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

Student   : Mgr. Zbyněk Šlajchrt

Supervisor   : Ing. Rudolf Pecinovský, CSc.

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

                 Doc. Ing. Miroslav Virius, CSc.

2016

Prohlášení:

Prohlašuji, že jsem dizertační práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze dne 31. 5. 2016
. . . .

Zbyněk Šlajchrt

Poděkování

Rád bych tímto vyjádřil své poděkování všem, kteří mne inspirovali k napsání této práce a současně i těm, kteří mne povzbuzovali k jejímu dokončení. Speciální dík patří mému školiteli Ing. Rudolfu Pecinovskému, CSc. za vedení práce i odbornou a stylistickou pomoc. Dále bych rád poděkoval doc. Ing. Vojtěchu Svátkovi, Dr. za cenné komentáře a důležité nasměrování, které mi pomohlo odhalit prostor pro další uplatnění myšlenek rozvíjených v této práci.

*Shouting and screaming, we launched an assault on him,
casting about him
arms-holds; nor did the old man forget his craft or his
cunning
but to begin transformed himself to a long-bearded lion,
then to a serpent and next to a panther and then to a
huge boar;
then he became running water, a tree next, lofty and leafy.
We maintained irresistible grips with a resolute spirit.*

Dionysiaca, Nonnus of Panopolis, Translated by R. Merrill

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

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 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 (Zhen & Tate, 2011; Misra & Akman, 2008; Booch, 1993; Gamma et al., 1994). According to the TIOBE programming community index the majority of the most popular programming languages adopt the object-oriented paradigm to some extent (TIOBE, n.d.). The latest polls show that Java, as a typical representative of an object-oriented language, has been dominating this chart over the decade and is currently on a rapid rise, which might be attributed to the rising popularity of Android, a mobile platform based on Java (IDC, 2015).

According to its advocates, the object-oriented approach has had a significantly positive impact on many aspects of software development. The most cited benefits of this approach are the reduction of complexity and the reuse of code, which can be attributed to the basic tenets of OO, such as encapsulation, inheritance, abstraction and polymorphism. Comparatively important are also other effects such as enhancement of understandability and improvement in rapid prototyping (Bourne, 1992; Parnas, n.d.).

Other proponents highlight the influence of OO principles on the overall project qualities. For instance, Booch claims that OO provides better project quality in comparison with procedural programming and helps reduce the overall project time (Booch, 1993).

In the 1990s the focus in the field of OO development shifted from the main interest being reusing the code to analysis and design. It was then that the design patterns for designing and developing OO software gained broad public awareness (Gamma et al., 1994; Parnas, n.d.). The existence of disparate notational systems used to design OO software was the main driver for the development of the Unified Modeling Language (UML), which was later adopted as a standard by the Object Management Group (OMG) (Booch, 2005). The rise of UML importance was followed by an upsurge of the development of various modeling tools and methodologies (Rational Software, n.d.).

In contrast to OO, which has been recognized by its community as an approach contributing notably to the reduction of complexity, UML has become a target of criticism for introducing a new kind of complexity, having an inconsistent schema and being ambiguous (Siau & Cao, 2001). Furthermore, despite its wide scope, UML fails in coping with intra-class complexities, such as the relationship between various methods and attributes (Zhen & Tate, 2011; Misra & Akman, 2008). According to Petre, UML is now used mainly in education and academic papers, while in industry it is used rather infrequently (Petre, 2013).

In addition, OO modeling is not able to capture well the tacit knowledge in the form of experience, heuristics and human competencies, which, as presented in (Garzás & Piattini,

2007), constitutes the major part of design knowledge today. Furthermore, no object-oriented notation is rich enough to be able to model all key aspects of the real world (Madsen, n.d.).

Aristotelian conceptual framework

In spite of the above-mentioned criticisms, modeling and design have established themselves as important parts of object-orientation. 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 (Madsen, n.d.; Deitel & Deitel, 2005).

A concept in the Aristotelian view is characterized by the following three terms: *extension*, *intension* and *designation*. The extension of a concept is described as the collection of phenomena, i.e. class instances in the context of class-based languages. On the other hand, the intension of a concept refers to the collection of properties characterizing the phenomena in the concept's extension; i.e. a class description. And finally, the designation of a concept is a collection of names used to label the concept; i.e. class names (Madsen, n.d.).

The concepts in this view are hierarchically organized and all phenomena in a concept's extension possess the so-called *defining properties*, which are a sub-set of the concept's intension. Such a view presumes that the presence or absence of a property in a phenomenon is objectively determinable.

Object Morphology

This thesis presents an alternative and "non-aristotelian" object-oriented paradigm called *object morphology*, 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 (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*.

The notion of a morph model is key in object morphology. A morph model defines all possible forms of objects instantiated from the model. To put it another way, a morph model can be seen as a representation of a prototypical concept and the individual alternatives as prototypes or exemplars. The alternatives are composed of the so-called model elements, which are analogous to a concept's intension (i.e. properties).

A morph model is in fact the prototypical analogy to class. A morph model includes a static description of alternatives/exemplars/prototypes that may be checked at compile-time for consistency, for example. 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 Structure of this Thesis

This thesis begins with the research section followed by two introductory case studies, whose aim is to examine three well-known object-oriented languages (Java, Groovy and Scala) that are used to develop an application processing simple non-uniform data in the first case and a simple adaptive service in the second case. Both case studies also suggest another solution based on the tenets of object morphology.

The next section establishes a theoretical basis for object morphology by introducing the notion of an abstract recognizer and developing formalism called *R-Algebra*. This theoretical part also presents a generalized version of the Liskov substitution principle that examines the conditions under which instances of one morph model may replace instances of another morph model.

In *Prototypical Analysis* the thesis establishes some grounds for an object-oriented analysis based on the prototypical (non-Aristotelian) view of concepts and object morphology.

Section *Reference Implementation* explains the design and implementation of the proof-of-concept implementation in Scala.

The *Final Study* section demonstrates how object morphology and its reference implementation may be used to model human emotions and their expression on the human face as an example of a complicated mutable system.

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.

2 Research And Related Works

2.1 Philosophical Aspects

While the Aristotelian conceptual framework has proven itself to be a useful tool for modeling concepts from many fields, it is surprising how difficult it is to apply this framework to everyday concepts. For a rather humorous example of this difficulty let us consider the following discussion at Quora about the nature of the watermelon (Quora, n.d.). The discussion members were putting forward various arguments or criteria to decide whether a watermelon is a fruit or a vegetable. One of the answers illustrates quite well how ostensibly illogical the human cognitive system might be: "(…) in culinary terms, a fruit is not a fruit if it isn't sweet". The watermelon case becomes even more humorous if coupled with the fact that in 2007, Oklahoma's House of Representatives declared the watermelon to be the state vegetable despite the fact that most people see it as a fruit (The Guardian, 2007; Factaz, n.d.).

Certain Problematic Cases

There are, moreover, other, more serious cases, in which the Aristotelian approach fails. For instance, the Diagnostic and Statistical Manual of Mental Disorders (DSM) offers standard criteria for classification of mental disorders (American Psychiatric Publishing, 2013). According to this manual a person might be diagnosed with a given disorder as long as N out of M symptoms have been present for a given period of time. Such criteria, however, obviously breaks the fundamental Aristotelian rule according to which all phenomena in the extension of a concept possess all the defining properties of the concept. Here, nevertheless, there may not be any defining symptoms at all, as all may be equally important. It also follows that no symptom must be necessarily present in all instances of the given disorder; i.e. the extension of the concept.

Modeling of evolving systems may also be a challenge for the Aristotelian conceptual framework. To illustrate this difficulty, let us consider the modeling of a computer operating system that is being gradually upgraded. Each upgrade is considered either as an improvement or as a new feature. From time to time, the upgrade takes away older functionality in favor of some new one. From the user's perspective the system apparently preserves its identity in the continuous process of upgrading. Yet the system turns into a different thing at some point; this metamorphosis might manifest itself by the inability of the system to run older applications, for instance. In this scenario the modeled concept is "my computer", whose extension consists of all versions of the computer. In contrast to the previous cases, here, the extension of the concept uses the temporal dimension and a single identity, instead of many identities in the spatial dimensions.

A similar phenomenon, called *ring species*, is known in biology. A ring species is a connected series of neighboring populations, in which each population can interbreed with closely situated related populations. As the related populations spread into a greater area there may emerge at least two populations of the same species that are too distantly related to interbreed. Over time, such non-breeding, though genetically connected, populations may close the ring by meeting each other in the same region (Irwin et al., 2005). Dawkins in (Dawkins, 2005) considers this phenomenon an important piece of evidence for evolution. He claims that ring species "are only showing us in the spatial dimension something that must always happen in the time dimension"

The two preceding examples illustrate the problem of transitivity, which is nicely described in (Hampton, n.d.): "People will say that a car headlight is a type of lamp, and that a lamp is a type of furniture, but the headlight is clearly not furniture." Unlike in this example, true classes in the Aristotelian view of concepts always obey transitivity.

Arguably the most defiant phenomena with respect to their classification can be found in the social sciences, such as economics and sociology. Such phenomena not only exhibit strong mutability both in time and space, but they also may include a feedback loop causing, for instance, that people respond to the classification by changing their behavior that may itself be a constitutive factor for the classification. These cases indicate that there is a conflict between the existence of kinds and the mutability of the objects that are supposed to belong to those kinds (Bird & Tobin, 2015).

Philosophical Paradoxes

In addition to those every-day cases there are a number of philosophical paradoxes and problems, whose aim is to elucidate the difficulties with classification of things, most of them based on vagueness of predicate, such as Sorites paradox, the Ship of Theseus (Stanford Encyclopedia of Philosophy, 2011) and George Washington's Axe .

Sorites paradox illustrates a fixed boundary problem (or a problem of unknowable boundaries); in other words, the impossibility to determine at what stage the heap of sand becomes a non-heap when removing individual grains from the heap.

On the other hand, the Ship of Theseus and George Washington's Axe are nice illustrations of the conflict between the identity of an object and a gradual replacement of its parts.

Nonetheless, probably the most influential problem in terms of theoretical significance is Wittgenstein's problem of the classification of games (Wittgenstein, 1953), on which Wittgenstein explains the idea of *family resemblance*. This idea suggests that things, which might be seen as sharing some common property, could in fact be connected by a series of overlapping similarities, whereas no property is held in common by all the things (Philosophy Index, n.d.). Wittgenstein illustrates this idea by examining the concept of games. It turns out that there is no game feature that is held in common by all games. He

comes to the conclusion that the absence of a single common feature makes abstraction impossible. A related problem called *wide-open texture* suggests that the concept of family resemblance does not suffice to limit the extension of concepts; in other words, it is not possible to define a closed set of phenomena by using family resemblance only and that anything and everything will fall under a given concept. The description of the problem and some solutions can be found in (Way, 1991; Andersen, 2000).

The preceding examples clearly highlight the contrast between the so-called *monothetic* and *polythetic* classifications. A monothetic class is defined in terms of properties that are both necessary and sufficient for the class members. In a monothetic class all members share the common properties. On the other hand, a polythetic class is defined in terms of a broad set of properties, which are neither necessary nor sufficient. In a polythetic type all members resemble one another, but they are not identical (Hjoerland, n.d.; Needham, n.d.; Beckner, 1968). The notion of a monothetic class corresponds to the Aristotelian view, while the notion of a polythetic class is associated with Wittgenstein's idea of "family resemblance", which he likened to overlapping fibers in a rope (Wittgenstein, 1953; Needham, n.d.; Beckner, 1968).

2.2   Prototype Theory

Polythetic classification is closely related to *prototype theory* formulated by Eleonor Rosch in the 1970s (Rosch, 1973). 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*.

Exemplar theory is another theory based on similarity and probability rather than on exact rules. According to this theory people categorize new stimuli by comparing them with instances already stored in memory (Nosofsky et al., 2011). This process distinguishes exemplar theory from prototype theory, which suggests that a new stimulus is compared to a single prototype in a category.

The distinction between prototypes and exemplars is aptly emphasized by Hampton in (Hampton, n.d.): "Unlike a best example, an abstract prototype allows for the representation of different possible values of relevant features – such as that apples can be red, green, brown, or yellow, or that furniture can be sat on, slept on, used for storing things, or provide a surface for supporting things. An apple that had all these colors or a piece of furniture that served all these functions would not necessarily be prototypical. … Prototypes then are the centers of clusters of similar objects, and prototype concepts form similarity-based categories."

According to (Rouder & Ratcliff, 2006) the learning process uses different strategies with respect to how familiar the subject is with a given concept. Rouder suggests that individu-

als use rules when the new things are confusable. In other cases, when the new things are distinct, the individuals use exemplars. Medin, Altom and Murphy found (Medin et al., 1984) that individuals who used a mixture of prototype and exemplar information made the most accurate judgments. To sum up, the rules strategy seems to be used for unfamiliar things and is based on exact criteria; in other words, it corresponds to the Aristotelian approach. It is also the slowest, most accurate and least economic strategy. On the other hand, the exemplar and prototype strategies are used for familiar things, with the prototype strategy being the fastest, least accurate and most economic.

In general, to demonstrate that a domain exhibits prototype structure, one or more of the following four phenomena should be present when examining the concepts in that domain (Hampton, n.d.):

- *Vagueness*. There exist cases whose membership in a category is uncertain

- *Typicality*. Items in a category differ in terms of their "goodness of example"

- *Genericity*. People tend to produce descriptions of a concept that are generically true of the class, though not true of all members

- *Opacity*. People are generally unable to formulate a universal rule for determining category membership

According to Hampton, as far as the opacity is concerned, people do not respect the constraints of set logic when attempting to define the rules for category membership; for instance: "chess is a sport which is a game, but that it is not a sport, they say that a mushroom is not a fruit and that it is not a vegetable, but that it is one or the other, and they say that a tent is not a dwelling, but that it is a dwelling that is not a building" (Hampton, n.d.). A similar formulation has been presented above in the case of the discussion of the nature of a watermelon. Furthermore, in (Hampton, n.d.) Hampton observes that the importance of a property decreases with the addition of more modifiers to the general class (which is in contrast to the logic). For instance, in the following three statements "Ravens are black, Jungle ravens are black, Young jungle ravens are black" the weight of the "black" property decreases.

Hampton further points out that while Zadeh's fuzzy set logic (Zadeh, 1965) found many useful applications in control engineering, it made the wrong predictions about behavioral data such as judgments of typicality, or categorization in complex concepts (Osherson & Smith, 1981; Roth & Mervis, 1983). After all, despite its fuzziness, the fuzzy set logic is still a consistent logic in itself, in contrast to the "logic" used by individuals.

Hampton then concludes that: "The discovery of this non-logical system for combining concepts is one of the key factors supporting prototype representations, since it flies in the face of grounding the meaning of terms in extensionally delineated classes in the world, and of grounding complex concepts in set logic" (Hampton, n.d.).

2.3   Prototype Theory And Object-Oriented Programming

As mentioned in (Taivalsaari, 1996), although the Aristotelian model was abandoned a long time ago, it remains the prevalent one in object-oriented programming.

Similarly, 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 (Madsen, n.d.).

Madsen further stresses the difference between prototypical concepts and prototype-based languages. He answers resolutely negatively to a question whether or not the prototypical concepts correspond to modeling based on prototype-based languages. The reason is that there is no notion of concepts involved in prototype-based languages in fact; the prototype-based modeling only models phenomena and relations between them. On the contrary, the prototypical view of concepts is based on a distinction between phenomena and concepts. He concludes that prototype-based languages may be useful in early stages of analysis and design in order to quickly understand a new phenomenon. However, sooner or later the acquired knowledge will require some organization by means of classification and composition, which will result in grouping phenomena into concepts along with forming the concepts into hierarchies.

Additional information on the relationship between prototypical concepts and prototype-based language can be found in (Taivalsaari, 1996; Krogh et al., 1996; Mihelič, 2011).

2.4   OntoUML

OntoUML was developed by Giancarlo Guizzardi as an extension of UML introducing new elements and diagrams to facilitate ontological modeling. 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. Since the new elements are introduced using the standard UML means, i.e. primarily UML stereotypes, OntoUML remains within the UML boundary and is thus compatible with the current CASE tools (Pergl et al., 2012).

OntoUML is the modeling language of its foundational ontology called Unified Foundational Ontology (UFO). UFO is subdivided into three levels: UFO-A, UFO-B and UFO-C. The lowest (basic) level – UFO-A – deals with static categories, such as objects, roles, types and the relationships among them. This ontology was completely developed by Guizzardi in his thesis (Guizzardi, 2005). On the other hand, the UFO-B ontology focuses on dynamic and temporal aspects of the categories identified in UFO-A. Examples of such

categories are events and causal links between them. The highest ontology UFO-C is built on top of the two preceding ontologies and introduces categories related to social aspects of the categories. In contrast to UFO-A, UFO-B and UFO-C are still under development (Pergl et al., 2012).

It follows from the above that UFO closely relates to the subject of this thesis, especially as far as the focus on the faithful modeling of complex and mutable phenomena is concerned. Therefore the following passages explain in more detail the key aspects of UFO-A.

As mentioned above, OntoUML introduces a set of categories of entity types. The individual categories constitute the hierarchy shown in Figure 1.
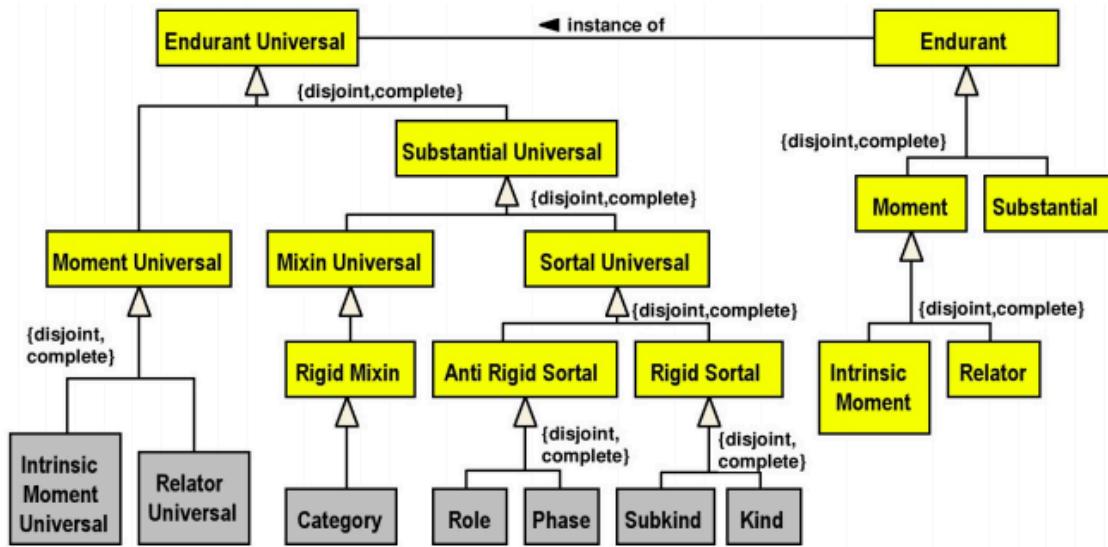

Figure 1: Excerpt of UFO taxonomy (Carvalho et al., 2015)

The two roots of the hierarchy — `Endurant Universal` and `Endurant` — capture the type-instance dichotomy. The `Endurant` category further forks into two categories `Substantial` and `Moment`, where the former represents independent individuals (objects), while the latter represents existentially dependent individuals; i.e. their existence depends on the existence of other individuals. The `Relator` sub-category of `Moment` represents relations, such as marriage.

The distinction between substances and moments is also present in the subdivision of the `Endurant Universal` category, which can be considered as the category of types; the `Moment Universal` and `Substantial Universal` categories represent the types whose instances are moments, resp. substantials.

To understand how the universals are organized in the schema, a couple of additional type properties should be explained.

Identity — a type with this property possesses identity. Some types may supply identity, while others may inherit it from their parent types. `Sortal Universal` thus represents the types with identity, while `Mixins Universal` is the category for the types without identity.

Rigidity — if a type is *rigid*, then no its instance can cease to be an instance of this type without ceasing to exist (Kind and Subkind). And analogously, a type is *anti-rigid*, if its instances may not always be instances of this type. The Role and Phase universals are examples of anti-rigid categories.

Figure 2 illustrates a number of UFO-A elements in a diagram modeling the entities involved in the process of school enrollment.
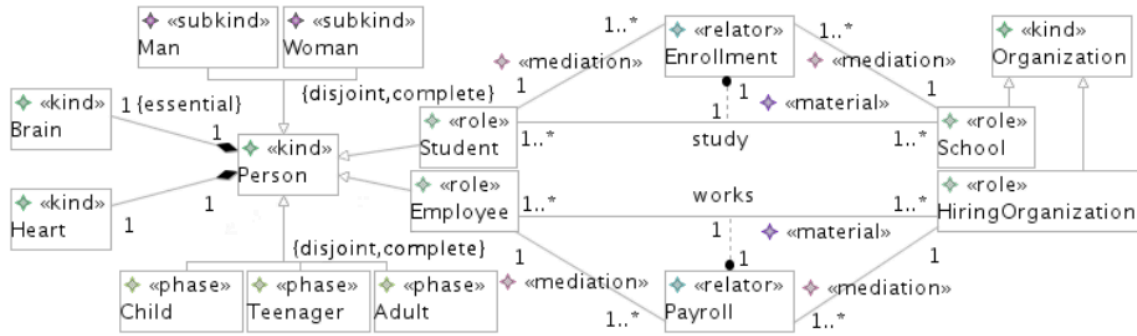

Figure 2: The school enrollment model (Benevides et al., 2009)

Further, OntoUML deals in detail with the analysis of part-whole relationships. In UML, this relationship is expressed using aggregation and composition, which are not, however, defined clearly and completely (Pergl et al., 2012). OntoUML distinguishes the following types of the part-whole relationships from the perspective of the whole:

Optional Part – the part is optional, the whole may exist without it.

Mandatory Part – the part is a mandatory part of the whole, the whole cannot exist without it. However, the particular instance of the part may change.

Essential Part – the part is a mandatory and unchangeable part of the whole.

Similarly, OntoUML defines other three types of this relationship with respect to the whole:

Optional Whole – the part may exist without the whole.

Mandatory Whole – the part must always belong to a whole. A particular part instance may be assigned to another whole.

Inseparable Part – a part always belongs to the whole. It cannot be re-assigned to another whole.

Since the resulting OntoUML models may be quite complex, it is necessary for a modeler to be able to validate the models. In (Braga et al., 2010), the authors present a method of the validation of such models using the logic-based language Alloy. The presented method explains how to use Alloy to validate OntoUML models by generating sequences of model

instances and assertion counter examples. The resulting sequences confront the modeler with his or her aims and make it easier to uncover the mistakes in the design.

The *oNTOjs* project is an adoption of UFO to the JavaScript language. It allows the definition of kinds and roles with multiple and dynamic role classification (Wagner, n.d.). In conjunction with other libraries *mODELcLASSjs*, *mODELvIEWjs*, it provides an MVC framework for web application. However, this project suffers from the lack of good documentation and real-world use cases.

Generally speaking, OntoUML and UFO find their applications in the fields where it is important to model reality with high accuracy. OntoUML provides better expressivity to a modeler while staying within the bounds of the traditional UML. It should, in theory, allow the modeler to keep using the CASE tools for developing models within the framework of OntoUML.

According to (Pergl et al., 2012) there are several obstacles preventing OntoUML from becoming a widely accepted approach.

- A lack of literature and information resources

- A tiny amount of widely accessible examples and use cases

- The absence of tutorials and training courses

- A small community and unavailable support

- The absence of specialized CASE tools

The author of this thesis may confirm the validity of at least some of the above-mentioned issues, which he encountered when doing the research for this section.

Comparison of OntoUML and Object Morphology

When modeling the reality using OntoUML, a modeler uses the UFO ontology from which he or she selects the right elements and relationships to describe real-world phenomena. The modeler's degrees of freedom are necessarily harnessed by the given ontology, which, on one hand, may make the modeler's life easier since the ontology implicitly guides the modeler when developing a model, but, on the other hand, this benefit may turn into an obstruction when modeling a borderline phenomenon that defies the given ontology.

On the contrary, within the framework of object morphology, the modeler models the observed phenomena by collecting individual instances of the phenomena and analyzing the variability in their properties. The goal of this process is to identify one or more prototypical instances and to describe the ways in which the fringe instances divert from the prototypes. The product of such a process is a morph model, which captures the prototypical forms as well as the mutations observed in the phenomena. In contrast to the ontology-based modeling, where the resulting model is necessarily coupled with the used ontology,

here, the product of the OM analysis, i.e. the morph model, reflects only the observations done by the modeler.

In general, the ontology-based modeling can be described as a top-down approach, while the OM modeling can be viewed as a bottom-up approach. Although substantially different in their nature, the two approaches may yield interesting results if brought together, as indicated in Appendix 1: Using OM to implement an OntoUML model. The code in this appendix demonstrates how to transform the school enrollment OntoUML model (shown in in Figure 2) to a morph model. That example suggests that OM could fill the gap between OntoUML models and their implementations in standard object-oriented languages. Should there exist a universal transformer generating Morpheus morph models from OntoUML models, it would be possible to construct the following code generation pipeline:

<div align="center">OntoUML -> Alloy -> Morpheus (Scala)</div>

The modeler would use his/her favorite UML tool to create an OntoUML model using the OntoUML stereotypes. When finished, the Alloy generator presented in (Braga et al., 2010) would be used to verify the consistency of the model. After the validation, the model would be transformed to Scala code consisting of one or more Morpheus morph models.

Although it is too early to assess the feasibility of the OntoUML->Morpheus transformer, it is certainly a vital direction for further work.

2.5   Alternative Approaches In Object-Oriented Programming

This subsection provides a brief overview of alternative approaches and paradigms developed in response to the insufficiencies and weaknesses of the traditional object-oriented programming.

2.5.1   Data-Context-Interactions

Data, context, interactions (DCI) is a software paradigm introduced by (Reenskaug, 2008) and further elaborated by (Coplien & Bjørnvig, 2010). Its main goal is to bring the end user's mental models and computer program models closer together. In other words, the user must feel that he or she directly manipulates the objects in computer memory that correspond to the images in his or her head. In this respect, 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 (Reenskaug & Coplien, 2009).

According to this paradigm, data, context and interactions are the three fundamental facets of the end user's interpretation of computer data. Each of these facets is considered a component of the application under development.

The data are represented by domain objects, which "know" everything about their state and how to maintain it, but "know" nothing about other objects in the system (Hasso & Carlson, 2014). Coplien et al. in (Coplien & Bjørnvig, 2010) use the term *dumb objects* to refer to those domain objects. The technique for the identification of domain objects responsibilities is often called "Do it Myself" and is described in (Wirfs-Brock & McKean, 2003) and (Coad et al., 1997).

Next to the domain objects there is another kind of objects called interaction objects. These objects capture the interactions between domain objects as system functionality. Importantly, the interaction objects are treated by DCI as first class citizens. In contrast to domain objects, the concept of interaction objects is a novelty introduced by DCI.

Only after the data are put into a context, in which they are subject to interactions amongst themselves, can the data yield some information (Reenskaug, 2008). The context can be seen as the locus of the use case enactment in the architecture encapsulating the interactions (or the roles) that define the system dynamics (Coplien & Reenskaug, 2012).

In summary, in DCI, every use case is represented by means of context. The context defines roles performing interactions among themselves. Each role in the context is played by one corresponding domain object (data, entity). The role contains the code that would otherwise reside in the object's class. Thus, roles effectively separate the stable part of the code from the unstable.

The context itself only defines roles and triggers the use-case. The context and roles should reside in a single dedicated file so that one can easily investigate the interactions.

DCI is anchored in object-oriented programming (OOP); however, it must cope with some inherent flaws in OOP. It is generally accepted that OOP is very good at capturing the system's state by means of classes and their properties. OOP is also good at expressing operations with the state captured by a class, unless these operations involve some kind of collaboration with instances of other classes.

Nevertheless, implementing applications within the DCI paradigm is quite difficult in the traditional OOP, by large on account of the fact that OOP fails to properly express collaborations between objects (i.e. use cases), as shown in (Savkin, 2012). Let us consider an object that appears in several use cases and behaves very differently in each case. Because of the conceptual insufficiency of OOP, a developer is forced to express such a use-case-specific behavior of the object as an operation in the object's class. Unfortunately, this has several undesirable consequences:

First, there is no single file or other artifact dedicated solely to one use case where a developer could see all interactions among objects. This makes orientation in and maintenance of the code quite difficult.

Second, the whole behavior of the use case is scattered across the classes of the collaborating objects. This forces the developer to add a number of unrelated methods to classes with every new use case (causing higher coupling and lower cohesion of classes).

Third, it is virtually impossible to separate the stable part of the code, i.e. that which captures the data, from the variable part, i.e. the use cases (behavioral part).

There are a number of DCI implementations in various programming languages, which more or less completely implement the DCI elements.

A simple example of how to build an application following the DCI principles on top Morpheus is demonstrated in (Šlajchrt, 2016).

2.5.2 Mixins

Mixins or *dynamic traits* provide a means for encapsulating a specific object behavior, such as interactions with other objects, into a special language construct reminding a class with no state (Groovy, 2016). 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 (e.g. 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.

2.5.3 Dependency Injection

Dependency injection allows segregating some object's behavior into an external object. This segregation is accompanied by the specification of the interface between the original and external object. The behavior of the original object may be adapted to a given context (use case) by injecting an external object (role) at runtime. However, this approach suffers from the problem of *object schizophrenia*, which is a condition in which the execution of some logic is carried out in the context of two or more objects (more selves), instead of only one. Such a condition may lead to subtle problems in the design (Sekharaiah & Ram, 2002).

2.5.4   Aspect-Oriented Programming

Aspect oriented programming (AOP) aims at increasing modularity of programs by the separation of crosscutting concerns, which is the goal shared with DCI. AOP provides a developer language tools for specifying the so-called *pointcuts* referring to places in the existing code, where additional behavior, such as logging or profiling, will be added. Nevertheless, Steinmann (Steimann, 2006) polemicizes with the claim that AOP improves the code quality. Instead, he argues that AOP has a negative impact on both modularity and readability of the code.

2.5.5   Role-Oriented Programming

Role-oriented programming (ROP) is in many respects similar to DCI, mainly with regards to the dependency of roles on context and the emphasis on the interaction among a group of roles within the context. There are, however, a couple of differences between these two paradigms. While ROP encourages using inheritance as an important concept in designs, DCI avoids using inheritance completely when modeling roles. Furthermore, according to (Coplien & Reenskaug, 2012), object schizophrenia remains a problem in ROP.

2.5.6   Subject-Oriented Programming

Subject-oriented programming (SOP) aims at facilitating the development and evolution of suites of cooperating applications. In SOP, these applications can be developed separately and composed afterwards. A new application can be introduced to an existing composition of applications without requiring modification of the other applications. The term subject refers to a collection of state and behavior of an object as seen by a particular application (Harrison et al., 1995). In other words, subjects capture subjective perceptions of objects. A subject may be seen as a close relative to a role in DCI. Also, there is close relationship to AOP. In contrast to AOP, SOP restricts placing join-points to field access and method call.

Subject-oriented programming is an object-oriented paradigm in which the properties of objects are provided by subjective perceptions; in other words, the properties are not intrinsic to the object themselves. SOP suggests organizing classes into composable "subjects", which may be composed to form larger subjects (Harrison & Ossher, 1993). This paradigm is notable mainly on account of the introduction of the subject into OO modeling. Unfortunately, at the time of this writing the development of SOP seems abandoned.

2.6   The Square-Rectangle Problem

The square-rectangle problem (SRP) exposes a couple of flaws inherent to object-oriented programming (OOP) (Cline & 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).

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 et al., 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, 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, 2009).

Although the problem looks quite simple at first, its solution is harder than it appears (Majorinc, 1998). 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 (Šlajchrt, 2016).

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.

3   Introductory Case Studies

3.1   1st Case Study: Structural Aspects Of Multidimensionality

For the sake of simplicity and clarity let us assume the following story:

An airport outsources its security check procedure to a company operating high-tech scanners, which are able to recognize three and two-dimensional objects in baggage. After each scan the scanner creates a record describing the recognized objects in a piece of baggage. The record is provided with some metadata and sent to a server at the mother company's premises through a wide-area network connection. The company's server receives a potent stream of such events from a number of installations at several airports. The events in the stream are stored in a clustered database, where they wait for further processing.

A major limitation of the scanner is that it is able to classify baggage items only by their geometrical properties and material. The recognized item is described as a plain geometrical object and not as a wallet, shaver etc.

The owning company comes to a conclusion that the data cannot be easily utilized as such due to its primitive characteristics and possible infringement of the contract by selling it in raw form to another entity, which could combine it with its own data and possibly misuse the insights by revealing sensitive information. The owner realizes that without discovering a new context the scanner data is virtually of no use.

But before such a context is discovered and applied, an overview of the scanner protodata should be given.

3.1.1   Modeling Multidimensional Data

Scanner Protodata Overview

The scan event consists of two parts: metadata and data. The metadata contains the time of the scan, the serial number of the scanner and the id of the baggage. The data part consists of a list of items recognized in the baggage. Each item contains space coordinates relative to the baggage and geometrical and material properties. A sample of a scan event is in Listing 1.

Listing 1: A scan event sample

```json
{
  "id":8488484,
  "scanTime": "2013-05-23T00:00:00Z",
  "scannerId": 78700,
  "items": [
    {
      "id": 0,
      "shape": "cylinder",
      "material": "metal",
      "x": 234.87,
      "y": 133.4,
      "z": 12.94,
      "diameter": 13.45,
      "height": 0.45,
      "reflexivity": 0.8
    },
    {
      "id": 1,
      "shape": "rectangle",
      "material": "paper",
      "x": 673.87,
      "y": 394.4,
      "z": 132.93,
      "width": 13.45,
      "height": 8.21,
      "reflexivity": 0.2
    },
    {
      "id": 2,
      "shape": "rectangle",
      "material": "metal",
      "x": 294.87,
      "y": 123.4,
      "z": 10.93,
      "width": 23.45,
      "height": 22.45,
      "reflexivity": 0.9
    }
  ]
}
```

The sample suggests that the structure of the scanned item has two degrees of freedom: shape and material. These two dimensions can be described as traits of the scanned item.

Using term trait seems quite adequate since some programming languages, such as Scala, implement the concept of trait (or mixin), which perfectly suits to modeling multidimensional objects.

The diagram in Figure 3 depicts the object model of the sample. Each of the three items has different combination of the "traits".
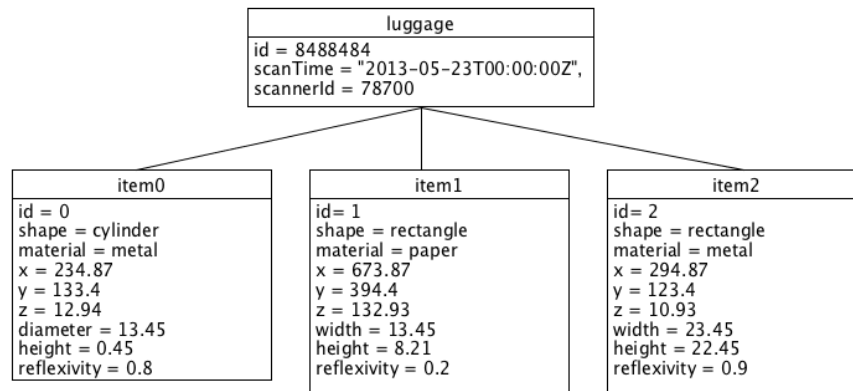
Figure 3: The object diagram of the objects in the sample

Modeling Multidimensional Data Without Traits

Modeling multidimensional objects in languages, which do not include traits or similar concepts, may become rather difficult and the resulting model may not accurately represent reality. For example in Java, which has no equivalent to traits, we have to implement the item's dimensions by composition (Figure 4). It follows that for each dimension there will be one property in the `Item` class (material, shape).

Considering such a no-trait model, a new item can be created as follows:

```
Item item = new Item(10.1, 20.4, 3.1, new Metal(), new Cylinder(0.3, 10.32));
```

However, such a composition comes with some serious design flaws. In Java, if one needs to know what an object is, (s)he uses the `instanceof` operator. For example, if an instance of `Item` is tested whether it is `Thing` or `Material` by this operator, the result will always be true by definition, since according to the model, every item is an instance of `Thing`, `Shape` and `Metal`, unless it is null. On the contrary, testing the item for being `Baggage` will always be false.
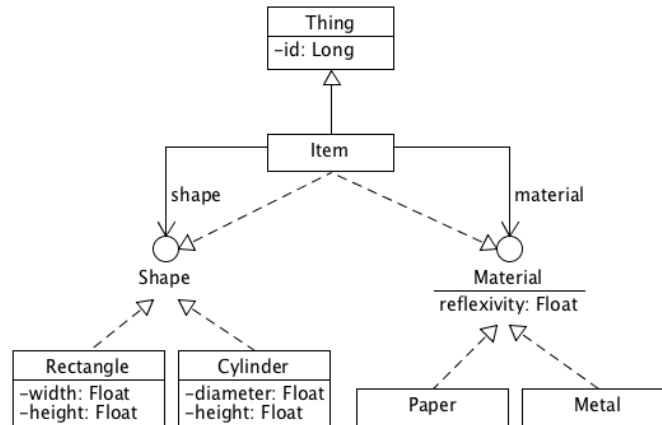
Figure 4: This class diagram shows the trait-less (Java) model of a scanned thing

```
boolean isThing = item instanceof Thing;        // always true
boolean isMaterial = item instanceof Material;  // always true
boolean isBaggage = item instanceof Baggage;    // always false
```

Clearly, such a trivial type examination is useless, since it does not reveal any additional information about the item's characteristics. A more interesting use of the "what-it-is" `instanceof` operator would be the inspection of the concrete "trait" of the item; in other words, whether the item is metal or not, for example. Such a condition could be expressed as follows:

```
boolean isMetal = item instanceof Metal; // always false, even if the item is metal
```

Unfortunately, this statement will be always false, since the type system would have to consult somehow the item's state to determine the "real" type of the item.

Here, the root of the problem is the model itself; the item's qualitative (type) attributes are included not only in the item itself, but also in the wrapped instances. It looks as if the item suffered from some a kind of schizophrenia, since its type information is scattered across three instances: the item itself and the wrapped material and shape objects.

Actually, this is just one of several problems arising when models use delegation or composition (e.g. in the following design patterns: composite, decorator, adapter, state, strategy, proxy, bridge). The problem of object schizophrenia in general is discussed (Sekharaiah & Ram, 2002; Herrmann, 2010).

To conclude, 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*.

Figure 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.

Modeling Multidimensional Data With Traits

Traits may be used to model dimensions of an object. Typically, each dimension (e.g. material) is represented by one abstract trait (e.g. `Material`). Such an abstract trait is the base for other concrete traits (`Metal`, `Paper`) corresponding to the "values" in the material dimension (Figure 6).

The concept of traits is dealt with in greater detail in (Groovy, 2016; Scala, n.d.).

In contrast to the trait-less model, here the `Thing` type and the two dimensions are separated. The composition is deferred until the moment of creation of a new item.
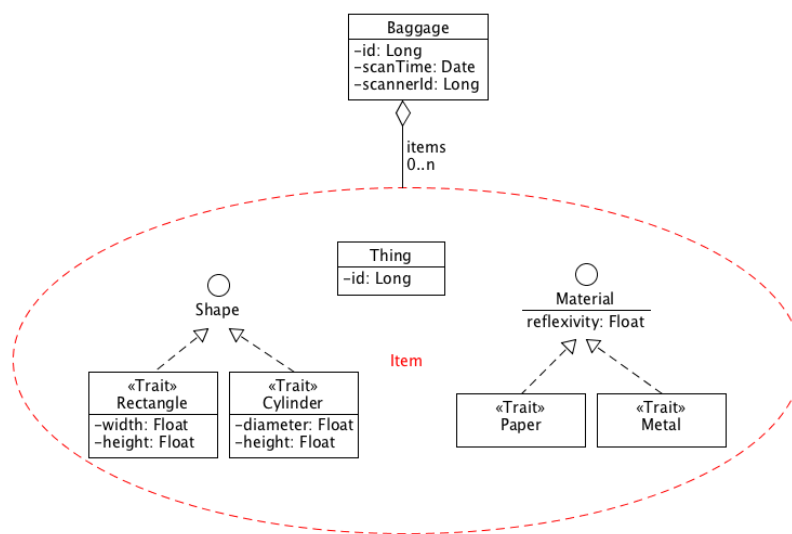

Figure 6: UML schema using traits to model a scanned thing

The following `Scala` code shows how such a creation with the deferred composition may look:

```scala
val item = new Thing with Metal with Cylinder {
  val x = 10.1
  val y = 20.4
  val z = 3.1
  val height = 0.3
```

```scala
    val radius = 10.32
  }
```

Then the item can be tested with `isInstanceOf`, which is analogous to Java's `instanceof`:

```scala
  val isMetal = item.isInstanceOf[Metal]         // true
  val isCylinder = item.isInstanceOf[Cylinder]   // true
```

Now, it is possible to determine the exact type, i.e. what the object exactly is. The reason is that the type is no longer scattered among more instances but is carried solely by the item.

Furthermore, one can also use composite types to test instances:

```scala
  val isMetalCylinder = item.isInstanceOf[Metal with Cylinder] // true or false
```

The absence of the type schizophrenia in models is the key prerequisite for a type-safe mapping between two distinct multidimensional domain models, as shown later.

Modeling Multidimensional Data With Static Traits in Scala

Before going further, we should take a closer look how a new multidimensional instance is actually created from external data in Scala. The concept of static traits seems to fit well with this multidimensional problem, since they can model the individual dimensions.

Static traits are applied to target types at compile-time, which contributes to the robustness of the code. The compiler checks, among other things, whether the resulting composite classes are complete (i.e. all abstract members are implemented) and that the used traits are applied to correct types.

When instantiating a multidimensional type, we must specify a list of traits used for the instantiation by selecting one concrete trait (e.g. `Paper`, `Rectangle`) for each dimension trait (`Material`, `Shape`) in accordance with the input data. This list of traits represents a point in the multidimensional space given by the dimensions and the traits in the list corresponding to the point's coordinates.

There is an important limitation, however, stemming from Scala's strong static type system: traits can be specified only as part of the class declaration. This fact actually rules out any step-by-step imperative way to build the list of traits from the input data (i.e. the builder pattern). Instead, the initialization procedure must handle all possible combinations of concrete dimension traits. The problem is illustrated in the following example.

Let us assume that the external data for the item is stored in a dictionary-like object `event`. In order to create a new item then for every (`material, shape`) combination there must be a dedicated test in the creation code, as shown in *Listing 2*.

Listing 2: Initialization of the common properties in Scala

```scala
val x = event.get("thing").get("x")
val y = event.get("thing").get("y")
val z = event.get("thing").get("z")
val item = if (event.containsKey("metal") && event.containsKey("cylinder")) {
  new Thing(x, y, z) with Metal with Cylinder {
    val height = event.get("cylinder").get("height")
    val radius = event.get("cylinder").get("radius")
  }
} else if (event.containsKey("metal") && event.containsKey("rectangle")) {
  new Thing(x, y, z) with Metal with Cylinder {
    val width = event.get("rectangle").get("width")
    val height = event.get("rectangle").get("height")
  }
} else if (event.containsKey("paper") && event.containsKey("cylinder")) {
  new Thing(x, y, z) with Metal with Cylinder {
    val height = event.get("cylinder").get("height")
    val radius = event.get("cylinder").get("radius")
  }
} else if (event.containsKey("paper") && event.containsKey("rectangle")) {
  new Thing(x, y, z) with Metal with Cylinder {
    val width = event.get("rectangle").get("width")
    val height = event.get("rectangle").get("height")
  }
}
```

It is obvious that such an instantiation is unsustainable since the number of lines (boiler-plate) grows exponentially with the number of dimensions. Unfortunately, there is currently no practical solution in Scala to cope with this problem.

Let us have a closer look at this problem. It is desirable that every item provides information about what it is through the type system only and not from the item's state. In other words this information must be encoded in the class of which the item is an instance. It follows that there must be as many classes in the system as the number of all possible forms which an item can assume.

Next, there are only two ways to define a class. Either it is compiled from its definition in the source code, or it is generated. Application developers normally use the former way only, while the latter is there for frameworks.

If we momentarily forget the possibility to generate classes then the source code must inevitably contain all possible composite class declarations allowed by the domain model. Such explicit declarations may assume basically two forms.

The concrete one (using Scala notation):

```scala
class ThingPaperRectangle extends Thing with Paper with Rectangle
```

or an anonymous one, which is part of the initialization statement:

```
new Thing with Paper with Rectangle
```

In any case, for all coordinates in the multidimensional space there is an exactly one distinct composite class declaration.

On one hand we need composite classes since their instances carry complete type information about themselves (what they are). On the other hand, using the standard means of statically typed languages leads to the excessive amount of boilerplate and large number of classes.

Solutions to the Exponential Growth

The exponential growth of code results from the inability to apply the concrete dimension traits in a step-by-step manner analogous to how instances are created when using the builder pattern (Gamma et al., 1994). Before creating an instance, the builder must be configured in several steps according to the context and the input data. This configuration directly or indirectly specifies the parts from which the builder will compose the instance.

Such a step-by-step approach effectively eliminates the exponential growth, because the number of statements is proportional to the number of dimensions (see Listing 4).

There are basically two types of such a multidimensional builder. A builder of the first type generates a composite class according to the dimension configuration. The resulting object is then instantiated from this generated class. On the contrary, a builder of the second type instantiates a preliminary instance from the common base class first, and this instance is then decorated by instances of the selected concrete dimension traits.

While the first approach, which generates a composite class, would require that the developer use some rather exotic low-level tools for byte-code generation, the second approach can be easily used with the help of a suitable dynamic language. Therefore, in this paragraph we will examine this second approach, based on the application of dynamic traits on an existing instance.

Modeling Multidimensional Data With Dynamic Traits in Groovy

Groovy is a dynamic language, which also supports some optional static language features, like a compile-time type check. It also contains traits that can be applied to both types and instances. To solve the exponential growth problem we will try to rewrite the previous code by means of Groovy's dynamic traits in a builder-like manner.

The common data of the item may be represented by trait `Thing`, which contains the coordinates (`x, y, z`) of the item in the luggage. The code in Listing 3 creates the item object implementing the `Thing` trait. The item object is then initialized from the input data.

Listing 3: Initialization of the common properties in Groovy

```groovy
def item = new Object() as Thing

Map thingData = event.get("thing")
item.x = thingData.get("x")
item.y = thingData.get("y")
item.z = thingData.get("z")
```

Now we can use the dynamic traits and extend the item step-by-step by the right traits. We use the `withTraits` method invoked on the item object to extend it by the selected trait. The selected trait is immediately initialized from the input data (Listing 4).

Listing 4: A step-by-step construction of item

```groovy
// Shape dimension
if (event.containsKey("cylinder")) {
    item = item.withTraits(Cylinder)
    Map cylinderData = event.get("cylinder")
    item.height = cylinderData.get("height")
    item.radius = cylinderData.get("radius")
} else if (event.containsKey("rectangle")) {
    item = item.withTraits(Rectangle)
    Map rectangleData = event.get("rectangle")
    item.height = rectangleData.get("height")
    item.width = rectangleData.get("width")
}

// Material dimension
if (event.containsKey("metal")) {
    item = item.withTraits(Metal)
} else if (event.containsKey("paper")) {
    item = item.withTraits(Paper)
}
```

It follows from the above that using dynamic traits and the step-by-step extensions resolves the problem of the exponential growth. The number of conditions is proportional to the number of dimensions and not to the number of all possible combinations.

The resulting object also carries all type information about itself. The `instanceof` operator can test it, as demonstrated in Listing 5.

Listing 5: Checking the type of the item

```groovy
def isThing = item instanceof Thing
def isRectangle = item instanceof Rectangle
def isPaper = item instanceof Paper

if (isThing && isRectangle && isPaper)
    println(/${item.x}, ${item.y}, ${item.z}, ${item.width}, ${item.height}/)
```

Although this solution, which is based on using the dynamic traits, seems to be perfect, because it allows us to capture the multidimensional characteristics of data by traits, preserves the types and prevents the code from exponential growth, there are still a couple of serious problems 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 select 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.

Since the above-mentioned responsibilities are solved on the application level and not on the platform level, the dynamic traits approach tends to develop unmanageable code.

Modeling Multidimensional Data By Generated Classes w. Run-Time Checks

There still remains one possibility how to instantiate an instance of a multidimensional entity: to generate the composite class at run-time. Class generation is usually chosen as a last resort to solve problems in applications. It is rather an extreme low-level technique requiring experienced developers and which should be used mostly in frameworks or platforms.

Let us assume that there is a framework whereby it is possible to create a class builder, which can be configured to build a class composed from selected traits. The resulting class is then used to create the final instance. In contrast to the dynamic trait approach, which creates a preliminary instance on which it then applies the selected traits, the class generating approach first creates the class composed of the selected traits, from which the final instance is created.

The following code shows how such a class builder could be created.

```scala
val classBuilder = new ClassBuilder[Thing with Material with Shape]()
```

The type parameter of the builder's constructor specifies the common type for all possible combinations of traits, which is `Thing with Material with Shape`. This composite type will also be type of the instance created from the generated class.

The builder's `addTrait` method serves to register a selected concrete trait. The method also accepts a function for the initialization of the trait when the resulting object is instantiated from the generated class.

