DeepRuby: Extending Ruby with Dual Deep Instantiation

Bernd Neumayr, Christoph G. Schuetz, Christian Horner, and Michael Schrefl

Department of Business Informatics – Data & Knowledge Engineering
Johannes Kepler University Linz, Austria
{bernd.neumayr, christoph.schuetz, michael.schrefl}@jku.at

Abstract. Clabjects, the central construct of multi-level modeling, overcome the strict separation of class and object in conceptual modeling. Ruby, a dynamic object-oriented programming language, similarly treats classes as objects and thus appears as a natural candidate for implementing clabject-based modeling constructs. In this paper we introduce DeepRuby, a Ruby implementation of the core constructs of Dual Deep Instantiation: clabject hierarchies and attributes with separate source potency and target potency. DeepRuby represents clabjects at two layers: the clabject layer and the clabject facet layer. At the clabject facet layer, a clabject with maximum source potency i-1 and maximum target potency j-1 is represented by a matrix of $i \times j$ clabject facets organized using Ruby's superclass and eigenclass constructs. Clabject facets can easily be extended with behavior implemented in custom methods.

Keywords: Multi-level Modeling, Object-oriented Programming

1 Introduction

Object-orientation is arguably the most important paradigm in programming and conceptual modeling. Statically-typed object-oriented programming languages, like Java, and traditional conceptual modeling approaches, like E/R and UML, come with a strict separation between class and object. The clabject as central construct of multi-level modeling [8] overcomes this separation and not only plays the roles of class and object but also of metaclass, potentially at many classification levels. Extending traditional modeling/programming languages to supporting clabjects is difficult, due to this inherent mismatch. Dynamically typed languages like Ruby overcome the strict separation between object and class: classes are also treated as objects and may be extended at runtime. Based on this kinship, Ruby suggests itself as a suitable language for implementing multilevel modeling constructs.

Deep Instantiation [2] is one of the most prominent approaches to multi-level modeling. A potency assigned to a clabject or a property indicates the maximum instantiation depth, i.e., the number of instantiation steps to reach the ultimate instance of the clabject or property. For example, a clabject *Product* with potency 3 is instantiated by clabject *Car* with potency 2 which in turn is

instantiated by clabject *BMW Z4* with potency 1 which in turn is instantiated by *Peters Car* with potency 0 which cannot be further instantiated. Clabject *Car* defines a property *engine* with potency 2 and target *CarEngine*, which is instantiated by members of *Car*, e.g., *BMW Z4 has engine EngineK5* and in turn by the members of *Car*'s members, e.g., *Peters Car has engine Engine123*; *Engine123* is a member of *EngineK5* which in turn is a member of *Car Engine*.

Dual Deep Instantiation [7] (DDI) allows to specify the number of instantiation steps separately for the source and for the target of a property. For example, clabject *Product*, as source, introduces a property *owner* with source potency 3 and target *Person* with target potency 1. The property is ultimately instantiated by *Peters Car has owner Peter*. *Peters Car* is an instantiation of *BMWZ4* which is an instantiation of *Car* which is an instantiation of *Product*. *Peter*, on the the target side, directly instantiates *Person*. In previous work, we implemented/-formalized different variants of DDI in deductive database languages, namely F-Logic [7] and ConceptBase [9], without support for implementing behavior.

It is sometimes argued that (dual) deep instantiation's support for concise modeling comes with the price of lack of conceptual clarity [3]: one clabject may represent multiple domain concepts which makes it more difficult to differentiate these different domain concepts. For example, clabject *Car* with potency 2 represents an individual product category with category manager *Ms Black*, the class of all car models with members such as *BMWZ4*, the class of all car individuals with members such as *Peters Car* and also the metaclass for car individuals. One of the challenges of realizing (dual) deep instantiation is to hide this complexity while making these different facets of a clabject explicit and accessible.

In this paper we introduce DeepRuby, an implementation of (a simple variant of) DDI in Ruby. DeepRuby makes heavy use of Ruby's dynamic programming and metaprogramming facilities [10], most notably of its *eigenclass* (also referred to as *singleton class*) construct. DeepRuby distinguishes a *clabject layer* and a *clabject facet layer*. At the clabject layer, each clabject corresponds to one Ruby object. At the clabject facet layer, there is one Ruby object for every clabject facet, i.e., for every potential combination of source potency and target potency. A clabject with maximal source potency i and maximal target potency j has $(i+1) \times (j+1)$ clabject facets. The objects representing clabject facets are also classes and the programmer can introduce custom methods with these clabject facets. Clabject facets of the same clabject and same source potency with different target potencies are connected by eigenclass relationships. The clabject facets of a clabject that instantiates another clabject (its class) are connected to the class's clabject facets by superclass relationships along which methods and attribute values are inherited.

In the remainder of the paper, we give, in Sect. 2, an introduction to Ruby's object model, followed, in Sect. 3, by an example DDI model represented in DeepRuby. Sect. 4 explains the clabject facet layer. Sect. 5 exemplifies the extension of clabject facets with custom methods. Sect. 6 gives an overview of related work. Sect. 7 concludes the paper with ongoing and future work regarding the implementation of advanced constructs of DDI [7] and Dual Deep Modeling [9].

2 Background: Ruby's Object Model

As a background for forthcoming sections this section explains some relevant aspects of Ruby's object model along a small but intricate example (see Fig. 1).

