Application of advanced programming concepts in metamodelling

Henning Berg, Birger Møller-Pedersen, Stein Krogdahl
Department of Informatics
University of Oslo

Abstract

Programming languages provide users with a rich palette of advanced mechanisms. Languages for metamodelling, on the other hand, usually provide simple mechanisms for making class models, i.e. classes with relations like specialisation, composition and associations. A metamodel defines the abstract syntax (and to some degree semantics) of Domain Specific Languages (DSLs), and with the increased interest in making more and more complex DSLs there is a need for advanced mechanisms for metamodelling. This paper investigates what can be achieved by applying advanced language mechanisms to metamodelling, and how such mechanisms may increase the capabilities of DSL design.

1 Introduction

Metamodelling has received increased attention the last decade. One of the reasons for this is the popularity of *Domain-Specific Languages (DSLs)*, in which metamodelling plays a central role. Metamodelling is the process of defining the properties and features of a language [1]. This process results in a metamodel which captures language syntax and often also semantics. Simple UML-like class models are the most commonly used mechanism for describing metamodels. A class model consists of classes related by three types of relations: specialisation, composition and association. Each class of the model represents a language construct.

Kermeta aspect weaving [2] is an available mechanism for weaving of metamodels. Aspect weaving [3] is a powerful mechanism for separation of concerns, and it shows that concepts traditionally used in general-purpose programming and modelling potentially are valuable within metamodelling as well.

In this paper we address four general-purpose language mechanisms and discuss how these may improve the capabilities of metamodelling.

- **Generic types** are classes or interfaces that are parameterised with types. This mechanism allows diversity by describing type specific details using actual type parameters. Generic types are much used in programming. Common usages are generic algorithms and data structures. Generic types are already supported in *Eclipse Modeling Framework (EMF)* [4] and Kermeta. Consequently, we will restrict the discussion to possible usage scenarios of genericity.

- **Nested (inner) classes** are used to model concepts whose details are of no interest outside the enclosing class. It is common to distinguish between two types of nested

This paper was presented at the NIK-2011 conference; see http://www.nik.no/.

classes: static and non-static. Static nested classes can only access static properties of the enclosing class. Non-static nested classes, on the other hand, are class-valued attributes of enclosing class objects. This means that an object of the enclosing class is required for identifying a non-static inner class. In programming, non-static nested classes are commonly used to define local data structures, inside objects of the enclosing class.

- **Virtual classes** is a mechanism that supports specialisation of inner classes. A virtual class is a non-static inner class, and resembles a virtual method in that it can be redefined (by extension) in a subclass of the enclosing class. This allows a virtual class to appear differently depending on the class of the enclosing object [5, 6].

- **Package templates** is a mechanism that provides means for reuse and tailoring of predefined collections of classes. This is achieved using package template instantiation and class merging. Template instantiation allows changing and adding properties to classes from a template, while class merging provides a tool for combining two or more classes from different templates.

We will use a simple DSL named *Route Planning* to illustrate applications of the different mechanisms. For space reasons, this language is kept small. The domain of the DSL is modelling of routes for public transport. We will use Kermeta throughout this paper. Kermeta is an object-oriented metalanguage and framework for defining DSLs. It resembles Java and supports classes, specialisation and virtual operations. The main difference from Java is that classes in Kermeta define metaclasses, i.e. language constructs, and thus are used to express metamodels. Figure 1 gives the metamodel for the route planning language as defined in Kermeta.

The metamodel for the route planning language consists of nine metaclasses. A model in the language consists of one or more routes that are modelled using destinations interconnected by transportations. There are four types of transportations: bus, train, ship and flight. The dynamic semantics of the language is captured in the operations: estimateTraffic() and calculateStatistics(). For brevity, the operations' code is not listed, however, we indicate what the operations are supposed to do. estimateTraffic() identifies whether the number of arrivals at each destination is within the capacity of the destination. calculateStatistics() processes the routes and generates some statistics regarding what transportations that are most frequently used. This information is shown on the screen. Well-formedness rules, or static semantics, are deliberately not covered.

```
package routePlanning;

class RoutePlan {
  attribute routes : Route[0..*]
  attribute destinations : Destination[0..*]

  operation estimateTraffic() is do  ... end
  operation calculateStatistics() is do ... end
}
class Route {
  attribute description : String
  attribute transportation : Transportation[0..*]
  reference startDestination : Destination[1..1]
}
class Destination {
  attribute name : String
  attribute type : String
  attribute capacity : Integer
}
abstract class Transportation {
  attribute id : String
  attribute description : String
  attribute departure :Date[1..1]
  attribute arrival : Date[1..1]
  reference lastDestination : Destination[1..1]
  reference nextDestination : Destination[1..1]
}
class Bus inherits Transportation { ... }
class Train inherits Transportation { ... }
class Ship inherits Transportation { ... }
class Flight inherits Transportation { ... }
class Date {
  attribute day : Integer
  attribute month : Integer
  attribute year : Integer
  attribute hour : Integer
  attribute minute : Integer
}
```

