Acta Informatica Pragensia, 2016, 5(1): 34–45
DOI: 10.18267/j.aip.83
Peer-reviewed paper

A Solution to the Square-Rectangle Problem Within the Framework of Object Morphology

Zbyněk Šlajchrt*

Abstract

The square-rectangle problem is often cited as an illustration of pitfalls arising when using object-oriented programming (OOP). A number of solutions have been proposed, however, according to the author, none of them solve the problem satisfactorily, mainly because they tackle the problem from within the current OOP paradigm. This paper presents another solution stemming from object morphology (OM), a new object-oriented paradigm developed to model mutable phenomena. In the framework of OM the problem can be solved directly under the basic OM principle that an object may mutate not only with regard to its state, but also with regard to its type. The main contrast between the presented and the other solutions is that constraint violations caused by changes in an object's state are no longer necessarily considered errors; instead, they may be interpreted as triggers initiating a mutation of the object's type. The solution is demonstrated using Morpheus, a proof-of-concept implementation of OM in Scala.

Keywords: Square-rectangle problem, Circle-ellipse problem, Liskov substitution principle, Object-oriented programming, Object morphology, Scala, Mutable objects.

1 Introduction

The square-rectangle problem (SRP) exposes a couple of flaws inherent to object-oriented programming (OOP) (Cline and Lomow, 1995). In particular, the problem is closely related to subtyping and inheritance and manifests itself as a violation of the Liskov substitution principle (LSU) (Majorinc, 1998, p. 629).

To solve SRP, one attempts to determine the proper subtyping, resp. inheritance relationship between the rectangle and the square. In mathematics, a square is a degenerated version of a rectangle having the same width and height. It follows that a square **is a** rectangle and thus the "IS-A" relationship should be used to model the relationship between squares and rectangles; in other words the square inherits from the rectangle (or the square is a subtype of the rectangle).

If the rectangle class exposes methods for setting the width and height of a rectangle then also the square class automatically inherits those methods (Rumbaugh, 1991). Given that according to the Liskov substitution principle squares are substitutable for rectangles, any square may be used in any routine processing rectangles, including the routines modifying the state of the processed rectangles using the public width and height setter methods (Liskov,

* Department of Information Technologies, Faculty of Informatics and Statistics, University of Economics, Prague, W. Churchill Sq. 4, 130 67 Prague 3, Czech Republic
✉ zslajchrt@gmail.com

1988). However, in cases when the setters are invoked on a square the resulting state of the square may be in violation of the constraint stipulating that the width and the height of a square must be the same.

This problem exemplifies a more general modeling problem, in which a class hierarchy consists of a base class and of subclasses that are constrained versions of the base class. The point here is that some methods of the base class can modify the object's state so that the modified state violates the constraints of a subclass. Such a modeling situation may occur quite often and may lead to serious design-related issues in the final application unless the problem is identified and solved early in the modeling phase of the development (Martin, 2000, p. 11).

Although the problem looks quite simple at first, its solution is harder than it appears (Majorinc, 1998, p. 627). There are a number of possible solutions of this problem and some of them are described in the literature review section. Broadly speaking, none of those solutions solve the problem comprehensively and often introduce secondary problems.

The purpose of this paper is to present a novel approach to the above-mentioned problem based on the conceptual framework called *object morphology* (OM), which is being developed in the author's doctoral thesis (Šlajchrt, 2015a). In OM, an object is, in principle, a mutable entity, whereas the mutation does not affect the object's state only, but also its type. There are no class or type hierarchies in OM; instead, objects are modeled by the so-called *morph models*, which describe the permitted structural mutations of the objects. More precisely, in OM a class is a special kind of a model; a model with only one alternative (i.e. one shape or type).

The presented approach entirely shifts the perspective on the constraint violations. In this perspective, a violation of constraints caused by the mutated object's new state is no longer necessarily considered an error; instead, the violation may simply be interpreted as a signal to mutate the object to a different alternative (type), with which the mutated state does not conflict. To put it in the context of SRP, a violation of the square constraints caused by setting the width and height properties to different values would lead to mutating the square instance into a rectangle, while preserving the identity of the object.

The solution to SRP is demonstrated using *Morpheus*, a proof-of-concept implementation of OM in the Scala language (Šlajchrt, 2015b). This paper focuses on the conceptual side of the problem, leaving aside other important aspects of the solution, such as performance, mainly on account of the fact that Morpheus is still in the proof-of-concept stage.