Ruby's modules provide a namespacing mechanism for constants, such as class names. Class `Person` (see line 1:2, that is line 2 in the listing in Fig. 1) is created within module `Social` and can be accessed outside the module by qualified name `Social::Person` (line 1:32, note: method `puts` writes a string representation of the given object to an IO stream).

Member attributes of a Ruby class are defined as getter and setter methods that access instance variables. Instance variables are created when set by a method. To avoid the need to write getters and setters by hand, class `Module` provides a method `attr_accessor` that creates getter and setter methods for an attribute of a given name. For example, in line 1:3, class `Person` (an instance of `Class` which inherits from class `Module`) calls `attr_accessor` for symbol `:age` to

```
1  module Social
2    class Person
3      attr_accessor(:age)
4      def initialize(a)
5        self.class.incCounter
6        @age = a; end
7    end
8    class << Person
9      def counter; @counter; end
10     def incCounter
11       if @counter.nil?; @counter = 0; end
12       @counter = @counter + 1; end
13   end
14   class Woman < Person; end
15   Mary = Woman.new(31)
16   Mary.age = 27
17   puts Mary.age         # => 27
18   puts Person.counter   # => nil
19   puts Woman.counter    # => 1
20   class << Mary
21     attr_accessor(:name)
22   end
23   Mary.name = "Maria"
24 end
25 class Object
26   def inspect2
27     str = "(#{self}"
28     instance_variables.each {|x| str = str +
29        " #{x}=#{instance_variable_get(x)}"}
30     str += ")"; end
31 end
32 puts Social::Person.inspect2
33 # => (Social::Person)
34 puts Social::Woman.inspect2
35 # => (Social::Woman @counter=1)
36 puts Social::Mary.inspect2
37 # => (#<Social::Woman:0x...> @age=27
     @name="Maria")
```

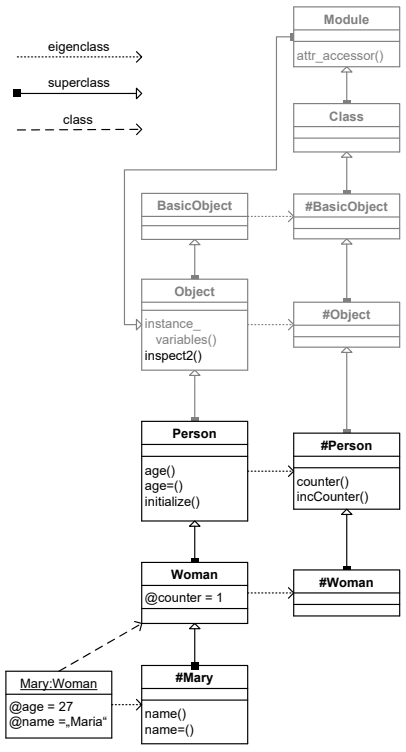

Fig. 1. Introductory example to Ruby's object model: Ruby code and custom graphical representation of Ruby objects. Predefined objects are depicted in grey

create setter method `age=` and getter method `age` in class `Person` to write and read instance variables `@age` (names of instance variables are marked by prefix '`@`') of instances of class `Person`, such as `Mary` (see line 1:16 and line 1:17).

In Ruby, classes are treated as objects and can have instance variables themselves, called class instance variables. Classes are instances of class `Class` and also may have an eigenclass (also referred to as singleton class). Methods defined with a class's eigenclass (also referred to as singleton methods or class methods) can be used to access a class's class instance variables. The eigenclass of a class has as superclass the eigenclass of the class's superclass. For example, `Person`'s eigenclass (labelled `#Person` in the graphical representation) is opened by '`class << Person`' at line 1:8. `Person`'s eigenclass defines a getter method `counter` (line 1:9) together with a method `incCounter` (line 1:10) which is called (line 1:5) to increment the counter every time a new object is created. Class `Woman` has superclass `Person` (defined by '`class Woman < Person`' at line 1:14) and, thus, `#Woman` (the eigenclass of `Woman`) has superclass `#Person` (the eigenclass of `Person`).

Class instance variables really belong to the class (as an object) and class methods are called in the context of a class object. For example, when calling `Woman.new` to create a new instance of class `Woman` the initializer defined with class `Person` (line 1:4) is called, `incCounter` is called in the context of class `Woman` setting instance variable `@counter` of `Woman` to 1 (see comment in line 1:19) and not of `Person` which remains undefined (see comment in line 1:18).

Single objects may also have singleton classes with singleton methods. For example, `Mary`'s singleton class (opened at line 1:20 by `class << Mary` and depicted as `#Mary`) defines getter and setter methods for accessing instance variable `@name` of `Mary`.

Ruby allows to open existing classes to add additional methods which then affect all direct and indirect instances of the class. For example, class `Object` (opened at line 1:25) is the direct or indirect superclass of all custom classes created in Ruby programs and also the superclass of class `Module` and `Class`. A method added to class `Object` can thus be called from any Ruby object (with Ruby classes being also Ruby objects). Method `inspect2` (line 1:26) is defined with class `Object`; when invoked on an object, it creates a string consisting of the object's name and its instance variables (see lines 1:32–1:36).

3 Dual Deep Instantiation in Ruby – an Example

In this section DeepRuby is explained along the example depicted and implemented in Fig. 2. Ruby's modules are used as namespacing mechanism. The clabjects of a DDI model are created within such a module/namespace. For example, module `SalesMgmt` (line 2:1) serves as namespace for a DDI model with depth 3 (line 2:2), i.e., a model with maximum source and target potencies of 3.

