JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©**JOT, 2002**

Vol. 6, No. 9, Special Issue: TOOLS EUROPE 2007, October 2007

Modularizing constructors

Viviana Bono, Dept. of Computer Science, Torino University
Partly supported by MIUR Cofin '06 EOS DUE project
Jarosław D. M. Kuśmierek, Institute of Informatics, Warsaw University
Partly supported by the Polish government grant 3 T11C 002 27
and by SOFTLAB - Poland, Warsaw, Jana Olbrachta 94

Object-oriented class-based languages provide mechanisms for the initialization of newly created objects. These mechanisms specify how an object is initialized and what information is needed to do so. The initialization protocol is usually implemented as a list of constructors. It is often the case that the initialization protocol concerns some orthogonal properties of objects. Unfortunately, if those properties have more than one option of initialization, the total number of constructors becomes exponential in the number of properties.
In this paper, we propose an alternative protocol. In our approach, instead of defining a list of constructors, it is possible to split blocks of definitions into smaller and composable pieces, in order to obtain reduction in the size of the code, better reusability, more expressiveness, and easier maintenance.

1 INTRODUCTION

Most object-oriented class-based languages are equipped with some form of specifications for the object initialization protocol. This protocol describes two aspects of the initialization:

- What kind of information must be supplied to a class, to create and initialize an object. A class may support more than one variant of object initialization, which means that there may be more than one accepted set of such information.
- What code is executed during this initialization. The sequence of instructions which should be executed depends on the kind of information supplied, therefore if the class supports distinct sets of information to be supplied during the initialization, then, for any such a set, a different sequence of instructions should be executed.

Usually, initialization protocols are specified by a list of *constructors*. Each constructor corresponds to one accepted set of information and consists of:

- a list of parameters (names and types), specifying a set of information required to initialize an object;
- a body, containing a list of instructions which must be executed in order to initialize an object.

Unfortunately, this traditional initialization protocol (abbreviated with *TIP* from now on) has some drawbacks: it requires additional copying of code, and it causes unnecessary dependencies. Therefore the code is harder to maintain and more vulnerable to the versioning of libraries. Disadvantages of TIP are discussed in some detail in the next section.

Our approach is based on the modularity and the composability of initialization blocks, in order to increment expressiveness and flexibility, and to reduce the size of the code. Our initialization protocol is presented by introducing a new Java-like language called JavaMIP. We present JavaMIP by giving some examples, then by defining its semantics by a translation into Java.

The paper is structured as follows. Section 2 discusses a list of problems induced by the TIP approach and some solutions existing in actual languages implementations and/or proposed in the literature. Section 3 introduces JavaMIP, first via a simple example, then by presenting its syntactical constructs and its semantics by translation into Java. Section 4 comments on the performance of JavaMIP. Section 5 summarizes our approach and makes some comparisons with related work. Section 6 concludes the paper presenting some future work directions.

2 BACKGROUND

Problems with the most common approach

The TIP approach is present in most of the contemporary object-oriented languages like, for instance, C++ [18], C♯ [10], Delphi [1], Java [9]. The main limitation of such an approach is the following: every time a programmer wants to reference an initialization protocol of a class, he must define or reference explicitly the whole list of parameters, or, in the case of inheritance, sometimes even repeat the whole list of constructors.

Nowadays large software systems, equipped with equally large libraries, are developed. If, on the one hand, the typical inheritance mechanisms scale well with respect to reusing properties and methods of a class, on the other hand, they do not scale as well with respect to reusing the initialization protocol, that is, the constructors.

We will analyze in more depth different flaws of TIP. We perform the analysis by exploiting the Java syntax and some Java examples; however, most of the problems occur in all of the mainstream object oriented languages.

TIP leads to an exponential number of constructors with respect to the number of optional parameters. In classes like `java.awt.TextArea`, there is often one "full" constructor declared, containing the largest set of parameters. However, when some of those parameters are optional, then additional constructors, with less parameters than the "full" one, must be declared.

Moreover, most of the time, all desired constructors cannot be introduced, because it is not allowed to place, in one class, two constructors with parameter lists of the same length and compatible types of the corresponding parameters. Even though such two constructors can represent two different subsets of parameters of the full one, the compiler will not recognize which constructor to call at object creation time, because the constructor is chosen via the types of its arguments. However, if the compiler could make such choice, then most probably a lot of classes would contain all possible constructors, leading to an exponential number of constructors.

TIP leads to an exponential number of constructors with respect to the number of properties with different initialization options. If a class contains some properties, where each of them can be initialized in more than one way, then the possible number of initialization options (thus the number of constructors) is a multiplication of the numbers of options of object properties. An example could be the combination of a property color (with two options, RGB and CMYK) with a property position (with three options, Cartesian, polar, and complex) in a class `ColorPoint`, which induces six constructors. The Java class `java.net.Socket` is a more sophisticated example that contains nine constructors, due to this and to the previous described problem.

TIP forces unnecessary code duplication. Let us assume that we have two attributes characterizing the state of an object, for example the usual position and color from a class `ColorPoint` (see Figure 1 on page 305). If those attributes can be initialized in multiple ways (the position can by supplied by three different coordinate systems and the color by two different palettes), then we would need six constructors where most of pairs of those constructors share some common code (see the Java example in Section 3).