The `Thing` trait is present in every item, so the builder is always configured to use `Thing` along with the trait initialization function.

```scala
classBuilder.addTrait[Thing](thing => {
  thing.x = event.get("thing").get("x")
  thing.y = event.get("thing").get("y")
  thing.z = event.get("thing").get("z")
})
```

Next, the procedure handles the `Material` dimension. According to the existence of `metal` or `paper` keys in the input data, the builder is configured for the corresponding trait.

```scala
if (event.containsKey("metal")) {
  classBuilder.addTrait[Metal]
} else if (event.containsKey("paper")) {
  classBuilder.addTrait[Paper]
}
```

The `Shape` dimension is handled similarly, except the additional trait initialization function passed to `addTrait`, by means of which the trait's properties will be initialized from the input data upon the instantiation.

```scala
if (event.containsKey("rectangle")) {
  classBuilder.addTrait[Rectangle](rect => {
    rect.width = event.get("rectangle").get("width")
    rect.height = event.get("rectangle").get("height")
  })
} else if (event.containsKey("cylinder")) {
  classBuilder.addTrait[Cylinder](cyl => {
    cyl.height = event.get("cylinder").get("height")
    cyl.radius = event.get("cylinder").get("radius")
  })
}
```

Now, it is possible to build the class and use it to create the new item instance.

```scala
val itemClass: Class[Thing with Material with Shape] = classBuilder.buildClass
val item: Thing with Material with Shape = item.newInstance
```

All problems associated with the dynamic traits mentioned in the previous paragraph might be solved by this fictional framework at run-time, but not at compile-time, because of the imperative character of the configuration. Thus the conclusion is that:

The generated classes approach would be a progress in modeling multidimensional objects in comparison to the dynamic traits; so far it is the best technique.

Nevertheless, it would be nice if all those necessary checks were done at compile-time.

Modeling Multidimensional Data By Generated Classes w. Compile-Time Checks

In this passage we will attempt to come up with an improvement of the previous class generator, which would allow moving the run-time consistency checks of the generated class to compile-time.

The previous class builder is very constrained in terms of static consistency checks primarily because there is no static model or schema for the generated classes. The only information available at compile-time is the type of the resulting instance, specified as the type parameter of the builder's constructor:

```scala
Thing with Material with Shape
```

This type is actually the *lowest upper type bound* of all possible trait combinations (LUB). From the LUB the compiler can infer the abstract dimensions, however, it cannot infer the concrete traits constituting these dimensions. The choice, which concrete traits come into consideration when building a class, is the task for the developer, not the compiler. It follows that the developer must pass this information to the compiler declaratively so as to allow a compile-time verification of multidimensional class models.

It follows that we have to introduce some declarative construct to allow a specification of multidimensional structures. On a statically typed platform such as Scala, the most natural way is to express the multidimensional structure by special type.

The principal idea is to substitute the abstract dimension traits in the type parameter in the builder constructor with a list of possible concrete traits. The traits in the list will be delimited by or to emphasize the mutually exclusive character of the traits in the list. (The or in the type expression would have to be introduced to Scala somehow).

```scala
Thing with (Metal or Paper) with (Rectangle or Cylinder)
```

The previous type could be treated as a logical formula, which can also be expressed in the disjunctive normal form shown in Listing 6.

Listing 6: Type expression's disjunctive normal form

```scala
(Thing with Metal with Rectangle) or
(Thing with Metal with Cylinder) or
(Thing with Paper with Rectangle) or
(Thing with Paper with Cylinder)
```

Such a type actually determines all possible forms of the generated composite class. Since this information is encoded in a type, it is also available to the compiler, which can use to perform the necessary consistency checks.

So, the statement in Listing 7 would create the new builder.

Listing 7: Instantiating a class builder performing static type checks. The type argument of the constructor specifies the model.

```scala
val classBuilder = new ClassBuilder[Thing with (Metal or Paper) with (Rectangle or
Cylinder)]({
    case rect: Rectangle =>
      rect.width = event.get("rectangle").get("width")
      rect.height = event.get("rectangle").get("height")
    case cyl: Cylinder =>
      cyl.height = event.get("cylinder").get("height")
      cyl.radius = event.get("cylinder").get("radius")
  })
```

For each trait in the model the builder maintains a trait factory creating an instance of the trait's stub object. Whenever the factory creates a new stub, the trait initializer passed as the optional class builder's constructor argument may additionally initialize the stub. The new stub might be initialized by one `case` block in the partial function (Scala Documentation, n.d., Scala School, n.d.) implementing the trait initializer. This would especially be useful if the new trait instance is initialized by some external data, as shown in Listing 7.

Regarding the model, it is assumed that there would be a compiler extension that could handle this special type expression and could also build a model, from which all possible trait compositions could be derived.

Once the compiler extension knows all trait compositions, it can take one by one and verify whether all traits in a given composition satisfy the rules, such as completeness, unambiguity etc.

The model would serve primarily to check the consistency of all trait compositions. There must be another concept, however, through which the developer can determine, which trait composition will be used to compose the generated class.

This concept is called class builder strategy and its primary task is to select the right trait composition from the model. A new instance of such a strategy could be obtained from the class builder instance as follows:

```scala
val classBuilderStrategy = new HintClassBuilderStrategy()
```

The `HintClassBuilderStrategy` strategy contains `hintTrait` method through which the developer specifies hints to the strategy, which traits must be in the resulting trait composition.

It is not necessary to provide a hint for the `Thing` traits, since according to the model it must be present always by default.

The trait hints for the material dimension can be handled in the following way:

```scala
if (event.containsKey("metal")) {
  classBuilderStrategy.hintTrait[Metal]
} else if (event.containsKey("paper")) {
  classBuilderStrategy.hintTrait[Paper]
}
```

And the shape dimension can be handled similarly:

```scala
if (event.containsKey("rectangle")) {
  classBuilderStrategy.hintTrait[Rectangle]
} else if (event.containsKey("cylinder")) {
  classBuilderStrategy.hintTrait[Cylinder]
}
```

Calling `newInstance` on the class builder and passing the configured strategy create the resulting instance.

```scala
val item: Thing with Material with Shape =
classBuilder.newInstance(classBuilderStrategy)
```

The `item`'s type is the LUB of the multidimensional model, however, its concrete type can be determined by `isInstanceOf`

```scala
val isPaperRectangle = item.isInstanceOf[Paper with Rectangle]
```

or by the `match` block

```scala
item match {
  case pr: Paper with Rectangle =>
    ...
  case mc: Metal with Cylinder =>
    ...
}
```

This analysis indicates that all run-time checks performed by the previous version of the class builder could be theoretically performed at compile-time in compiler extension. Such a statically typed class builder would give developers a powerful tool for type-safe processing of multidimensional or non-uniform objects.

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 (characteristics, type) of the object. We cannot determine from the top object's type that it is a rectangular paper, for instance. 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, the one 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.

- The identity of the composite object is scattered across the object and its components, which results in object schizophrenia.

- Traits are a natural concept for modeling multidimensional objects, which helps preserve full object's type information and avoid type schizophrenia.

- Modeling multidimensional objects by means of static traits (Scala) leads to the exponential explosion of code making the static traits practically unusable.

- One solution to the exponential growth could be to use a dynamic language such as Groovy with a capability to extend object at run-time. This approach however also suffers from a couple of serious issues as shown earlier in this chapter.

- Another solution could be to create an instance from a generated class composed of the selected traits. However, without a new declarative language construct for specifying multidimensional models such a class builder could perform the important consistency checks at run-time only.

- Should the type system be extended in such a way that it allows declaring optional types (type-union-like), the developer could specify all possible trait compositions, i.e. the model, by means of a type expression reminiscent of Boolean expressions. Such a type expression could be processed by the compiler extension and provide all necessary checks at compile-time.

3.1.2   Mapping Multidimensional Data

This section deals with the issues encountered when mapping multidimensional structures from one domain onto structures from another domain. It is assumed that the natural relationship between the target and source type is "is-a"; in other words, the target domain object preserves the identity and type of the source domain object. The target object also may or may not change the behavior of the source object.

Several approaches are examined and the findings are discussed in the summary of this section.

New Context Domain For Protodata

Let us turn the attention back to the airport scanner protodata and the effort to find some use for it.

Let us assume that someone comes up with an idea to build an application performing some analytics of the luggage contents and publishing the insights via a web portal to customers, which could be for example the statistics bureau, economic chambers or customs.

The first analytics performed by the application will be the publishing of the aggregate amount of cash detected in luggage per some time period.

Such a task requires that the application be able to recognize coins and banknotes among luggage items. Considering the scanner protodata contains information about the shape and material only, it seems inevitable to use other source to detect reliably coins and banknotes. Such a source might be a database of international currencies containing comprehensive information about current and historical world currencies including physical properties.

Note: There is a limitation in this scenario, since all U.S. banknotes have the same dimensions. The database could not recognize individual dollar banknotes just on the basis of their size. For the sake of simplicity, it is assumed that the banknotes may be distinguished through their dimensions only. The domain model of the new context is depicted in Figure 7.
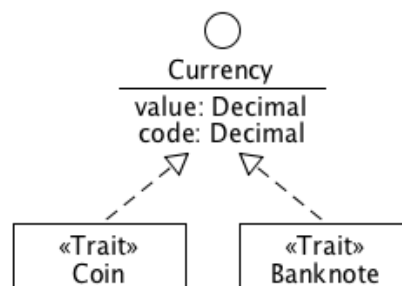
Figure 7: The new context domain of currencies

The basic logic of the currency flow analysis is very simple. It scans a segment of the protodata corresponding to the selected time period and it attempts to convert each luggage item into a currency. Such a set of currency objects is then aggregated into a statistical report and published on the portal. The relation between the currency and luggage items domain is sketched in Figure 8.

Each concrete type in the target domain is mapped to a subset of the source domain concrete types. Moreover, this mapping is *orthogonal* since all these subsets do not overlap with one another.

The only missing part is the construction of currency objects from the source objects, i.e. from appropriate items and currency database records. In this case study, this task is called *mapping* and is analyzed in the next paragraph.
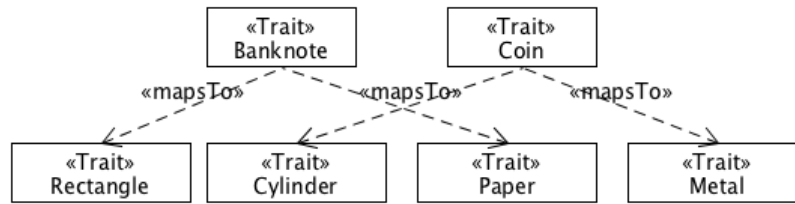
Figure 8: Mapping context domain (currencies) to proto-domain diagram (scanned items)

Mapping Target Objects To Source Ones

The purpose of the mapping is to bind objects from the target domain directly to the objects from the source domain in order to avoid any intermediary processes normalizing and transforming the source domains objects into the target domain ones. Such processes often discard some source data, which could potentially be utilized in subsequent transformations.

The goal is to construct a lossless mapping. In other words, the target objects preserve both data and behavior of the source objects.

A target object constructed by binding it directly to its source objects allows tracking its origins, which is as desirable property in the case when the target object becomes a source in another mapping. Such a secondary mapping may exploit the information about the origin of the source (then-target) object to perform a finer-grained binding.

In the following paragraphs we will attempt to design such a lossless mapping procedure.

Mapping Currencies To Items by Properties

Listing 8 shows an example of a low-level lossless mapping in Scala. The source and target objects are plain maps, whose values can be other maps or scalar values. The result of the mapping procedure is a map representing a scanned currency, whose single property `isExported` combines properties from the item and the currency record in the database. The result also wraps the source objects (item, exemplar) so as not to loose the track of its origin.

The method first checks whether the `shape` and `material` dimension properties hold rectangle and paper data. If that is the case, the method retrieves the width, height and color from the two components. Then the currency database is queried for a banknote having the corresponding properties. If a banknote exemplar is found, a new map representing the banknote is created.

Although this mapping is lossless, it suffers from several flaws, such as:

- It is too low-level; domain objects are represented as maps, not as domain types. It is prone to type mismatch errors.

Listing 8: Making a currency map from an item map

```scala
def makeCurrency(item: Map[String, Object]): Option[Map[String, Object]] = {
  val shape: Map[String, Object] = item.get("shape")
  val material: Map[String, Object] = item.get("material")

  if (shape != null && "rectangle" == shape.get("type") ||
      material != null && "paper" == material.get("type")) {

    val w = shape.get("width");
    val h = shape.get("height");
    val c = material.get("color");

    currencyDb.findBanknote(w, h, c) match {
      case None => None
      case Some(banknote) =>
        val isExported = banknote.get("issuingCountry") == item.
          get("luggage").asInstanceOf[Map[String, Object]].
          get("scanner").asInstanceOf[Map[String, Object]].
          get("location").asInstanceOf[Map[String, Object]].
          get("country")
        val banknote = Map("type" -> "scanned-banknote",
                           "isExported" -> isExported,
                           "item" -> item,
                           "currency" -> banknote)
        Some(banknote)
    }
  } else ... // a similar code for a coin
}
```

- There is no behavior on the domain objects (maps). Target objects should be able to inherit some behavior from the source objects. For example, the `Shape` trait could define a method calculating the area (or surface) of the shape. It makes good sense for banknotes and coins as the target objects to inherit this method.

- There must be some convention on how the source data are embedded into the target objects. Such a convention makes the mapping subsystem very proprietary.

- It is too verbose, too much boilerplate.

The absence of types and behavior is evidently the most annoying issue of this property-based approach. Let us try therefore to evaluate an approach using properties and types of the domain objects.

Mapping Currencies To Items by Properties and Static Traits

The mapping procedure in this paragraph combines the properties of the domain objects as well as their types. The item contains two properties, `shape` and `material`, representing the two independent dimensions of the item. The mapping procedure first reads the wrapped objects held in these two dimension properties and examines their concrete types. In other

words, in order to find out what the item really is we have to look into its properties and determine the types of their values (i.e. a symptom of type schizophrenia).

The item, its dimensions (i.e. material, shape) and the concrete "values" in these dimensions (i.e. paper, rectangle etc.) are represented by Scala static traits.

Let us assume first, that the multidimensional character of the item is modeled by means of composition. This approach would have to be used on platforms without dynamic traits (such as Scala) because of the problems with the exponential growth of code described previously.

Listing 9: Item modeled by composition

```scala
trait Item {
  def shape: Shape
  def material: Material
  ...
}
```

The source domain is described in the previous section, so here we focus on the description of the currency database types and the target domain types.

The currency database has a simple model consisting of two types of currency records: `Banknote` and `Coin`, both having the same ancestor `Currency` (Listing 10).

Listing 10: Currency and its subtypes

```scala
trait Currency {
  def currencyValue: BigDecimal
  def currencyCode: String
  def issuingCountry: String
}
trait Banknote extends Currency
trait Coin extends Currency
```

The `Currency` trait consists of a couple of basic properties, while the other traits may carry some specific properties. (They are omitted since they are not important for the purpose of this example.)

The target types mimic the structure of the currency database domain. The `ScannedCurrency` trait represents a currency found in a luggage and its single property `isExported` combines data from the two source domains: a currency is considered *exported* if the currency's issuing country is the same as the country where the item was scanned (Listing 11).

Scala's self-type annotation `=>` in `ScannedCurrency` constrains the application of this trait only to the compositions containing `Currency` and `Item` traits. Since any object containing `ScannedCurrency` must also contain the two traits it is possible to access the `Currency` and `Item` members from the body of `ScannedCurrency`.

Listing 11: Using self-type annotations in the target domain traits to refer the source traits

```scala
trait ScannedCurrency {
  this: Currency with Item =>

  val isExported: Boolean = issuingCountry == luggage.scanner.location.country
}

trait ScannedBanknote extends ScannedCurrency {
  this: Banknote with Item =>
}

trait ScannedCoin extends ScannedCurrency {
  this: Coin with Item =>
}
```

Note: The self-type annotation is key in the so-called Cake Pattern in Scala. This pattern and the self-type annotation are explained in detail in (Rollins, 2014).

ScannedBanknote constrains its application even more by narrowing the self-annotation to the objects with the Banknote and Item traits. Similarly, ScannedCoin narrows its use to the contexts containing Coin and Item.

The self-type constrains in ScannedBanknote and ScannedCoin could be even narrower if we could replace Item with Paper with Rectangle, resp. Metal with Cylinder. However, it is not possible, because of the type schizophrenia of the item; i.e. the concrete material and shape type information is not part of the item but of the item's properties.

Listing 12: Making a currency object from an item object

```scala
def makeCurrency(item: Item): Option[ScannedCurrency] = {
  if (item.shape.isInstanceOf[Rectangle] && item.material.isInstanceOf[Paper]) {
    val rect = item.shape.isInstanceOf[Rectangle]
    val paper = item.material.isInstanceOf[Paper]
    val w = rect.width;
    val h = rect.height;
    val c = paper.color;

    currencyDb.findBanknote(w, h, c) match {
      case None => None
      case Some(banknote) =>
        val scannedBanknote = new ScannedBanknote with Banknote with Item {
          val itemDlg = item
          val banknoteDlg = banknote
          // Delegating methods ...
        }
        Some(scannedBanknote)
    }

  } else ... // a similar code for a coin
}
```

Now, let us turn our attention to the mapping method `makeCurrency` using the above-mentioned types.

The method `makeCurrency` in Listing 12 must first check the types of the two dimensions `shape` and `material` included in the item as its properties. Then the values are type-casted to the concrete types to retrieve the width, height of the rectangle and the paper color.

Then the database is asked to find the corresponding banknote. If it is found, a new instance of `ScannedBanknote` is created, which implements both `Banknote` and `Item` traits by delegation.

At first sight, the code looks much tidier than the property-based one. However, there are still some issues:

- `ScannedBanknote` might be instantiated with a wrong (i.e. not banknote-like) item in case the developer makes a mistake, e.g. in the condition. The likelihood of such a mistake could be significantly reduced if some responsibilities were transferred to the compiler. For instance, if the self-type of `ScannedBanknote` were more specific about the context, e.g. `Banknote with Paper with Rectangle`, the compiler could verify that the type of a given item could be used as the context of `ScannedBanknote`. It is not possible here, however, because of the type schizophrenia of the item; the compiler does not know the types of the item's properties `shape` and `material`.

- Another issue related to delegation concerns "forgetting": on each consecutive mapping the item instance "forgets" some behavior because the target object is not obliged to implement all interfaces implemented by the source object. The wrapped item instance gradually sinks one delegation level down with each mapping until it becomes virtually unreachable.

These two findings can be formulated more generally:

- A multidimensional source object modeled by **composition** makes subsequent compositions prone to type inconsistencies.

- The construction of target objects by means of **delegations** tends to forget behavior and type.

Mapping Currencies To Items by Static Traits Only

In this passage we will examine a purely static type-based approach to mapping target objects on the source ones by means of static traits.

Since one of the problems described in the previous paragraph was caused by composition, i.e. by the dimensions wrapped in the item object, in this example we will try to model the multidimensionality of items by Scala static traits (no type schizophrenia). It follows that there will be no properties in `Item` representing the two dimensions. Instead, the dimensions become part of the item's type expressed by means of traits.

Note: Let us temporarily forget the problem with the exponential growth of code linked to using statically specified traits as described in the previous section.

Since now the shape and material dimension are part of the item's type, we can refine the self-type in the `ScannedBanknote` so as to refer to `Banknote with Rectangle with Paper` instead of `Banknote with Item` (Listing 13).

Listing 13: ScannedBanknote with a more specific self-type

```scala
trait ScannedBanknote extends ScannedCurrency {
  this: Banknote with Rectangle with Paper =>
}
```

This refinement actually solves the problem of combining `ScannedBanknote` with incompatible item; the type system ensures that `ScannedBanknote` can be instantiated only with an item implementing `Rectangle` and `Paper`.

Thanks to the presence of dimension types in the item's type the mapping method can perform the test for paper rectangles directly on the item. Instead of using `isInstanceOf[Rectangle with Paper]` we use Scala's `match` block, which is more suitable for this purpose (Listing 14).

Listing 14: Making a currency from a non-composite item

```scala
def makeCurrency(item: Item): Option[ScannerCurrency] = {
  item match {
    case papRec: Rectangle with Paper =>
      currencyDb.findBanknote(papRec.width, papRec.height, papRec.color) match {
        case None => None
        case Some(banknote) =>
          val scannedBanknote =
            new ScannedBanknote with Banknote with Rectangle with Paper {
              val paperRectDlg = papRec
              val banknoteDlg = banknote
              // Delegating methods ...
            }
          Some(scannedBanknote)
      }
    case _ => None
  }
}
```

Unfortunately, the new instance cannot avoid the delegation. Therefore, ignoring the problem of the exponential growth of code, the conclusion is that:

- The composition issue is over, `ScannedBanknote` may be accompanied by paper rectangles only

- The delegation issue still persists, i.e. the behavior and type degrade on every successive mapping

Mapping Currencies To Items by Dynamic Traits

This passage examines if the above-mentioned problems disappear when using dynamic traits (i.e. Groovy).

The semantics of the `makeCurrency` method is the same as in the previous paragraphs except the return type, which is not optional (Listing 15). The method simply returns `null` in case it cannot construct a currency object.

Listing 15: Using dynamic traits to make a currency from an item

```groovy
public Currency makeCurrency(Item item) {
    if ((item instanceof Paper) && (item instanceof Rectangle)) {
        Banknote banknote = currencyDb.findBanknote(item.width, item.height,
          item.color)

        if (banknote != null) {
            ScannedBanknote scannedBanknote = item.withTraits(ScannedBanknote, Banknote)
            // we cannot avoid the copying of the banknote's state to scannedBanknote
            scannedBanknote.adoptBanknote(banknote)
            return scannedBanknote;
        } else {
            return null;
        }
    } else ... // a similar code for a coin
}
```

As shown in the previous section, on a platform with dynamic traits it is possible to extend objects in such a way that the extending objects can completely preserve the type information carried by the extended objects. It allows us to determine what the object is by means of `instanceof`.

Thus, the `makeCurrency` method determines whether the item is a paper rectangle by two invocations of `instanceof`. Then the method asks the currency database if such a banknote is registered. If that is the case, it uses `withTraits` to create a new proxy object from the item object. The new proxy object has all the item's traits plus `ScannedBanknote` and `Banknote`.

Unfortunately, using `withTraits` helped us only in part. We avoided copying the item's state to the new instance, but there is still the `banknote` object, whose state must be copied to the new instance. This is a conceptual problem stemming from the nature of traits in Groovy, which are primarily supposed to extend the target object's behavior, not its state. Although a trait may declare some state (through attributes), such a state is, however, part of the target object. This concept is reflected in the definition of `withTraits`, which is a method invoked on the target object, and which accepts a list of trait types, not trait "instances".

The conclusion is:

- The use of dynamic traits solves the problem with forgetting of the item's type and behavior, because `withTraits` automatically implements all interfaces on the source object.

- The removal of the delegation was helpful in the previous issue, however it brings another problem: the `banknote's` state must be copied to the `scannedBanknote` instance. The `banknote` may theoretically be an instance of a `Banknote` subclass carrying additional data. This extra state will not be copied to `scannedBanknote`, thus not all information may be transferred to the target object when cloning.

- The problem of combining `ScannedBanknote` with a wrong item emerges again, because the weak type system and the dynamic nature of Groovy does not allow checking the inter-trait relationships.

Mapping Currencies To Items by Class Generator

So far no mapping attempt has offered an ideal solution. The Scala showed to be too rigid while Groovy too flexible. This paragraph is dealing with an alternative approach searching for a middle course, which would lead us to a statically checked yet dynamic mapping.

The basic building block in such an approach is the fictional class builder presented in the previous section. This builder is able to build consistent compositions of traits according to the model specified as a special type expression in the builder's constructor. The compiler is supposed to execute the consistency checks of the model so that the builder can rely on the model's consistency at run-time.

In Scala it is possible to define type aliases by means of the so-called type members. To make the code more maintainable and readable, we will use such type aliases instead of the original type expressions throughout the code. The first one is the alias of the item's multi-dimensional model.

```scala
type ItemModel = Thing with (Metal or Paper) with (Rectangle or Cylinder)
```

This model can be rewritten to the equivalent disjunctive form (using the new `or` operator), which illustrates better the four possible compositions of the traits.

Listing 16: The disjunctive normal form of the item's model

```scala
(Thing with Metal with Rectangle) or
(Thing with Metal with Cylinder) or
(Thing with Paper with Rectangle) or
(Thing with Paper with Cylinder)
```

The item builder is created by instantiating `ClassBuilder` and specifying the `ItemModel` type alias as the constructor's type argument and the trait initializer as the other argument.

Listing 17: Using `ClassBuilder` to create an item

```scala
val itemBuilder = new ClassBuilder[ItemModel]({
  case rect: Rectangle =>
    rect.width = event.get("rectangle").get("width")
    rect.height = event.get("rectangle").get("height")
  case cyl: Cylinder =>
    cyl.height = event.get("cylinder").get("height")
    cyl.radius = event.get("cylinder").get("radius")
})
val itemBuilderStrategy = new HintClassBuilderStrategy[ItemModel]()
... // configuring the strategy
val itemRef: Ref[ItemModel] = itemBuilder.newInstanceRef(itemBuilderStrategy)
```

At this stage, for each trait in the model, the builder holds a trait factory, which may be asked to create the trait's stub object.

Then the developer configures the builder strategy, which is used by the builder during the instantiation to consult which trait composition should be selected for the instantiation. The `HintClassBuilderStrategy` strategy allows specifying traits that should be present in the selected trait composition.

The strategy object may actually contain inconsistent hints, but they will never lead to using an inconsistent composition of traits, as only one of the four (consistent) compositions may be used. The hints only help the builder to select the best matching trait composition to the given hints. Without any hint the class builder selects the first composition from the list of all possible compositions.

After the builder selects the valid trait composition in cooperation with the builder strategy, it invokes the corresponding trait factory for each trait in the selected composition.

The builder can be used repeatedly to instantiate various trait compositions.

```scala
val otherItemRef: Ref[ItemModel] = itemBuilder.newInstanceRef(otherItemBuilderStrategy)
```

Since the builder uses always the same trait factories, it becomes very important, whether a factory is stateless or stateful. If a factory is stateless it always creates a new trait stub instance. On the other hand, if a factory is stateful it may implement some instance management such as pooling. In the simplest case it works as a singleton factory, which creates only one instance. Should all trait factories are singleton factories, then the same trait stub objects may appear in different trait line-ups produced by the same builder. The figures Figure 9, and Figure 10 depict the state of the singleton trait factories. The red color indicates that the factory has not yet created the instance.

Such a configuration reminds *object morphing*. The instances returned by the builder with singleton factories only may be seen as different forms of a single protean instance consisting of the trait stub objects held by the singleton factories in the builder.
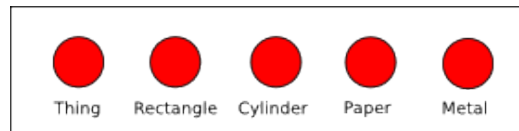
Figure 9: The state of the singleton factories before the first instantiation

Figure 10: The state of the singleton factories after the first instantiation
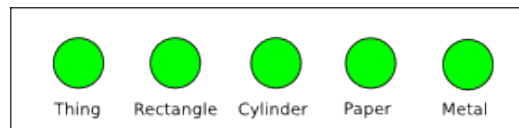
Figure 11: The state of the singleton factories after the second instantiation

After the item builder is configured, it returns the result wrapped in a special `Ref` object.

This object, in contrast to the plain reference returned by `newInstance`, contains a link to the model, from which the instance is created. Using the `Ref` object reference instead of the plain reference allows performing additional compile-time checks when the reference is passed to a method as its argument.

The item instance is retrieved through the `instance` method of the reference object:

```
val item: Thing with Material with Shape = itemRef.instance
```

Now, let us turn our attention to the target domain model. It may be expressed in terms of a one-dimensional model with two concrete traits `ScannedBanknote` and `ScannedCoin` belonging to the `ScannedCurrency` dimension.

```
type ScannedCurrencyModel = ScannedBanknote or ScannedCoin
```

This model generates only two trivial trait compositions and it may be tempting to create an instance of this in a similar way as the item instance, as shown in the previous section.

```
val curBuilder = new ClassBuilder[ScannedCurrencyModel]()
... // the strategy configuration
val curRef = curBuilder.newInstanceRef(curBuilderStrategy)
```

Unfortunately, it will not work. The reason is that the traits `ScannedBanknote` and `ScannedCoin` depend on other traits via their self-type. In other words, the model used in the builder is incomplete.

To make it complete we would have to extend it by the required dependencies:

Listing 18: Currency model completed by the dependencies

```
type ScannedCurrencyModel =
  (ScannedBanknote with Banknote with Rectangle with Paper) or
  (ScannedCoin with Coin with Cylinder with Metal)
```

Now we could make an instance of this model, however, we would have to copy data from outer instances of the selected traits. And this is exactly what we do not want. What we want is to reuse those outer trait instances by the scanned currency builder.

Reusing the item's trait stubs in the currency builder should be as easy as passing the item's reference object to the constructor of the currency builder:

```
def makeCurrency(itemRef: Ref[ItemModel]): Ref[ScannerCurrencyModel] = {
  val curBuilder = new ClassBuilder[ScannedCurrencyModel](itemRef)
  ...
}
```

Now, the compiler has all necessary information about the target model (which is incomplete) and the source model retrieved from the object reference type (i.e. `ItemModel`). The compiler must be able to determine whether the source model can deliver all trait stubs required by the traits in the incomplete target model.

In this case the compilation should fail, since the source model cannot deliver `Banknote` and `Coin` trait stubs; i.e. the traits belonging to the target domain.

The proposed solution to this problem is to complete the model by the dependencies, as above, and to distinguish the target and source traits by means of a special annotation `$`. All traits encompassed by this annotation will be treated as the target traits, which are created by the class builder; i.e. the source reference is not required to deliver the stubs of these traits. On the other hand, the stubs of the other traits must be delivered by the source reference.

Listing 19: Currency model distinguishing the target and source traits by $ annotation

```
type ScannedCurrencyModel =
  ($[ScannedBanknote with Banknote] with Rectangle with Paper) or
  ($[ScannedCoin with Coin] with Cylinder with Metal)
```

The target model effectively shrinks the number of possible input compositions from four to two, because only `Thing with Metal with Cylinder` and `Thing with Paper with Rectangle` matches with the target model.

```
ScannedBanknote with BanknoteLoader with Banknote -> Thing with Metal with Cylinder
ScannedCoin with CoinLoader with Coin -> Thing with Paper with Rectangle
```

It is important to emphasize, that there is no builder strategy used in this case, since the resulting composition is determined by the composition referred by `itemRef`. The currency builder in fact inherits the strategy from the `itemRef`.

The effective joined model may be expressed as follows:

```
(Thing with Metal with Cylinder with Coin with ScannedCoin) or
(Thing with Paper with Rectangle with Banknote with ScannedBanknote)
```

There are two scenarios, however, in which the reference's `instance` method may fail:

- The item reference carries an unmapped composition (i.e. `Thing with Metal with Rectangle` or `Thing with Paper with Cylinder`)

- A currency loader does not find the corresponding currency record in the database.

Both scenarios are considered application exceptions, i.e. are part of the use-case and must be handled by the application.

In order to facilitate handling of such conditions, the reference object provides the `maybeInstance` method, which returns `Some(instance)` in case of success or `None` otherwise.

```
val itemOpt: Option[Thing with Material with Shape] = itemRef.maybeInstance
```

The conclusion is that mapping based on the class builder:

- May ensure consistency of target compositions at compile-time

- May avoid loosing behavior and data in successive mappings

- May reuse more stubs

Summary

- The data objects in the protodata manifest a kind of *multidimensional polymorphism*. This term expresses the fact that the set of all forms, which an object can assume, is equivalent to the Cartesian product of the so-called dimensions. A dimension represents an abstract trait of the object and consists of types implementing this trait.

- Transformation and normalization are lossy processes, thus they remove data, which the application might otherwise use in the future. Next, such a processing may cause delays and put additional burden on the infrastructure. In the course of time the transforming processes tend to become unmaintainable.

- The solution is to map the new context domain directly to the source domains and to compose the target domain objects of the objects from the source domains.

- Such a mapping engine should be domain-agnostic, i.e. its functionality should not be principally driven by the knowledge of the participating domains. In other words, the mapper should be able to map one domain onto another only by means of the underlying language platform, especially the type-system. There more reasons for it:

- **Stability**: The semantics of domain entities is more stable than the structure of individual entities. It follows that if the semantics is expressed by types and not by state, then possible structural changes in the domain model should not affect the mapping schema.

- **Type safety**: The type system verifies the consistency of the mapping rules, i.e. that the types involved in the mapping can be mapped and composed as intended. It can detect missing parts and makes the mapping schema robust against changes in the domain models.

- **Early error discovery**: If a static language is used then the consistency checks and the verifications are carried out at compile-time. It follows that possible errors and inconsistencies in the mapping may be caught early.

- **Clean code**: Since many of responsibilities can be delegated on the underlying platform, the code contains only the necessary stuff.

- Java's type system does not provide sufficient means to express the real character of objects by type. Therefore, designers must resort to the modeling of the object character by state (e.g. delegation) whereby they incorporate type schizophrenia into the model.

- The concept of traits as implemented in Scala or Groovy fits very well to the needs of such a mapping.

- However, using static traits to express multidimensional objects leads to the explosion of class declarations and the code as such. Using dynamic traits solves this problem.

- Neither static nor dynamic traits solve the problem of copying states between source and target instances during the mapping.

- The solution may be a mapper based on the concept of the class builder

3.2   2nd Case Study: Behavioral Aspects Of Multidimensionality

3.2.1   Protean Services

In contrast to the previous case study, this one focuses on behavioral aspects of the adaptive applications development (i.e. protean applications).

Before delving into the analysis, let me introduce the subject by the following story:

A company plans to offer an email service for both its customers and its employees. Although the functionality of the service is mostly the same for both types of users, such as the virus scan, it differs in a couple of details, such as the insertion of the employee's signature into the end of every email and the verification that the customer's account has not expired.

For the eligible users the email service may provide some extending functionality, such as sending fax by email for premium customers.

A number of employees are also the company's customers. Such employees are allowed to use the service under their employee account when they are in the office; otherwise they use the service under their customer account. To allow a continuous usage for such users, the email service is able to switch transparently between the accounts either automatically or manually.

The description suggests that the service's behavior is very contextual. How it actually works depends on several factors, such as the type of the current account (employee, customer), the user's traits (e.g. premium), the message being sent and also time, since the service is able to switch between the accounts in the background.

Additionally, the service's interface is also mutable, since, under certain circumstances, it may provide additional features. This fact puts some special requirements on the service's user interface, which must be able to reflect these mutations. For example, if a user is both an employee and a premium customer the user interface should adapt its look as soon as the service switches in the background the account to the premium customer one so that the user could send faxes by email.

The following sections examine how such a service may be designed on the following platforms:

- With no concept of traits
- With traits
- With the class builder introduced in the previous case study

3.2.2   Modeling the Service Without Traits

Before modeling the service itself, let us begin with the domain model. It consists of two domain entities only: `RegisteredUser` and `Employee`. Since the two entity classes have no common ancestor, it is necessary to add an adapter interface, which will establish the common basis for the service, as shown in the diagram in Figure 12.

The `PremiumUser` interface is a marker interface indicating that a registered user is a premium user; it follows the strategy to use types instead of "is-a" properties.
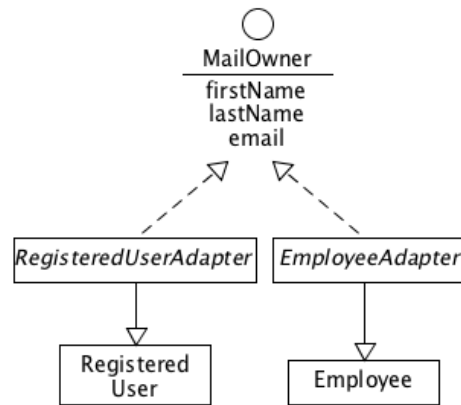
Figure 12: Mail service user model
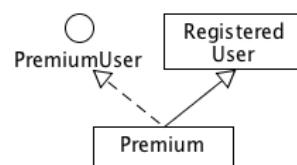
Figure 13: Premium user marker interface

The `MailOwner` interface contains properties having their counterparts in both entity classes. There are further two adapter classes implementing `MailOwner`, for each entity class one.

Now, when the domain entities can be adapted to a common base, it is possible to model the mail service.

The `UserMail` interface declares two methods. Method `sendEmail` is the core method for sending emails. The other method, `validateEmail`, allows the user to validate a message before sending it.
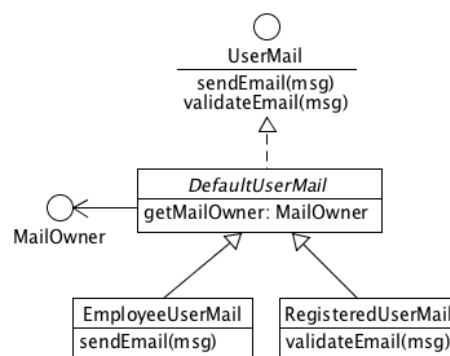
Figure 14: `UserMail` interface and its implementations

The default implementation `DefaultUserMail` works in the context of the current user account, from which it retrieves the field `from` for emails, among others. This user account is held in a reference attribute of `DefaultUserMail`.

The `sendEmail` method in `DefaultUserMail` invokes the `validateEmail` method before the email is actually sent. It allows for subclasses to override the default validation by adding more specific checks.

As mentioned above, the service's behavior reflects the type of the user account. Therefore there are two specializations of `DefaultUserMail` for both account types.

`EmployeeUserMail` overrides method `sendEmail` so as to add the employee's signature at the end of every email.

On the contrary, `RegisteredUserMail` overrides `validateEmail` in order to check if the user's account is not expired.

The service also contains a stack of pluggable email processors forming a processing pipeline. These processors implement `UserMail` interface and are connected through delegation. One of such processors is `VirusDetector`.
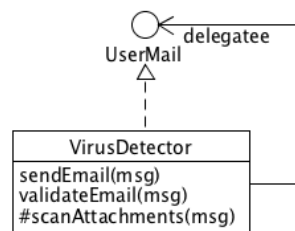
Figure 15: `VirusDetector` class

Here, one important weakness related to the absence of traits is evident. The `VirusDetector` class overrides `sendEmail` as well as `validateEmail`, although it would make sense to override `validateEmail` only. The virus scan in email attachments may be considered a part of the validation performed before sending the mail by `DefaultUserMail`.

Unfortunately, since the virus detector and the wrapped email service (`delegatee`) are separate objects connected by the reference, the invocation of `validateEmail` from within `DefaultUserMail` will always either remain in `DefaultUserMail` or jump to some of its subclasses overriding `validateEmail`. Since `VirusDetector` is not a subclass of `DefaultUserMail`, its implementation of `validateEmail` will never be called from within the wrapped instance. Therefore `validateEmail` must be explicitly invoked from `sendEmail` in `VirusDetector` so that the virus scan could be part of the sending of email. Such a design, however, has some unfortunate consequences: the validations are supposed to be invoked from the `sendEmail` method in the last object in the chain of `UserMail` implementations. This last object might or might not invoke the validations with respect to the settings of the user's preferences. In the case that this last object decides to skip the validations, the `VirusDetector,` which wraps directly or indirectly this object, will ignore this decision, since it has no clue about the decision procedure performed underneath, and will validate the email anyway, possibly throwing a validation exception if it detects some virus, despite the user's wish not to perform any validation.

This complication is just one of the unfortunate consequences of object schizophrenia.

In the following paragraph it is explained that should the platform be equipped with the concept of traits, the processors could be linked by means of the stacking of traits and the above-mentioned problem could be elegantly avoided.

Besides the general email service, there is also an extension offering sending faxes by email. This service add-on is available to premium customers only.
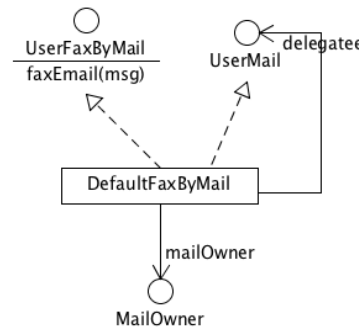
Figure 16: Fax service add-on

The only method in `UserFaxByMail` sends faxes and has the same parameter as `sendEmail`. The default implementation `DefaultFaxByMail` works in the context of the current user accessible through the reference to `MailOwner`. `DefaultFaxByMail` implements `UserMail` by delegation to compose `UserFaxByMail` and `UserMail` interfaces.

On top of the model there is the `AlternatingUserMail` class, whose purpose is to switch transparently between the two accounts if the user has both. `AlternatingUserMail` contains two references to two instances of the mail services and the class is implemented by means of the state pattern (Gamma et al., 1994).
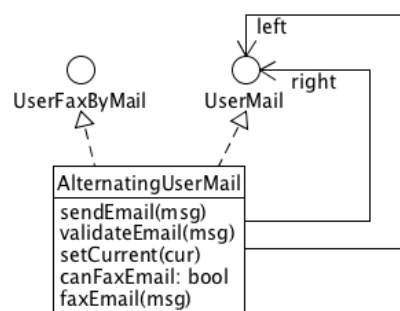
Figure 17: AlternatingUserMail diagram

This class also implements the `UserFaxByMail` interface, which is only a conditional implementation, since unless the current mail service implements `UserFaxByMail`, the `faxEmail` method throws an exception.

Whether it is possible to invoke `faxEmail` may be determined by `canFaxEmail`. The positive test, however, does not guarantee that the invocation of `canFaxEmail` will be pass, since the current mail service in the `AlternatingUserMail` instance might change in the meantime.

Such design is error-prone, because it requires that the topmost wrapper in the fax-by-mail service instance compose `UserMail` and `UserFaxByMail` interfaces, such as `DefaultUserFaxByMail`, in order to indicate the presence of both interfaces. The reason is that `AlternatingUserMail` determines whether the email service instance is `UserFaxByMail` by checking its type. It follows that it may happen that another wrapper wraps the topmost wrapper of such an email service and hides the `UserFaxByMail` type; in such a case the `AlternatingUserMail` will not determine the availability of `UserFaxByMail`.

Additionally, it would be practically impossible for the client to determine the actual user account type without using some reflection tricks or introducing a new `getUserAccount` method in the `UserMail` interface. Both solutions are bad.

The former would be a plain hack, while the latter introduces a backward incompatible change to the general contract declared by the `UserMail` interface. It would force all `UserMail` implementations to add the new method even if it would make no sense for them, since some implementation may not wrap any user account object at all. Such classes would have to return `null` as an indication there is no underlying user account object. Moreover, the return type of the `getUserAccount` method would have to be `Object`, since there is no common ground shared by all potential user accounts; the diversity of user accounts is actually the main presupposition of this case study.

Note: The issue of the addition of `getUserAccount` in the `UserMail` base interface could be in part solved using the default interface method introduced to Java 8. The `getUserAccount` method would be added to `UserMail` as a default method throwing an exception indicating that the method is not supported. Such an extension of `UserMail` would not affect its current implementations, however, there may suddenly appear instances that newly and only formally implement the `UserMail` interface in the application, and are thus usable in the places where they could not be used before. It may lead to an increase of unhandled exceptions thrown from the default method `getUserAccount`. Besides, the default method does not solve the issue of having `Object` as the return type.

The ideal solution to the above problem would be to determine whether the topmost email service (i.e. `AlternatingUserMail`) is an instance of trait `HasUserAccount` with method `getUserAccount`. If that were the case the service could be type-casted to `HasUserAccount` so as to invoke `getUserAccount`. This cannot be, however, achieved by composition, since it does not allow implementing an optional interface, such as `HasUserAccount`.
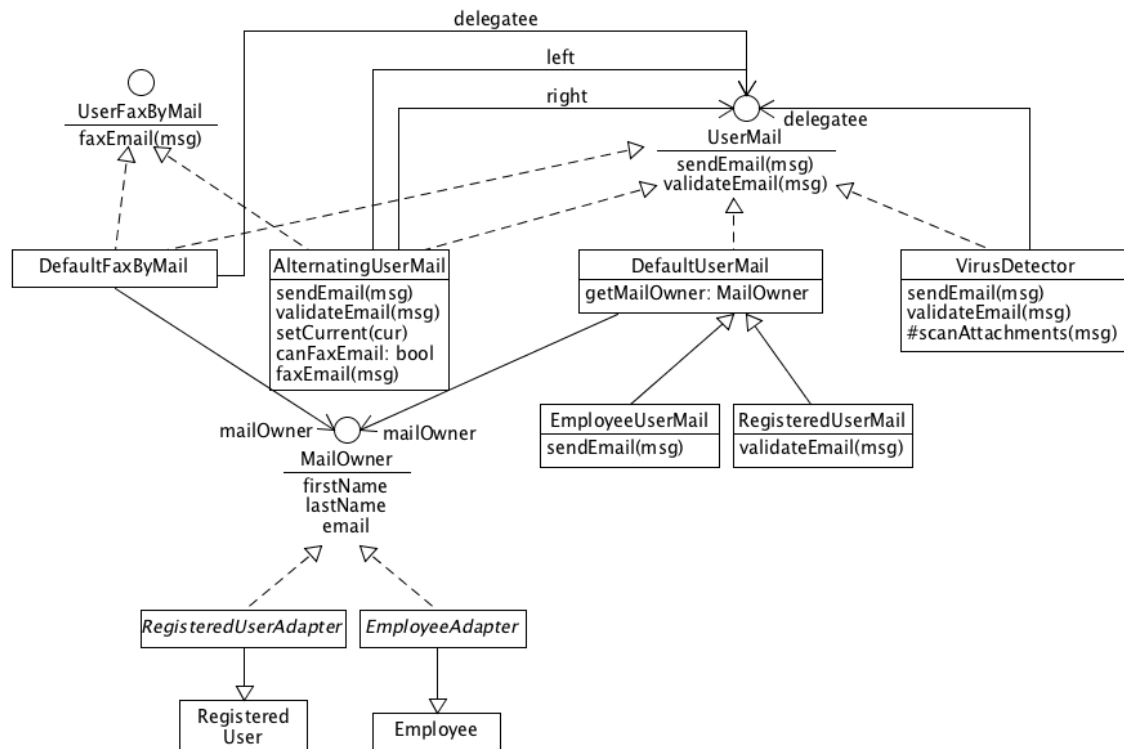
Figure 18: The overall schema of the no-trait email application

Assembling the Service

The following code sketches how the email service may be assembled from individual components.

Listing 20: Assembling and using the email service

```java
public static void useMailService(Map<String, Object> employeeData,
  Map<String, Object> regUserData) {

    Employee employee = initEmployee(employeeData);
    RegisteredUser registeredUser = initRegisteredUser(regUserData);

    // The client must use the concrete type AlternatingUserMail through
    // which the client can determine whether the service supports fax or not.
    AlternatingUserMail userMail = initMailService(employee, registeredUser);

    Message msg = new Message();
    msg.setRecipients(Collections.singletonList("pepa@gmail.com"));
    msg.setSubject("Hello");
    msg.setBody("Hi, Pepa!");

    userMail.sendEmail(msg);
    if (userMail.canFaxEmail()) {
      userMail.faxEmail(msg);
    }
```

```java
    }

    public static Employee initEmployee(Map<String, Object> employeeData) {
        Employee employee = new Employee();
        // Copying the data to the fields in Employee
        employee.load(employeeData);
        return employee;
    }

    public static RegisteredUser initRegisteredUser(Map<String, Object> regUserData) {
        RegisteredUser regUser;
        if (Boolean.TRUE.equals(regUserData.get("isPremium"))) {
            class Premium extends RegisteredUser implements PremiumUser {}
            regUser = new Premium();
        } else {
            regUser = new RegisteredUser();
        }
        // Copying the data to the fields in RegisteredUser
        regUser.load(regUserData);

        return regUser;
    }

    public static AlternatingUserMail initMailService(Employee employee,
      RegisteredUser regUser) {
        EmployeeAdapter employeeAdapter = new EmployeeAdapter(employee);
        final UserMail employeeMail =
            new VirusDetector(new EmployeeUserMail(employeeAdapter));

        final RegisteredUserAdapter regUserAdapter =
            new RegisteredUserAdapter(regUser);
        final UserMail regUserMail =
            new VirusDetector(new RegisteredUserMail(regUserAdapter));
        final UserMail regUserMailPremium =
            new DefaultFaxByMail(regUserAdapter, regUserMail);

        return new AlternatingUserMail() {
            @Override
            protected UserMail getDelegate() {
                Calendar c = Calendar.getInstance();
                int h = c.get(Calendar.HOUR_OF_DAY);
                if (!(h >= 8 && h < 17)) {
                    return getEmployeeMail();
                } else {
                    return getRegUserMail();
                }
            }

            UserMail getEmployeeMail() {
                return employeeMail;
            }
```

```
        UserMail getRegUserMail() {
          if (regUser.isPremium() &&
              regUser.getValidTo() != null &&
              regUser.getValidTo().compareTo(Calendar.getInstance().getTime()) >= 0)
            return regUserMailPremium;
          else
            return regUserMail;
        }
      };
    }
```

It is assumed here that the user owns both types of accounts. The variables `employee` and `registeredUser` are loaded from external data retrieved from a persistent storage.

The `initRegisteredUser` method turns the `isPremium` attribute included in the data record into the `PremiumUser`, which is composed with class `RegisteredUser` through the local class `Premium`.

Note: If the data had many independent "is-a" properties, such as `isPremium`, the code would quickly explode due to such local composition classes.

The two accounts are adapted to the common ground by corresponding adapters and used as the argument to `EmployeeUserMail` and `RegisteredUserAdapter` constructors.

There are three compositions of email service: one for an employee, one for a registered user and one for a premium registered user. The first two include `VirusDetector` and the third one also includes `DefaultFaxByEmail` to provide the premium fax service.

Then `AlternatingUserMail` is implemented and instantiated. This instance is able to switch the accounts in the background according to the current time. (If it is between 8am-17pm then `employeeMail` is used.) When the account of the registered user is to be used, the implementation also checks whether the registered user is a premium one and whether his account is valid. If so, the `AlternatingUserMail` instance uses the `regUserMailPremium` mail service.

`AlternatingUserMail` must be the topmost wrapper so that the client can use the `FaxByMail` interface to send emails by fax explicitly.