Creating clabject hierarchies. A DDI model consists of one or more clabject hierarchies. Every clabject hierarchy has one root clabject. A root clabject has a fixed clabject potency (specifying the number of instantiation levels beneath

the root) and typically has a name. For example, clabject `Person` (line 2:3) and clabject `Product` (line 2:11) are the root clabjects in the `SalesMgmt` model and have a potency of 1 and 3, respectively.

Clabjects are instantiated by sending message `new`. The new clabject is in the same module as its class and has a potency 1 lower than its class. For example, clabject `Person` with potency 1 is instantiated by `MsBlack` (line 2:4) and by `Peter` (line 2:5), which get potency 0. Clabject `Product` with potency 3 is instantiated by `Bike` (line 2:14) and by `Car` (line 2:19) which get potency 2.

Defining and instantiating attributes. Attributes are defined with a source clabject, a name, a source potency, a target potency, and a target clabject. For example, clabject `Product` defines an attribute with name `owner`, source potency 3, target potency 1, and target clabject `Person` (line 2:13).

A clabject has many clabject facets, one for each combination of source potency and target potency. In order to set attribute `engine` at source potency 1 and target potency 1 at clabject `BMWZ4` to `EngineK5`, one first selects the clabject facet (`BMWZ4.^(1,1)`) to which one sends `engine=EngineK5` (line 2:25).

```
1  module SalesMgmt
2     p = DDI::Model.new(SalesMgmt,3)
3     DDI::Clabject.new(p,1,:Person)
4     Person.new(:MsBlack)
5     Person.new(:Peter)
6     Peter.define(:spouse,0,1,Person)
7     Peter.^(0,0).spouse = MsBlack
8     DDI::Clabject.new(p,2,:CarEngine)
9     CarEngine.new(:EngineK5)
10    EngineK5.new(:Engine123)
11    DDI::Clabject.new(p,3,:Product)
12    Product.define(:categoryMgr, 1, 1, Person)
13       .define(:owner, 3, 1, Person)
14    Product.new(:Bike)
15    Bike.new(:BromptonM6L)
16    Bike.^(0,0).categoryMgr = MsBlack
17    BromptonM6L.new(:PetersBike)
18    PetersBike.^(0,0).owner = Peter
19    Product.new(:Car)
20    Car.define(:engine, 2, 2, CarEngine)
21    Car.categoryMgr = MsBlack
22    Car.new(:BMWZ4)
23    BMWZ4.^(1,1).engine = EngineK5
24    BMWZ4.new(:PetersCar)
25    PetersCar.^(0,0).owner = Peter
26    PetersCar.^(0,0).engine = Engine123
27    Person.define(:favouriteItem, 1, 3, Product)
28    Peter.^(0,0).favouriteItem = PetersCar
29    MsBlack.^(0,1).favouriteItem = BromptonM6L
30    puts Peter.favouriteItem.name
31    # =>  PetersCar
32    puts PetersCar.^(0,1).engine.name
33    # => EngineK5
34    Product.getMembersN(2).each{|c| puts c.name }
35    # => BromptonM6L \n BMWZ4
36 end
```

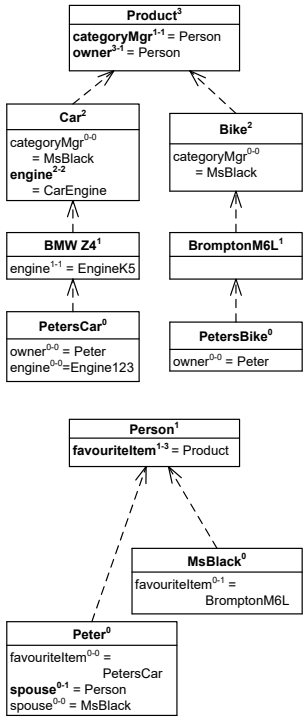

Fig. 2. Running example: A DeepRuby program (left) realizing a DDI model (right)

Clabjects with potency 0 have no members, yet they may define attributes with a target potency higher than 0, similar to what can be accomplished in Ruby with singleton classes of an object (e.g., attribute `name` defined with `Mary`'s singleton class at line 21 in Fig. 1). For example, clabject `Peter` defines an attribute `spouse` with source potency 0, target potency 1, and target `Person` (line 2:6) and instantiates it with target potency 0 and target `MsBlack` (line 2:7).

Root clabjects with clabject potency 1 are akin to 'normal' classes in that they have individuals as members. They are different from normal classes in that their attributes may have a range defined at a higher classification level. For example, `Person` (line 2:3) has individuals `MsBlack` (line 2:4) and `Peter` (line 2:5) as members, yet it defines an attribute `favouriteItem` (line 2:27) with target `Product` and target potency 3, meaning that the range of `favouriteItem` is given by the members of the members of the members of clabject Product.

Querying clabject hierarchies and attributes. The values and (meta) types of a clabject's attributes are queried by sending the attribute name to the clabject facet which is identified by the clabject together with source potency and target potency. For example, sending attribute name `engine` to PetersCar's clabject facet with source potency 0 and target potency 1 (line 2:32) returns the type of `engine` of `PetersCar`, which is `EngineK5`, which is inherited from `BMWZ4`.

For getting or setting attributes with source potency 0 and target potency 0 it is not necessary to specify the clabject facet. If a message is sent to a clabject it dispatches it to its 0-0 facet. For example, when sending attribute name `favouriteItem` to `Peter` (line 2:30) it is dispatched to `Peter^(0,0)` and retrieves `Peter`'s favourite item, which is his car.