TIP makes the extension of a class by a new subclass cumbersome. In most cases, the designer of a subclass wants only to modify the initialization protocol of a parent class, not to redefine the parent's protocol completely. However, using the TIP approach, he must declare the whole resulting protocol in a subclass. Let us consider, for example, a class of blinking buttons, which blink for some time after clicking. It is possible to declare it by extending the class of ordinary buttons, for example `javax.swing.JButton`, which has five constructors. Some information is needed for such a button, for instance the frequency of blinking and the time for which the button must blink. Therefore, the subclass must contain five constructors, as the parent class. Additionally, the declaration of each of those constructors will begin with the identical parameters of the corresponding constructor in the parent class, but it will contain two additional parameters, time and frequency. This may lead also to code duplication.

Modifications of existing classes force unnecessary changes in subclasses. Let us imagine such a situation:

- there exists class C_1 with some constructors;
- another class C_2 is declared by another developer as a subclass of C_1. Class C_2 needs some additional initialization information, so it has all parent constructors redeclared by adding one parameter *Par'*, which is needed by subclass C_2. Then, all those constructors will most probably call the corresponding constructors in the base class C_1 and execute some sequence of code concerning *Par'*;
- another constructor is added to C_1. Unfortunately, the class C_2 will not inherit automatically the corresponding constructor. This class, depending on the language design, will either retain the original constructor without the additional parameter *Par'*, or it will not inherit it at all (as it is in case of Java), but it will never inherit the modified version (with the extra parameter) of this constructor, which could be

instead the intention of the subclass designer.

Overloaded constructors make safe-looking changes non-conservative. When a Java (or C++ or C♯) class contains many different options of initialization implemented as many different constructors, the choice of the constructor actually called during object creation is done in the same way as the choice of an overloaded method variant. Thus, it suffers the same problems, as this example shows:

```
interface I1 {...}
interface I2 {...}
class C1 implements I1, I2 {...}
class C2 implements I2 {...}

class ClassWithOptions
{   ClassWithOptions (I1 a, I2 b);
    ClassWithOptions (I2 a, I1 b);
}
 new ClassWithOptions(new C1(), new C2());
```

This code will compile, because only one of constructors of class `ClassWithOptions` matches the last `new`. But if the class `C2` is enriched in order to make it implement also the interface `I1`, then this "safe-looking" change will make the previous code not working because the last `new` will be ambiguous. Notice that this is not a problem in languages in which it is possible to name constructors (e.g., in Delphi), therefore the overloading can be avoided.

In languages with checked exceptions TIP forces unnecessary dependencies. In Java, if a constructor of a parent class throws some exceptions and it is called by a constructor of a subclass, the subclass constructor must repeat the whole list of exception declarations. While the repetition of such declarations in normal methods is important, because methods have choices (to catch the exceptions, or throw them further), the situation with constructors is different. A constructor cannot catch the exceptions of a superclass, therefore the repetition of the declarations is a not justified additional work for the programmer, and even worse: it creates unneccessary dependencies.

As an effect, when a superclass constructor is refined by replacing one exception with two other subexceptions of the original one (which is a conservative extension), this refinement is not visible in the subclasses.

Traditional mixins also suffer the same problems. A mixin is a class parameterized over a superclass, that was introduced to model some forms of multiple inheritance and improve code modularization and reusability, [17, 4, 8, 2]. Mixins can be combined with other mixins and/or applied to a class to obtain a fully-fledged subclass; the class argument plays the role of the parent. A mixin declaration, like any other subclass declaration, may contain declarations of new constructors [1], which, in turn, may reference the superclass constructors.

Those dependencies between constructors cause similar problems as within classes, and constrain the number of possible mixin applications. Therefore, while, on the one

[1]In fact, not all mixin-related proposals allow some form of constructors.

hand, mixins are designed for a wider reuse than classical subclasses, on the other hand, the problems with initialization protocol may occur more often. Also the designers of Jam [2] have noticed this problem. In order to simplify the matter, they decided to disallow the declarations of constructors in mixins, thus forcing programmers to write constructors manually in all classes resulting from mixin application.

Some proposals to solve the initialization problems

There are at least six techniques which designers use to solve some of the problems concerning initialization:

1. by avoiding explicit initialization protocols, with the optional support of some formal specification languages,
2. by declaring a constructor with one parameter of type "container" containing all the initialization parameters (such as, for example, of type `Vector` or of type `Dictionary`),
3. by using default parameter values,
4. by referencing parameters by name,
5. by using the constructor propagation mechanism of Java Layers, [5],
6. and by using the object factories mechanism.

The first two solutions are, in fact, programming techniques which can be used in most object-oriented languages, while the last four ones are actual language features implemented in existing languages or languages extensions. In the following section, we will discuss those solutions in more detail.

Solution 1. A class written using the approach of avoiding explicit initialization protocols contains one parameterless constructor (or none), while the real initialization process is implemented in a list of ordinary methods. Those methods must be called on objects explicitly after their creation. Here is an example. Instead of declaring the following class:

```
class ColPoint {
  ColPoint(int x,int y,int R,int G,int B)  {...}
  ColPoint(int x,int y,int C,int M,int Y,int K)  {...}
 }
```

one might declare a class with a parameterless constructor, and a set of methods responsible for the initialization:

```
class ColPoint
{ ColPoint();
  void setPosition(int x, int y)
  void setColRGB(int R, int G, int B)
  void setColCMYK(int C, int M, int Y, int K)
}
```

In a program written in such a manner, a class may contain many small methods, each of them responsible for different options of a different layer of the initialization, so that enough level of modularity is achieved. However, a programmer using this class has to figure out by himself, consulting some documentation, how to initialize its objects properly, in order to establish the object invariants. Additionally, it will not be statically verified by the compiler if an object is properly initialized and if the object invariants are established. As an effect, the programmer may create an object from the class and make any of the following errors: (*i*) forget to call some of the methods responsible for the initialization; (*ii*) call too many of them; (*iii*) call them in an incorrect order.

