Magda: A New Language for Modularity

Magda: A New Language for Modularity *

Viviana Bono[1], Jarek Kuśmierek[2,3], and Mauro Mulatero[1]

[1] Dipartimento di Informatica, University of Torino, Italy
[2] Google Research, Krakow
[3] MIMUW, University of Warsaw, Poland

Abstract. We introduce Magda, a modularity-oriented programming language. The language features lightweight mixins as units of code reuse, modular initialization protocols, and a hygienic approach to identifiers. In particular, Magda's modularity guarantees that the client code of a library written in Magda will never break as a consequence of any addition of members to the library's mixins.
Keywords: modularity, mixin, constructor, accidental name clash.

1 Introduction

The limitations of mainstream object-oriented languages we are particularly concerned with, and which have been our motivations for the design of a new language, are the following.

- *Limitations of composition mechanisms.* The most widely applied composition and reuse mechanism is inheritance. In its classical, single-inheritance version, this mechanism is simple, however often not sufficient. In order to overcome its limits, other proposals were developed, like multiple inheritance, mixins [43, 17, 15, 8, 24, 14, 7] and traits [46, 22]. Multiple inheritance is perceived as too complicated and too dangerous [20]. The two other constructs (mixins and traits) have been designed to tame multiple inheritance and managed to improve reusability in many aspects. However, they suffer from problems related to name clashes as described in Section 2.1.
- *Non-modular initialization protocol.* In most object-oriented languages, the initialization protocols are implemented as *constructors*. Constructors are responsible for the initialization of properties coming from different places in the class hierarchy, yet they are monolithic, which means that they have to perform all the job in one block of code. While a constructor can call a superclass' constructor to delegate part of the work, it still needs to keep all the superclass constructor's parameters in its own signature. This increases the amount of work which must be performed when the initialization protocol needs to be modified. Moreover, assumptions about the superclass' constructors are imposed, making those constructors more difficult to change.
- *Accidental name clashes.* Classes can be implemented from different components (in standard languages, by extending other classes, or by implementing some interfaces); it may then occur that the same method name has different meanings in different components. This causes problems, from non-compilation to unexpected behavior during the program execution.

* Work partially funded by the MIUR Projects DISCO.

Our new language Magda is built around the notion of *mixin*. A mixin is a class parameterized over a superclass, introduced to model some forms of multiple inheritance and improve code modularization and reusability [43, 17, 15, 8, 24, 14, 7, 3, 45, 16]. There are usually two operations defined on mixins: (*i*) application, by which a mixin is applied to a class to obtain a fully-fledged subclass (the class argument plays the role of the parent); (*ii*) composition, which makes a more specialized mixin by composing two existing ones. Notice that an indirect form of composition is possible even in the presence of application only, by applying a chain of mixins to a class (which is the way a *linearized multiple inheritance* is obtained). A mixin can defer definition and binding of methods until runtime, though attributes and instantiation parameters are still defined at compile time.

Our mixin construct has many points in common with its previous versions, with one noticeable difference: in Magda there is no concept of a class, therefore mixins are not interpreted as functions from classes to classes. Mixins are used directly to create new objects from, and also induce types in the nominal static type system of the language (as proposed independently in McJava [30]). Additionally, to take advantage of mixin reusability and enhance it, Magda contains two unique features. The most innovative one is the modularization of constructors, in such a way the part of the state initialization related to a feature is declared together with that feature. Then, mixins with independent initialization protocols can be combined without the need to copy any code. This feature was presented as a part of a Java extension called JavaMIP [13], and in [12] we proved the type soundness of Featherweight JavaMIP, an extension of Featherweight Java [29] with the modular initialization protocol. The second distinctive feature of Magda is the way of how declarations of new methods, overriding of existing methods, and method calls are specified, by declaring and referencing identifiers in a univocal way. This results in the absence of name clashes and accidental overridings. A version of this feature is the base of the HygJava language, presented in [37].

Magda was introduced in the second author's PhD thesis [35], where the language was presented together with its formal semantics and related properties. In this paper we present Magda by examples, with particular emphasis on the initialization design, which is the new and novel part of the language. In Section 2 we detail our motivations for introducing a new language with respect to the composition constructs and the initialization design, in Section 3 we present Magda by examples, in Section 4.3 we hint at the name-clashes related problems, in Section 4 we compare Magda to other languages, and Section 5 summarizes our work.

A proof-of-concept implementation of Magda, with examples (including an implementation of the Decorator pattern) and a how-to, is available [36].

Part of the material of this paper, notably the part present in Section 2 and in Section 4, was already presented in some form in [13], for motivating modular constructors for Java. Nevertheless, we believe it is necessary to include it also in the present paper, in order to motivate Magda's design choices.

2 Object-oriented languages: limitations

In this section we present some limitations of the best known object-oriented languages, which gave us motivations to introduce Magda.

2.1 Limitations of composition mechanisms

The ultimate goal of object-oriented programming should be code reuse. However, this goal is still not reached completely, despite various constructs oriented to modularity introduced in different languages. We illustrate our point via a hierarchy:

- `BaseStream` - an abstract class (or interface), with abstract methods `read` and `write`.
- `FileStream` - a class representing streams which transfer data to and from a file.
- `NetworkStream` - a class representing streams transfering data across the network.
- `BufferedStream` - a class representing streams which buffer data before sending them in one big batch.
- `CompressedStream` - a class representing streams which compress and decompress data on the fly.
- `StatsStream` - a class representing streams which calculate different statistics.
- `EncryptedStream` - a class representing streams which encrypt data when written, and decrypt during reading.
- `DatabaseStream` - a class representing streams which transfer data to and from a database.

We assume that `BufferedStream` has a method `SetBufferSize`, which sets the amount of data in memory before the data is sent to the communication device. Moreover, class `CompressedStream` uses the method `SetBufferSize` to decide about the amount of data to be compressed. Additionally, we expect to be able to obtain combinations of the above features, like compressed and encrypted network stream, or buffered file stream with statistics.

Single inheritance. With single inheritance, each class can have at most one ancestor. When we want a class to reuse features from more than one class, we can make it inherit from the class which contains most of the needed features and then: either (*i*) we copy manually the code from the remaining classes; or (*ii*) we use object composition, declaring fields of the types of the left-out classes and then implementing methods associated to appropriate delegate objects. This approach can however cause problems, when all the classes in question share common ancestors defining a state. Then every change of the common state of the "proper object" needs to be propagated also to the delegate objects. This results in behavior which is in many aspects similar to the one of virtual multiple inheritance in C++.

Object composition/Decorator design pattern. One of the approaches which exploit the object composition is the *Decorator pattern* [25]. In this approach, to create, for instance, an `EncryptedStream`, a new class is implemented which contains a reference to a stream object (called *delegate*). Then, in the new class we declare all the methods of the `BaseStream` class. Methods whose behavior is modified (like `read` and `write` in the stream hierarchy), perform their new tasks and then call the corresponding methods in the delegate. All other methods are implemented to just call their counterparts in the delegate object. This approach has often the required compositional flexibility, and it is sometimes preferred over inheritance. It is especially useful when additional features of objects should be enabled and disabled dynamically during the life of an object. However, it has numerous disadvantages when used instead of inheritance, because: it requires declarations of methods which will only call the same method in the delegate object; it creates unwanted dependencies in the code, because any modification or addition of a method in the delegate object's class requires

a repetition of the same operation in other classes; when a class `A` is composed with a class `B` and redefines some of the methods from `B` by providing a new implementation in its declaration (that corresponds to method override in inheritance), then this redefinition is visible only from the point of view of external clients, since the other methods in the object calling this method will call the original implementation, not the redefined one.

