Object Metamorphism

Type-Safe Modeling of Protean Objects in Scala

Zbyněk Šlajchrt
The Department of Information Technologies
University of Economics
Prague, Czech Republic
zslajchrt@gmail.com

***Abstract*—Modeling protean objects, i.e. objects adapting their structure and behavior dynamically with respect to changeable environment, may be challenging in traditional object oriented languages. While some dynamic languages make the implementation of changeable behavior of objects possible by injecting code into the objects at run-time, their lack of an in-depth compile-time code analysis makes the resulting code fragile. The concept of object metamorphism (OM) targets the gap between the two language types by introducing a declarative modeling of protean objects. Such a model, which is validated at compile-time, defines all possible compositions of a given object from smaller parts represented by traits. The actual composition used to instantiate the object is chosen dynamically with respect to the current environment. The result of this research will provide the theoretical concept of OM along with a proof-of-concept adoption of OM to Scala.**

Keywords—dynamic languages; static languages; object-oriented design; Java; Scala; Groovy; subject-oriented programming; data-context-interactions paradigm (DCI); mixins; static program analysis; union types; hybrid dynamic and static typing

I. Introduction

Object-oriented programming (OOP) has proven successful in a wide area of applications mainly because it makes easier to model entities from real domains and the relationships among them. Although it has become a standard in today's application development, there are still some programming problems, which are difficult to solve by means of the current object-oriented languages. In this paper I am focusing on a subset of such problems emerging from the gray zone where neither statically nor dynamically typed languages fit fully the problems. In particular, this paper explores two scenarios, which the traditional OOP languages fail to fully support.

The first problem occurs in applications where the exact type and composition of an object is not known until the moment of its instantiation. The second problem occurs when the type and composition of the object may vary during its lifetime.

Although there is always a way to solve the above-mentioned problems in both kinds of languages, a developer must usually make some design concessions, which lead at least to degrading some non-functional properties of the program, such as extensibility, coupling and reusability.

The concept of object metamorphism presented in this paper attempts to overarch the gap between the static and dynamic languages by a sort of "controlled" or "statically checked" dynamism applicable in the domain of statically typed languages.

OM targets the systems consisting of objects whose structure and behavior may be composed of fragmentary components, whereas the number of possible compositions may be large. An example may be a system modeling the manifestation of emotions in the human face. A relatively small number of face muscles are able to produce thousands of emotional expressions, however not every combination of the muscles represents an emotional expression. The task of such an application might be to guess the emotion according to the activity of individual muscles or, contrarily, to control virtual muscles in a dummy according to a given emotion.

Another example might be a complex service, which must apply a number of optional security constraints, whereas some may depend on others. The selection of the proper constraints is done according to the properties of the user. Such a system of constraints may also generate a large number of combinations.

In both examples OM builds a morph model producing all possible combinations, which are verified by checking their dependencies at compile-time.

The research presented in this paper is built on the ideas from Subject-oriented programming (SOP) [1][2] and the Data-Context-Interactions paradigm (DCI)[3]. In these paradigms the objects are seen as capable of assuming different forms with respect to the context or use-case. Such forms are called subjects in SOP and roles in DCI.

II. Example

Let us imagine an airport luggage scanner capable of recognizing three and two-dimensional objects in baggage by their geometrical properties and material. The scanner outputs each scanned item in a form sketched in Fig. 1.

```
{
   "id": 0,
   "shape": "cylinder",
   "material": "metal",
   "x": 234.87,
   "y": 133.4,
   "z": 12.94,
   "radius": 13.45,
   "height": 0.45,
   "density": 3.8
}
```

Fig. 1. Scanner data sample

For the sake of simplicity let us assume that besides the common attributes, such as the item's position, the scanner recognizes two shapes – cuboid and cylinder – and two materials – paper and metal. It follows that the record emitted by the scanner may take on four forms corresponding to the Cartesian product of the shapes and materials sets. It also follows that with the increasing number of independent dimensions, added as the scanner is improved, the number of record forms exponentially grows.

Now let us suppose that we have to write a program reading the item records and represent them as objects. We will examine how the three established OO languages Java, Scala and Groovy perform in modeling the scanner data.

A. Composition in Java

The most natural approach to model the items is to use composition to model the item as a two-part composite and delegation to expose the common methods from both dimensions (i.e. `volume()` and `density()`). The Fig. 2 depicts the UML diagram of such a design.

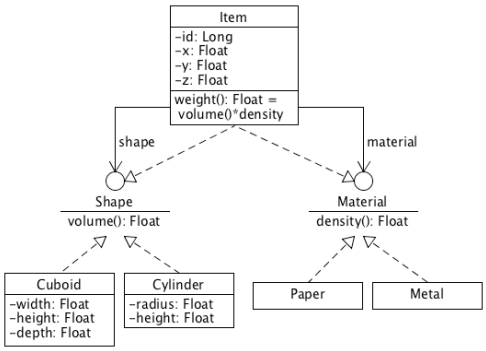

Fig. 2. The item model using composition

Then instantiation of an `Item` instance initialized with the data from the Listing 1 would look like this:

```
Item item = new Item(0, 234.87, 133.4,
12.94, new Metal(3.8),new Cylinder(13.45,
0.45));
```

Fig. 3. Composing the item instance in Java

At first sight everything seems to be all right; the item object encapsulates all data from the record and we can invoke the common methods, such as `volume()`, on the item. But what if we wanted to determine whether the item is a metal cylinder, for instance? One could naturally tend to use the `instanceof` operator, since it is used to detect what an object **is**.

```
boolean isMetal = item instanceof Metal;
boolean isCyl = item instanceof Cylinder;
```

Fig. 4. Attempting to determine the type of the item

However, both previous statements are always false, because the objects, which may be instances of the two particular types, are wrapped as components of the `Item` class and hence they are not part of the item's type. In order to make possible to determine the item's real type from the item itself we would have to incorporate a custom management of the item types into the `Item` class. Such a solution would be a mere workaround fixing the deficiency of the Java type-system.