Existing formal specification languages, like JML, [13] and the Design by Contract in Eiffel, [15], can allow some verification of this protocol. However, such approaches require to specify additional assertions, which is, in general, a difficult and time-consuming task. Additionally, due to the general nature of assertions and to a great freedom of possible combinations, those assertions can be checked during the execution of the program, but they cannot be verified statically. A statical verification is sometimes possible with the support of theorem provers, but usually such tools require some manual support from the programmer.

Solution 2. Constructors may have one parameter of a "container" type, such as `Vector` or `Dictionary`. The container structure will contain values of all the initialization parameters, for instance indexed by their names. Then, a constructor may perform a dynamic verification inside, like in the following example:

```
class ColorPoint
{ ColorPoint (Dictionary d)
  { if ( (d.get("R")<>null) && ...)       {... /* process RGB data */ ...}
    else if ((d.get("C")<>null) && ...)   {... /* process CMYK data */...}
    else                       throw new Exception ("Wrong parameters!!");
  }
}
```

This approach has the following disadvantages:

- the class contains one constructor which must perform a lot of checks for miscellaneous cases of the initialization process;
- it does not allow any static checking of the parameters.

Solution 3. There is another mechanism which, in principle, was not designed to solve those problems, but can be used to solve one of them: methods and constructors with *default* parameter values (which is present, for example, in Delphi [1] and C++ [18]). Thanks to this mechanism, it is possible to declare less constructors, being possible to treat some parameters as optional. This mechanism does not help, though, when it is necessary to have a mutually exclusive choice among different parameters (of different types).

Solution 4. Another feature which can be found in some languages (but not in any of the main-stream ones like Java and C♯), which may help, is referencing parameters of methods and constructors by their names, that is, not necessarily by passing the actual parameter values in the order in which they are declared. Such an approach, present for example, in Flavors [17], Objective-C [11], and Ocaml [14] solves two problems:

- it discards some of the ambiguities caused by constructor's overloading, because parameters with the same (or compatible) type can have different names;

- it allows a wider use of default parameter values because normally we can use default values only for a sequence of parameters being a suffix, while in this approach it is possible to leave as default any of the available parameters.

However, this feature only solves problems of optional parameters and discards some ambiguities, but does not prevent an exponential number of constructors and code duplication in the case of multiple options of initialization of orthogonal object properties.

Solution 5. The Java Layers language [5], has a feature called "constructor propagation", which can be illustrated by this example [2]:

```
class Class1
{ propagate Class1(String s) {I_1;}
  propagate Class1(int i)    {I_2;}
}
class Class2<T> extends T
{ propagate Class2(int j)    {I_3;}
}
```

With such declarations the class `Class2<Class1>` would in fact have the same list of constructors as the following class:

```
class Class3
{ Class3 (String s, int j) {I_1; I_3;}
  Class3 (int i   , int j) {I_2; I_3;}
}
```

This approach solves the problem of the exponential number of constructors if the sets of options of parameters are in different classes. However, this can only do a Cartesian product of sets of constructors from different classes. Therefore, if we want to write a subclass of a class `C` with the purpose of adding another option for initializing the existing set of properties declared in `C`, then we cannot do this using Java Layers. This will be re-addressed with an example in Section 5, in order to be compared directly with our approach.

Solution 6. A recent work concerning initialization protocols is [6]. It shows how object factories can be integrated in a language like Java in a way they use the same syntax as for normal object creation. Using this approach, it is possible to override the constructors of a class, and even to instantiate an interface. As an effect, in some situation this reduces the amount of code which needs to be written. However, in this approach, each allowed set of initialization parameters must correspond to one constructor, therefore, in general, it does not avoid the problem of the exponential growth.

3 THE JAVAMIP APPROACH

In order to solve the above mentioned problems, we developed an approach based on the idea of splitting big constructor declarations into smaller and composable parts, the *ini-*

[2]The example is a modified version of an example taken from the web page of the Java Layers `http://www.cs.utexas.edu/~richcar/cardoneDefense.ppt`. Also the syntax is slightly modified to look more Java-like.

tialization modules, called also from now on *ini modules*. The decomposition mechanism will still allow the static verification of declarations of the required parameters and of the object creation expressions.

This approach is implemented as a part of a larger project, the design of a new mixin-based language called Magda, but in order to study the main ideas and effects of our approach we start by extending the Java language[3] into *JavaMIP*.

A Java-like example

We illustrate the main ideas of our approach by providing an example, first in Java (in Figure 1), then in JavaMIP (in Figure 2).

In the Java version, the class `ColorPoint` suffers from the fact that two orthogonal aspects of the object creation must be put into every constructor. Notice that, if we had abstracted the "color" property by creating a mixin, then we could not apply such a mixin to a `3DPoint`.

In Figure 2 there is the same example written in JavaMIP, with the initialization protocol is split into"pieces". When the line `new ColorPoint[angle:=0.7, rad:=4, c:=0, m:=1, y:=1, k:=0]` is executed, an object of class `ColorPoint` is created and then the initialization protocol proceeds as follows:

1. the ini module `M5`, dealing with the optional parameters `c,m,y,k` of `ColorPoint`, is called;
2. after converting the colors in RGB, `M5` calls the `M4`, the "main" ini module of `ColorPoint` (that actually stores the color in the state variables),
3. which, in turn, invokes the appropriate parent ini module, which, in this case, is `M2`, the one for polar coordinates, which are translated into Cartesian ones;
4. this will eventually call the `M1`, which is the "main" ini module of the class `Point`, that actually stores the Cartesian coordinates in the state variables.