Summary

Developing a protean service on a platform without traits significantly reduces design options and tends to hide type information about the domain objects.

- As far as the two account adapters are concerned, there is no serious design flaw. The combination of inheritance and interface implementation preserves both types in instances (i.e. no object schizophrenia). Thus it is possible to check the type of an in-

stance and then to cast it to either `Employee` or `RegisteredUser` to get access to the account specific members.

- A potential problem lurks behind the converting of the `isPremium` property into the `PremiumUser` "trait". Such an approach does not scale with the growing number of independent "is-a" properties. This limitation is, however, inherent to the underlying "no-trait" platform.

- Another weakness is the cloning, because the state of the original account instances must be copied to the fields in the new adapted instances.

- The implementations of `DefaultUserMail` as well as of its two specializations `EmployeeUserMail` and `RegisteredUserMail` are pretty straightforward. The `DefaultUserMail` is designed as a package private abstract class. Thus it cannot be instantiated and only classes in from its package may extend it. This package contains two such classes only: `EmployeeUserMail` and `RegisteredUserMail`.

- Composition does not allow implementing optional interfaces, which may result in introducing hacks into the design, as illustrated on method `getMailOwner`.

- `VirusDetector` is forced to implement the `sendMail` method so as to perform the virus check before sending the email. However, it might not reflect the optional invocation of the email validation implemented in the underlying object.

- The implementation of the optional fax service `DefaultFaxByMail` also hides the account type. Additionally, there is a possibly dangerous assumption that it must be the topmost wrapper. Also, the test on premium user cannot be done by type.

- Probably the most problematic component is `AlternatingUserMail`. Besides its serious object schizophrenia stemming from the state pattern used to implement the switching, its most serious flaw is the conditional implementation of `UserFaxByMail`. Further, it must always be the topmost wrapper, which is handed over to the mail service client, which is tightly coupled with `AlternatingUserMail`, through which the client can determine the presence of the fax extension and invoke it.

- The preservation of types when adapting or extending other classes helps loosen the coupling between the instances and their client. For example, the user interface would be able to adapt itself to the type of the account and its functionality extensions (such fax-by-mail) just on the basis of the mail service instance's type.

- The presence of various `isSomething` properties may be alarming since it may indicate mixing of "has" and "is" aspects in the design. Such properties often exist in persistence models as `boolean` or as enumeration columns. In order to avoid the propagation of object schizophrenia through the domain model, such properties should be converted to traits or interfaces.

3.2.3   Modeling Protean Service With Static Traits

The previous analysis showed that modeling a protean component without traits leads inevitably to a design suffering from a number of flaws, most of them caused by object schizophrenia, which results from using composition or delegation instead of inheritance or extension.

This passage deals with modeling the email service by means of static traits as implemented in Scala. The domain model designed by means of the static traits is at first sight very similar to that of the non-trait case:
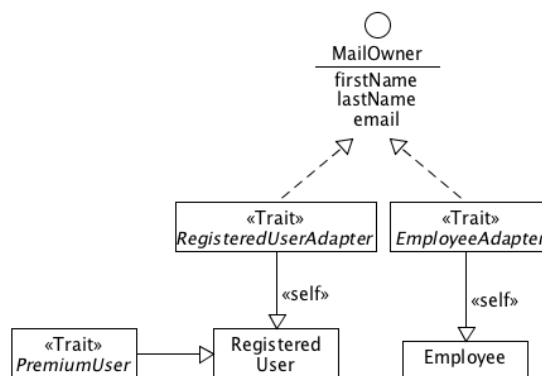
Figure 19: User model using static traits

However, there are several substantial differences. First, the `PremiumUser` type is a real trait, which, in contrast to the interface `PremiumUser`, extends the concrete class `RegisteredUser`. This extension explicitly declares that it is meant to be a trait of the registered user. Also, this trait can be applied on `RegisteredUser` only.

Second, the adapter types are also traits and no longer concrete classes (Listing 21).

Listing 21: Employee adapter with the self-type annotation

```scala
trait EmployeeAdapter extends MailOwner {
  this: Employee =>

  override def nick(): String = employeeCode

  override def email(): String = employeeCode + "@bigcompany.com"

  override def birthDate(): Date = birth
}
```

Implementing the adapters as traits makes it possible to combine them with other types in the application, in particular with the `UserMail` implementations. This is the important step toward the elimination of the delegation and composition.

The adapter traits do not extend the corresponding account classes as their counterparts in the no-trait design do. Instead, each adapter specifies the self-type to refer to the corresponding account class. Thus, the `EmployeeAdapter` may only exist in the context of `Employee`. All public and protected members of the types listed in the self-type are accessible to the referring trait; i.e. in effect, it is as if the trait "secretly" extended the self-type without inheriting.

As far as the mail service is concerned, it also utilizes traits as much as possible with the intent to eliminate delegation. All `UserMail` implementations `DefaultUserMail`, `EmployeeUserMail` and `RegisteredUserMail` are traits. It makes possible to combine them with the adapters as well as with the account classes. Each mail service trait also specifies the self-type to express its dependency on other types.

The combination of the account classes, adapters and mail service types is depicted in Figure 19.
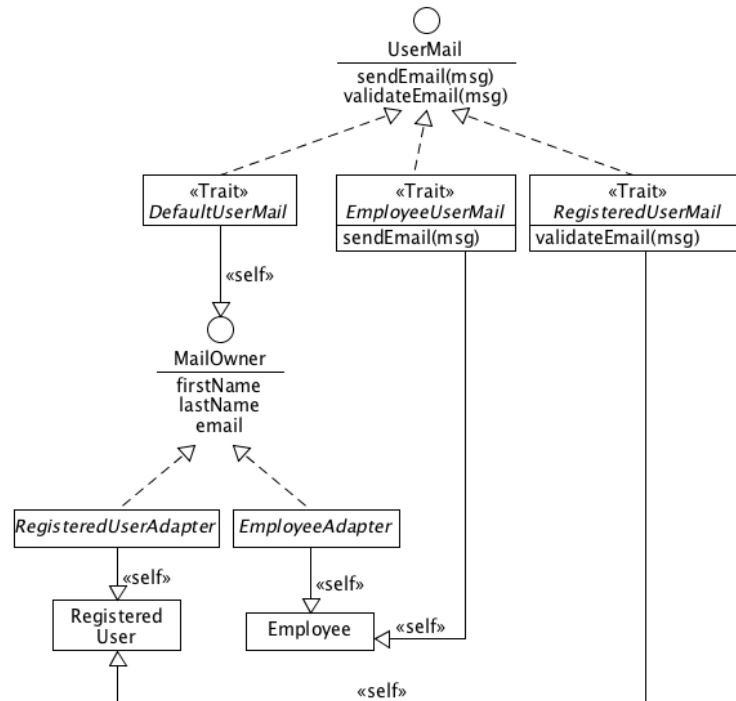
Figure 20: Mail service schema using static traits

If we compare this diagram with its no-traits counterpart, there is a significant difference: there is no delegation. Since all references are replaced by extensions and self-types it follows that there is no object-schizophrenia. For example, it is possible to invoke `mailService.isInstanceOf[Employee]` to determine whether the email service is built for an employee account.

Not even `VirusDetector` introduces any delegation in contrast to the no-trait design. As expected, `VirusDetector` is implemented also as a trait.
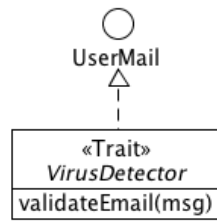
Figure 21: VirusDetector trait

Since VirusDetector is a trait, it is possible to implement the validateEmail method only and avoid the overriding of sendEmail. This workaround is no longer necessary, since there is no delegation and the trait becomes one monolith including DefaultUserMail. This default email service implementation invokes validateEmail within its sendEmail method. The invocation jumps into the top-most trait overriding the validateEmail method, from where the invocation may be propagated downward by super.validateEmail(msg).

Similarly to the VirusDetector, the UserFaxByMail is also implemented as a trait, as shown in the Figure 25.
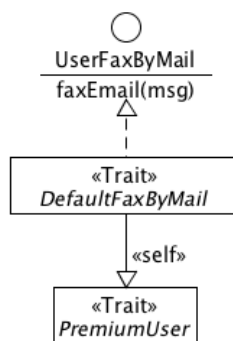
Figure 22: Fax service and its dependency on PremiumUser

The PremiumUser trait is combined with RegisteredUser as long as the registered user is a premium customer. Further, UserFaxByMail may be combined with premium users only, as specified in its self-type.

So far the design has manifested no flaws. Unfortunately, the implementation of the AlternatingUserMail must introduce some delegation. As seen in Figure 23, the model is identical to that of the no-trait case.
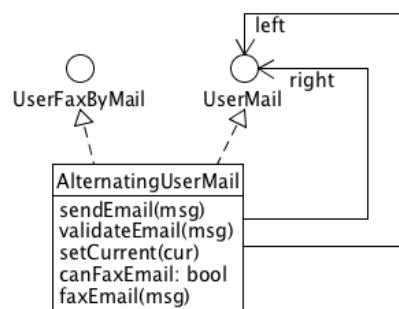
Figure 23: AlternatingUserMail still uses delegation

The reason is that the static trait platform has no type-safe tool to implement an instance, whose composition could be modified in real-time. Thus, all issues described in the no-trait case hold here as well.

The whole system is depicted in Figure 24.
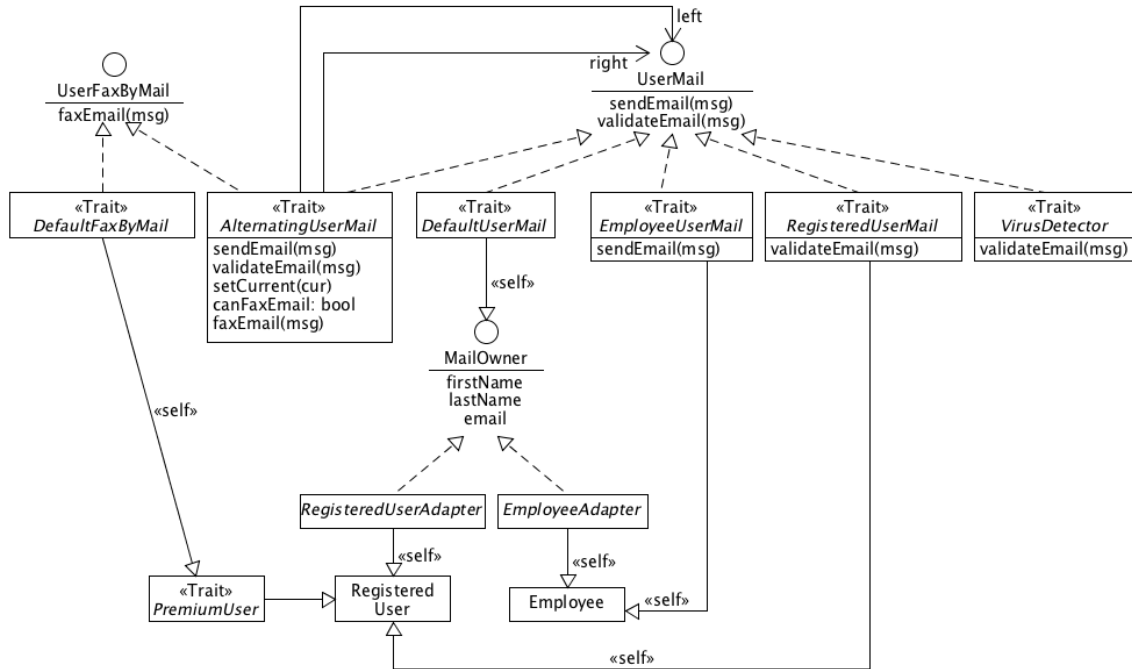
Figure 24: Schema of static-trait implementation of mail service

Assembling the Service

The following code sketches how the email service may be assembled from individual components.

Figure 25: Assembling email service based on static traits

```scala
def initializeMailUser(employee: Employee, regUser: RegisteredUser): UserMail = {

  val employeeMail = new Employee() with
    EmployeeAdapter with
    DefaultUserMail with
    EmployeeUserMail with
    VirusDetector

  employeeMail.adoptState(employee)

  val regUserMail = new RegisteredUser() with
    RegisteredUserAdapter with
    DefaultUserMail with
    RegisteredUserMail with
    VirusDetector
```

```scala
regUserMail.adoptState(regUser)

val regUserMailPremium = new RegisteredUser() with PremiumUser with
  RegisteredUserAdapter with
  DefaultUserMail with
  RegisteredUserMail with
  VirusDetector with
  DefaultFaxByMail
regUserMailPremium.adoptState(regUser)

new AlternatingUserMail {
  override protected def getDelegate: UserMail = {
    Calendar c = Calendar.getInstance()
    def h = c.get(Calendar.HOUR_OF_DAY)
    if (h >= 8 && h < 17) {
      getEmployeeMail
    } else {
      getRegUserMail
    }
  }

  def getEmployeeMail = {
    employeeMail
  }

  def getRegUserMail = {
    if (regUser.premium &&
      regUser.validTo != null &&
      regUser.validTo.toCalendar().after(Calendar.getInstance()))
      regUserMailPremium
    else
      regUserMail
  }

}
```

The first look on the code suggests that the assembling code is more compact. It is caused mainly by the Scala's syntax and by using traits in part. For example, it is not necessary to define the auxiliary `Premium` class, since the `PremiumUser` trait can be specified as part of the anonymous class signature in the `new` statement.

The `AlternatingUserMail` is an abstract class, whose `useLeft` method must be implemented to determine, which one of the two wrapped instances of `UserMail` will be used for the delegation. Here, the decision is made according to the current time. If it is between 8am-17pm then the employee email service is used (i.e. the "left" one).

Summary

There are two statements creating the registered user's mail instance; the only difference is that the latter creates an instance with the additional `Premium` trait. It is obvious that if there were another dimension, which should be expressed by type, such as gender, it would lead to further bifurcation. It follows that there would have to be as many pre-initialized instances as the number of all possible compositions of the registered user's mail service. It means, however, that modeling multidimensional objects by means of static traits is practically impossible for a higher number of dimensions because of the combinatorial explosion of instantiating statements.

We still need to clone the state of both preexisting `employee` and `registeredUser` instances.

`EmployeeUserMail` and `RegisteredUserMail` are now more general since they extend `UserMail` and not `DefaultUserMail`.

It would make sense to share a single `VirusDetector` instance between the employee and the registered user. `VirusDetector` may, for example, contain a virus counter field, which could count virus occurrences per-person instead of per `Employee` and per `RegisteredUser` separately. Unfortunately, static traits do not allow such a construction. A solution could be to mix the `VirusDetector` with `AlternatingUserMail`. Unfortunately, in such a case the virus detector's `validateEmail` method would no longer be invoked from `DefaultUserMail.sendEmail` because of the delegation.

The complete type information propagates well until the instantiation of the `AlternatingUserMail`. The client must be therefore still tightly coupled with `AlternatingUserMail`; nothing really changes from its perspective.

3.2.4 Modeling Protean Services With Dynamic Traits

As mentioned in Modeling Multidimensional Data With Dynamic Traits in Groovy, dynamic traits provide us with a way to mix various traits with the target object at runtime in a step-by-step way; in contrast to the static traits, where the traits are specified declaratively at once in the initialization statement. This section examines how to develop the protean mail service using the dynamic traits.

Modeling the Domain

When adding a trait to the target object, Groovy actually creates a proxy wrapping the target object and implementing all *interfaces* of the target object including the newly added traits, which are, in fact, also interfaces. The proxy, however, ignores the classes in the target object's type hierarchy and thus the proxy's type does not include such classes, as illustrated in Listing 22.

Listing 22: Ignoring classes when mixing traits into a target object

```
class X
trait Y1
trait Y2

def x = new X();
def y1 = x.withTraits(Y1)
def y2 = y1.withTraits(Y2)

assert x instanceof X
assert !(y1 instanceof X) // y1 forgets X, since X is a class
assert y1 instanceof Y1
assert y2 instanceof Y1   // y2 implements both Y1 and Y2
assert y2 instanceof Y2
```

The above-described limitation may be overcome by using traits instead of classes when modeling a domain. A Groovy trait may carry a state and thus it can be used instead of a class in many cases. This strategy will be followed in the following paragraphs. See Listing 23 showing the `Employee` trait modeling the employee entity.

Listing 23: Employee entity as a trait

```
trait Employee {

    private String firstName;
...
}
```

In order to instantiate an entity instance, we have to attach it to an "empty" object first, since a trait cannot be instantiated alone in Groovy (Listing 24).

Listing 24: Instantiating account objects in Groovy

```
def employee = new Object() as Employee;
employee.load(employeeData)
def regUser = new Object() as RegisteredUser;
regUser.load(regUserData)
```

The `as` keyword actually creates a proxy object, which wraps the "empty" object and implements `Employee`, resp. `RegisteredUser`.

Note: Although Groovy is primarily a dynamic language it also supports static compilation. Types or other parts of code annotated by `@CompileStatic` are compiled statically (Java Performance, n.d.).

Implementing the Behavior

Now, when the domain objects are created in a way, which allows loss-less extension, it is possible to implement the functionality of the email service.

The diagrams are practically identical to those for the static traits case. The only main difference is that the types specified in the self-type are added to the list of implemented interfaces. However it might sound confusing, the self-type in Scala is essentially a syntactic sugar for the same thing.

Listing 25: Default email service as a trait implementing `UserMail` and `MailOwner`

```
@CompileStatic
trait DefaultUserMail implements UserMail, MailOwner {
...
```

As mentioned above, the step-by-step extensions of objects sounds like a solution to the exponential explosion of statements. In order to make it more obvious, a couple of additional dimensions are introduced, such as `Gender` with "values" `Male` and `Female`, and `AgeGroup` with values `Child` and `Adult`. Listing 28 illustrates how these new dimensions are mixed into the registered user instance.

In contrast to the previous implementations, here, the `AlternatingUserMail` class is designed as an abstract class that does not wrap any email service explicitly; instead, it declares the abstract `getDelegate` method returning the current email service instance. In this case there are basically two email services, one for each account type. The employee's mail service is used during the office hours, i.e. 8am-17pm, and the registered user's one otherwise. The following listing shows the implementation of `getDelegate` the anonymous class extending `AlternatingUserMail` (Listing 26).

Listing 26: Implementing `getDelegate` in anonymous class extending `AlternatingUserMail`

```
AlternatingUserMail userMail = new AlternatingUserMail() {

    @Override
    UserMail getDelegate() {
      Calendar c = Calendar.getInstance();
      def h = c.get(Calendar.HOUR_OF_DAY);
      if (!(h >= 8 && h < 17)) {
          return getEmployeeMail();
      } else {
          return getRegUserMail();
      }
    }
    ...
```

The `getEmployeeMail` method is trivial; it always returns the same instance, since the composition of the employee's mail service is fixed. Its instance is held in `empMail` field in the anonymous implementation of `AlternatingUserMail` (Listing 27).

Listing 27: Employee email service

```
UserMail empMail = employee.withTraits(EmployeeAdapter, DefaultUserMail, VirusDetector)

    UserMail getEmployeeMail() {
        return this.empMail;
    }
```

The getRegUserMail method is more complex, since the actual composition of the service depends on several circumstances. The rules for the composition can be summarized as follows (Listing 28):

- If the registered user owns a premium license and this license is still valid, then the resulting mail service instance is marked by the Premium trait and extended by DefaultFaxByMail.

- If the user is a male, the instance is marked with Male, otherwise with Female.

- If the user is older than 18 years, the instance is marked with Adult.

Listing 28: Constructing a multi-dimensional registered user

```
  UserMail getRegUserMail() {
      UserMail regUserMail = regUser.withTraits(RegisteredUserAdapter, DefaultUserMail,
VirusDetector);

      def calendar = Calendar.getInstance()

      // Dimension 1
      if (regUser.premium && regUser.validTo.toCalendar().after(calendar)) {
          regUserMail = regUserMail.withTraits(Premium, DefaultFaxByMail);
      }

      // Dimension 2
      if (regUser.isMale()) {
          regUserMail = regUserMail.withTraits(Male);
      } else {
          regUserMail = regUserMail.withTraits(Female);
      }

      // Dimension 3
      def isAdult = (calendar.get(Calendar.YEAR) -
        regUser.birthDate.toCalendar().get(Calendar.YEAR)) > 18;
      if (isAdult) {
          regUserMail = regUserMail.withTraits(Adult);
      }

      return regUserMail;
  }
```

The above-mentioned rules represent three independent dimensions constituting a configuration space, where every point corresponds to one possible composition of the resulting mail service instance for the registered user.

A key fact is, however, that the assembling code no longer blows up exponentially with the number of dimension, which is a consequence of the step-by-step extensions.

Nevertheless, there are several downsides. First, a new composite instance is created again and again on each invocation of `getDelegate`, even when the structure of the new instance remains the same. This problem can be mitigated by storing the information about the structure of the last instance into a field. A new instance would be created only if the new composition differs from that of the previous one.

Second, each extension step leads to the creation of a new proxy object wrapping the object or proxy from the previous step. It follows that the final proxy may consist of a cascade of up to four proxies (three dimensions plus the proxied `RegisteredUser` entity). Thus, getting a value of a property in the `RegisteredUser` entity may amount to four delegated invocations on the nested proxies.

Third, since the state of any extension attached to the last instance is reset on every re-instantiation, it makes no sense for those extension traits to hold any state, in fact. It concerns especially the `VirusDetector` trait, since its counter of found viruses is periodically reset. A solution, or rather a workaround, could be to hold the extension's state out of the extension, which would, however, violate the encapsulation of data.

Summary

There is no doubt that using dynamic traits helped resolve some problems.

- Combinatorial explosion is over; the number of initialization statements ($O(n)$) is proportional to the number of dimensions (3) and not to the cardinality of the Cartesian product of the dimensions ($O(2^n)$).

- It is no longer necessary to clone the state of domain objects to the new extended instances.

On the other hand,

- Dynamic traits did not help solve the problem of metamorphism, and the client is still tightly coupled with `AlternatingUserMail` class, which provides the client with an interface to determine the current functionality of the mail service instance, for example. The metamorphism must still be implemented by delegation.

- Dynamic traits also bring a couple of new problems; some are inherently connected to the concept of dynamic traits, while others to Groovy's implementation.

- A trait does not have to implement all methods from the extended type, because it may be assumed that the missing methods will be delivered by other traits when extending an object. Since the extension mechanism includes no "trait-completeness" check, it may happen that some methods will be missing after extending the object by such a set of traits.

- A trait may also depend on traits that are not implemented by the target object. In dynamic languages it is in general quite difficult to check the dependencies earlier than at the moment of the actual extension at runtime. Composing more complex structures by dynamic traits is therefore prone to errors resulting from missing dependencies.

- An object can be extended by a trait type, not by a trait instance. It has important consequences especially for stateful traits, since their state is initialized at the moment of extending the target object. It follows that although the extended object remains the same during a series of repeated extensions, the attached traits themselves are always newly instantiated. It would be useful if we could extend two objects by the same trait instance to share the trait's state between the two objects.

- A cascade of step-by-step extensions leads to stacked proxies. As a result, an invocation of a method belonging to the most bottom trait propagates down through the whole stack of proxies.

- Groovy has a weak type system, which does not allow constructing composite types, such as `RegisteredUser & Male & Premium & Adult`. Such types could serve as a sort of type query language used to determine concrete traits in multiple dimensions:

Listing 29: Using a composite type when checking the type of an object

```
if (userMail instanceof RegisteredUser & Male & Premium & Adult) {
    //...
}
```

In Scala we could use the `match` construct to distinguish between various combinations of traits (Listing 30).

Listing 30: Using Scala's pattern matching to determine complex types

```
userMail match {
  case u: RegisteredUser with Male with Premium with Adult =>
    //...
}
```

The above-mentioned findings indicate that because of the lack of a strong static type system, modeling multidimensional domains by dynamic traits with complex dependencies is inherently prone to errors and the resulting implementation will tend to result in an unmaintainable code. With the growing number of dimensions and concrete traits there will

be an increasing probability that some method or some dependency will be missing in the final composition of traits extending an object.

The step-by-step concept of extending objects seems to be too flexible and unconstrained; it is the responsibility of the developer to ensure the consistency of trait compositions. However, considering that the number of trait combinations grows exponentially with the number of dimensions, the system will quickly run out of control of any developer.

It leads us to the conclusion that the platform, which is suitable for developing complex multidimensional mutable applications, should be equipped with a strong static type system and with the ability to perform safe dynamic compositions of traits.

3.2.5   Modeling Protean Services With Class Builder (static/dynamic)

The section dealing with the modeling of multidimensional data (see page 37) concludes that the current languages and platforms do not match perfectly with such a task. Scala, as a representative of a statically typed platform, is not able to represent statically the multitude of combinations arising from the multidimensionality. On the other hand, Groovy as a dynamic language with some support of static features, offers a very high flexibility, which, however, cannot guarantee the consistency of trait compositions made at run-time.

It has also been demonstrated that the concept of a class builder, which combines declarative and dynamic features, may be a solution to the problem of modeling complex multidimensional domains.

The goal of this section is to examine whether the class builder is able to solve the issues revealed in the previous sections when developing the protean email service.

Modeling the Domain

Let us begin with the modeling of the data objects, on which the mail service is built on. The multiform character of the data may be easily express declaratively by the following type expression in Scala extended by the "or" type operator. The Unit type represents objects with no state and no behavior. Since two instances of such a type would be indistinguishable there is exactly one such object (singleton). Here, the Unit type is an analogy to the unit element in algebra.

```
type UserModel = Employee or (RegisteredUser with (PremiumUser or Unit))
```

The type may be interpreted as follows: a user may be either an employee or a registered user, which may also be a premium user.

When rewritten to the equivalent disjunctive normal form, the type reveals the three possible compositions.

Listing 31: The disjunctive normal form of the user model

```
Employee or
(RegisteredUser with PremiumUser) or
RegisteredUser
```

The class builder in Listing 32 creates instances of the compositions from the above model. To create such a class builder the model type expression is specified in the type parameter of `ClassBuilder` and the trait initializer function is passed in as the constructor parameter. The trait initializer allows for an additional initialization of trait stubs.

Listing 32: The user class builder

```
val userBuilder = new ClassBuilder[UserModel]({
    case emp: Employee => emp.initFromMap(employeeData)
    case ru: RegisteredUser => ru.initFromMap(registeredUserData)
})
```

Since the model is specified as a type, the compiler extension is able to check the consistency of the model by analyzing the model type expression. For example, `PremiumUser` requires (through its self-type) that it may appear only in the compositions including `RegisteredUser` too.

The builder collaborates with the builder strategy when creating a new instance. The strategy recommends, which compositions or *alternatives* matches best with the context or input data. The strategy implements `isRecommended` method returning a logical value reflecting the strategy's opinion on the alternative passed as the argument (Listing 33).

Listing 33: `ClassBuilderStrategy` interface

```
trait ClassBuilderStrategy {
  def isRecommended(alt: Alternative): Boolean
}
```

The mutating nature of the user, which alternates between being `Employee` and `RegisteredUser`, requires that we create a special implementation of `ClassBuilderStrategy` (Listing 34).

Listing 34: The strategy recognizing whether the input user data represents an employee, a registered non-premium user or a registered premium user

```
val userBuilderStrategy = new ClassBuilderStrategy {
  def isRecommended(alt: Alternative): Boolean = if (isOfficeHours) {
    alt.containsTrait[Employee]
  } else {
    userData.get("premium") match {
      case true => alt.containsTrait[RegisteredUser] && alt.containsTrait[PremiumUser]
      case _ => alt.containsTrait[RegisteredUser] && !alt.containsTrait[PremiumUser]
    }
  }
}
```

During the office hours this strategy is recommending the alternative containing `Employee`, otherwise it is recommending the alternatives containing `RegisteredUser` either with or without `PremiumUser` according to the value of the `premium` property in the registered user input data.

The user reference is created as follows:

```
val userRef = userBuilder.newInstanceRef(userBuilderStrategy)
```

Such a reference object may be passed to other methods for subsequent mappings.

Modeling the Behavior

Now let us turn our attention to the modeling of the email service. Its interface consists of fixed and optional parts. The fixed one provides methods for the sending and validation of emails, while the optional one, which is available for premium users only, allows the user to send faxes by email.

The behavior of the fixed interface has an invariant component, which scans email attachments for viruses, while the variant component varies according to the underlying user object. If the user is an employee, the service automatically appends the employee's signature. If the user is a registered user, the service checks if the user's license is still valid.

The described protean behavior may be decomposed into traits and modeled by the expression in Listing 35.

Listing 35: Mail service model type expression

```
type MailServiceModel = DefaultUserMail with VirusDetector with
  ((EmployeeUserMail with EmployeeAdapter) or
   (RegisteredUserMail with RegisteredUserAdapter with (DefaultFaxByMail or Unit)))
```

This model is incomplete since some traits in the expression depend on one or the other user account, which are not part of this expression.

This type may be rewritten to its disjunctive normal form (Listing 36) to reveal the three alternatives (trait compositions) corresponding to the states of the email service.

Listing 36: The disjunctive normal form of the mail service model

```
(DefaultUserMail with VirusDetector with EmployeeUserMail with EmployeeAdapter) or
(DefaultUserMail with VirusDetector with RegisteredUserMail with RegisteredUserAdapter
with DefaultFaxByMail) or
(DefaultUserMail with VirusDetector with RegisteredUserMail with RegisteredUserAdapter)
```

The class builder creating instances of the above model is constructed by specifying the model type and passing the user account reference, which will deliver the missing trait stubs to complete the mail service model.

Listing 37: The mail service class builder

```scala
val mailServiceBuilder = new ClassBuilder[MailServiceModel](userRef)
val mailServiceRef = mailServiceBuilder.newInstanceRef
```

Let us have a look now, how the service model alternatives actually map to the alternatives of the user model.

Listing 38: The mapping of the mail service alternatives to the alternatives of the model of the user accounts

```scala
DefaultUserMail with VirusDetector with EmployeeUserMail with EmployeeAdapter->
Employee
DefaultUserMail with VirusDetector with RegisteredUserMail with RegisteredUserAdapter
with DefaultFaxByMail -> RegisteredUser with PremiumUser
DefaultUserMail with VirusDetector with RegisteredUserMail with RegisteredUserAdapter->
(RegisteredUser or (RegisteredUser with PremiumUser))
```

As shown in Listing 38, the first mail service alternative maps exactly to one alternative of the user model, because `EmployeeAdapter` depends on `Employee`. Similarly, the second alternative maps exactly to one user model alternative, since only one has both `RegisteredUser` and `PremiumUser` as required by `RegisteredUserAdapter` and `DefaultFaxByMail`.

However, the third alternative depending on `RegisteredUser` through `RegisteredUserAdapter` can be satisfied by two user model alternatives. This ambiguity may cause problems when the class builder instantiates a mail service and is selecting the target alternative for a user instance implementing both `RegisteredUser` and `PremiumUser` traits, since there are two eligible alternatives in the mail service model. This ambiguity may be solved by taking into account the sorting order of the target alternatives: the first matching target alternative wins. So in this case the second alternative (with `DefaultFaxByMail`) would win.

Runtime Morphing

For the client of the email service, such as the user interface, the email service mutations, which take place in the background, should be as transparent as possible. The core functionality should always be available regardless of the current composition of the service. Thus, the client must be given an email service reference, which handles such changes quietly and is usable all the time the service is offered.

Such a reference may be obtained from the reference wrapper object through the `mutableInstance` method.

```scala
val mailService: UserMail = mailServiceRef.mutableInstance
```

The returned object is in fact a proxy holding a reference to the current mail service composition of traits. The proxy implements the lowest upper bound type (LUB) of all target alternatives, which is `UserMail` in this case.

It could be the responsibility of the proxy itself or of an external agent other than the client to keep the proxy's delegate up to date.

Besides the LUB, the proxy also implements the "control" interface `MutableInstance`, which allows the client to control the proxy:

```
val mailService: UserMail with MutableInstance[MailServiceModel] =
mailServiceRef.mutableInstance
```

The control interface exposes the reference through the `delegate` method. This method may be used in situations, when the client wishes to customize its appearance, for example.

To determine for which type of user the email service is currently provided, one may call `delegate` and check the result type by `isInstanceOf` or by the `match` block (Listing 39).

Listing 39: Handling various subtypes of the user account by pattern matching

```
mailService.delegate match {
  case Employee => // handle an employee
  case RegisteredUser with PremiumUser => // handle a premium registered user
  case RegisteredUser => // handle a registered non-premium user
}
```

In a similar manner the client may detect that the service currently implements the `FaxByMail` extension (Listing 40).

Listing 40: Using the fax service via pattern matching

```
mailService.delegate match {
  case fax: FaxByMail => // use the fax
  case _ =>
}
```

Summary

- All forms, which a particular object may assume during its lifetime, may be expressed declaratively by a special type expression. The same expression is used in the previous case study to describe all forms, with which object can be born with. Thus, the concept of the class builder developed in this section generalizes the concept from the previous case study in such a way that the object created by the class builder may change the form (composition) given upon its creation.

- The type expression defines an object's morphing model and is analogous to a class. In fact, classes seem unnecessary in this approach.

- All advantages described in the previous case study hold here as well, such as the consistency check at compile-time, preservation of types and behavior during mappings, no combinatorial explosion, no need for copying, wrapping and delegating.

- The only delegation is present in the mutable proxy, which is, however, handled on the platform level, and not on the application level. The instance held by the delegating reference in the proxy never sinks, in contrast to the case of the repeated wrappings in the other approaches.

- It is possible to avoid the creation of redundant trait instances. This issue was illustrated in the previous section on `VirusDetector`.

- The client may check the availability of extension interfaces (e.g. `FaxByMail`) through the `delegate` method on the mutable instance; `AlternatingUserMail` is no longer needed.

- All trait stub objects (trait instances), regardless of many mappings they underwent, remain on the top of the instance, i.e. they are not "sinking" after every mapping.

4   Case Studies Analysis

The two case studies analyzed in this chapter showed 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, however 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.

In case the class builder is instantiated with an incomplete model (i.e. it has some missing dependencies), a special reference object must be passed to the class builder's constructor. Through this reference, the compiler extension can trace the model type associated with the reference and verify whether that model is able to deliver all the missing dependencies in the form of trait stubs.

The combination of an incomplete model with this special reference may be seen as an analogy to the mapping between two domains. Such a mapping has all the appealing properties formulated in the analysis for domain mappings: it uses neither delegation nor composition (no type schizophrenia) and the source and target trait stubs form one monolithic composition (the source stubs do not sink).

The following paragraphs focus on coining some basic terminology and on specifying the conceptual requirements for the class builder.

4.1   Terminology And Identified Concepts

The terminology defined here should be seen as conceptual only, which is going to be refined in the course of this work.

4.1.1   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.

4.1.2   Object Morphology

Object morphology is a general approach to 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.

4.1.3   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.

4.1.4   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.

A *mutable morph* is a morph, which may be *re-morphed* to another form (alternative) "under cover". It is used in situations where the client of the morph should be shielded from changes in the morph's composition.

All morphs created from the same model are compatible with the lowest upper bound type (LUB) of the model.

4.1.5   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). Each alternative consists of the so-called fragments, which represent properties or features of the protean objects (i.e. the concept's intension).

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.

An *incomplete morph model* is a morph model in which some alternatives are incomplete. Such an incomplete model must be merged with another morph model before it can be used to create morphs. The other morph model may also incomplete, however, the two together must produce a complete morph model.

4.1.6   Morph Model Alternative

A *morph alternative* describes one of the forms of a protean object.

A *complete morph alternative* is an alternative, in which all dependencies of all fragments are satisfied. A fragment's dependency is satisfied in an alternative as long as the alternative contains another fragment compatible with the dependency.

A *consistent alternative* is an alternative containing no mutually exclusive fragments.

An *alternative's type* is the type defined as the composition of all fragment types in the alternative.

4.1.7   Morph Fragment

A *morph fragment* is a 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 morph fragment represents a typological, behavioral and structural element of protean objects.

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.

The order, in which multiple wrappers overriding the same behavior occur in an alternative, is important, because the overall effect of the individual wrappers on the resulting behavior may depend on their position in the alternative.

4.1.8   Morph Strategy

During the instantiation of a morph a recognizer uses the *morph strategy* to select the best alternative to instantiate. The morph strategy receives all possible alternatives and selects the one that matches best the protean object (phenomenon) to be represented. The strategy may also reorder the fragments in the selected alternative to improve the representation, which is particularly important when some fragments in the alternative may override the behavior of other fragments; i.e. in such a situation the order of the overriding fragments may be important and it is up to the strategy to sequence properly the overriding fragments.

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.

4.1.9   Morph Type

A *morph type* is a static description of a morph model. Such a type is often a result of a special type expression.

4.1.10   Lowest Upper Bound (LUB) of Morph Model

The *lowest upper bound* (LUB) of a morph model is the most specific type that is compatible with the types of all alternatives defined by the morph model.

4.1.11   Morph Model Mapper

The *morph mapper* is a compile-time component composing one morph model with another. It performs a series of checks to guarantee that the two models produce a complete morph model. The link between the target and the source morph models is represented by a

special *morph reference*. The compiler may analyze such a reference and check whether the target and source models are compatible and form a complete morph model.

4.2 Code Samples

The following subsections refine the key listings from the case studies illustrating the use of the class builder (i.e. the recognizer). These snippets are supposed to capture how to express the key concepts and operations programmatically. The code snippets are not meant to provide a sort of specification; they should rather be seen as a conceptual pseudo-code suggesting possible constructions. The corresponding code developed with the help of the Reference Implementation (RI) may look differently.

Listing 41: A fragment with no dependency

```scala
trait Rectangle {
  var width: Float = _
  var height: Float = _

  def calculateArea: Float = width * height
}
```

Listing 42: A fragment with dependencies

```scala
trait ScannedBanknote {
  this: Banknote with Rectangle with Paper =>

  //...
}
```

Listing 43: A complete type expression

```scala
Thing with (Metal or Paper) with (Rectangle or Cylinder)
```

Listing 44: This expression may be rewritten to the disjunctive normal form

```scala
(Thing with Metal with Rectangle) or
(Thing with Metal with Cylinder) or
(Thing with Paper with Rectangle) or
(Thing with Paper with Cylinder)
```

Listing 45: An incomplete type expression

```scala
(ScannedBanknote with Banknote) or (ScannedCoin with Coin)
```

Listing 46: Using post-initializer when creating a morph

```scala
val itemBuilder = new ClassBuilder[Thing with (Metal or Paper) with
  (Rectangle or Cylinder)]({
```

```scala
      case rect: Rectangle =>
        rect.width = event.get("rectangle").get("width")
        rect.height = event.get("rectangle").get("height")
      case cyl: Cylinder =>
        cyl.height = event.get("cylinder").get("height")
        cyl.radius = event.get("cylinder").get("radius")
  })
  val item = Thing with Material with Shape = classBuilder.newInstance
```

Listing 47: Custom Morph Strategy

```scala
  val userBuilderStrategy = new ClassBuilderStrategy {
    def isRecommended(alt: Alternative): Boolean = if (isOfficeHours) {
      alt.containsTrait[Employee]
    } else {
      userData.get("premium") match {
        case true => alt.containsTrait[RegisteredUser] && alt.containsTrait[PremiumUser]
        case _ => alt.containsTrait[RegisteredUser] && !alt.containsTrait[PremiumUser]
      }
    }
  }

  val userRef = userBuilder.newInstanceRef(userBuilderStrategy)
```

Listing 48: Composing Incomplete Model to Produce Complete One

```scala
  type ItemModel = Thing with (Metal or Paper) with (Rectangle or Cylinder)
  type ScannedCurrencyModelWithLoaders = (ScannedBanknote with BanknoteLoader) or
    (ScannedCoin with CoinLoader)

  def makeCurrency(itemRef: Ref[ItemModel]): Ref[ScannerCurrencyModel] = {
    val curBuilder = new ClassBuilder[ScannedCurrencyModelWithLoaders](itemRef)
    loaderBuilder.newInstanceRef
  }
```

Listing 49: Creating a mutable morph

```scala
  val mailService: UserMail with MutableInstance[MailServiceModel] =
  mailServiceRef.mutableInstance
```

Listing 50: Evaluating the mutable morph's delegate reference

```scala
  mailService.delegate match {
    case Employee => // handle an employee
    case RegisteredUser with PremiumUser => // handle a premium registered user
    case RegisteredUser => // handle a registered non-premium user
  }
```

5   Theoretical Foundations of OM

This goal of this chapter is to develop the theoretical foundations of object morphology, on top of which the reference and other implementations of OM may be built.

The notion of recognizer is central in this chapter. A recognizer is defined as a device, roughly analogous to a cognitive system, producing representations of perceived phenomena. The "cognitive" capability of such a device is determined by the resolution of the morph model installed in the recognizer and also by the "intelligence" encapsulated in the morphing strategy used by the recognizer.

Recognizers are inherently composable systems; i.e. new complex recognizers may be constructed by composing simpler ones. Such a composition may be an evolutionary, a step-by-step, process, as described in 5.1.8. The section 5.1.9 deals with the conformance between morph models by generalizing the Liskov substitutions principle.

As mentioned above in the case study summary, a recognizer creates morphs; i.e. representations of the modeled protean objects (phenomena). These representations are composed of components called fragment stubs, which are de-facto instances of the fragments describing the properties and features of the protean objects. The composition of the morph is determined by the morph alternative selected by the recognizer's morphing strategy. The strategy may also reorder the fragments in the selected alternative in case the alternative contains more overriding fragments; i.e. wrappers. Before composing the morph, the recognizer creates instances of all fragments in the selected alternative.

An example of the recognizer may be the fictional airport scanner from the case study or a server-side software component identifying the remote user according to the properties transferred from the remote host. In the case of the airport scanner the protean objects are the input data consisting of the data produced by the scanner sensor array. The input data is further processed by the morphing strategy implementing the procedure for recognizing only such objects that can be matched to the alternatives given by the morph model. In the case of the server-side recognizer of remote users the protean objects are the session data such as IP address, type of the browser, cookies etc. The same user may be identified in different contexts (mobile, PC) and thus the composition of his/her representation may differ.

The recognizer is also able to *recompose* the morph to reflect the changes in the original object. The morph may be recomposed in essentially three ways. First, the recognizer uses the same morph model and morphing strategy; in such a case the final composition of the morph will reflect the same aspects of the original object. Second, the recognizer uses another strategy; in this case, the new composition will reflect different aspects of the original object than the original composition. And third, the object is recomposed by means of a richer morph model; i.e. a model with a better resolution. In this case the recomposed object may contain new fragments and new alternatives in addition to that of the original model.

The following subsections study theoretical aspects of recognizers through developing the concept of *abstract recognizer*.

5.1   Recognizer Formalism (R-Algebra)

The abstract recognizer (AR) is a virtual device "perceiving" an original object (phenomenon) and providing the original object's representation, if recognized. The schema of AR is shown in Figure 26.
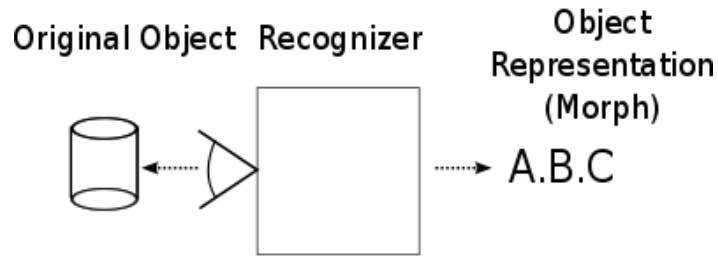
Figure 26: Schema of an abstract recognizer

5.1.1   Basics

First, in order to make the meaning of symbols obvious, it is necessary to introduce the following convention, which is used throughout the text:

1. Symbols referring to morph models: a capital letter in bold face italic face; i.e. `M`

2. Symbols referring to alternatives: a capital letter in bold face is used; i.e. `A`

3. Symbols referring to fragments: a capital letter is used; i.e. `F`

4. $M(x)$ stands for a recognizer using morphing strategy x to recognize the alternatives from the model M.

A morph model may be formally expressed as a list of morph alternatives.

```
M -> {A₁,…,Aₙ} N>0
```

Members A_i correspond to the individual alternatives and `N` is the number of alternatives produced by the model. The number `N` is called the *resolution* of the morph model.

Each alternative can be further expressed as a list of fragments:

```
Aᵢ = (F₁ᵢ,…,Fᵢᵢ)
```

Members F_{ji} represent the fragments constituting the i-th alternative and `I` is the number of the fragments in this alternative.

Using this formalism, the morph model of scanned items from the case study can be expressed as follows:

```
{(Paper, Rectangle), (Paper, Cylinder), (Metal, Rectangle), (Metal, Cylinder)}
```

The alternatives of this model are:

```
A₁ = (Paper, Rectangle)
A₂ = (Paper, Cylinder)
A₃ = (Metal, Rectangle)
A₄ = (Paper, Cylinder)
```

The previous morph model may be seen as a model of a recognizer composed of two simpler recognizers $S(x)$ and $M(x)$ using the following morph models:

```
S -> {(Paper), (Metal)}
M -> {(Rectangle), (Cylinder)}
```

The partial recognizers $S(x)$ and $M(x)$ can be further decomposed to additional four recognizers $P(x), M(x), R(x)$ and $C(x)$, which use the models outlined in the following figure:

```
P -> {(Paper)}
M -> {(Metal)}
R -> {(Rectangle)}
C -> {(Cylinder)}
```

The list of representations produced by the coupled recognizers $S(x)$ and $M(x)$ is in this case the Cartesian product of the partial representations. On the contrary, the coupled recognizers $P(x)$ and $M(x)$, resp. $R(x)$ and $C(x)$ produce the union of the two partial lists. These two types of interaction between two recognizers correspond to the two elementary relationships between two morph models: *co-occurring* and *mutually exclusive*. Besides these two boundary relationships other possible relationships may occur between more complex morph models, as shown later in this passage.

The following paragraphs examine possible relationships between morph models and introduce a formalism called *R-algebra*, whose purpose is to provide some formalism for expressing complex morph models.

First, an investigation of simple cases will be done. Let A and B be two fragments and *A* and *B* be single-alternative-single-fragment morph models:

```
A -> {(A)}, B -> {(B)}
```

Further, let $AB(x)$ be a recognizer composed of the partial recognizers $A(x)$ and $B(x)$. Without knowing the relationship between fragments A and B; i.e. between occurrences of features A and B in the original object, it is impossible to determine the morph model of the

composite recognizer **AB**(x). The **A** and **B** may be mutually exclusive or may occur side-by-side; i.e. co-occur, or may have other form of relationship. There are actually four possible model compositions outlined in Listing 51.

*Listing 51: Four possible relationship between two single-alternative-single-fragment models **A** and **B***

```
1. {(A, B)}
2. {(A), (B)}
3. {(A), (A, B)}
4. {(A, B), (B)}
```

The first model reveals that the two features A and B co-occur; neither A nor B occurs alone. Contrarily, the two features in the second model are mutually exclusive; i.e. they never occur side-by-side. In the third model the feature A is always present and the feature B is optional. Here, it is said that B depends on A, since B always occurs side-by-side A. The fourth model actually corresponds to the same situation as the previous model; here A depends on B.

The above-mentioned four situations may be generalized for models with arbitrary number of alternatives.

```
M1 = {A₁,…,Aₙ}, Aᵢ = (Aᵢ₁,…,AᵢI)
M2 = {B₁,…,Bₘ}, Bⱼ = (Bⱼ₁,…,BⱼJ)
```

First, let us assume that **M1** and **M2** are mutually exclusive, which means that no two alternatives, one from **M1** and the other from **M2**, are *co-occurring*. Obviously, the morph model **M** composed of **M1** and **M2** will generate the union of the partial alternatives:

```
M = {A₁,…,Aₙ,B₁,…,Bₘ}
```

The resolution of this composite model is the sum of the partial resolutions:

```
Res(M) = Res(M1) + Res(M2)
```

Now, let us turn the attention to co-occurring models; the two models are co-occurring if every two alternatives, one from **M1** and the other from **M2**, are co-occurring. In other words, any alternative from **M1** may be combined with any other alternative from **M2**. This fact can be expressed by a more complicated formula, which is de-facto the result of the Cartesian product of the two lists of alternatives:

```
M = {A₁B₁,A₁B₂,…,AₙBₘ}
AᵢBⱼ = (Aᵢ₁,…,AᵢI,Bⱼ₁,…,BⱼJ)
```

The resolution of this composite model is the product of the partial resolutions:

```
Res(M) = Res(M1).Res(M2)
```

The preceding two generalizations prepared the ground for the definition of two operations *join* "." and *union* "|" for two different compositions of morph models.

Let us define first the "|"operation:

*Let **M1** and **M2** be two morph models, **M1** = {A_1,…,A_N}, A_i = (A_{i1},…,A_{iI}), **M2** = {B_1,…,B_M}, B_j = (B_{j1},…,B_{jJ}). The binary **union** operation "|" applied to the two models **M1** and **M2** returns a new model **M** that generates the following alternatives: **M** = {A_1,…,A_N,B_1,…,B_M}*

The definition of the "." operation follows:

*Let **M1** and **M2** be two morph models, **M1** = {A_1,…,A_N}, A_i = (A_{i1},…,A_{iI}), **M2** = {B_1,…,B_M}, B_j = (B_{j1},…,B_{jJ}). The binary **join** operation "." applied to the two models **M1** and **M2** returns a new model **M** that generates the following alternatives: **M** = {A_1B_1,A_1B_2,…,A_NB_M}, where A_iB_j = (A_{i1},…,A_{iI},B_{j1},…,B_{jJ}).*

It follows directly from the definitions that for every model `M = {`A_1`,…,`A_N`}`, `A`$_i$`=(`A_{i1}`,…,`A_{iI}`)` there exists a composite model having the form shown in Listing 52.