In other words, Java forces a developer to model a trait of an object (i.e. "is-a" relationship) as if the trait were a component of the object (i.e. "has-a" relationship).

Sometimes the choice of the relationship depends on the subject's perception. For example the object in Figure X may be modeled as a *wine bottle box* or a *wine bottle in a box*.

Fig. 5. Do we see a wine bottle box or a wine bottle in a box?

In the former case we see primarily a box containing a wine bottle ("has-a"), while in the latter case we perceive the box as a decoration or a "trait" of the wine bottle ("is-a"). The problem we face in Java is that we must always model this situation as the wine bottle box [4].

Such a design inherently suffers from the complication called *object schizophrenia*. The name suggests that the object's identity is scattered across the wrapper and the components (there are more "selves")[5]. This complication may be illustrated on the following example. Let us assume that some items need to be additionally corrected because of a temporary malfunction of the scanner resulting in wrong measurement of material density. This defect may be corrected in the application by decorating the affected items by the `DensityCorrection` implementation of `Item`.

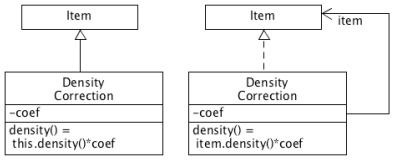

Fig. 6. `DensityCorrection` extension designed by inheritance and composition

Fig. 6 shows two possible implementations of such a correction class. The left uses inheritance while the right uses composition. Let us compare the two designs with respect to the behavior of the `weight` method defined in `Item`.

Since the correction and the item share the same identity (self), the `weight` method in the left design reflects the overridden density by `DensityCorrection` and returns the corrected weight. On the contrary, the `weight` method invoked on the correction in the right case will return the original value instead of the corrected one, since the method invocation is delegated to the wrapped item where the method calculates the weight according to the original density, which is presumably incorrect behavior.

Thus the solution based on inheritance fits better to the problem. Unfortunately, it is not possible to arbitrarily stack more such dedicated extensions by means of inheritance due to the single inheritance in Java.

The concept of traits overcomes this limitation. It makes possible to stratify an arbitrary number of extensions of a class while avoiding the multiple inheritance issues, such as the diamond problem [6], by the so-called *trait linearization* [7]. Let us examine how the problem of modeling multidimensional data may be solved in Scala and Groovy, i.e. the languages, which adopted the concept of traits.

B. Static Traits in Scala

In Scala, we can model the item quite straightforwardly; every piece of it is expressed as a trait.

```
trait Shape {
   def volume: Float
}
trait Cylinder extends Shape {
   def height: Float
   def radius: Float
   def volume = PI * radius * radius *
height
}
trait Material {
   def density: Float
}
trait Metal extends Material {}
trait Item {
   this: Shape with Material =>
   val id: Int
   def pos: (Float, Float, Float)
   def weight: Float = density * volume
}
```

Fig. 7. Item elements modeled by traits in Scala

The item instance is then composed from the right traits at the moment of its creation. The initialization of the abstract members is done at the same time.

```
val item = new Item with Metal with
Cylinder {
   val pos = (234.87, 133.4, 12.94)
   def density = 3.8
   def radius = 13.45
   def height = 0.45
}
```

Fig. 8. Creating and initializing a metal cylinder in Scala

The item constructed this way may be tested by means of `isInstance` to determine if it is a metal cylinder, for example.

```
val isMetalCylinder =
 item.isInstanceOf[Metal with Cylinder]
```

Fig. 9. Testing whether the item is a metal cylinder in Scala

This approach also eliminates the problem with object schizophrenia, since we can design the `DensityCorrection` extension as a trait, which will share its identity with the item. This trait will be combined with the other traits only if the input data should be corrected.

Although it seems that the approach using Scala traits has coped with all problems identified above, there is in fact looming another serious problem relating to the number of all possible combinations of the item's forms. Each combination would have to be tested individually by a separate condition. Furthermore, there would have to be one class for each combination of traits. Thus the code and the classes would quickly become unmaintainable since the number of combinations grows exponentially with every additional dimension.

Since this problem is rooted in the static nature of Scala, let us examine, if the problem persists in Groovy, which provides a dynamic version of traits.

C. Dynamic Traits in Groovy

In Groovy, it is possible to attach traits to an object at run-time. Every object provides the **withTraits** method for adding one or more traits to the object. This feature allows us to initialize the item in a step-by-step way, i.e. per feature and not per combination. (Since the traits themselves are defined in a similar way as in Scala they are not showed because of the limited space in this paper.)

```
def item = new Object() as Item

item.x = itemData.get("x")
item.y = itemData.get("y")
item.z = itemData.get("z")

// Shape dimension
if (itemData.containsKey("cylinder")) {
    item = item.withTraits(Cylinder)
    item.radius = itemData.get("radius")
    item.height = itemData.get("height")
} else if (event.containsKey("cuboid")) {
```

```
    item = item.withTraits(Cuboid)
    item.width = itemData.get("width")
    item.height = itemData.get("height")
    item.depth = itemData.get("depth")
}
// Material dimension
if (itemData.containsKey("metal")) {
    item = item.withTraits(Metal)
…
```

Fig. 10. A step-by-step assembling in Groovy

As in the Scala case, also here we can use the **instanceof** operator to determine what the item really is

```
def isMetal = item instanceof Metal
def isCylinder = item instanceof Cylinder
```

Fig. 11. Testing whether the item is a metal cylinder in Groovy

Also the `DensityCorrection` trait could be implemented similarly.