The initialization of the object is therefore performed in a modular fashion. In a class, instead of a list of constructors, we have a list of ini modules introduced by a syntactic construct `initializes`. Each ini module contains a `new [...]` instruction, which either calls another ini module in the same class, or another ini module from the superclass. For instance, an intermediate call to an ini module would be performed if we add another ini module in the `ColorPoint` class: `ColorPoint (z) initializes (c,m,y,k) {...}`. Then an object created via `new ColorPoint[angle := 0.7, rad:=4, z:=...]` will be initialized first by executing the ini module concerning `z` and then the computation will go on as described above.

Therefore, class `ColorPoint` written in the JavaMIP language: (*i*) is simpler; (*ii*) does not dependent on the number of parent ini modules; (*iii*) does not contain any code duplication; (*iv*) can be easily abstracted to extend a `3DPoint` class with a property color.

[3]We could also apply the same changes to C++ and *C*♯, or even to Jam [2] and MixedJava [8]. The differences would be insignificant.

```
// Class of points definable by three different coordinate systems
class Point
{ float x, y; //object state variables

  Point (float x, float y)
  { this.x = x; this.y = y; }

  Point (Complex comp)
  { x = comp.x;  y = comp.y;  }

  //the third parameter is required only for the compiler
  //to distinguish between this one and the (x,y) constructor
  Point (float angle, float rad, boolean PolarDef)
  { x:=cos(angle)*rad; y:=sin(angle)*rad; }
}

// Class of colored points whose color is definable by two different
// color palettes
class ColorPoint extends Point
{ float r, g, b; //object state variable

  //Here are constructors (for three different coord. systems) with RGB
  ColorPoint (float x, float y, float r, float g, float b)
  { super(x,y);
    this.r = r; this.g = g; this.b = b;}
  ColorPoint (float comp, float r, float g, float b)
  { super(comp);
    this.r = r; this.g = g; this.b = b;}
  ColorPoint (float angle, float rad, boolean PolarDef,
              float r, float g, float b)
  { super(angle, rad, PolarDef);
    this.r = r; this.g = g; this.b = b;}

  //While here are constructors  with CMYK
  ColorPoint (float x, float y, float c, float m, float y, float k)
  { super(x,y);
    r = somefun1(c,m,y,k); g = somefun2(c,m,y,k); b = somefun3(c,m,y,k);}
  ColorPoint (Complex comp, float c, float m, float y, float k)
  { super(comp);
    r = somefun1(c,m,y,k); g = somefun2(c,m,y,k); b = somefun3(c,m,y,k);}
  ColorPoint (float angle, float rad, boolean PolarDef,
              float c, float m, float y, float k)
  { super(angle, rad, PolarDef);
    r = somefun1(c,m,y,k); g = somefun2(c,m,y,k); b = somefun3(c,m,y,k);}
}
```

Figure 1: `Point` and `ColorPoint` – Java code

```
class Point
{ float x,y;

  // M1 - main "ini module" for Point
  required Point (float x, float y) initializes ()
  { new []; this.x = x; this.y = y; }

  // module declaring that comp can be supplied instead of (x,y)
  optional Point (Complex comp) initializes (x,y)
  { new [x:=comp.x, y:= comp.y]; } //this instructions say how the
                                   //translation from comp to (x,y) is done

  // module responsible for translation of the polar coordinates
  optional Point (float angle, float rad) initializes (x,y)
  { new [x:=cos(angle)*rad, y:=sin(angle)*rad];}
}
//this is an expression creating a Point with polar coord:
 new Point [angle := 0.7, rad:=4];

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

  // M4 - main "ini module" for ColorPoint
  required ColorPoint (float r, float g, float b) initializes ()
  { new []; //calls the appropriate ini modules in the parent class
    this.r = r; this.g = g; this.b = b; };

  // M5 - optional "ini module",
  // allowing the initialization of color via CMYK palette
  optional ColorPoint (float c, float m, float yc, float k) initializes (r,g,b)
  { float R1=somefun1(c,m,yc,k);
    float G1=somefun2(c,m,yc,k);
    float B1=somefun3(c,m,yc,k);
    new [r:=R1, g:=G1, b:=B1]; //calls the main "ini module" of ColorPoint
  }
}
//this is an expression creating a ColorPoint with polar coord. and CMYK color:
 new ColorPoint [angle := 0.7, rad:=4, c := 0, m:= 1, yc:=1, k:=0];
```

Figure 2: `Point` and `ColorPoint` – JavaMIP code

The JavaMIP constructs

The new construct. As shown in the example in Section 3, the initialization protocol of an object is modularized. The code performing this task is included in different modules, some of which may be in the class the object is being created from, while some others are in the class' ancestors. The execution of those modules is controlled by the new construct, which has two forms:

- an expression containing a class name and a list of parameters equipped with their actual values via initialization expressions: new Class$_1$[p$_1$:=e$_1$, ..., p$_n$:=e$_n$]
- an instruction containing only a list of formal parameters equipped with their actual values via initialization expressions: new [p$_1$:=e$_1$,..., p$_n$:=e$_n$];

The first form is an expression and it is used when a Java new would be used, therefore it creates an object, but it also indicates the class Class$_1$ from which the initialization protocol will start to look for the appropriate first ini module, that will be chosen according to the list of parameters supplied.

The second form is an instruction and it dispatches the call to another ini module,[4] which can be either in the current class or in the parent class. The second form is therefore used inside the ini modules themselves, as we will soon see.