*Listing 52: Exclusive normal form (ENF) **M'** of morph model **M***

```
M' = A₁|…|Aₙ
Aᵢ = Aᵢ₁.….AᵢI
Aᵢⱼ = {(Aᵢⱼ)}
```

The composite model `M'` is called the *exclusive normal form* (ENF) of `M` and generates the list of alternatives identical to the list generated by `M`. The model `M'` may be used in situations inverse to the one studied above; so far it has been supposed that there are two or more known morph models and the goal was to construct a new morph model composed of the two partial models. However, there may be another situation, in which the morph model alternatives are known (and thus the exclusive normal form `M'`) and the task is to factor them so as to reveal potential partial morph models by means of the R-Algebra rules.

To illustrate such a task let us consider the following model `M`.

```
M = {(A1,A2),(A1,A3),(A4,A2),(A4,A3)}
```

The ENF of `M` is then

```
M' = A1.A2 | A2.A3 | A4.A2 | A4.A3
```

Let us further assume that the "." operation distributes over "|" (proven below). Then `M'` could be modified in the following way:

```
M' = (A1 | A4).(A2 | A3)
```

The two groups actually reveal two partial morph models:

```
  M' = M1.M2
  M1 = A1|A4
  M2 = A2|A3
```

Provided that *M'* generates the identical list of alternatives as *M* the original model *M* may be considered a composition of the two independent sub-models *M1* and *M2*:

```
 M = M1.M2
 M1 = {(A1),(A4)}
 M2 = {(A2),(A3)}
```

The proof of the distributive property of ".", follows.

Let *M1*, *M2* and *M3* be three morph models such that

```
 M1 = {A₁,…,Aₙ}, Aᵢ = (Aᵢ₁,…,AᵢI)
 M2 = {B₁,…,Bₘ}, Bⱼ = (Bⱼ₁,…,BⱼJ)
 M3 = {C₁,…,Cₖ}, Cₖ = (Bₖ₁,…,BₖK)
```

Assuming that the sub-model *X=M2|M3* generates the union of the partial lists of *M2* and *M3*, the resulting list of *M1.X* will be this Cartesian product:

```
 M1.(M2|M3) = M1.X -> {A₁B₁, A₁B₂, …, AₙBₘ, A₁C₁, A₁C₂, …, AₙCₖ}
```

Next, let us suppose that the distributive property holds:

```
 M1.(M2|M3) = M1.M2 | M1.M3
```

Then the partial lists of alternatives of *M1.M2* and *M1.M3* will be:

```
 M1.M2 -> {A₁B₁, A₁B₂, …, AₙBₘ }
 M1.M3 -> {A₁C₁, A₁C₂, …, AₙCₖ}
```

The "|" operator between *M1.M2* and *M1.M3* produces the union of the two partial lists:

```
 M1.M2 | M1.M3 -> {A₁B₁, A₁B₂, …, AₙBₘ, A₁C₁, A₁C₂, …, AₙCₖ}
```

The preceding list of alternatives matches exactly the one derived in the beginning of the proof and therefore the distributive property of "." over "|" has been proved. Q.E.D.

The distributive property is especially useful if the composite model consists of many optional and non-exclusive single-fragment morph models. Given N optional fragments F_1 … F_N the model *M* will generate the following list of alternatives:

```
 M -> {(F₁,…,Fₙ), … ,(1)}
```

Note: The fragment 1 represents "anything else than" $F_1...F_N$ and is explained in section 5.1.5.

The resolution of this model is 2^N, which makes it practically impossible to specify the composed model by means of the original exclusive normal form *M'*. However, using the distributive property, *M'* may be significantly simplified:

```
M' = (F1 | 1).(F2 | 1). … .(FN | 1)
```

This refactored model consists only of N terms (*F_i* | *1*), which makes the specification of the formula feasible.

The third model in Listing 51 can also be expressed with the help of the unit fragment 1:

```
M' = A | A.B = A.(1 | B) = {(A), (A, B)}
```

R-algebra declares as an axiom that the two operations "." and "|" are *commutative*:

```
A.B = B.A
A|B = B|A
```

Although, as mentioned in 4.1.8, the order of overriding fragments (wrappers) in an alternative is important, it is not relevant at the morph model level; all fragment permutations in the alternative are possible from the morph model's perspective. It is up to the morphing strategy to sort the fragments in the selected alternative so as to match best the context conditions.

In terms of *associativity*, both operators obviously satisfy it since the order of expression processing (i.e. left-to-right or right-to-left) does not affect the position of fragments in alternatives or the ordering in lists of alternatives.

```
(A.B).C = A.(B.C)
(A|B)|C = A|(B|C)
```

Let us examine now whether the following identity holds:

```
M | M = M
```

The result of this union can be expressed as:

```
{A1,…,AN,A1,…,AN}
```

Since a set does not contain the same element twice, the result set actually shrinks to the alternatives of *M*:

```
{A1,…,AN}
```

The following rules concern the two special fragments 0 and 1. The 0 fragment may be seen as a representation of nothing. It corresponds to the situation when a recognizer recognizes nothing in the original object and thus it returns nothing as the object representation. There is a special recognizer called *null recognizer* described in 5.1.4 that always produces nothing, i.e. 0.

On the other hand, the 1 fragment represents *something* and the special unit object 1 is returned by recognizers detecting only the existence of a feature instead of materializing the feature into a fragment stub. There is a recognizer called the *universal detector* described in 5.1.5 that returns 1 for every object.

Joining any model A with 0 naturally results in 0, since "something" cannot occur at the same time with "nothing":

```
A.0 = 0
```

Further, the composition of any model A with 1 obviously results in the model A; the unit object carries no additional information, which could be added to A, and therefore the following holds:

```
A.1 = A
```

5.1.2   Saturated Models

Let us now examine the result of joining the same morph model multiple times with itself.

```
M.M.….M = MᴺN
```
$$M.M.….M = M^N$$

In the simplest case, when M consists of only one alternative, the result of $M.M$ is M itself.

```
M.M = M, if Res(M) = 1
```

The same property has also the following model with three alternatives:

```
M = A | A.B | B, Res(A) = Res(B) = 1
```

It can be shown that $M.M = M$ by using the commutativity rule, the fact that $M.M = M$ if $Res(M) = 1$ and the rule that $M | M = M$ for any M:

```
M.M = (A | A.B | B).(A | A.B | B) = A.A | A.A.B | A.B | A.B.A | A.B.A.B | A.B.B
| B.A | B.A.B | B.B = A | A.B | B = M
```

The section 5.1.6 introduces the so-called *inverse fragments*, which may be used as normal fragments and for which holds the following rule:

```
F.~F = 0, ~F is the inverse fragment of F
```

For the following model containing inverse fragment it also holds that $M.M = M$:

```
M = F1.~F2 | ~F1.F2
```

This fact can be easily proven by applying the rule $F.{\sim}F = 0$:

```
M.M = (F1.~F2 | ~F1.F2).(F1.~F2 | ~F1.F2) = F1.~F2 | ~F1.F2 = M
```

So far it has been shown that there exists certain models for which it holds that $M.M = M$. The other models do not have this property, such as

```
M = A | B
M.M = A | A.B | B != M
```

However, as long as $M.M$ is joined once more with M the result remains $M.M$:

```
M³ = M.M.M = (A | A.B | B).(A | B) = M.M = M²
```

The model M^2 is called *saturated*, as additional joins with the original model M do not produce bigger models. The exponent of the saturated model is called the model's *saturation* and is denoted $\text{Sat}(M)$.

```
M^Sat(M) = saturated model M
```

The *maximum saturation* of any model with N alternatives is N, as shown in the following formulas.

```
M  = A₁ | … | Aₙ
Mᴺ = A₁ | … |
     A₁.A₂ | … |
     A₁.A₂.A₃ | … |
      … |
     A₁.….Aₙ
Mᴺ.M = Mᴺ.(A₁ | … | Aₙ) = Mᴺ
```

The join of M^N with M certainly produces no new combination of fragments by joining mutually the alternatives $A_1..A_N$. The real saturation of M may be less than N in case that the alternatives share some fragments.

Interestingly, any model M may be transformed easily to the saturated model by completing every alternative from M by the inverse fragments of the fragments not included in that alternative:

```
M = F1 | F2 | F2.F3
M_sat = F1.~F2.~F3 | ~F1.F2.~F3 | ~F1.F2.F3
M_sat.M_sat = M_sat
```

It follows from the above that the models for which $M.M = M$; i.e. having saturation 1, are equivalent to their saturated model M_{sat}; i.e. $\mathrm{Sat}(M)=\mathrm{Sat}(M_{sat})= 1$. In other words, the alternatives of such models contain (implicitly or explicitly) the complementary inverse fragments.

The significance of the saturation property may be illustrated on the scenario in which two recognizers equipped with the same model are used to recognize features in the same object in parallel.

First, let us consider two recognizers working with the same **non-saturated** model. It is further assumed that each recognizer uses a different morphing strategy. Then it may happen that those recognizers simultaneously recognize two distinct alternatives A1, A2 in the real object, such that the join *A1.A2 does not belong* to the used (non-saturated) morph model. In other words the two recognizers running in parallel may reveal new alternatives; i.e. *new forms*.

On the other hand, two recognizers equipped with the same **saturated** model and using different strategies may also recognize two different alternatives **A1**, **A2** at the same time. However, in this case the join *A1.A2* will *always belong* to the (saturated) morph model used by the two recognizers.

It may be concluded from the above that the parallel recognizers in the former case (non-saturated models) *increase the resolution* of the recognition, i.e. the number of alternatives, while the recognizers in the latter case *improve the accuracy* (quality) of the recognition.

The above-described effect may be illustrated on a simple model consisting of three single-fragment alternatives A, B, C (Listing 53). The goal is to examine how the resolution and the accuracy change in response to adding new recognizers.

Listing 53: Increasing the resolution and the accuracy by saturating the model

```
M = A | B | C

1. M(s1) Res = 3
2. M(s1).M(s2) Res = 6
3. M(s1).M(s2).M(s3) Res = 7, saturated
4. M(s1).M(s2).M(s3).M(s4) Res = 7, begins to raise quality, the resolution
stops increasing
```

The saturation of this model is 3; it follows that the resolution will increase until the number of recognizers exceeds 3. After that, every new recognizer will improve the accuracy of the recognition.

The first scenario begins with a single recognizer M(s1) (the parameter represents the strategy used by the recognizer). The resolution of the model is 3. In the second scenario a new recognizer using strategy s2 is added, which results in the increase of the resolution to 6. In the third scenario the joined model of the three recognizers becomes saturated; i.e. the

resolution is 7. In the fourth scenario, the new recognizer does not change the overall resolution (it remains 7), but it improves the accuracy (quality) of the recognition (two independent suggestions are composed).

Having analyzed the "exponentiation" of a morph model, the question of finding the analogy to the inverse operation should automatically follow; i.e. the question of determining the "nth root" of a morph model.

Let M be a morph model. The nth root R of M, denoted $M^{1/n}$, is a morph model fulfilling the following condition:

```
M^{1/n} = R; R^n = M
```

Let us illustrate the concept of the nth root on several examples. The following example shows that while the 2nd root of model $A \mid A.B \mid B$ is $A \mid B$, the 3rd root is not defined.

```
(A | A.B | B)^{1/2} = A | B
(A | A.B | B)^{1/3} = undefined
```

The next example presents the 2nd and 3rd root of a more complex model, while the 4th root is found undefined.

```
(A | B | C | A.B | B.C | A.C | A.B.C)^{1/2} = A | A.B | B
(A | B | C | A.B | B.C | A.C | A.B.C)^{1/3} = A | B
(A | B | C | A.B | B.C | A.C | A.B.C)^{1/4} = undefined
```

It follows from the above that for every morph model M there exists such n, denoted $Root(M)$ and called simply *the root of M*, that the nth root of the model exists, while the $(n+1)$th root is undefined:

```
Root(M) = n; M^{1/n} exists, M^{1/(n+1)} is undefined
```

In view of the preceding findings it may be concluded that for every morph model M there exist two boundary models: $M^{1/Root(M)}$ and $M^{Sat(M)}$. Because of the similarities with the logical operators XOR and OR the two boundary models are named accordingly:

```
XOR(M) = M^{1/Root(M)} a.k.a. the root model of M
OR(M)  = M^{Sat(M)}    a.k.a. the saturated model of M
```

The $XOR(M)$ model assumes that no two alternatives of M may occur simultaneously; i.e. they are mutually exclusive. On the other hand the $OR(M)$ model assumes that any two alternatives of M may co-occur.

There are a couple of basic relations between XOR and OR models.

Having in mind that the highest estimation of the saturation of any model is equal to its resolution, it must hold that

$$XOR(M)^{Res(XOR(M))} = XOR(M)^{Sat(XOR(M))} = OR(XOR(M))$$

The previous identity uses the estimation of the saturation of $XOR(M)$:

$$Sat(XOR(M)) \mathrel{<=} Res(XOR(M))$$

Furthermore, it may be proved that $OR(XOR(M)) = OR(M)$. From the definition of $OR(X)$ as the saturated model of X follows:

$$OR(XOR(M)).XOR(M) = OR(XOR(M)) \Rightarrow$$
$$OR(XOR(M)).XOR(M)^{Root(M)} = OR(XOR(M)).M = OR(XOR(M))$$

It follows, however, that $OR(XOR(M))$ must be the saturated model of M on account of the definition that the saturated model of M is the non-empty model S for which

$$S.M = S = M^{Sat(M)}$$

Thus it must hold that

$$OR(XOR(M)) = M^{Sat(M)} = OR(M)$$

The preceding identity allows deducting that the saturated model of M is identical to the root model of M powered by the root's resolution.

$$OR(M) = XOR(M)^{Res(XOR(M))}$$

The preceding formula suggests that there is a spectrum of *or-like* morph models associated with any morph model M. The spectrum begins with $XOR(M)$ on one end and ends with $OR(M)$ on the other end. The other models in the spectrum may be considered hybrid XOR/OR models; in other words, the closer is a model to the XOR end (the lesser is the root of the model) the more exclusive the model is and vice versa.

The span of the spectrum is defined in Listing 54.

Listing 54: The span of the model's spectrum

$$Span(M) = Res(XOR(M))$$

Finally, a remark concerning the models containing inverse fragments should be given. Let us consider two recognizers equipped with one such a model. When recognizing the same object there may occur two situations: either the two recognizers recognize the same alternative or different ones. In the latter case, however, the two alternatives cannot contain contradicting fragments; in other words, it would be an error if one alternative recognized F while the other recognized ~F.

To illustrate the problem, let us consider the following example:

```
M = A.~B | ~A.B
M(s1).M(s2)
```

Let us say that the first recognizer `M(s1)` recognizes the first alternative; i.e. the **presence** of A and the **absence** of B. In such a case the second recognizer `M(s2)` must also recognize the first alternative, since, otherwise, the other strategy would suggest the **absence** of A and the **presence** of B; i.e. a contradictory recognition. The preceding analysis provides the grounds to the conclusion that the saturation of `M` is 1 (and not 2).

```
OR(M) = M, Sat(M) = 1
```

In the following sections a closer look at special types of abstract recognizers is taken. Also additional rules and definitions concerning R-algebra will be described during that examination.

5.1.3   Fragment Recognizer

A fragment (or wrapper) recognizer is an elementary recognizer capable of recognizing a certain fragment; i.e. a fragmentary feature in the original object. In case it recognizes the fragment, it produces a representation consisting of an instance of the fragment initialized by the "measured" values of the real object's properties. It does not produce any representation as long as it does not recognize the fragment; i.e. it produces nothing (0). The resolution of the fragment recognizer is 1 (see Figure 27).
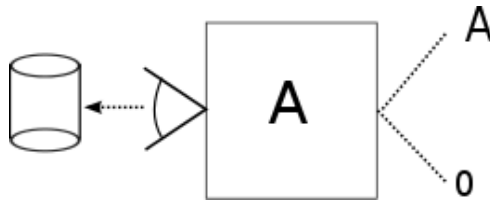
Figure 27: Schema of a fragment recognizer

An example of a fragment recognizer might be a recognizer determining a particular shape or a particular material of a scanned item in the case study.

Fragment Dependencies

Some features described by fragments may occur along with other features in the original object. As said earlier, this co-occurrence may be understood as an indication of dependency between the co-occurring fragments. The dependency relationship is symmetrical, which means that as long as feature A depends on B, then B automatically depends on A. It does not follow, however, that A and B must always occur side by side; the two fragments may alternatively co-occur with other fragments too. For example, the fragments A and B

can have the dependencies shown in *Listing 55* that allow the co-occurrence of A and C, resp. B and D besides the combination A and B stemming from the symmetry of the dependency between A and B.

Listing 55: Alternative dependencies of fragments A and B

```
A => B or A => C
B => A or B => D
```

The same configuration can be expressed using R-algebra:

```
A => B | C
B => A | D
C => A
D => B
```

The dependencies of a fragment actually constitute a morph model whose alternatives correspond to the alternative dependencies of the fragment.

```
A => Deps(A)
```

The *Deps*(A) is a morph model generating alternatives containing the fragments on which A depends. The composition of the fragment with its dependencies morph model *A.Deps(*A*)* can be interpreted as a definition of the *fragment context*, i.e. a morph model in which all dependencies of fragment A will be satisfied.

```
Actx = A.Deps(A)
```

The context model of the fragment A from the introductory example is then

```
A.(B | C) = A.B | A.C
```

There are two trivial contexts involving *0* and *1* models. First, let us consider the following dependency:

```
A => 1
```

The context of the fragment A is *A.1 = A*, which means that A may exist in any model containing A. This dependency actually represents no-dependency, as the fragment A depends on itself only.

Another trivial dependency is the dependency on *0*:

```
A => 0
```

The context of A is *A.0 = 0*; in other words, there is no morph model, in which the fragment A might exist and therefore the fragment A is not recognizable by any recognizer.

The context morph model of a fragment may not always be complete; the fragments, on which the fragment depends, may also have their own dependencies, which the context model may not satisfy. Therefore it makes sense to consider another morph model satisfying not only the fragment's dependencies but also the dependencies of the dependencies and so on. Such a model is called the *fragment closure*.

The closure of a fragment is constructed by recursive substitution of all fragments, on which the fragment depends either directly or indirectly, by their context models.

To illustrate the construction of the fragment closure let us examine the following system of dependencies:

```
A => B.X | C
B => A.Y | D
C => A
D => B
X => A.B
Y => A.B
```

The fragment A's closure and context may be obtained using the algorithm shown in *Listing 56*.

Listing 56: Derivation of closure dependency models

$$A_{closure} = B_{ctx}.X_{ctx} \mid C_{ctx} = B_{ctx}.X.A_{ctx}.B_{ctx} \mid C.A_{ctx} = B_{ctx}.X \mid C =$$
$$B.(A_{ctx}.Y_{ctx} \mid D_{ctx}).X \mid C = B.(Y \mid D).X \mid C = B.X.Y \mid B.D.X \mid C$$

$$B_{closure} = A_{ctx}.Y_{ctx} \mid D_{ctx} = A.(B_{ctx}.X_{ctx} \mid C_{ctx}).Y_{ctx} \mid D.B_{ctx} = A.(X \mid C).Y \mid D =$$
$$A.X.Y \mid A.C.Y \mid D$$

$$C_{closure} = A.(B.X.Y \mid B.D.X \mid C)$$

$$D_{closure} = B.(A.X.Y \mid A.C.Y \mid D)$$

$$X_{closure} = A_{ctx}.B_{ctx} = A.B.(B_{ctx}.X_{ctx} \mid C_{ctx}).(A_{ctx}.Y_{ctx} \mid D_{ctx}) =$$
$$A.B.(A_{ctx}.Y_{ctx} \mid D_{ctx} \mid C_{ctx}).(A_{ctx}.Y_{ctx} \mid D_{ctx}) =$$
$$A.B.(A_{ctx}.Y_{ctx} \mid D_{ctx} \mid C_{ctx}).(A_{ctx}.Y_{ctx} \mid D_{ctx})$$
$$A.(B.X.Y \mid B.D.X \mid C).B.(A.X.Y \mid A.C.Y \mid D) =$$
$$A.B.(B.X.Y.A.X.Y \mid B.D.X.A.X.Y \mid C.A.X.Y \mid B.X.Y.A.C.Y \mid B.D.X.A.C.Y \mid$$
$$C.A.C.Y \mid B.X.Y.D \mid B.D.X.D \mid C.D) = A.B.(X.Y \mid D.X.Y \mid C.X.Y \mid C.X.Y \mid D.C.Y \mid$$
$$C.Y \mid D.X.Y \mid X.D \mid C.D) = A.B.(X.Y \mid D.X.Y \mid C.X.Y \mid D.C.Y \mid C.Y \mid X.D \mid C.D) =$$
$$A.B.Y \mid A.B.D.Y \mid A.B.C.Y \mid A.B.D.C.Y \mid A.B.C.Y \mid A.B.D \mid A.B.C.D$$

$$Y_{closure} = A.B.X \mid A.B.D.X \mid A.B.C.X \mid A.B.D.C \mid A.B.C \mid A.B.X.D \mid A.B.C.D$$

Except the basic rules of the R-algebra the closure derivations does not expand any dependence's closure twice in the same recursion branch. Further, it uses the fact that the closure's expression does not contain the fragment for which the closure is being derived.

The *ideal morph model* for a set of fragments is constructed by joining their closures. Such a morph model fully satisfies all fragment dependencies. The ideal model for the fragments from the preceding example is derived in *Listing 57*.

Listing 57: Derivation of the ideal model from closures

```
Ideal(A,B,C,D,X,Y) = A.Aclosure | B.Bclosure | C.Cclosure | D.Dclosure | X.Xclosure |
Y.Yclosure = A.B.X.Y | A.B.D.X.Y | A.C | A.B.X.Y | A.B.C.X.Y | B.D | A.C.B.X.Y |
A.B.C.D.X.Y | A.C | A.B.D.X.Y | A.B.C.D.X.Y | B.D | A.B.X.Y | A.B.D.X.Y |
A.B.C.X.Y | A.B.D.C.X.Y | A.B.C.X.Y | A.B.D.X | A.B.C.D.X |
A.B.X.Y | A.B.D.X.Y | A.B.C.X.Y | A.B.D.C.Y | A.B.C.Y | A.B.D.X.Y | A.B.C.D.Y =
A.B.X.Y | A.B.D.X.Y | A.C | A.B.C.X.Y | B.D | A.B.C.D.X.Y | A.B.D.X |
A.B.C.D.X | A.B.C.D.Y | A.B.C.Y
```

It may be easily verified that the ideal morph model satisfies fully all fragments' dependencies by factoring out the verified fragment:

```
A: A.(B.X.Y | B.D.X.Y | C | B.C.X.Y | B.C.D.X.Y | B.D.X | B.C.D.X | B.C.D.Y |
B.C.Y) = A.(B.X.(Y | D.Y | C.Y | C.D.Y | D | C.D) | C.(1 | B.D.Y))
```

The result expression implies that if an alternative from the ideal model contains A, it also contains B.X or C, on which A depends. The underlined fragments indicate the dependencies of A.

The fragment closure model plays an important role when validating dependencies of fragments in a morph model. The dependency validation procedure examines all fragments by constructing the closure model for each fragment and the fragment's context in the morph model by factoring out the fragment from the morph model as shown in *Listing 58*.

Listing 58: The dependency check of fragment A in model M verifies that all alternatives from A's closure model conforms to the A's context obtained by factoring out A

```
Aclosure = B.X.Y | C
M = A.B.X.Y.P | A.C.Q | E = A.(B.X.Y.P | C.Q) | E
```

The A's context in the model *M* is *B.X.Y.P|C.Q*. The dependency checking procedure then determines whether the fragment's context in *M* "conforms" to A's dependency closure. The notion of the morph model conformance is explained in more detail in the section 5.1.9, however, for the purpose of this paragraph it is sufficient to outline it as follows: a morph model *A* *surjectively conforms* to model *B* if all alternatives from *B* may be substituted by some alternatives from *A*; and *A* *injectively conforms* to *B* if all alternatives from *A* may substitute some alternatives in *A*. And *A* conforms to *B* *totally* if it conforms both surjectively and injectively.

It follows immediately from the above that the contexts of the fragments from the ideal model totally conform to the fragments' closures.

Dimensions

Groups of mutually exclusive models are called *dimensions*. For example in the first case study it is possible to identify two such dimensions **Shape** and `Material`.

```
Shape ::= Rectangle | Cylinder
Material ::= Metal | Paper
```

If the fragments have the same common dependency it is possible to factor it out prior to the group, as shown in the following formula.

```
A.B | A.C = A.(B | C)
```

This common model may be seen as the abstraction, or the super-model, of the exclusive models in the group. Following this notion the **Shape** model can be considered a super-model of `Material` and vice versa.

Dimensions can also be organized hierarchically. The following formula represents a hierarchy of three groups **A**, **B** and **Y**.

```
A | B.(X | Y.(S | T)) = A | B.X | B.Y.S | B.Y.T
```

Wrappers

A wrapper is a special kind of fragment that overrides the behavior of another fragment. The wrapper has at least one dependency referring to the wrapped fragments. Besides this special dependency the wrapper may have plain (i.e. not overriding) dependencies used by the logic in the overridden methods.

The snippet in *Listing 59* shows how a wrapper specifying one overriding and one plain dependency can be implemented in Scala. The wrapped fragments are specified in the `extends` list, while the plain dependencies are specified as the self-type. The `overrides` keyword indicates that the `aMethod` method overrides the `aMethod` method inherited from A. The overriding functionality uses the `bMethod` inherited from the plain dependency B.

Listing 59: A wrapper with one overriding (B) and one plain dependency (A)

```
trait WrapperOfA extends A {
   this: B =>

   overrides def aMethod(x: Int): Int = super.aMethod(bMethod(2*x))
}
```

Wrappers are stackable fragments, which means there may exist more wrappers of the same fragment in an alternative; it follows, however, that the order of such wrappers in the alternative is significant, as the order may influence the result of the processing of the pipeline made up of the wrappers in the alternative.

Wrappers are not meant to contain any publicly accessible property or method in addition to the wrapped fragment. Therefore it makes no sense to specify a wrapper as a dependency of another fragment, since it would have the same effect as specifying the wrapped fragment as the dependency; the wrapper and the wrapped have the same interface.

5.1.4   Null Recognizer

The null recognizer is an extreme type of recognizer that recognizes no object by definition. Therefore its resolution is 0; i.e. the morph model of the null recognizer is empty (Figure 28).
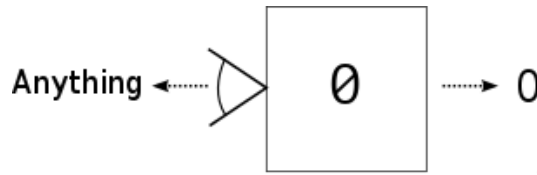
Figure 28: Schema of the Null recognizer

The null recognizer may be also seen as a special fragment recognizer recognizing "the absence of anything" feature. The symbol of this "nothing" fragment and its representation is 0. A morph model consisting of the 0 fragment only has the following characteristics:

```
A.0 = 0
A|0 = A
```

Furthermore, the null recognizer is also related to a special situation, in which a fragment depends on its inverse fragment; i.e. A<=>~A. To satisfy this dependency the morph model alternative must contain the following fragment formation, which is by definition 0; i.e. it never happens.

```
A.~A=0
```

5.1.5   Unit Recognizer (Universal Detector)

The unit recognizer recognizes any original object. However, it is not able to recognize anything else than the existence of the object (Figure 29). Therefore it is also called a *universal detector*. The resolution of the unit recognizer is 1.
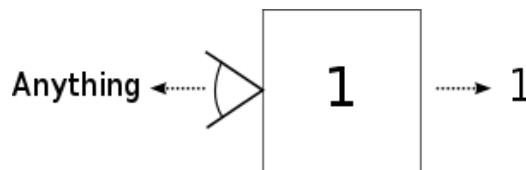
Figure 29: Schema of the Unit recognizer

The representation of the original object always consists of the *unit object*, which may be considered the stub (i.e. the instance) of a special *unit fragment* representing any object holding no information. The symbol of both the fragment and representation is 1.

The unit object itself is a special object that holds no information. It has neither behavior nor properties (its existence is rather a quantifier than a property). Although it makes sense to consider the existence of more such objects, they are inherently indistinguishable and therefore identical. Therefore the cardinality of the set of all such objects is 1. This set is called the *unit set* (or singleton set).

In the R-algebra the following rules hold:

```
A.1 = A
```

It is shown in 5.1.6 that in a composite exclusive recognizer the unit recognizer also functions as the inverse (or complementary) recognizer to the other partial recognizers in the exclusive recognizer.

5.1.6   Inverse Recognizer

Every recognizer may be associated with its *inverse recognizer* functioning as the antidote of the original recognizer. The inverse recognizer produces nothing (i.e. 0) when the original recognizer reacts on the input object. And conversely, the inverse recognizer produces the unit object (i.e. singleton 1) when the original recognizer does not react. The schema of the inverse recognizer is in Figure 30.
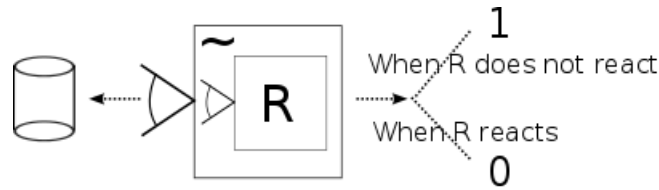
Figure 30: Schema of an inverse recognizer. The original recognizer is R.

The inverse recognizer virtually works as a detector of the absence of the feature(s) recognized by the original recognizer; it actually translates nothing to something. Since there is no other information available than the signal of the absence of some feature in the input object, the representation produced by the inverse recognizer consists of the unit object only; the unit object holds no information except the signal of its existence (in this case the signal of the absence of the feature).

Regarding the special cases of recognizers, it turns out that the unit and null recognizer are inverse recognizers of each other.

Considering the assumption that every recognizer has its inverse recognizer, any inverse recognizer must also have its inverse recognizer. Such a double-inverse recognizer works essentially as a detector of the presence of the feature(s) recognized by the original recognizer; it detects the "absence of the absence" of the feature. Therefore it is important to emphasize that the double-inverse recognizer is not always identical to the original recognizer. The double-inverse recognizer is actually identical to the original recognizer only in two cases: if the original recognizer produces the unit object only (i.e. it is just a detector of some feature), or if it is the null recognizer.

The fact that the inverse recognizer recognizes the absence of the features recognized by its original recognizer is expressed by the following equation:

```
A.~A = 0
```

The symbol *~A* represents the *inverse morph model* of *A*. A couple of inverse model examples are in *Listing 60*.

Listing 60: Examples of inverse models

```
~(A | B) = ~A.~B
~(A.B) = ~A | ~B
~(A.B | B.C | A.C) = ~A.~B.~C
~(~A.~B.~C) = A or B or C
```

Furthermore, if *A* and *B* are mutually exclusive models, then the composition of the inverse model *~A* with *B* will result in *B*. The reason is that if *B* is recognized then *~A* is recognized too and since *~A* produces the unit object only, it inflicts no changes to the object representation produced by *B*. Under this condition the inverse model works as the unit model:

```
~A.B = B = 1.B, if A.B=0
```

The previous rule may be applied to mutually exclusive models, where the rule's condition is always fulfilled:

```
A | B = A.~B | ~A.B
```

More generally, the previous rule may be applied to any mutually exclusive models, as shown in the following formula.

```
A1 | A2 | A3 | … = A1.~A2.~A3.… | ~A1.A2.~A3.… | ~A1.~A2.A3.… | …
```

It also follows that a branch model in an exclusive model may be joined with any number of inverse models of other branch models:

```
A1 | … = A1.~A2 | … = A1.~A3 | … = A1.~A2.~A3 | …
```

Let us further investigate inverse models of composite models. The following two identities allow converting the inversion of co-occurring composite models into mutually exclusive inversions and vice versa; i.e. converting the inversion of mutually exclusive models into co-occurring inversions.

```
~(A.B) = ~A.B or A.~B or ~A.~B
~(A | B) = ~(A.~B | B.~A) = A.B or ~A.~B
```

Dependencies of Inverse Recognizer

Another interesting topic related to the inverse recognizer is the examination of the dependencies of its model. Similarly to fragments, even the inverse fragments may depend on other fragments or inverse fragments. Let us examine the following two mutually exclusive fragments A and B.

```
A => ~B
B => ~A
```

The dependencies express the exclusive character of A and B; the presence of A is conditioned by the absence of B and vice versa. It automatically follows, owing to the symmetrical character of dependencies, that the dependencies of both inverse fragments are:

```
~A => B
~B => A
```

5.1.7   Composite Recognizer

A key property of recognizers is their composability, which allows creating more complex recognizers from simpler ones.
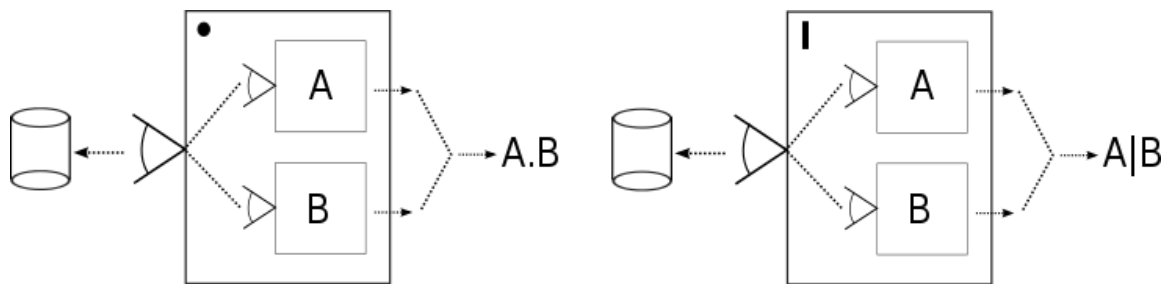
Figure 31: Schema of a composite recognizer composed of two recognizers A and B

The composite recognizer consists of at least two partial recognizers, which take part in recognizing of the original object Figure. The representation produced by the composite recognizer is composed of the representations produced by the partial recognizers.

The resolution of the composite recognizer depends on how the partial recognizers react on the original object. As long as they react simultaneously the resolution of the composite

recognizer is the product of all partial resolutions. In this case the composite recognizer behaves as if it contained the model *A.B*. On the other hand, if the partial recognizers are mutually exclusive, the composite resolution is the sum of all partial resolution and the recognizer behaves as if it contained the model *A|B*. Not to mention that besides these two extreme cases there may exist other mixed compositions.

Dependencies of Composite Recognizers

To elucidate the problem of dependencies of composite recognizers let us investigate a composite recognizer having a similar structure to the one in Figure with the exception that the two partial recognizers are also composite ones consisting of mutually exclusive fragment recognizers such as *Metal.Paper* and *Rectangle.Cylinder* from the case study. Let us further assume that the fragments `Metal` and `Paper` depend on `Rectangle` or `Cylinder` and vice versa as shown in Figure 32.
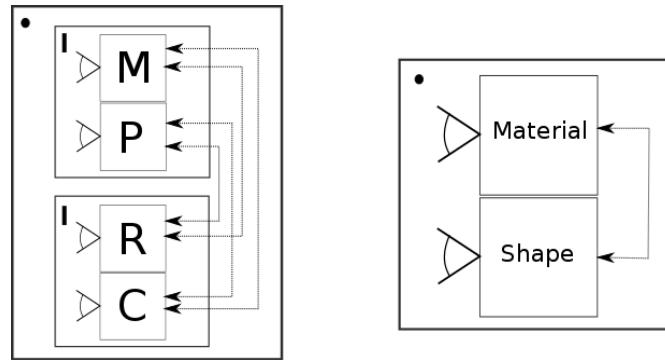
Figure 32: a) Dependencies between two composite recognizers that are parts of another composite recognizer, b) the two composite recognizers may be seen as dependent on each other

The schema in Figure 32a may also be interpreted as if any fragment depended on the common abstraction of the fragments from the opposite group; for example the `Metal` fragment would depend on the `Shape` dimension; instead on `Rectangle` or `Cylinder`. Therefore it is possible to lower the granularity of the schema by ignoring the internal structure of the two partial composite recognizers `Material` and `Shape`, which can be seen now as two mutually dependent monolith recognizers; i.e. reacting simultaneously (Figure 32b). To put it more generally, the composite recognizer inherits the unsatisfied dependencies from the partial recognizers.

```
Material => Shape
Shape => Material
```

5.1.8   Evolving Recognizers

The recursive composition illustrated in the preceding section may also be seen as an evolution of a recognizer, in which the old recognizer is replaced by a new composite one that

includes the old one. This evolution may be taking place at all levels of the composite recognizer leading to an improvement of both the recognition capabilities and the overall information value produced by the top recognizer.

The following paragraphs present a number of evolutionary steps that may be used to evolve a recognizer, resp. its morph model. It is assumed throughout the following text that all partial recognizers belonging to the same composite recognizer are either mutually exclusive or co-occurring; the mixed cases are not considered for the sake of simplicity and clarity.

Default Recognizer

Let us begin with the default recognizer that is going to be evolved into a more complex one. Let us suppose that the default recognizer is only able to detect any object and the issued representations do not carry any additional information. Such a description corresponds to the unit recognizer producing the unit object as the only representation.

```
1
```

In UML, the unit recognizer may be symbolized by a class labeled by the `singleton` stereotype, which suggests that there is only one instance of this class (Figure 33). The unit class declares no members to indicate that there is no information held in the single instance.
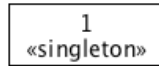
Figure 33: The UML representation of the unit singleton

Union

The union evolutionary step assumes the existence of a recognizer $A(x)$ that will be replaced by a composite recognizer consisting of the $A(x)$ and another recognizer $B(x)$, whose model B is mutually exclusive to A.

```
Original model:  A
Added model:     B, A.B = 0
New model:       A | B
```

The resulting composite recognizer will then behave according to the model $A \mid B$ and its resolution is simply the sum of the partial resolutions.

```
Res(A | B) = Res(A) + Res(B)
```

The corresponding UML diagram (Figure 34) consists of two classes implementing the lowest upper bound type (see 4.1.10).
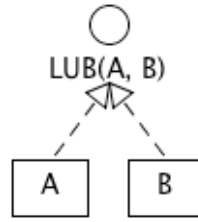
Figure 34: UML representation of a union

To illustrate the union step let us suppose that the existing recognizer **Paper**(x) is going to be accompanied by the **Metal**(x) recognizer. Since the new fragment **Metal** is obviously mutually exclusive with the existing fragment **Paper**, the condition for the union step is fulfilled.

```
Paper.Metal = 0
```

The Paper(x) recognizer is replaced by a composite recognizer containing the model Paper | Metal. The model of the top recognizer then will be:

```
Paper | Metal
```

The LUB of the two fragments may be identified as **Material** containing the density property, which is recognized by both fragments. The corresponding UML diagram is in *Figure 35*.
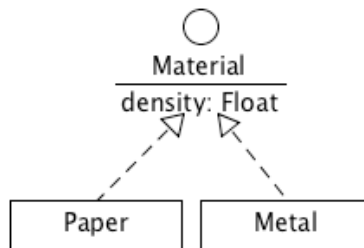
Figure 35: UML model of the union of Paper and Metal

Join

The join evolutionary step assumes the existence of a recognizer **A**(c) that will be replaced by a composite recognizer consisting of **A**(x) and another recognizer **B**(x), whose model **B** is (mutually) co-occurring with **A**. This condition may be expressed by formula Res(**A.B**) = Res(**A**).Res(**B**).

```
Original model:  A
Added model:     B, Res(A.B) = Res(A).Res(B)
New model:       A.B
```

The resulting composite recognizer will then behave according to the model **A.B**.

To illustrate the join step let us suppose that the recognizer recognizing paper or metal objects is joined with a new recognizer recognizing two shapes: rectangles and cylinders. The morph model of the resulting composite recognizer will then be:

```
(Paper | Metal).(Rectangle | Cylinder)
```

The corresponding UML diagram is in Figure 36.
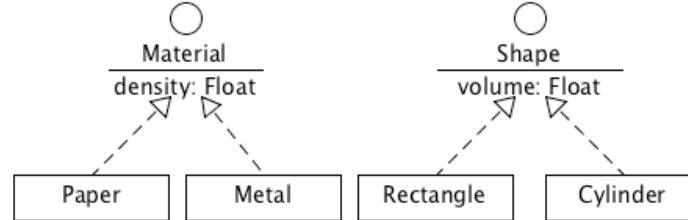
Figure 36: UML model of the join of two unions `Material` *and* `Shape`

The objects instantiated according to the schema above will be instances of any combination of any type from `Material` and any type from `Shape`. Here, however, one may spot a significant difference between the traditional OO modeling and the concept of recognizers. An object in OOM is usually born as an instance of a given combination of the parent types, whereas this configuration persists for the rest of its life; in other words, the object is not expected to change its implementation after its creation.

On the contrary, it is the primary assumption of the concept of abstract recognizers that an object, when created, may assume one of the forms described by its morph model and, in addition, it may morph to another valid form during its life. To shed some light on this idea, let us suppose that the recognizer driven by the model from this paragraph recognizes a paper rectangle in the original object. The representation returned by the recognizer is an instance of the `Paper` and `Rectangle` types. Now let us assume that the paper rectangle is rolled up to create a cylinder. Although the identity of the object has not changed, the object has ceased to be a rectangle and has morphed into a cylinder. It follows, thus, that the object has changed one of its parent type from `Rectangle` to `Cylinder`.

$t = T0 \qquad t = T1 > T0$
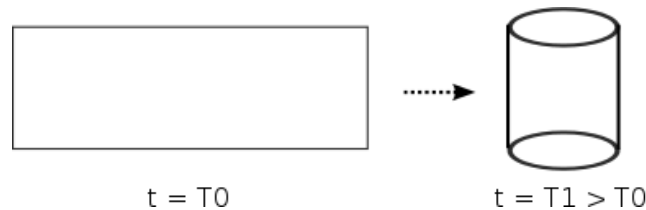
Figure 37: The mutation a rectangle to a cylinder preserves the identity

The transformation is schematically shown as follows:

```
T0: Paper(d).Rectangle(w,h) -> T1: Paper(d).Cylinder(w/(2*pi),h)
```

The attributes of the cylinder fragment are calculated from the rectangle attributes.

Extension

The extension evolutionary step assumes an existing recognizer *A*(x) that will be replaced by a composite recognizer consisting of the original recognizer *A*(x) and another recognizer *B*(x), whose model *B* depends on *A*.

```
Original model:  A
Added model:     B => A
New model:       A | A.B = A.(~B | B) = A.(1 | B)
```

The resulting composite recognizer will behave according to the model *A.(1 | B)*, as shown in Figure 38. The extension relationship is actually analogous to the inheritance relationship in the traditional UML.

The resolution of the new model is given by the following formula.

```
Res(A.(1 | B)) = Res(A) + Res(A).Res(B)
```

The new fragment B may be considered an extension of the fragment A, as the representations of B always occur side-by-side the representations of A. The joined two fragments A.B model a specialized version of the object modeled by A only.
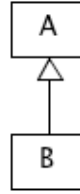
Figure 38: UML representation of an extension is analogous to the inheritance relationship between B and A

In contrast to the union and join steps the extension step is not atomic since it consists of two steps: one union followed by one join:

```
1. B -> 1 | B
2. A -> A.(1 | B)
```

To illustrate the extension step let us assume that the default recognizer is going to be enhanced by the *Paper*(x) recognizer. Although essentially independent the Paper fragment depends, as all fragment, on the unit fragment.

```
Paper => 1
```

The model of the new composite recognizer is

```
1 | Paper = ~Paper | Paper
```

The UML diagram equivalent to the new morph model uses the extension relationship, as shown in Figure 39.
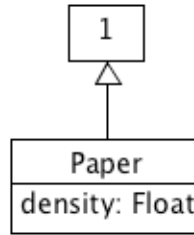
Figure 39: UML model of the optional `Paper` *fragment*

If the other fragments `Metal`, `Rectangle` and `Cylinder` are extended from `1` as well, then the model of the top recognizer from the previous paragraph will look like this:

```
(1 | Paper | Metal).(1 | Rectangle | Cylinder) = (1 | Material).(1 | Shape)
```

The corresponding UML diagram is in Figure 40.
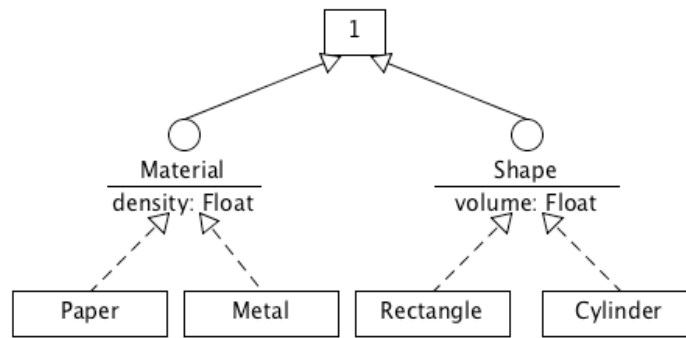
Figure 40: The scanned item's UML model in which all fragments are optional

In the paragraph dedicated to the join transformation it is shown on the wrapping-up a rectangle into a cylinder that an object need not to be stuck in the type hierarchy; in other words it may replace one parent type (`Rectangle`) with another mutually exclusive one (`Cylinder`). A similar situation may happen in the case of extension; after all the extension is a composition of the union and join steps.

Aspects

The model of the joined recognizer *(1 | Material).(1 | Shape)* may be also written as a union of four models *1*, *Material*, *Shape* and *Material.Shape*.

```
(1 | Material).(1 | Shape) = 1 | Material | Shape | Material.Shape
```

The fourth partial model *Material.Shape* is especially interesting since it brings new combinations of fragments into the top model. This partial model may be subject of all evolutionary steps described in this section. It may also be seen as a new type introduced in the

system. An example of an evolution of *Material.Shape* can be joining it with the `Body` fragment that uses the members from both models *Material* and *Shape* to calculate the weight of the modeled object as indicated in Figure 41.
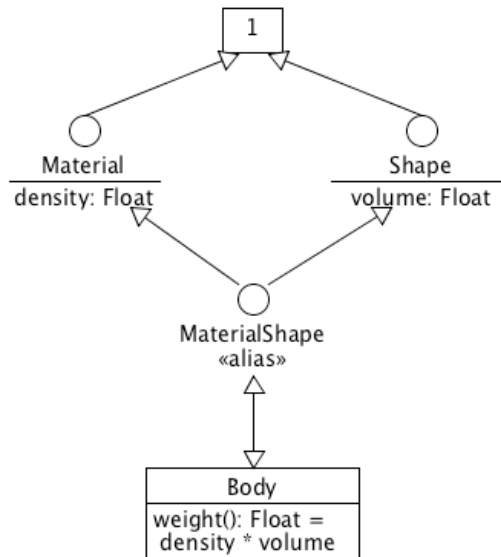
Figure 41: The Body fragment joining the material-shape model
through the `MaterialShape` *alias type*

There is a new type in the diagram – `MaterialShape` – which, however, does not extend the types `Material` and `Shape` in the true sense of the word; instead, it establishes an alias for the joined type, as indicated by the `alias` stereotype.

The UML symbol for the join operation uses a modified version of the symbol for extension relationship with two arrows indicating the co-occurrence of the types connected with the relationship.

The resulting model of the top recognizer is then:

```
1 | Material | Shape | Material.Shape.Body =
1|Paper|Metal|Rectangle|Cylinder| Body.(Paper|Metal).(Rectangle|Cylinder)
```

Generally, the join transformation may be considered a crosscutting operation affecting all alternatives in the transformed model. The `Body` fragment is in fact an *aspect*, which introduces the new `weight` calculated property in all alternatives of model *Material.Shape*, however, the *Body*(x) fragment recognizer does not recognize anything new by itself.

Categorization (Inverse Extension)

The categorization evolutionary step assumes the existence of a recognizer *A*(x) that will be replaced by a composite recognizer consisting of *A*(x) and another recognizer *B*(x), whereas the model *A* becomes dependent on *B*.

```
Original model:        A
Added model:           B => 1 | A
Upgraded orig. model:  A => B
New model:             B.(1 | A)
```

The resulting composite recognizer will then behave according to the model *B.(1 | A)* and its resolution is given by the formula:

```
Res(B.(1 | A)) = Res(B) + Res(A).Res(B)
```

The corresponding UML diagram is in Figure 42 and is in fact the inverse version of the extension diagram.

Figure 42: UML representation of categorization

The categorization usually introduces new fragments to a group of existing fragments. This evolutionary step often happens when the recognizer is composed of many particular fragment recognizers, which may be grouped according to some shared properties represented by the newly introduced recognizers.

To illustrate this situation, let us assume that the Shape model contains four mutually exclusive fragments: Circle, Rectangle, Cylinder and Cuboid. The four fragments may be clearly split into two groups reflecting the number of occupied spatial dimensions. This enforced categorization of the Shape model may be expressed by the following formula:

```
Shape = TwoDim.(Circle | Rectangle) | ThreeDim.(Cylinder.Cuboid)
```

The corresponding UML diagram is in Figure 43.
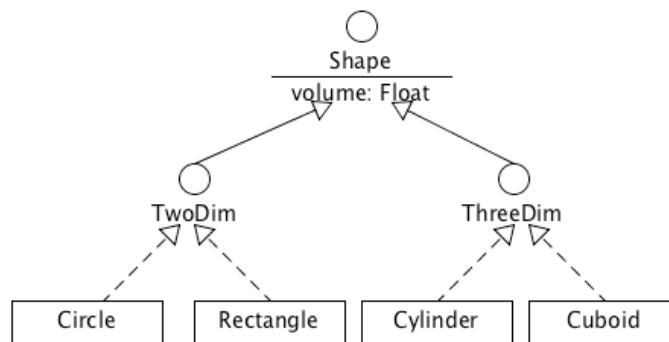
Figure 43: Introducing two categories TwoDim and ThreeDim into the existing hierarchy

5.1.9   Morph Model Conformance (Liskov Substitution Principle in OM)

The evolution of a recognizer may affect the applications using the recognizer and therefore it is important to examine the question of the backward compatibility of morph model changes. This issue is an analogy to the Liskov substitution principle (LSU) in object morphology; the goal of this section is to investigate the conditions under which one morph model may substitute another.