2 Related works

A number of attempts to solve SRP have been made so far. Some of them are quite obvious and should be called workarounds or "hot fixes" rather than solutions. One of those solutions suggests raising an exception from the problematic method in the affected subclass. A similar solution alters the previous one so that instead of throwing an exception the modifier would return the modified state. Provided that the inherited modifier cannot change the state due to a possible violation of the constrains it would return the unchanged state. A client of this modifier would acknowledge the contract that the returned data represents the actual effect of the modifier invocation. Another solution proposes imposing preconditions on modifiers; this approach is usable in the languages with support for method preconditions, such as Eiffel.

While the above-mentioned solutions remain at the method level, other solutions approach the problem from a structural point of view. One of such solutions is based on a popular rule,

according to which a concrete class must be derived from an abstract class and never from a concrete one (Meyers, 1996; Grosberg, 1997, pp. 36-42). Using this rule there would an abstract ancestor of both the square and the rectangle classes containing the "harmless" methods only, such as read-only methods. However, this approach may result in complicated hierarchies and may also break encapsulation of data declared in the abstract ancestor as explained in (Majorinc, 1998, p. 630). Other solutions address mutability of classes either by avoiding mutability completely or by factoring out modifiers to other classes. A more profound approach can be found in (Majorinc, 1998, p. 630). The authors come with an interesting concept of inverse inheritance, according to which certain methods defined in a subclass, which represents a subset of the set represented by its superclass, would be automatically and implicitly (inversely) inherited by its superclass. And conversely, only certain methods defined in the superclass would be automatically inherited by the subclass. A typical example of an inversely inherited method is a method assigning a new state to the inherited properties. Since such a method is defined in the subclass it may be assumed that the method does not corrupt the constraints of the subclass. It follows, however, that this same method would work in the superclass and thus it may be inversely inherited by the superclass. Boult et al. (1994) present the concept of dynamic attributes. It separates inheritance of object state from inheritance of storage, which leads to more flexible object derivations. Each object property is associated with one dynamic attribute, which comes in several forms differing in how they store or retrieve the value etc. Importantly, the form of an attribute in a subclass may differ from the form of the same attribute in the superclass. The author demonstrates the usefulness of dynamic attributes on the eclipse-circle problem, which is just another version of SRP.

The common ground of the above-mentioned solutions is that they consider violations of constrains as an error. In contrast to them the solution presented in this paper is based on the assumption that a constraint violation may be a signal to mutate the object so that the new state no longer violates any constraint.

Interestingly, most of the work related to SRP was published in the 90's and since then there has been no paper published with some novel approach to this problem. The reasons might be that SRP the presented workarounds are sufficient in most cases and that a potential rectification on the language level would require a rather fundamental revision of the philosophy of OOP languages. The primary goal of this paper is not to provide a practically usable solution of SRP; instead, it aims at the philosophical and conceptual aspects of SRP and presents a novel object-oriented paradigm, in which SRP may be naturally solved.

3 Solution

The solution of SRP described in the paper is developed in the framework of object morphology, which is a novel approach to object-oriented modeling. Therefore, in order to make the explanation of the solution method clearer, a brief introduction to OM is given.

3.1 Object morphology

Object morphology is a general approach to modeling primarily the so-called *protean objects*. A *protean object* is a term referring to a phenomenon occurring in a multitude of forms and defying the traditional Aristotelian class-based categorization (Madsen, 2006, p. 8). The concepts (abstractions) of such objects may often be only loosely defined, e.g. by means of family resemblance rather than by specifying strict rules for class membership.

Examples are fetal development, insect metamorphosis, phase transitions, autopoietic (self-maintaining and self-reproducing) systems such as cells, roles in society, crisis and other biological, social or economic phenomena. Instead of building type or class hierarchies, protean objects are modeled through the construction of morph models describing the forms that the protean objects may assume. The individual forms are called *morph alternatives*.

In its essence, a morph model is an abstraction (or concept) of related protean objects. The individual alternatives in the model are in fact the abstractions of the prototypical or exemplary instances among the abstracted protean objects (i.e. the concept's extension). Each alternative consists of the so-called fragments, which represent properties or features of the protean objects (i.e. the concept's intension).