The initialization module. The initializes keyword introduces the new form of initialization module. It is named after the class (as in Java), and replaces the classical Java constructor. An ini module contains two lists of parameters and has the following syntax:

{required ‖ optional } Class$_1$ (T^1 p^1[=e^1],...,T^n p^n[=e^n])
initializes (p$_1$,..., p$_k$) [throws ...] {I$_1$; new[p$_1$:=e$_1$,..., p$_n$:=e$_n$]; I$_2$;},
where I$_1$ and I$_2$ may be empty.

The parameters in the first list are called *input parameters*, while the parameters in the second list are called *output parameters*. The input parameters are declared by the current ini module. The output parameters refer to input parameters already declared in other ini modules placed either in the same class and written above the current ini module, or placed in an ancestor class. Each input parameter name must be different from the names of input parameters of other modules in the hierarchy of a given class. Each input parameter can be associated with its default value denoted by the expression e^i. The keyword required denotes that the input parameters associated to an ini module must be supplied (by default or by the output parameters of other ini modules) during object initialization, thus the execution of the corresponding ini module is mandatory. The keyword optional denotes that those parameters are optional, thus the execution of the corresponding ini module is not mandatory. Default values of input parameters are only allowed in required modules. It is important to notice that the names used for the parameters of the initialization modules do not have any necessary relation to the field (instance variable) names.

Any of those lists of parameters can be empty:

[4]Except in the last ini module placed at the top of the hierarchy.

- The list of output parameters can be empty:
  `... Class`$_1$ `(T`^{1} `p`^{1}`,...,T`^{n} `p`^{n}`) initializes () {I`$_1$`; new []; I`$_2$`;}`.
  This can be utilized when there are some input parameters that are used during the object initialization, but they are not exploited to calculate any parameters of other ini modules. In the most common case, such input parameters will be used to initialize some of the fields of the created object.
- The list of input parameters can be empty:
  `required Class`$_1$ `() initializes (p`$_1$`,..., p`$_k$`)`
  `{I`$_1$`; new [p`$_1$`:=e`$_1$`,..., p`$_n$`:=e`$_n$`]; I`$_2$`;}`.
  This can be used in a situation when a class C has some initialization parameter, but these are not required by a subclass C': the designer might have decided that for C' some parameters can be, for instance, constants, or can be read from some configuration file (this will be done in the `e`$_i$). Such modules are always executed, therefore, for the uniformity of the syntax, we decide to always mark them as `required`.
- Both of those list can be empty:
  `required Class`$_1$ `() initializes() {I`$_1$`; new []; I`$_2$`;}`.
  This is used when a class must perform some actions not connected explicitly with the initialization of instance variables, like acquiring some resources, registering the object in some collections, initializing its subcomponents, etc.

The `throws` clause denoting a list of exceptions which can be thrown by a module has exactly the same syntax of the typical Java `throws` clause.

The list of supplied parameters of a `new` instruction is strictly defined by the context: its list of parameters must be equal to the list of the output parameters of the ini module in which this `new` occurs.

Intuitively, the ini module has four main tasks:

- a specification task: it declares new parameters (the input parameters), which must (or can) be supplied to a new expression at object creation time. Additionally, it specifies another list of parameters (the output parameters) that are considered supplied when the ones of the first list (the input ones) are supplied.
- a translation task: it supplies the code to translate the input parameters into the output parameters, which are the parameters for the next ini modules to be executed;
- an initialization task: it contains code initializing the state variables of an object by using the input parameters, or some other means;
- a control task: it dispatches the call to the next ini module to be executed via the `new` instruction.

Type checks, which must be performed to ensure the soundness of the above defined constructs, are left out of the present paper for lack of space, and can be found in [12].

A semantics for JavaMIP

The only differences between Java and JavaMIP concerns the object initialization protocol, therefore we define a semantics of JavaMIP via a translation function into Java, which, for most of the fragments of code, will be an identity function.

It is worth noticing that it is possible to write directly Java code in a style inspired by the code resulting from the translation of JavaMIP. However, such Java code could not be statically verified, as it is instead JavaMIP code, see [12]. In fact, the translation we present in this section leads directly to the implementation of a preprocessor *javamip2java* [16], which performs those static checks.

The main ideas of the translation are the following:

- every ini module is translated into two methods: the first method contains all the instructions from the ini module that are before the `new [...]` call, followed by instructions assigning the expressions supplied by the `new` to the corresponding parameters; the second method contains all the instructions that are after the `new [...]` call;
- every `new` expression creating a new object is converted into a sequence of calls to the ini modules (in fact, to those pairs of methods mentioned in the previous point);
- the values of the initialization parameters calculated by the current module to be passed later to another module will be held in additional *container* objects.

The reminder of the section presents the details of the translation. At the end of this section (on page 306), there is Figure 3 presenting the result of the translation of the `Point` class from Figure 2, which can be useful for understanding the translation details.

Passing of the initialization parameters. In order to pass the initialization parameters during the execution of the initialization process, we need, for every class, a special class of objects (called *containers*) used for passing the initialization parameters between the ini modules.

Therefore, for each JavaMIP class `C1` extending a class `C2` we declare a class `C1_Init`[5] as a subclass of the corresponding parent's class `C2_Init`, and for each input parameter of each ini module of class `C1` we declare a public field with the same type and name. For instance, for the JavaMIP class ColorPoint (see Figure 2) we have:

```
class ColorPoint_Init extends Point_Init
{ public float r, g, b;
  public float c, m, y, k;
}
```

Translation of the initialization modules. Every ini module declaration (let be the following one the *i*-th of its class):

$\texttt{Class (T}_1\ \texttt{g}_1\texttt{, ..., T}_k\ \texttt{g}_k\texttt{) initializes (f}_1\texttt{, ..., f}_n\texttt{) throws Ex}_1\texttt{, ..., Ex}_e$
$\{\ I_1\texttt{; new [f}_1\texttt{:=e}_1\texttt{, ..., f}_n\texttt{:=e}_n\texttt{]; } I_2\texttt{; }\},$

is substituted with two resultless methods.