Multiple inheritance. Multiple inheritance, whose best known implementation is the one in C++, is a powerful feature, however its complicated semantics make it difficult and dangerous to use, as summarized by Cook [20]: "Multiple inheritance is good, but there is no good way to do it." The main problem with multiple inheritance is the way it deals with ambiguous and conflicting features. In the example, such problems would occur in a class inheriting from both `CompressedStream` and `BufferedStream`, because both classes have method `SetBufferSize`. Moveover, multiple inheritance does not allow the user to keep in the resulting class both of the methods with the same name. In addition, when the same method is defined or overridden in superclasses which share a common ancestor, then the order in which overridden variants of the method are called is not always obvious. As a result, many mainstream languages developed after C++ (like Java and C♯), borrowed numerous features of that language, however not the multiple inheritance. There exist other languages which adopted different flavors of multiple inheritance, like Loglan [34] and Python [10]. Those languages use linearization algorithms to change the graph of ancestors into a list over which dynamic dispatch is performed. However, linearization leads to cases when the ordering of the overridings is non-trivial to understand and fragile to innocent-looking changes in the hierarchy.

Mixins. In the case of methods with the same name occurring in different mixins (like the aforementioned `SetBufferSize`), mixins permit explicit direct control of the order in which those methods will override each other, however, they might not allow the resulting class to have all variants. Nevertheless, there are two proposals offering this feature: MixGen [3] and MixedJava [24]. In the MixGen language, the class obtained from applying both `CompressedStream` and `BufferedStream` mixins will keep both variants, while giving access to one of them at each moment, depending on the static type of the variable referring to the object. However, this approach does not allow the user to access both of them at the same time and in some cases it is not obvious for the user which implementation will be called in a given expression.

Traits. Traits [46, 22] have been designed as an alternative mechanism of code reuse. Initially they were introduced in a untyped setting as an extension of Smalltalk [22, 46]. Later on they were studied also in typed (thus often restricted) settings [47, 44, 11, 6]. One of the design goals of this approach was to overcome problems with mixins mentioned above. Each trait is a minimal unit of reuse, containing a set of implemented methods and a set of required methods. A class is then built by composing traits (with or without the presence of single inheritance). The advantage of the composition mechanism available in trait-based languages is that they issue warnings when name clashes occur and offers operators (like hiding, aliasing and renaming) to modify the way traits are composed in a class definition. Some additional code in the class, called *glue code*, might be needed to make the composition work.

In our example, one can specify traits `CompressedStream` and `EncryptedStream`, where each of them would require methods `read` and `write` and contain their overriding variants supporting compressed and encrypted data. Finally, using those traits, classes like `CompressedFileStream`, or `EncryptedNetworkStream` can be created. Thanks to this flex-

ible method manipulation mechanism, the programmer has the choice of how to deal with the method names conflicts. However, this does not come without a price. Consider the following example (written in a Smalltalk-like syntax):

```
// Library containing trait EncryptedStream and class CertificateManager
trait named: #EncryptedStream
  instanceVariableNames: 'aCertificate'
  setCertificate: certificate
    ...do checks on the certificate...
  aCertificate := certificate.

  autoFindCertificate
    | manager |
    manager = getGlobalCertificateManager.
    manager findCertificateFor: self.

Object subclass: #CertificateManager
  findCertificateFor: encrypted_connection
    | some_certificate |
    some_certificate := ... do something to establish certificate....
    encrypted_connection setCertificate: some_certificate.


// Client code with class CryptFileStream and its usage
FileStream subclass: #CryptFileStream
  uses: {EncryptedStream @ {#setCertificate: -> #setEncryptCert:} -
         {#setCertificate:}} + otherTrait
     ... a class declaration ....

// Usage of CryptFileStream class.
// The call to method autoFindCertificate fails, because the renamed method
// setCertificate is used by the CertificateManager object (created internally).
// This dependency is not visible in the signature!!!!!!!!!!!!!!!!!!!!!!!!!!
  stream := CryptFileStream new.
  stream autoFindCertificate.
```

Trait `EncryptedStream` encrypts data, and comes with a `CertificateManager` (trait or class). This `CertificateManager`, depending on the user logged in, sets certificates automatically for the encrypted streams. To do this, the `CertificateManager` calls method `setCertificate` on the `EncryptedStream` and also a getter to choose the certificate needed for the stream. Unfortunately, when the new `CryptFileStream` class will be defined using the `EncryptedStream` trait, and the `setCertificate` method will be renamed, then the certificate manager will stop working on that stream. Notice that such a dependency is not reflected in the specification of the methods required by a trait, because the latter lists only methods required to be present in the same object. In general, when traits are used to create functionalities spanning many objects, it might be hard to predict which dependency will be broken, that is, when any method in the trait will be renamed or hidden. This problem has already been spotted and worked around using different solutions. The first solution was developed in Chai, an extension of Java with traits [47], and independently in the Fortress language [6, 4, 5]. The second solution is in the work adapting traits to statically typed languages [44]. Finally, another solution is based on *freezable traits*

[23]. In the first solution, when a method is removed, then another implementation of the method with the same signature must be added to the class. However, those methods can sometimes be sematically different. Additionally, this restriction implies that conflicts of two different methods with the same name and different result type cannot be resolved. In the Fortress language, traits also induce types, therefore Fortress language is even more restrictive than Chai: method renaming and hiding are forbidden. As a consequence, when two traits with methods with the same name are composed, then one method overrides the other one. In the second solution, traits themselves are not visible in the nominal type system of the language. Therefore, it is necessary to declare a public interface containing (some of) the methods of a trait, and make sure that all classes using that trait implement the interface. This implies that, when a method is renamed in the declaration of a class, then it either requires the class to stop implementing that interface or imposes additional changes in the code. In the third solution based on the notion of method freezing [23], it is possible to keep two versions of conflicting methods: one as private, and other one as public. The private version is executed when a self call is performed, which means a call from the trait in which this private, or frozen, method has been declared. However, this approach can only be used to solve the problem when the conflicting method is called internally, by another method declared in the same trait. Unfortunately, it cannot be used when this method is required by another class/trait, as the method `setCertificate:`, called in method `findCertificateFor:` of class `CertificateManager`.

2.2 Non-modular initialization protocol

Class-based languages are usually equipped with an object initialization protocol. Such protocol describes: (*i*) the information needed to create and initialize an object; (*ii*) the code performing the initialization. In most languages (for instance, C++ [48], C♯ [28], Delphi [2], Java [27], and Visual Basic [1]), the initialization protocol of a class is specified by *constructors*. Each constructor consists of: (*i*) a list of parameters; (*ii*) a body. Unfortunately, this traditional approach to initialization (TIP from now on) has drawbacks. We present them via some Java examples, even though most of the problems occur also in other mainstream languages.

Optional parameters. Traditional initialization protocol leads to an exponential number of constructors with respect to the number of optional parameters. In Java, in classes like `java.awt.TextArea`, there is often one constructor with the largest possible set of parameters. However, additional constructors with a subset of those parameters must be declared when some parameters are optional. Moreover, it is not possible to have two constructors with parameter lists of the same length and compatible types of the corresponding parameters in the same class. These drawbacks can be partially solved in languages containing named parameters (like Python [10]) and default parameter values (like Delphi [2], Python [10], C++ [48]).

Multiple initialization options and code duplication. TIP leads also to an exponential number of constructors with respect to the number of object properties with different initialization options. If each property can be initialized in more than one way, then the possible number of initialization options of a given class (thus the number of constructors) is a multiplication of the numbers of options of the object properties. An example could be the combination of a property color (with two palettes, RGB and CMYK) with a property position (with three options, cartesian, polar, and complex) in a class `ColorPoint`, which

induces six constructors, with some duplication of code. The Java class `java.net.Socket`, which contains nine constructors, is a more sophisticated, real-life example of this problem.