DeepRuby provides methods to navigate clabject hierarchies to facilitate flexible querying of DDI models. For example, line 2:34 retrieves the members of the members of `Product`, these are `BMWZ4` and `Brompton`.

DeepRuby provides (1) generic query mechanisms (1a) to retrieve attribute values and (meta) types including inherited values and types (1b) to navigate clabject hierarchies and retrieve a clabject's members at a specific level and (2) takes care of keeping DDI models consistent when defining and setting attributes: (2a) correct number of instantiation steps at the source and the target (2b) target clabjects are compatible with targets at higher potencies, (2c) a newly introduced target does not produce type conflicts at lower potencies and at descending clabjects.

4 DeepRuby under the Hood

By freely combining source and target potencies, a clabject c with maximum source potency m (given by the clabject's potency) and maximum target potency n (given by the DDI model's depth) has $(m + 1) \times (n + 1)$ clabject facets. Every such facet corresponds to a combination of source potency and target potency. The basic idea of DeepRuby is to represent every such clabject facet as a 'flat' Ruby object (which in the current approach is always a class) with instance variables and methods. For example, clabject `Car` with potency 2 in

a model with depth 3 has 12 (3×4) clabject facets. The object `Car^(0-0)` holds `@catMgr`=MsBlack and `Car^(2-2)` holds `@engine=CarEngine` as instance variable. The relationships between clabject facets are represented using Ruby constructs:

- The *eigenclass* of clabject facet $c^{i,j}$ is clabject facet $c^{i,(j+1)}$. For example, the eigenclass of class `Car^(0,0)` is clabject facet `Car^(0,1)`.
- If clabject c is an instantiation of clabject d, then every clabject facet $c^{i,j}$ has clabject facet $d^{(i+1),j}$ as *superclass*. For example, `Car^(0,0)` has superclass `Product^(1,0)` and `Car^(0,1)` has superclass `Product^(1,1)`.

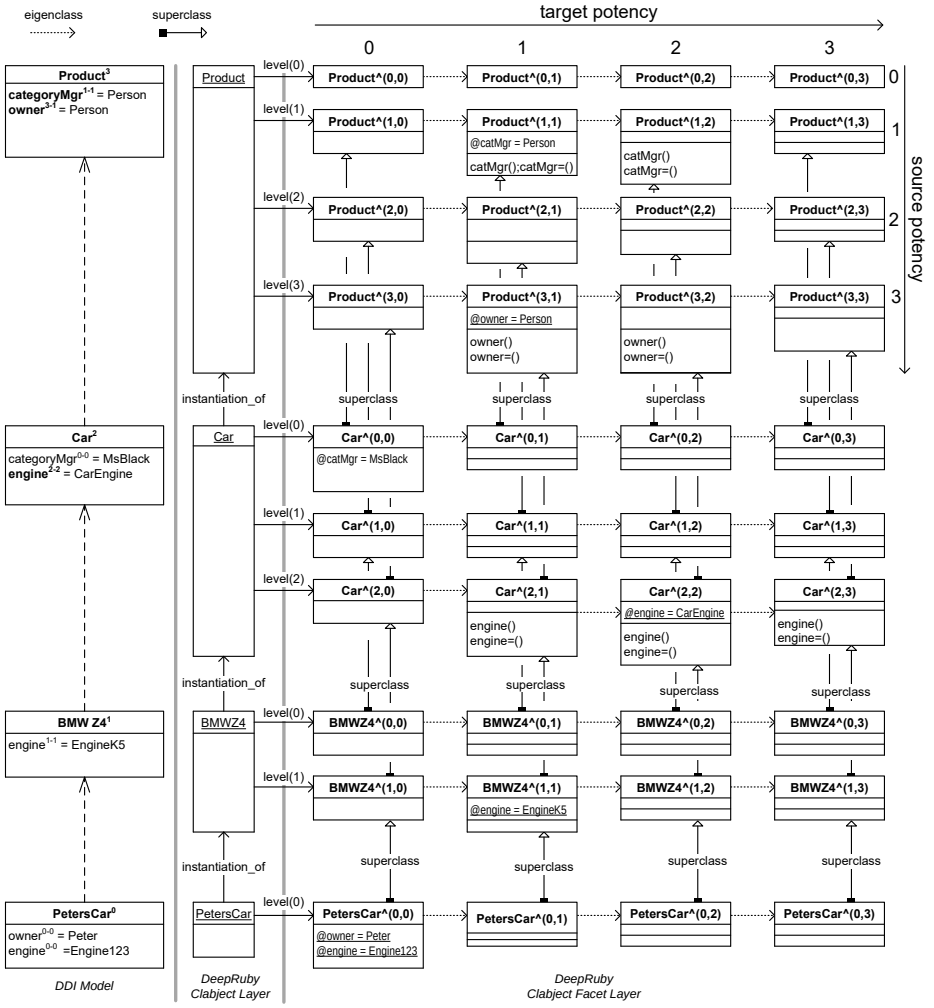

Fig. 3. Realizing clabject facet matrices in Ruby using superclass and eigenclass

A clabject is first represented as an instance of class `Clabject` (line A:25 in the Appendix) with an array `levels` which holds for each source potency a reference to the respective instance of class `ClabjectFacet` (see line A:211) with target potency 0, from there one can navigate to other clabject facets along eigenclass relationships. Sending message `^(i,j)` to a clabject c returns clabject facet $c^{i,j}$. A clabject facet's attribute `clabject` (line A:214) allows to navigate back from clabject facet to clabject; for example, from clabject facet `Car^(2,1)` to clabject `Car`.