The first method is named $\texttt{PreMethod}_i$, and it has the following parameters:

- the container of the initialization parameters, of type `Class_Init` and name `___init` (we assume that this name is not used elsewhere in the program). This will be used by this method to store the output parameters of the ini module;

[5]We assume that this will not cause any name clash.

- plus the input parameters of the ini module.

The second method, named `PostMethod`$_i$, has the same parameters as the input parameters of the ini module. Those methods have the following declarations:

```
void PreMethod_i(Class_Init ___init, T_1 g_1, ..., T_k g_k) throws Ex_1, ..., Ex_e
{ I_1;
  ___init.f_1 = e_1;
  ...
  ___init.f_n = e_n;
}
void PostMethod_i(T_1 g_1, .., T_k g_k) throws Ex_1, ..., Ex_e
{ I_2; }
```

In the resulting code, all ini modules are removed from the program, leaving only the Pre- and Post-methods. As the effect, in such translated code, no class has any constructor in the sense of Java.

Translation of an object creation expression. For this expression we must build a sequence of instructions which will perform the initialization process. Those instructions will actually: (*i*) create a new object and pass the initialization parameters which are supplied explicitly in the *Exp*; and (*ii*) call some Pre-methods and Post-methods in a specific order.

Therefore, we build a linear ordering on the ini modules triggered by the object creation expression (we call them “activated”, see the definition below), which will determine the order of the calls to Pre-methods. Post-methods will be called in the opposite order with respect to the one of Pre-methods. The order induces a dispatch resolution that is part of the translation of the object creation expression.

Let *Exp* = `new Class [p`$_1$`:=e`$_1$`, ..., p`$_n$`:=e`$_n$`]` be an object creation expression. We say that a ini module declared in `Class` or one of its superclasses is *activated* by *Exp* if each of its input parameters: (*i*) has a default value, or (*ii*) is one of the p$_1$ ... p$_n$, or (*iii*) is an output parameter of another ini module activated by *Exp*.

Then we can build a *construction order*, which is a linear ordering of the activated modules, called from now on $CO(Exp)$. The comparison relation for this order is “later”, with respect to the order of execution. We put in $CO(Exp)$:

- all the activated ini modules declared in `Class`. We put them into $CO(Exp)$ in an reverse order with respect to that of the declarations in the JavaMIP code;
- we execute recursively the same procedure for the direct superclass of the current class and append the result to $CO(Exp)$.

Notice that, if we have dependencies between modules, for example R_1 = `Class (T`$_1$` b, T`$_2$` c) initializes (a)` and R_2 = `Class (T`$_3$` d) initializes (b,g)`, where the common parameter is `b`, then R_2 will always be before R_1 in CO. Therefore it is possible to control the order of execution of the ini modules by changing their mutual order in the classes. Only combinations which do not make sense are disallowed.

The construction order determines the ordering within the initialization process. All Pre-methods corresponding to the activated ini modules will be executed following this order, and then the Post-methods will be called following the opposite order. Thus, those calls will be simulating the call stack of dispatch defined by the `new` calls placed inside the various ini modules.

Summarizing, every expression *Exp* =`new Class [...]` will be implemented by a sequence of instructions which will perform the following five tasks:

1. create an uninitialized object;
2. create an object responsible for passing parameters (the container);
3. set in the container all parameters passed explicitly (directly) by the *Exp*, with their respective values;
4. Each parameter which is not supplied explicitly by *Exp* is set in the container with its default, taken from the definition of the module in which the parameter is declared.
5. call the Pre-methods of the activated ini modules according to the construction order $CO(Exp)$, and passing to them the appropriate input parameters and the container;
6. call the Post-methods of the activated ini modules in the reverse order with respect to the construction order $CO(Exp)$, and passing to them the appropriate input parameters.

Since a `new Class [...]` expression can occur in a bigger expression, we must enclose all the above instructions in a method. Let us assume that for $Exp =$ `new Class[`p_1`:=` e_1`, ...,` p_k`:=` e_k`]`) we have $CO(Exp) = (cp_1, cp_2, ..., cp_n)$. Then, the effect of the translation of this expression from JavaMIP to Java will be the following method (placed inside the same class where the method containing *Exp* is declared):

```
static Class_1 UniqName (T_1 v_1, ..., T_k v_k) throws Ex_1, ..., Ex_e
{ Class_1 obj=new Class_1();          //1.  creates an non-initialized object
  Class_1_Init i=new Class_1_Init();  //2.  creates container for parameters
  i.p_1=v_1; ...  i.p_k=v_k;          //3.  sets all explicit parameters
  i.p_{d_1}=e_1; ...  i.p_{d_l}=e_l;  //4.  sets all default-supplied
  obj.PreMethod_{cp_1}(i, i.p_1^1, ..., i.p_1^{k_1}); //5.  calls the Pre-methods
  ...
  obj.PreMethod_{cp_n}(i, i.p_n^1, ..., i.p_1^{k_n});
  obj.PostMethod_{cp_n}(i.p_n^1, ..., i.p_1^{k_n}); //6.  calls the Post-methods
  ...                                // with the same parameters as Pre-
  obj.PostMethod_{cp_1}(i.p_1^1, ..., i.p_1^{k_1});
  return obj;
}
```

where the set `Ex`$_1$, ..., `Ex`$_e$ is the set of all the exceptions declared to be thrown by $CO(Exp)$. Moreover, the `new Class`$_1$`[`p_1`:=`e_1`,...:=` e_k`]` expression is replaced by the expression *`UniqName`* `(`e_1`, ...,` e_k`)`.