Unnecessary constructor dependencies. It may happen that modifications of a class force unnecessary changes in subclasses. Consider the following scenario: class C_1 with a set of constructors; class C_2 declared as a subclass of C_1, containing all parent constructors redeclared by adding a parameter *Par'*, calling the corresponding constructors in C_1; if another constructor is added to C_1, C_2 will not inherit automatically the corresponding constructor. This class, depending on the language design, will either retain the constructor added in C_1 without the additional parameter *Par'*, or just will not inherit it at all (as in the case of Java). We think that neither of these options is good enough.

Fragile overloaded constructors. Overloaded constructors in TIP make safe-looking changes non conservative. When a Java (or C++ or C♯) class contains different constructors, the choice of the constructor 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 compiles, because only one of constructors of class `ClassWithOptions` matches the `new` expression. However, if the class `C2` implements also the interface `I1`, then the previous code will not work anymore (because the last `new` becomes ambiguous). Notice that this is not a problem in languages in which it is possible to name constructors (e.g., in Delphi [2]), since the overloading can be avoided.

Unnecessary redeclarations of checked exceptions. In Java, if a constructor throws some exceptions and is called by a constructor of a subclass, this one must repeat the whole list of exception declarations. While such a repetition in 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 thrown by a superclass' constructors, therefore the repetition of the declarations is an additional, useless work.

Problems with traditional mixins. A mixin declaration, like any other subclass declaration, may contain declarations of new constructors, which, in turn, may reference the superclass constructors. There are cases in which it would be desirable to compose independently designed mixins. We consider the following example: a class `Button` and two mixins `Blinking` and `Ringing`, that, when applied separately to `Button`, will result in, respectively, a class of blinking buttons and a class of ringing buttons. Then, we want to compose them to obtain a class of blinking and ringing buttons (the order should not matter). Unfortunately, if the mixins modify the interface of a constructor of the parent class, for example by adding one parameter, then they are non-composable. Assume that `Button` has a constructor with n parameters and that the mixin `Blinking` must define a new constructor (that replaces the old one), having n+1 parameters (one more for color) and calling the superclass con-

structor. Similarly, the `Ringing` mixin may need, for example, the frequency, therefore it will have an (n+1)-parameter constructor as well. Now, if we apply one of those mixins to the `Button` class, then the resulting class will have a constructor with $n + 1$ parameters, and it will not be an appropriate argument for the other mixin. Those problems have, in fact, a similar nature as those occurring with standard subclassing. However, mixins are designed for a wider reuse than subclasses, therefore problems may occur more often and are more difficult to foresee. Notice that also the designers of Jam [7] have encountered 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 classes resulting from a mixin application.

3 The Magda language

In this section we present the Magda language by examples. A detailed description of its syntax can be found in [35].

Every program in Magda consists of two parts: a list of *mixin declarations* and a list of instructions, called *main instructions*. The main instructions may use the mixin declarations. Any type in Magda is a sequence of mixin names. Values are either `null` or object references. Value `null` is the default value for variables and fields. Magda expressions include:

- the creation of a new object. Each object in Magda is created from a non-empty sequence of mixins. The sequence of mixins plays a role similar to the one of a class in other languages.
- the method call. In Magda, as in most object-oriented languages, the set of methods "understood" by an object depends on the "schema" from which it was created. Therefore, the set of understood methods depends on the sequence of mixins used to create the object.

This is a simple program printing "Hello World" written in Magda:

```
mixin HelloWorld of Object =
  new Object MainMatter()
  begin
    "Hello world".String.print();
  end;
end;
//
(new HelloWorld []).HelloWorld.MainMatter();
```

The program consists of the declaration of a mixin named `HelloWorld` and one main instruction. The mixin `HelloWorld` contains a method named `MainMatter`. This method begins with the keyword `new` which indicates the introduction of a *new method identifier*, as opposed to *method redefinition*, and *abstract methods*, both described later on. The main instruction starts with the creation of an object from mixin `HelloWorld`, using the `new HelloWorld[]` expression. Then it calls the method `MainMatter` on the object, that, in turn, calls the method `print`, declared in mixin `String`. In Magda, `String` is a built-in mixin and contains methods like `print` and `add` (which concatenates two strings). Similarly, Magda contains other built-in mixins like `Integer` and `Boolean`. The `Boolean` mixin cannot be used in object creation expressions. Then, the only way to obtain a `Boolean` value is to

use one of the `Boolean` constants: `true` and `false`. Since booleans are object values, `null` is also the default value for `Boolean`. Moreover, Magda features two control instructions: a conditional instruction `if` and a loop instruction `while`.

In each method call, the method name is prefixed with the name of the mixin in which it was introduced. This is to enforce *hygiene* of identifiers in order to avoid accidental clashes, as explained earlier in the introduction.

In the current version of Magda all methods are visible. This results in a similar behavior as the one of `public` methods in Java. Visibility is however orthogonal to the Magda's specific features presented in this paper, therefore it is not discussed, but kept in mind for future versions of the language. In practice, for a better code organization, the declarations of mixins should be split into different modules, with their own namespaces. However, this issue could be dealt with as in other languages (like Java and $C\sharp$) and in this paper we assume that every name of mixin is unique.

Object fields and local variables. All fields of an object are declared, with a name and a type, in the mixins from which the object is created. The type of a field is a sequence of mixin names. The value of each field can be either the `null` value or an object value, that is, a reference to an object. For simplicity, we have chosen that each field in a Magda object can be only accessed by methods of that object, via an invocation of the form `this.fieldId`. As a result we obtain a behavior close to the one of *protected* fields in other languages. Furthermore, similarly to method identifiers, all fields are prefixed with the name of the mixin in which the fields have been declared.

In the following example there is a declaration of a mixin `Point2D` containing two field declarations, `x` and `y`.

```
mixin Point2D of Object =
  x:Integer;
  y:Integer;

  new Object setCoords( ax:Integer; ay:Integer)
  begin
    this.Point2D.x := ax;
    this.Point2D.y := ay;
  end;

  new Integer getX()
  begin
    return this.Point2D.x;
  end;
end;
//
mixin MainClass of Object =
  new Object MainMatter()
    p1:Point2D;
    p2:Point2D;
  begin
    p1 := new Point2D[];
    p2 := p1;
    p1.Point2D.setCoords( 11, 10);
    p2.Point2D.getX().Integer.print();  //this line prints out 11
  end;
```

```
end;
//
(new MainClass []).MainClass.MainMatter();
```

As in `MainClass.MainMatter` method, variable declarations are in the header of the method, after the signature, before the keyword `begin`, (like in Pascal [9]). The type of a variable is a sequence of mixin names; in our example, variable `p1` has type `Point2D`. Similarly to fields, local variables can be assigned with object values.

Inheritance. The inheritance mechanism in Magda works in a different way than in traditional single-inheritance languages (like Java, C♯) or multiple-inheritance ones (like C++), where a new class extending an existing class or classes automatically includes all of their members (fields and methods).

In Magda, a mixin `C` can specify that it *requires* other mixins (with the keyword `of`), for instance mixin `A`. In this case, we say that `A` is a *base mixin* of `C`. Moreover, we call *indirect base mixins* the mixins beloging to the transitive closure of the relation "requires'' with respect to the mixin declarations in a program. All code referring to an object created from mixin `C` can safely use methods declared in the base mixin `A`. This applies also to the code of the methods declared within mixin `C` itself. However, unlike in traditional languages, all the methods and the fields declared in `A` are not implicitly included in `C`, that is, all the members declared within `A` are not visible in the same way as if they had been declared in `C`. In particular it means that: (*i*) an object creation expression using mixin `C` needs also to explicitly mention mixin `A`. Moreover, in the sequence of mixins used to create an object from, `A` must occur at an earlier position than `C`; (*ii*) every method call (and field dereference) on an object created from the set of mixins {`A`, `C`}, is prefixed with the name of the mixin from which the method originates. For instance, if the method was declared originally within `A`, then the method call needs to specify it. Each mixin which is not declared to extend explicitly any specific mixin extends implicitly `Object` which is the base mixin of every mixin.