Nevertheless, the most important difference is that we have gotten rid of the combinatorial explosion of classes and code, which is the consequence of the dynamic addition of traits allowing constructing the item per feature and not per combination.

Unfortunately, also this approach suffers from some serious issues resulting from the step-by-step approach as well as from the dynamic nature of the language and its weak type system:

- **Incompleteness**: The configuration procedure may forget to choose a trait for some dimension
- **Redundancy**: The **withTraits** method can possibly add two mutually exclusive traits from the same dimension to the builder
- **Missing Dependencies**: A trait from one dimension may depend on another trait from another dimension. The configuration procedure must take these inter-trait dependencies into account, because it is beyond the capabilities of dynamic languages. This additional requirement makes the code fragile and open to inconsistencies.
- **Ambiguous Dependencies**: The configuration procedure must also guarantee that there is no ambiguity in the dependencies.

D. Summary

The traditional approach to model multidimensional data objects uses composition. The composition produces an object (item) by wrapping other objects (material, shape), which can assume various forms. The number of wrapped objects corresponds to the number of dimensions. For each dimension the top object exposes the corresponding interface by means of delegation.

Composition hides the real shape (i.e. type) of the object. We cannot determine from the top object's type that it is a metal cylinder, for example. Instead, we just find out that the item is something of some shape and material. To determine the real type, one cannot use the `instanceof` operator only. Instead, s/he must resort to examining the object's attributes, i.e. the state, holding references to the wrapped objects (`getMaterial()`, getShape()) and additionally apply `instanceof` to each wrapped instance. Alternatively, the classes may be equipped with a custom type-management complementing the platform's type system.

Next, the identity of the object is scattered across the object and its components, which results in object schizophrenia. In Java there is virtually no way to cure this problem, however the concept of traits adopted by languages such as Scala or Groovy naturally solve the problem.

However, modeling multidimensional objects by means of static traits in Scala leads to the exponential explosion of code making the static traits practically unusable.

Groovy's capability to apply traits to an object at run-time efficiently solves the problem of the exponential explosion. This approach however also suffers from a couple of serious issues outlined above.

We may, therefore, conclude that the outlined problems cannot be solved ideally if they are to be implemented either in a static or dynamic OO language only. It seems that the gap between the two types of language should be filled by a static/dynamic approach combining the benefits of both language types. The following section presents one such a hybrid concept called *object metamorphism*, which has been developed by the author.

III. Object Metamorphism

A. Introduction

Object metamorphism may be defined as a capability of an object to assume one or more forms defined by the object's *morph model*, which defines all possible form *alternatives* of the object. The compiler analyses the morph model and performs various checks to guarantee that all alternatives are consistent. Every alternative consists of the so-called *fragments*, which are the building blocks in multidimensional compositions of objects and correspond semantically to the concept of trait as defined in Scala or Groovy. A morph fragment represents a typological, behavioral and structural element of multifaceted objects.

The run-time component of OM instantiates one of the alternatives selected with the assistance of the so-called *morphing strategy*, which usually selects the alternative according to the context or input data. The instance of the alternative, which is called a *morph*, may be subject to further re-morphing, in case the context changes and the morph should reflect the change (the morph keeps a link to its morph model).

The theoretical concept of OM is accompanied by a reference implementation (RI), whose main goal is to serve as a proof-of-concept tool. RI is designed as an extension of the Scala compiler and may be freely downloaded from [8]. It should be remarked that the reason why Scala was selected as the platform for RI and not Groovy, for example, is based on the experience that it should be easier to add some dynamicity

to a static language than to introduce some static type checking into a dynamic language.

In order to elucidate the idea of OM we will try to apply it to the example, which we have gone through in the previous section. We will use RI to implement the example.

B. Using OM

Let us begin with designing the morph model of the scanner items. In RI, morph models are defined by means of a special type expression based on the syntax for trait composition in Scala. RI extends the Scala type system by introducing the *disjunctor* type `or`, which may be seen as a union operator. The disjunctor type is in fact a generic type with two parameters `or[X, Y]`, which may be written as `X or Y`. This "syntax sugar" makes the type expressions more comprehensible and allows us to express the morph models as logical formulas.

Using the disjunctor we can define the morph model type for the item as follows:

```
Item with (Metal or Paper) with (Cuboid
or Cylinder)
```

Fig. 12. The morph model type of the item

This is an almost human-like expression of all possible compositions of the item. In order to translate it to a more machine-like form, we can treat it as a logical formula and transform it to the equivalent *disjunctive normal form* (DNF):

```
(Item with Metal with Cuboid) or
(Item with Metal with Cylinder) or
(Item with Paper with Cuboid) or
(Item with Paper with Cylinder)
```

Fig. 13. The DNF of the item morph model

The DNF clearly reveals all four alternatives produced by the morph model.

Another useful term is the *lowest upper bound* (LUB) of the morph model. It is the most specific type yet compatible with the types of all alternatives from the morph model. It follows that the LUB of the item morph model is

```
Item with Material with Shape
```

Fig. 14. The LUB of the item morph model

This type actually corresponds to the type that we obtained when modeling the item by composition.