What is the role of superclass relationships in DeepRuby?

- Methods are inherited from superclass $d^{i+1,j}$ to subclass $c^{i,j}$ (this comes for free, since this is what class hierarchies are traditionally used for). For example, setter method `engine=` defined at `Car^(2,1)` is inherited by `BMWZ4^(1,1)` and in turn by `PetersCar^(0,1)`.
- Target clabjects (represented as instance variables) are inherited from superclass $c^{i+1,j}$ to sublass $c^{i,j}$ (this is implemented generically as part of DeepRuby). For example, when sending message `engine` to `PetersCar^(0,1)` one gets `EngineK5`, which is inherited from `BMWZ4^(1,1)`.

What is the role of eigenclass relationships in DeepRuby?

- The eigenclass $c^{i,j+1}$ of a clabject facet $c^{i,j}$ provides methods for accessing instance variables of $c^{i,j}$ (this comes for free with the eigenclass construct). For example, setter method `catMgr=` defined in `Product^(1,2)` is called for setting `@catMgr=Person` in `Product^(1,1)`
- Target clabjects (represented as instance variables) at $c^{i,j+1}$ act as constraint for target clabjects at $c^{i,j}$ (this is implemented generically as part of DeepRuby). For example, target clabject `engine=EngineK5` of `PetersCar^(0,1)` (inherited from `BMWZ4^(1,1)`) acts as constraint when invoking setter method `engine=` on `PetersCar^(0,0)`.

In this section we have explained the basic principles of DeepRuby's implementation and use of Ruby's eigenclass construct to implement Dual Deep Instantiation. For more details we refer the interested reader to the source code which is provided in the Appendix.

5 Simple Attributes and Custom Methods in DeepRuby

Using classes/eigenclasses arranged in superclass hierarchies for realizing the clabject facet matrix allows to use standard Ruby constructs to implement simple attributes (attributes with non-clabjects as range) and behavior (custom methods) on top of the clabject facet matrix, and to specialize behavior (i.e., overwrite methods, add additional methods) along the clabject hierarchy.

To demonstrate these features, the running example from Fig. 2 is extended in Fig. 4 with clabject hierarchies `Currency` with simple attributes for exchange

rate (with Euro as reference currency), isocode and value, and `Country` with a local currency. Clabject `Product` is extended with a `listPrice` and a method `priceInCountry` to convert the list price to the local currency of the given country.

First-level members of `Currency` receive getters and setters for simple attributes `exchRate` and `isocode` by invoking standard Ruby method `attr_accessor` on the eigenclass of `Currency.^(1,0)` (which is `Currency.^(1,1)`) (see line 4:4).

```
1  module SalesMgmt
2    DDI::Clabject.new(Product.model,2,:Currency)
3    class << Currency.^(1,0)
4      attr_accessor(:isocode, :exchRate)
5    end
6    class << Currency.^(2,0)
7      attr_accessor(:value)
8      def pretty
9        "#{clabject.cclass.isocode} #{value}"
10     end
11     def toCurrency(c)
12       raise "#{c} is not a currency" \
13             unless (c.isMemberN(1,Currency))
14       obj = c.new
15       obj.value = (value * clabject.cclass
16            .exchRate / c.exchRate).round(2)
17       return obj
18     end
19   end
20   Currency.new(:Pound);
21   Pound.isocode = "GBP"; Pound.exchRate = 1.12;
22   Currency.new(:Euro)
23   Euro.isocode = "EUR"; Euro.exchRate = 1
24   Currency.new(:Yen)
25   Yen.isocode = "JPY"; Yen.exchRate = 0.0077
26   Pound.new(:GBP38200); GBP38200.value = 38200

27   puts GBP38200.pretty # => GBP 38200
28   puts GBP38200.toCurrency(Euro).pretty
29   # => EUR 42784.0
30   class << Yen.^(1,0)
31     def pretty; "\u00A5 #{value}"; end
32   end
33   puts GBP38200.toCurrency(Yen).pretty
34   # =>  ¥ 5556363.64
35   DDI::Clabject.new(Product.model,1,:Country)
36   Country.define(:localCurrency,1,1,Currency)
37   Country.new(:UK).localCurrency = Pound
38   Country.new(:Japan).localCurrency = Yen
39   puts Japan.localCurrency.exchRate #=> 0.0077

40   Product.define(:listPrice, 2, 2, Currency)
41   class << Product.^(2,0)
42     def priceInCountry(country)
43       listPrice.toCurrency(country.localCurrency)
44     end
45   end
46   BMWZ4.^(0,0).listPrice = GBP38200
47   puts BMWZ4.priceInCountry(Japan).pretty
48   # =>  ¥ 5556363.64
49 end
```

