAD, DALOAD` `FALOAD, IALOAD` `LALOAD, SALOAD`
Write value in array	*array[index]*`=` `array[index]+1;`	`AASTORE, BASTORE` `CASTORE, DASTORE` `FASTORE, IASTORE` `LASTORE, LASTORE`
Getting array lentgh	`array.`*length*	`ARRAYLENGTH`
Creating array	*new* `Object`*[3]*	`NEWARRAY` `MULTIANEWARRAY` `ANEWARRAY`
Returning	*return* `null`	`ARETURN, DRETURN` `FRETURN, IRETURN` `LRETURN, RETURN`

Information

	Static information driving the macro expansion Information given to `Alterator` Typed as `String`	**Dynamic information passed to the macro** at runtime Typed as `Object[]`	Index and type
Message sent	qualified name of invoked method	qualified name of invoked method	0: `String`
		qualified name of altered method	1: `String`
		instance of invoker	2: `Object`
		instance of invoked	3: `Object`
		values of arguments	4: `Object[]`
Constructor	qualified name of constructor	qualified name of invoked constructor	0: `String`
		qualified name of altered method	1: `String`
		instance of invoker	2: `Object`
		argument values	3: `Object[]`
Casts	qualified name of destination type	qualified name of destination type	0: `String`
		qualified name of altered method	1: `String`
		instance of altered object	2: `Object`
		casted instance	3: `Object`
Field read	qualified name of field read	qualified name of field read	0: `String`
		qualified name of altered method	1: `String`
		instance of altered object	2: `Object`
		instance on which the fied is read	3: `Object`
Field written	qualified name of field written	qualified name of field written	0: `String`
		qualified name of altered method	1: `String`
		instance of altered object	2: `Object`
		instance on which the field is written	3: `Object`
		new field value	4: `Object`

An example

- I State of the art
- II Naive
- **III An example**

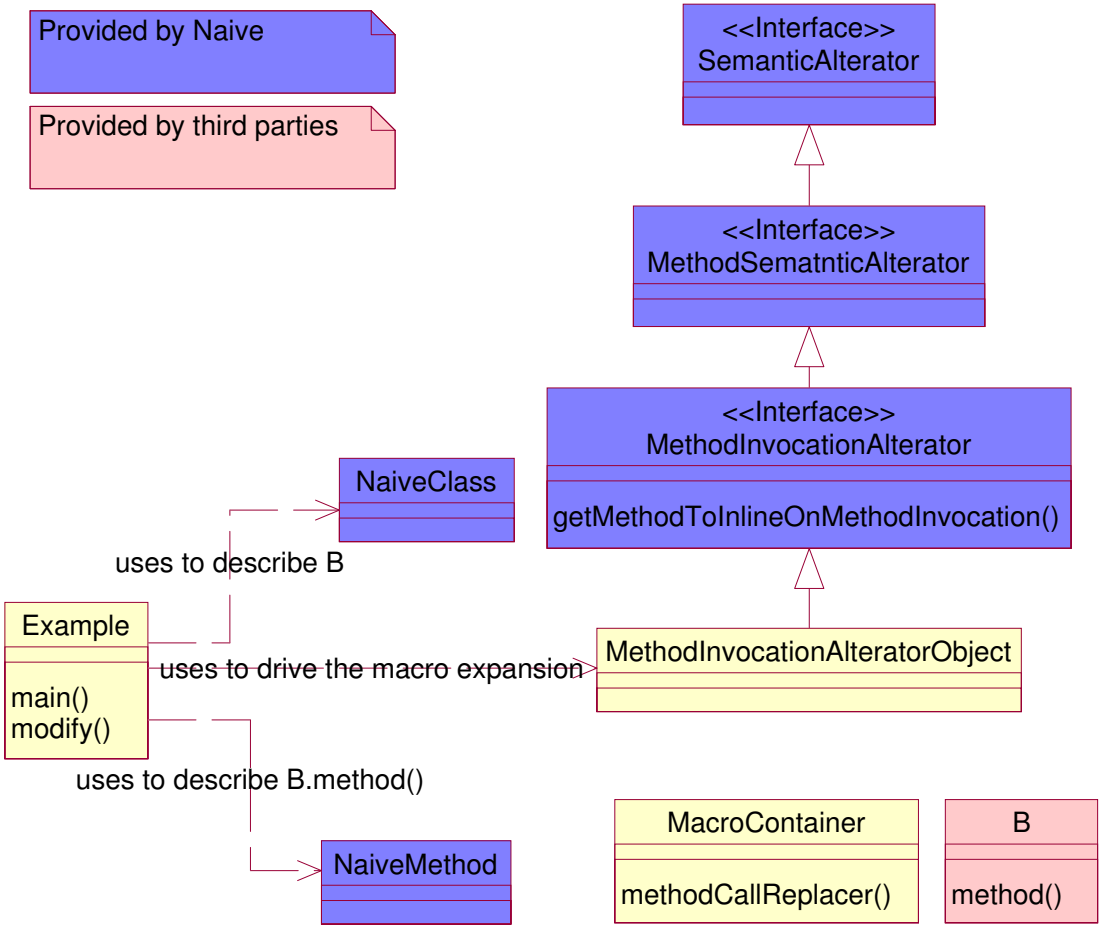
The original code

```
class B {
    public void method(Enumeration e) {
        while(e.hasMoreElements()) {
            A anA = (A) e.nextElement();
            anA.print(System.out);
        }
    }
}
```