A simple example of a program containing the declaration of two mixins, `Point2D` the base mixin and `Point3D` extending it, follows.

```
mixin Point2D of Object =
  x:Integer;
  y:Integer;

  new Object setCoords2D( ax:Integer; ay:Integer)
  begin
    this.Point2D.x := ax;
    this.Point2D.y := ay;
  end;

  new Integer getX()
  begin
    return this.Point2D.x;
  end;
end;

mixin Point3D of Point2D =
  z:Integer;

  new Object setCoords3D( ax:Integer; ay:Integer; az:Integer)
```

```
  begin
    this.Point2D.setCoords2D(ax, ay); //a call to the method coming from
    this.Point3D.z := az;             //another mixin is prefixed with its name
  end;
end;

mixin MainClass of Object =
  new Object MainMatter()
    x:Point3D;
  begin
    x := new Point2D, Point3D[];
    x.Point3D.setCoords3D( 10, 11, 12); //this method comes from Point3D
    x.Point2D.getX().Integer.print();   //while that one from    Point2D
  end;
end;
(new MainClass []).MainClass.MainMatter();
```

Multiple inheritance. In this section we present how Magda mixins can be combined to obtain a form of multiple inheritance. The program in Figures 1 features the declarations of five mixins. The first two mixins `DisplayableObject`, `Point2D` do not inherit from any

```
// Declarations of two mixins (DisplayableObject, Point2D)
mixin DisplayableObject of Object = ...
end;

mixin Point2D of Object =  ...
end;

// Two other mixins extending the Point2D mixin
mixin Point3D of Point2D =  ...
end;

mixin ColorPoint of Point2D = ...
end;

// A mixin extending three independent mixins
mixin Displayable3DColorPoint of DisplayableObject, Point3D, ColorPoint =  ...
end;

// Composition of independent mixins in object creation
new Point2D, Point3D, ColorPoint [...]
```

Fig. 1. Multiple inheritance in Magda: code snippets

other mixin. Mixins `Point3D`, `ColorPoint` extend mixin `Point2D` in a single-inheritance fashion. Then, there is mixin `Displayable3DColorPoint`, extending three independent mixins (Figure 2 depicts the inheritance graph). Finally, the program contains the creation of an object, which uses three mixins to create an object from.

In Magda there are two features for modularizing the code and composing components: (*i*) a mixin `A` can declare more than one base mixin, that is, it can extend multiple mixins

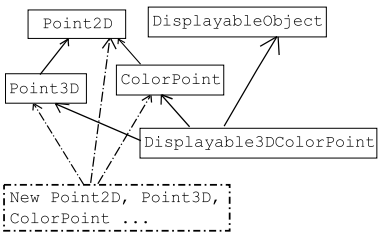

Fig. 2. Multiple inheritance in Magda: mixin hierarchy

at once; then, in an object creation expression using the mixin, all of its base mixins must be listed explicitly and the base mixins must occur earlier than `A` in the expression, however their relative ordering is arbitrary (see the example on virtual methods shown below, which presents different results depending on ordering); (*ii*) the declaration of base mixins represents only the minimal set of mixins required by a given mixin `A` to be combined with. Therefore, an object creation expression can combine more mixins with the mixin `A` than specified in its base mixin declarations.

Virtual methods. Methods in Magda work like virtual methods of other languages (for example, C++ [48]). Unlike in the Java language, method redefinition has a different syntax from method introduction. The redefinition begins with the `override` keyword instead of the `new` keyword used in the introduction. Additionally, each method redefinition contains the name of the mixin in which the redefined method was first introduced. The body of a method redefinition can use a `super(...)` expression to call the previous implementation of the method. When a `super(...)` expression is evaluated in a method mt of a mixin M_c belonging to a mixin sequence $\overrightarrow{M}$, the previous implementation of mt is the one present in the last mixin in $\overrightarrow{M}$ preceding M_c, which contains a definition/redefinition of the method mt. As a result, a `super(...)` call expression within a method redefinition used in different objects can call different method implementations. That is, the actual method body called by the `super(...)` expression depends on the mixins (and their order) which have been used to create the object.

An example of a program using method redefinitions in which the behavior of `super(...)` call changes depending on the ordering of mixins is the following.

```
mixin BaseMixin of Object=
  new String GetActualName()
  begin
    return "Base ";
  end;
end;

mixin Extension1 of BaseMixin =
  override String BaseMixin.GetActualName()
  begin
    return super().String.add("Extension1 ");
  end;
end;
```

```
mixin Extension2 of BaseMixin =
  override String BaseMixin.GetActualName()
  begin
    return super().String.add("Extension2 ");
  end;
end;

(new BaseMixin, Extension1, Extension2[]).BaseMixin.GetActualName().String.print();
"\n".String.print();                    // this prints the "end of line" character
(new BaseMixin, Extension2, Extension1[]).BaseMixin.GetActualName().String.print();
// Program generates output of the form:
// Base Extension1 Extension2
// Base Extension2 Extension1
```

Because we choose explicitly which method from which mixin to redefine when declaring a method redefinition, there are no problems when two base mixins of a given mixin contain method definitions with the same name. Moreover, this explicitness also ensures that accidental name clashes are avoided when a new method is added to a mixin extended by other mixins.

Abstract methods. A mixin declaration can also contain a declaration of a new method identifier without the body, marked with the keyword `abstract` (similarly to Java). Then, another mixin can supply its body, by marking it with the keyword `override`. If a mixin containing an `abstract` method is used in an object creation expression, then another mixin with the `override` version of the same method is required. The following example shows some features of the abstract methods.

```
mixin M1 of Object =
  abstract String Met1();
end;

mixin M2 of M1 =
  override String M1.Met1()
  begin
    return "Implementation from M2";
  end;
end;

mixin M3 of M1 =
  override String M1.Met1()
  begin
    return super().String.add(" with redefinition from M3");
  end;
end;

(new M1, M2, M3 []).M1.Met1().String.print();
// OK and prints: "Implementation from M2 with redefinition from M3"

(new M1, M3, M2 []);
// does not compile, M3.Met1 not suited as first definition of M1.Met1
// (contains call to super())

(new M1 []);
// does not compile, implementation of M1.Met1 missing
```

Object initialization. We developed for Magda a *modular initialization protocol* approach based on small, composable pieces called *initialization modules* (*ini modules* from now on).

Each ini module has a list of formal parameters, called *input parameters*, which can be supplied at object creation. Its signature also declares whether its usage is *required* or *optional*, meaning whether if its input parameters are required or are optional in an object creation expression including the mixin containing that ini module. An ini module also specifies a list of *output parameters*, referring by name to input parameters declared in other modules, which will be computed by the ini module and supplied to such other modules.

The pseudo-syntax of an ini module is as follows:

```
<modifier> MixinName(in_param) initializes (out_param_id)
  local_var
  I1
  super[out_param_id := expr];
  I2
end;
```

where: `<modifier>` is either `required` or `optional`; `MixinName` is the name of the mixin in which the ini module is declared (we include it in order to make ini modules look similar to constructors in Java and C♯); $\overrightarrow{\texttt{in\_param}}$ is the (possibly empty) list of the input parameters of the ini module, declared with their types; $\overrightarrow{\texttt{out\_param\_id}}$ is the (possibly empty) list of the output parameter (hygienic) identifiers whose values are computed by the ini module; `super[`$\overrightarrow{\texttt{out\_param\_id := expr}}$`]` is an assigment to the output parameters and a call to other ini modules, which the output parameters will be supplied to as input parameters.