The morph model type expression is used in the application code as the type argument to various macros such as `compose`, which creates the so-called *morph assembler*.

```
val itemAsm = compose[Item with (Metal or
Paper) with (Cuboid or Cylinder)]
```

Fig. 15. Creating the item assembler

The morph assembler is responsible for generating classes for all alternatives in the morph model and for creating morphs from these classes. The `morph` method creates a new morph according to the suggestion made by the morphing strategy passed as the argument. The return type of that method is the LUB of the morph model.

```
val item = itemAsm.morph(strategy)

assert(item.isInstanceOf[Item with
Material with Shape])
```

Fig. 16. Creating the item morph

In case the morph should be updated according to the current context, one may invoke the `remorph` method on it. The item will get reassembled to the alternative selected by the strategy specified upon the morph's creation.

```
item.remorph()
```

Fig. 17. Remorphing the item morph

Now let us turn our attention to the validations carried out by the compiler extension. One of such validations is checking dependencies. The dependencies are specified in RI by means of the self-type of the fragment trait as illustrated on the Item fragment, which depends on the `Material` and `Shape` dimensions:

```
trait Item {
   this: Shape with Material =>
…
```

Fig. 18. The item's dependencies expressed as the self-type

The compiler extension must verify that all alternatives containing fragment `Item` must also contain fragments implementing `Shape` and `Material`. The alternatives from the model mentioned above clearly satisfy this condition. However, the following model is not valid, since its alternatives have no shape fragment as required by `Item`.

```
Item with (Metal or Paper)
```

Fig. 19. A model with missing dependencies

On the other hand, the compiler must also check that there is no ambiguous dependency. The next model contains two ambiguous dependencies `Metal` and `Paper` that both appear in same alternatives.

```
Item with Metal with Paper with (Cuboid
or Cylinder)
```

Fig. 20. A model with redundant fragments Metal and Paper

C. Morph Model Reification

As long as the compiler extension detects no error in the model, it performs the so-called *reification* transforming the model into a special syntax tree, which is injected as a synthetic variable into the syntax tree of the assembler being produced by the macro.

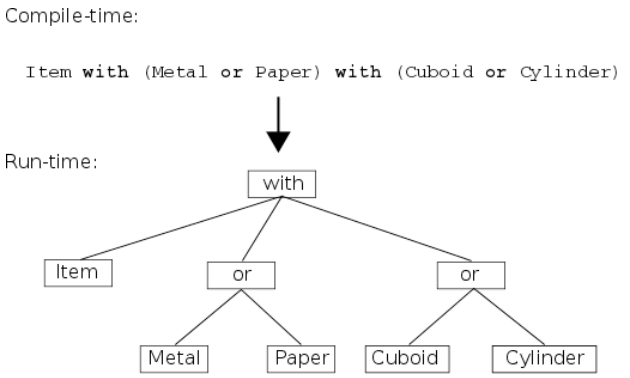

Fig. 21. Morph model reification

Thanks to the reification, the morph model may be used at run-time by the assembler. The model is stored in the `model` attribute of the assembler and is publicly accessible as shown in the following statement printing the model tree.

```
println(itemAsm.model.rootNode)
```

Fig. 22. The reified morph model is accessible through the `model` attribute of the assembler

The output would look like this:

```
ConjNode(List(FragmentNode(0),DisjNode(Li
st(FragmentNode(1),FragmentNode(2))),Disj
Node(List(FragmentNode(3),FragmentNode(4)
))))
```

Fig. 23. Textual form of the reified morph model

RI implements an algorithm for transforming a morph model to the corresponding disjunctive normal form, which is used to extract the model alternatives. This functionality is available through the alternative iterator, which unfolds the model into a series of alternatives. The code in Fig. 24 shows how to print all alternatives in the model.

```
val ai = itemAsm.model.altIterator
while (ai.hasNext) {
  println(ai.next)
}
```

Fig. 24. Using the alternatives iterator to print the alternatives

The first line of the output should be:

```
List(FragmentNode(0,false),FragmentNode(1
,false),FragmentNode(3,false))
…
```

Fig. 25. Textual form of the alternatives

The ordering of alternatives is determined by the structure of the morph model tree. We can attach indices to the fragment nodes increasing from left to right:

```
Item(0) with (Metal(1) or Paper(2))
   with (Cuboid(3) or Cylinder(4))
```

Fig. 26. Indexing the fragments in the morph model

Then the set of alternatives can be written as this: `{(0, 1, 3), (0, 2, 3), (0, 1, 4), (0, 2, 4)}`. It allows us to define the ordering of the alternatives in the set quite easily just by comparing indices from right to left between two alternatives. The comparison of two alternatives goes from the common right-most index to the left.

In other words, given two alternatives `A1={a1, b1, c1}` and `A2={a2, b2, c2, d2}`, then `A1` precedes `A2` if and only if relation `(c1 < c2) || ((c1 == c2) && (b1 < b2)) || ((c1 == c2) && (b1 == b2) && (a1 < a2))` is true.

Applying this rule to our morph model, we obtain the following ordering of the alternatives:

```
{0, 1, 3} < {0, 2, 3} < {0, 1, 4} < {0,
2, 4}
```

Fig. 27. Ordering the alternatives

D. *Morhing Strategies*

A morphing strategy is a key concept in OM determining the composition of morphs. Whenever a new morph is created or re-morphed, a morphing strategy must be supplied. Although it is possible to implement custom strategies, in practice only three are of a particular importance: *promoting*, *masking* and *rating* strategies. To select the winner alternative the three strategies use distinctive methods, which may be freely combined. The following paragraphs will shortly introduce these strategies.

