University of Economics, Prague

Faculty of Informatics and Statistics

Department of Information Technologies

Programme: Applied Informatics

Field of Study: Informatics

**Object Morphology—A Protean Generalization of Object-Oriented Paradigm**

Dissertation Thesis Abstract

Student: Mgr. Zbyněk Šlajchrt

Supervisor: Ing. Rudolf Pecinovský, CSc.

Opponents: Doc. Ing. Vojtěch Merunka, Ph.D.

Doc. Ing. Miroslav Virius, CSc.

**Prague, June 2016**

This dissertation thesis was elaborated within the doctoral study in the field of Informatics at the Faculty of Informatics and Statistics, University of Economics, Prague (VŠE).

The defence of the dissertation thesis will take place on 15th September 2016 in front of the board of examiners in the field of Informatics at VŠE, W. Churchill Square 4, 130 67.

The departmental defence took place during the monthly meeting of the Department of Information Technologies on 16th May 2016 at VŠE. The author’s publications related to the thesis are mentioned in the References section.

# Abstract

Modeling protean objects, i.e. objects adapting their structure and behavior dynamically with respect to a changeable environment, is often challenging in traditional object-oriented languages. According to the author, the root cause of this problem lies in the class-based conceptual framework embedded in the foundation of the object-oriented paradigm. The proposed paradigm Object Morphology (OM) is greatly influenced by prototype theory developed in the field of cognitive psychology. OM abandons the notion of *class* and suggests, instead, that the abstractions of protean objects should be established through the construction of *morph models* describing the possible forms of those objects. This thesis defines the theoretical foundations of OM, which is further used to specify the elements of prototypical object-oriented analysis. An important part of this work is also a proof-of-concept implementation of an OM framework in Scala.

**Keywords:** Object-orientation, modeling, conceptual framework, abstraction mechanism, prototype theory, metamorphism, Aristotelian logic, Scala.

# Abstrakt

Modelování objektů, které mohou měnit svoji strukturu a chování dynamicky s ohledem na změny v prostředí, je v tradičních objektově orientovaných jazycích velmi obtížné. Podle autora tkví problém v konceptuálním rámci, který je základem objektově orientovaného paradigmatu. Navrhované paradigma nazvané *morfologie objektů* (MO) je výrazně ovlivněno teorií prototypů z oblasti kognitivní psychologie. MO opouští pojem *třída* a navrhuje, aby se abstrakce proměnlivých objektů vytvářely sestavováním tzv. *modelů proměnlivosti*, které popisují možné formy modelovaných objektů. Práce definuje teoretické základy paradigmatu, které jsou použity pro následnou specifikaci metodiky analýzy a návrhu aplikací. Součástí práce je také referenční implementace rámce vybudovaného nad zmíněným paradigmatem v jazyce Scala.

**Klíčová slova:** Objektově orientované programování, modelování, konceptuální rámec, abstrakční mechanismus, teorie prototypů, metamorfismus, aristotelovská logika, Scala