The pseudo-syntax of an object creation instruction providing initialization parameters is as follows:

```
obj_id := new mixin_sequence[id := expr];
```

where: `mixin_sequence` is the sequence of mixins the object is created from; $\overrightarrow{\texttt{id := expr}}$ are the initialization parameters together with their initialization expressions.

We describe the semantics of the object initialization procedure with an informal algorithm (the complete formalization of the semantics can be found in [35]).

Given an object creation expression `new mixin_sequence[`$\overrightarrow{\texttt{id := expr}}$`]`:

- the $\overrightarrow{\texttt{expr}}$ are evaluated in $\overrightarrow{\texttt{val}}$ and assigned to their each respective `id`;
- (*) a sequence of ini modules $\overrightarrow{\texttt{mod}}$ is extracted from `mixin_sequence`, respecting the order of the mixins in `<mixin_sequence>` and, within each mixin, respecting the syntactic order from top to bottom (i.e., a module declared first in the mixin comes first in the sequence). An ini module is a tuple containing: the sequence of input parameters, the sequence of output parameters, the body, the name of the mixin it comes from.

Then, consider a set of assigned parameters $\overline{\texttt{id} := \texttt{val}}$ and a sequence of ini modules $\overrightarrow{\texttt{mod}}$:

- if the set of assigned parameters and the sequence of ini modules are empty, then the initialization process terminates;
- otherwise, if the sequence of ini modules is not empty, let `mod` be the last ini module of the sequence, `Ip` be the set of input parameter identifiers of `mod`, and the instruction `super[`$\overrightarrow{\texttt{out\_param\_id := expr}}$`]` be contained in the body of `mod`:
  - if `Ip` is not equal to any subset of $\overline{\texttt{id}}$ and `Ip`$\neq \emptyset$, then the inizialization process is resumed with the sequence of ini modules $\overrightarrow{\texttt{mod}}$/`mod` (`mod` is discarded) and the same set of assigned parameters $\overline{\texttt{id} := \texttt{val}}$;

- if `Ip` is a subset of $\overline{\texttt{id}}$ we say that `mod` is *activated* and we proceed as follows: (*1*) the identifiers in `Ip` are mapped onto their actual values (which are among the $\overline{\texttt{val}}$ in $\overline{\texttt{id} := \texttt{val}}$); (*2*) the sequence of instructions `I1` is executed; (*3*) the expressions in `super[`$\overrightarrow{\texttt{out\_param\_id := expr}}$`]` are evaluated and assigned to their respective output parameter identifiers; (*4*) the initialization process is resumed with the sequence of ini modules $\overrightarrow{\texttt{mod}}$`/mod`, and the set of assigned parameters where the ones corresponding to the identifiers in `Ip` are removed and the assigned output parameters of `mod` are added; (*5*) the sequence of instructions `I2` is executed.

A static check ensures that all `required` ini modules are activated (see [35]). Note that ini modules with an empty sequence of input parameters are always activated.

We say that a parameter `par` is *not consumed* if it is either supplied in an object creation expression, or calculated as an output parameter of an already executed ini module, but no module taking `par` as an input parameter has been executed yet; *consumed*, otherwise.

It is important to remark that the order in which ini modules are written in a mixin has an impact on the object creation. In particular, considering the order in which the sequence of ini modules is built (see the step marked with (*)) and then examined during the initialization process, if an ini module `mod1` in a mixin outputs a parameter which is consumed by another module `mod2` in the same mixin, then `mod2` must be written above `mod1`. This condition is verified by the static checker (see [35]). The need of a syntactic ordering among ini modules in a mixin may look inconvenient, but it seems difficult to discharge it without complicating the operational semantics of the modular initialization protocol in a significant way. In [26], there is a version of JavaMIP [13] where the ordering among ini modules in a mixin can be random; this version has a simple semantics, but some restrictions on the modular protocol are present. For example, it is not possible to split the initialization of a particular version of a property over more than one ini module (for instance, there would be no possibility of initializing `x` and `y` of the example in Figure 3 with two different ini modules, one for each).

We present a program based on two mixins containing ini module declarations in Figure 3. The corresponding executable code can be downloaded [36].

The first mixin (`Point2D`) contains the declaration of ini module `mod1`, requiring parameters `x` and `y` and not computing any output parameter. That is why the `super[]` instruction in `mod1` does not contain any parameter assignment.

The second mixin (`Point3D`) contains the declarations of two ini modules. The first one (`mod2`) expects one input parameter `z`, which is used to initialize the field `fz` of the object. The second one (`mod3`) has one input parameter `other` and three output parameters. Those output parameters refer to the input parameter of the module `mod2` declared in the same mixin, and to the parameters of the module `mod1` declared in the base mixin. Its body contains a `super[...]` instruction, which computes the values of the three output parameters. Finally, the `MainClass.MainMatter` method contains two object creation expressions. The first expression supplies the values of three initialization parameters: `z` of mixin `Point3D`, and `x` and `y` of mixin `Point2D`. The first parameter `Point3D.z` is supplied to module `mod2`, and that module is the first one to be executed. When its execution reaches `super[]`, the search begins for the next module to be executed, which is `mod1`, because parameters `x` and `y` are yet to be consumed. The last instruction of the module `mod1`, the `super[]` call, does nothing, since all parameters have been consumed. The second object creation expression supplies the value of one initialization parameter, `Point3D.other`, to the optional module `mod3`. When this module is executed, it computes the three parameters `x`,`y`,`z`. The

```
mixin Point2D of Object =
  fx:Integer; fy:Integer;

  required Point2D(x:Integer; y:Integer) initializes ()         //module mod1
  begin
    this.Point2D.fx := x;
    this.Point2D.fy := y;
    super[];
  end;

  new Integer getX() ....;
  new Integer getY() ....;
end;

mixin Point3D of Point2D =
  fz:Integer;

  required Point3D(z:Integer) initializes ()                    //module mod2
  begin
    this.Point3D.fz := z;
    super[];
  end;

  optional Point3D(other:Point3D) initializes
      (Point2D.x, Point2D.y, Point3D.z)                         //module mod3
  begin
    super[Point2D.x:= other.Point2D.getX(), Point2D.y:= other.Point2D.getY(),
          Point3D.z:= other.Point3D.getZ()];
  end;

  new Integer getZ() ....;
end;

mixin MainClass of Object =
  new Object MainMatter()
    p1:Point3D; p2:Point2D;
  begin
    p1:= new Point2D, Point3D[ Point3D.z:= 12, Point2D.x:=10, Point2D.y:=11 ];
    p2:= new Point2D, Point3D[ Point3D.other := p1 ];
  end;
end;
(new MainClass []).MainClass.MainMatter();
```

Fig. 3. Object initialization in Magda

third parameter is passed to the module `mod2` and then, when the execution reaches the `super[...]` call, the remaining two parameters are supplied to module `mod1`. As a result, all the ini modules declared in the program are executed during the creation of that object, in a bottom-up order: `mod3`, `mod2`, `mod1`.

Notice that the `super[...]` instruction in an ini module is different from the `super(...)` expression within overriding method bodies, because the parameters supplied in `super[...]` do not have to be the parameters which will be passed to the ini module activated by this `super[...]` instruction. Those parameters can be in fact passed to other modules which will be executed later on. To understand this better, consider again the example in Figure 3. The `super[...]` instruction in the module `mod3` supplies the values of three initialization parameters and calls the module `mod2`. However, only one of these parameters (`z`) is consumed by `mod2`, while the remaining parameters (`x`, `y`) will be supplied to another ini module (`mod1`). Moreover, `super[ ]` instruction within `mod2` does not contain any parameters, however it calls the module `mod1` which takes two parameters which have been computed by `mod3`.

In the following, we will discuss how Magda's ini modules solve the problems of traditional constructors described in Section 2.