1) Promoting Strategy

Promoting strategy swaps model tree branches in such a way that the promoted (i.e. preferred) alternatives get as high in the generated list of alternatives as possible. To illustrate this method let us use a simplified morph model consisting of four fragments only:

```
 (Metal or Paper) with (Cuboid or
Cylinder)
```

Fig. 28. A simplified version of the item morph model

This model can be depicted as a tree shown on the following figure.

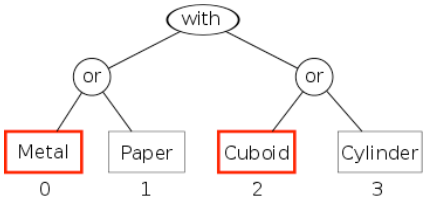

Fig. 29. The default morph model tree

The model has two dimensions that correspond to the two disjunctors (or) in the model. Each disjunctor contains two alternative fragments, thus the number of all alternatives defined by this model is four. A very important aspect is the order of the alternatives in the list since the **first alternative in the list is by default taken as the winner**. According to the rules described above the list of the alternatives generated by the model will be sorted in this way:

```
1. {0, 2}
```

```
2. {1, 2}
3. {0, 3}
4. {1, 3}
```

Fig. 30. The default list of alternatives

The goal of the promoting strategy is to "promote" one or more independent fragments (i.e. fragments under different disjunctors), which means to transform the morph model tree in such a way that the promoted fragments are present in the first alternative in the list generated from the tree, i.e. the default-winning alternative.

We will use a new notation `(materialCoord, shapeCoord)` to denote the fragments to be promoted, where each coordinate reflects the zero-based index of the referred fragment under the given disjunctor. Then `(0,0)` is the default promotion leading to choosing the `{Metal, Cuboid}` alternative.

In order to promote fragments referred by the coordinates `(1, 0)` in Fig. 29, the promoting strategy must transform the tree into the form depicted in Fig. 31; in other words, to swap Paper and Metal fragments in order for Paper to have index 0.

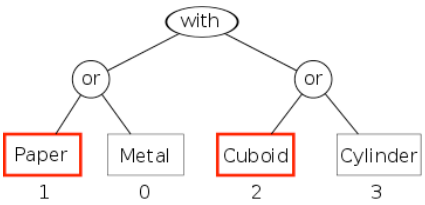

Fig. 31. The Paper fragment promoted

The new tree yields this list of alternatives.

```
1. {1, 2}
2. {0, 2}
3. {1, 3}
4. {0, 3}
```

Fig. 32. The list of alternatives with the Paper fragment promoted

We can continue by promoting coordinates (1, 1) in Fig 29. The corresponding tree will be the one in Fig 33.

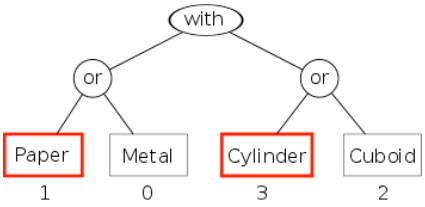

Fig. 33. Additional promotion of the Cylinder fragment

The list of alternatives corresponding to the tree in Fig 33 is:

```
1. {1, 3}
2. {0, 3}
3. {1, 2}
4. {0, 2}
```

Fig. 34. The list of alternatives after promoting Paper and Cylinder

And finally the remaining combination, which is described by coordinates `(0, 1)` from Fig 21, gives the tree in Fig. 35 producing the alternatives listed in Fig. 36.

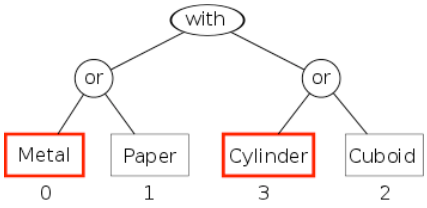

Fig. 35. The model tree after promoting Metal

```
1. {0, 3}
2. {1, 3}
3. {0, 2}
4. {1, 2}
```

Fig. 36. The list of alternatives after promoting Metal

2) Masking Strategy

In contrast to the promoting strategy, which only resorts the generated list of alternatives, the *masking strategy* is destructive as it effectively eliminates alternatives containing unwanted fragments. Masking allows suppressing alternatives, which do not match the so-called *fragment mask*. The fragment mask specifies which fragments may be present in the winning alternative. By turning some fragments off we can effectively reduce the set of the alternatives. By default, all fragments are turned on.

In order to make the explanation more understandable, let us introduce the following notation, which represents an alternative along with a bit vector indicating the fragments contained in the alternative.

```
{fragments}[MetalBit, PaperBit,
CuboidBit, CylinderBit]
```

The four alternatives from our model then look like this:

```
1. {0, 2}[1, 0, 1, 0]
2. {1, 2}[0, 1, 1, 0]
3. {0, 3}[1, 0, 0, 1]
4. {1, 3}[0, 1, 0, 1]
```

Fig. 37. The default mask status

Now, the fragment mask for our model is a vector of four bits. A bit `fN` set to `1` indicates that the corresponding fragment may be present in the winning alternative.

```
mask = (f0 f1 f2 f3)
```

We can use this mask to reduce the original list of alternatives `A` by clearing some bits in the mask. The sublist `S ⊂ A` consists of all alternatives whose bit vector `a` masked with the fragment mask (bitwise AND) gives `a`. It can be formulated more precisely by the following formula.

$$S = \{a \in A;\ a \,\&\, mask = a\}$$

For a better manipulation with the fragment bits, let us construct the following matrix corresponding to the previous list of alternatives.

$$F = \begin{vmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{vmatrix}$$

Fig. 38. The mask status matrix

When no fragment is explicitly specified, the mask is `(1 1 1 1)`. Masking matrix `F` gives the same matrix. Thus there is no reduction of the original list of alternatives.

$$F \,\&\, mask = \begin{vmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{vmatrix}$$

If we wish to constrain the list of alternatives only to those **not** containing the `Cylinder` fragment, the mask will be `(1 1 1 0)`. Masking matrix `F` then will yield the following matrix.

$$F \,\&\, mask = \begin{vmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix}$$

Here, only the first two rows remain same after the mask is applied. The first alternative in the sublist, which preserves the order of the original list, is the winner.