The model describing the square/rectangle objects contains two alternatives: one representing pure (non-square) rectangles and another for squares (Fig. 1). This model expresses the type/state correspondence: a pure rectangle (Alt1), whose width and height are set to the same values, becomes a square, or more accurately, a rectangle with the square feature (Alt2). The square feature is preserved until the two dimensions violate the square constraint.

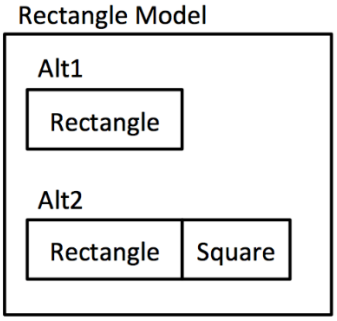

Fig. 1. *The schema of the rectangle/square morph model. Source: Author.*

A morph model is an analogy to a class in the traditional (Aristotelian) OO programming. On a statically typed OO platform, the compiler may build the morph model by parsing the model's type expression at compile-time. The compiler may analyze the morph type expression, build the model instance and perform various checks to guarantee that all alternatives in the model are complete and consistent.

A morph alternative describes one of the forms of a protean object and consists of one or more *fragments*. A fragment is a building block representing a typological, behavioral and structural element of protean objects. It represents a property or feature of a protean object and semantically corresponds to the concept of trait[†] as defined in Scala or Groovy (Scala, 2016; Groovy, 2016). Fig. 2 depicts the `Rectangle` and `Square` fragments (modeled using traits) with their attributes and methods.

[†] A trait is a concept possessing some features of an abstract class and some of an interface. Just as a class, it may define behavior and also contain a state (in Scala). As regards the similarity to an interface, a trait may not be instantiated on its own and a single class may extend multiple traits (a kind of multiple inheritance). This feature enables horizontal composition of behavior, in contrast to the vertical nature of inheritance.

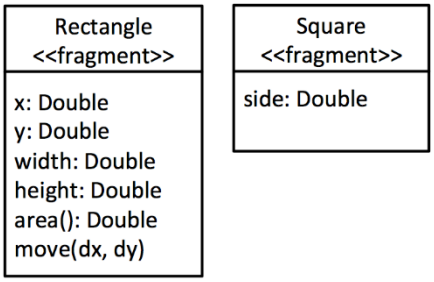

Fig. 2. *Rectangle and Square fragments. Source: Author.*

Instances of alternatives are called *morphs*, which are created by a *recognizer* according to the alternative selected by the recognizer's *morphing strategy*. On every morph instantiation the morph strategy evaluates all possible alternatives and selects the one that matches best the current state of the object or outer conditions such as input parameters. In the case of the square/rectangle model, the morphing strategy would be selecting the second alternative if the width and height are the same, otherwise it would select the first alternative.

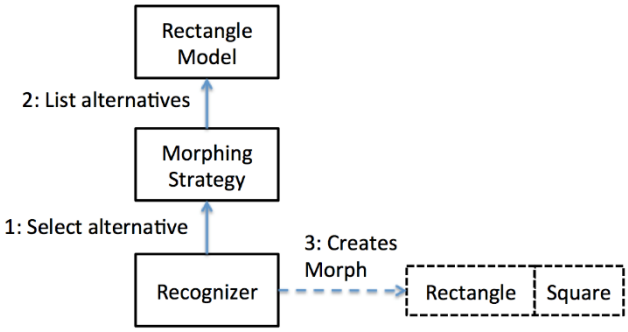

Fig. 3. *A basic collaboration schema depicting the creation of the square/rectangle morph. Source: Author.*

It follows from the above that the recognizer can never instantiate an invalid composition of fragments. The only risk is that the strategy may be improperly configured or implemented, which may result in selecting inappropriate or invalid alternatives. Fig. 3 illustrates the collaboration between the individual components when instantiating a square/rectangle morph. First, the recognizer asks the morphing strategy to select one of the alternatives from the model, which will be used as a template for the initial form of the morph. When selecting the alternative the strategy takes into account the input (constructor) parameters specifying the initial width and height of the morph. If no parameters are specified, the strategy uses the default values of the attributes. The recognizer then uses the selected alternative to create the morph.

Note: Although type mutation is somehow doable in languages featuring dynamic traits, such as Groovy, it can be shown (Šlajchrt, 2015a) that a solution based on dynamic traits does not guarantee the consistency of manually assembled trait compositions and tends to an unmaintainable code.