The result of the traslation of class `Point` can be found in Figure 3. It is easy to see that, in the case of such simple classes, some of the genarated methods are empty and therefore unnecessary. In fact, the implementation of the language [16] eliminates such empty methods and the respective calls.

4 EMPIRICAL EVALUATION

Some tests on JavaMIP were performed by exploiting the preprocessor *javamip2java*, which is a translator from JavaMIP to Java (see [16]) and it is based on the semantics by translation we presented. The preprocessor also performs the static checks described in [12], ensuring that each `new` expression creates a fully initialized object, and that the whole initialization process is confluent (a formal proof of a confluence property can be found in [3]).

Larger examples of code, presenting in detail some benefits of the modularization, can also be found at [16]. In particular, our performance benchmarks show that the object creation using ini modules of JavaMIP is in between 0% and 40% slower than using Java traditional constructors, depending on the complexity of the example.

Additionally, it is worth noticing that, if the modular initialization protocol were embedded in the language itself, it would be significantly faster. For example, the `_Init` does not have to be a fully-fledged garbage-collected object allocated in heap. It would perform better as a local structure kept in the stack-frame of the `UniqName` method. Therefore, we believe that the present performance costs of our approach are not significant, and should not be a objection against the integration of our modular initialization protocol in a new language.

Moreover, to emphasize the effects of the modularization, we performed some measurements on the code. We have chosen a few Java classes with a nontrivial initialization protocol, designed the modular initialization protocol for them, and measured the complexity of the signatures by counting the number of constructors/ini modules, and the total number of parameters of all constructors/ini modules (to take into account the sizes of all pieces of the protocol). The results, with comparison to Java, are the following:

	constructors		**parameters**	
class name	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

5 CONCLUSIONS

In this paper we addressed the problems of reusability and maintenance of code by analyzing and (hopefully) solving problems related to object initialization, which is, in our opinion, a cornerstone for modularity. We presented a new approach to the design of the initialization protocol, which has the following advantages over the TIP approach:

```
public class Point {
  float x, y;
  void PreMethod_0(Point_Init ___init, float x, float y)
  {}//this is empty since the corresponding module does nothing before new[];

  void PostMethod_0(float x, float y)
  { this.x = x;
    this.y = y;  //the actual instructions of the ini-module
  }

  void PreMethod_1(Point_Init ___init, Complex comp)
  { ___init.x=comp.x; //instructions responsible for passing
    ___init.y=comp.y; //calculated parameters to the subsequent ini-module
  }

  void PostMethod_1(Complex comp) {}

  void PreMethod_2(Point_Init ___init, float angle, float rad)
  {  ___init.x=cos(angle)*rad;
     ___init.y=sin(angle)*rad;
  }

  void PostMethod_2(float angle, float rad) {}
}

public class Point_Init
{ public float x, y;
   public Complex comp;
   public float angle, rad;
}

static Point UniqueMethod(float angle, float rad)
{ Point obj = new Point();
  Point_Init i = new Point_Init();
  i.angle = angle;
  i.rad = rad;
  obj.PreMethod_2(i, i.angle, i.rad);
  obj.PreMethod_0(i, i.x, i.y);
  obj.PostMethod_0(i.x, i.y);
  obj.PostMethod_2(i.angle, i.rad);
  return obj;
}

 UniqueMethod(0.7, 4); //result of the translation of the "new" expression
```

Figure 3: Translation of the class `Point` into Java

- It reacts better when a superclass is extended: if a parent class is extended with some new optional parameter, then the subclass gains automatically the corresponding set of options of instantiation.
- It reduces the number of "constructors" from exponential to linear: different aspects of instantiation can be defined separately.
- As an effect, it discards code duplication: if a subclass must add something to the initialization protocol, it does not have to reference all the parent constructors.
- It removes redundant repetitions of exception declarations that may cause unnecessary dependencies.
- It discards ambiguities introduced by constructor overloading.

Notice that our approach to object initialization is independent from the type system (whether it is structural or nominal), from the subtyping rules, and from the presence of overloading. We believe that, thanks to this independency, and to the possibility of a backward compatibile interpretation of TIP, our approach is a good proposal for the development of new languages, as well as extensions of existing languages.

Our idea is modelled via a Java modification, JavaMIP, but it could be applied in general to most of the existing object-oriented languages. In particular, it can also be applied to a language equipped with mixins (such as those in [2, 8]), with the benefit of extending the number of allowable combinations of mixins. In fact, as long as mixins do not need referencing nor modifying constructors of the parent classes, they allow the desired level of code modulation and reusability. But when a mixin needs to have something to do with the initialization of objects, the designer has two choices: (*i*) either defining the constructors in the resulting classes after the application of mixins, but this very often requires copying the code; (*ii*) or define them in mixins, if the language allows this, thus reducing their reusability. With our approach, a mixin would not have to reference explicitly the parent class constructors, therefore it can be applied to classes with different initialization protocols, sharing some common part, maybe even only one initialization parameter. As an effect, it makes mixins a more reusable and expressive tool, while keeping a good level of static checking of the initialization protocol.