1. **{0, 2}[1, 0, 1, 0]**

2. {1, 2}[0, 1, 1, 0]

Fig. 39. Remaining two alternatives after the masking

If we use the mask to clear an additional fragment, let us say `Paper`, the mask will be `(1 0 1 0)`. Then the masked matrix `F` will be:

$$F \,\&\, mask = \begin{vmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

Only the first row remains unchanged, which becomes the winner.

1. {0, 2}[1, 0, 1, 0]

3) Rating Strategy

The *rating strategy* is used to rate individual alternatives. It may be used as a complement to the promoting strategy to fine-tune the resulting list of alternatives. The winner alternative is selected only from those alternatives having the highest rating with respect to their order. For a better understanding let us use the following notation, which depicts a rating of an alternative.

`{fragments}:Rating`

By default the alternatives have no rating, i.e. their rating is 0. The four alternatives from our model then look like this:

1. {0, 2}:0
2. {1, 2}:0
3. {0, 3}:0
4. {1, 3}:0

Fig. 40. The default list of alternatives with the default rating

We can explicitly assign new rating to a selected sub-set of the original alternatives. Let's say we will rate alternatives 1 and 4 by 1. Then the sub-list will look like that in Fig. 41 since only the two alternatives have the highest rating. The winner is the first alternative in the sub-list.

1. {0, 2}:1

2. {1, 3}:1

Fig. 41. The remaining alternatives having the same highest rating

4) Summary

To wrap it up, the masking strategy is used to eliminate unsuitable alternatives. If more non-orthogonal masking strategies are chained there is a risk that there will be no alternative left to create the morph.

On the other hand, the promoting strategy is used to recommend suitable alternatives, i.e. promote them higher in the list, and no alternative is removed.

The rating strategy may be also used to shrink the list of the alternatives by means of rating the alternatives.

A key property of morphing strategies is their *composability*; they do not have to provide the definite decision on which alternative is the winner. Instead, a strategy may constrain its decision on a certain sub-model of the main morph model. The final strategy then may be composed of several partial strategies, whose sub-models are orthogonal and complete. In other words, the individual partial decisions are not mutually contradicting and unambiguous when combined. I will illustrate this topic on the following example.

It follows from the above that we do not have to create one big strategy selecting the winner alternative with respect to the whole morph model. Instead, we can create two partial strategies, one for the material dimension and the other for the shape dimension. The orthogonal sub-models of those strategies will be `Metal or Paper` and `Cuboid or Cylinder`.

The partial masking strategy suggesting the winner in the Shape dimension is created by means of the `mask` macro:

```
val shapeStrat = mask[Cuboid or
Cylinder](itemData.get("shape") match {
  case "cuboid" => 0
  case "cylinder" => 1
})
```

Fig. 42. Creating the mask strategy for the shape dimension.

The number returned by the match block represents the index of an alternative in the morph model specified as the type argument of the `mask` macro.

This macro is invoked with the sub-model type and a function returning the index of the winning alternative in the sub-model. The function returns 0 if the input item data contains cuboid and 1 if it contains cylinder.

The partial strategy is constructed similarly except it is chained with the shape strategy, which is passed as the first argument to the macro.

```
val materStrat = mask[Paper or
Metal](shapeStrat,
  itemData.get("material") match {
    case "paper" => 0
    case "metal" => 1
})
```

Fig. 43. Creating the chained mask strategy for the material dimension.

The two strategies are complementary; each selects one pair of alternatives from the main model, while the two pairs have always one common alternative.

The material strategy linked to the shape strategy may be used as a complete strategy for creating the item morph.

```
val item = itemAsm.morph(materStrat)
```

E. Determining Morph Type

RI uses a special macro for determining the type of a morph. The `select` macro is invoked with the type to be tested and with the morph instance. The macro returns optional result, i.e. `None` or `Some` in case the item conforms to the tested type.

```
select[Metal with Cylinder](item) match {
   Some(metalCylinder) =>
   None =>
}
```

Fig. 44. The `select` macro allows determining the type of the morph

The `select` macro provides additional static type safety, since it can reveal invalid types passed as the argument. As long as the type argument is incompatible with the morph model of the morph instance, the macro aborts the compilation. The following statement would not compile because the item can never be compatible with `Metal with String`.

```
select[Metal with String](item)
```

Fig. 45. An invalid invocation of the `select` macro.

F. Wrappers

OM distinguishes fragments, which implement the core behavior, from *wrappers*, which are used to implement stackable modifications of the underlying traits. There are two types of wrappers: *dimension* and *fragment wrappers*. The difference lies in the way they override their inherited methods. A dimension wrapper is designed to override a dimension, i.e. abstract methods only (by means of abstract override), while a fragment wrapper is used to override a fragment's methods (by means override).