3.2 Using Morpheus to Solve SRP

The solution to CRP in the framework of OM will be demonstrated using Morpheus, the proof-of-concept OM implementation in Scala. Morpheus is in fact an extension to the Scala language whose purpose is to parse and validate morph models at compile time and to provide

a runtime environment for object morphing. Morph models are constructed by means of special types and a set of macros, through which the morph model types are processed at compile time. The morph model types resemble Boolean algebra formulas specifying the valid forms of an object.

The following paragraphs illustrate the use of Morpheus to model an object capable of changing dynamically its type that can be either a rectangle or a square. First of all, the let us begin with the building block definitions; i.e. the shape types. All types are defined using the Scala traits.

The `Shape` trait represents the abstract base for all other concrete shapes; i.e. the rectangle and the square.

```
trait Shape {
  def shapeName: String
  def area: Double
  def printShape(): Unit
}
```

The `shapeName` method returns the name of a particular `Shape` instance. Method `area` calculates the area occupied by the shape and the `printShape` method outputs the textual representation of the shape.

Rectangular shapes are represented by the `Rectangle` trait. It extends the `Shape` trait and implements its all methods.

```
trait Rectangle extends Shape {
  var x: Double = 0
  var y: Double = 0
  var width: Double = 0
  var height: Double = 0

  def area: Double = width * height

  def printShape(): Unit = {
    printf("%s(%d,%d,%d,%d)\n", shapeName(), x, y, width, height)
  }

  def shapeName: String = "Rectangle"
}
```

In addition to the inherited methods, the `Rectangle` trait defines four mutable attributes `x`, `y`, `width` and `height` holding the position and the dimensions of a rectangle (the **var** keyword declares a mutable attribute). The `printShape` method creates a textual representation of a rectangle consisting of the rectangle's name and the values of its position and dimension attributes.

The `Square` trait is defined as a special form of a rectangle by "extending" the `Rectangle` trait. The quotes surrounding the word extending are to suggest that what is modeled here is not an extension, but a specialization; in contrast to an extension, which increases the degrees of freedom of the new type generally by adding new members independent of the parent type, a specialization stipulates special conditions (i.e. constraints) on the inherited members and thus decreases the degrees of freedom of the new (specialized) type. In specializations the new members depend on the parent type and their logic is constructed with respect to the constraints. In mixed cases a type establishes some constraints and introduces new independent members at the same time. It should be diagnosed as a weakness of Scala and

other object oriented languages that they use the same construct for the two distinct operations.

```
trait Square extends Rectangle {
  def side = width
  def side_=(s: Double): Unit = {
    assert(width == height)
    this.width = s
    this.height = s
  }
  override def shapeName: String = "Square"
}
```

The couple of methods `side` and `side_=` defines a modifier, which allows retrieving and setting the square's side. This definition follows a Scala code comprehension rule for attribute accessors. A client of a Square instance may set and get the value of the attribute covered by such a couple of methods in the same way the client accesses normal attributes (Martin, 2009). The `side` setter first performs an assertion verifying the constraint, i.e. the condition that the width and the height are the same. Then the setter assigns the input value to both dimension attributes `width` and `height`. The `shapeName` method overrides the method from `Rectangle`; this fact is emphasized by marking the method with the `override` keyword.

Having defined the shape types, it is possible to create some instances. First, in order to demonstrate the difficulties described by SRP, a square instance is created in the traditional way.

```
val square = new Rectangle with Square {}
square.printShape()
```

The preceding statement creates and prints an instance of type `Rectangle` extended by the `Square` trait. As mentioned above, traits cannot be instantiated on their own; thus an anonymous class must be created by appending the curly brackets to the end of the `new` statement. Since the traits leave no method unimplemented, the body in the brackets is empty.

The printed textual representations reveals that the square's state is consistent with the constraint `width=height`.

```
Square(0.0,0.0,0.0,0.0)
```

Since the `Square` inherits from the `Rectangle`, it is possible to access the inherited attributes, such as `width`.

```
square.width = 200
square.printShape()
```

Setting this attribute to 200 makes the square state inconsistent with the constraints, as indicated by printing the square.

```
Square(0.0,0.0,200.0,0.0)
```

Also assigning a new square side value through the square's `side` method will fail due to the assertion rule in the `side` setter.

```
rect.side = 100

java.lang.AssertionError: assertion failed
```

One would have wished that the `Square` trait were removed automatically from the instance when the constraint is violated. The same object would have ceased to be a square and would have become a generic rectangle instead.