What we would like

```
class B {
    public void method(Enumeration e) {
        while(e.hasMoreElements()) {
            A anA = (A) e.nextElement();
            anA.print(System.err);
        }
    }
}
```

Static structure



Bootstrapping

```
package naive.thesisexamples.methodinvocation;
public class Example  public static void
main(java.lang.String[] args)  modify(); //do the
modifications
public static void modify()  NaiveMethod method =
NaiveMethod.getNaiveMethod(
"naive.thesisexamples.methodinvocation.B.method()" );
MethodInvocationAlteratorObject alterator = new
MethodInvocationAlteratorObject();
method.accept(alterator); try  NaiveClass naiveClass =
method.getDeclaringClass(); naiveClass.save();
catch(IOException exp)  System.err.println(exp);
```

Driving the macro expansion

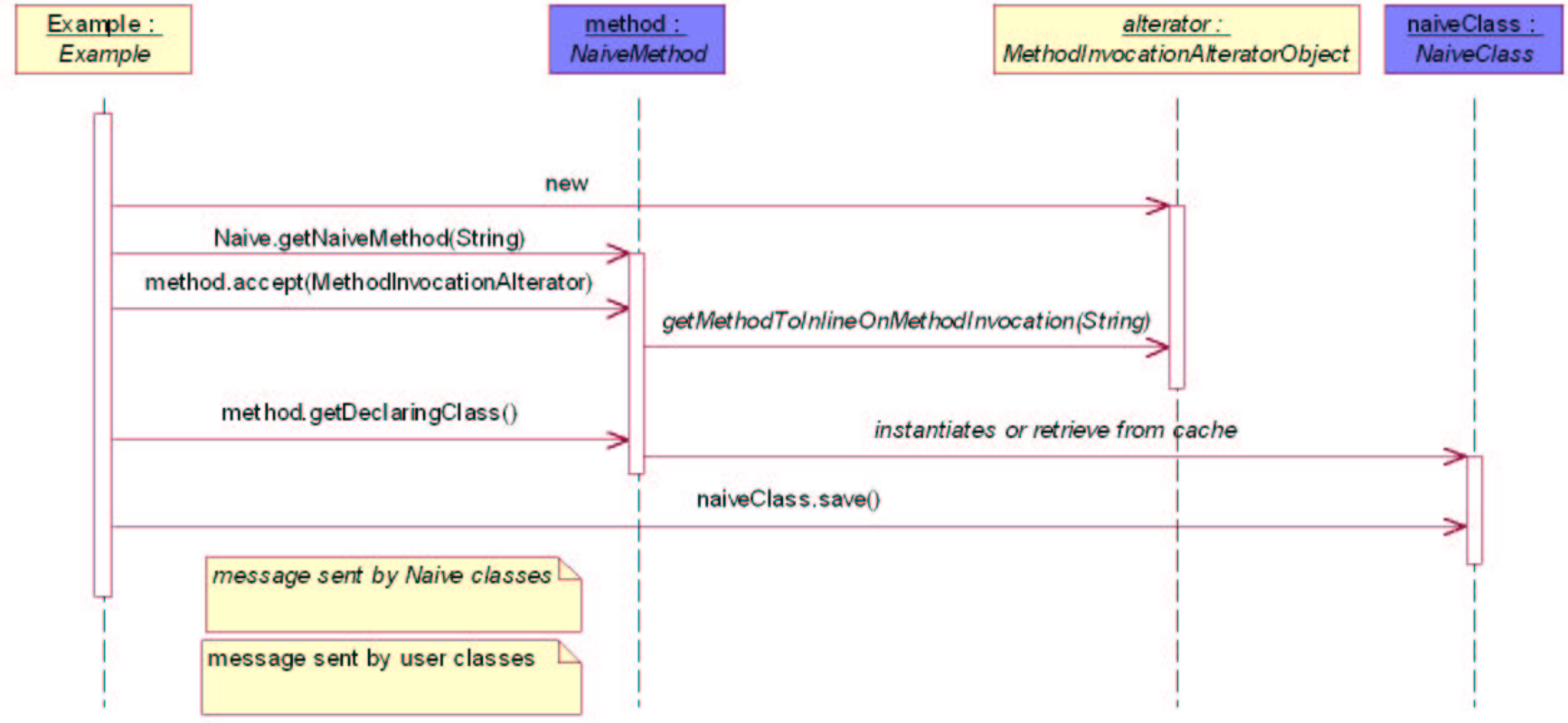
```
; public class MethodInvocationAlteratorObject
        implements MethodInvocationAlterator {
    public String getMethodToInlineOnMethodInvo
    String inReplacementOfMethodCall
                                ) {
        if(inReplacementOfMethodCall.equals
            ("naive.thesisexamples.
            methodinvocation.A.print(PrintStrea
           ) {
            //do a macro expansion
            return "naive.thesisexamples.metho
                    MethodBodyContainer.
                    methodCallReplacer(Object[
                    ";
            }
        return null; //no alteration
```

The macro itself

```
    \begin{verbatim}
package naive.thesisexamples.methodinvocation;

public class MethodBodyContainer {
    public Object methodCallReplacer(
    Object[] args) {
     //extract the message receiver (A)
        A anA = (A)(args[3]);
//perform the call
        A.print(System.err);
        return null;
    }
}
```

Example summary



conclusion