Integration of BETA with Eclipse

eTX presentation Barcelona 2004

Peter Andersen

Mads Brøgger Enevoldsen

Ole Lehrmann Madsen

Powerful BETA IDE already available. - So why Eclipse?

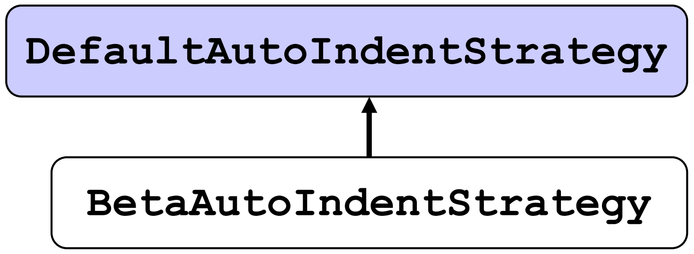
- Eclipse is gaining more and more users
- Programmers use several languages
  - Easier if the same IDE
- With language interoperability then a multi language IDE is a must
- Use of existing infrastructure
- Multiple platforms
- Reuse of tools produced for other languages
- Easier to maintain

Requirements

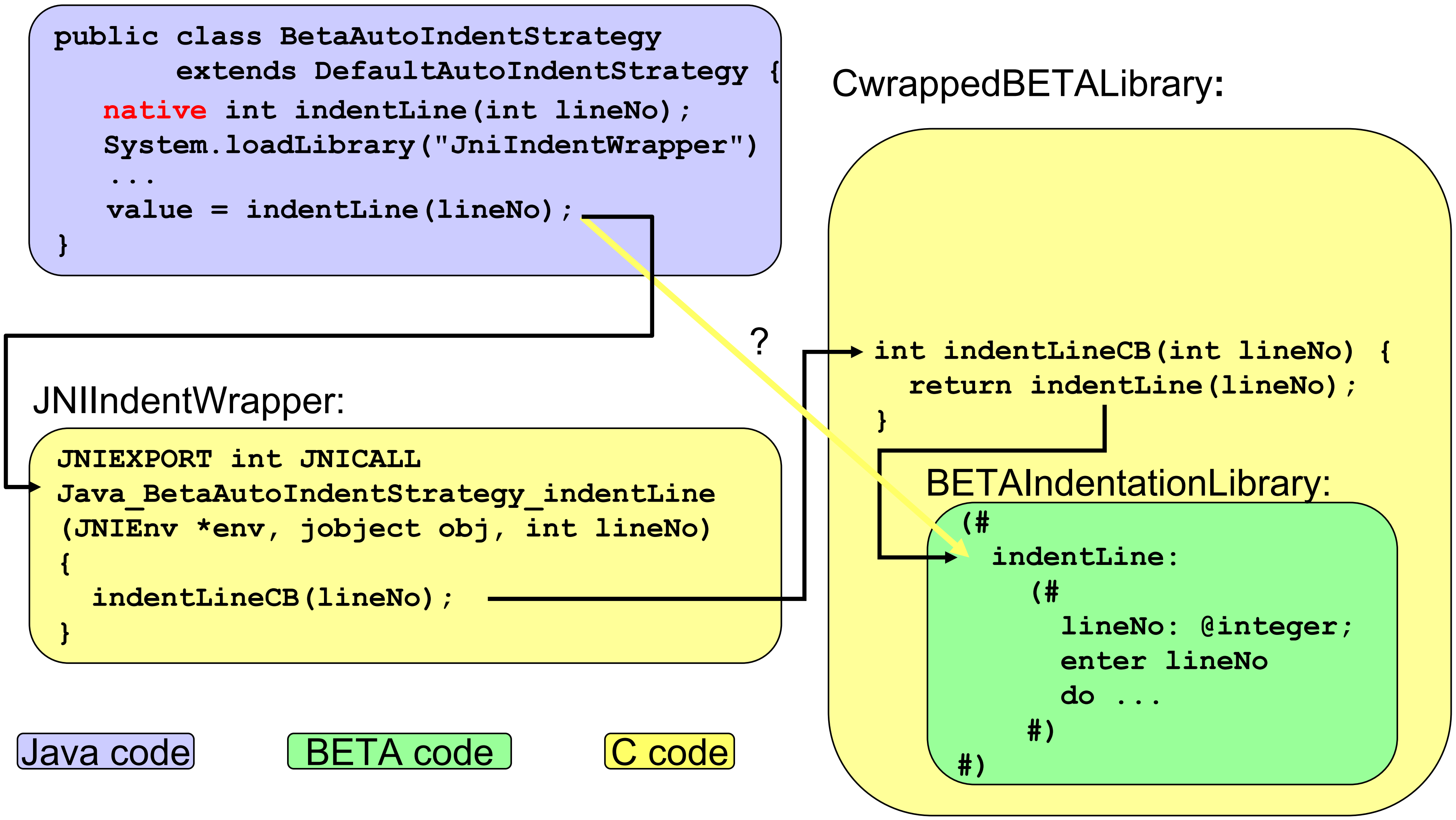
- Plug-ins for BETA should be implemented in BETA
  - To reuse code from BETA IDE
  - BETA programmers use BETA
- Eclipse plug-ins must be Java classes
- Interoperability between BETA and Java is necessary

Example: Add indent strategy

- Implement class `BetaAutoIndentStrategy` inheriting from `DefaultAutoIndentStrategy` from the Eclipse library
- Existing BETA code exists for analysing BETA source line and returning indentation level
- Two different approaches:
  - Java Native Interface (JNI) interoperability
  - Compiling BETA to Java bytecodes in `.class` files



JNI based BETA plug-in (complex!)