Let us call the substituted version of the morph model used by the application the *target morph model*, while the substituting version will be called the *source morph model*. The two models may differ in the resolution and in the composition of individual alternatives.

Conforming Fragments

Prior to the examination of the relationship between the two models it is essential to define the conformity between two fragments. Let `F` and `F'` be two fragments. Then the source fragment `F'` *conforms* the target fragment `F` if `F'` can substitute the fragment `F` in terms of LSU.

Conforming Alternatives

The notion of conforming alternatives may be directly derived from the notion of conforming fragments. The source alternative `A'` *conforms* to the target alternative `A` if

1. For all target fragments from `A` there is a conforming fragment in the source alternative `A'`.

2. And the other target fragments not included in `A` but having their counterparts in `A'` are not mutually exclusive with fragments from `A`.

The Figure 44 illustrates a conforming alternative scenario on the left, while on the right is shown a non-conforming source alternative, which conforms to neither `T1.T2` nor `T3.T4` because it contains the counterparts of both mutually exclusive target alternatives.

Figure 44: Conforming (left) and non-conforming alternatives (right)

As another illustration of the definition let us consider the following two models consisting of the same fragments `A` and `B`.

```
A.B
A | B
```

Here both models are mutually non-conforming.

Conforming Morph Models

The above-mentioned definition ensures that no source alternative may conform to more than one target alternative. Otherwise there might exist two alternatives (which are always mutually exclusive), whose fragment counterparts would sit side-by-side in a single source alternative. This situation would violate the exclusivity of the two target alternatives. Therefore the conformity relation between the source and target models is a function:

```
Conf: S -> T
```

There are five basic scenarios of how the source model can substitute the target one. In the first examined scenario all target alternatives may be substituted by one or more source alternatives (Figure 45). Therefore this situation is labeled as the *surjective conformance*.
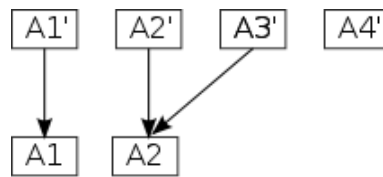
Figure 45: Surjective conformance

The surjective conformance is required as long as the client application module uses a recognizer as a sort of factory to create morphs described by the target model. In such a case the source model recognizer may substitute the target model recognizer used by the application module. The new recognizer will continue to operate in a compatible way within the application module; i.e. it will be creating morphs conforming to those produced prior to the substitution, although the new morphs may contain additional "hidden" stubs of fragments from the source model. The non-mapped alternatives in the source model (`A4'`) will never be used.

An example of a surjective conformance is shown in *Listing 61*.

Listing 61: An example of two surjectively conforming models

```
T: A | B
S: (X.A | 1).(Y.B | 1) = X.A.Y.B | X.A | Y.B | 1
```

From the four source alternatives only two conform to the target alternatives.

```
X.A -> A
X.B -> B
```

In the next studied situation it is all source alternatives that have their counterparts among the target alternatives, while not all target alternatives are involved in the mapping (Figure 46). Such a situation is labeled as *injective conformance*.
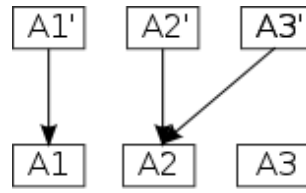
Figure 46: Injective conformance

The source model must be fully conforming to the target model as long as the client application module consumes the representations produced by the target model recognizer, while the application does not actively uses the recognizer; i.e. as a factory. A morph is created by the source model recognizer and passed to the consuming application module; its composition corresponds to one alternative in the source model. Therefore it is necessary that for all source alternatives there be a target alternative, since otherwise the source recognizer might generate a representation that would not conform to any target model alternative and the client application module could not consume it.

The injective conformance also allows the consuming application module to re-morph the source model morph passed as an argument, as the new form of the morph will always be conforming to some target model alternative.

Listing 62 shows an example of an injectively conforming source model; all source alternatives conform to some target alternatives.

Listing 62: An example of two injectively conforming models

```
T: (A | 1).(B | 1) = A.B | A | B | 1
S: X.A | Y.B
```

The third situation combines the two previous ones. It is labeled as the *total conformance*, as all source and all target alternatives are part of the relation.
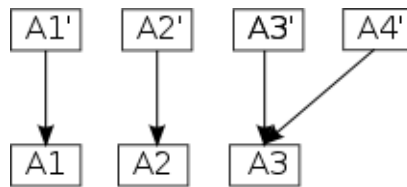
Figure 47: Total conformance

The total conformance is necessary if the client application module uses actively the recognizer to create new representations and consumes and possibly re-morphs other representations described by the same target model at the same time. Such an application may be for example a processor in a pipeline processing and transforming morphs (i.e. representations).

An example of a total conformance is in *Listing 63*.

Listing 63: An example of two totally conforming models

```
T: A | B
S: X.A | Y.B
```

In the next situation neither model is fully covered by the conformity relation, however, there is at least one source alternative conforming to a target alternative (Figure 48). This scenario is called a *partial conformance*.
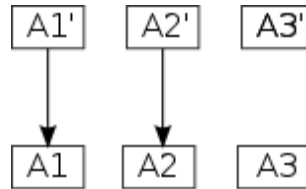
Figure 48: Partial conformance

The client application must be able to recuperate from the following conditions: First, in case the application module uses a partially conforming recognizer as a factory it will not obtain a morph for some target alternatives (such as **A3**), since the substituting recognizer cannot produce any conforming representations for those target alternatives. Second, in case the application module consumes morphs produced by the substituting recognizer, it could get a non-conforming representation (such as **A3'**). This situation can be detected at compile-time and abort the compilation.

The source model in *Listing 64* is neither surjectively nor injectively conforming.

Listing 64: An example of two partially conforming models

```
T: A | B
S: A.X | Y
```

The application may fix this problem by evolving its (target) model to the *optional* version:

```
T | 1
```

The new unit fragment alternative will glean all source alternatives, which did not conform to any target alternative. This modification will then effectively turn the erroneous situation into the injective conformance.
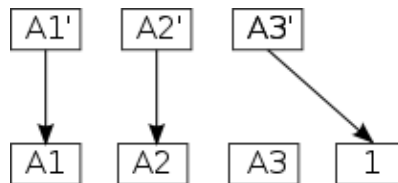
Figure 49: Fixing partial conformance to injective one by introducing the singleton alternative (1)

And the last conformance situation occurs if there is no alternative in the source model that conforms to some target alternative. This situation is considered a compile-time error.

Morph References

In order to allow passing recognizers and morphs across various application modules the OM platform introduces special kinds of references. Each kind of reference corresponds to one of the valid situations described above; i.e. total, surjective or injective. The client application module uses these references to declare the intended usage of the source model. On every occurrence of such a reference the compiler checks whether the source model conforms to the target model correspondingly.

A surjective reference is used to refer to a recognizer, while the injective reference is used to refer to a morph (representation). A total and partial reference may refer to either one.

The section 7.3.6 deals with the references in more detail.

Checking Fragment Dependencies

As already mentioned in Fragment Dependencies the fragment closure model is used to check the dependencies of fragments within a morph model. The dependency checking procedure just constructs the context of a fragment in the containing model and verifies whether the context model rightly conforms to the fragment's closure model. To illustrate the procedure let us examine the dependency model of the fragment A defined as follows:

```
A => B | C
A_closure = B.X | C.Y
```

The closure model of A also contains "hidden" fragments X and Y brought by B and C. Some morph model examples conforming to the closure are in *Listing 65*; the model *M1* conforms $A_{closure}$ surjectively, *M2* injectively, *M3* totally and *M4* partially.

Listing 65: Examples of models conforming to $A_{closure}$

```
M1 = A.(B.X.Z | C.Y | E) | F;   A_ctx = B.X.Z | C.Y | E;   A_ctx ~ A_closure surjectively
M2 = A.B.X.Z | F;               A_ctx = B.X.Z              A_ctx ~ A_closure injectively
M3 = A.(B.X.Z | C.Y) | F;       A_ctx = B.X.Z | C.Y        A_ctx ~ A_closure totally
M4 = A.(B.X.Z | E) | F;         A_ctx = B.X.Z | E          A_ctx ~ A_closure partially
```

Every fragment declares the so-called *conformance level*, which specifies the kind of the conformance relationship between the containing model and the fragment's dependency closure. The *default conformance level* of fragments is the injective conformance.

The compiler uses the conformance level of fragment to verify whether the dependencies all fragments in a morph model are satisfied in the required level. If, for example, the fragment A from the example declared its conformance level as surjective, then only models *M1* and *M3* would satisfy its dependencies.

Target Models With Placeholders

It is considered a compile-time error as long as the target model contains some fragments that have no counterparts in the source model. However, the client application module may mark certain fragments in the target model, specified in a reference, as *placeholders*. If a fragment is marked as a placeholder then it does not have to have its counterpart in the source model.

The dollar symbol `$` is used to distinguish placeholders from other fragments.

```
$A
```

The symbol `$` functions as an operator switching on or off the current placeholder status of its operand. The default placeholder status of a fragment is off. The following figure shows how the placeholder operator distributes over a morph model expression:

```
$(A | $B) = $A | $$B = $A | B
```

Applying the placeholder operator twice on `B` results in the non-placeholder `B`.

This notation allows marking not only particular fragments as placeholders, but also more complex sub-expressions or whole models.

```
M = A | B.C | D
$M = $A | $B.$C | $D
```

Unlike the other target fragments the placeholder fragments are supposed to be instantiated by the application module itself and integrated with the source morphs.

Since the precondition, which assumes that all target fragments have their counterparts in the source model, no longer holds, it is necessary to take the placeholders into account in the definition of conforming alternatives.

The source alternative **A'** *conforms* the target alternative **A** if

1. For all *non-placeholder* target fragments from **A** there is the conforming fragment in the source alternative **A'**.

2. And the other target fragments not included in **A** but having their counterparts in **A'** are not mutually exclusive with fragments from **A**.

3. And all placeholder fragments in **A** can be *integrated* with the fragments in the source alternative **A'**.

The meaning of the integration of placeholders with source model alternatives is explained in the following section.

Integrating Placeholders with Source Model Alternatives

The distributive property of the placeholder notation is especially useful when the target morph model consists of placeholders only. It does not follow, however, that such a target model is completely independent of any source morph model; for example, some placeholder fragments may have dependencies, which are not satisfied within the target model and it is supposed that the source model delivers the dependencies; some placeholders may also replace their counterpart fragments in the source model alternatives or others may be inserted into the source alternative as wrappers of some source fragments. This process of mixing placeholders and the source fragments is called the *placeholder integration*.

The result of the integration of the placeholders from the target model into the source model is called the *integrated model*. It is basically the source model with the placeholders placed at the right places. The integrated model must be validated in terms of checking dependencies of the integrated placeholders. This procedure is done as described in Checking Fragment Dependencies.

There are essentially two basic types of the placeholder integration: *replacement* and *insertion*.

The replacement occurs if the placeholder has its counterpart in the source model.

```
T: $A | B
S: A.X | B.Y
I: A.X | B.Y
```

The replacement does not change the source model; i.e. the integrated model is the same as the source model, however, the recognizer using the integrated model produces morphs, in which the placeholder stubs substitute the stubs of the placeholder's counterpart.

The insertion occurs, if the placeholder has no counterpart in the source model. The integrated model thus contains additional fragments with respect to the source model.

```
T: $A | B; A => X
S: X | B
I: A.X | B
```

The recognizer using the integrated model produces morphs enhanced by the target model placeholders.

Coupling Morph Models

A source model used as a substitution may also be substituted by another model via recognizer or morph reference. Provided that there are four essential types of references (partial, injective, surjective and total) there are 16 possible reference-reference combinations, which are examined in the following paragraphs.

There are three models in each scenario and a pair of conformance relationships given by the name pattern *FirstRel-SecondRel*. The first relationship refers to that between the top and middle models, where the middle one plays the role of the target model. The second relationship refers to that between the middle and bottom models, where the bottom one is now the target model. The context of the examination is the place in the program where the second reference (i.e. *SecondRel*) is declared; it means that the middle and the bottom models are supposed to be completely known and therefore it is possible to verify whether the middle model conforms accordingly to the bottom one. On the contrary, the top model is unknown; the only thing, which is known, is that the top model conforms to the middle model in accordance to the first relationship.

The goal is to prove that the composition of the first and second relationships between the top and bottom models is always satisfied under the assumption that:

- The middle and bottom models are known and the relationship between them is satisfied

- The top model, although unknown, satisfies the first conformance relationship.

Let us begin with the *Partial-Partial* combination, which requires that there are at least one alternative in the top, resp. middle, model that conforms an alternative from the middle, resp. bottom, model (Figure 50).
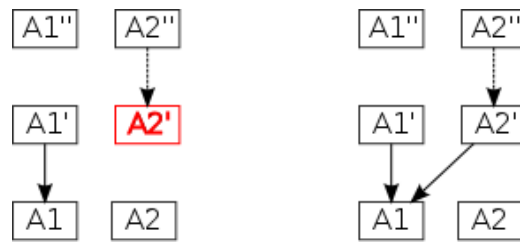
Figure 50: An invalid (left) and a valid (right) Partial-Partial coupled substitution

It is easy to find a case, in which both relationships are satisfied, however, their composition fails. The middle-bottom and top-middle partial relationships in Figure 50(left) are both satisfied, nevertheless, the composition of the two relationships is broken, as the alternative **A2'** does not conform to any alternative from the bottom model; there is no path from the top to the bottom.

However, in case the second relationship is in fact injective, as shown in Figure 50(right), it is possible to find a path from **A2''** to **A1** and thus the requirement for the partial relationship will be satisfied. Therefore, since the validator knows exactly the bottom and the middle models it is able to perform an additional check whether the relationship between the middle and bottom models is actually injective. If it is the case the composite relationship will be verified. The same conclusion holds for the combination *Injective-Partial*, which is just a special case of *Partial-Partial*.

The *Partial-Injective* and *Partial-Total* combinations are never valid since they require that the middle model be fully addressed by the top model. This is, however, not possible to check, as the top model is not known to the validator and therefore the validator must expect the worst case.

Similarly, the *Partial-Surjective* combination is not valid either, as the validator cannot verify that the alternatives in the middle model, to which the top model conforms, coincide with those conforming to the bottom model.

The combinations *Surjective-Partial* and *Total-Partial* are always valid, since all middle model alternatives are addressed by some top alternatives and thus the situation described in Figure 50(left) may never happen.

On the other hand, the *Injective-Injective* combination is always valid (Figure 51). In this case the target application module consumes a morph created by a recognizer using the top model.
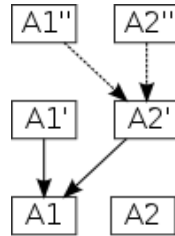
Figure 51: Injective-Injective scenario

All alternatives from the top model conform to a subset of alternatives from the middle model. Coupled with the fact that all middle model alternatives conform to a subset of the bottom alternative, it may be concluded that there must exist a path from any top alternative to some bottom alternative; in other words the composition relationship is injective.

Similarly, the *Surjective-Surjective* combination is also always valid. In this scenario, the target application module consumes a recognizer (factory) using the top model, which is passed through the intermediate second reference (Figure 52).
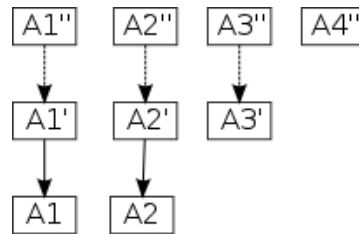
Figure 52: Surjective-Surjective scenario

In this scenario all bottom alternatives are addressed by some middle model alternatives. Since the top model conforms to all middle model alternatives there must be some top model alternatives that conform transitively to all bottom alternatives.

The *Injective-Surjective* combination is not automatically, however (Figure 53). Now the target module consumes a recognizer that created the morph referred by the first reference. The recognizer is obtained from the morph and then assigned to the second reference.
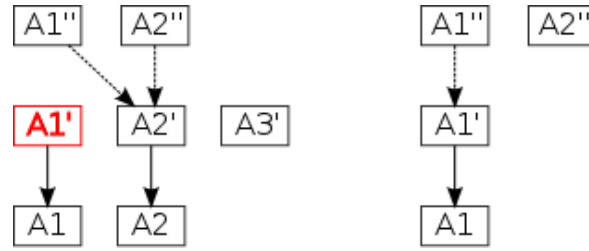
Figure 53: An invalid (left) and a valid (right) Injective-Surjective scenario

Unfortunately, not all bottom alternatives may be reachable from some top alternative, as indicated in Figure 53(left), where `A1` is addressed by `A1'`, which is, however, not addressed by any top alternative. Nonetheless, as long as the middle and bottom model consists of only one alternative (Figure 53(right)), the only bottom alternative must be addressed by the only middle one, which must be addressed by some top alternative, as the top model conforms injectively, which guarantees at least one conforming top alternative. This condition may be checked by the validator, since it knows the middle and the bottom model and can see their resolutions.

In the *Surjective-Injective* case the target module consumes a morph created by the recognizer referred by the first reference (Figure 54). The morph is brought into existence with the help of a morphing strategy working with the middle model and hence it is guaranteed that the morph will be an instance of no other alternative than those from the middle model. The new morph is then assigned to the second reference.
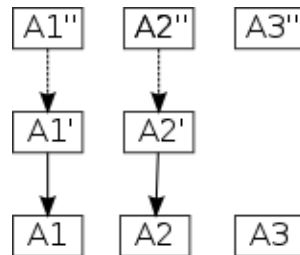
Figure 54: Surjective-Injective scenario

This situation is always valid in view of the fact that the morph will always be an instance of some conforming alternative from the middle model, whereas this conforming alternative will be addressed by some alternative from the top model, how is required by the recognizer creating the morph. It may never happen that the client re-morphs the morph into a non-conforming alternative from the top model, since the selection of the alternatives is driven by the strategy selecting the alternatives from the middle model.

The *Surjective-Total* case is one (Figure 55), which is also always valid and is similar to the preceding one. If the second reference refers to a recognizer, the target module receives

a recognizer working with a strategy choosing the alternatives from the middle model only. Similarly, if the second reference refers to a morph, the target module receives a morph created by the recognizer with the same strategy. Thus, whether the module uses the recognizer or the morph, both will be operating in the domain of the middle model thanks to the strategy.
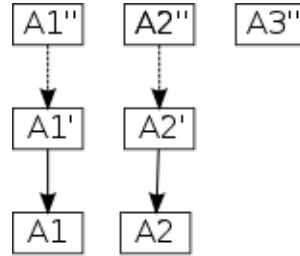
Figure 55: Surjective-Total scenario

The *Total-Total* case (Figure 56) is also always valid since all top model alternatives conform to all middle model alternatives, which conform to all bottom alternatives.
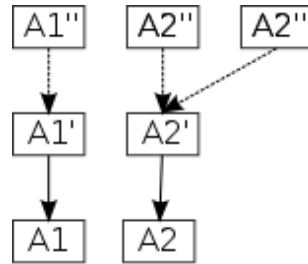
Figure 56: Total-Total scenario

The *Injective-Total* combination is never satisfied, as the total conformance between the middle and the bottom model requires that all middle alternatives be addressed by some top model alternatives; this condition is not fulfilled by the definition of the injective conformance.

The last two combinations *Total-Injective* and *Total-Surjective* are always satisfied since the top model totally conforms the middle model.

5.1.10   The Using Morphs in Client Applications

The client module used in the previous section is supposed to consume morphs or recognizers through a special reference. However, the text is not mentioning any detail on how such consumption should actually be carried out. This section provides a brief description of how such a client module might be developed.

Table 1 summarizes the results of the analysis.

5.1.11 Using Morphs in Client Applications

The client module used in the previous section is supposed to consume morphs or recognizers through a special reference. However, the text is not mentioning any detail on how such consumption should actually be carried out. This section provides a brief description of how such a client module might be developed.

Table 1: Summary of all conformance combinations

First Ref. Conf.	Second Ref. Conf.	Valid	Note
Partial	Partial	Yes in special case	The middle model must be injectively conforming to the bottom one
Partial	Injective	No	
Partial	Surjective	No	
Partial	Total	No	
Injective	Partial	Yes in special case	Same as Partial-Partial
Injective	Injective	Yes	
Injective	Surjective	Yes in special case	Single-alternative middle and bottom models only
Injective	Total	No	
Surjective	Partial	Yes	
Surjective	Injective	Yes	
Surjective	Surjective	Yes	
Surjective	Total	Yes	
Total	Partial	Yes	
Total	Injective	Yes	
Total	Surjective	Yes	
Total	Total	Yes	

A client module might be able to consume a wide range of morphs differing in the morph model according to which the morphs are created. Therefore the client needs to specify

which alternatives it is able to consume; in other words the client specifies the target morph model. Then the client declares a morph reference and uses the target model as the type argument of the reference.

Further, for each supported alternative the client has a dedicated placeholder fragment. Such placeholders have a common abstract parent fragment (placeholder's dimension) and thus the LUB of the reference contains also the placeholder dimension interface, as long as all alternatives are extended by one of such placeholders.

What is important is that the unsupported (unknown) alternatives are represented by the unit fragment 1, resp. a free placeholder bound to 1, which plays the role of the default handler.

The goal is to integrate the client's handling code encapsulated in the placeholders into the morph itself and to avoid the branching of the handling code completely. This approach is de facto a replacement of the switch statement (or match in Scala) and is reminiscent of the visitor pattern.

The code snippet in *Listing 66* is written in Scala and shows a client code consuming a morph, which may assume two forms: `Rectangle` or `Cylinder`. The type `ShapeRenderers` defines the target model of the morph reference. The target model contains three place-holders: two for the accepted alternatives and one as the default handler.

Listing 66: A client integrating its placeholders with the morph passed as the argument through the morph reference `shapeRef`

```
type ShapeRenderers = $[RectangleRenderer]*Rectangle |
$[CylinderRenderer]*Cylinder | $[DefaultRenderer]

def renderShape(shapeRef: &[ShapeRenderers]) {
   val shape = *(shapeRef)
   shape.render
}
```

The `renderShape` method declares the single argument, which is the morph reference. The body of the method consists of only two statements. The first statement dereferences the morph reference to obtain the "real" object reference to the shape morph. The type of the shape variable is the LUB of the `ShapeRenderers` model, which is `Shape.Renderer`; i.e. the common base for all placeholders used in this example. Thus the second statement may invoke the render method declared in `Renderer`.

The following snippet in *Listing 67* sketches the code, which passes the morph to the `renderShape` method.

Listing 67: Passing the morph to the client code

```
type Item = (Rectangle | Triangle | Cylinder | Cuboid)*(Paper | Metal)

val itemRecognizer = createRecognizer[Item]()
val item = itemRecognizer.recognize(new ItemRecStrategy(inputData))

renderShape(item)
```

First, the code declares the type of the morph, which is different from the model used in the consuming code. Next, a recognizer is created using the `Item` model and then used to create the morph with the help of the morphing strategy passed as the argument to the recognize method. The strategy instance gets the input data according to which it choses the right alternative from the `Item` model. The last statement invokes the consuming `renderShape` method and passes the morph.

The last statement is the place, where the compiler verifies the injective conformance of the `Item` model to the `ShapeRenderers` model.

5.2   Morphing 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. 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 a 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. To select the winner alternative the three strategies use distinctive approaches, which may be freely combined into a composite morphing strategy. The following paragraphs will shortly introduce these strategies.

5.2.1   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 the scanned item morph model consisting of four fragments:

```
(Metal | Paper).(Rectangle | Cylinder)
```

This model can be depicted as a tree shown in Figure 57.
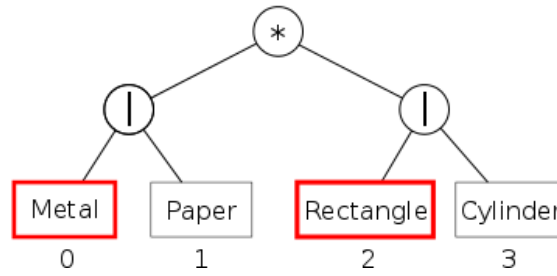
Figure 57: The default expression tree of the morph model

The model has two dimensions that correspond to the two disjunction operators (|) in the model. Each disjunction operator 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"*. 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}

The goal of the promoting strategy is to "promote" one or more independent fragments (i.e. fragments under different disjunction operators), 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.

In order to promote the `Paper` fragment, the promoting strategy must transform the model expression tree into the form depicted in Figure 58; in other words, to swap the `Paper` and `Metal` fragments.
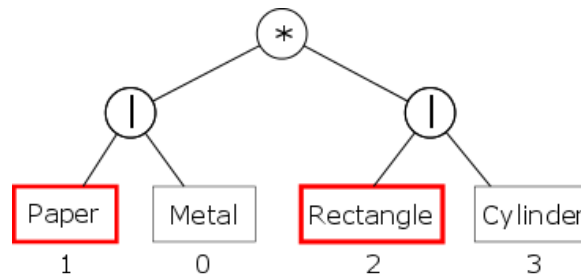
Figure 58: Promoting the `Paper` fragment

The new tree yields this list of alternatives.
1.**{1, 2}**
2.{0, 2}
3.{1, 3}
4.{0, 3}

We can continue by promoting the `Cylinder` fragment, as shown in Figure 59.
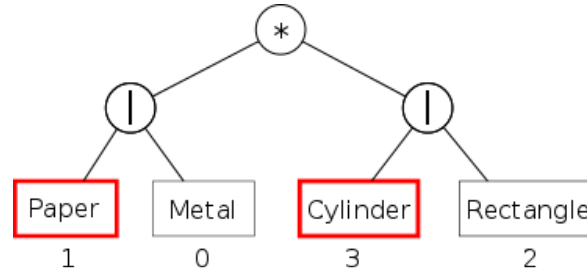
Figure 59: Promoting the `Cylinder` *fragment*

The corresponding list of alternatives is:

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

And finally the fig shows the promotion of the `Metal` fragment.
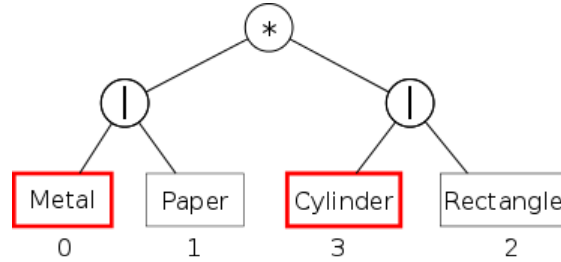
Figure 60: Promoting the `Metal` *fragment*

The list of alternatives is then as follows:

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

5.2.2   Masking Strategy

In contrast to the promoting strategy, which only reorders 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, RectangleBit, 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]
```

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 sub-list $S \subseteq A$ 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 ∈ 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.

```
    |1 0 1 0|
    |0 1 1 0|
F = |1 0 0 1|
    |0 1 0 1|
```

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.

```
          |1 0 1 0|
          |0 1 1 0|
F & mask = |1 0 0 1|
          |0 1 0 1|
```

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.

```
          |1 0 1 0|
          |0 1 1 0|
F & mask = |1 0 0 0|
          |0 1 0 0|
```

Here, only the first two rows remain same after the mask is applied. The first alternative in the sub-list, 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]
```

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:

```
            |1 0 1 0|
            |0 0 1 0|
  F & mask = |1 0 0 0|
            |0 0 0 0|
```

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

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

5.2.3   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
```

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 Listing 68 since only the two alternatives have the highest rating. The winner is the first alternative in the sub-list.

Listing 68: The resulting list containing the highest rated alternatives

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

5.2.4   Summary

To wrap it up, the masking strategy is used to eliminate unsuitable alternatives. If more non-orthogonal (i.e. overlapping) masking strategies are chained, then there is a risk that there may 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.

5.3   UML Notation

The main goal of this section is to provide a link between morph models described by the R-Algebra and the Unified Modeling Language (UML). The link between the two descriptions is built by examining a number of distinct cases on which are illustrated the commonalities and differences with focus on inheritance and sub-typing issues. This section deals only with a quite limited set of UML; nevertheless, it is necessary to introduce a couple of new extensions into UML so as to express the specific constructions.

Let us begin with the plain inheritance. Figure 61 shows the correspondence between the traditional inheritance or specialization and morph model expressions.
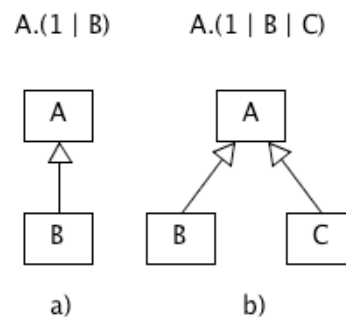
Figure 61: Two examples of the correspondence between inheritance/specialization and morph models

The morph model expression `A.(1 | B)` describes objects having always the feature `A`, which may be accompanied by the feature `B`; thus there are two sets of objects: `A` and `A.B`,

where A.B may be considered a sub-set of A. This actually corresponds to the notion of inheritance (or specialization) and thus it is possible to symbolize the above-mentioned expression as shown in Figure 61a. The expression A.(1 | B | C) can be interpreted and symbolized analogously.

The root type A is not abstract, as it can exist independently. Thus, with this in mind, an abstract type will correspond to a fragment that may never exist alone. This condition may be expressed by the expression A.B, where either type/fragment may be seen as an abstract type, since the expression rules out the possibility of the sole existence of A and B. Figure 62a shows the traditional UML notation for an abstract type A and one derived type B. As long as no additional class derived from A is introduced, the two types A and B will always accompany each other, which is expressed by the bidirectional relationship shown in Figure 62b.
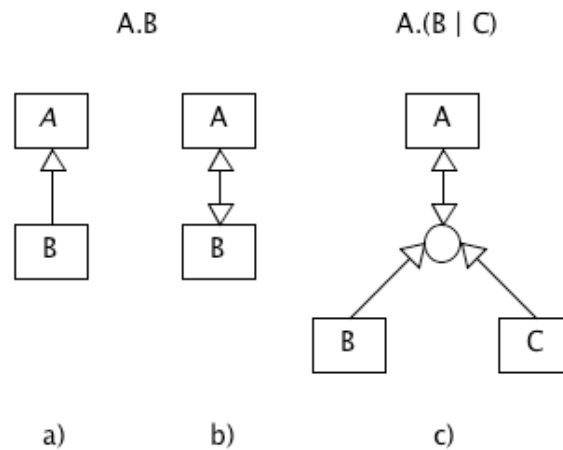
Figure 62: In figure a) the traditional way is used to express inheritance from an abstract class, while the figure b) uses the bidirectional arrow to emphasize the coocurrence of the two types (fragments). Figure c) shows two classes B and C inheriting from the abstract A.

Figure 62c depicts a simple hierarchy of two types B, C derived from an abstract type A. The bidirectional relationship of A with the grouping element, symbolized by the circle, expresses the fact that A will always be accompanied either by B or C, which makes A abstract; and this is exactly what the model A.(B | C) describes.

The Figure 63a is a slightly modified version of Figure 62c, in which all relationships are bidirectional and the grouping element is marked with the union stereotype.

A.B | B.C | A.C   A | B | A.B

a)   b)
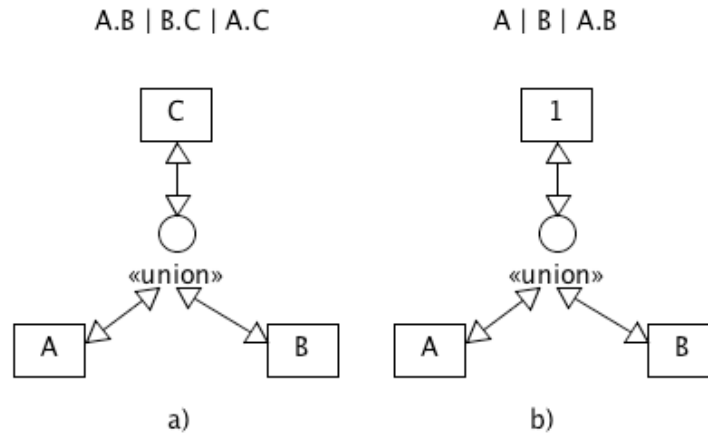
Figure 63: Using the `union` *stereotype to model mutually exclusive combinations of types*

The system of arrows makes up a directed graph, from which one may determine, which types are always co-occurring. A type `X` is always co-occurring with type `Y` as long as there is a continuous path in the graph from `X` to `Y`. Furthermore, the `union` stereotype at the grouping element declares that any incoming path may go out of the node via one outgoing path only, i.e. no forking on the exit from the union. It follows that the graph in Figure 63a generates the following co-occurring types: `AB`, `BC` and `AC`, which actually corresponds to the model `A.B | B.C | A.C`.

On the contrary, the `join` stereotype dictates that any incoming path forks to all available outgoing paths. Figure 64a is almost the same as Figure 63a except the stereotype at the grouping element. The diagram represents one triplet of co-occurring types `ABC`. The same model may be expressed alternatively as shown in Figure 64b.

A.B.C

c)   d)
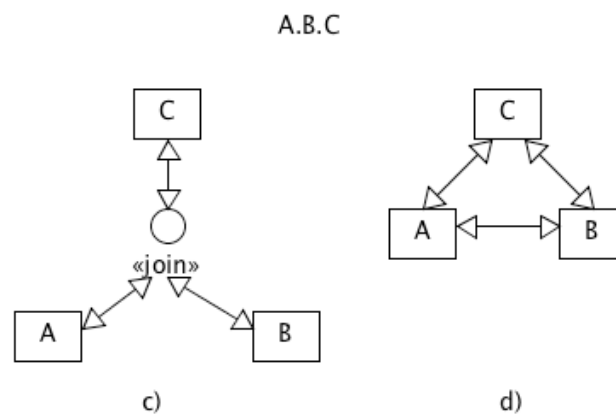
Figure 64: Using the `join` *stereotype to model coocurring combinations of types*

Figure 65a depicts the join of two groups of types `(A, B)` and `(C, D)`. The arrows indicate that, for example, the type `C` may co-occur with `A` or `B`, but it is mutually exclusive with `D`. In Figure 65b is shown a selective join of the two groups. Such a join may be expressed using inverse fragments in the model expression.

$$(A \mid B).(C \mid D) \qquad\qquad A.C \mid B.D = (A \mid B).(C.\sim B \mid D.\sim A)$$

a)   b)
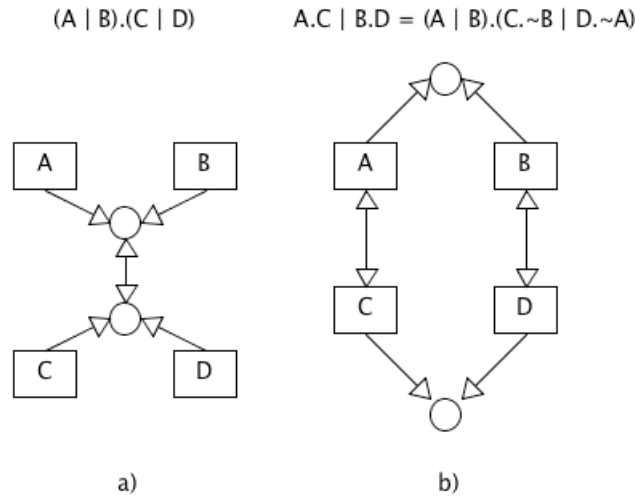
Figure 65: a) Two coocurring exclusive groups b) Excluding selected combinations of types (`A.D` and `B.C`) using the inverse fragment in the morph model expression

The following three diagrams illustrate various aspects of the extended UML syntax. In Figure 66*a*, types `A` and `B` are co-occurring not only with each other, but also with the group (`C, D`). For example `C` always co-occurs with `A` and `B`, but `A` always co-occurs only with `B` and vice versa. This fact makes the pair `AB` independent from `C` and `D` and thus the pair may be considered non-abstract.

Figure 66*b* a shows a similar case, in which the pair `AB` is no longer independent from `C` and `D` because of the two-headed relationship between the two groupings. Figure 66*c* is just an extended version of the preceding one with the additional group (`E, F`), which is always co-occurring with `A`, `B` and the group (`C, D`).

A.B.(1 | C | D)   A.B.(C | D)   A.B.(C | D).(E | F)

a)   b)   c)
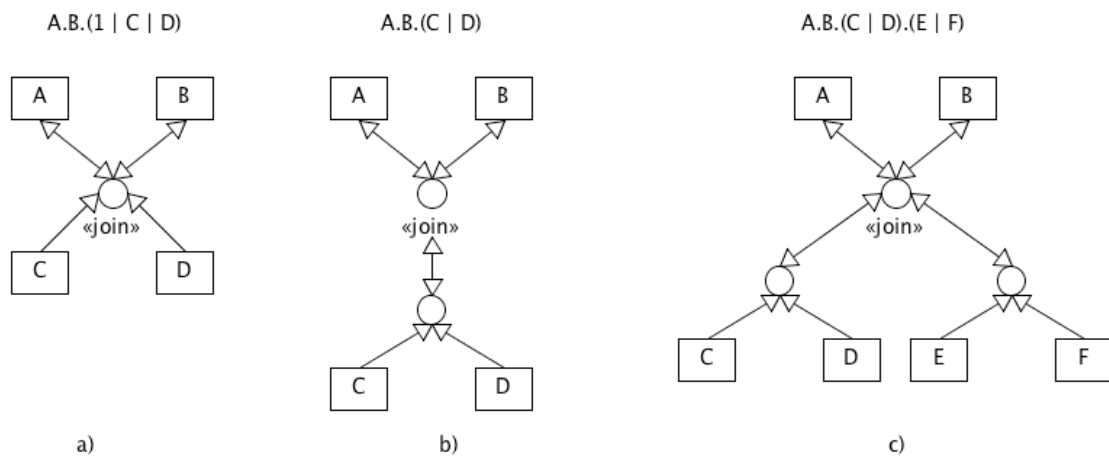
Figure 66: Various aspects of the extended UML syntax

5.4   Orthogonality of Fragments

Object morphology distinguishes between two categories of fragments that are antithetical in several respects. The former kind includes the fragments that do not override the behavior of existing fragments; instead they add new public members (properties, methods) only. On the contrary, the latter kind does exactly the opposite, i.e. it includes the fragments that do not add any new member, but they override the behavior of existing fragments.

The fragments of the former kind are called *pure fragments*, while those of the latter kind are *wrappers*. On account of the mutually exclusive properties, the two kinds are labeled as *orthogonal*.

Interestingly, such a division of fragments allows identifying a number of consequential properties, whereas each property from one category has its antipode in the other category. These antipodal properties make the division between the two fragment categories even more distinctive.

The first property concerns the number of dependencies of a fragment. As far as the pure fragments are concerned, they may have an arbitrary number of properties, i.e. possibly no dependency at all. On the contrary, the wrappers must have at least one dependency, which corresponds to the overridden fragment.

The second property determines whether other fragments are allowed to refer a given fragment. The fact that the pure fragments introduce new public members makes them referable by definition; the new members are exposed with the intention to find their user. On the other hand, the wrappers do not introduce anything new and therefore it makes no sense to refer them explicitly; a user will refer the wrapped fragment instead of the wrapper. Also, since the wrappers cannot be referred, then no wrapper may wrap another wrapper.

The third property is the fragment optionality, which determines if the model requires the fragment's presence or not. In the case of the pure fragment its optionality is determined by the optionality of its dependers. On the other hand, every wrapper is intrinsically optional, as no fragment depends on the wrapper; in other words the removal of the wrapper does not break any dependency.

The fourth property pertains to the number of instances of a fragment in a morph. Given that the pure fragments are referable there must be only one instance of the referred fragment so as to prevent the ambiguous dependencies (references). On the contrary, a morph may contain an arbitrary number of instances of the same wrapper since no other fragment depends on it and hence no ambiguity in dependencies can occur. Furthermore, it is possible that each wrapper instance is initialized differently, which may result in a different effect on the overridden behavior. The possibility of multiple instances of the same wrapper in a morph raises the question about the order of the wrapper instances, which is certainly significant, as the logic in the overridden method may not commute in general. The order

of the multiple wrapper instances must be therefore determined explicitly when creating the morph.

The fifth property determines how instances of the fragments are organized in the morph. As far as a pure fragment is concerned, the position of its single instance in the morph is not significant with respect to the instances of other fragments, except the instances of its wrappers. The wrappers of a fragment must always have a greater index than the index of the wrapped fragment; i.e. the wrappers must always be located to the right of the wrapped fragment. This organization of the wrappers allows determining the method invocation flow order of the chained (stacked) fragments (the so-called super-invocations): the right-most wrapper in the chain is considered as the first one and the wrapped fragment (i.e. the leftmost one) as the last one.

Figure 67 and Figure 68 outline the properties along with their causal relationship.
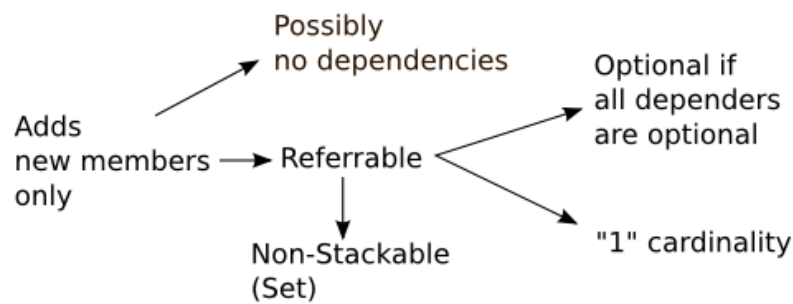
Figure 67: Causal relationship between the properties of pure fragments
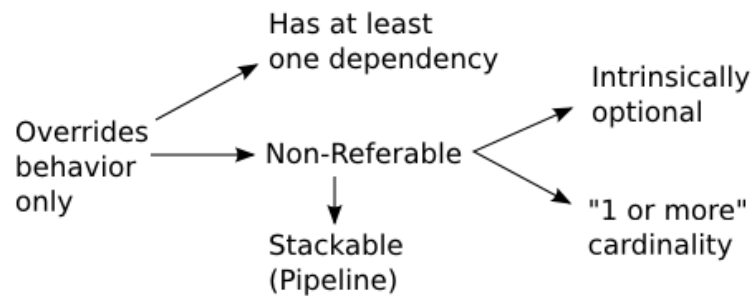
Figure 68: Causal relationship between the properties of wrappers

5.5   Translating Type Hierarchies

This section examines the compatibility of two most common inheritance models used in the contemporary languages with object morphology and especially with the concept of the orthogonality of fragments. In particular, this examination focuses on finding out how the studied inheritance models might be turned into the equivalent morph models. The goal is to show that the studied models may be actually seen as special cases of OM. The first rep-

resentative is the single-inheritance model used for instance in Java. The other model is the linearized traits model used in Scala and Python, for example.

5.5.1 Single-Inheritance Hierarchies

The languages using this type of inheritance, such as Java, allow specifying one super-type and a number of interfaces. As interfaces contain signatures only and hence they do not contain anything that could be inherited or overridden, they are not included in this examination. Thus the single-inheritance hierarchies always form trees. For the sake of simplicity this paragraph deals with a one-branch tree model, whose example is in Figure 69.
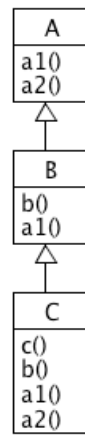
Figure 69: Single inheritance example

The type A contains two methods `a1()` and `a2()` that are overridden in the subtypes B and C. The type B adds a new method b() that is overridden in C. Before trying to translate this hierarchy into a morph model, it is necessary to consider the orthogonality of fragments in OM, as described in 5.4; a fragment cannot be a pure fragment and a wrapper at the same time. It is obvious, however, that both types B and C introduce new members and override some inherited members. With this in mind, it is not possible to map directly the hierarchy to one fragment to a morph model.

To do so, the types B and C must be split so that the individual parts either add new members or override some methods only. The result of the splitting is shown in Figure 70.

A.(1 | B.Ba.(1| C.Ca.Cb))
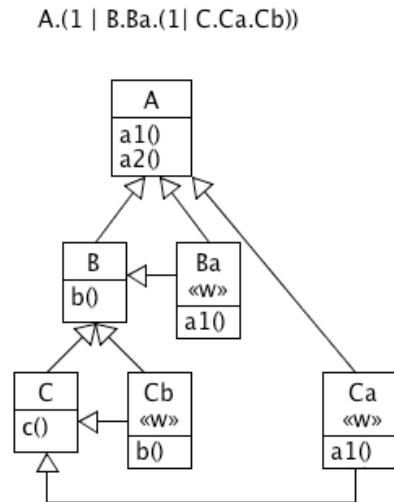
Figure 70: Splitting the types in the hierarchy to follow the orthogonality properties of the fragments from the corresponding morph model

This new structure may now be directly translated to a morph model. The wrappers are decorated with the stereotype w; the other elements are pure fragments. It should be noted that all fragments follow the properties outlined in 5.4; for instance, no fragment depend on any wrapper and the wrappers are hence optional. All wrappers have two dependencies: the wrapped fragment and the pure fragment containing the additional members of the split-up original type. The latter dependency allows the wrapper to use those additional members in the logic of the overridden methods.

Figure 71 depicts the structure of a morph corresponding to an instance of the type C. Each pure fragment instance establishes a "kernel" surrounded by the respective wrapper instances.

A.Ba.Ca.B.Cb.C
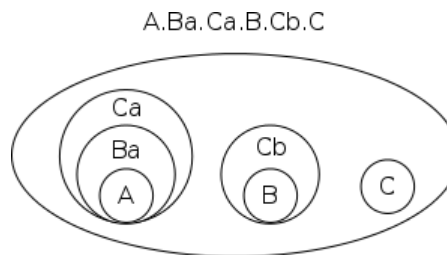
Figure 71: A depiction of a morph corresponding to an instance of C

5.5.2   Linearized Trait Hierarchies

Programming languages such as Scala and Python allow a type to inherit from more than one parent types. Although it could be called multiple-inheritance, this term is deliberately avoided so as to retain its original concept implemented in languages such as C++ and Eiffel. In contrast to the original concept of multiple inheritance the model, which is examined in this section, does not suffer from the so-called diamond problem (Malayeri & Aldrich, 2009) which refers to an ambiguity that occurs when two classes B and C inherit from class

A, and class D inherits from both B and C (Figure 72). In the case that A contains a method that is overridden by B or C, but not by D, it is not clear, which version of the method D actually should inherit.

Python mitigates this issue by applying the C3 linearization algorithm (CPAN, n.d.), which enforces two constraints: children precede their parents and if a class inherits from multiple classes, they are kept in the order specified in the tuple of base classes.

Also Scala solves the diamond problem by a linearization, however, it applies a different approach designed with respect to the underlying Java platform. In Scala, the linearization of any class must include the linearization of the extended class (Scala allows extending one class only due to the Java limitations). Next, the linearization of the class must contain all traits and classes from the linearization of any extended traits, whereas the traits may not be in the same order as they appear in the linearization of the traits mixed in. And finally, no class or trait may appear more than once in the linearization (McBeath, n.d.).

The approach used in OM is an extension of the Scala linearization. Let us assume that the linearization of type D is to be built. Before splitting the types in the hierarchy, the linearization of D is made and expressed by the following morph model:

```
Lin(D) = A.B.C.D
```
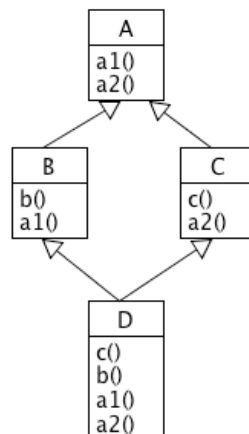

Figure 72: The diamond problem in multiple inheritance

The order of fragments in the model is important, as it determines the invocation flow through the stacked wrappers added in the next step.

Next, the types in the hierarchy are split in the same way as is done in the preceding section. The result is depicted in Figure 73.

A.(1 | Ba.B | Ca.C | Ba.Ca.B.C.(1 | Da.Db.Dc.D)
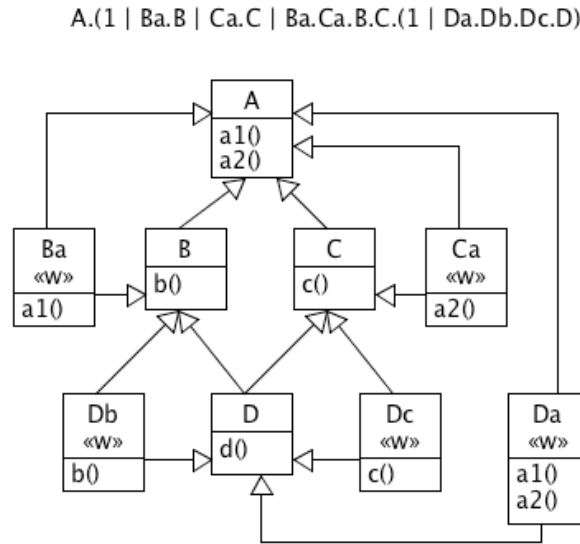
Figure 73: Splitting the types in the multiple inheritance diagram to map it to a morph model

Then each fragment in the linearized morph model is substituted with the fragment's splits.

```
Lin(D) = A.B.C.D = A.(B.Ba).(C.Ca).(D.Da.Db.Dc) = (A.Ba.Ca.Da).(B.Db).(C.Dc).D
```

The order of the splits of a given original type is irrelevant because the overridden methods never interfere; i.e. `D.Da.Db.Dc` has the same effect as `D.Dc.Db.Da` since `Da` cannot override the methods overridden by `Db` and `Dc` and so on. What is important, however, is the relative order of the fragments related to the same wrapped fragment; hence `Ba.Ca` is not the same as `Ca.Ba`. The splitting mechanism allows grouping the wrappers with the wrapped fragment while preserving the relative order between them.

The linearization of the other types is outlined in *Listing 69*.

Listing 69: Linearization of types by means of morph model expressions

```
Lin(B.C) = A.B.C = A.(B.Ba).(C.Ca) = (A.Ba.Ca).B.C
Lin(C)   = A.C = A.(C.Ca) = (A.Ca).C
Lin(B)   = A.B = A.(B.Ba) = (A.Ba).B
Lin(A)   = A
```

And to obtain the final morph model of the hierarchy the union of all linearization morph models is created.