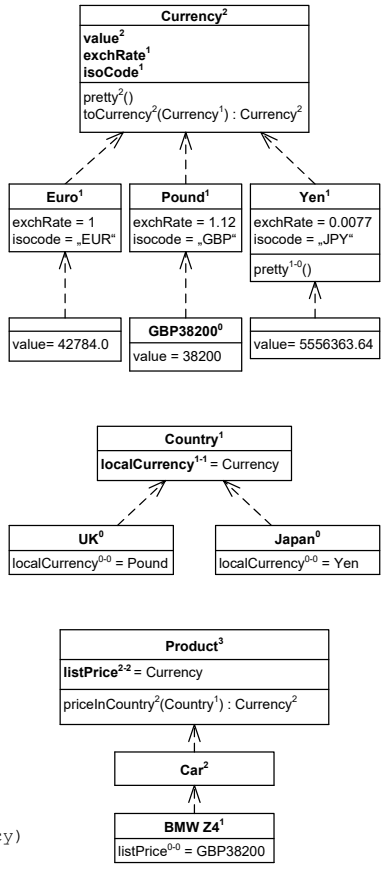

Fig. 4. Custom methods in DeepRuby

Second-level members of `Currency` get getter and setter for attribute `value` by invoking `attr_accessor` on the eigenclass of `Currency.^(2,0)` (which is `Currency.^(2,1)`) (line 4:7). Currencies `Pound`, `Euro`, and `Yen` are created with their isocode and exchange rate (lines 4:20–4:25). `GBP38200` is an instantiation of `Pound` (and a second-level member of `Currency`) with value 38200 (line 4:26).

Second-level members of `Currency` have a method for `pretty` printing (defined with the eigenclass of `Currency.^(2,0)` which is `Currency.^(2,1)`), making use of `value` and `isocode`. In order to get the `isocode` the method needs to first navigate from the clabject facet (instance of `ClabjectFacet`) to the corresponding clabject (instance of `Clabject`) along attribute `clabject` and from there along attribute `cclass` to the corresponding first-level member of `Currency` (line 4:9). Sending `pretty` to `GBP38200` results in 'GBP 38200' (line 4:27).

Second-level members of `Currency` further have a method `toCurrency` which takes a first-level member of `Currency` as parameter (line 4:11). Sending `toCurrency` with parameter `Euro` to `GBP38200` produces a new instantiation of `Euro` which is pretty printed as 'EUR 42784.0' (line 4:28).

The eigenclass of `Yen.^(1,0)` (which is `Yen.^(1,1)`) overwrites method `pretty` inherited from `Currency.^(2,1)`) to use unicode symbol ¥ instead of isocode `JPY` . The new instantiation of `Yen` created by sending `toCurrency` with parameter `Yen` to `GBP38200` is pretty printed as '¥ 5556363.64' (line 4:33).

Clabjects `UK` and `Japan` instantiate `Country` and have local currencies `Pound` and `Yen`, respectively. Asking for the exchange rate of `Japan`'s local currency returns `0.0077` (line 4:39).

Method `priceInCountry` (see line 4:42) of second-level members of `Product` takes a country as parameter and converts the `listPrice` of second-level instantiations of `Product` to the country's local currency, returning a new instantiation of the given currency with the value being the result of the conversion. Sending `priceInCountry` with parameter `Japan` to `BMWZ4` (see line 4:48) returns a new clabject pretty-printed as '¥ 5556363.64' (line 4:48).

6 Related Work

With the advent of multi-level modeling, the question of multi-level model execution emerges. Melanee [1], DeepTelos [4], MetaDepth [6], and DeepJava [5] are modeling tools and frameworks that support model execution, each pursuing a different strategy with respect to supporting model execution.

The Melanee multi-level modeling tool [1] supports model execution through a service API and a plug-in mechanism. The communication between modeling and execution environment can be realized using socket-based communication. Changes in the modeling environment then automatically reflect in the execution environment, and vice versa. The execution environment can be implemented as a Java program. Concerning the definition of execution semantics, different approaches exist. A "pragmatic" approach, for example, employs a Java representation of the multi-level model where each clabject in the multi-level model

corresponds to a single Java class, with execution semantics defined using plain Java code.

DeepTelos [4] extends the Telos metamodeling language with "most general instances" to add support for deep instantiation. Since DeepTelos defines the extensions as a set of Datalog axioms, DeepTelos models are compatible with ConceptBase, an implementation of a Telos variant. ConceptBase also allows for the definition of executable models using event-condition-action rules.

MetaDepth [6] is a text-based multi-level modeling framework with potency-based deep instantiation. MetaDepth is a Java-based implementation using a custom syntax. Among the primary features of MetaDepth are multi-level constraints and derived attributes at different meta-levels. Execution semantics is defined using an OCL extension. MetaDepth provides an interpreter for the thus defined multi-level models. MetaDepth also supports code generation complying to the Java Metadata Interface.

DeepJava [5] is an extension of the Java programming language with a mechanism for potency-based deep instantiation. Internally, a compiler transforms DeepJava code into plain Java. Hence, each DeepJava class translates into a set of Java classes, one for each clabject facet. The compiler also generates code for clabject instantiation at runtime, which is realized using Java's reflective functions. Clabject instantiation results in the dynamic generation of a number of interfaces. As a limitation, direct access without getters and setters is restricted to attributes with potency values smaller than two. With respect to Java, Ruby's eigenclass concept much better suits the clabject philosophy of multi-level modeling. As opposed to DeepJava, DeepRuby supports deep instantiation with both a source and a target potency, resulting in the generation of a matrix of Ruby classes for each clabject.

7 Conclusion

In this paper we introduced DeepRuby, a Ruby implementation of the core language constructs of Dual Deep Instantiation [7]: clabject hierarchies and attributes with dual potencies. The system takes care of consistent instantiation of clabjects and attributes and provides methods for querying multi-level models. Our experiences with implementing DeepRuby have confirmed our initial conjecture that a dynamic programming language like Ruby that does not strictly separate classes and objects is a good platform for implementing clabject-based modeling constructs.