Contents

1 Contents of the Thesis 6

2 Introduction 7

3 Goals 7

4 Related Works 8

5 Methods 8

6 Achievements 9

6.1 Case Studies 9

6.2 Theoretical Foundations of OM 9

6.2.1 Morph Models 9

Examples 10

6.2.2 Morphing Strategies 14

6.3 Prototypical Analysis 14

6.4 Reference Implementation (RI) 15

6.5 Applications 15

6.5.1 Onto UML 15

6.5.2 DCI 16

6.5.3 Square-Rectangle Problem 16

6.5.4 Final Case Study: Modeling Human Emotional Facial Expressions 16

7 Evaluation 18

8 Further Work 19

9 References 19

10 Terminology 20

Contents of the Thesis

1 Introduction 7

2 Research And Related Works 11

2.1 Philosophical Aspects 11

2.2 Prototype Theory 13

2.3 Prototype Theory And Object-Oriented Programming 15

2.4 OntoUML 15

2.5 Alternative Approaches In Object-Oriented Programming 19

2.6 The Square-Rectangle Problem 23

3 Introductory Case Studies 24

3.1 1st Case Study: Structural Aspects Of Multidimensionality 24

3.2 2nd Case Study: Behavioral Aspects Of Multidimensionality 52

4 Case Studies Analysis 79

4.1 Terminology And Identified Concepts 80

4.2 Code Samples 83

5 Theoretical Foundations of OM 85

5.1 Recognizer Formalism (R-Algebra) 86

5.2 Morphing Strategies 128

5.3 UML Notation 133

5.4 Orthogonality of Fragments 137

5.5 Translating Type Hierarchies 138

6 Prototypical Analysis 144

6.1 Understanding Phenomena 144

6.2 Identifying Prototypes 145

6.3 Property Analysis 145

6.4 Morph Model Construction 146

6.5 Binding Properties To Context 147

6.6 Morphing Strategy Construction 148

6.7 Creating Recognizer 148

7 Reference Implementation (RI) 149

7.1 Design Overview 149

7.2 Compile-Time Component 150

7.3 Run-Time Component 158

8 Final Case Study 186

8.1 Analysis and Design 187

8.2 Summary 209

9 Conclusion 211

10 Appendix 1: Using OM to implement an OntoUML model 213

Introduction

Object-orientation (OO) in software development has proven successful in a wide area of applications and has played a major part in software development over the past few decades [1][2][3]. Nevertheless, as shown as presented in [4], OO modeling is not able to capture well the tacit knowledge in the form of experience, heuristics and human competencies, which constitutes the major part of design knowledge today. Furthermore, according to [5], no object-oriented notation is rich enough to be able to model all key aspects of the real world.

Pivotal in OO modeling is the Aristotelian conceptual framework, in which an object represents a physical *phenomenon*, while a class may be used to represent a *concept* from the real world. Such a notion is expected to make it easier to model entities from real domains and the relationships among them [5][6].

The thesis presents an alternative and “non-aristotelian” object-oriented paradigm called *object morphology* (OM), whose domain is primarily the modeling of the so-called *protean objects*. A *protean object* is a term referring to Proteus – a sea god famous of his metamorphic feats. Hence the term protean object denotes a phenomenon occurring in a multitude of forms and defying the traditional Aristotelian class-based categorization [5]. 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*.

# Goals

The main goal of this thesis is to introduce Object Morphology as a novel paradigm for modeling protean objects through the construction of morph models describing the forms that the protean object may assume. The outputs of this goal are the theoretical foundations of object morphology as well as a mathematical formalism used to specify morph models.

Equally important is the goal to provide a proof-of-concept implementation of object morphology – Morpheus. The purpose of this software is to provide a means for the verification of the theoretical concepts of OM on real-world use cases. Morpheus will be developed as open-source software available to download from a GitHub repository. Morpheus will also be accompanied by an online tutorial and other source codes illustrating various applications of Morpheus. Morpheus will be developed as an extension of the Scala programming language.

An additional goal is to provide rudimentary theoretical grounds for prototypical analysis of complex mutable phenomena avoiding the Aristotelian conceptual framework. The output of this goal is a proposal of rather general guidelines that are to be followed when modeling the above-mentioned phenomena.

Related Works

OM is based on prototype theory formulated by Eleonor Rosch in the 1970s [7]. This theory is used in the field of cognitive psychology to describe how people classify things. It claims that some members of a class are more central than others, and those “more central” members are called *prototypes*. In his discussion of open issues in object-oriented programming, Madsen proposes an adoption of the prototypical view of concepts into object-oriented methodologies and languages [5]. Additional information on the relationship between prototypical concepts and prototype‑based language can be found in [8][9][10].

As far as rather practical effort made in this field, one must mention an UML extension called OntoUML developed by Giancarlo Guizzardi [11]. Another representative of such effort is Data-Context-Interaction paradigm (DCI) proposed by Reenskaug and Coplien [12]. Sections 6.5.1 and 6.5.2 deal with those approaches in more detail and in the context of OM. Other related approaches include Aspect-Oriented Programming [13], Subject-Oriented Programming [14], Role-Oriented Programming and mixins. In [15] the author paper analyses the problem of class generalization from the non-traditional point of view – class life cycles. The paper also shows that this problem is closely connected with the “problem of conflicting identities” in generalization trees discussed on the border of the conceptual modeling and ontology engineering fields.