Optional parameters. We can introduce an optional ini module for each set of optional attributes as (input) parameters without increasing exponentially the number of ini modules with respect to the number of the attributes, since each ini module takes care only of its input parameters. Also, there is no problem with parameter lists of the same length and compatible types, thanks to named parameters.

Unnecessary constructor dependencies. Any new ini module introduced in a mixin is independent from the ini modules already present. We see this in the following example (continuation of the example from Figure 3):

```
mixin ColorPoint of Point2D =
  fcr:Integer;
  fcg:Integer;
  fcb:Integer;

  required ColorPoint(cr:Integer, cg:Integer, cb:Integer) initializes ()
  begin
    this.ColorPoint.fcr := cr;
    this.ColorPoint.fcg := cg;
    this.ColorPoint.fcb := cb;
    super[];
  end;
end;

new Point2D, Point3D, ColorPoint[Point2D.x := 1, Point2D.y := 2,
                  Point3D.z := 10, ColorPoint.cr := 255, ....];
```

The initialization module in mixin `ColorPoint` is independent from the ones in mixin `Point2D`, in particular it does not refer to any of `Point2D` ini modules' parameters. An ini module contains references to input parameters of other ini modules only when it computes their actual values (in this case, they are present as output parameters of the ini module); for instance, the ini module from Figure 3 with input parameter `other:Point3D` refers to parameters `x`, `y`, and `z` as its output parameters.

Multiple initialization options and code duplication. Each option corresponds to a certain ini module, and each ini module deals with one option without need of code duplication.

An example is the one above that mixes color with coordinates: this is a typical case that would cause code duplication when using classical constructors.

Fragile overloaded constructors. In Magda, the choice of the appropriate ini modules is done using the names of the parameters and no form of overloading is present. As a result, there can be no ambiguities.

Unnecessary redeclaration of checked exceptions. A natural way of adding checked exceptions to Magda would be to add them to ini modules. Then, there would be no need to repeat the declarations in the ini modules of the derived mixins, since they work as extensions, not as replacements of the parent initialization modules.

Problems with traditional mixins. Since within mixin-based inheritance initialization problems are amplified with respect to the ones in class-based inheritance, we believe that ini modules are a natural way to enhance mixin modularity.

Types and type expressions. Magda is a statically typed language and enjoys a type soundness property guaranteeing absence of message-not-understood errors. This is formally defined and proved in [35]. What distinguishes Magda from other languages, then, is the way the type expressions are formed and the way the subtyping is verified. Every type in a Magda program is a sequence of mixin names. The ordering of mixin names in a type expression is insignificant. When a variable or a field is declared of type T, it means that its value can be either `null` or an object created from a sequence of mixins which contains at least the mixins present in T (even though it can contain more), except the mixin `Object` which is always used implicitly during each object creation. Similarly, if a method parameter is of type T, it means that in each method call the actual value must be an object created from all the mixins in T (and maybe more).

For example, consider the program in Figure 3. The type of variable `p1` in method `MainClass.MainMatter` is `Point3D`, while the type of variable `p2` is `Point2D`. The second declaration means that the variable `p2` can hold only `null` value, or a reference to an object created using a sequence of mixins containing `Point2D`. However, notice that `Point3D` mixin has `Point2D` as its base mixins. As a result, every object created from `Point3D` is also created from `Point2D`. Therefore the type `Point2D`, `Point3D` is equivalent to the type `Point3D` as well as to the type `Point3D`, `Point2D`. On the one hand, in an object creation expression the base mixins cannot be omitted because the order in which they are present in the object creation expression is significant (see again the example on virtual methods). On the other hand, in types the ordering and the removal of base mixins are insignificant.

We say that type T_2 is the *fully expanded form* of type T_1 if T_2 is the type obtained by appending to T_1 all direct and indirect base mixins of T_1. In the example in Figures 1 the fully expanded form of type `Displayable3DColorPoint` is: `Object`, `Point2D`, `DisplayableObject`, `Point3D`, `ColorPoint`, `Displayable3DColorPoint`.

Now consider the program from Figure 1, extended with the mixin:

```
mixin Test of Object =
  new Object SomeMethod ()
    v1: Point3D;
    v2: Point3D, ColorPoint;
  begin
    ...
    v1 := v2; //OK
    v2 := v1; //not OK
  end;
end
```

In the method `SomeMethod` there are two variables, `v1` and `v2`. The requirements enforced by the type of variable `v2` are stricter than the ones enforced by the type of variable `v1`. As a result, each value of variable `v2` can be also a value of variable `v1`. However, the opposite does not hold. Therefore, the first assignment present in this method is type correct, however the second one is not. Thus this program will not compile.

In general, we say that type T_2 is a *subtype* of type T_1 (denoted $T_2 \preceq T_1$) when the fully expanded form of T_1 is a subset of the fully expanded form of T_2. As a consequence of this definition, we define the type of `null` value as the set of all mixin names used within a program.

Properties of Magda. Our modular approach to initialization is motivated by the fact that we want each mixin to be as composable with other mixins as possible, in order to minimize the amount of the code to be written (or, worse, duplicated). Magda enjoys an unusual safety property which intuitively means that no accidental conflicts can happen.

We say that an identifier `p` is a *transitive output parameter* of an ini module `m` if: (i) either `p` is an output parameter of `m`; (ii) or `p` is an output parameter of an ini module `n`, such that at least one input parameter of `n` is a transitive output parameter of `m`.

We say that two mixins `M_1` and `M_2` are *exclusive* if: (i) there exists a mixin `M_b` which is a direct or an indirect base mixin of both `M_1` and `M_2`; (ii) and there exists an input parameter `ip_b` of an ini module in `M_b`, such that both `M_1` and `M_2` contain a required ini module each, `m_1` and `m_2`, and `ip_b` is a transitive output parameter of `m_1` and `m_2`.

We say that a sequence of mixins $\overrightarrow{M}$ is *valid* if `new` $\overrightarrow{M}$ `[`$\overrightarrow{\texttt{par} := \texttt{expr}}$`]` is typable in the type system in [35], for some $\overrightarrow{\texttt{par} := \texttt{expr}}$.

We can now sketch Magda's *safety property*: for every two valid sequences of mixins $\overrightarrow{M_1}$, $\overrightarrow{M_2}$ where each pair of mixins taken one from $\overrightarrow{M_1}$ and one from $\overrightarrow{M_2}$ are not exclusive, any sequence of mixins obtained by combining $\overrightarrow{M_1}$, $\overrightarrow{M_2}$, in such a way the original reciprocal order of the mixins in each sequence is retained, is also a valid sequence.

Intuitively, this property ensures that if there is no ambiguity in the choice of ini modules (non-exclusivity), then any sequence of mixins can be combined with another.

Magda enjoys another important property: as hinted before, Magda's modularity guarantees that client code of a library written in Magda will never break as a consequence of any addition of members to the library's mixins (modulo exclusivity).

The property is as follows: for any set of well-typed mixins and any well-typed client code which uses it, if it is possible to extend a mixin in the set with a new method, or to add a new optional ini module to it, in such a way that the mixin itself is still well typed, Magda guarantees that: (i) all the other mixins in the set and the client code will still be well typed (i.e., they will still compile); (ii) the result of the execution of the client code will not change.

This property does not extend to the case of method override, however this is unavoidable, as override may change the semantics of a method. Magda's features are such that this is the only case in which it happens, as opposed in other languages, where also additions of members can cause unexpected changes.

Both properties can be proved by induction on the type derivation (and on the operational semantics execution tree), as defined in [35].

4 Related work

In this section, we compare our solution with other approaches to reusability of components and we discuss different views to modularization of the initialization protocol. Moreover, we discuss how encapsulation works in Magda.