The `DensityCorrection` extension is an example of a dimension wrapper, since it overrides the `density` method defined in the `Material` dimension trait. In RI the `DensityCorrection` wrapper would be implemented as shown in the following listing.

```
@dimension @wrapper
trait DensityCorrection extends Material{
   val coef: Float
   override def density() = coef *
      super.density()
}
```

Fig. 46. The `DensityCorrection` extension as a dimension wrapper

The wrapper should be incorporated into the item morph model as an optional fragment, since it is to be applied to some items only. Optional fragments are expressed as a disjunction of the fragment and the neutral type `Unit`, i.e. `(Unit or Fragment)` or `(Fragment or Unit)`.

The order determines whether the fragment will be by default active or not, since the left operand has a lower index and thus it will be in higher alternatives.

```
val itemAsm = compose[Item with (Metal or
Paper) with (Cuboid or Cylinder) with
(Unit or DensityCorrection)]
```

Fig. 47. Incorporating `DensityCorrection` to the item morph model

In order to activate the wrapper selectively with respect to the input data, we have to create an additional morphing strategy, which will activate the wrapper for some item data only. This new strategy will be chained with the shape strategy and become the top strategy.

```
val densCorStrat = mask[Unit or
DensityCorrection](shapeStrat,
  if (shouldCorrect(itemData)) 1 else 0)
```

Fig. 48. The mask strategy controlling the presence of `DensityCorrection` in morphs.

Note: By adding the optional wrapper we made the morph model three-dimensional.

G. Fragment Factories

In RI, a morph instance is in fact a dynamic proxy delegating invocations to the so-called *fragment stubs*. These fragment stubs correspond to the fragments in the model alternative used as the template for creating the morph. By default, RI uses default stub factories, which are analogous to default constructors, i.e. they do not allow passing arguments. The default stub factory creates a new stub instance on every morph instantiation. If one wishes to reuse the same stub instance when re-morphing, s/he can override the default stub factory by declaring an implicit variable prior to the statement creating the assembler. This implicit variable holds an alternative stub factory, such as one created by the `single` macro, which produces the singleton factory.

```
implicit val cylFact = single[Cylinder]

val itemAsm = compose[Item with (Metal or
Paper) with (Cuboid or Cylinder)]
```

Fig. 49. Overriding the default fragment factory.

The same mechanism may be used to initialize the fragment stub by some external data, i.e. to mimic a constructor with arguments. There is an overloaded `single` macro with one argument for the so-called *configuration object*. The following listing illustrates how to pass a configuration object to the singleton factory creating the `Cylinder` fragment stub. The configuration object is actually injected into the new stub instance in order for the corresponding stub's methods to delegate on it [9].

```
implicit val cylFact = single[Cylinder](
CylinderInitData(itemData.get("radius"),
                 itemData.get("height")))
```

Fig. 50. Supplying the configuration object to the fragment factory.

The configuration object must implement an interface, which is also implemented by the fragment. In this case the common interface is `CylinderData`. There is also a helper case class `CylinderInitData` implementing it to facilitate the creation of the configuration object.

```
trait CylinderData {
   val radius: Float
   val height: Float
}

case class CylinderInitData(radius:
Float, height: Float) extends
CylinderData
```

Fig. 51. The configuration interface and the helper configuration case class.

The `Cylinder` fragment trait extends `CylinderData` decorated with the `dlg` marker to indicate that the abstract methods from the interface are to be automatically implemented by delegating on the configuration object injected on the creation of the fragment proxy.

```
trait Cylinder extends Shape with
dlg[CylinderData] {
   def volume = PI * radius * radius *
height
}
```

Fig. 52. The `Cylinder` fragment trait declaring the configuration interface.

H. Putting the pieces together

The following listing wraps up the pieces scattered across the preceding text. The procedure overrides the default fragment stub factories to initialize the stubs with the input data and then it creates the item assembler and chains three partial strategies, which are used to create the item morph instance. In the end, the item instance is tested whether it is a metal cylinder.

```
implicit val cylFact = single[Cylinder](
CylinderInitData(itemData.get("radius"),
                 itemData.get("height")))

implicit val cubFact = single[Cuboid](
CuboidInitData(itemData.get("width"),
                 itemData.get("height"),
                 itemData.get("depth")))

implicit val metFact = single[Metal](
MetalInitData(itemData.get("density"))

implicit val papFact = single[Paper](
PaperInitData(itemData.get("density"))

val itemAsm = compose[Item with (Metal or
Paper) with (Cuboid or Cylinder) with
(Unit or DensityCorrection)]

val materStrat = mask[Metal or
Paper](itemData.get("material") match {
  case "metal" => 0
  case "paper" => 1
})

val shapeStrat = mask[Cuboid or
Cylinder](materStrat,
  itemData.get("material") match {
    case "cuboid" => 0
    case "cylinder" => 1
})

val densCorStrat = mask[Unit or
DensityCorrection](shapeStrat,
  if (shouldCorrect(itemData)) 1 else 0)

val item = itemAsm.morph(densCorStrat)

select[Metal with Cylinder](item) match {
   Some(metalCylinder) =>
      // use the metal cylinder
```

```
    None =>
}
```

Fig. 53. Putting the pieces together

IV. Other OM Features

Because of the limited space in this paper I could not present in detail other interesting features of OM. If the reader is interested in them s/he may visit the home site of the project [8], where a lot of additional resources are available. In this section I am going to briefly explain only one of them.