Methods

Chosen methods reflect the character of the problem, which lies on the border between basic and applied science:

1. Case studies (one structural, one behavioral)
2. Evaluation of three OOP languages (Java, Scala, Groovy)
3. Analysis of problematic aspects identified in the case studies
4. Generalization of the existing paradigms and methods (OOP, UML, Liskov principle)
5. Verification through development of P-o-C and its application on a set of problematic scenarios

Achievements

Case Studies

The two case studies analyzed in the thesis show that modeling multidimensional objects might be surprisingly difficult in Java, Scala and Groovy as three representatives of current popular languages. Java as a non-trait language has proven to be the least suitable language for multidimensional modeling. Because of its lack of traits, one must resort to compositions and delegations, which leads to obscuring both type and behavior during consecutive mappings between domains. In Scala 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 quickly becomes unsustainable. Groovy can cope with this problem by means of dynamic traits mixed in with objects at runtime, how-ever in contrast to Scala, Groovy’s weak type system is not able to guarantee the consistency of trait compositions made in a step-by-step way (i.e. imperatively).

In the course of the case study analysis a new concept of the class builder has been developed and examined as a potential new platform evaluated in each scenario. The class builder was conceived as an extension of Scala and its purpose is to generate a class for any possible trait composition, as given by the model defined by means of a special type expression. The compiler extension must be able to check the consistency of the model by decomposing the model type and examining the dependencies of all individual traits specified by the traits’ self-type.

When instantiating a new object the class builder selects the best trait composition, called an alternative, according to the recommendation of the class builder strategy. Since the strategy does not determine the trait compositions explicitly and only selects the best matching alternative, it is guaranteed that the builder always generates a class made up of a mutually consistent set of traits described by the selected alternative. The concept of the builder is elaborated in a greater detail later in the thesis and is given the name *recognizer*.

## Theoretical Foundations of OM

This section outlines the theoretical foundations of Object Morphology. With respect to its limited scope, this abstract introduces the concepts in a rather brief manner.

### Morph Models

When modeling a protean phenomenon in the outer world, the modeler focuses on the description of the features exhibited by individual instances of the phenomenon. The individual instances may or may not share a set of common features, and even a single instance may not exhibit the same set of features during its existence. A typical example of such a phenomenon is a butterfly and its development; there is little or nothing in common between individual developmental stages, except the butterfly’s identity.

In Object Morphology, the concept of a *fragment* is used to describe a feature of a modeled phenomenon. A fragment can be likened to a potentially complex attribute with possible behavior. In contrast to fragments, attributes hold data only and do not define any behavior, nor do they possess identity. On the other hand, a fragment should not be mistaken for a part of a phenomenon. The reason is that a fragment, in contrast to a part, may constitute or contribute to the identity of its phenomenon, while a part possesses its own identity distinct from its owner’s identity.

A collection of fragments describing a given phenomenon at certain moment is called *alternative*. The individual fragments in the alternative may be *passive* or *active*. A passive fragment’s behavior cannot be influenced by the presence of other fragments, while the behavior of an active fragment can be overridden by other fragments (it is an analogy to final/non-final classes). An instance of an alternative is called a *morph*, which is a representation of a modeled phenomena instance (an analogy to an object).

The collection of all alternatives describing a given phenomenon is called a *morph model*. A morph model is in fact an expression of the concept of the phenomenon. A morph model may describe a phenomenon whose instances are either *immutable* or *mutable*. The immutable instances exhibit the same features until they cease to exist. The individual immutable instances differ in the composition of the fixed feature set, while the diversity of this set among the instances is governed by the morph model. On the other hand, the mutable instances may undergo many mutations during their lifetime, while the only possible mutations are dictated by the morph model.

There may exist one or more disjoint groups of alternatives sharing a certain collection of fragments. These common fragments are called *prototypes* of the morph model and the group sharing a given prototype is called the prototype’s *attractor*. It should be remarked that there may exist more ways to partition a morph model to attractors. Then, it is on the modeler to select the best partitioning.

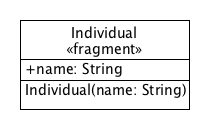
Examples

The rest of this section presents a couple of examples illustrating the-above mentioned concepts. The models in the examples are expressed using the R-Algebra, a mathematical formalism developed in the presented thesis [16], used to construct morph model expressions.