Java class:

```
public class BetaAutoIndentStrategy
       extends DefaultAutoIndentStrategy {
   native int indentLine(int lineNo);
   System.loadLibrary("JniIndentWrapper")
   ...
   value = indentLine(lineNo);
}
```

JNIIndentWrapper:

```
JNIEXPORT int JNICALL
Java_BetaAutoIndentStrategy_indentLine
(JNIEnv *env, jobject obj, int lineNo)
{
  indentLineCB(lineNo);
}
```

CwrappedBETALibrary:

```
int indentLineCB(int lineNo) {
  return indentLine(lineNo);
}
```

BETAIndentationLibrary:

```
(#
  indentLine:
    (#
      lineNo: @integer;
      enter lineNo
      do ...
    #)
#)
```

Java code

BETA code

C code

JVM Based BETA Plugin

- Alternative: Compile existing BETA code to Java bytecode
- Write `BetaAutoIndentStrategy` in BETA, inheriting directly from Java `DefaultAutoIndentStrategy`
- Requires mapping of BETA language to JVM using dedicated BETA compiler

BETA vs. Java

- Class and method unified in *pattern*
- General nesting of patterns, i.e. also of methods
- INNER instead of super
- Enter-Do-Exit semantics
- Genericity in the form of virtual patterns
- Multiple return values
- Active objects in the form of Coroutines
- No constructors
- No dynamic exceptions

The mapping

- Generating bytecode for JVM corresponds to making a BETA source mapping into Java source code
- Challenges for the mapping:
  - Must be complete
  - Must be semantically correct
  - Must be “nice”, i.e. classes generated from BETA must be understandable for Java programmers.

Naive mapping into Java .

```
Calculator:
 (# R: @integer;
    set:
      (# V: @integer enter V do V → R #);
    add:
      (# V: @integer enter V do R+V → R exit R #);
 #);
```

```
Class Calculator extends Object
{ int R;
  void set(int V) { R = V; };
  int add(int V) { R = R + V; return R;}
}
```

Naive mapping into Java ..

```
C: @Calculator; X: @integer;
12 → C.set;
5 → C.add → X
```

```
Calculator C = new Calculator(); int X;
C.set(12);
X = C.add(5);
```

Instances of add

- More complex mapping needed
- Possible to create instances of pattern add

```
Calculator:
  (# R: @integer;
       …
       add:
         (# V: @integer enter V do R+V → R exit R #);
   #);
C: @Calculator; X: @integer;
A: ^C.add;

&C.add[] → A[];
6 → A → X
```

Creation of an instance of C.add

Execution of the C.add instance

Inner class add

```
Class Calculator extends Object {
   int R;
   class add extends Object{
      int V;
      void Enter(int a) { V = a; }
      void Do() { R = R + V };
      int Exit() { return R; }
   }
   int add(int V){
      add A = new add();
      A.Enter(V);
      A.Do();
      return A.Exit();
   }
   …
}
```

```
Calculator:
  (# R: @integer;
     …
     add:
       (# V: @integer
        enter V
        do R+V → R
        exit R #);
  #);
```

Use of `add` as a class:

```
C: @Calculator;

X: @integer;
A: ^C.add;
&C.add[] → A[];
5 → A → X
```

```
Calculator C
   = new Calculator()
int X;
Calculator.add A;
A = C.new add();
A.Enter(5);
A.Do()
X = A.Exit();
```

Use of `add` as a method

```
C: @Calculator;

X: @integer;
5 → C.add → X
```

```
Calculator C
   = new Calculator()
int X;
X = C.add(5);
```

Not described here…

- **Inner call** mechanism – implemented by declaring new methods at each inheritance level
- **Virtual classes** – corresponding to generics (Java 1.5) – implemented with virtual instantiation methods and a lot of casting
- **Coroutines** and **concurrency** – implemented with threads
- **Pattern variables**: Classes and methods as first-class values – implemented with reflection
- **Leave/restart** out of nested method activations – implemented with exceptions
- **Multiple return values** – implemented with extra fields
- Use of **external classes** and **interfaces**
- Numerous minor details!

JVM Based Structure

BETA pattern:

```
BetaAutoIndentStrategy: DefaultAutoIndentStrategy
  (#
  do …
     lineNo -> indentLine -> value;
  #)
```

BETAIndentationLibrary:

```
(#
  indentLine:
    (#
      lineNo: @integer;
      enter lineNo
      do ...
    #)
#)
```

Eclipse plug-in code now written directly in BETA

Debugger Integration

- Existing BETA debugger hard to port to Eclipse
- Since we compile to JVM, perhaps use Eclipse JDT debugger for Java?
- Turns out to work! ☺
  - After some pain… ☹

Demo

Evaluation

- Two approaches for Eclipse plug-in writing using non-Java language:
  1. JNI: possible, but complex and error-prone
  2. JVM: Elegant, but hard work to compile your language to JVM.
     Allows for JDT reuse.
     - Problems with our JVM solution:
       - JVM code is **slow**, we have not yet looked at optimizations
       - BETA developers want to build native applications, but have to debug via JVM target.

Contacts: Questions?

- Mads Brøgger Enevoldsen
  mailto:brogger@daimi.au.dk
- Peter Andersen
  mailto:datpete@daimi.au.dk
- Ole Lehrmann Madsen
  mailto:olm@daimi.au.dk
- Download:
  http://www.daimi.au.dk/~beta/eclipse
  http://www.daimi.au.dk/~beta/ooli