A. Extending Morph Models

OM allows extending existing morph models by other models. The extending model is typically incomplete, which means that not all alternative it produces are valid because of missing dependencies. The model being extended is supposed to deliver those missing dependencies.

To illustrate the extension of a morph model let us consider a model describing currencies recognized among the scanned items. There would be two types of currencies, coins and banknotes, represented by fragments `Coin` and `Banknote` extending the dimension trait `Currency`.

```
trait Currency {
  …
}

trait Coin extends Currency {
   this: Metal with Cylinder =>
  …
}

trait Banknote extends Currency {
   this: Paper with Cuboid =>
  …
}
```

Fig. 54. Currency traits

Assuming that every coin corresponds to a metal cylinder item, we express this relationship by the self-type of the Coin fragment. In a similar way we express the binding between banknotes and paper cuboids (very thin).

The morph model for the currencies among items is very simple:

```
type CurrencyModel =
   Coin or Banknote or Unit
```

The model includes the neutral fragment `Unit` addressing the metal cylinders and paper cuboid not recognized as coins or banknotes.

The model is evidently incomplete because of the missing dependencies. However, by joining it with the item morph model we obtain a complete model, of which DNF is shown in the following listing.

```
(Coin with Metal with Cylinder) or
(Banknote with Paper with Cuboid) or
Unit
```

Fig. 55. The joined morph model

Two morph models are joined through the so-called morph references: `&[TargetModel]`.

```
val curRef: &[CurrencyModel] = item
```

The curRef variable is a reference to a new assembler initialized with the joined morph model, which is created at compile-time. The new assembler also contains the so-called *default morphing strategy*, which adapts the strategy used to create the item to the joined model. This adapted default strategy is automatically chained with any other strategy used to create morphs from the new joined model.

By dereferencing the reference we obtain the new assembler:

```
val curAsm = *(curRef)
```

Then we may use the assembler as usual, i.e. to create a currency morph:

```
val currency =
curAsm.morph(CurrencyValidatingStrategy)
```

The `CurrencyValidatingStrategy` custom strategy validates if the proportions of metal cylinders and paper cuboids correspond to coins or banknotes. If not, the strategy selects the `Unit` alternative indicating an invalid currency. This strategy is chained with the above-mentioned default strategy adapted to the new model. In case the item is neither a metal cylinder nor a paper cuboid the default strategy selects the `Unit` alternative and eliminates the others. The `CurrencyValidatingStrategy` cannot override this decision, since it is a follow-up strategy in the chain.

The `currency` morph may be tested for its actual type by the `select` macro as usual:

```
select[Coin](currency) match {
   case Some(coin) =>
     // use the coin
   case None =>
}
```

V. Conclusion

One of the goals of this paper was to show that modeling protean objects might be surprisingly difficult in established OO languages such as Java, Scala and Groovy as three representatives.

Java as a non-trait language has proven to be the least suitable language for the multidimensional modeling. Because of the lack of traits, one must resort to composition, which forces us to model “is-a” relationship as “has-a” one. The complications stemming from this approach are known as object schizophrenia.

In Scala, as a representative of the static languages, each form of a given object must be declared as a class. Since the number of forms grows exponentially with the number of dimensions, the number of class declarations and lines of the code quickly becomes unsustainable.

Groovy, which is a dynamic language, can cope with this problem by means of the dynamic traits applied to objects at runtime, however in contrast to Scala, its weak type system is not able to guarantee the consistency of trait compositions made in a step-by-step way.

It seems that neither static nor dynamic OO languages are capable of coping with the multidimensional modeling. As suggested in this paper, the problem could be solved by a hybrid approach combining some features from the dynamic and static languages and adding new ones.

As a candidate to such a hybrid approach is presented object metamorphism, which describes all possible forms of an object by the so-called morph model, which is verified at compile-time. This approach guarantees that all possible forms of the object, called alternatives, are consistent. At run-time, these alternatives are used as templates for assembling instances from fragment stubs.

The application of the reference implementation of OM to the example showed that OM is able to resolve all major issues detected in the analysis.

Additionally, OM comes up with other concepts such as morph model extensions, which allows joining incomplete morph models with complete ones. This concept may be used for domain mapping or for raising the level of abstraction.

REFERENCES

[1] W. Harrison, H. Ossher, "Subject-oriented programming (a critique of pure objects)." In Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93), Washington, D.C., ACM, September 1993.

[2] W. Harrison, F. Budinsky, I. Simmonds, "Subject-oriented programming: Supporting decentralized development of objects", IBM TJ Watson Research Center, 1995.

[3] J. O. Coplien, T. M. H. Reenskaug, "The data, context and interaction paradigm". In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity (SPLASH '12) , ACM, New York, NY, USA, 227-228, 2012

[4] O. Alam, J. Kienzle, "Designing with inheritance and composition", In Proceedings of the 3rd international workshop on Variability & Composition (VariComp '12), ACM, New York, NY, USA, 19-24. 2012.

[5] S. Herrmann, "Demystifying object schizophrenia". In Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI '10), ACM, New York, NY, USA, 2010.

[6] D. Malayeri and J. Aldrich, "CZ: multiple inheritance without diamonds", In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09), ACM, New York, NY, USA, 2009.

[7] M. Odersky, L. Spoon, B. Venners, "Programming in Scala", Artima, 0981531644, 2008

[8] Z. Šlajchrt, "Morpheus: A Scala extension introducing metamorphism of objects", https://github.com/zslajchrt/morpheus, 2015

[9] K. Wright, "A Taste of Scala: The Autoproxy Plugin", http://www.artima.com/weblogs/viewpost.jsp?thread=275135, 2009.