The *Person* model in (**1**) represents a person and consists of a single fragment – *Individual*. Such singular models consisting of only one alternative with one fragment virtually correspond to the traditional concept of class.

|  |  |
| --- | --- |
| Person = Individal | (**1**) |

The notation of fragments follows the notation of classes (**Figure** 1). To distinguish the two concepts the fragment notation applies the *fragment* stereotype. It should be reminded that despite the notation similarity, the two concepts are fundamentally different, as a fragment should be seen rather as an attribute on steroids.



**Figure** : Fragment *Person*

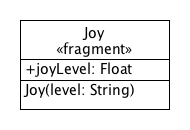
A morph creation can be schematically expressed as shown in the pseudo-code[[1]](#footnote-1) (**2**). The *p* morph can be used in the same way as if it were an instance of a class having the same structure as fragment *Individual*.

|  |  |
| --- | --- |
| p: Person = Individual(“Peter”) | (**2**) |

The model in formula (**3**) models the six basic human emotions. The vertical bar is the *union* operator delimiting mutually exclusive emotions (in terms of this model).

|  |  |
| --- | --- |
| Emotion = Joy | Surprise | Fear | Sadness | Disgust | Anger | (**3**) |

**Figure** 2 shows the structure of the *Joy* fragment. The structure of the remaining emotion fragments is analogous.



**Figure** : Fragment *Joy*

The listing (**4**) sketches two instantiations of emotion morphs.

|  |  |
| --- | --- |
| e1: Emotion = Joy(0.7) e2: Emotion = Surprise(0.2) | (**4**) |

Besides the union operator, the R-Algebra formalism introduces the *join* (.) operator, by which two sub-expressions are declared as co-occurring. To illustrate the join operator let us suppose that the modeler comes to conclusion that the original model is too coarse-grained since it does not include mixed emotions. A finer-grained model may be constructed by joining the original model with itself, as shown in (**6**).

|  |  |
| --- | --- |
| Emotion = (Joy | Surprise | Fear | Sadness | Disgust | Anger). (Joy | Surprise | Fear | Sadness | Disgust | Anger) =  (Joy | Surprise | Fear | Sadness | Disgust | Anger)*2* | (**5**) |

In case of the emotion fragments, both operators fulfill the associativity, commutativity and distributivity law with the join operator as the one being distributed. Besides those, for any two emotion fragments holds that *F.F* = *F* and *F* | *F* = *F*. Applying these rules results in the following *canonical* form of the model.

|  |  |
| --- | --- |
| Emotion = Joy | Surprise | Fear | Sadness | Disgust | Anger | Joy.Surprise | Joy.Fear | Joy.Sadness | Joy.Disgust | Joy.Anger | Surprise.Fear | Surprise.Sadness | Surprise.Disgust | Surprise.Anger | Fear.Sadness | Fear.Disgust | Fear.Anger | Sadness.Disgust | Sadness.Anger | Disgust.Anger | (**6**) |

A morph consisting of *Joy* and *Surprise* emotions may be instantiated as shown in the following pseudo-code:

|  |  |
| --- | --- |
| e1: Emotion = Joy(0.6).Surprise(0.2) | (**7**) |

So far, the example models have dealt with the fragments for which the above-mention R-Algebra rules hold. Such fragments are called *entities*. An important consequence of those rules is that no alternative in a model may consist of two occurrences of the same entity fragment. Besides entities there is another kind of fragments, which are governed by different rules. Such fragment are called *wrappers*, which owe their name to the fact that they can be stacked on top of entity fragments and override their behavior. As a wrapper example, let us consider the morph model in (**8**), which models a path.

|  |  |
| --- | --- |
| Path = Origin.(1 | Move | Left | Right) *N* | (**8**) |

The model includes the *Origin* entity fragment, which encapsulates the origin of the path. The other three fragments *Move*, *Left* and *Right* represent the three basic movement steps. A composition of those three steps results in the transformation of the origin into the end point of the path. For wrappers, the distributive and associative laws still hold, but the commutative law and the rule *F*.*F* = *F* are not guaranteed. A sample path morph evolution is in the listing (**9**).

|  |  |
| --- | --- |
| Origin Origin.Move Origin.Move.Left Origin.Move.Left.Move | (**9**) |

The *1* element in the model represents the so-called *unit fragment*, which plays the role of the unit element in the R-Algebra, for which the rules *1* | *F* = *F* a *1.F* = *F* hold.

