Extending modularity of object oriented languages.

Modular initialization, and

Hygienic identifiers

Viviana Bono i Jarosław Kuśmierek

26 maja 2007

Systems grow larger and larger, so:

- Almost no system is build from scratch. Every system is build on top of another libraries;
- Almost no system is written, deployed and left as it is. Every system evolves, and is modified;
- Many systems are deployed on different sites, in different context, of different systems.

Crucial properties of an Object-Oriented language as a tool of building a real-world system:

- maximal flexibility of composition of modules
- easy maintenance of code
- resilence to versioning

Two of the flaws compromising "good properites"in languages:

- cumbersome initialization protocol via constructors.
  Subclass needs to repeat all the constructors, creating unnecessary dependencies. Results of changes in base classes are not satisfactory without a respective changes in subclasses.

- lack of hygiene of declarations.
  Many constructs are ambiguous, so their meaning can unexpectedly change due to versioning.
  Very hard to state formal properties about some parts of the system (containing dependencies). Negative assumptions required.

```
class Point           //three different ways for position
{ Point (float x, float y) ...
  Point (Complex comp)     ...
  Point (float angle, float rad, boolean PolarDef) ...
}

class ColorPoint extends Point  //two options for color
{ ColorPoint (float x, float y, float r, float g, float b)
  ColorPoint (float comp, float r, float g, float b)
  ColorPoint (float angle, float rad, boolean PolarDef,
              float r, float g, float b);

  ColorPoint (float x, float y, float c, float m, float y, float k)
  ColorPoint (Complex comp, float c, float m, float y, float k)
  ColorPoint (float angle, float rad, boolean PolarDef,
              float c, float m, float y, float k)
}
```

Modular object intialization protocol

Idea:

- one module corresponds to one option of initialization of independent set of object fields;
- subclass only declares “modifications” with respect to its superclass;
- client code constructing object, declares only the information it can supplied - rest is deduced.

```
//JavaMIP approach:
class Point
{ float x,y;

  required Point (float x, float y) intializes ()
  { new []; this.x = x; this.y = y; }

  optional Point (Complex comp) initializes (x,y)
  { new [x:=comp.x, y:= comp.y]; }

  optional Point (float angle, float rad) initializes (x,y)
  { new [x:=cos(angle)*rad, y:=sin(angle)*rad];}
}
```

```
class ColorPoint extends Point
{ float r, g, b;

  required ColorPoint (float r, float g, float b) initializes ()
  { new [];
    this.r = r; this.g = g; this.b = b; };

  optional ColorPoint (float c, float m, float yc, float k)
     initializes (r,g,b)
  { float R1=....; float G1=...;  float B1=...;
    new[r:=R1, g:=G1, b:=B1];
  }
}

new ColorPoint [angle := 0.7, rad:=4, c := 0, m:= 1, yc:=1, k:=0];
```

Implementation of JavaMIP exists and is efficient:

- In cases which are easy to optimize (indepndant modules : cartesian product) there is no different in performance.
- Creation of 1 milion of `ColorPoint` objects, in JavaMIP is 41% slower than in Java (688 ms vs 485 ms).

Approach is usefull (some metrics of code):

	constructors		**parameters**	
class	Java	JavaMIP	Java	JavaMIP
`java.util.Formatter`	14	5	23	7
`java.math.BigDecimal`	18	8	35	14
`javax.swing.JCheckBox`	8	4	12	4
`java.awt.Dialog`	14	7	34	7
`java.sql.Exception` (+ 13 subclasses)	120	4 or 1	224	4

Higienic identifiers and their properties...

Idea:

Distinction between this notions:

- Method introduction
- Method implementation
- Method call

And precise binding between those

```
package p;

class A
{ String getName();
  implement String p.A.getName() {...}
}
class B extends A
{ implement String p.A.getName()
  { ...; super (); ... }
}
class C extends A
{ String getName();
  implement String p.C.getName() {...}
}

C obj = new C();
obj.p.A.getName();
obj.p.C.getName();
```

The issue is serious...

Number of occurences in Java RT library code:

method	**introductions in**		
	interf.	**classes**	**total**
getName()	59	148	207
getType()	36	71	107
close()	30	38	68
getLength()	28	33	61
getValue()	24	45	69
item(int)	19	12	31
getId()	20	32	52
setName(String)	15	7	22
getAttributes()	13	33	46
remove(int)	6	19	25
setValue(String)	9	2	11
getWidth()	13	17	30
clear()	8	61	69
isEmpty()	8	32	40

isEmpty a funkcja charakterystyczna

- interfejs java.util.Set
- klasa java.util.Dictionary

clear

- interfejs java.util.List
- interfejs java.util.Map
- klasa java.awt.List

close

- interfejs java.sql.Connection
- interfejs java.sql.ResultSet
- klasa java.net.Socket

setValue

- interfejs java.awt.Adjustable
- klasa javax.swing.JProgressBar