4.1 Code reuse mechanisms

Multiple inheritance. The most popular implementation of the multiple inheritance is present in C++ [48]. Other versions are present also in Dylan [21], Python [10], and Loglan [34]. However, all of them suffers from the problems discussed in Section 2.1. One of the features of C++ multiple inheritance is private inheritance. This feature also influences multiple inheritance: when a class A inherits from two classes B_1 and B_2, where each of those inherit privately from a class C, then class A will in fact contain two instances of class C which are not visible via the public interface. One of those instances will be visible by the B_1 part of A, while the other one through the B_2 part of A. Private inheritance is not directly available in Magda, however, this could be simulated via object composition.

Traditional mixins. It is believed that one of the main reasons why mixins have not yet achieved a wide acceptance is the *fragile class hierarchies* problem [42], which is avoided in Magda, thanks to our hygienic approach to identifiers. Another difference between Magda and most of the other mixin-based proposals is that in the latter the mixin is a construct which is used to transform one class into another, and classes as well as interfaces still play a significant role in such languages. Instead, in Magda the mixin is the only inheritance construct and it is exploited to create objects from, to reuse code, and to define nominal types. As a consequence, the requirements on the "parametric superclass" in Magda (that is, the base mixin) are also specified using a sequence of mixin names.

CZ. The CZ proposal [40] addresses the diamond problem directly, without restricting the expressive power of multiple inheritance, by eliminating multiple inheritance implementation hierarchies in favour of multiple inheritance subtyping hierarchies. However, this approach forces the programmer to have almost twice as many classes as within a traditional hierarchy, moreover it does not solve the problems about the modularity of constructors.

4.2 Modularization of constructors

Avoiding the initialization protocol. A class can be written using the approach of avoiding explicit initialization protocols by inserting one parameterless constructor (or none), while the real initialization process is implemented in ordinary methods. Unfortunately, with this approach, there is no direct way to check if the object is properly initialized. The programmer may create an object from the class and: (*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; without being warned by the compiler. Dynamic checks can be added by using formal specification languages, like JML [38] and the Design by Contract in Eiffel [41], but the assertion can grow complicated, moreover static checks would be more desirable.

Container parameter. Constructors may have one parameter of a "container" type, such as `Vector` or `Dictionary`. The container contains values of all the initialization parameters,

for instance indexed by their names. Then, the constructor performs a dynamic verification (a static one is impossible), that can become complicated if there are many initialization options.

Container classes. The idea behind the *container classes* design pattern is to use a class for passing the set of parameters used to initialize a property. Such a class must have as many constructors (with their respective parameters) as the possible options of initialization of the given property. The use of such an approach allows one to avoid the problems of: (*i*) exponential growth of the number of constructors; (*ii*) unnecessary code duplication. However, it is necessary to preview the need of the pattern even if in earlier versions of a class there is only one option of initialization for each property. Moreover, it works only in cases when the set of options of initialization of a class coincides with the cartesian product of the initialization options of its properties. It cannot be used in more complicated cases, for example when there is an option of initialization using one value to initialize more than one property (using a container classes each), or when there is the need to initialize a subset of fields, all packaged into one contaneir class.

Optional parameters. Constructors with *default* parameter values (present, for example, in Delphi [2] and C++ [48]) make it possible to declare fewer constructors, being able 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), because one cannot limit which combinations of optional parameters are allowed. This solution works well used together with referencing parameters by their names.

Referencing by name. Another useful feature which can be found in some languages (but not in any of the main-stream ones like Java and C♯) is referencing parameters of methods and constructors by their names. Such approach, present for example, in Flavors [43], Objective-C [33], and Ocaml [39], solves two problems: (*i*) it discards some ambiguities (caused by constructor overloading), because parameters with the same (or compatible) type can have different names; (*ii*) it allows a wider use of default parameter values. 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.

Constructor propagation in Java Layers. The Java Layers language [18] has a feature called *constructor propagation*, which can be illustrated by an example[4].

```
class Class1
{ propagate Class1(String s)  {I_1;}
  propagate Class1(int i)     {I_2;}
}
class Class2<T> extends T
{ propagate Class2(double j)  {I_3;}
  propagate Class2(boolean k) {I_4;}
}
```

Class `Class2<Class1>` has all the combinations of the "propagated constructors" of both classes, which means containing the same list of constructors as the following class.

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

```
class Class3
{ Class3 (String s, double j) {I_1; I_3;}
  Class3 (int i   , double j) {I_2; I_3;}
  Class3 (String s, boolean k) {I_1; I_4;}
  Class3 (int i   , boolean k) {I_2; I_4;}
}
```

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 produce a cartesian product of sets of constructors. In fact, if we want to implement a subclass of a class `C` with the purpose of adding a new option for initializing a property declared in `C`, then we cannot do this using Java Layers. To better understand this, we consider the following example written in Magda, which is an extension with a new palette of the example of the colored cartesian point discussed in Section 2.2, *Multiple initialization options and code duplication*.

```
mixin HSBColorPoint of ColorPoint =
 optional HSBColorPoint(h:float, s:float, b:float)
   initializes (ColorPoint.r, ColorPoint.g, ColorPoint.b)
 begin
   ...
 end;
end;
```

This code, in the Java Layers approach, would require copying all the combinations of the propagated constructors.

Object factories. A recent work concerning initialization protocols is [19]. It shows how object factories can be integrated in a language like Java in such a way that they use the same syntax as normal object creation. Using this approach, it is possible to override the constructors of a class, and even to write an object creation expression of the form `new` I`(...)`, where I is an interface, thus giving more flexibility to the initialization code. As an effect, in some situations this reduces the amount of code which needs to be written. For example, when we want to add one initialization option to an existing class, we just extend the list of the constructors of that class. The benefits of this approach are: (i) the separation of the initialization from the class itself, so that a client instantiating an interface can even not know the implementing class; (ii) the possibility of modifying an initialization protocol in a way in which, for instance, the object returned by `new` expression is an already existing one (not a newly created one). However, in this approach, each set of initialization parameters must correspond to one constructor, therefore, in general, it does not avoid the problem of the exponential growth.

4.3 Dealing with accidental name clashes

The concept of method is realized by three different actions: (i) the introduction of a new method; (ii) the implementation/override of an existing method; (iii) the method call. The bindings between (ii) and (i), as well as between (iii) and (i) are typically made using the method name, which is not guaranteed to be univocal. Additionally, in many popular languages (like Java, C♯ and C++), the distinction between (i) and (ii) is also based upon names. Therefore, modifications of existing classes (even conservative ones) may introduce errors.This can occur even more frequently, and it is more difficult to predict, when the modifications happens in a library written by third-party developers.

In the work [37] we presented three kinds of ambiguity problems (name clashes caused by the implementation of an interface, name clashes caused by the addition of a new method, name clashes caused by mixin application), together with an empirical analysis of the Java standard library (1.5) from the point of view of name clashes.

There are various proposals in the literature to tackle accidental name clashes. Notably, there are languages offering constructs and mechanisms to deal with conflicts: Delphi [2], C♯ [28], Eiffel [41], MixedJava [24], MixGen [3][5], McJava [32, 31] are some examples. An analysis of the main features of these proposals can be found in [37].

4.4 Encapsulation in Magda

Another characteristic which makes our approach different from most of other approaches to component reuse is encapsulation. In Magda, the set of mixins from which an object is created is always visible, since all the references to methods and fields need to be prefixed with the name of the mixin from which the method or field comes. This is in contrast with the choices made in most of the other languages, and at the first sight it may look as a drawback. However, it is often the case that the internal structure of a class is not completely transparent to the user, even though a form of encapsulation is enforced. In C++, for example, in the case of multiple inheritance it is necessary to know if there are common superclassess, in oder to choose one of the semantics of inheritance (private, public or virtual). In the cases of MixGen [3] and MixedJava [24], a class can have many distinct implementations of one method. Then, to call a method declared in a specific superclass or mixin, one has to cast the type of an object to that specific type. In the case of freezable traits [23], to unfreeze a method in a trait or class one needs to know in which supertrait this method has been declared. Additionally, the user needs to be aware of which methods are called in which other methods (as described in Section 2.1), therefore the encapsulation is also violated even at the level of methods.