The subsequent passages present a solution using Morpheus. This solution is also based on the dynamic addition and removal of the `Square` trait, however, in contrast to the dynamic traits approach, these operations are executed in a more controlled and safer way utilizing declaratively specified morph models.

The first step is the construction of the morph model describing all possible forms of the modeled object. In this case there are only two alternatives:

```
Rectangle

Rectangle with Square
```

In Morpheus, such a model can be specified and parsed using the `parse` macro invoked with the morph model type expression `Rectangle with \?[Square]`, in which the `\?[]` operator marks the `Square` type as optional.

```
val rectModel = parse[Rectangle with \?[Square]]
```

Note: The `\?[]` operator is not a feature of Scala but an extension introduced by Morpheus. It is in fact a "syntax sugar" for the equivalent union-like type expression `Unit | Square`. Here, the type operator `|` is another extension originating in Morpheus that serves to express type unions.

It should be remarked that although the parse macro invocation looks as a normal method invocation, it is actually executed at **compile time**. It produces a special abstract syntax tree structure representing the morph model, which is substituted for the macro invocation tree by the compiler. Nevertheless, the `rectModel` variable is assigned with a reference to the parsed and validated morph model.

Prior to using the model to instantiate a morph, a morphing strategy must be defined. A morphing strategy determines which morph alternatives may be realized with respect to outer (parameters, context) or inner conditions (the object's state). In this case, the strategy takes into account solely the inner conditions; in particular the width and the height attributes. Although a morphing strategy may be created from the scratch by implementing the `MorphingStrategy` interface, there are a couple of macros making this task easier. One of such macros is `promote`, which uses a sub-model of the main model to determine the right alternative.

```
val rectStg = promote[Square](rectModel, {
   case None => Some(0)
   case Some(rect) if rect.width == rect.height => Some(0)
   case _ => None
})
```

The sub-model is specified as the type argument of the macro and the main model is passed as the first argument. The second argument is a pattern matching closure[1] implementing the logic of the strategy. The *implicit* argument to the closure function is an optional morph (`Some(rect)`) or nothing (`None`). `None` is passed if the closure is invoked for the first time when the morph does not exist yet. The closure's task is to select a fragment type from the

[1] A pattern match closure reminds the traditional `switch` statement known from C and Java. However, it provides a more advanced apparatus for specifying the individual cases (the so-called patterns) and the handling code. A pattern match closure function does not have the traditional parameter declaration, as the parameter is expected to assume more specific types, which can be inferred from the case statements.

sub-model, which will be *promoted*; if unable to make a selection it returns `None`. A promotion of a fragment type is a special reordering of the main model, which leads to the selection of an alternative containing the promoted fragment type.

In this case the sub-model consists of one fragment type only - `Square`. There are three outcomes in the closure. The first outcome takes place when the morph is not created yet and thus defines the default form of the object. In this case the closure returns `Some(0)`, where 0 is the index of the promoted fragment. This return value instructs the strategy to promote an alternative containing the `Square`. The second outcome occurs when the morph's width and height are the same. This condition is in fact the constraint for a square and therefore the returned value is also `Some(0)` resulting in the selection of the square form alternative. The remaining case happens if the dimensions of the existing morph are different. In such a case the closure returns `None`, as there is no fragment type in the sub-model for this situation. This outcome will actually indirectly lead the strategy to pick the default alternative, which is the first one in the model. Here, it is the alternative with `Rectangle` only.

When the morphing strategy is defined, the morph model may be used to create a recognizer, which may be seen as a morph factory instantiating morphs according to the given morph model. The morphing behavior of the resulting morphs is governed by the given morphing strategy. The recognizer for rectangle/square morphs is created by means of the `singleton` macro as follows:

```
val rectRkg = singleton(rectModel, rectStg)
```

The singleton macro creates a recognizer using singleton factories that ensure that the individual fragments are not re-instantiated and the same instances are used instead when re-morphing the morph.