```
A.(1 | Ba.B | Ca.C | Ba.Ca.B.C | Ba.Ca.Da.B.Db.C.Dc.D) =
A.(1 | Ba.B | Ca.C | Ba.Ca.B.C.(1 | Da.Db.Dc.D)
```

Figure 74 depicts the internal structure of the morph corresponding to an instance of `D`.
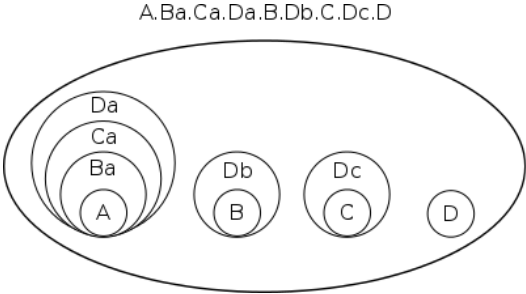
Figure 74: The internal structure of a D *instance*

6 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.

In order to provide some justification for these raw guidelines, they are used in the final case study presented in section 8 for analysis.

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 following subsections deal with the individual steps in more detail.

Note: This chapter uses both the terminology of the prototypical theory as well as the terminology of object morphology. For example a property in prototypical theory corresponds to a fragment or a wrapper in OM. The following text makes its best to avoid potential confusion.

6.1 Understanding Phenomena

At the beginning of this stage the analyst is trying to understand individual phenomena. (S)he notices what happens, however, neither understanding the essence nor the relationship between individual phenomena [OpenOOIssues]. Nor does (s)he realize similarities between different phenomena.

The goal in the stage is to acquire as much knowledge as possible at the level of individual instances; i.e. to understand well the extension of the concept being defined.

Let us illustrate this step on a variation of the airport scanner example. In this variation the scanner works correctly, however, the produced records are not documented. The analyst is examining individual records produced by the scanner and is trying to understand each record by comparing the scanned item with the produced record. After doing some research (s)he comes to the conclusion that two record properties carry information about density and conductivity, while another property describes the 3D wireframe model of the scanned item. (S)he also finds out that all these properties are optional, since the scanner's sensors may fail to measure the respective properties.

6.2   Identifying Prototypes

After the analyst has become familiar with the phenomena (s)he may begin to isolate the most distinctive instances, which will provide the basis for the description of one or more prototypes representing typical phenomena. The analyst may use a higher level of abstraction when describing the prototypes.

In the scanner example this step would result in the description of the "ideal" rectangle, cylinder, metal and paper items, which would become the prototypes of the concept of a scanned item. The language used in the description of the prototypes uses shape and material names, rather than the lower-level terminology from the previous step, i.e. density, conductivity etc.

In addition to the four prototypes, the analyst also includes the unit object (1), which will represent the detected, but non-categorized phenomena.

6.3   Property Analysis

When the concept is delimited by the description of the prototypes, it is necessary to define the intension of the concept; in other words, the analyst uses the prototype descriptions to abstract the prototype properties. Some of the properties may be present in all prototypes, while others may be specific to certain prototypes only.

After the properties in the concept's intension are specified, the analyst examines dependencies and constraints between the properties. This analysis may be based on the empirical knowledge acquired in the previous stages, i.e. on the observed co-occurrences or mutual-exclusivities of the individual properties, or it may take into account some inherent constraints of the properties.

In the context of the scanner example, the analyst would identify the following four properties: Metal, Paper, Rectangle and Cylinder. Taking into account the nature of the properties, (s)he concludes that both the metal and paper properties may co-occur with the rectangle or cylinder and vice versa. Furthermore, the paper and properties metal are mutually exclusive, just as the rectangle and cylinder ones. Moreover, all properties may occur alone.

The above property analysis may be summarized as follows:

Dependencies:

```
Metal => 1 | Rectangle | Cylinder
Paper => 1 | Rectangle | Cylinder
Rectangle => 1 | Paper | Metal
Cylinder => 1 | Paper | Metal
```

Constraints:

```
Metal * Paper = 0
Rectangle * Cylinder = 0
```

6.4   Morph Model Construction

Having done the property analysis, the analyst can proceed to the construction of the morph model, which represents the concept being modeled. The morph model plays the role analogous to that of the class in the Aristotelian conceptual framework.

A morph model does not usually consist of the alternatives corresponding to the concept's prototypes only, since such a morph model would not faithfully represent the concept's extension (phenomena). Instead, to construct the morph model, the analyst examines as many prototype combinations as permitted by the dependencies and constrains unraveled during the property analysis. This process is called *prototype expansion*.

Further, the property analysis may determine the significance of each property with respect to every alternative in the morph model containing this property. The significance of the properties may influence the selection of the most suitable alternative from the model when creating a representation of a particular phenomenon (i.e. a morph).

In the scanner example, the analyst would construct the following morph model:

```
Model = (1 | Metal | Paper)*(1 | Rectangle | Cylinder)
```

This morph model reflects all dependencies and constrains determined during the property analysis.

In terms of property significance, all properties are equally important in all respective alternatives, which may be outlined as follows:

```
1
Metal(1.0)
Paper(1.0)
Rectangle(1.0)
Cylinder(1.0)
Metal(1.0).Cylinder(1.0)
Metal(1.0).Rectangle(1.0)
Paper(1.0).Cylinder(1.0)
Paper(1.0).Rectangle(1.0)
```

6.5   Binding Properties To Context

In contrast to the previous steps, in which the analyst delivers a description of the concept, beginning with this step (s)he turns her attention on the runtime aspects of modeling. These aspects include primarily the creation of the representations of the concept's extension (i.e. phenomena) according to the contextual information referring to the phenomenon be represented; i.e. the input data.

The primary purpose of this step is to establish the binding between the properties in the concept's intension and the phenomenon's context data. The analyst examines each property with respect to the context data and defines the property's *binding function*.

The input of a property's binding function is the context data. The function evaluates the presence of the property in the context data and returns result of this evaluation. In case the function does not detect the property in the context data, it returns a special value or throws an exception. Otherwise, the function returns a value consisting of the *property factory* function accompanied by a weight indicating the property's "level of presence" in the context data. The property factory function is able to create an instance of the property in case the property will be part of the alternative selected for the instantiation. The weight is then combined with the property's significance in a given alternative when selecting the best alternative to represent the phenomenon.

Considering the example, the context data are in fact the records produced by the scanner, i.e.:

```
ctx: {density, conductivity, wireframe}
```

The binding function for the metal property could be defined as follows (in Scala):

```scala
def metalBind(ctx) =
  if (density >= 2000 && <= 10000 && conductivity >= 10 && conductivity <= 7)
    Some((() => new Metal, 0.75))
  else
    None
```

This metal property binding function checks whether the context properties fall into the valid ranges. If it is the case the function returns the property factory function and the weight 0.75). The factory function simply creates a new instance of Metal and the weight value 0.75 indicates that the binding function is certain on 75% that the context data corresponds to a metal thing.

6.6   Morphing Strategy Construction

Having all binding functions defined, the analyst may construct the chain of morphing strategies, which will be responsible for selecting the best alternatives for representing a phenomenon referred by the context data. See 5.2 for more information on the topic of morphing strategies.

In the case of the example, the analyst could use the strategy, which rates the alternatives according to the fact whether the alternative contains a given fragment. Let us assume that the `metalBind` binding function recognizes a metal thing in the context data and returns the weight `0.75`. At same time `rectBind` recognizes a rectangle in the context data and returns the weight `0.5`. Then, if the alternative being rated contains both properties and the significance of these properties in this alternative is 1, the total rating is given as `0.75*1 + 0.5*1 = 1.25`.

```
strategy =
 rateFragment[Metal](metalBind,
  rateFragment[Paper], paperBind,
   rateFragment[Rectangle], rectBind,
    rateFragment[Cylinder], cylBind))))
```

The strategy chain is created as a composition of all particular strategies.

6.7   Creating Recognizer

The purpose of this final step is to create a recognizer, which will produce representations of the concept's phenomena. At this stage all pieces necessary for the creation of the recognizer are already available, as to create a recognizer one need the model and morphing strategy.

The creation of the recognizer can be formally expressed as follows:

```
recognizer = Model(strategy)
representation = Recognizer.recognize(context)
```

7 Reference Implementation (RI)

The goal of this work is not only to develop the theoretical concept of object morphology but also to provide a reference implementation as a proof-of-concept prototype. The preceding study indicates that the Scala language might be a suitable candidate for an extension that would allow the implementation of Object Recognizer. Having developed the case study's code examples in three languages the author of this thesis have come to the conclusion that it would be much easier to extend the Scala compiler and other Scala subsystems to achieve some dynamicity rather than to extend a dynamic language such as Groovy to obtain a robust and static type safety.

The RI design elaborates the fundamental OM entities, such as morph models and morphing strategies, morphs, morph references and fragment factories etc. It also defines the application interface (API).

The RI is also called Morpheus and thus these two names are used interchangeably in the following text.

7.1 Design Overview

The RI may be broken down into two 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. The compile-time component also checks if the target morph model conforms to the source model and the run-time component composes target and source fragment instances to create a new joined instance.

The responsibilities of both components are outlined in the following lists. The text that follows the bullet points elaborates the individual topics in the bullet points in more detail.

Compile-Time

- Parses and validates morph models at compile-time
- For every valid morph model it generates a special code representing the model. Such a code is loaded and used at run-time by the run-time component.
- Transforms fragment traits so that they may be composed dynamically
- Generates auxiliary artifacts such as fragment classes
- Verifies whether target and source morph models can be mapped; i.e. whether the target model conforms to the source model.

- For every mapping it generates a special code describing how to map the target alternatives to the source alternatives. Such a code is loaded and used at run-time by the run-time component.

Run-Time

- Loads morph models

- Uses the loaded models and morph strategies to create recognizers

- Creates morphs

- Manages fragment factories

- Handles remorphing, i.e. changes a morph's shape according to the morph model and supplied morphing strategy

- Provides API for implementing custom morphing strategies

- Performs run-time code transformation of fragment classes

- Joins fragment instances from source and target models and creates morph instances according to mapping models

- Chains target and source morphing strategies

All code snippets in this section assume the following import statements:

```scala
import org.morpheus._
import org.morpheus.Morpheus._
```

7.2   Compile-Time Component

As mentioned above, the compile-time component primarily deals with the static analysis (validation) of morph models and with the auxiliary code generation. The following subsections deal with the implementation of all morph model elements.

The compile-time component is implemented by means of Scala macros, which are available via object `org.morpheus.Morpheus`.

7.2.1   Morph Model Elements

All model elements are represented by Scala traits. To distinguish one model element from another the developer uses a set of special annotations `dimension`, `fragment`, `category` and `wrapper`.

The model element validation performs a check of *typogenic* dependencies (e.g. inheritance). In the case of concrete elements (fragments, wrappers) the validation also checks the context dependencies.

For each concrete model element the RI generates an auxiliary class, the so-called *stub class*. The purpose of the stub classes is to help create instances of model elements, i.e. *stubs*. Although the stubs are not meant to be accessed directly, they are in fact real objects, which may have some state. Provided that all model elements are implemented as traits it is not possible to instantiate the stubs directly from these traits, because in order to create an object in Scala, one needs a class.

Additionally, the RI generates also an auxiliary trait for every dimension on behalf of the stub classes of dimension wrappers, as explained in 7.2.2.

7.2.2 Dimensions

Since a dimension essentially corresponds to an interface, it is natural to use a pure trait (i.e. signatures only) to represent a dimension in Scala. In order to distinguish a dimension from an ordinary trait, the dimension trait is annotated with `@dimension` annotation. The code in *Listing 70* demonstrates an example of a dimension trait.

Listing 70: An example of a dimension trait

```scala
@dimension
trait Shape {
    def calculateArea(): Float
}
```

Dimension Stub Class

For every dimension trait the RI generates the dimension trait's auxiliary class making it easier to generate stub classes of dimension wrappers. The generated stub class for the `Shape` dimension would consist of the code shown in *Listing 71*.

Listing 71: The stub class generated for the Shape dimension

```scala
class Shape$dimension extends org.morpheus.SuperBase[Shape] with Shape {

  def calculateArea(): Float = $super$.calculateArea()

}
```

The name of the stub class is made up of the dimension trait's name and the postfix `$dimension`. It extends from the abstract class `org.morpheus.SuperBase[Shape]`, which belongs to the RI, and from the dimension trait `Shape`. All methods are implemented by the

delegation of the call to the inherited method $super$, which returns the reference to the next stub handling the call. The next stub may be another dimension wrapper stub or a fragment stub.

The stub classes of all dimension wrappers extending the Shape dimension will extend from this stub class. A dimension wrapper stub will be an instance of SuperBase[Shape] as well as of Shape.

7.2.3   Fragments

In RI, a fragment is implemented as a trait annotated with the fragment annotation. The fragment trait may extend from dimensions or abstract categories. In contrast to a dimension, a fragment contains no abstract members. An example of a fragment is in *Listing 72*.

Listing 72: An example of a fragment

```scala
@fragment
trait Rectangle extends Shape {
    var width: Int = _
    var height: Int = _
    def area = width * height
}
```

Dependencies

A fragment depending on other elements declares its dependencies in the self-type, which is actually the standard way for traits in Scala to specify the context in which they may occur. However, RI goes further in the interpretation of the self-type since it may refer to a morph model, where each alternative corresponds to one dependency alternative. All dependency alternatives implicitly contain the depending fragment. *Listing 73* shows a fragment with two dependencies.

Listing 73: A fragment with two dependencies expressed by the trait's self-type

```scala
@fragment
trait Banknote extends Currency {
    this: Paper with Rectangle =>

    …
}
```

Such a trait actually can be instantiated in both Scala and RI, as shown in *Listing 74*.

Listing 74: Instantiation of the `Banknote` *fragment in Scala and Morpheus*

```scala
// Scala
val bankote = new Banknote with Paper with Rectangle {}
// Morpheus
val banknote = compose[Banknote with Paper with Rectangle].!
```

Listing 75 illustrates a fragment with two alternative dependencies. It also shows how to determine which alternative is currently in use by means of the `select` macro (see 7.3.3). The `currencyToString` method returns a textual representation of the currency with respect to the context alternative.

Listing 75: A fragment with alternative dependencies

```scala
@fragment
trait Currency {
   this: (Paper with Rectangle) or (Metal with Cylinder) =>

   def currencyToString() = {
      select[Paper with Rectangle](this) match {
         case Some(paperRect) =>
            // create a textual repr. of the paper rectangle
            …
         case None => select[Metal with Cylinder](this) {
            case Some(metalCyl) =>
               // create a textual repr. of the metal cylinder
               …
            case _ => // not reachable
         }
      }
   }
}
```

Fragment Stub Class

The same fragment stub may appear in various morphs, i.e. instances of morph model alternatives. Stubs may encapsulate and preserve mutable state between successive engagements in morphs. Thus a stub may change its state when being a part of one morph and use this changed state later when being in another morph. Even though a stub exists as an object, it cannot be used directly, but only through a morph, which the stub is a part of.

Every stub class is given the name composed of the name of the fragment and the postfix `$fragment`. For fragments with no dependency RI generates an empty class extending the fragment (*Listing 76*).

Listing 76: The stub class generated for fragment `Rectangle`

```scala
class Rectangle$fragment extends Rectangle {}
```

Stub classes of fragments with dependencies have the same self-type as the fragment (*Listing 77*).

Listing 77: The stub class generated for fragment `Banknote`

```scala
class Banknote$fragment extends Banknote {
  this: Paper with Rectangle =>
}
```

Fragment Initialization

RI also solves the problem of initialization of immutable final fields (values) in fragment traits. For example, the `Rectangle` fragment in *Listing 78* has two such fields. It is not clear how to initialize them from external data, since there is no constructor with parameters (traits have no constructors). These fields cannot even be left abstract, since fragments must not have any abstract member. Nevertheless, in Scala, one can use the symbol "???" to mark a member as "not-yet-implemented". The actual implementation of those members is a part of the generated stub class of the fragment.

Listing 78: Using immutable fields in a fragment

```scala
@fragment
trait Rectagle extends Shape {
  val width: Float = ???
  val height: Float = ???
  …
}
```

Even though a fragment trait cannot declare a constructor with parameters, the stub class of the fragment can. Unfortunately the stub class is out of the direct control of the developer, because it is generated.

RI comes up with the following solution. The immutable part of the fragment trait is separated as a new trait called *configuration trait*. The fragment trait adds the configuration trait to its extensions list and marks it with the special marker type `dlg[T]` as a trait whose members are not to be implemented explicitly in the fragment trait, but automatically by delegation in the fragment's stub class.

Listing 79 demonstrates this approach on the `Rectangle` fragment. The `RectangleCfg` configuration trait only contains the immutable fields moved from `Rectangle`, which newly extends `RectangleCfg` decorated with `dlg[_]`. Since the abstract members originate in a trait

marked with `dlg[_]`, the fragment does not have to implement them, because they are to be automatically implemented in the stub class.

Listing 79: Separating immutable fields into the configuration trait

```scala
trait RectangleCfg {
   val width: Float
   val height: Float
}


@fragment
trait Rectagle extends Shape with dlg[RectangleCfg] {
   def area(): Float = width * height
}
```

The `dlg` marker is defined as the identity generic type thus it has no effect on the extension list; it is just a marker processed by the compiler extension.

```scala
type dlg[T] = T
```

The stub class generated for `Rectangle` now contains a constructor with one parameter for passing an object carrying the immutable data for the stub (*Listing 80*). The two immutable members inherited from `Rectangle` are implemented by delegation on the object passed as the argument.

Listing 80: The stub class generated for the `Rectangle` fragment extending its configuration trait `RectangleCfg`

```scala
class Rectagle$fragment($config$: RectangleCfg) extends Rectangle {
   val width = $config$.width
   val height = $config$.height
}
```

For the sake of convenience, every config-trait should be accompanied by a case-class implementing the config-trait.

```scala
case class RectangleInit(width: Float, height: Float) extends RectangleCfg
```

An instance of this auxiliary class is then passed to a macro preparing the fragment factory. For details see 7.3.3.

```scala
val rectFragStubFactory = frag[Rectangle, RectangleCfg](RectangleInit(100, 50))
```

Fragment Conformance Level

The fragment conformance level specifies how the context morph's model must conform to the dependency model of the fragment's dependencies. The level is specified in the

`confLevel` attribute of the fragment annotation and may have two string values: "partial" or "total". The two values correspond to the injective, resp. surjective conformance of the contextual model to the dependency model (see 5.1.9). The default value is "partial". The fragment `ItemRenderer` in *Listing 81* requires that the context morph model satisfy all four alternative dependencies. Were the `confLevel` set to "partial" (the default one), the model would be required to satisfy at least one dependency alternative.

Listing 81: Specifying the conformance level for the dependencies of `ItemRenderer`

```
@fragment(confLevel="total")
trait ItemRenderer {
    this: (Rectangle | Cylinder)*(Paper | Metal) =>
    def render() { … }
}
```

7.2.4   Dimension Wrappers

A dimension wrapper in RI is implemented as a trait marked with two annotations `dimension` and `wrapper`. The dimension wrapper is supposed to extend one dimension trait, as shown in *Listing 82*.

Listing 82: A dimension wrapper example

```
@dimension @wrapper
trait ZoomedShape extends Shape {
    val factor: Float
    abstract override def area(): Float = factor * super.area()
}
```

All overridden methods are decorated with `abstract override` keywords to indicate that the methods override abstract methods.

For each dimension wrapper the compiler extension generates a stub class that extends the stub class of the dimension and the trait of the wrapper (*Listing 83*).

Listing 83: The stub class generated for dimension wrapper `ZoomedShape`

```
class ZoomedShape$fragment extends Shape$dimension with ZoomedShape {}
```

A dimension wrapper may also separate its immutable final members to the configuration trait so as to allow the constructor-like initialization.

To illustrate the flow of an invocation of an overridden method, let us assume we create a morph consisting of `Rectangle` and `ZoomedShape` and invoke the `area` method on it, as shown in *Listing 84*.

Listing 84: Invocation of the `area()` method overridden by `ZoomedShape` dimension wrapper

```
val zsf = compose[Rectangle with ZoomedShape].!
zsf.area()
```

Then, considering the trait linearization (see 5.5.2), the invocation of `area()` will proceed in the steps outlined in *Listing 85*.

Listing 85: The proceeding of the `area()` invocation

```
1. ZoomedShape$fragment.area()
2. ZoomedShape.area()
3. Shape$dimension.area()
4. Rectangle.area()
```

The invocation on the last line corresponds to the invocation of the next stub in the chain, which is the `Rectangle` stub. The next stub in the chain is obtained dynamically from the `$$super$` dynamic variable in the `SuperBase` class, which is the direct super class of the dimension stub class `Shape$dimension`. The actual chain of wrapper stubs is created at the time of the morph creation.

7.2.5   Fragment Wrappers

In RI, a fragment wrapper is implemented as a trait annotated with `fragment` and `wrapper` annotations and extending one fragment trait. Every overriding method is marked with the `override` keyword to indicate that the method overrides a concrete method (*Listing 86*).

Listing 86: An example of a fragment wrapper

```
@fragment @wrapper
trait ReversedRectangle extends Rectangle {
    override def width(): Float = super.height()
    override def height(): Float = super.width()
}
```

For each fragment wrapper RI generates a stub class extending the fragment trait only. In this respect the fragment wrapper stub classes are the same as the fragment stub classes.

Listing 87: The stub class generated for fragment wrapper `ReversedRectangle`

```
class ReversedRectangle$fragment extends ReversedRectangle {}
```

A fragment wrapper may also separate its immutable final members to the configuration trait so as to allow the constructor-like initialization.

To illustrate the flow of an invocation of an overridden method, let us assume we create a morph consisting of `Rectangle` and `ReversedRectangle` and invoke the `width` method on it, as shown in *Listing 88*.

Listing 88: Invocation of the `width()` method overridden by `ReversedRectangle` fragment wrapper

```
val revRect = compose[Rectangle with ReversedRectangle].!
revRect.width()
```

The expected flow of nested calls is outlined in *Listing 89*.

Listing 89: The proceeding of the `area()` invocation

```
1. ReversedRectangle$fragment.width()
2. ReversedRectangle.width()
3. Rectangle.height()
```

The last line is the invocation of the next stub in the chain, which is the `Rectangle` stub. The reference to the next stub is obtained dynamically by an injected byte code replacing the original byte code sequence performing the "super" invocation.

7.3   Run-Time Component

The run-time component of the RI is primarily responsible for building morph models from morph model expressions and for creating morphs. The following subsections describe the sub-components participating in these activities.

7.3.1   Morph Model Parser

The morph model parser is a component creating an instance of a morph model specified as a morph model expression. The functionality of the morph builder exposed through a set of macros enclosed in object `org.morpheus.Morpheus`.

Note: Although macro invocations appear in the code as the standard method invocations, they are actually resolved at compile-time (Scala Documentation, n.d.). They produce a syntax tree that is embedded in the place of the invocations. Also the arguments passed to the macros are resolved at compile time. Therefore it makes no sense to pass scalar values that cannot be resolved statically (i.e. at compile-time).

The `parse` macro is used to build a morph model according to the morph model expression specified in the type argument (*Listing 90*). The `checkDeps` argument indicates whether the model builder checks the contextual dependencies of the model elements.

Listing 90: The `parse` *macro*

```
def parse[M](checkDeps: Boolean): MorphModel[M]
```

The statement in *Listing 91* builds the morph model of the scanned item as used in the case study.

Listing 91: Parsing a morph model expression including the dependency check

```
val itemModel = parse[Item*(Rectangle | Cylinder)*(Paper | Metal)](checkDeps =
true)
```

The model expression uses the join operator * and the union operator |, which are defined in object `Morpheus` (*Listing 92*).

Listing 92: Morph model expression operators

```
type *[F0, F1]
type |[F0, F1]
```

Note: The older Morpheus code uses another pair of operators: "`with`" and "`or`". The "`with`" operator is actually a standard Scala keyword for mixing traits. The "`or` " operator is an additional type defined just as operators "`*`" or "`|`" (*Listing 93*).

Listing 93: Alternative use of operators `with` *and* `or`

```
val itemModel =
    parse[Item with (Rectangle or Cylinder) with (Paper or Metal)] (true)
```

The morph model instance returned by the parse macro is used primarily to assemble morphs (see the next section); nevertheless, it may also be used as an introspection tool to examine the structure of the morph model, as shown in the following examples.

The `altIterator` method returns an iterator whereby one can obtain the descriptors of all alternatives of the model (*Listing 94*).

Listing 94: The alternatives iterator

```
for (alt <- itemModel.altIterator())
    println(alt)
```

The preceding code snippet produces the following output:

```
List(FragmentNode(0,false), FragmentNode(1,false), FragmentNode(3,false))
List(FragmentNode(0,false), FragmentNode(2,false), FragmentNode(3,false))
List(FragmentNode(0,false), FragmentNode(1,false), FragmentNode(4,false))
List(FragmentNode(0,false), FragmentNode(2,false), FragmentNode(4,false))
```

The `FragmentNode` is a case class representing a concrete element node (i.e. a fragment or a wrapper) in the morph model expression tree. Its first property holds the index of the element and the second property indicates whether the element is a placeholder (see page 119). The console output reveals that the model consists of four alternatives, each consisting of three fragments.

To obtain more information about individual elements in alternatives one may invoke the `fragmentDescriptorsList` method.

```
itemModel.fragmentDescriptorsList
```

As its name suggests, this method returns a list of the concrete element descriptors in the model. *Listing 95* illustrates the use of this method to print all fragments in the morph model.

Listing 95: Using `fragmentDescriptorsList` to print all fragments in the morph model

```
for (fd <- itemModel.fragmentDescriptorsList)
    println(s" ${fd.index}:${fd.fragTag}")
```

The console output of the preceding loop is:

```
0:TypeTag[org.morpheus.tutorial.Item]
1:TypeTag[org.morpheus.tutorial.Rectangle]
2:TypeTag[org.morpheus.tutorial.Cylinder]
3:TypeTag[org.morpheus.tutorial.Paper]
4:TypeTag[org.morpheus.tutorial.Metal]
```

Besides the index and type tag of the concrete model element the descriptor also provides the optional type tag of the configuration trait, if any (`cfgTag`).

Another useful introspection method is `lubComponents`. It returns a list of classes that constitute the LUB of the model. *Listing 96* shows how to use this method to print the components of the morph model's LUB.

Listing 96: Using `lubComponents` to print the component types of the morph model's LUB

```
for (lc <- itemModel.lubComponents)
    println(lc)
```

The output of the previous loop is:

```
class org.morpheus.tutorial.Item
class org.morpheus.tutorial.Shape
class org.morpheus.tutorial.Material
```

Synthetic Types

The morph model builder also generates secondary types derived from the morph model expressions. These types are stored as public type members in morph model instances (`org.morpheus.MorphModel`) and are used in various places, such as morph variable declarations, type arguments, return types etc.

- `Model`: a copy of the morph model expression

- `LUB`: the lowes upper bound of the morph model

- `Recognizer`: the type of recognizers instantiated from the model

- `MutableMorph`: the type of mutable morphs created by the recognizers instantiated from the model. It is defined as: `LUB with org.morpheus.MutableMorphMirror[Model]`

- `ImmutableMorph`: the type of immutable morphs created by the recognizers instantiated from the model. It is defined as: `LUB with org.morpheus.ImmutableMorphMirror[Model]`

7.3.2 Recognizer

The recognizer is a component creating morphs as instances of the morph model alternatives. The selection of the instantiated alternative is done by the morphing strategy, which determines the right alternative by reflecting outer conditions (or also inner conditions when re-morphing).

There are two basic macros creating an instance of a recognizer, both taking two arguments: the model and strategy (*Listing 97*).

Listing 97: The `compose` and `singleton` macros creating a recognizer

```
def compose[M](model: MorphModel[M],
               strategy: MorphingStrategy[M]): model.Recognizer

def singleton[M](model: MorphModel[M],
                 strategy: MorphingStrategy[M]): model.Recognizer
```

The two macros differ in the factories used to create the stubs of the concrete elements of the morph model. While the `compose` macro uses the non-singleton factories, which always create new stubs, the `singleton` macro uses the singleton factories, which create the stubs only once. The term "once" relates to the recognizer instance; in other words, all morphs the recognizer creates contain the same stubs, even when the morphs are re-morphed. On the other hand, the re-morphing of a morph consisting of stubs created by the non-singleton factories will result in a new morph made up of new stubs.

The described difference in the behavior of the two macros has significant consequences with respect to the state of the stubs. The re-morphing of a morph created by the `compose` macro resets the state of the morph, as all stubs are re-created. On the contrary, a morph created by the `singleton` macro preserves the state. After re-morphing a morph the re-placed stubs are just suspended until their next appearance in another morph.

As shown in section 7.3.4, the default factories may be selectively overridden. It allows creating morphs, in which some fragments preserve the state when re-morphing, while the others are reset. The overriding of the factories is especially important when the stubs must be initialized from some external data. In such a case the fragment declares its configuration trait and the overriding factory of that fragment takes an instance of the configuration trait as its parameter.

Note: `Recognizer` is an alias for the `MorphKernel` class. Since the core of Morpheus was developed before the terminology settled down there are a number of classes named in the context of the old terminology. Therefore several type aliases had to be introduced to adapt those classes to the current terminology.

The `Recognizer` type member of the morph model passed as the firsts argument defines the return type of these macros (*Listing 98*).

Listing 98: The type of the created recognizer is defined in the morph model (i.e. `itemModel.Recognizer`*)*

```
val itemRec: itemModel.Recognizer =
    singleton(itemModel, rootStrategy(itemModel))
```

The `rootStrategy` macro creates the default strategy for a given morph model. This default strategy always picks the first alternative from the list of alternatives provided by the morph model.

The model type argument `M` needn't be specified as it is inferred from the `model` argument.

The recognizer instance contains two lazily instantiated default morphs. The first one is an immutable morph, while the other is a mutable morph. One obtains the two morphs by invoking the "!", resp. "~" members (*Listing 99*).

Listing 99: Immutable and mutable default morphs of the recognizer

```
val immutableItem: itemModel.ImmutableMorph = itemRec.!
val mutableItem: itemModel.MutableMorph = itemRec.~
```

The difference between the two types of morphs lies in the distinct behavior when re-morphing. When re-morphing an immutable morph a new instance of the morph is produced; i.e. the internal structure of the original morph remains the same. On the contrary,

when re-morphing a mutable morph its internal structure is modified so as to reflect the newly selected morph alternative; i.e. no new morph instance is created.

The type of the mutable morph, resp. the immutable, is given by the `MutableMorph`, resp. `ImmutableMirror` type member of the morph model, which is a composition of the LUB type and `org.morpheus.MutableMorphMirror`, resp. `org.morpheus.ImmutableMorphMirror` (*Listing 100*).

Listing 100: Types for mutable and immutable morphs

```
type LUB = Item with Shape with Material
type MutableMorph = LUB with MutableMorphMirror[Model]
type ImmutableMorph = LUB with ImmutableMorphMirror[Model]
```

Since the LUB is `Item with Shape with Material`, it is possible to access any member of those three types through any morph created by the `itemRec` recognizer, e.g. the `density` property of the `Material` trait.

```
immutableItem.density = 1.0
immutableItem.density = 1.0
```

New morph instances may be created by the `make` and `make_~` methods. The first one creates immutable morphs, while the other mutable ones (*Listing 101*).

Listing 101: The `make` and `make_~` methods for creating immutable and mutable morphs governed by the default morphing strategy (associated with the recognizer)

```
def make: model.ImmutableLUB

def make_~: model.MutableLUB
```

It is further possible to create new morphs using another morphing strategy than the default one associated with the recognizer (*Listing 102*).

Listing 102: The `morph` and `morph_~` methods for creating immutable and mutable morphs governed by a custom morphing strategy

```
def morph(strategy: MorphingStrategy[model.Model]): model.ImmutableLUB

def morph_~(strategy: MorphingStrategy[model.Model]): model.MutableLUB
```

All above-mentioned factory methods have their counterparts returning the optional result (*Listing 103*). These methods are useful in situations when the morphing strategy may fail to pick the right alternative, for example if the item being scanned does not correspond to any known shape.

Listing 103: "Optional" versions of the morph factory methods

```
def maybeMake: Option[model.ImmutableLUB]

def maybeMake_~: Option[model.MutableLUB]

def maybeMorph(strategy: MorphingStrategy[model.Model]):
   Option[model.ImmutableLUB]

def maybeMorph_~(strategy: MorphingStrategy[model.Model]):
   Option[model.MutableLUB]
```

Just as the morph model instance also the recognizer provides a couple of introspection methods such as `fragmentList`, `fragmentHolder` and `altIterator`.

The `fragmentList` method is analogous to `fragmentDescriptorsList` in the morph model with the exception that `fragmentList` returns a list of the so-called fragments holders that hold the concrete element stubs (instances) along with the corresponding descriptors.

Listing 104: A usage of the `fragmentList` method

```
for (fh <- itemRec.fragmentList)
   println(fh.fragment)
```

The loop in *Listing 104* prints the descriptors associated with the fragment holders in the `itemRec` recognizer.

Through the `fragmentHolder[F]` method it is possible to look up certain concrete model elements (*Listing 105*). The type argument specifies the type of the requested element.

Listing 105: A usage of the `fragmentHolder` method

```
itemRec.fragmentHolder[Rectangle] match {
   case None => // rectangle not found
   case Some(rectangleFrag) => // rectangle found
}
```

The `altIterator` is analogous to the `altIterator` method in the morph model. It returns a sequence of lists, where each list corresponds to one alternative in the model. Each list consists of the fragment holders along with the corresponding fragment nodes.

Besides the recognizer creating macros mentioned in the beginning of this section there are two simplified versions of them (*Listing 106*).

Listing 106: The simplified versions of `compose` and `singleton` macros

```
def compose[M]: Recognizer[M]
def singleton[M]: Recognizer[M]
```

These macros build the morph model and the creation of the recognizer in one step.

```
val itemRec = singleton[Item*(Rectangle | Cylinder)*(Paper | Metal)]
```

The resulting recognizers use the default morphing strategy; the morphs created by such recognizers will always be assembled according to the first morph model alternative. However, these morphs may be re-morphed into another shape later by using another morphing strategy.

7.3.3   Using Morphs

Besides using morphs as ordinary instances of the model's LUB it is possible to perform additional operations on them. Those operations are accessible through the `org.morpheus.MutableMirror` trait mixed in the morph instance.

The first essential operation is re-morphing, which allows assembling the morph again in order to reflect the changes in the outer or inner conditions. As mention in the previous section, the re-morphing of an immutable morph creates a new morph instance assembled according to the newly selected morph model alternative (*Listing 107*).

Listing 107: Explicit re-morphing of an immutable morph

```
val newImmutableItem = immutableItem.remorph
assert(newImmutableItem ne immutableItem)
```

The assert statement verifies that the new and old morphs are different instances.

On the other hand, the re-morphing of a mutable morph does not result in a new instance. Instead, the internal structure of the original morph instance is re-arranged so as to conform to the newly selected alternative (*Listing 108*).

Listing 108: Explicit re-morphing of a mutable morph

```
val newMutableItem = mutableItem.remorph
assert(newMutableItem eq mutableItem)
```

It is also possible to re-morph the morph by means of a different morphing strategy. The overloaded `remorph` method takes the strategy as its only argument (*Listing 109*).

Listing 109: Re-morphing of an immutable morph with a custom strategy

```
val newStrategy = …
val newImmutableItem = immutableItem.remorph(newStrategy)
```

To find out which alternative was used to construct the morph one can use the `myAlternative` method (*Listing 110*).

Listing 110: Method `myAlternative`

```
def myAlternative: List[FragmentHolder[_]]
```

The result of `myAlternative` is a list of fragment holders (stubs) constituting the morph.

```
println(mutableItem.myAlternative)
```

The preceding statements may output:

```
List(org.morpheus.tutorial.Item, org.morpheus.tutorial.Rectangle,
org.morpheus.tutorial.Paper)
```

The select macro allows testing the actual type of the morph. If the morph matches the type specified in the type argument of the select macro the result is `Some(ref)`, where ref is the type-cast reference to the morph. Otherwise the macro returns None (*Listing 111*).

Listing 111: Using the `select` macro

```
select[Rectangle with Paper](mutableItem) match {
   case None => // not a paper rectangle
   case Some(paperRectangle) =>
     paperRectangle.width += 100
     paperRectangle.height += 100
}
```

The important aspect of the select macro is that it is statically checked for compatible type expressions in the type argument. The macro aborts the compilation as long as the type argument is not compatible with any morph model alternative.

```
select[Banknote](mutableItem) // does not compile
```

To examine the composition of the morph in more dynamically one may use the `findFragmentHolder` method. This method returns an optional reference to the fragment holder of the requested fragment.

```
println(mutableItem.findFragmentHolder[Banknote]) // None
```

The preceding statement prints None, as the model of the `mutableItem` does not contain the `Banknote` fragment.

The mutable sub-trait of `MorphMirror` contains only one additional member. The `delegate` property returns the reference to the immutable morph wrapped by the mutable morph. The mutable morph is in fact a proxy that delegates all invocation on this wrapped immutable instance. This is the only mutable part of the mutable morph, except the mutable state of the stubs, of course.

Listing 112: Illustration of the `delegate` member of a mutable morph

```scala
val dlg = mutableItem.delegate
if (dlg.isInstanceOf[Rectangle with Paper]) {
    val paperRectangle = dlg.asInstanceOf[Rectangle with Paper]
    paperRectangle.width += 100
    paperRectangle.height += 100
}
```

The code in *Listing 112* is actually a lighter version of the select macro example. It is deliberately divided into two steps in order to avoid race conditions; the delegate reference might be modified in the background between the test and the type-cast operation, which could lead to a type-case exception.

The `owningMutableProxy` method, which is present in the `MorphMirror` trait, allows determining whether an immutable morph is owned by some mutable morph. Its use is illustrated along with the mirror macro in the following example.

The mirror macro may be invoked from a concrete model element only. It returns the *immutable* context `MorphMirror` instance; i.e. the morph that contains the stub of the concrete model element. This immutable morph may or may not be wrapped by another mutable morph (i.e. the immutable morphs is stored in the `delegate` field of the mutable morph). The `owningMutableProxy` method may be used to determine the actual situation.

The example in *Listing 113* illustrates a typical use of the mirror macro in concrete model elements containing mutable attributes. The `Rectangle` fragment contains two mutable properties width and height. Since setting the same value to the width and height properties turns the rectangle into the square. Let us assume that the model of the morph containing the rectangle is:

```scala
Rectangle with (Unit or Square)
```

The morphing strategy associated with the morph checks the values of the width and height and in case they are the same the strategy promotes the `Square` fragment. However, the signal for the re-morphing must be sent from within the morph; i.e. from the `Rectangle` stub, as a notification about the change in the values of its properties. Thus the Rectangle fragment must have access to the context morph on which it invokes the `remorph` method. However, the `mirror` macro provides only the reference to the context immutable morph. Thus it is further necessary to invoke `owningMutableProxy` in order to obtain the reference to the context mutable morph, if any; in other words, if there is no mutable context morph then it is not possible to re-morph the context morph from the contained fragment since only mutable morphs may be re-assembled.

The two setters in `Rectangle` update the corresponding properties and call the `reshape` private method. The `reshape` method obtains the reference to the context morph (in a sort of

peculiar way) and invokes the `reshape` method on it, which may result in mutating the rectangle into a square.

Listing 113: Using the mirror macro to re-morph the context morph

```
@fragment
trait Rectangle {
  private var w: Float = 0
  private var h: Float = 0

  def width = this.w
  def width_=(w: Float) = {
    this.w = w
    reshape
  }

  def height = this.h
  def height_=(h: Float) = {
    this.h = h
    reshape
  }

  private def reshape {
    for (m <- mirror(this);
         mm <- m.owningMutableProxy) {
      mm.remorph()
    }
  }
}
```

The result returned by the `mirror` macro is optional, since the stub may not be part of any morph; in such a case the macro returns `None`. Similarly, the `owningMutableProxy` method also returns an optional result, as the immutable morph needn't be wrapped by any mutable morph.

7.3.4   Fragment Factories

The RI offers several macros producing fragment factories that may override the default factories used by macros compose and singleton. *Listing 114* outlines the fragment factory macros.

To override the default factory for a given fragment one declares an implicit variable prior to the invocation of `compose` or `singleton`. Then a factory instance created one of the above-mentioned macros is assigned to the variable, as shown in *Listing 115*.

.

Listing 114: Macros for overriding the default fragment factories used by recognizers

```
def frag[F]: FragmentFactory[F, Unit]
def frag[F, C](cfg: => C): FragmentFactory[F, C]
def single[F]: FragmentFactory[F, Unit]
def single[F, C](cfg: => C): FragmentFactory[F, C]
def expose[F](otherRecognizer: MorphKernel[_]): FragmentFactory[F, Unit]
def external[F](extStub: F): FragmentFactory[F, Unit]
```

Listing 115: Overriding the default singleton fragment factory by a non-singleton one (frag)

```
implicit val fragFact = frag[Rectangle]
val itemRec = singleton[Paper*Rectangle]
```

The preceding sample overrides the `Rectangle` factory only, while the factory of the `Paper` fragment is the singleton one by default.

The `frag[F]` macro creates an instance of the non-singleton factory of fragment F. *Listing 116* shows the effect of using the non-singleton factory for the `Rectangle` fragment.

Listing 116: The effect of using the non-singleton factory with a singleton recognizer

```
implicit val rectFact = frag[Rectangle]
val prRec = singleton[Paper*Rectangle]
val pr = prRec.~

pr.width = 100
pr.height = 100
pr.density = 1.0

pr.remorph

println(s"${pr.width},${pr.height},${pr.density}")
```

Prior to the re-morphing, the state of `pr` is updated. The `width` and `height` properties belong to `Rectangle`, while the density property belongs to `Paper`. On account of the fact that the non-singleton factory created the rectangle stub, the stub is re-created when re-morphing, which leads to loosing the previously set state. Thus the output will be:

```
0,0,1.0
```

The density is preserved, as the `Paper` stub was created by the singleton factory.

The `single[F]` macro works as the `frag[F]` with the exception that it creates a singleton factory of F.

The two macros `single` and `frag` have their overloaded versions that allow initializing the stubs externally. The overloaded versions have two type arguments and a single normal argument. The type arguments require the fragment type and the type of the fragment's configuration trait. The normal argument is used to pass an instance of the configuration trait. Provided that the configuration instance is passed by value (`cfg: => C`) the factory can get the fresh version of the configuration instance always when creating the stub.

The `RectangleInit` class implements the configuration trait `RectangleCfg` of the `Rectangle` fragment. Similarly the `CylinderInit` class implements the configuration trait `CylinderCfg` of the `Cylinder` fragment. The initialization classes extract the initial property values from the item record passed as their only constructor parameter (*Listing 117*).

Listing 117: Implementing a configuration trait using a map

```
class RectangleInit(itemData: Map[String, String]) extends RectangleCfg {
  override val width = itemData.get("width").toFloat
  override val height = itemData.get("height").toFloat
}
```

The possibility to initialize stubs from external objects allows designing concrete elements as immutable classes; i.e. the `Rectangle` trait may use "vals" instead "vars" inherited from its configuration trait. In order to initialize the stub of `Rectangle` from the item record an implicit variable is declared and initialized by the overloaded frag macro, as shown in *Listing 118*.

Listing 118: Initializing a fragment using an instance of its configuration trait

```
implicit val rectFact = frag[Rectangle, RectangleCfg](
  new RectangleInit(itemData))
val prRec = singleton[Paper*Rectangle)]
val pr = prRec.~
```

If the `itemData` record passed to the fragment factories is continuously updated then the new values will appear in the new stubs after re-morphing. This is the effect of the re-evaluation of the expression `new RectangleInit(itemData)`, which is passed by value to the frag macro.

```
itemData.put("width", "100")
itemData.put("height", "100")
pr.remorph
println(s"${pr.width},${pr.height}")
```

The preceding code should output the updated properties `width` and `height`.

```
100,100
```

The `expose[F]` macro allows reusing an existing stub from some recognizer in another recognizer, as illustrated in *Listing 119*.

Listing 119: Reusing a fragment stub from another recognizer using the `expose` macro

```
val prRec = singleton[Paper*Rectangle]

implicit val pf = expose[Paper](prRec)
implicit val rf = expose[Rectangle](prRec)
val bnkRec = singleton[Banknote*Paper*Rectangle]
```

The purpose of the `external[F]` macro is to use a normal instance instead of a stub. In contrast to the stub, the normal instance is created from a class, which was not generated by the RI. The `Triangle` type in *Listing 120* is a traditional class implementing `Shape`. The `external[F]` allows using the instance of `Triangle` in the recognizer as a fragment.

Listing 120: Using the `external` macro to use an ordinary object (not a fragment stub) in a recognizer

```
val triangle = new Triangle((0, 0), (1, 0), (0, 1))
implicit val trf = external[Shape](triangle)
val trRec = singleton[Shape*(Paper | Metal)]
```

Such an external instance can be accessed by other stubs in the morph; however, the external object cannot communicate with the other stubs. Using an external instance actually introduces the object schizophrenia; for example, if a method is invoked on the external instance and that method invokes another method that is actually overridden by some wrapper, the overridden method is never invoked.

7.3.5   Morphing Strategies

The morphing strategies were introduced in section 5.2. There are three fundamental and complementary strategies: promoting, masking and rating. For each strategy type there is a couple of macros in the Morpheus object.

The pivotal artifact in this section is the `MorphingStrategy` trait, which is implemented by morphing strategies in the RI (*Listing 121*).

Listing 121: `MorphingStrategy` trait

```
trait MorphingStrategy[M] {
  def chooseAlternatives(recognizer: MorphKernel[M])(
    morph: Option[recognizer.ImmutableLUB]): Alternatives[M]
}
```

The `RootStrategy[M]` class is the default implementation of this trait (*Listing 122*). It simply returns the default list of the alternatives, which is sorted according to the morph model expression.

Listing 122: RootStrategy class

```scala
case class RootStrategy[M]() extends MorphingStrategy[M] {
  override def chooseAlternatives(recognizer: MorphKernel[M])
               (morph: Option[instance.ImmutableLUB]): Alternatives[M] =
      recognizer.model.alternatives
}
```

The `chooseAlternatives` method is invoked by the recognizer, with which the strategy is associated. The method has two arguments: the first refers to the recognizer, while the second contains the optional reference to the context morph; i.e. the morph that is being remorphed. This reference is `None` if the morph has not been created yet.

The returned object is an instance of the `org.morpheus.Alternatives[M]` class, which represents the list of all alternatives of the model M. The recognizer invokes the `toMaskedList` method on the returned object and takes the head of the list as the template alternative for the new structure of the morph.

The strategies usually form a chain that transforms the initial `Alternatives[M]` instance into the final one. To transform the alternatives, the individual strategies in the chain may invoke the following methods:

- `promote(promotedAlts: Set[List[Int]])`: promotes the alternatives in `promotedAlts`
- `maskAll()`: deactivates all fragments in the mask; i.e. `mask = 0…0`
- `unmaskAll()`: activates all fragments in the mask; i.e. `mask = 1…1`
- `mask(fragments: Set[Int])`: activates the given fragments
- `unmask(fragments: Set[Int])`: deactivates the given fragments
- `rate(ratingFn: (List[FragmentNode], Double) => Double)`: rates the alternatives by means of `ratingFn`
- `strict()`: switches on strict mode (see below)
- `nonStrict()`: switches off the strict mode

The `Alternatives[M]` class further contains two methods converting its instances into a Scala list. The atlternatives are sorted with respect to the order given by the model expression tree (wich may differ from the original one due to promotions). Each alternative is accompanied with its rating.

- `toList()`: converts the alternatives to a Scala list. The list contains also the excluded alternatives.

- `toMaskedList()`: converts the alternatives to a Scala list. The list does not contain the excluded alternatives.

The following paragraphs are dealing with the macros that facilitate the creation and chaining of morphing strategies.

Promoting

The promote macro creates an instance of the promoting strategy for a given morph model `M`. The promoting strategy does not have to operate on the whole scale of the original morph model. Instead, it may specify a sub-model of the original model and select the best alternative from this sub-model. All original alternatives conforming to the selected best alternative from the sub-model are then promoted.

The type argument of the `promote` macro is used to specify the sub-model expression. The first normal argument is the original morph model instance and the second argument is the function that selects the winning alternative from the sub-model. This function returns the index of the alternative wrapped in `Some` or `None` if the function cannot determine the alternative. The selector function may use the optional argument, which contains the morph being re-morphed. This argument is `None` when the strategy is used for the first time; i.e. when the recognizer is creating the morph.

Listing 123: The `promote` macro

```
def promote[S](model: MorphModel[_])(
  s: (Option[model.ImmutableLUB]) => Option[Int]): MorphingStrategy[model.Model]
```

The example in *Listing 124* shows how the promote macro may be used to create a strategy that selects between the square and non-square rectangle shapes. The `rectModel` morph model consists of two alternatives: the first represents a non-square rectangle, while the other is for squares only. The strategy examines the values of the width and height properties of the current morph and if they are the same it returns `Some(1)`; i.e. the second alternatives. Otherwise it returns `Some(0)`. If there is no context morph (i.e. the morph is being created), then the function returns `None`.

Listing 124: Using the `promote` macro to select between the square and non-square rectangle shapes

```
val rectModel = parse[Rectangle*(Unit | Square)](true)

val rectStrat = promote[Unit | Square](rectModel)({
  case None => None
  case Some(morph) =>
   if (morph.width == morph.height)
      Some(1)
```

```
        else
            Some(0)
  })
  val rect = singleton(rectModel, rectStrat).~
```

The code snippet in *Listing 125* first modifies the dimensions of the rectangle so that they are different. Then the morph is re-morphed and checked that it is not a square. Then the height is modified so as to have the same value as the width property and the morph is re-morphed again. The morph must be now an instance of Square.

Listing 125: Reshaping the rectangle by modifying its attributes

```
  rect.width = 100
  rect.height = 200
  rect.remorph

  assert(!rect.delegate.isInstanceOf[Square])

  rect.height = 100
  rect.remorph

  assert(rect.delegate.isInstanceOf[Square])
```

There are also several overloaded versions of the promote macro, which allow strategy chaining. The macro in *Listing 126* is the simplest one, whose first argument is the reference to the previous strategy and the second argument is the morph alternative selector function. This selector function selects the alternative according to the outer conditions only, since it does not obtain the reference to the context morph, as is the case in the preceding example.

Listing 126: A version of the promote macro for chaining with other strategies

```
  def promote[M, S](prevStrat: MorphingStrategy[M],
                    sw: () => Option[Int]): MorphingStrategy[M]
```

The code snippet in *Listing 127* illustrates the chaining of two strategies. The former determines whether the outer object, which is represented by the itemData record, is a rectangle or cylinder, while the latter selects the material. If the strategies cannot determine the right alternative they return None.

Listing 127: An example of strategy chaining

```
  val strat1 = promote[Rectangle | Cylinder](rootStrategy(itemModel),
    if (itemData.get("shape") == "rectangle") Some(0)
    else if (itemData.get("shape") == "cylinder") Some(1)
    else None)
```

```
val strat2 = promote[Paper | Metal](strat1,
   if (itemData.get("material") == "paper") Some(0)
   else if (itemData.get("material") == "metal") Some(1)
   else None)

val itemRec = singleton(itemModel, strat2)
```

Masking

The fragment mask plays the key role in the masking strategy. By default, the fragment mask indicates the allowed fragments; i.e. if an alternative contains a single fragment that is not allowed it is excluded from the list of alternatives. Given that the `alt` is the fragment bit mask of an alternative then the alternative is not excluded from the list of alternatives as long as the following expression holds:

```
alt & mask == alt
```

In other words, the zeros in the mask specify the fragments, which must not be in any alternative. It follows that the mask having all bits set to 1 excludes no alternative. Such a setting may be achieved by using the strategy created by the `maskAll` macro (*Listing 128*).

Listing 128: The `maskAll` macro

```
def maskAll[M](prevStrat: MorphingStrategy[M]): MorphingStrategy[M]
```

The above-mentioned condition for the exclusion of alternatives suits well in the situations in which the morph model may be expressed in the following way, where the branches `Bi` in the factored-out sub-model are disjoint; i.e. no two branches have a common fragment.