In an internal (not yet publication-ready) prototype we have also implemented DDI's advanced modeling constructs (which are missing from the DeepRuby version presented in this paper): multi-valued properties, bi-directional properties, and clabject generalization. We have also been experimenting with an alternative representation of the clabject facet matrix where clabject facets with target potency 0 are represented as simple objects and not as classes; a seemingly reasonable design choice since these facets do not act as classes, yet it

makes the implementation a bit more complex. The fine-tuning of the advanced prototype is subject to ongoing work.

Dual deep modeling (DDM) [9], an extended version of DDI, additionally comes with multi-level cardinality constraints, property specialization hierarchies, and distinguishes between property value and property range. Implementing these constructs in DeepRuby is subject to future work.

Moving beyond previous implementations of DDI/DDM in ConceptBase [7] and F-Logic [9], DeepRuby allows to extend clabject facets with custom methods. In this paper we have exemplified the implementation of such methods and their inheritance and specialization along the clabject facet hierarchy.

References

1. Atkinson, C., Gerbig, R., Metzger, N.: On the execution of deep models. In: Mayerhofer, T., Langer, P., Seidewitz, E., Gray, J. (eds.) Proceedings of the 1st International Workshop on Executable Modeling. CEUR Workshop Proceedings, vol. 1560, pp. 28–33. CEUR-WS.org (2015)
2. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: Gogolla, M., Kobryn, C. (eds.) Proceedings of the 4^{th} International Conference on the UML 2001, Toronto, Canada. LNCS, vol. 2185, pp. 19–33. Springer Verlag (Oct 2001)
3. Carvalho, V.A., Almeida, J.P.A.: Toward a well-founded theory for multi-level conceptual modeling. Software & Systems Modeling (2016)
4. Jeusfeld, M.A., Neumayr, B.: DeepTelos: Multi-level modeling with most general instances. In: Comyn-Wattiau, I., Tanaka, K., Song, I., Yamamoto, S., Saeki, M. (eds.) ER 2016. LNCS, vol. 9974, pp. 198–211. Springer (2016)
5. Kuehne, T., Schreiber, D.: Can programming be liberated from the two-level style: Multi-level programming with DeepJava. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. pp. 229–244 (2007)
6. de Lara, J., Guerra, E.: Deep meta-modelling with MetaDepth. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 1–20. Springer (2010)
7. Neumayr, B., Jeusfeld, M.A., Schrefl, M., Schütz, C.: Dual deep instantiation and its conceptbase implementation. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE. Lecture Notes in Computer Science, vol. 8484, pp. 503–517. Springer (2014)
8. Neumayr, B., Schuetz, C.G.: Multilevel modeling. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems. pp. 1–8. Springer New York, New York, NY (2017), http://dx.doi.org/10.1007/978-1-4899-7993-3_80807-1
9. Neumayr, B., Schuetz, C.G., Jeusfeld, M.A., Schrefl, M.: Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic. Software & Systems Modeling pp. 1–36 (2016)
10. Perrotta, P.: Metaprogramming Ruby 2. The Pragmatic Programmers (2014)