A new square/rectangle morph is created by invoking the `make_~` method on the recognizer.

```
val rect = rectRkg.make_~
rect.printShape()
```

Printing the morph will yield the following output, as expected.

```
Square(0.0,0.0,0.0,0.0)
```

Now, let us modify the `width` attribute of the morph that will now be in violation of the square constraint, which is encoded in the morphing strategy.

```
rect.width = 200
rect.remorph
rect.printShape()
```

However, this violation can be immediately resolved by re-morphing the morph explicitly invoking the `remorph` method on the morph. This method is engages the morphing strategy specified when creating the recognizer to choose the proper alternative reflecting the updated state of the morph. Behind the scenes, the morph is being re-assembled according to the new alternative, which contain the `Rectangle` fragment type only. This fact is confirmed by printing the morph; the `printShape` method internally invokes the `shapeName` method belonging to the Rectangle trait now resulting in the following output.

```
Rectangle(0.0,0.0,200.0,0.0)
```

The morph can be reshaped to a square by assigning 200 to the height attribute and re-morphing the morph.

```
rect.height = 200
rect.remorph
rect.printShape()
```

The console output should read now:

```
Square(0.0,0.0,200.0,200.0)
```

The morph can be safely cast to a particular alternative type using the `select` macro. This macro determines whether the morph in the argument is compatible with the type specified in the square brackets. If so the macro returns `Some(x)`, where x is a reference of the requested type; otherwise it returns `None`. With this on mind, it is possible to declare a variable of type `Square with Rectangle` and initialize it with the result of the `select` macro. It is quietly assumed that the `rect` morph is a square and thus the macro does not return `None`.

```
val sq: Square = select[Square](rect).get
```

Note: The use of the `select` macro is preferred to the simple typecasting since the select macro performs some compile-time analysis, which for example can detect that a type in the square brackets is incompatible with the model of the morph in the argument.

Now it is possible to access the square specific members, such as `side`.

```
sq.side = 80
rect.printShape
```

The effect of assigning 80 to the `side` attribute may be verified by printing the square.

```
Square(0.0,0.0,80.0,80.0)
```

Since `sq` is also an instance of `Rectangle`, it is possible to access the dimension attributes and violate the square constraint by assigning 50 to the `height` attribute.

```
sq.height = 50
rect.remorph
```

After re-morphed, the morph becomes a rectangle again. (Note: to re-morph the morph the original morph reference `rect` must be used instead of the typecast `sq` reference, which does not expose the morphing interface.) However, if the `printShape` method is invoked on the `sq` reference to check out the current shape the following exception is thrown:

```
Exception in thread "main" org.morpheus.StaleMorphException
```

The reason is that the square reference `sq` has become invalid after the morph mutated from a square to a rectangle. Any invocation on an invalid reference results in that exception signaling that the reference references a “stale” morph. To print the current shape the original reference `rect` must be used:

```
rect.printShape
```

The previous statement prints the correct output:

```
Rectangle(0.0,0.0,80.0,50.0)
```

4 Results

The presented solution has demonstrated how the square-rectangle problem can be tackled within the framework of OM, in particular by using Morpheus, a P-o-C of OM in Scala. It was shown how to model and use mutable objects, such as square-rectangle morphs. Also was shown how to construct a simple morph model using the special type expressions and how to parse the model and instantiate a morph from it using Morpheus macros.

Further, the solution sketches a typical workflow and the possible issues that may arise when working with morphs. It is explained how to re-morph an existing morph into another type and how to safely typecast the morph to a specific alternative type. Special attention was also paid to the issue of stale morph references.

There is a small issue in the presented solution concerning the explicit re-morphing done by invoking the `remorph` method after any change violating the square constraint. Between the change and the re-morphing, the morph exists in an inconsistent state for a moment. If this is an issue, it could be mitigated by making the re-morphing implicit; in other word to invoke the `remorph` method from within the morph, for example from the setter methods. Any change in the type-related state would be then subject to re-morphing. However, the implicit re-morphing might not be suitable in situations when the client wishes to perform bulk changes in various attributes first and then to re-morph the morph according to those changes. In this case the intermediate re-morphing after each change in an attribute would certainly be an undesirable behavior. It is therefore up to the designer to decide which approach to follow.

The code contains some simplifications done for the sake of code brevity. The full code can be viewed on or downloaded from (Šlajchrt, 2016).