```
M = (B₁ | … | Bₙ).Mrest
```

A typical example of such a model is the well-known scanned item morph model.

```
(Rectangle | Cylinder).(Paper | Metal).Item
```

Then the strategy chain may consist of two *exclusive* masking strategies, each operating on a sub-model corresponding to one dimension in the morph model. The exclusive masking/unmasking strategy activates/deactivates (masks/unmasks) all fragments from the selected sub-model alternative and at the same time it deactivates/activates the other sub-model fragments. It follows that this strategy may override the modifications in the list of alternatives made by the preceding strategies.

Listing 129: Chaining two strategies created by two invocations of the `mask` macro

```
val itemModel = parse[Item*(Rectangle | Cylinder)*(Paper | Metal)](true)
val strat1 = mask[Rectangle | Cylinder](rootStrategy(itemModel),
```

```
      if (itemData.get("shape") == "rectangle") Some(0)
      else if (itemData.get("shape") == "cylinder") Some(1)
      else None)

  val strat2 = mask[Paper | Metal](strat1,
      if (itemData.get("material") == "paper") Some(0)
      else if (itemData.get("material") == "metal") Some(1)
      else None)

  val itemRec = singleton(itemModel, strat2)
```

Except the usage of the `mask` macro is the *Listing 129* quite similar to that using the `promote` macro in *Listing 127*. In this case the effect of using the mask or promote macros is identical. The difference becomes obvious only when the item morph is assigned to a morph reference (see 7.3.6), such as:

```
  val banknoteRef: &[$[Banknote].Paper.Rectangle] = itemRec
```

Let us assume that the item morph represents a metal cylinder. In case the item morph was constructed with the assistance of the masking strategy, the assignment will fail with the complaint that there is no source alternative that could be used as the template for the target banknote morph.

On the other hand, as long as the promoting strategy assisted in the creation of the item morph instead then the assignment will succeed despite the source morph is not compatible with the target model.

The promoting strategy is thus suitable in the situations when the morph may be re-morphed to any alternative. Such a morph will then be fully flexible and will cause no problem when being assigned to morph references.

Conversely, if the morph is not fully flexible, e.g. it is tightly connected with the input data and the fragment factories could not initialize the fragments not corresponding to the input data, the morph should be constructed with the help of the masking strategy.

Using the macros operating on sub-models actually avoids the activation of two mutually exclusive fragments since the macros verify internally that the main morph model conforms to the sub-model.

```
  mask[Rectangle*Cylinder](rootStrategy(itemModel), …) // does not compile
```

The `Alternatives[M]` class may use another condition for excluding alternatives from the list. Then it is said that the alternatives are in the strict mode. One may use the `strict` macro to turn the strict mode on (*Listing 130*).

Listing 130: The `strict` *macro*

```
def strict[M](prevStrat: MorphingStrategy[M]): MorphingStrategy[M]
```

The `strict` macro changes the condition used to exclude alternative from the list of alternatives so that an alternative is not excluded as long as it does not contain all fragments allowed by the mask; i.e. this condition may be expressed as follows:

```
alt & mask == mask
```

In other word, the ones in the mask specify the fragments, which must be in any alternative. It follows that the mask having all its bits set to `0` excludes no alternative. Such a mask may be created using the `unmaskAll` macro.

```
def unmaskAll[M](prevStrat: MorphingStrategy[M]): MorphingStrategy[M]
```

The strict mode is useful in the situations, in which the model may be expressed as follows:

```
M = (B₁ | … | Bₙ).….(B₁ | … | Bₙ).M_rest
```

In case the number of the occurrences of the factored-out union group is the same as the number of the branches, the morph model is analogous to the inclusive "or", as illustrated in the following example.

```
(B1 | B2).(B1 | B2) = B1 | B1.B2 | B2
```

In the light of this analogy, the single occurrence of the group is analogous to the exclusive "or". Interestingly, there may exist a number of hybrid cases, when the number of the occurrences is greater than one and less than the number of the branches. It gives rise to a whole spectrum of "or"-like models with the exclusive "or" as one extreme and the inclusive "or" as the opposite extreme (see 5.1.2).

Such morph models may be driven by a chain of *cumulative* masking strategies, each operating on the same sub-model corresponding to the union group. The cumulative masking/unmasking strategy activates/deactivates (masks/unmasks) all fragments from the selected alternative. As its name suggests, this strategy does not deactivate/activate any fragment activated/deactivated by a preceding strategy.

Let us illustrate the usage of the strict model and the cumulative masking strategy on the example in *Listing 131* modeling a rectangle that may be decorated by one or two decorators selected from three ones.

Listing 131: An example of construction of a cumulative masking strategy

```
type Decors = (DecorA | DecorB | DecorC)

val rectModel = parse[Rectangle * Decors * Decors](true)

var firstDecorId = 0 // DecorA
var secondDecorId = 1 // DecorB

val strat0 = unmaskAll(rootStrategy(rectModel))
val strat1 = mask_+(strat0, () => Some(firstDecorId))
val strat2 = mask_+(strat1, () => Some(secondDecorId))
val strat3 = strict(strat2)

val rect = singleton(reactModel, strat3).~
// the Rectangle is decorated by DecorA and DecorB

firstDecorId = secondDecorId = 2
rect.remorph
// the Rectangle is decorated by DecorC only
```

The requirement that only one or two decorators may be used is expressed by joining the Decors group with itself. The two mutable variables firstDecorId and secondDecorId select the active decorators. The first strategy in the chain clears all bits; the following two strategies cumulatively activate the selected decorators and the final strategy turns the strict mode on.

Rating

In contrast to the promoting and masking strategies, which exactly determine the fragments that must be present in the winning alternative, the rating strategy rates individual alternatives according to some criteria. The rating strategies usually form a chain, in which each strategy represents a certain rating criteria. The alternative with the highest rating (fitness) then becomes the winner alternative.

The rate macro creates an instance of the rating strategy.

Listing 132: The rate macro

```
def rate[S](prevStrat: MorphingStrategy[_],
    altRatingFn: () => Set[(Int, Int)]): MorphingStrategy[prevStrat.Model]
```

Just as the promote and mask macros, the rate macro also allows specifying a sub-model in the type argument for easier rating. The first normal argument refers to the previous strategy and the second argument is used to pass a function rating a subset of the alternatives of the sub-model S. The returned set consists of integer pairs, whose first number indicates the

rated alternative and the second number represents the rating. The returned set may be empty, as long as the strategy cannot rate any alternative under the given circumstances.

To illustrate an application of the rating strategy, let us examine the example in *Listing 133* modeling a breakfast. The chained rating strategies rate individual breakfast alternatives with respect to the characteristics of the guest who is going to eat the breakfast. There are four characteristics considered: female, male, child and obesity.

Listing 133: Using the `rate` *macro*

```
val breakfastModel =
  parse[(Tea | Coffee | Beer)*(Tuna | Ham | Egg)*(Cucumber | Apple)](true)

val rootStrat = rootStrategy(breakfastModel)
val femaleStrat = rate[Beer](rootStrat,
    () => if (guest.isFemale) Set((0, -1)) else Set()) // women dislike beer
val maleStrat = rate[Cucumber | Tuna](femaleStrat,
    () => if (guest.isFemale) Set((0, -2), (1, -1))) else Set()
val childStrat = rate[Coffee | Beer | Tuna](maleStrat,
    () => if (guest.age < 15) Set((0, -1), (1, -2), (2, 2)) else Set()
val obeseStrat = rate[Beer | Tuna | Cucumber](childStrat,
    () => if (guest.weight > 100) Set((0, -3), (1, 1), (2, 1)) else Set()

val breakfast = singleton(breakfastModel, obeseStrat).!
```

The breakfast model consists of three dimensions: beverage, meat and fruit/vegetable. The female strategy takes into account the women's dislike of beer; therefore if the guest is a female then the returned set consists of a single pair `(0, -1)` rating the beer alternatives negatively. The male strategy reflects the men's dislike of cucumber and tuna. The child strategy takes into account the fact that children should not drink beer or coffee, but it recommends tuna as healthy meat. Regarding obese people, the `obeseStrat` suppresses beer and recommends tuna and cucumber.

Custom Strategies

As long as the prefabricated strategies described in the preceding paragraphs do not suffice one may implement a custom morphing strategy. The `MorphingStrategy[M]` trait contains the single method `choseAlternatives`, which is described in the beginning of this section.

Let us demonstrate an implementation of `MorphingStrategy[M]` on the example in *Listing 134*, which is a continuation of the last example. The goal is to incorporate into the strategy chain a special morphing strategy, which will eliminate all alternatives containing the products that are out of the stock.

The `OutOfStockStrategy` class implements the trait `MorphingStrategy` and has two constructor arguments. The first argument is the reference to the previous strategy, which allows

incorporating `OutOfStockStrategy` into the strategy chain. The second argument refers to the stock object keeping the track of the unavailable products.

Listing 134: Implementation of a custom morphing strategy

```
class OutOfStockStrategy[M](prevStrat: MorphingStrategy[M], stock: Stock)
extends MorphingStrategy[M] {
  override def chooseAlternatives(recognizer: MorphKernel[M])(
    morph: Option[instance.ImmutableLUB]): Alternatives[M] =
      val prevAlts = prevStrat.chooseAlternatives(recognizer)(morph)
      val outOfStockBits: Set[Int] = stock.unavailableFood()
      prevAlts.unmask(outOfStockBits)
}
```

The implementation is pretty straightforward. First, the alternatives object is obtained from the preceding strategy in the chain. Then the stock object is asked to return the unavailable products as a bit set, which is compatible with the fragment indices in the morph model. And finally, the unmask method is called on the `prevAlts` object, creating so a clone with the unavailable product fragments deactivated.

The new strategy may be incorporated into the strategy chain as follows:

```
val outOfStockStrat = new OutOfStockStrategy(obeseStrat, stock)
val breakfast = singleton(breakfastModel, outOfStockStrat).!
```

7.3.6   Morph References

Section 5.1.9 deals with the conformance between two morph models, which is actually an application of the Liskov principle on morph models. That section examines under which conditions a source morph model may substitute a target morph model. In practice it means that one may declare a reference variable, a method parameter or a return value of the target morph model type in order to reference any (source) morph or recognizer whose morph model conforms the morph model of the reference.

The RI offers two special classes that serve as the morph model references. The `org.morpheus.&[T]` class is used to refer the *surjectively* conforming morphs or recognizers, while `org.morpheus.~&[T]` refers to *partially* conformant morphs or recognizers. In other words, when using `&[T]` all alternatives in the target model `T` must have their counterparts in the source model. On the contrary when using `~&[T]` at least one target alternative must have its counterpart in the source model.

Note: The current version of RI does not support injective and total morph references.

Note: The RI uses the mechanism of implicit conversions to convert the standard Scala reference to the morph reference (Scala Documentation, n.d.).

The example in *Listing 135* shows a valid assignment of the item morph to the `rectRef` reference whose target model contains only one alternative with the `Rectangle` fragment.

Listing 135: Assigning a morph to a morph reference

```
val item = singleton[(Rectangle | Cylinder)*(Paper | Metal)].!
val rectRef: &[Rectangle] = item
```

The compiler aborts the compilation as long as the assignment is not valid.

In order to use the referenced instance one must *dereference* it first. In the RI, there are two overloaded versions of the * macro dereferencing morph references (*Listing 136*).

Listing 136: Dereference macros

```
def *[M](ref: MorphKernelRef[M, _], placeholderFactories: Any*): MorphKernel[M]

def *[M](ref: MorphKernelRef[M, _],
         strategy: MorphingStrategy[_],
         placeholderFactories: FragmentFactory*): MorphKernel[M]
```

The * macro produces a recognizer operating on the target model specified in the reference argument `ref`. This recognizer is used to construct morphs in the standard way with the exception that the recognizer reuses the stubs from the referenced recognizer. Only if the target morph model contains some placeholders the fragment factories of all placeholders must be specified in the `placeholderFactories` argument. The strategy argument in the second version of the * macro allows overriding the default morphing strategy.

The snippet in *Listing 137* illustrates the simplest dereferencing of a morph reference:

Listing 137: Dereferencing a morph reference

```
val rectRec = *(rectRef)
val rect = rectRec.!
```

And the example in *Listing 138* shows how to specify a placeholder in the target model of the reference. One uses type `org.morpheus.Morpheus.$[T]` to wrap the placeholder elements in the target model.

Listing 138: Using a placeholder in a morph reference

```
val banknoteRef: &[$[Banknote]*Rectangle*Metal] = item
val banknoteRec = *(banknoteRef, single[Banknote])
val banknote = banknoteRec.!
```

Since the placeholder is not expected in the source morph model by default, one must provide the dereferencing macro * with the factory for all placeholders. It is possible to use all factory macros that are described in 7.3.4.

It is important to reiterate that the creation of the new morph from the dereferenced recognizer may fail as long as the source morph was built using a masking strategy that eliminated all suitable alternatives for the new morph. Such a condition cannot be easily revealed at compile-time since the decision which alternatives will be eliminated is done at run-time. Thus the failure to construct a morph from a reference should be considered an application (expected) exception.

Note: The RI throws `org.morpheus.NoViableAlternativeException`, which should be caught and handled by the dereferencing code.

The placeholder operator $ may wrap whole sub-models. In such a case all elements in the wrapped sub-model are treated as placeholders. If the $ operator is used in the sub-model, then the wrapped elements are no-longer considered placeholders; i.e. $ is contextual in the sense that it wraps placeholders if there is no superior $ or if the superior $ wraps non-placeholders. And $ wraps non-placeholders if the superior $ wraps placeholders.

```
A*$[B*(C | D | $[E * $[F]])] = A*$[B]*($[C] | $[D] | E*$[F])
```

Coupling References

The following paragraphs study the coupling of references; i.e. the situations when a reference refers to a morph obtained from another reference. For this purpose let us consider the following two *primary* references. The `shapeRef1` reference is surjective and `shapeRef2` partial.

```
val shapeRef1: &[Rectangle | Cylinder] = item // surjective
val shapeRef2: ~&[Rectangle | Cylinder] = item // partial
```

Listing 139 outlines a couple of examples of the *secondary* references, which are initialized by the recognizers obtained by dereferencing the primary references.

Listing 139: Valid and invalid secondary assignments to various morph references

```
//surjective-surjective
val rectRef1: &[Rectangle] = *(shapeRef1)
//partial-surjective
val rectRef2: &[Rectangle] = *(shapeRef2) // does not compile
//surjective-partial
val rectRef3: ~&[Rectangle] = *(shapeRef1)
//partial-partial
val rectRef4: ~&[Rectangle] = *(shapeRef2) // does not compile
//partial-partial
val rectRef5: ~&[Rectangle | Cylinder | Cuboid] = *(shapeRef2)
//surjective-surjective
val rectRef6: &[Rectangle | Cylinder | Cuboid] = *(shapeRef1)// does not compile
```

The first assignment is fine, since the sourceRef1 certainly can provide all alternatives from the `Rectangle | Cylinder` model, as it is surjective. The only target alternative `Rectangle` will thus always find its counterpart in the source model.

On the contrary, the second assignment will abort the compilation, since there is no guarantee that the partial reference `shapeRef2` can provide the `Rectangle` alternative. It might happen that the item's morph model contained a `Cylinder` alternative but none `Rectangle` alternative.

The third assignment is ok, as it is a weaker version of the first assignment.

The fourth assignment will fail, from the same reason that was mentioned in the case of the second assignment.

On the other hand, if the target model is extended with `Cylinder` and `Cuboid` as shown in the fifth assignment, then one of the two source alternatives in `shapeRef2` will certainly map onto the `Rectangle or Cylinder` target alternatives and thus the condition for the partial reference is satisfied.

The extension of the target model will not help in the case of the surjective sixth assignment; the `Cylinder` alternative will not find its counterpart in `shapeRef1`.

Type-casting morphs

The morph references may be used as a tool for casting a morph to another type in a type-safe manner. The following macro `asMorphOf[T]` allows casting the source morph into the type specified in the type argument `T`.

Listing 140: The asMorphOf macro

```
def asMorphOf[T](srcMorph: Any,
                 placeholderFacts: FragmentFactory*): T with MorphMirror
```

The type `T` must describe a one-alternative morph model to which the source morph's model conforms. Since model `T` has only one alternative, it is irrelevant to take into account the fact whether the conformance should be partial or surjective. In such a case it is possible to use the morph model type `T` as the return type of the macro.

The macro is in fact an abbreviation of the following code:

```
val ref: ~&[T] = srcMorph
*(ref, placeholderFacts).!
```

The next statement shows a sample usage of the macro:

```
val rect: Rectangle = asMorphOf[Rectangle](item)
```

The target model in `asMorphOf` may also contain placeholders. In such a case the factories must be passed for each placeholder when invoking the macro.

```
val banknote = asMorphOf[$[Banknote]*Rectangle*Paper](item, single[Banknote])
```

Self Reference

The RI provides the `self` macro to obtain the recognizer operating with the morph model of the contextual dependencies of a fragment. Such a recognizer may be used to obtain the context morph or be assigned to other references in the implementation of a concrete model element.

The `self` macro is illustrated in the example in *Listing 141*. The `ItemRenderer` fragment is designed to render a scanned item, to which it is joined by means of the `asMorphOf` macro:

```
val itemRenderer = asMorphOf[$[ItemRenderer]](item, single[ItemRenderer])
itemRenderer.renderItem
```

The `ItemRenderer` fragment specifies its dependencies as the two-dimensional morph model of the scanned item. The default dependency conformance level is partial (the default one) and thus at least alternative dependency must find its counterpart in the morph model of the contextual morph.

Listing 141: Using the `self` macro to obtain the reference to the contextual morph

```
@fragment
trait ItemRenderer {
   this: (Rectangle | Cylinder)*(Paper | Metal) =>

   def renderItem() {
      val rendRef: &[$[RectangleRenderer | CylinderRenderer]] = self(this)
      val rend = *(rendRef,
                      single[RectangleRenderer],
                      single[CylinderRenderer]).!
      rend.render()
   }
}

@fragment
trait RectangleRenderer extends ShapeRenderer {
   this: Rectangle*(Paper | Metal) =>

   def render() {
      mirror(this) match {
         case pr: Rectangle with Paper =>
            // render a paper rectangle
            …
```

```
        case mr: Rectangle with Metal =>
            // render a metal rectangle
            …
      }
    }
  }
  // the CylinderRenderer would be designed analogously
```

The `renderItem` method first obtains the dependency model recognizer and assigns it to the `rendRef`. The `rendRef` target model consists of two particular recognizers for rectangles and cylinders. Their factories must therefore be passed when dereferencing the `rendRef` to obtain the `rend` morph, which will render the item.

The `RectangleRenderer` fragment depends on context model `Rectangle*(Paper | Metal)`. It uses the mirror macro explained earlier to obtain the context morph. The actual type is determined and handled by means of the standard match construct. The `CylinderRender` would be designed analogously.

There is also the macro `&&`, which returns a morph reference to the contextual dependencies, instead of the standard reference as the `self` macro. The `self` macro is actually an abbreviation for the following code:

```
val selfRef = &&(this)
val self = *(selfRef)
```

8   Final Case Study

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 described in Prototypical Analysis.

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 (see *Figure 75*). The contractions of the muscles bend and morph various facial features, such as lips, lids, eyebrows etc., and render so the characteristic expressions. It has been shown by Darwin in (Darwin et al., 1998) that the emotional expressions are mostly inherited and are more or less identical across all cultures.

Figure 75: Electrically stimulated facial expressions (Duchenne, 1862)

8.1   Analysis and Design

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: Eybrows, Upper Lid, Lower Lid, Upper Lip, etc.

The first three sub-sections of this analysis deal with the application tiers, the fourth illustrates the assembling of the tiers and the last one provides a summary. For the sake of brevity the code presented in this section consists of the essential parts only. The complete source code may be consulted or checked out from the project's repository (Šlajchrt, 2015).

8.1.1   Emotions Model

As mentioned above, throughout this analysis we will follow the raw guidelines of the prototypical analysis. According to the first step the analyst should acquire as much knowledge as possible on the level of individual phenomena. In the context of this application the analyst may begin with examining a large set of pictures of faces displaying various emotions. The goal is to become familiar with the facial expressions and to be able to exactly determine the emotions behind the expressions, such as in *Figure 76*.

Note: Throughout this section, a number of generated facial images are used. These images are generated by software developed in Grimace Project by Fadrus and Spindler (Spindler & Fadrus, n.d.; Spindler, n.d.).

Figure 76: Schematic examples of human expressions (Spindler & Fadrus, n.d.)

The next step in the analysis is the identification of prototypical exemplars among the phenomena. The analyst comes to the conclusion that all examined face examples did not dis-

play more than two of the six human emotions: joy, surprise, fear, sadness, disgust and anger. This finding makes possible for her to identify six prototypes of the basic emotions (the first six faces in the upper row in *Figure 76*).

Having finished the identification of the prototypes, the analyst proceeds to abstracting properties from the prototypes. It would be, however, quite difficult for the analyst to state anything more specific about, for instance, the joy prototype than that it includes the joy-ness property only. The basic emotions are already so abstract that no internal structure can be described in individual emotion prototypes.

It follows that the intension of the concept of human emotion consists of six properties each corresponding to one basic emotion prototype. The only constraint on these properties derived from the initial observation is that no more than two properties may occur in a phenomenon.

At this stage the analyst can construct the morph model representing the concept. The basic emotions model consists of six alternatives each representing one emotion prototype. This first approximation assumes that the emotions are mutually exclusive (*Listing 142*). The `Emotions1D` type declares the morph model expression of the basic emotions model. The `Unit` type is used to represent the absence of any emotion.

Listing 142: The basic (one-dimensional) model of emotions

```
type Emotions1D = Unit | Joy | Surprise | Fear | Sadness | Disgust | Anger
```

In order to represent the real emotional states more faithfully, the basic model must be expanded to produce more alternatives. This expansion cannot be arbitrary; instead, it must be driven by the constraints and dependencies between properties. Nonetheless, the only constraint identified in this analysis is that the expanded morph model must not produce any alternative containing more than two emotion properties. Such an expanded morph model may be obtained by joining the basic model with itself (*Listing 143*).

Listing 143: A more detailed (two-dimensional) model of emotions

```
// Emotions2D is Emotions1D "squared"
type Emotions2D = Emotions1D * Emotions1D
```

The final morph model of the emotion tier consists of `Emotions2D` and one additional fragment `Base`, which serves as the base fragment for the concrete emotion fragments and determines the lowest upper bound (LUB) of the expression (*Listing 144*).

Listing 144: The model of emotions used in the application

```
type Emotions = Base * Emotions2D
```

The three above-mentioned type definitions are actually type members of the structural type `EmotionsType`, which is used as a morph model type constructor (*Listing 145*). The emotion fragment types are not actually the real fragment types; instead, they are the type parameters of the enclosing `EmotionsType` type.

Listing 145: Abstract type (a template) of the emotions model

```
type EmotionsType[Base, Joy, Surprise, Fear, Sadness, Disgust, Anger] = {
  type Emotions1D = (Unit | Joy | Surprise | Fear | Sadness | Disgust | Anger)
  type Emotions2D = Emotions1D * Emotions1D
  type Emotions = Base * Emotions2D
}
```

The actual morph model type is constructed by substituting the type parameters by the actual fragment types (*Listing 146*).

Listing 146: Concrete type of the emotions model

```
type EmoTypesActual = EmotionsType[EmotionBase,
  JoyFragment * JoyWrapper, SurpriseFragment * SurpriseWrapper,
  FearFragment * FearWrapper, SadnessFragment * SadnessWrapper,
  DisgustFragment * DisgustWrapper, AngerFragment * AngerWrapper]
```

Note: Such a separation of the generic morph model type structure from the actual one makes it possible to easily declare specific morph model types for various environments; for instance one type can be used in tests, while another one in the production.

The `EmotionBase` fragment (*Listing 147*) provides an interface by mean of which the user communicates with the emotions tier. In this simplest case the user can get only the active emotions and their levels. The `EmotionBase` fragment, just as all fragments in the emotions tier, is designed as an immutable fragment whose state is initialized during the construction through its configuration trait `EmotionBaseInit` (see p. 153). An instance of the configuration class `EmotionBaseInitData` carrying the selected emotions and their levels is passed on the fragment factory, through which the configuration object gets into the fragment's instance, as shown later in the assembling code (8.1.4).

Listing 147: `EmotionBase` fragment and its configuration trait

```
// The configuration trait
trait EmotionBaseInit {
  val initEmoLevels: Map[Int, Float]
}

// The configuration case class for two co-occurring emotions
case class TwoEmotionsInitData(emo1Lev: Option[(Int, Float)],
    emo2Lev: Option[(Int, Float)]) extends EmotionBaseInit {
```

```
      override val initEmoLevels: Map[Int, Float] =
          (emo1Lev.toSet ++ emo2Lev.toSet).toMap
}


@fragment
trait EmotionBase extends dlg[EmotionBaseInit] {
    def actualEmoLevels: List[(Int, Float)] = Nil
}
```

The `EmotionBaseInit` trait does not constrain the user to specify at most two emotions only as assumed by the emotions morph model; considering that there may exist other emotion models the configuration trait must remain independent from the morph model structure and allow specifying an arbitrary number of co-occurring emotions.

Through the `initEmoLevels` methods one may obtain the levels of the emotions specified during the construction. The methods return `None` in case no emotion is specified.

On the contrary, the `actualEmoLevels` method allows the user to retrieve the levels of the active emotions only, which may differ from the values in the `initEmoLevels`. As long as the emotions morph model allows at most two simultaneous emotions, `actualEmoLevels` will return a list of at most two elements regardless how many emotions are specified in `initEmoLevels`.

Each emotion fragment's wrapper overrides the `actualEmoLevels` method and adds the respective emotion identifier and the level to the resulting list (*Listing 148*). In effect, only the wrappers present in a morph will contribute to the list returned by `actualEmoLevels`.

Each emotion is represented by a couple of traits: a fragment and its wrapper. This design follows the orthogonality rule, which dictates to separate the part introducing new members from the part overriding the existing members.

Listing 148: JoyFragment and JoyWrapper traits

```
@fragment
trait JoyFragment {
  this: EmotionBase =>

  lazy val joyLevel = initEmoLevels(joy)
}


@fragment @wrapper
trait JoyWrapper extends EmotionBase {
  this: JoyFragment =>

  override def actualEmoLevels: List[(Int, Float)] =
      (joy, joyLevel) :: super.actualEmoLevels
}
```

The `JoyFragment` fragment only consists of the single member `joyLevel` returning the level of joy. The value is retrieved lazily from the map of the emotion levels held in `Emotion-Base`, on which `JoyFragment` depends.

The `JoyWrapper` wrapper depends on `JoyFragment`, from which it retrieves the joy level. The joy level together with the joy identifier is appended to the list of the actual emotion levels returned by the overriding `actualEmoLevels` method.

Emotions Context

In this section, the individual emotion properties are bound to the context variables representing the phenomenon being modeled, i.e. a particular emotional state. Here, the analyst puts together the building blocks introduced in the preceding section to create the emotions tier instance. The emotion tier is represented as a whole by the `EmotionsContext` class and its companion object.

The `EmotionsContext` object (*Listing 149*) encapsulates the static components of the tier, which are the emotions morph model type and the morph model parsed from the type.

Listing 149: `EmotionsContext` object

```
object EmotionsContext {
  type EmoTypesActual = EmotionsType[EmotionBase,
      JoyFragment * JoyWrapper, SurpriseFragment * SurpriseWrapper,
      FearFragment * FearWrapper, SadnessFragment * SadnessWrapper,
      DisgustFragment * DisgustWrapper, AngerFragment * AngerWrapper]

  val model = parse[EmoTypesActual#Emotions](true)
}
```

The remaining tier components are wrapped into the `EmotionsContext` class representing essentially the emotions recognizer and its input. In contrast to the emotions model, there may be many instances of the emotions recognizer, which is why it is held in a class and not in a singleton object.

The `emo1Lev` and `emo2Lev` variables (*Listing 150*) represent the input (or outer, contextual) object emotions and their levels. In other words, these two variables may be considered the objective reality (signals) to be recognized and modeled by the emotions recognizer.

Listing 150: Variables `emo1Lev` and `emo2Lev` determine the combination of two concurrent emotions

```
var emo1Lev: Option[(Int, Float)] = None
var emo2Lev: Option[(Int, Float)] = None
```

The user is supposed to set directly these variables right before creating the emotions morph or before re-morphing the morph.

Note: In a more complicated (and realistic) version of this case study the two variables would not be set explicitly, instead, their values could be obtained from an analysis of the occurrence of emotional words in a text, for instance.

Another important component of the tier is the morphing strategy used by the emotions recognizer (*Listing 151*). The strategy is created in the initialization block of the strategy variable.

Listing 151: Morphing strategy of the emotions morph model

```scala
val strategy: MorphingStrategy[model.Model] = {
  var strat: MorphingStrategy[model.Model] = unmaskAll(rootStrategy(model))

  strat = maskFull_+[EmoTypesActual#Emotions1D](model)(strat, _ => {
    for (el <- emo1Lev) yield el._1
  })

  strat = maskFull_+[EmoTypesActual#Emotions1D](model)(strat, _ => {
    for (el <- emo2Lev) yield el._1
  })

  strat = strict(strat)

  strat
}
```

The type argument of the `MorphingStrategy` trait is taken from the model's type member `Model`. This type member contains the morph model type expression, from which the model was parsed.

The strategy chain follows the cumulative masking pattern (see p. 176) selecting independently two emotions according to the values in the input variables. The first strategy in the chain clears the fragment mask, the following two pick the emotion fragments and the last turns on the strict mode demanding that only the alternatives containing the selected emotion fragments are valid. The two emotion fragment selection strategies operate with `EmoTypesActual#Emotions1D` type consisting of mutually exclusive emotion fragments including `Unit` to allow selecting no emotion.

As explained earlier, the `EmotionBase` fragment is designed as an immutable fragment initialized from its configuration object. On account of this fact it is necessary to override the default factory of the `EmotionBase` fragment by an implicit variable declared prior to the creation of the recognizer and initialized by the (non-singleton) `frag` macro. This macro is given an instance of the `TwoEmotionsInitData` configuration class initialized from the input variables.

Listing 152: Constructing the emotion recognizer

```
// Overriding fragment factory with the configuration object
implicit val emoBaseFact = frag[EmotionBase, EmotionBaseInit](
    TwoEmotionsInitData(emo1Lev, emo2Lev))

val recognizer = compose(model, strategy)
```

The recognizer is instantiated by means of the `compose` macro, which uses non-singleton fragment factories for the remaining fragments and wrappers in the model (*Listing 152*).

Usage

The dynamics of the emotions tier may be examined by the code in *Listing 153*.

Listing 153: Executing the emotions model

```
val emoCtx = new EmotionsContext
val emoMorph = emoCtx.recognizer.~

println(emoMorph.myAlternative)
println(emoMorph.actualEmoLevels)

emoCtx.emo1Lev = Some((joy, 0.7f))
emoCtx.emo2Lev = Some((surprise, 0.2f))

emoMorph.remorph

println(emoMorph.myAlternative)
println(emoMorph.actualEmoLevels)
```

First, an instance of the tier is created by instantiating class `EmotionsContext`. Next, a local copy of the default mutable morph of the emotions recognizer is stored to the `emoMorph` variable. Then the composition of the morph and the actual emotion levels are printed for a comparison with the subsequent outputs. After that, the input emotions are specified in the context along with their levels and the morph is re-morphed. Printing the new composition of the morph and the actual emotion levels allows comparing the current morph with the original one.

Listing 154: The output produced by the code in Listing 153

```
List(EmotionBase)
List()
List(EmotionBase, SurpriseFragment, SurpriseWrapper, JoyFragment, JoyWrapper)
List((1,0.7), (2,0.2))
```

8.1.2   Muscles Model

The used muscles model is the most complex one in the application. It consists of 26 muscles (*Figure 77*), which is still only a subset of the real facial muscles, and furthermore, it has the highest resolution (i.e. the number of alternatives). The analyst identifies those 26 muscles as the properties of the concept of the human facial musculature.

The simplest approach assumes that all muscles may be stimulated simultaneously. This assumption is quite oversimplifying, as in reality the real facial muscles have complex relationships among themselves. For example, people use a different set of muscles when they frown than when they smile. Furthermore, there are pairs of antagonist muscles, such as the muscles turning the head left and right or up and down, of which only one may contract. The muscles used in the model does not contain such antagonist muscles, thus it is tolerable to consider all muscles in the model as simultaneously stimulable.

Figure 77: A simplified schema of facial musculature (Spindler & Fadrus, n.d.)

Such a concept of the human facial musculature would include only one prototype represented by the alternative composed of those 26 muscles.

It directly leads the analyst to the identification of the only property constraint, according to which all properties, i.e. muscles, always occur (or stimulated) simultaneously.

The corresponding morph model can be then expressed as a join of all 26 muscles plus one auxiliary base fragment (*Listing 155*).

Listing 155: Basic model of facial musculature

```
type Muscles = Muscle * M0 * M1 * M2 * …
```

If the assumption about the musculature took into account the antagonists mentioned above there would be one prototype for each antagonist muscle (*Listing 156*).

Listing 156: An enhanced facial musculature model including the two antagonist muscle pairs

```
type Muscles = Muscle * M0 * M1 * M2 * (MLeft | MRight) * (MUp | MDown) * …
```

The source code also contains some examples of more complex muscle models types producing models with hundreds of alternatives. In contrast to the preceding model types, in these complex types the muscle fragment types occur in various places, as indicated in *Listing 157*.

Listing 157: A more complex facial musculature model consisting of several groups of antagonist muscles. Some muscle fragments appear at more places.

```
(M0 | … | M6) * (M0 | … | M7) … * (M1 | … | M6 | M7)
```

And it is exactly the case when using the type template in *Listing 158* becomes very useful. Otherwise, every actual muscle type, which may consist of more fragments and wrappers, would have to be copied to each its occurrence in the type expression.

Listing 158: Abstract type (a template) of the facial muscles model

```
type MusclesType[Base, M0, M1, M2, …] {
    type Muscles = Base * M0 * M1 * M2 * …
}
```

Just as in the case of the emotions tier, here a pair of a muscle fragment and its wrapper substitutes the respective formal muscle type arguments.

The `MuscleBase` fragment substitutes the `Base` formal type argument and serves as the common base for all muscle fragments and wrappers (*Listing 159*).

Listing 159: `MuscleBase` fragment and its configuration trait

```
trait MuscleBaseInit {
  val tensions: Map[String, Float]
}

case class MuscleBaseInitData(tensions: Map[String, Float])
    extends MuscleBaseInit

@fragment
trait MuscleBase extends dlg[MuscleBaseInit] {
  def contractions: Map[String, (Float, Float)] = Map.empty
}
```

The base fragment is also designed as an immutable fragment initialized during its creation from the configuration object. This configuration object is an instance of `MuscleBaseInit` trait and encapsulates tensions of muscles. These tensions are calculated from the levels of

emotions in the emotions tiers before each re-morphing of the muscles morph. Each muscle fragment present in the selected morph alternative initializes itself lazily from the tensions map.

The `contractions` method returns a map of muscle contractions. Each muscle fragment overrides this method and contributes to the resulting map. This method is analogous to the `actualEmoLevels` method in `EmotionBase` in the sense that only the active muscles contribute to the map even if the initial tensions map may contain tensions of other fragments, which could not be included to the actual morph model.

Note: The contraction is expressed as a difference of two points on the muscle's spline curve for the curve parameter `t=currentTension` and `t=0`.

The code in *Listing 160* shows the muscle fragment and its wrapper. The `m0` field holds a data object encapsulating the current muscle tension and the spline representing the muscle on the face. Other muscle wrappers may use quadratic or cubic Bezier curves instead. The contraction method of the `Muscle` class (*Listing 161*) calculates the contraction as described in the previous note.

Listing 160: The muscle fragment and its wrapper

```
@fragment
trait M0Fragment {
  this: MuscleBase =>

  lazy val m0 = new Muscle(m0Spline, this)
}


@fragment @wrapper
trait M0Wrapper extends MuscleBase {
  this: M0Fragment =>

  override def contractions: Map[String, (Float, Float)] =
    super.contractions + ("m0" -> m0.contraction())


}
```

The `M0Fragment` depends on `MuscleBase`, as it needs to lazily initialize the data member `m0`. The `MuscleBase` instance initializes itself through the `this` reference passed on its constructor. The `M0Wrapper` depends on `M0Fragment` in order to retrieve the contraction, which is added to the map returned by the overriding `contractions` method.

Listing 161: Auxiliary class `Muscle` *holding the muscle spline and calculating muscle contractions*

```
class Muscle(val spline: Spline, muscleBase: MuscleBase) {
  val name = spline.name
```

```
    val tension: Float = muscleBase.tensions.getOrElse(spline.name, 0f)

    def contraction(): (Float, Float) = {
      val pos0 = spline(0)
      val pos1 = spline(tension)
      (pos1._1 - pos0._1, pos1._2 - pos0._2)
    }
  }
```

Muscles Context

Just as in the case of the emotions tier, the muscle tier is composed of the building blocks presented in the previous section. Also the building process will essentially follow the same steps up to some important differences, on which this section focuses. The MusclesContext class (*Listing 163*) and its companion object (*Listing 162*) represent the muscles tier. The companion object contains the actual model type obtained by substituting the actual muscle fragments and their wrappers for the formal type arguments of the MusclesType structural type.

Listing 162: MuscleContext object

```
object MusclesContext {
  // Actual morph type
  type MusclesTypeActual = Muscles.MusclesType[MuscleBase * $[EmotionBase],
    M0Fragment * M0Wrapper, M1Fragment * M1Wrapper,…]
}
```

The joined type MuscleBase * $[EmotionBase] applied to the first type argument is the common base (LUB) of the muscles morph model type. The placeholder operator $ (see p. 180) indicates that the EmotionBase fragment is not part of the muscles morph model; instead, it is supposed to be delivered through a morph reference. Fragment EmotionBase serves as the link between the emotions and muscles tiers and is used by the muscles morphing strategy to determine, which emotions are active in the emotions tier and what are their levels. The above-mentioned reference to the emotions tier is specified in the constructor argument of the MusclesContext class. This reference is the only coupling between the two tiers.

Listing 163: MuscleContext class

```
class MusclesContext(emoRef: &[EmotionBase]) extends TensionCalculator {
  …
}
```

MusclesContext extends the TensionCalculator trait that contains helper methods for calculating tensions from emotion levels. This trait is not presented here in detail due to its rather auxiliary role.

The context data referring to the phenomenon to be represented is the tensions map, which contains the initial tensions for the selected set of muscles. Just as in the case of the emotions, not all muscle tensions specified in the map must be applied to the respective muscle fragments, as the morph model may prohibit certain combinations of simultaneously activated muscles, such as the antagonist muscles turning the head left and right.

```
private var tensionsMap: Map[String, Float] = Map.empty
```

The goal of the musclesRef reference is to join the emotions model with the muscles model. The MusclesTypeActual#Muscles type is enclosed in the placeholder operator, since all its fragments and wrappers, except the linking fragment EmotionBase, are placeholders.

```
val musclesRef: &[$[MusclesTypeActual#Muscles]] = *(emoRef)
```

The muclesRef reference could be used instead of the emoRef reference in the constructor argument; however, it would have a negative impact on the compiler performance. In such a case the compiler extension would have to check N*M combinations of alternatives, where N is the number of the alternatives in the emotions model and M is the number of the muscles model's alternatives. But, if the additional emoRef reference is inserted between the two models then the compiler extension will check N*1 alternatives in the first assignment and 1*M in the other; i.e. it yields N+M combinations in total, which grows much slowly with growing N and M than N*M.

The unveil macro is used to determine the morph model and the default morphing strategy of the joined morph model associated with the musclesRef.

```
val (model, defaultStrategy) = unveil(musclesRef)
```

The morph model and the default strategy are necessary to construct the morphing strategy used by the muscles tier's recognizer. The default strategy is in fact a bridge strategy connecting the two morph models and translating the alternatives chosen from the source model to those from the target model (see org.morpheus.BridgeStrategy).

The morphing strategy chain begins with the default strategy followed by a series of fragment rating strategies (*Listing 164*). For each muscle fragment there is one fragment rating strategy in the series. The fragment rating strategy allows rating the alternatives containing the fragment associated with the strategy. The rating value is determined by the function passed as the second argument to the rateFragment macro, which creates this kind of strategy. It should be noted that the rating is additive; i.e. if one alternative is rated by two strategies then its overall rating is the sum of the two ratings. The fragment to be rated is

specified as the type argument of the `rateFragment` macro. The first (normal) argument is used to chain the strategy with the previous one.

Listing 164: The morphing strategy of the facial musculature model

```scala
val strategy: MorphingStrategy[model.Model] = {
  // Chaining partial strategies
  var strat: MorphingStrategy[model.Model] = defaultStrategy

  strat = rateFragment[M0Fragment](model)(strat, rateMuscle("m0"))
  strat = rateFragment[M1Fragment](model)(strat, rateMuscle ("m1"))
  …
  // The last link in the strategy chain is a custom strategy (see below)
  new MusclesStrategy(strat)
}
```

The `rateMuscle` method returns the rating function for the muscle fragment, whose name is stored in the `muscleName` argument (*Listing 165*).

Listing 165: The `rateMuscle` method rates the muscle fragment identified by `muscleName`

```scala
def rateMuscle(muscleName: String): (Option[model.ImmutableLUB]) => Double =
  (_) => if (tensionsMap.contains(muscleName)) {
        1d
      } else {
        0d
      }
```

The rating function is invoked whenever the alternative being rated contains this muscle fragment. The rating function cooperates with the `tensionsMap` map containing the muscle tensions derived from the emotion levels in the emotions tier. Prior to the rating, the `tensionsMap` is updated by the topmost strategy in the chain - `MusclesStrategy` (see below) so as to reflect the current emotion levels.

The rating function rates the alternative (containing the associated muscle fragment) by `1` as long as the `tensionsMap` contains the muscle fragment. Otherwise it does not rate the alternative, i.e. it returns `0`.

The rating function does not use its argument (`_`), which contains a reference to the muscles morph being re-morphed (it is `None` if the first morph is under construction).

The strategy chain's last link, which is actually the top one, is the custom strategy `MuscleStrategy` (*Listing 166*). It extends the `model.Strategy` abstract class, which is offered by the morph model for an easier implementation of strategies for this model.

Listing 166: The top link in the chain of the muscle model's morphing strategies

```scala
class MusclesStrategy(prevStrat: MorphingStrategy[model.Model])
extends model.Strategy {
  override def chooseAlts(instance: model.Recognizer)
   (morphProxy: Option[instance.ImmutableLUB]): Alternatives[instance.Model] = {

    // update the tensions map in the context
    MusclesContext.this.tensionsMap = {
      morphProxy match {
        case None => Map.empty
        case Some(em) => createTensionsMap(em.actualEmoLevels)
      }
    }
    prevStrat.chooseAlternatives(instance)(morphProxy)
  }
}
```

The main goal of the `chooseAlts` method is to update the `tensionsMap` member prior to the rating the alternatives. This method uses the `morphProxy` argument to obtain the current emotions levels from the emotions tier. The emotions tier is accessible thanks to the `EmotionBase` fragment in the LUB of the muscle morph model. `EmotionBase` has the `actualEmoLevels` that returns the active emotions and their levels. The `createTensionsMap` method, which is defined in `TensionCalculator`, determines the muscles and calculates their tensions from the input emotion levels.

After the `tensionsMap` variable is updated, the strategy delegates the processing to the preceding rating strategies.

Having the strategy initialized, it is possible to create the recognizer of the muscles tier by dereferencing the `musclesRef` reference (*Listing 167*). Besides the reference, the strategy and the fragment factory for `MuscleBase` are passed to the * macro. The fragment factory overrides the default one that does not support configuration traits. The immutable `MuscleBase` fragment is initialized through the instance of `MuscleBaseInitData` carrying a reference to the fresh version of the `tensionsMap` map.

Listing 167: Creating the facial muscles recognizer

```scala
val recognizer = *(musclesRef, strategy, frag[MuscleBase,
    MuscleBaseInit](MuscleBaseInitData(tensionsMap)))
```

Usage

The illustration of the usage of the muscles tier is based on the illustration of the emotions tier (*Listing 168*). First of all the two tiers are created by instantiating `EmotionsContext` and `MusclesContext`. The muscles tier is linked to the emotions tier by passing the recognizer

from the emotions tier as the argument of `MusclesContext`'s constructor. The recognizer from the muscles tier is stored to the local variable `musMorph`.

Listing 168: Executing the emotions and muscles models

```scala
val emoCtx = new EmotionsContext
val musCtx = new MusclesContext(emoCtx.recognizer)
val musMorph = musCtx.recognizer.~

def printInfo(): Unit = {
  println("Morph Alt: " + musMorph.myAlternative)
  println("Actual Emo Levels: " + musMorph.actualEmoLevels)
  println("Actual Contractions: " + musMorph.contractions)
}

println("Beginning")
printInfo()

emoCtx.emo1Lev = Some((Emotions.joy, 0.7f))
emoCtx.emo2Lev = Some((Emotions.surprise, 0.2f))

musMorph.remorph

println("\n1st wave")
printInfo()

musMorph.remorph

println("\n2nd wave:")
printInfo()
```

The `printInfo` local method is used to print reports at the individual stages of the illustration. After adjusting the levels of the selected two emotions the muscles morph is remorphed in order to reflect these new emotions.

Interestingly, according to the "1st wave" outputs (*Listing 169*), only the emotions tier reflected the changes in the context. The muscles tier did not react. The reason why is that the muscles strategy obtained the tensions map from the old morph, as the new morph was in the process of its creation. In order to propagate the new tensions to the muscles tier it is necessary to perform an additional re-morphing. As seen in the "2nd wave" report, after the second re-morphing the structure of the morph and the muscle contractions reflect the current emotions.

Listing 169: The output produced by the code in Listing 168

```
Beginning
Morph Alt: List(MuscleBase, EmotionBase)
```

```
Actual Emo Levels: List()
Actual Contractions: Map()

1st wave
Morph Alt: List(MuscleBase, EmotionBase, SurpriseFragment, JoyFragment,
SurpriseWrapper, JoyWrapper)
Actual Emo Levels: List((1,0.7), (2,0.2))
Actual Contractions: Map()

2nd wave:
Morph Alt: List(MuscleBase, M7Fragment, M2Fragment, H_Mey8Fragment,
H_Mli9Fragment, M1Fragment, M12Fragment, H_Mey12Fragment, H_TeethUpperFragment,
M0Fragment, M15Fragment, M11Fragment, H_Meb7Fragment, M6Fragment, M042Fragment,
M3Fragment, H_TeethLowerFragment, H_Meb6Fragment, M041Fragment, M5Fragment,
M8Fragment, H_Mli10Fragment, M9Fragment, M10Fragment, M16Fragment, EmotionBase,
SurpriseFragment, JoyFragment, SurpriseWrapper, JoyWrapper, M7Wrapper,
M2Wrapper, H_Mey8Wrapper, H_Mli9Wrapper, M1Wrapper, M12Wrapper, H_Mey12Wrapper,
H_TeethUpperWrapper, M0Wrapper, M15Wrapper, M11Wrapper, H_Meb7Wrapper,
M6Wrapper, M042Wrapper, M3Wrapper, H_TeethLowerWrapper, H_Meb6Wrapper,
M041Wrapper, M5Wrapper, M8Wrapper, H_Mli10Wrapper, M9Wrapper, M10Wrapper,
M16Wrapper)
Actual Emo Levels: List((1,0.7), (2,0.2))
Actual Contractions: Map(m7 -> (0.0,0.0), m2 -> (0.36825562,-6.628601), m8 ->
(0.0,0.0), h_mey8 -> (4.230072,5.9449615), h_mli9 -> (0.0,3.0257263), m1 ->
(0.0,0.0), m12 -> (0.0,0.0), h_mey12 -> (0.0,0.0), h_mli10 ->
(6.648136,8.420959), m9 -> (51.967438,-28.066406), h_teethupper -> (0.0,0.0), m0
-> (0.0,0.0), m15 -> (0.0,17.324036), m11 -> (0.24630737,1.6921082), h_meb7 ->
(0.0,0.0), m6 -> (0.0,-10.327087), m10 -> (0.0,0.0), m16 -> (-16.71281,0.0),
m042 -> (0.0,8.0928955), m3 -> (0.0,0.0), h_teethlower -> (0.0,0.0), h_meb6 ->
(0.0,11.334427), m041 -> (0.0,-2.594513), m5 -> (0.0,-7.863266))
```

It follows from the above that a stacked multi-tier morph, in which the morphing strategy of an upper tier uses information from a lower tier, requires multiple re-morphing, the so-called *morphing waves*, in order to propagate the changes in the lowest tier throughout the whole morph.

8.1.3 Facial Features Model

The facial features morph model is not the most complex one in terms of the number of fragments and alternatives, however, in contrast to the other models in this application, it contains some fragments that depend on others. The behavior of those fragments needs to collaborate with the others to construct properly the splines of the facial features. For example, the fact that the upper and lower lips share the boundary points of their splines is expressed by introducing a dependency of the lower lip on the upper lip (it could also be done vice-versa).

The facial morph model consists of seven features implemented, as usual, as pairs of feature fragments and their wrappers (*Figure 78*).

Figure 78: A simplified schema of facial features (Spindler & Fadrus, n.d.)

The facial features morph model expressed by the expression in *Listing 170* results in 30 alternatives.

Listing 170: The morph model of the facial features

```
Base * \?[Eyebrow] * \?[UpperLid * \?[LowerLid]] * \?[UpperLipJoiner *
\?[UpperLip * \?[LowerLip * \?[LowerLipJoiner]]]]
```

It may be decoded from the expression that the `LowerLipJoiner` depends on `LowerLip`, `LowerLip` on `UpperLip`, `UpperLip` on `UpperLipJoiner`, considering that the dependency of fragment `B` on `A` can be expressed as:

```
A * \?[B]
```

The expression of the morph model is simplified by using type `\?[F]` is defined as:

```
type \?[F] = Unit | F
```

This auxiliary type makes it easier to specify optional fragments or sub-models. The first version makes the fragment `F` by default off, while the second version `/[F]` makes `F` by default on.