There exists a companion paper to the present one, [3], that models a calculus called called FJMIP, which is the core of JavaMIP, in order to state and prove the soundness of our approach. Moreover, a prototype implementation of a preprocessor *javamip2java* is available at [16], together with some working examples.

There are other different works connected not with the reusability only, but specifically with the problem of constructors. One idea which is connected closely with out research and solves some (but not all) problems related to constructors is referencing the formal parameters of functions (e.g., methods and constructors) by name. Such an approach can be found for example in Flavors [17]. However, the Flavors language lacks on the statical type-checking side.

Another work, presented in [7], is a study on constructors for a mixin-based programming style in C++, where mixins are implemented by using templates. The paper describes some of the problems we have also pointed out, and proposes a solution for the problem on the non-composability of mixins (see Section 2). Nevertheless, the proposal solves this problem only, moreover it requires automatic generation of additional C++ code, which in fact can be exponential in the size of all mixin code. It also requires the declaration of additional type parameters in each mixin, which must be passed to the constructors of its ancestor classes. This may cause some overhead when programming large libraries of mixins.

The most closely related work to ours is the *Java Layers* language, which is an extension of Java with mixins equipped with a tool called *constructor propagation*. It solves some (but not all) problems related to constructors, as described in Section 2 as **Solution 5**. One of the things that cannot be done in the Java Layers approach is, as we mentioned before, the addition of new options of initialization of some properties defined in the original class. The following JavaMIP code (which is an extension of the example of Figure 2), in the Java Layers approach would require copying all combinations of the propagated constructors:

```
class HSBColorPoint extends ColorPoint {
  optional HSBColorPoint(float h, float s, float b)
  initializes(r, g, b) {...}
}
```

6 FUTURE WORK

The purpose of this paper was to present our approach to object initialization in its essential form, but some modifications/extensions are already work-in-progress: (*i*) assigning names to ini modules. This would allow the overriding of the implementation of ini modules (*not* of the lists of parameters), as for virtual methods; (*ii*) introducing a *disable* operation for disabling some of the parent class' ini modules; (*iii*) allowing the hiding of some of the initialization modules. Further research can be done with respect to the application of the modularization approach also to methods, not only to constructors. It would be useful in some cases to avoid many overloaded versions of the same method.

Acknowledgments. The authors would like to thank the anonymous referees for helping to improve this paper, Pawel Urzyczyn for inviting the first author to Warsaw, and Marcin Kowalczyk for many fruitful discussions on object-oriented languages.

REFERENCES

[1] *Delphi Language Guide*. Borland Software Corporation, 2004.

[2] D. Ancona, G. Lagorio, and E. Zucca. Jam – a smooth extension of Java with mixins. In *Proc. ECOOP '00*, volume 1850 of *LNCS*, pages 145–178. Springer-Verlag, 2000.

[3] V. Bono and J. Kuśmierek. Featherweight JavaMIP: a calculus for a modular object initialization protocol. Accepted for pubblication in FCT '07 Proc., 2007.

[4] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. OOPSLA '90*, pages 303–311. ACM Press, 1990.

[5] R. J. Cardone. *Language and Compiler Support for Mixin Programming*. PhD thesis, The University of Texas at Austin, 2002.

[6] T. Cohen and J. Gil. Better construction with factories. *Journal of Object Technology*, July/August 2007, 2007. To appear.

[7] U. W. Eisenecker, F. Blinn, and K. Czarnecki. A solution to the constructor-problem of mixin-based programming in C++. In *First Workshop on C++ Template Programming*, 2000.

[8] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL '98*, pages 171–183. ACM, 1998.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, Sun Microsystems, 2005.

[10] A. Hejlsberg, P. Golde, and S. Wiltamuth. *C♯ Language Specification*. Addison-Wesley, 2003.

[11] S. Kochan. *Programming in Objective-C*. Sams, 2004.

[12] J. Kuśmierek and V. Bono. A modular object initialization protocol. Manuscript, available at http://www.di.unito.it/∼bono/papers/JavaMIP02.pdf, 2007.

[13] G. Leavens and Y. Cheon. Design by contract with JML, 2003.

[14] X. Leroy. *The Objective Caml system release 3.09*. Institut National de Recherche en Informatique et en Automatique, 2005.

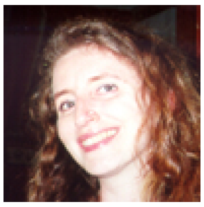
[15] B. Meyer. An Eiffel Tutorial. Technical report, ISE Technical Report TR-EI-66/TU, 2001.

[16] G. Monteferrante. *javamip2java*: A preprocessor for the JavaMIP language. Available at http://www.di.unito.it/∼bono/papers/javamip/, 2006.

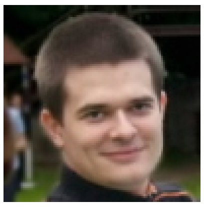
[17] D. A. Moon. Object-oriented programming with Flavors. In *Proc. OOPSLA '86*, pages 1–8. ACM Press, 1986.

[18] B. Stroustrup. *The C++ programming language*. AT&T, 1997. Third edition.

ABOUT THE AUTHORS



Viviana Bono is an Associate Professor in Computer Science at the University of Torino, Italy. Her main research interests are theoretical foundations, semantics, and design of object-oriented and functional languages. She can be reached at bono@di.unito.it and at `http://www.di.unito.it/~bono`.



Jarosław Dominik Mateusz Kuśmierek is a PhD student in Computer Science at the University of Warsaw, Poland. His main research interests are the base concepts and the design of object oriented languages, and the component oriented approach to software development. He can be reached at jdk@duch.mimuw.edu.pl.