The *N* parameter in the exponent determines the maximum number of steps of the paths modeled by the Path model. As long as the model should be able to describe paths consisting of unlimited number of steps, the model would have to be designed recursively, as shown in (**10**).

|  |  |
| --- | --- |
| Path = Origin.(1 | Move | Left | Right).Path | (**10**) |

The morph model of a compass in (**11**) illustrates a use of the so-called *inverse fragments*. The inverse fragment of fragment *F* represents the antipodal feature to the feature represented by *F*. For a fragment and its inverse fragment it holds that *F.~F*=*0*, where 0 is the *null* fragment playing the role of the zero element in R-Algebra.

|  |  |
| --- | --- |
| Compass = (N.~S | W.~E | S.~N | E.~W)*N* | (**11**) |

The fragments *N*, *W*, *S* and *E* represent the four cardinal directions. Unsurprisingly, these fragments are modeled as wrappers, as they can be stacked to express directions more precisely. The fragments prefixed by the tilde correspond to the inverse fragments of the respective directions. Hence, the sub-expression *N*.*~S* expresses the fact that the north fragment may not occur in the same alternative as S. This formula prohibits invalid compositions, such as N.S or W.E.S. These combinations simply cancel out from the model due to the rule *F.~F* = *0*. Just as in (**8**), the *N* coefficient determines the resolution of the compass.

Having constructed the four morph models, it is possible to compose those models and model so a traveller’s mind, as shown in (**12**).

|  |  |
| --- | --- |
| Traveller = Person.Destination.Path.Compass.Emotion | (**12**) |

A traveller’s state of mind includes the concept of the “self” represented by the *Person* model. The traveller is aware of his or her destination represented by the additional entity fragment *Destination*. In contrast to the two preceding models, which may be considered immutable per a traveller’s instance, the remaining models are mutable. The ongoing journey is represented by an evolution of the *Path* model and the current direction to the destination is represented by an alternative of the *Compass* model. The current compass state (alternative) influences the traveller’s decisions and hence the path. The Emotion model reflects various factors, such as the distance from the destination, the weather condition (outer condition example) etc. Its presence may also influence the path decisions. The dynamics of the whole system and the influence relationships between the sub-models are defined and driven by the morphing strategy of the morph model.

### Morphing Strategies

A morphing strategy is a key concept in OM, which determines the composition of morphs. Whenever a new morph is created or re-morphed, a morphing strategy must be supplied. The morphing strategy is responsible for selecting the model alternative fitting best the outer or inner conditions of the modeled object. Together with a morph model, a morphing strategy forms the so-called *recognizer*. In other words, while a morph model represents the static part of a recognizer, a morphing strategy is its dynamic part. Although it is possible to implement custom strategies, there are three general implementations: *promoting*, *masking* and *rating* strategies [16].

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.

Prototypical Analysis

As mentioned in the introduction, one of the main motivations of this work is the inability of object-oriented programming to provide a prototypical conceptual framework that would be more suitable for modeling every-day phenomena. It has also been suggested that object morphology could help close that gap.

So far only the theoretical aspects of object morphology have been examined without explicit connection to any prototypical conceptual framework. Although it is too early to establish a full-blown methodical manual for prototypical analysis through object morphology, some essential guidelines may already be recognized and formulated. The following list outlines the basic steps of performed when analyzing and modeling a protean phenomenon.

1. Understanding Phenomena
2. Identifying Prototypes
3. Property Analysis
4. Morph Model Construction
5. Binding Properties to Context
6. Morphing Strategy Construction
7. Creating Recognizer

The steps 1-4 constitute conceptual analysis, as they establish the analyzed concept. On the other hand, the remaining steps represent contextual analysis that captures the relationship between the morph model and its context. The result artifact of this process is the recognizer, which is a composition of the morph model and the morphing strategy. The recognizer serves to create morphs according to outer conditions, i.e. the context. It is also responsible for reshaping the morphs as a result of changes in their inner state or in the context.