```
type \?[F] = F | Unit
```

Similarly to the previous tiers, also here the fragment types in the morph model expressions are formal type arguments of the enclosing structural type `FeaturesType` (*Listing 171*).

Listing 171: The type template for various morph models of facial features

```
type FeaturesType[Base, Eyebrow, UpperLid, LowerLid, UpperLipJoiner, UpperLip,
LowerLip, LowerLipJoiner] = {
```

```
    type Features = Base * \?[Eyebrow] * \?[UpperLid * \?[LowerLid]] *
  \?[UpperLipJoiner * \?[UpperLip * \?[LowerLip * \?[LowerLipJoiner]]]]
  }
```

The actual base fragment of the morph model is `FeatureBase`, which, just as the other base fragments in the other tiers, is designed as an immutable fragment initialized from the `FeatureBaseInit` configuration trait (*Listing 172*).

Listing 172: `FeatureBase` fragment and its configuration trait

```
  trait FeatureBaseInit {
    val warps: Map[String, FeatureWarp]
  }

  case class FeatureBaseInitData(warps: Map[String, FeatureWarp]) extends
  FeatureBaseInit

  @fragment
  trait FeatureBase extends dlg[FeatureBaseInit] {
    def collectSplines: List[Spline] = Nil
  }
```

The state of the base fragment consists of the map of the so-called *feature warps*. A feature warp is a data record containing shifts of the feature spline's definition points.

Listing 173: The auxiliary `FeatureWarp` class

```
  case class FeatureWarp(deltas: Map[Int, (Float, Float)]) {
    def update(newDeltas: Map[Int, (Float, Float)]): FeatureWarp = {
    … // creates an updated copy of this warp
    }
  }
```

Every feature fragment contains its spline transformed (warped) according to the warp record found in the warps map held by the `FeatureBase` fragment (*Listing 174*). This is also why every feature fragment depends on this base fragment.

Listing 174: `LowerLidFragment` fragment and its spline

```
  @fragment
  trait LowerLidFragment {
    this: FeatureBase =>

    lazy val lowerlidSpline = CBezier("lowerlid", (43f, 212f), (56f, 212f), (80f,
  235f), (125f, 210f)).transform(warps("lowerlid").deltas)


  }
```

The `collectSplines` method in `FeatureBase` returns the splines of all activated features in the morph. All feature fragment wrappers override this method and contribute their splines to the resulting list.

Listing 175 shows the `LowerLidWrapper` wrapper, which before adding the lower lid spline to the result list in `collectSpline` reads the upper lid spline from `UpperLidFragment` and uses its boundary points `p1` and `p2` to replace the boundary points of its own spline.

Listing 175: `LowerLidWrapper` wrapper and its `collectSplines` method

```scala
@fragment @wrapper
trait LowerLidWrapper extends FeatureBase {
  this: LowerLidFragment with UpperLidFragment =>

  override def collectSplines: List[Spline] = {
    val uls = upperlidSpline
    lowerlidSpline.copy(p1 = uls.p4, p4 = uls.p1) :: super.collectSplines
  }
}
```

Facial Features Context

The facial features context is built analogously to the muscles context. The `FeaturesContext` companion object (*Listing 176*) contains the actual morph model type expression with two "foreign" fragments `MuscleBase` and `EmotionBase`, which link the facial features tier with the muscles and emotions tier.

Listing 176: `FeatureContext` object

```scala
object FeaturesContext {
  // Actual type
  type FeaturesTypeActual = Features.FeaturesType[
    FeatureBase * $[MuscleBase with EmotionBase],
    EyebrowFragment * EyebrowWrapper,
    UpperLidFragment * UpperLidWrapper,
    LowerLidFragment * LowerLidWrapper,
    UpperLipJoinerFragment * UpperLipJoinerWrapper,
    UpperLipFragment * UpperLipWrapper,
    LowerLipFragment * LowerLipWrapper,
    LowerLipJoinerFragment * LowerLipJoinerWrapper]
}
```

The constructor of the `FeatureContext` class (*Listing 177*) has one argument for passing a reference to the two underlying tiers.

Listing 177: `FeatureContext` class

```
class FeaturesContext(musRef: &[MuscleBase with EmotionBase]) extends
TensionCalculator {
…
}
```

The input object of the facial features tier is the `warpsMap` map that is updated always before invoking the morphing strategy chain by `FeaturesStrategy`, which is the top of the chain.

```
private var warpsMap: Map[String, FeatureWarp] = Map.empty
```

The morph model and the default strategy can be obtained from the reference joining the facial features and muscles tiers by means of the `unveil` macro.

```
val featuresRef: &[$[FeaturesTypeActual#Features]] = *(musRef)
val (model, defaultStrategy) = unveil(featuresRef)
```

Similarly to the muscles strategy, also the facial features strategy is a chain beginning with the default strategy followed by a series of fragment rating strategies and finished by the custom `FeatureStrategy` (*Listing 178*).

Listing 178: The chain of partial morphing strategies of the facial features model

```
val strategy: MorphingStrategy[model.Model] = {
  var strat: MorphingStrategy[model.Model] = defaultStrategy

  strat = rateFragment[EyebrowFragment](model)(strat, rateFeature("eyebrow"))
  strat = rateFragment[UpperLidFragment](model)(strat, rateFeature("upperlid"))
  strat = rateFragment[LowerLidFragment](model)(strat, rateFeature("lowerlid"))
  strat = rateFragment[UpperLipJoinerFragment](model)(strat,
      rateFeature ("upperlipjoiner"))
  strat = rateFragment[UpperLipFragment](model)(strat, rateFeature("upperlip"))
  strat = rateFragment[LowerLipFragment](model)(strat, rateFeature("lowerlip"))
  strat = rateFragment[LowerLipJoinerFragment](model)(strat,
      rateFeature("lowerlipjoiner"))

  new FeaturesStrategy(strat)
}
```

The rating function (*Listing 179*) rates the alternatives containing the respective feature fragment by 1 if the `warpsMap` contains a warp for the facial feature; otherwise it does not rate.

Listing 179: The `rateFeature` method selecting the muscle fragment identified by featureName

```
  def rateFeature(featureName: String): (Option[model.ImmutableLUB]) => Double =
    (_) => if (warpsMap.contains(featureName)) {
        1d
    } else {
        0d
    }
```

The custom `FeaturesStrategy` (*Listing 180*) uses the same tactics as the `MusclesStrategy`. Prior to the rating the alternatives, it updates the `warpsMap` context field. To calculate the fresh warps the strategy calls the helper `createWarpMap` from `TensionsCalculator`.

Listing 180: The top link in the chain of the facial features model's morphing strategies

```
  class FeaturesStrategy(prevStrat: MorphingStrategy[model.Model])
  extends model.Strategy {
    override def chooseAlts(instance: model.Recognizer)
    (morphProxy: Option[instance.ImmutableLUB]): Alternatives[instance.Model] = {
        warpsMap = {
          morphProxy match {
            case None => Map.empty
            case Some(mm) => createWarpMap(mm.contractions)
          }
        }
        prevStrat.chooseAlternatives(instance)(morphProxy)
    }
  }
```

Then the recognizer is created by dereferencing the `featuresRef` reference and using the strategy and the `FeatureBase` factory, which allows passing the configuration object carrying the `warpsMap` (*Listing 181*).

Listing 181: Creating the facial features recognizer

```
  val recognizer = *(featuresRef, strategy,
    frag[FeatureBase, FeatureBaseInit](FeatureBaseInitData(warpsMap)))
```

Usage

The usage of the facial features tier extends the usage of the muscles tier. Here, the features context is created and the muscles tier recognizer is passed on constructor of `FeaturesContext` (*Listing 182*). The list of activated face feature splines is added to the report of the morph. Since now the morph consists of three tiers it is necessary to carry out three consecutive morphing waves to propagate the changes in the bottom emotions tier up to the facial features tier.

Listing 182: Executing the emotions, muscles and features models

```
val emoCtx = new EmotionsContext
val musCtx = new MusclesContext(emoCtx.recognizer)
val featCtx = new FeaturesContext(musCtx.recognizer)
val featMorph = featCtx.recognizer.~

def printInfo(): Unit = {
  println("Morph Alt: " + featMorph.myAlternative)
  println("Actual Emo Levels: " + featMorph.actualEmoLevels)
  println("Actual Contractions: " + featMorph.contractions)
  println("Face Splines: " + featMorph.collectSplines .map(_.name))
}

println("Beginning")
printInfo()

emoCtx.emo1Lev = Some((joy, 0.7f))
emoCtx.emo2Lev = Some((surprise, 0.2f))

featMorph.remorph

println("\n1st wave")
printInfo()

featMorph.remorph

println("\n2nd wave:")
printInfo()

featMorph.remorph

println("\n3rd wave:")
printInfo()
```

8.1.4   Assembling

Now, when all tiers are implemented, it is possible to put the pieces together and run the application. *Listing 183* sketches the basic steps to assemble the application.

Listing 183: Assembling the application from the three tiers

```
// Link the tiers
val emoCtx = new EmotionsContext
val musCtx = new MusclesContext(emoCtx.recognizer)
val featCtx = new FeaturesContext(musCtx.recognizer)

// Adjust the levels of the selected emotions in the emotions tier
```

```scala
    emoCtx.emo1Lev = Some((joy, 0.8f))
    emoCtx.emo2Lev = Some((surprise, 0.3f))

    // 1st morphing wave
    featCtx.morph.remorph
    // 2nd morphing wave
    featCtx.morph.remorph
    // 3rd morphing wave
    featCtx.morph.remorph

    // Render the face as SVG
    val splines = featCtx.morph.collectSplines
    val outFile = new File(s"face.svg")
    Features.flushSvg(splines, outFile)
```

8.2   Summary

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.

Although rather simplistic, since the emotion levels are set explicitly, the application could be easily extended to simulate emotional reactions and the relevant facial expressions of a human reading a text, which would be the input of the application. The application might also support different human character types reacting to the text in various ways. The description of each of these different personality types would be stored in its own XML file containing the values for binding the properties from neighboring tiers.

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. While the tiers in the traditional architecture are static or rather inflexible at best, the tiers in the protean architecture are fundamentally adaptive. Other important difference concerns the tier deployment. In contrast to the traditional tiers, which are essentially detachable so that they may be deployed on physically separate machines, the protean tiers are embedded one into another constituting so a single complex and compact object. Such a configuration allows that a method in a protean persistence tier

may invoke a method (typically an overriding one) from a higher tier, such as the protean presentation tier; e.g. a method receiving notifications of the state modifications. A similar invocation flow would be quite problematic in the traditional three-tier architecture, as it requires that a method in one tier may invoke only the methods in the adjacent lower tier; i.e. the flow is unidirectional. While such a requirement is well justifiable with respect to the potential bottleneck caused by the limited network capacity, it may have a negative impact on the quality of the user interface, which might tend to display outdated information, for instance.

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.

It has been shown that in order to propagate changes in a morph consisting of stacked tiers with immutable fragments, a series of consecutive re-morphing rounds, called morphing waves, must be carried out. The number of these waves equals to the number of the tiers in the morph, as long as the change affects all tiers.

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. This problem may be mitigated by curbing the number of source and target alternatives, for example as shown in the case of introducing an auxiliary morph reference between the source and target models, which reduced the number of necessary checks from `N.M` to `N+M`. At run-time, the slowest operation is re-morphing.

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.

9 Conclusion

Research in the field of cognitive psychology shows that the prototypical view of concepts is more suited than the Aristotelian view to describe a majority of everyday concepts. However, although the Aristotelian approach has been abandoned in many fields such as biology, psychology etc., it remains the foundation of OO programming. Neither prototype-based languages can promise a viable solution, since they remain on the level of individual instances and lack any notion of concepts.

The main goal of this thesis is to fill the above-described gap by 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). The basic tenet of OM is that the concepts of the phenomena consisting of protean objects may be built through the construction of morph models describing the possible forms of the objects. The notion of a morphing strategy forms a bridge between the concept and its extension; in other words, the strategy's task is to select the most suitable morph alternative from the model to instantiate a representation - a morph - of a given protean object. The instantiation of morphs is the responsibility of a recognizer in cooperation with the morph model and the morphing strategy.

In order to illustrate the difficulties that developers face when using the current OO languages to model protean objects, two initial case studies were presented, in which the same task was implemented using three popular OO languages: Java, Groovy and Scala. The case studies revealed several critical issues in each language. In Java, it was the inability to properly model the "is-a" relationship caused by the absence of the notion of traits. The other languages fix this problem, however, the modeling of non-uniform data led either to combinatorial explosion of code (Scala) or to unmanageable code (Groovy). Each case study further included a fictional code introducing relevant OM features to illustrate possible solutions to the diagnosed problems.

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 this 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. Although designed as a compiler extension of Scala, the morph instances may be used in any JVM-based language.

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. The emotion, muscle and facial feature tiers were designed as autonomous protean objects and the neighboring tiers were loosely coupled by means of a special morphing strategy selecting the most suitable form for one tier according to the current form of the neighboring tier. The behavior of such a system bears a striking resemblance to that of living organisms in how it responds to external stimuli. A change in the lowest tier (emotions) does not influence the other tiers directly; instead, the change propagates in waves through the whole system (organism).

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

In view of these convincing achievements it may be concluded that OM represents a promising new direction in the field of OO programming, which may bring real-world phenomena and their models in OO applications closer together, especially as long as applications are to model complex, organic phenomena that defy the traditional Aristotelian class-base approach.

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.

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.

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

10 Appendix 1: Using OM to implement an OntoUML model

The goal of this appendix is to demonstrate how to use an existing OntoUML model and transform it to a Morpheus application. In the course of this application, the model used in section 2.4 to explain the basics of OntoUML and UFO-1 will be transcribed to a number of fragments, morph models and other Morpheus artifacts. The model, which is again shown in Figure 79, captures the person and organization entities assuming a couple of forms and having two relationships between them.
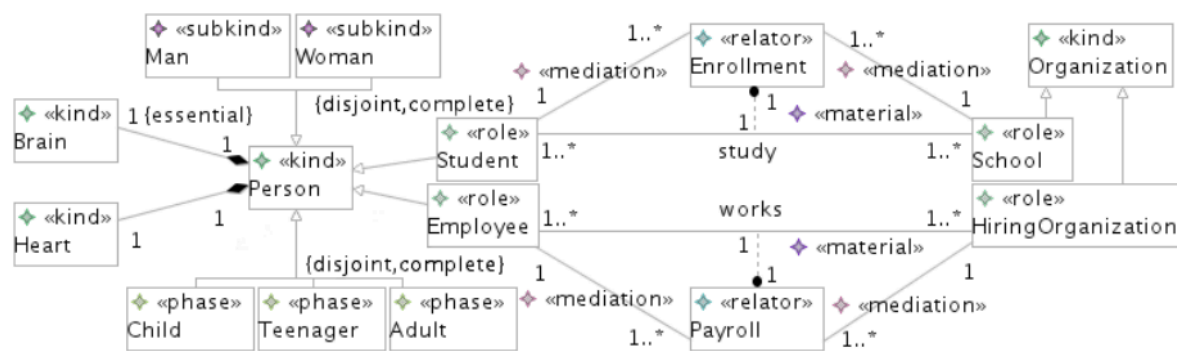
Figure 79: The school enrollment model (Benevides et al., 2009)

The resulting application implements creating instances of organizations and persons, morphing the instances into roles and phases, and establishing and abolishing relationships.

The implementation of the model begins with expressing the universals as fragments. The `Brain` and `Heart` kinds can be directly transcribed to Morpheus as shown in Listing 184.

Note: A kind is a type carrying and supplying an identity (the so-called *sortal* universal).

Listing 184: `Brain` and `Heart` kinds expressed as fragments

```
@fragment
trait Brain {

}

@fragment
trait Heart {

}
```

For the sake of simplicity, the model ignores the structure of the two kinds, which is why they contain no properties or members.

The `Person` kind is a bit more complex, since it consists of a set of attributes and of the relationships to the two preceding kinds. It is usually preferable to specify values of some attributes during the creation of an object. Here, these attributes are gathered to the `PersonData` initialization trait, from which the `Person` fragment trait extends. The `dlg[T]` type used to wrap `PersonalData` commands Morpheus to generate a constructor with one argument for passing an instance of `PersonalData`.

Listing 185: Person kind with its initialization trait `PersonalData`

```
trait PersonalData {
  val personId: Long
  var firstName: String
  var lastName: String
  val birthDate: Date
}


@fragment
trait Person extends dlg[PersonalData] {
  // essential
  val brain = singleton[Brain].!

  var heart = singleton[Heart].! // replaceable

  def age = {
    val diffInMillis = new Date().getTime - birthDate.getTime
    val a = TimeUnit.DAYS.convert(diffInMillis, TimeUnit.MILLISECONDS) / 365
    a
  }
}

case class PersonalDataParams(val personId: Long, fn: String, ln: String, val
birthDate: Date) extends PersonalData {
  override var firstName: String = fn
  override var lastName: String = ln
}
```

The brain attribute of `Person` represents the relationship with the `Brain` kind. The "essential" character of this attribute in the OntoUML sense is expressed by the `val` keyword making the attribute immutable. On the other hand, the heart attribute, which represent the relationship with the `Heart` kind, is in principle replaceable and thus it is defined as `a var,` i.e. a mutable, attribute. The age attribute is a calculated attribute calculating the age of the person. Instances of the `PersonalDataParams` class are used to pass the initial values as the argument to the generated constructor of the `Person` fragment trait.

The `Man` and `Woman` are two sub-kinds of the `Person` kind. In contrast to kinds, sub-kinds do not supply an identity; they inherit it from the parent kind.

Listing 186: `Man` and `Woman` sub-kinds

```
@fragment
trait Man {
  this: Person =>
}


@fragment
trait Woman {
  this: Person =>
}
```

The dependency on the provider of the identity is expressed by means of the self-type nota-tion (`this: Person =>`) (Listing 186).

The `Child`, `Teenager` and `Adult` universals represent three life phases of a person. Phases and roles are anti-rigid sortals; in other words, objects may become instances and cease to be instances of these universals freely during their life. Phases differ from roles in the fact that phases are organized in groups, from which no two phases are assumed at the same time by an object (so-called disjoint groups). This difference will be demonstrated when constructing morph models later in this chapter.

Listing 187: The age phases of a person

```
@fragment
trait Child {
  this: Person =>
}

@fragment
trait Teenager {
  this: Person =>
}

@fragment
trait Adult {
  this: Person =>
}
```

The age phases also bear the identity inherited from the person (they are sortals) and thus the corresponding fragment traits include the self-type notation referencing the `Person` trait, from which the identity is obtained (Listing 187).

The `Student` and `Employee` universals are roles; i.e. anti-rigid sortals that may occur in the same object independently, in contrast to phases. Nevertheless, at this stage, they are ex-pressed in the same way as phases (Listing 188).

Listing 188: Student and Employee roles

```
@fragment
trait Student {
  this: Person =>

  var credits: Int = _
}


@fragment
trait Employee {
  this: Person =>
}
```

The `Organization` universal is another kind in the model. It is expressed in Morpheus in the same was as the `Person` kind. The attributes, which are supposed to be specified during the initialization, are encapsulated in the `OrganizationalData` initialization trait, which is specified in the "extends" list of the `Organization` fragment trait.

Listing 189: Organization kind

```
trait OrganizationData {
  val organizationId: Long
  var name: String
}


@fragment
trait Organization extends dlg[OrganizationData] {

}


case class OrganizationDataParams(val organizationId: Long, nm: String) extends
OrganizationData {
  override var name: String = nm
}
```

There are two roles that can be assumed by an organization: `HiringOrganization` and `School`. The two roles can be expressed in Morpheus as shown in Listing 190.

Listing 190: HiringOrganization and School roles

```
@fragment
trait HiringOrganization {
  this: Organization =>
}


@fragment
trait School {
  this: Organization =>
```

```
    var schoolRating: Float = _
}
```

At this moment all universals from the OntoUML model have been transcribed to Morpheus and can be used to construct morph models.

First, the morph model of an organization is constructed (Listing 191). The morph model type is held in the `OrganizationModel` type variable, while the parsed model instance is stored in the `organizationModel` variable.

Listing 191: The morph model of an organization

```
type OrganizationModel = Organization *
  /?[School] *
  /?[HiringOrganization]
val organizationModel = parse[OrganizationModel](true)
```

The `Organization` fragment represents the core of the model, which is present in all model alternatives. Each alternative represents one possible form of the organization and the `Organization` fragment carries and provides the identity of the object. The `School` and `Hiring` organization fragments are specified as optional using the `/?[T]` type operator. The general pattern for specifying roles in Morpheus models is in Listing 192. (Note: The "*" operator is just an alias to the Scala `with` keyword.)

Listing 192: The pattern for specifying roles in Morpheus

```
Kind * /?[Role1] * /?[Role2]
```

The resulting list of morph model alternatives is as follows:

```
{Kind, Role1, Role2}
{Kind, Role1}
{Kind, Role2}
{Kind }
```

On the other, the phases, which are mutually exclusive universals, are expressed using the union operator "|", as shown in Listing 193.

Listing 193: The pattern for specifying phases in Morpheus

```
Kind * (Phase1 | Phase2)
```

The same pattern is used to specify complete sub-kinds (Listing 194).

Listing 194: The pattern for specifying complete sub-kinds in Morpheus

```
Kind * (SubKind1 | SubKind2)
```

The preceding pattern model yields the following alternatives:

```
{Kind, SubKind1}
{Kind, SubKind2}
```

If the sub-kinds do not constitute a complete group then the pattern model would contain an additional `Unit` fragment, as shown in Listing 197.

Listing 195: The pattern for specifying incomplete sub-kinds in Morpheus

```
Kind * (Unit | SubKind1 | SubKind2)
```

This model produces the following alternatives:

```
{Kind}
{Kind, SubKind1}
{Kind, SubKind2}
```

The Listing 196 shows a general pattern summarizing how to specify various types of universals in Morpheus.

Listing 196: A general pattern for specifying various universals in Morpheus

```
Kind * (SubKind1 | SubKind2) * (Phase1 | Phase2) * /?[Role1] * /?[Role2]
```

Having explained how various universals are used in morph models, the `Person` morph model may be constructed.

Listing 197: The morph model of a person

```
type PersonModel = Person *
  (Man | Woman) *
  (Child | Teenager | Adult) *
  /?[Student] *
  /?[Employee]

val personModel = parse[PersonModel](true)
```

The morph model of a person follows the general pattern from Listing 196.

There are two relationships between a person and an organization. Each relationship is represented by its relator universal. In Morpheus, the relators may be implemented using standard Scala classes having a constructor with two arguments for passing morph references to the two sides of the relationship. Listing 198 contains the relator class implementing the `Enrollment` relator universal.

Listing 198: Enrollment relator class

```
class Enrollment(studentRef: &[Student * Person], schoolRef: &[School *
Organization]) {
```

```
   val student: Student with Person = *(studentRef).~
   val school: School with Organization = *(schoolRef).~
}
```

The studentRef constructor argument declares that the passed morph must be an instance of the Student role. Similarly, the schoolRef declares that the passed morph must assume the School role. The morph references allow passing any morph, which is generally capable of assuming the type of the reference; in other words, the morph model of the morph must conform to the morph model of the reference. The actual morphing to the required form takes place during dereferencing. It is also the moment when the dereferencing may raise an error in case the morph cannot be re-morphed to the required form. Both morph references are dereferenced to the two corresponding student and school variables.

The Payroll relator is implemented in the same way as the Enrollment relator, as shown in Listing 199.

Listing 199: Payroll relator class

```
class Payroll(employeeRef: &[Employee * Person], hiringOrgRef:
&[HiringOrganization * Organization]) {
   val employee: Employee with Person = *(employeeRef).!
   val hiringOrg: HiringOrganization with Organization = *(hiringOrgRef).!
}
```

The existing enrollments and payrolls in the application are stored in the enrollments and payrolls variables. Both variables hold a map associating a person's id with the person's enrollments, resp. payrolls. These two maps may be seen as lightweight database tables.

```
var enrollments: Map[Long, List[Enrollment]] = Map.empty
var payrolls: Map[Long, List[Payroll]] = Map.empty
```

Now, the following passages will explain the instantiation of organization and person morphs from their morph models. First, the creation of an organization morph will be described. A new organization morph is created through the invocation of the createOrganization method shown in Listing 200. The method actually creates a recognizer instance, from which its default morph is used (through the ~ accessor).

Listing 200: The factory method instantiating an organization

```
def createOrganization(orgId: Long, name: String) = {

   implicit val orgFact =
     single[Organization, OrganizationData](OrganizationDataParams(orgId, name))

   singleton(organizationModel, rootStrategy(organizationModel))
}
```

The `createOrganization` has two arguments for passing the initial values of the organization's attributes `organizationId` and `organizationName`. Since the `Organization` fragment trait must be initialized through an instance of its initialization trait `OrganizationData`, the implicit variable holding the fragment factory must be declared. The fragment factory assigned to the `orgFact` variable will create an instance of the `Organization` fragment trait initialized by the data carried by the `OrganizationDataArgs` object.

The organization recognizer is created by means of the `singleton` macro. This recognizer will use the default morphing strategy when instantiating morphs.

The factory method for creating a person follows the similar way as the preceding method; however, it consists of more steps because of a more complex morphing strategy responsible for governing the forms of a person from multiple perspectives.

The factory method begins with building the chain of partial morphing strategies.

Listing 201: The factory method instantiating a person

```
def createPerson(personId: Long, firstName: String, lastName: String, birthDate:
String, woman: Boolean) = {

  var stg: MorphingStrategy[personModel.Model] = rootStrategy(personModel)
```

The first link in the morphing strategy chain is the default strategy of the morph model, which is created by means of the `rootStrategy` macro. This strategy returns the list of the alternatives in their natural order given by the morph model's expression.

```
  stg = maskAll()
```

The next step in the chain is the strategy that explicitly enables all fragments in the morph model as the initial step in each re-morphing. This strategy returns all alternatives in their natural order accompanied by a mask indicating that all fragments are enabled (ones in the mask bit set, where every bit corresponds to a certain fragment in the model). This mask is used to filter out the alternatives containing the disabled (unmasked) fragments (zeroes in the mask bit set).

The following statement creates the next strategy governing the genre role of the person by enabling one genre fragment and disabling the other. The strategy masks (enables) the `Woman` fragment and unmasks (disables) the `Man` fragment at the same time if the `woman` flag is true. Otherwise the fragments are handled vice versa.

```
  stg = mask[Woman | Man](stg, () => if (woman) Some(0) else Some(1))
```

This strategy uses a sub-model of the main model to determine the fragments to be masked, resp. unmasked. The lambda expression passed as the second argument identifies the alternative whose fragments will be masked (enabled) by its ordinal number in the sub-

model. The fragments from the other alternatives will be unmasked (disabled). Thus if the woman flag is true, the lambda always returns Some(0), which refers to the first alternative in the sub-model; i.e. the alternative with the single fragment Woman. Thus, the Woman fragment will always be enabled, while the Man fragment from the second alternative will always be disabled. Should the lambda return None instead of Some(x), the strategy would bypass the processing step in the chain.

The next partial strategy governs the age phases. In contrast to the previous strategy, which functions according to the outer parameter woman, which is constant for every instance of a person, the age-governing strategy works with respect to the current age of the person in question; i.e. it reflects the current inner state of the object.

```
stg = maskFull[Child | Teenager | Adult](personModel)(stg, {
  case Some(person) if person.age < 15 => Some(0)
  case Some(person) if person.age < 18 => Some(1)
  case Some(person) if person.age >= 18 => Some(2)
  case None => None
})
```

The maskFull macro also produces a strategy that enables the fragments from the alternative of the sub-model indicated by the number returned by the lambda, while the fragments from the other alternatives are disabled. However, this macro allows the lambda to examine the state of the object for which the new alternative is being chosen. The lambda can be specified as a partial function specifying a set of the case statements distinguishing between distinct states corresponding to the age phases. (Note: the None case should never happen; it is there to make the lambda complete).

Further, the following strategy governs the presence of the role in the resulting morph.

```
stg = unmaskFull[Student](personModel)(stg, {
  case Some(person) if !enrollments.contains(person.personId) => Some(0)
  case _ => None
})
```

The unmaskFull macro is complementary to maskFull macro, in terms of its process of enabling and disabling fragments. The value returned by the lambda indicates the alternative whose fragments will be explicitly disabled. And conversely, the fragments from the other alternatives will be enabled. The logic in this strategy may be read as follows: *Disable the* Student *fragment if there is no enrollment for the person in question. Otherwise, do nothing*.

An analogical logic is applied in the next strategy governing the presence of the Employee role in a Person morph.

```
    stg = unmaskFull[Employee](personModel)(stg, {
      case Some(person) if !payrolls.contains(person.personId) => Some(0)
      case _ => None
    })
```

At this stage, all partial strategies have been defined and the person recognizer can be created using the same mechanism as that used in the organization factory. An implicit fragment factory is declared that instantiates the `Person` fragment trait using the initial data encapsulated in the instance of the `PersonalDataParams` class.

```
    implicit val personFact =
      single[Person, PersonalData](PersonalDataParams(personId, firstName,
  lastName, dateFormat.parse(birthDate)))

    val rec = singleton(personModel, stg)

    rec.~.remorph // now the strategies will get the person reference
    rec
}
```

Before returning the recognizer its default morph is explicitly re-morphed (the second pass) so that all partial strategies obtain the reference to the morph. The reason is that in the first pass the strategies obtained `None` instead of `Some(person)`, since the person morph was just about to be created. In the second pass the strategies obtain the reference to the morph created previously. It follows that the resulting shape of the morph will reflect its state.

Once the factory methods for the two morphs are defined it possible to define the method for establishing the "study" relationship between a student and a school. The method in Listing 202 shows the enroll method, which has two arguments for passing the references to the student and school.

Listing 202: Establishing a new enrollment

```
  def enroll(person: personModel.Recognizer,
             schoolRef: &[School with Organization]) = {
```

The reference to person is passed as a reference to the person's recognizer. This reference is later passed to the constructor of the `Enrollment` class expecting a morph reference to `$[Student with Person]`. One always passes the morph's recognizer, instead of the morph itself, as long as the morph is supposed to be re-morphed when dereferencing the morph reference, so that the dereferenced morph reflects the current state. This is the case of the person, who might not be a student before establishing the new enrollment. On the other hand, one passes the morph, instead of its recognizer, if the morph is not supposed to

change its shape, which is the case of the school organization, since it was a school before establishing the new enrollment relationship.

The next step in the `enroll` method ensures that the `enrollments` map contains an entry for the person. It actually makes the person a student, since when the person reference is dereferenced and re-morphed, the partial strategy responsible for disabling the `Student` role will find this entry, and thus this strategy does not disable the `Student` role, even if there may not be any actual enrollments associated with the person.

```
enrollments.get(person.~.personId) match {
  case None =>
    enrollments += (person.~.personId -> Nil)
}

val enrollment = new Enrollment(person, schoolRef)

enrollments +=
  (person.~.personId -> (enrollment :: enrollments(person.~.personId)))
```

After creating the new enrollment entry, the `enrollments` map is updated. Before returning the enrollment it is necessary to explicitly re-morph the default morph of the person recognizer in order to reflect the changes. (Note: the dereferenced morph in the enrollment stored in the student field is already in the right shape, but as it is another morph instance than the default morph, it is necessary to re-morph the default morph independently.)

```
    person.~.remorph
    enrollment
}
```

The `employ` method establishes a new employment relationship between a person and an organization. Since it uses the same logic as the `enroll` method, it is not described here.

```
def employ(person: personModel.Recognizer, hiringOrgRef: &[HiringOrganization
with Organization])
```

The dismiss method in Listing 203 removes the enrollment of a student at the given school from the `enrollments` map.

Listing 203: Dismissing a student

```
def dismiss(person: personModel.Recognizer, from: Organization): Unit = {
  enrollments.get(person.~.personId) match {
    case None =>
    case Some(existingEnr) =>
      val newEnr = existingEnr.filter(_.school.organizationId !=
                                    from.organizationId)
      if (newEnr.isEmpty)
```

```
        enrollments -= person.~.personId
      else
        enrollments += (person.~.personId -> newEnr)
  }


  person.~.remorph
}
```

The default morph of the person's recognizer must be re-morphed in order to reflect the new situation.

The `fire` method does the same thing for the `employment` relationship and is not therefore described here.

```
def fire(person: personModel.Recognizer, from: Organization): Unit
```

The remaining passages of this chapter describe a simple program creating sample person and organization morphs. Then it establishes and abolishes relationships between them.

The sample morphs are created as shown in Listing 204.

Listing 204: Creating sample morphs of a person and an organization

```
val personRec = createPerson(123, "John", "Doe", "2000-06-02", woman = false)
val organizationRec = createOrganization(1000, "VŠE")
```

In order to visualize the structure of the sample morphs, one can use the `myAlternative` method returning the alternative used to create the given morph.

```
println(personRec.~.myAlternative)
println(organizationRec.~.myAlternative)d
```

The console output should look like this:

```
List(org.cloudio.morpheus.ontouml.Person,
org.cloudio.morpheus.ontouml.Man,
org.cloudio.morpheus.ontouml.Teenager)

List(org.cloudio.morpheus.ontouml.Organization,
org.cloudio.morpheus.ontouml.School,
org.cloudio.morpheus.ontouml.HiringOrganization)
```

The first enrollment is done in the following listing. The enrollment is followed by printing the student's data obtained from the `student` reference encapsulated by the enrollment object.

```
val enr: Enrollment = enroll(personRec, organizationRec)
```

```
println(s"${enr.student.firstName}:${enr.student.lastName}:
         ${enr.student.personId}:${enr.student.birthDate}:
         ${enr.student.credits}:${enr.school.schoolRating}")
```

The last piece of code prints the person's current morph alternative. Further, it employs the person, fires the person and dismisses it from the school. The accompanying output should reveal the changes in the structure of the person morph.

```
println(personRec.~.myAlternative)

employ(personRec, organizationRec)
println(personRec.~.myAlternative)

fire(personRec, organizationRec.~)
println(personRec.~.myAlternative)

dismiss(personRec, organizationRec.~)
println(personRec.~.myAlternative)
```

The source code of the application developed in this chapter is available at (Šlajchrt, 2016).

Bibliography

American Psychiatric Publishing. (2013) *Diagnostic and Statistical Manual of Mental Disorders*. 5th ed. American Psychiatric Publishing.

Andersen, H. (2000) Kuhn's Account Of Family Resemblance: A Solution To The Problem Of Wide-Open Texture. *Erkenntnis: An International Journal of Scientific Philosophy*, 52(3), pp.313-37.

Beckner, M. (1968) *The Biological Way of Thought*. Los Angeles, CA, USA: University of California Press.

Benevides, A.B., Guizzardi, G., Braga, B.F. & Almeida, J.P. (2009) Assessing Modal Aspects of OntoUML Conceptual Models in Alloy. In Carlos Alberto Heuser, G.P., ed. *Advances in Conceptual Modeling - Challenging Perspectives*. Gramado, Brasil, 2009.

Bird, A. & Tobin, E. (2015) Natural Kinds. In E.N. Zalta, ed. *The Stanford Encyclopedia of Philosophy*.

Booch, G. (1993) *Object-Oriented Analysis and Design with Applications*. 2nd ed. Redwood City, CA: Addisson-Wesley Professional.

Booch, G. (2005) *Unified Modeling Language User Guide*. Addison-Wesley.

Bourne, J.R. (1992) *Object-Oriented Engineering: Building Engineering Systems Using Smalltalk-80*. 1st ed. CRC Press.

Braga, B.F., Almeida, J.P., Guizzardi, G. & Benevides, A.B. (2010) Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. *Innovations Syst Softw Eng.*.

Browne, R.B. (1982) *Objects of Special Devotion: Fetishism in Popular Culture*. Popular Press.

Carvalho, V.A., Almeida, J.P., Fonseca, C.M. & Guizzardi, G. (2015) Extending the Foundations of Ontology-based Conceptual Modeling with a Multi-Level Theory. In *Conceptual Modeling, 34th International Conference*. Stockholm, 2015.

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

Coad, P., North, D. & Mayfield, M. (1997) *Object Models: Strategies, Patterns, and Applications*. Upper Saddle River: Yourdon Press.

Coplien, J. & Bjørnvig, G. (2010) *Lean Architecture: For Agile Software Development*. 1st ed. Wiley, West Sussex.

Coplien, J. & Reenskaug, T. (2012) The data, context and interaction paradigm. In

Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12. Tuscon, AZ, 2012.

CPAN. (n.d.) *C3 Algorithm* [Online]. Available at: http://search.cpan.org/~haarg/Algorithm-C3-0.10/lib/Algorithm/C3.pm .

Darwin, C., Ekman, P. & Prodger, P. (1998) *The Expression of the Emotions in Man and Animals*. 3rd ed. London, GB: Harper Collins.

Dawkins, R. (2005) *The Ancestor's Tale: A Pilgrimage to the Dawn of Evolution*. Mariner Books.

Deitel, H.M. & Deitel, P.J. (2005) *Java How To Program*. 6th ed. Upper Saddle River, New Jersey: Prentice Hall.

Duchenne, G.B. (1862) *Mécanisme de la physionomie humaine, ou analyse électro-physiologique de l'expression des passions*.

Factaz. (n.d.) *Is watermelon a fruit or a vegetable?* [Online]. Available from: http://www.factaz.com/post.php?id=16 .

Fowler, M. (1997) *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

Gamma, E. et al. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Garzás, J. & Piattini, M. (2007) Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices. Hershey PA, 2007. Idea Group Publishing.

Groovy. (2016) *Traits* [Online]. Available at: http://docs.groovy-lang.org/latest/html/documentation/core-traits.html .

Guizzardi, G. (2005) *Ontological Foundations For Structural Conceptual Models*. Centre for Telematics and Information Technology, University of Twente.

Hampton, J.A. (n.d.) *Concepts As Prototypes* [Online]. City University, London Available at:
http://www.staff.city.ac.uk/hampton/PDF%20files/Concepts%20as%20prototypes%202006.pdf .

Hampton, J.A. (n.d.) *Inheritance of attributes in natural concept conjunctions* [Online]. The City University, London, England Available at:
http://www.staff.city.ac.uk/hampton/PDF%20files/Hampton%20Inheritance%20M&C%201987.pdf .

Harrison, W., Budinsky, F. & Simmonds, I. (1995) Subject-oriented programming: Supporting decentralized development of objects.

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.

Hasso, & Carlson, C. (2014) Software composition using behavioral models of design patterns. *Journal of Software Engineering and Applications*, 7(2), pp.104-17.

Herrmann, S. (2010) Demystifying object schizophrenia. In *Proceedings of the 4th Workshop on Mechanisms for Specialization, Generalization and Inheritance (MASPEGHI '10)*. New York, USA, 2010. ACM.

Hjoerland, B. (n.d.) *Monothetic / polythetic classification* [Online]. Copenhagen University: Copenhagen University Available at: http://www.iva.dk/bh/lifeboat_ko/CONCEPTS/monothetic.htm .

IDC. (2015) *Smartphone OS Market Share* [Online]. Available at: http://www.idc.com/prodserv/smartphone-os-market-share.jsp .

Irwin, D.E., Bensch, S., Irwin, J.H. & Price, T.P. (2005) Speciation by Distance in a Ring Species. *Science*, 307(5708).

Java Performance. (n.d.) *Static code compilation in Groovy 2.0* [Online]. Available at: http://java-performance.info/static-code-compilation-groovy-2-0/ .

Krogh, B., Levy, S., Subrahmanian, E. & Dutoit, A. (1996) Strictly Class-Based Modeling Considered Harmful. In *System Sciences*. Wailea, HI, 1996. IEEE.

Liskov, B. (1988) Data abstraction and hierarchy. In *SIGPLAN Notices*., 1988. ACM.

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 .

Majorinc, K. (1998) Ellipse-Circle Dilemma and Inverse Inheritance. In *Proceedings of the 20th International Conference of Information Technology Interfaces*. Pula, 1998.

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

Martin, D. (2009) *Getters and Setters in Scala* [Online]. Available at: http://dustinmartin.net/getters-and-setters-in-scala/ .

McBeath, J. (n.d.) *Scala Class Linearization* [Online]. Available at: http://jim-mcbeath.blogspot.cz/2009/08/scala-class-linearization.html .

Medin, D.L., Altom, M.W. & Murphy, T.D. (1984) Given versus induced category representations: Use of prototype and exemplar information in classification. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, (10(3)), pp.333-52.

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 .

Misra, S. & Akman, K. (2008) *Weighted Class Complexity. A Measure of Complexity for Object Oriented System*. Journal Of Information Science And Engineering.

Needham, R. (n.d.) *Polythetic classification: Convergence and consequences* [Online]. University of Oxford: University of Oxford Available at: https://archive.org/stream/PolytheticClassificationConvergenceAndConsequences/65935-Rodney-Needham-Polythetic-Classification-Convergence-and-Consequences#page/n1/mode/2up. pp. 349-357.

Nosofsky, R.M., Pothos, E.M. & Wills, A.J. (2011) The Generalized Context Model: An Exemplar Model of Classification. *Formal Approaches to Categorization*, pp.18-39.

Osherson, D.N. & Smith, E.E. (1981) On the adequacy of prototype theory as a theory of concepts. *Cognition*, (9), pp.35-38.

Parnas, D.L. (n.d.) *On the Criteria To Be Used in Decomposing Systems into Modules. Programming Techniques* [Online]. Available at: https://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf .

Pergl, R., Buchtela, D., Rybola, Z. & Ryant, I. (2012) OntoUML a UFO-A pro softwarové inženýrství. In *Proceedings of conference Objekty 2012*., 2012.

Petre, M. (2013) UML in practice. In *35th International Conference on Software Engineering (ICSE 2013)*. San Francisco, 2013.

Philosophy Index. (n.d.) *Family Resemblance in Wittgenstein* [Online]. Available from: http://www.philosophy-index.com/wittgenstein/family-resemblance/ .

Philosophy, S.E.o. (2011) *Identity Over Time* [Online]. Available from: http://plato.stanford.edu/entries/identity-time/ .

Quora. (n.d.) *Is watermelon a fruit or a vegetable?* [Online]. Available from: https://www.quora.com/Is-watermelon-a-fruit-or-a-vegetable .

Rational Software. (n.d.) *Rational Software White Paper (TP026B)* [Online]. Available at: http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_best practices_TP026B.pdf .

Reenskaug, T. (2008) *The Common Sense of Object Orientated Programming* [Online]. Available from: http://folk.uio.no/trygver/2008/commonsense.pdf .

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 .

Rollins, A. (2014) *The Cake Pattern in Scala - Self Type Annotations vs. Inheritance* [Online]. Available at: http://www.andrewrollins.com/2014/08/07/scala-cake-pattern-self-type-annotations-vs-inheritance/ .

Rosch, E. (1973) Natural categories. *Cognitive Psychology*, (4), pp.328-50.

Roth, E.M. & Mervis, C.A. (1983) Fuzzy Set Theory and Class Inclusion Relations in Semantic Categories. *Journal of Verbal Learning and Verbal Behavior*, (22), pp.509-25.

Rouder, J.N. & Ratcliff, R. (2006) Comparing exemplar- and rule-based theories of categorization. *Current Directions In Psychological Science*.

Rumbaugh, J. et al. (1991) *Object-Oriented Modeling and Design*. New Jersey: Prentice-Hall.

Savkin, V. (2012) *Data Context Interaction: The Evolution of the Object Oriented Paradigm* [Online]. Available at: http://www.sitepoint.com/dci-the-evolution-of-the-object-oriented-paradigm/ .

Scala Documentation. (n.d.) *Def Macros* [Online]. Available at: http://docs.scala-lang.org/overviews/macros/overview.html .

Scala Documentation. (n.d.) *Implicit Conversions* [Online]. Available at: http://docs.scala-lang.org/tutorials/tour/implicit-conversions .

Scala Documentation. (n.d.) *Scala Documentation: Pattern matching* [Online]. Available at: http://docs.scala-lang.org/tutorials/tour/pattern-matching.html .

Scala School. (n.d.) *Pattern matching & functional composition* [Online]. Available at: https://twitter.github.io/scala_school/pattern-matching-and-functional-composition.html .

Scala. (n.d.) *A Tour of Scala: Traits* [Online]. Available at: http://www.scala-lang.org/old/node/126 .

Sekharaiah, K.C. & Ram, D.J. (2002) Object Schizophrenia Problem in Modeling Is-role-of Inheritance. In *Object-Oriented Information Systems: 8th International Conference, OOIS.*, 2002.

Siau, K. & Cao, Q. (2001) The Unified Modeling Language (UML) – A Complexity Analysis. *Journal of Database Management*, 12(1), pp.26-34.

Spindler, O. & Fadrus, T. (n.d.) *Grimace Project* [Online]. Available at: http://www.grimace-project.net/ .

Spindler, O. (n.d.) *Affective space interfaces* [Online]. Wien, AU: Fakultät für Informatik der Technischen Universität Available at: http://www.grimace-project.net/assets/affectivespaceinterfaces.pdf .

Stanford Encyclopedia of Philosophy. (2011) *Identity Over Time* [Online]. Available from: http://plato.stanford.edu/entries/identity-time/ .

Stanford Encyclopedia of Philosophy. (2011) *Sorites Paradox* [Online]. Available from: http://plato.stanford.edu/entries/sorites-paradox/ .

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.

Šlajchrt, Z. (2015) *A Demonstration of Using Morpheus to Model Human Facial Expressions* [Online]. Available at: https://github.com/zslajchrt/morpheus-emo .

Šlajchrt, Z. (2016) A Solution to the Square-Rectangle Problem Within the Framework of Object Morphology. *Acta Informatica Pragensia*, 5(1), p.34–45.

Šlajchrt, Z. (2016) Designing Applications Using Data-Context-Interactions Architecture In Morpheus. *Journal of Systems Integration*, 7(2).

Šlajchrt, Z. (2016) *OntoUML in Morpheus Demonstration* [Online]. Available at: https://github.com/zslajchrt/morpheus-tutor/tree/master/src/main/scala/org/cloudio/morpheus/ontouml .

Taivalsaari, A. (1996) Classes vs. Prototypes, Some Philosophical and Historical Observations. *Journal of Object-Oriented Programming*, pp.44-50.

The Guardian. (2007) *It's a scandal: Oklahoma declares watermelon a vegetable* [Online]. Available from: http://www.theguardian.com/world/2007/apr/18/usa.matthewweaver .

TIOBE. (n.d.) *Index for January2016* [Online]. Available at: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html .

Wagner, G. (n.d.) *oNTOjs* [Online]. Available from: https://bitbucket.org/gwagner57/ontojs .

Way, E.C. (1991) *Knowledge Representation and Metaphor*. Kluwer Academic Publishers.

Wirfs-Brock, R. & McKean, A. (2003) *Object Design: Roles, Responsibilities, and Collaborations*. Boston: Addison-Wesley.

Wittgenstein, L. (1953) *Philosophical Investigations*. Blackwell Publishing.

Zadeh, L. (1965) Fuzzy sets. *Information and control*, pp.338-53.

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.

List of Figures

Listings

List of Tables