5 Conclusions

Our purely mixin-based design offers: (*i*) a simple mechanism for reuse based on mixin inheritance, without the problems present in other implementations of the mixin construct (see Section 2.1); (*ii*) initialization protocols that can be composed from independent ini modules coming from different mixins, thus avoiding the drawbacks described in Section 2.2; (*iii*) a mechanism for identifier references using fully qualified names, which guarantees that programs will never fail to compile, or behave unexpectedly, as a result of changes causing name clashes (this way we avoid the problems hinted in Section 4.3). The reuse mechanism, together with the modular constructors and the hygienic identifiers, provides modularity: it is guaranteed that the client code of a library written in Magda will never break as a consequence of any addition of members to the library's mixins.

The hygienic approach to identifiers improve inter-component compatibility. It can be argued, though, that full-path references are lengthy to write and difficult to read, and that this is one main drawback of Magda. However, this can be solved by implementing an IDE tool for expanding short names and hiding long ones on demand. The tool could be based upon the following rule: for each non-hygienic method or field identifier in an expression,

[5] They introduced the notion of *hygienic mixin*.

find the first mixin in the type of the expression such that it contains the introduction of an identifier with the same name, where introduction means the first declaration of the identifier in the mixin hierarchy. In case of ambiguity, all possible valid hygienic hierarchies would be shown.

Acknowledgments. The authors would like to thank the anonymous referees.

References

1. *Visual Basic .NET Language Reference*. Microsoft Press, 2002.
2. *Delphi Language Guide*. Borland Software Corporation, 2004.
3. E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *Proc. OOPSLA '03*, pages 96–114, 2003.
4. E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, and G. L. S. Jr. Project Fortress: A multicore language for multicore processors. *Linux Magazine*, September:38–43, 2007.
5. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification. Technical report, Sun Microsystems, 2008.
6. E. Allen, J. J. Hallett, V. Luchangco, S. Ryu, and G. L. Steele Jr. Modular multiple dispatch with multiple inheritance. In *Proc. SAC '07*, pages 1117–1121. ACM, 2007.
7. 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.
8. D. Ancona and E. Zucca. An algebra of mixin modules. In *Proc. Workshop on Algebraic Development Techniques '97*, volume 1376 of *LNCS*, pages 92–106. Springer-Verlag, 1997.
9. D. W. Barron, editor. *Pascal: The Language and its Implementation*. John Wiley, 1981.
10. D. Beazley and G. V. Rossum. *Python. Essential Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 1999.
11. V. Bono, F. Damiani, and E. Giachino. On traits and types in a Java-like setting. In *Proc. Ifip-TCS '08*, pages 367–382. Springer-Verlag, 2008.
12. V. Bono and J. D. M. Kuśmierek. FJMIP: A calculus for a modular object initialization. In *Proc. FCT '07*, volume 4639 of *LNCS*, pages 100–112. Springer-Verlag, 2007.
13. V. Bono and J. D. M. Kuśmierek. Modularizing constructors. *Journal of Object Technology*, vol. 6 no. 9. Special Issue: TOOLS EUROPE 2007:297–317, 2007.
14. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. ECOOP '99*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.
15. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, 1992.
16. G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Mirand. The Newspeak programming platform. Technical report, Cadence Design Systems, 2008.
17. G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. OOPSLA '90*, pages 303–311. ACM Press, 1990.
18. R. J. Cardone. *Language and Compiler Support for Mixin Programming*. PhD thesis, The University of Texas at Austin, 2002.
19. T. Cohen and J. Gil. Better construction with factories. *Journal of Object Technology*, vol. 6 no. 6 July/August 2007:109–129, 2007.
20. S. Cook. OOPSLA '87 panel P2 varietes on inheritance. In *OOPSLA '87 Addendum to Proc.*, pages 35–40. ACM Press, 1987.
21. I. D. Craig. *Programming in Dylan*. Springer-Verlag New York, Inc., NJ, USA, 1996.
22. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *TOPLAS*, 28(2):331–388, 2006.
23. S. Ducasse, R. Wuyts, A. Bergel, and O. Nierstrasz. User-changeable visibility: resolving unanticipated name clashes in traits. In *Proc. OOPSLA '07*, pages 171–190. ACM, 2007.

24. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL '98*, pages 171–183. ACM, 1998.
25. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
26. M. Gialluca. Modular initialization protocol: a new implementation of the JavaMIP language. Tesi di Laurea Trieannale, Torino University, Dipartimento di Informatica, 2010.
27. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, Sun Microsystems, 2005.
28. A. Hejlsberg, P. Golde, and S. Wiltamuth. *C♯ Language Specification*. Addison-Wesley, 2003.
29. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.
30. T. Kamina and T. Tamai. McJava - a design and implementation of Java with mixin-types. In *Proc. APLAS '04*, pages 398–414, 2004.
31. T. Kamina and T. Tamai. Flexible method combination based on mixin subtyping. *Journal of Object Technology*, 4(10):95–115, 2005.
32. T. Kamina and T. Tamai. Selective method combination in mixin-based composition. In *Proc. SAC '05*, pages 1269–1273, 2005.
33. S. Kochan. *Programming in Objective-C*. Sams, 2004.
34. A. Kreczmar, A. Salwicki, and M. Warpechowski. *LOGLAN '88—report on the programming language*. Springer-Verlag, 1990.
35. J. Kuśmierek. *A Mixin Based Object-Oriented Calculus: True Modularity in Object-Oriented Programming*. PhD thesis, Warsaw University, Departement of Informatics, 2010. Available at http://www.mimuw.edu.pl/∼jdk/mixiny.pdf.
36. J. Kuśmierek and M. Mulatero. The Magda language implementation. Available at http://sourceforge.net/projects/magdalanguage.
37. J. D. M. Kuśmierek and V. Bono. Hygienic methods, Introducing HygJava. *Journal of Object Technology*, vol. 6 no. 9. Special Issue: TOOLS EUROPE 2007:209–229, 2007.
38. G. Leavens and Y. Cheon. *Design by Contract with JML*. 2003.
39. X. Leroy. *The Objective Caml System Release 3.09*. Institut National de Recherche en Informatique et en Automatique, 2005.
40. D. Malayeri and J. Aldrich. CZ: multiple inheritance without diamonds. In *Proc. OOPSLA '09*, pages 21–40, 2009.
41. B. Meyer. An Eiffel Tutorial. Technical Report TR-EI-66/TU, ISE, 2001.
42. L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proc. ECOOP '98*, volume 1445 of *LNCS*, pages 355–382. Springer-Verlag, 1998.
43. D. A. Moon. Object-oriented programming with Flavors. In *Proc. OOPSLA '86*, pages 1–8. ACM Press, 1986.
44. O. Nierstrasz, S. Ducasse, S. Reichhart, and N. Schärli. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, University of Bern, Switzerland, 2005.
45. M. Odersky, P. Altherr, V. Creme, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The Scala Language Specification, version 1.0. Technical report, Programming Methods Laboratory, EPFL, 2006.
46. N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behaviour. In *Proc. ECOOP '03*, volume 2743 of *LNCS*, pages 248–274. Springer-Verlag, 2003.
47. C. Smith and S. Drossopoulou. Chai: Traits for Java-like Languages. In *Proc. ECOOP '05*, volume 3586 of *LNCS*, pages 453–478. Springer-Verlag, 2005.
48. B. Stroustrup. *The C++ programming language*. AT&T, 1997. Third edition.