Reference Implementation (RI)

The goal of the presented thesis is not only to develop the theoretical concept of object morphology but also to provide a reference implementation as a proof-of-concept prototype, also known as Morpheus [16]. The RI is developed as an extension of the Scala programming language and consists of two main components: compile-time and run-time. The compile-time component parses and validates morph models expressed by means of a special type, while the run-time component instantiates objects according to morph models. Because of the chosen language, developers may enjoy the interoperability with other code developed in any language running on the Java Virtual Machine platform.

Applications

So far, OM has been used in a couple of scenarios, mostly in a proof-of-concept way. This section presents two of the most interesting applications. Other applications are described in the thesis [16].

### Onto UML

OntoUML was developed by Giancarlo Guizzardi as an extension of UML [11]. The main motivation for its development was to provide an analyst with a set of special entity types with exactly defined properties to allow faithful modeling of reality. It is based on the findings of Cognitive Science as well as on the rules of modal and set logic.

The Appendix 1 of the thesis [16] demonstrates how to transform the school enrollment OntoUML model to a morph model. That example suggests that Morpheus could fill the gap between OntoUML models and their implementations in standard object-oriented languages. Should there exist a universal transformer generating morph models from OntoUML models, it would be possible to construct the following code generation pipeline: OntoUML -> Alloy -> Morpheus (Scala).

### DCI

Data, context, interactions (DCI) is a software paradigm introduced by [12]. DCI can be seen as a further development of the Model-View-Controller design pattern, whose goal is to provide the illusion of a direct connection from the end user’s brain to the computer’s “brain”, i.e. its memory and processor. In [17] the author of the thesis demonstrates how to build an application following the DCI principles on top Morpheus.

### Square-Rectangle Problem

The square-rectangle problem (SRP) exposes a couple of flaws inherent to object-oriented programming (OOP) [18]. In particular, the problem is closely related to subtyping and inheritance and manifests itself as a violation of the Liskov substitution principle (LSU) [19]. 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.

Although the problem looks quite simple at first, its solution is harder than it appears [19]. There are a number of possible solutions of this problem, but, broadly speaking, none of those solutions solve the problem comprehensively and often introduce secondary problems [20]. In the same article author demonstrates how the square-rectangle problem can be solved when modeled in the framework of object-morphology and implemented using Morpheus.

### Final Case Study: Modeling Human Emotional Facial Expressions

In contrast to the introductory case studies, whose purpose was to define the problem to be solved and to sketch the solution, this final study focuses on applying object morphology to a real world case. The goal of this chapter is to demonstrate how to use the RI presented in the previous chapter to model a complex system while utilizing as much of the RI features as possible. Furthermore, the analysis follows the guidelines of the prototypical analysis developed.

The goal of this case study is to develop an application modeling emotional expressions in the human face. The human face is an example of a complexly varying object (i.e. protean object, morph), whose individual forms correspond to the expressions reflecting the human emotions. The emotions cause electric stimulation of the facial muscles, which are connected to the bones by the one end and to the skin by the other. The contractions of the muscles bend and morph various facial features, such as lips, lids, eyebrows etc., and render so the characteristic expressions.



**Figure** : *Electrically stimulated facial expressions [21]*

The emotions rendering application is designed as a three-tier system, in which each tier is in fact an autonomous protean object. The following list outlines the three tiers along with some constituting elements (i.e. fragments):

* Emotions: Joy, Surprise, Fear, Sadness, Disgust, Anger
* Facial Musculature: Depressor Labii, Levator Palpebrae, etc.
* Facial Features: Eyebrows, Upper Lid, Lower Lid, Upper Lip, etc.

The developed application models facial expressions as reactions on emotions. The analysis of all application tiers followed the raw guidelines for prototypical object-oriented analysis. The application was designed and developed in the framework provided by Morpheus, the reference implementation of object morphology.

The architecture used in this case study exhibits significant resemblance to the three-tier applications. The emotion model is analogous to the persistence tier, which captures the state of the application (i.e. data). The musculature corresponds to the application layer, which can be seen as an engine processing and mediating the data between the storage and the display. And finally, the facial features model clearly corresponds to the presentation tier displaying the state.

The main difference between the traditional three-tier architecture and the protean architecture lies in the adaptivity of the tiers.

Another interesting aspect is that although all building blocks of the tiers are immutable, the whole is mutable. The mutability of the whole is, however, under control of the platform and is transparent to the developer. The developer may thus design the building blocks as immutable units and enjoy all advantages of this approach.