A DeepRuby Implementation

```
module DDI

class Model
  attr_reader(
    :depth, # maximum potency of user clabjects
    :modul, # module/namespace for clabjects
  )
  def initialize(mod, n)
    raise "initial maximum-depth too low (n < 1)"
        if n < 1
    @modul = mod
    @depth = n
    # create module-specific ClabjectLevel-class
    # because of different eigenclass depth
    cls = Class.new
    modul.const_set(:ClabjectLevel, cls)
    # include module ClabjectFacet in eigenclasses
    for n in (0..depth)
      cls = cls.singleton_class
      cls.send(:include, ClabjectFacet)
    end
  end
end # end Model


class Clabject
  attr_reader(
    :model,   # reference to DDI model
    :levels,  # array of clabject levels
    :name,
    :instantiations,   # first-level members
    :instantiation_of, # class clabject
    :potency
  )

  def cclass
    instantiation_of
  end

  def members
    instantiations
  end

  def ^(srcPtcy,tgtPtcy)
    facet(srcPtcy,tgtPtcy)
  end

  # get clabject facet
  def facet(srcPtcy,tgtPtcy)
    raise "Target potency #{tgtPtcy} above max
        potency #{model.depth}+1." if tgtPtcy >
        (model.depth+1)
    return levels[srcPtcy].eigenclassN(tgtPtcy)
  end

  def initialize(model, potency, name=nil,
      parent=nil)
    @name = name
    @instantiation_of = parent
    @potency = potency
    @model = model
    @levels = Array.new(potency+1)
    for m in (0..potency)
      if parent.nil?
        @levels[m] = createClabjectLevel(model
            .modul.const_get(:ClabjectLevel), m)
      else
        @levels[m] = createClabjectLevel(parent
            .levels[m+1], m)
      end
    end
    @instantiations = Array.new
    model.modul.const_set(name, self) if name
    return self
  end

  def new(name=nil)
    obj = Clabject.new(
      self.model, self.potency-1, name, self)
    instantiations << obj
    return obj
  end

  def createClabjectLevel(supercls,levelNr)
    cls = Class.new(supercls)
    facet = cls
    for n in (0..model.depth)
      facet.clabject = self
      facet.tgtPtcy = n
      facet.srcPtcy = levelNr
      if(n < model.depth)
        facet = facet.singleton_class
      end
    end
    return cls
  end

  def to_s
    name
  end

  def getMembersN(n)
    if n == 0
      return [self]
    elsif n == 1
      return instantiations
    elsif n > 1
      tempAry = []
      ary = [self]
      for m in (1..n)
        ary.each do |cbj|
          tempAry = tempAry + cbj.instantiations
        end
        ary = tempAry
        tempAry = []
      end
      return ary
    end
  end

  def isMember(cbj)
    if self == cbj
      true
    elsif self.respond_to?(:instantiation_of) &&
        !self.instantiation_of.nil?
      self.instantiation_of.isMember(cbj)
    else
      false
    end
  end

  def isCompatibleWith(cbj)
    return self.isMember(cbj) || cbj.isMember(self)
  end

  def isMemberN(n, cbj)
    if n == 0 and self == cbj
      true
    elsif self.respond_to?(:instantiation_of) &&
        !self.instantiation_of.nil?
      self.instantiation_of.isMemberN(n-1, cbj)
    else
      false
    end
  end

  def method_missing(method, *args)
    if levels[0].respond_to?("#{method}", *args)
      levels[0].send("#{method}", *args)
    else
      raise NoMethodError.new("There is no method
          called #{method} here")
    end
  end

  def define(attribute, srcPtcy, tgtPtcy, value)
    for n in (1..(tgtPtcy+1))
      obj = facet(srcPtcy,n)
      #crete getter
      obj.class_eval("
        def #{attribute}
          if @#{attribute}
            @#{attribute}
          else
            inherited('#{attribute}')
          end
        end"
      )
      #create setter
      obj.class_eval("
        def #{attribute}=(val)
          if valueSettingAllowed(:#{attribute}, val)
            @#{attribute} = val
```

```
            end
          end"
        )
      end
      set(attribute, srcPtcy, tgtPtcy, value)
      return self
    end

    def checkDownwardCompatibility(attribute,
        srcPtcy, tgtPtcy, value)
      return true unless
          levels[srcPtcy].getMostSpecific(attribute)
      return true if value.nil?
      for potency in (0..srcPtcy)
        getMembersN(potency).each do |cbj|
          actMsVal = cbj.levels[srcPtcy-potency]
              .getMostSpecific(attribute)
          raise "#{value.name} is not compatible with
              #{actMsVal.name}" if
              !value.isCompatibleWith( actMsVal )
        end
      end
    end

    def set(attribute, srcPtcy, tgtPtcy, value,
        doDownwardCheck = true)
      checkDownwardCompatibility(attribute, srcPtcy,
          tgtPtcy, value) if doDownwardCheck
      facet(srcPtcy,tgtPtcy).send("#{attribute}=",
          value)
      return self
    end

    def get(attribute, srcPtcy, tgtPtcy)
      facet(srcPtcy,tgtPtcy).send(attribute)
    end

    def getValueSettingObject(attribute, srcPtcy,
        tgtPtcy)
      facet(srcPtcy,tgtPtcy)
        .getValueSettingObject(attribute.to_s.to_sym)
    end

    def getMethodDefiningClass(attribute, srcPtcy,
        tgtPtcy)
      facet(srcPtcy,tgtPtcy)
        .method("#{attribute.to_sym}").owner
    end

  end # end Clabject


  module ClabjectFacet

    attr_accessor(
      :clabject,
      :srcPtcy,
      :tgtPtcy
    )

    def to_s
     clabjectname = (clabject.respond_to?(:name))?
         clabject.name : clabject
     "#{clabjectname}^(#{srcPtcy},#{tgtPtcy})"
    end

    def parent
     superclass
    end

    def eigenclass
     singleton_class
    end

    def inherited(attribute)
     if parent.respond_to?(attribute.to_s.to_sym)
      parent.send(attribute.to_s.to_sym)
     else
      false
     end
    end

    #get value-defining object for attribute
    def getValueSettingObject(attribute)
     if instance_variable_get("@#{attribute.to_sym}")
      self
     else
      getInheritedValueSettingObject(attribute)
     end
    end

    def getInheritedValueSettingObject(attribute)
     if parent.respond_to?(attribute.to_s.to_sym)
      parent.getValueSettingObject(attribute.to_s.to_sym)
     else
      false
     end
    end

    def getMostSpecific(attribute)
     getMostSpecificN(attribute)[:val]
    end

    def getMostSpecificN(attribute)
     if respond_to?(attribute)
      val = self.send(attribute)
        if val
          return {:ptcy => 0, :val => val}
        elsif eigenclass.respond_to?(attribute)
          x = eigenclass.getMostSpecificN(attribute)
              #recursion
          if x[:val]
            return {:ptcy => x[:ptcy]+1, :val =>
                x[:val]}
          end
        end
      end
      return {:val => false}
    end

    def valueSettingAllowed(attribute, value)
      if value.kind_of?(Clabject)
        ms = getMostSpecificN(attribute)
        if ms[:val] && !value.isMemberN( ms[:ptcy],
            ms[:val] )
          raise "#{value.name} is not
              memberN(#{ms[:ptcy]}) of
              #{ms[:val].name}"
        end
        return true
      else
        raise "#{value} is no Clabject"
      end
    end

    def eigenclassN(n)
      obj = self
      for m in (1..n) # returns self if n=0
        obj = obj.singleton_class
      end
      return obj
    end

  end # end ClabjectFacet

  end # end module DDI
```