5 Discussion

Object metamorphism sheds new light on the square-rectangle problem. In the framework of OM the problem virtually vanishes under the basic assumption that an object may mutate not only with regard to its state, but also with regard to its type; in other words, the state and the type of the object may be two interconnected aspects. With this in mind, the square and rectangle are just two types of the same object. Which type is active depends on the values of the object properties. From the standpoint of OM the square-rectangle problem is perceived as a symptom of an inherent OOP insufficiency to address well the link between an object's type and state; i.e. that these two aspects may influence, depend on and be the cause of each other.

The main difference between the OM solution and the other ones is that constraint violations are no longer necessarily considered errors; instead, they may be handled as signals to re-morphing. The mutation behavior of an object is specified by a morph model, which is constructed by means of special type expressions and validated at compile-time. In OM the morph models actually replace classes. The declarative nature of OM and the compile-time morph model validation also make an important difference, compared to the approaches using dynamic traits or mixins to change the type of objects, in which the developer must take care of the tasks performed by the compiler in OM. OM, in fact, introduces a hybrid dynamic/static approach to modeling and implementing mutable phenomena, combining declarative modeling and static code analysis (validation) with controlled dynamism. As a result of this, a mutable object can never assume an invalid form at runtime. The main downside of this design is poor performance both at compile-time and run-time. This problem may be attributed to the fact that Morpheus is still only proof-of-concept software, which has yet to be subjected to any optimization efforts.

One of the future directions in developing OM could be an implementation of Morpheus on top of a prototype-based dynamic language, such as JavaScript. Notwithstanding the fact that these languages usually lack a compilation phase, during which morph models could be parsed and validated, the morph models might be authored as special artifacts that are parsed and validated at the time of developing the application. The resulting compiled artifacts would be loaded by applications in a way similar to how classes are loaded into the runtime environment. The loaded morph models would then be used to create morphs.

References

Boult, T. E., Fenster, S. D., & Kim, J. W. (1994). Dynamic Attributes, Code Generation and the IUE. In *Proceedings of the ARPA Image Understanding Workshop* (pp. 405-422). Monterey.

Cline, M., & Lomow, G. (1995). *C++ FAQs: Frequently Asked Questions*. Reading: AddisonWesley.

Groovy Documentation. (2016). *Traits*. Retrived from: http://docs.groovy-lang.org/latest/html/documentation/core-traits.html.

Grosberg, J. (1997). Design guidelines for Is-A hierarchies. *Dr. Dobb's Journal*. Retrived from: http://www.drdobbs.com/database/design-guidelines-for-is-a-hierarchies/184410208

Liskov, B. (1988). Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5), 17-34. doi: 10.1145/62138.62141

Majorinc, K. (1998). Ellipse-Circle Dilemma and Inverse Inheritance. In *Proceedings of the 20th International Conference of Information Technology Interfaces*. Pula. Retrieved from: http://kazimirmajorinc.com/Documents/1998,-Majorinc,-Ellipse-circle-dilemma-and-inverse-inheritance.pdf.

Martin, D. (2009). Gettersand Setters in Scala. Retrieved from: http://dustinmartin.net/getters-and-setters-in-scala/.

Martin, R. C. (2002). *Agile Software Development, Principles, Patterns, and Practices*. London: Pearson.

Meyers, S. (1994). *More Effective C++*. Reading: Addison-Wesley.

Madsen, O. L. (2006). Open Issues in Object-Oriented Programming – A Scandinavian Perspective. *Software: Practice and Experience,* 25(S4), 3-43. doi: 10.1002/spe.4380251303

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-Oriented Modeling and Design*. New Jersey: Prentice-Hall.

Scala Documentation. (2016). *A Tour of Scala: Traits*. Retrieved from: http://www.scala-lang.org/old/node/126.

Šlajchrt, Z. (2015a). Object Morphology. *A draft of the dissertation thesis*. Retrieved from: https://github.com/zslajchrt/morpheus/raw/master/src/main/doc/thesis/thesis.pdf.

Šlajchrt, Z. (2015b). *Morpheus, a Proof-of-Concept Implementation of Object Morphology in Scala*. Retrieved from: https://github.com/zslajchrt/morpheus.

Šlajchrt, Z. (2016). *The Square-Rectangle Solution in Morpheus*. Retrieved from: https://github.com/zslajchrt/morpheus-tutor/blob/master/src/main/scala/org/cloudio/morpheus/square/Square.scala.