The main downside of this design is poor performance both at compile-time and run-time. At compile-time the bottleneck is definitely the check of the conformance between two morph models. Much of these problems may be attributed to the fact that the reference implementation is still only proof-of-concept software, which has not been subjected to any optimization efforts. Another relevant objection could be raised with respect to the tendency to produce repetitive code in the model elements, especially in the case of the wrappers. The solution could be using a macro to generate a series of wrappers or to apply some approach from aspect-oriented programming.

Evaluation

The main goal of the presented thesis is to developing a new OO paradigm called Object Morphology, a conceptual framework for modeling protean objects; i.e. objects that may assume various forms upon their creation (non-uniform objects) or also during their existence (metamorphing objects).

In the theoretical sections, the thesis develops the OM formalism called R-Algebra. The purpose of this formalism is to provide a tool for the construction and validation of morph models. Another key contribution of the theoretical section is the generalization of the Liskov substitution principle examining the conditions under which one morph model may substitute another one.

The Prototypical Analysis section suggests raw guidelines, formulated in terms of Prototype theory, for analyzing the problems featuring protean objects. The guidelines were applied and roughly verified during the development of the final case study.

One significant achievement of the thesis is the reference implementation of OM, also known as Morpheus. The purpose of Morpheus is to provide a proof-of-concept prototype to allow for the evaluation of the OM paradigm in real applications. There are, nonetheless, some limitations and issues that have yet to be overcome. The main drawback concerning the OM paradigm lies in the fact that OM has not been used in any real-world application yet. Regarding Morpheus, the most important issues are its performance issues and tendency to produce boilerplate code. These issues might be attributed to the immaturity of the software as well as the limitations of the Scala platform, which had to be heavily customized.

The final case study demonstrated the ability of OM and Morpheus to model organic systems, such as emotional facial expressions. The design, which followed the guidelines of prototypical analysis, led to an unusual form of the three-tier architecture.

Other encouraging indicators of OM’s potential are:

* A solution to the Rectangle/Square Problem
* An implementation of the Data-Context-Interactions architecture
* Mapping OntoUML elements onto morph models

# Further Work

In terms of future development, it could be worthwhile to introduce OM into some dynamic language, preferably into JavaScript. Today, the applications developed in JavaScript are often quite complex and organic presentations, which might benefit from being designed and developed in an OM framework. Such a framework could bring a sort-of compile-time validation to those applications, as morph models can be validated at compile-time, in principle.

Next, the theoretical foundations and R-Algebra do not elaborate in detail the formalism for composition of stackable fragments (i.e. wrappers). Such additional formalism could become the base for an advanced sort of type system used to design complexly evolving systems.

Should OM become more recognized as a vital paradigm, it would be beneficial to further develop the prototypical methodology outlined in the thesis. That methodology should also include an extension of UML, which would introduce new stereotypes and notation suitable for OM analysis and design.

# References

1. Zhen, L. & Tate, D. (2011) *Managing Intra-Class Complexity With Axiomatic Design and Design Structure Matrix Approaches*. Dajeon: Proceedings of ICAD2011, The Sixth International Conference on Axiomatic Design.
2. Booch, G. (1993) *Object-Oriented Analysis and Design with Applications*. 2nd ed. Redwood City, CA: Addisson-Wesley Professional.
3. Gamma, E. et al. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.
4. Garzás, J. & Piattini, M. (2007) Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices. Hershey PA, 2007. Idea Group Publishing.
5. Madsen, O.L. (n.d.) *Open Issues in Object-Oriented Programming, A Scandinavian Perspective* [Online]. Computer Science Dept., Aarhus University Available at: https://users-cs.au.dk/olm/index.html/PUB/OpenIssuesInOO.pdf .
6. Deitel, H.M. & Deitel, P.J. (2005) *Java How To Program*. 6th ed. Upper Saddle River, New Jersey: Prentice Hall.
7. Rosch, E. (1973) Natural categories. *Cognitive Psychology*, (4), pp.328-50.
8. Taivalsaari, A. (1996) Classes vs. Prototypes, Some Philosophical and Historical Observations. *Journal of Object-Oriented Programming*, pp.44-50.
9. Krogh, B., Levy, S., Subrahmanian, E. & Dutoit, A. (1996) Strictly Class-Based Modeling Considered Harmful. In *System Sciences*. Wailea, HI, 1996. IEEE.
10. Mihelič, J. (2011) *Prototype-based Object-Oriented Programming* [Online]. Univerza v Ljubljani Available at: http://lalg.fri.uni-lj.si/~uros/LALGinar/arhiv/ProtoOOP.pdf .
11. Guizzardi, G. (2005) *Ontological Foundations For Structural Conceptual Models*. Centre for Telematics and Information Technology, University of Twente.
12. Reenskaug, T. & Coplien, J. (2009) *The DCI Architecture: A New Vision of Object-Oriented Programming* [Online]. Available at: http://www.artima.com/articles/dci\_vision.html .
13. Steimann, F. (2006) The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. New York, NY, USA, 2006. ACM.
14. Harrison, W. & Ossher, H. (1993) 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., 1993. ACM.
15. Řepa, V. (2012) Modelling Life Cycles of Generic Objects. In *Proceedings of the International Conference on Information Systems Development*. ISBN 978-1-4614-7539-2.
16. Šlajchrt, Z. (2016) Object Morphology – A Protean Generalization of Object-Oriented Paradigm. *Dissertation Thesis*. [Online]. Available at: <https://github.com/zslajchrt/morpheus/raw/master/src/main/doc/thesis/2016-06-03_Thesis-finalR.pdf>.
17. Šlajchrt, Z. (2016) Designing Applications Using Data-Context-Interactions Architecture In Morpheus. *Journal of Systems Integration*, 7(2).
18. Cline, M. & Lomow, G. (1995) *C++ FAQs: Frequently Asked Questions*. Reading: AddisonWesley.
19. Majorinc, K. (1998) Ellipse-Circle Dilemma and Inverse Inheritance. In *Proceedings of the 20th International Conference of Information Technology Interfaces*. Pula, 1998.
20. Šlajchrt, Z. (2016) A Solution to the Square-Rectangle Problem Within the Framework of Object Morphology. *Acta Informatica Pragensia*, 5(1), p.34–45.
21. Duchenne, G.B. (1862) *Mécanisme de la physionomie humaine, ou analyse électro-physiologique de l'expression des passions*.

# Terminology

**Alternative –** an *alternative* describes one of the forms of a protean object.

**Fragment –** a *fragment* is the basic building block in object morphology. It represents a property or feature of a protean object. It semantically corresponds to the concept of trait as defined in Scala or Groovy and as it has been used throughout the case study. A fragment represents a typological, behavioral and structural element of protean objects.

**Morph –** a *morph* is a representation of a protean object. As such it is an instance of an alternative from the morph model describing the protean object. The morph is created by a recognizer according to the alternative selected by the recognizer’s morphing strategy.

**Morph Model –** a *morph model* describes the alternative forms (alternatives) of a protean object. 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).

**Morphing Strategy –** a *morphing strategy* receives all possible alternatives and selects the one that matches best the protean object (phenomenon) to be represented.

**Object Morphology –** *Object Morphology* is an object-oriented paradigm for modeling protean objects, in which a protean object is modeled through the construction of a morph model describing the forms that the protean object may assume. The forms are called morph alternatives.

**Protean Object –** a *protean object* is a term referring to a phenomenon occurring in a multitude of forms and defying the traditional Aristotelian class-based categorization. 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, different roles of an individual in society, crisis and other biological, social or economic phenomena.

**Recognizer –** a *recognizer* corresponds to the concept of the class builder identified in the case studies. It is a device designed to compose and recompose objects, the so-called morphs, from smaller parts called fragment stubs in accordance with the alternative selected by the morphing strategy, which is a part of the recognizer. The previously illustrated airport scanner is an example of such a device.

**Wrapper** – a *wrapper* is a special kind of fragment. The wrappers are designed to override the behavior of other fragments. Thus they cannot exist in an alternative alone, as they are dependent on the overridden fragment. This condition must be checked during the consistency and completeness analysis of morph alternatives.

1. The pseudo-code in this section is inspired by Scala, which is the language on top of which the proof-of-concept implementation of OM is developed. [↑](#footnote-ref-1)