Figure 1. *Metamodel for the route planning DSL in Kermeta notation*

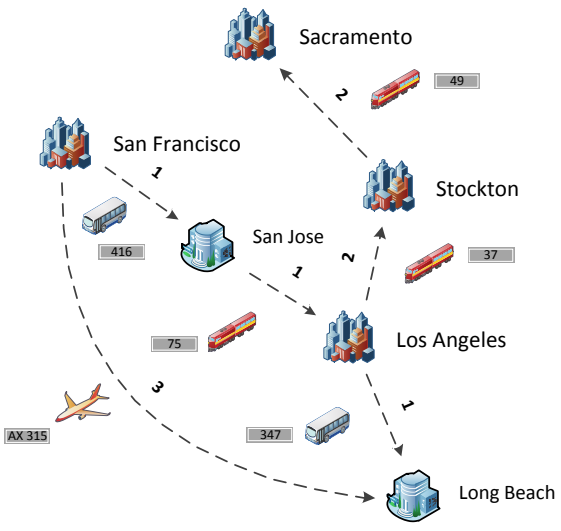

Figure 2. *An example route plan model*

We will not consider concrete syntax since this is not important here. However, to acquire a better understanding of the language we have included an example model in a graphical notation, see Figure 2. The route plan model comprises three routes denoted by '1', '2' and '3'. Route 1 goes from San Francisco to Long Beach via San Jose and Los Angeles. Route 2 goes from Los Angeles to Sacramento via Stockton. Finally, route 3 goes directly from San Francisco to Long Beach.

2 Generic types

Both EMF and Kermeta support generic types. An example illustrates how this mechanism can be used to provide alternatives when modelling.

```
package genericRoutePlanning;

class Destination { ... }
class DestinationWithHotel inherits Destination {
  attribute hotels : Hotel[1..*] }
class Hotel { ... }
class RoutePlan<D : Destination> {
  attribute destinations : D[0..*] }
...

var rp : RoutePlan<Destination> init RoutePlan<Destination>.new
var rph : RoutePlan<DestinationWithHotel> init
  RoutePlan<DestinationWithHotel>.new
```

Figure 3. *Use of generic types in Kermeta*

Figure 3 shows a subset of the route planning language. It includes two different destination constructs where one inherits from the other. RoutePlan is parameterised with the type variable D; bounded by Destination. This construction allows modelling of two kinds of route plans. A plan can either consist of simple destinations, or destinations that contain information about hotels. In other words, the user can choose what kind of language construct to instantiate in the model depending on requirements. Note that models containing instances of DestinationWithHotel also contain Hotel instances (due to containment). This illustrates how type parameters can be used to provide additional constructs for certain scenarios.

3 Nested classes

EMF and Kermeta support the use of packages in metamodels. A package can be understood both as a container for classes, but also as a mechanism that differentiates between namespaces. The notion of namespace in this context is, however, a subtle one, as any given class can refer to an arbitrary class in any other package within the metamodel[1]. Clearly, this violates commonly accepted static scoping rules and compromises true encapsulation of classes.

Class nesting is a well-known concept within programming. Nested classes have so far not been used in metamodels. One reason for this may be the seemingly added complexity. However, we argue that nesting of metamodel classes is both realisable and valuable.

Figure 4 gives the route planning language, expressed with nested classes. Each construct is represented by an inner class.

[1]Both EMF and Kermeta are based on the Ecore meta-metamodel.

```
class RoutePlanning {
  class RoutePlan {
    attribute routes : Route[0..*]
    attribute destinations : Destination[0..*]

    operation estimateTraffic() is do  ... end
    operation calculateStatistics() is do ... end
  }
  class Route { ... }
  class Destination { ... }
  abstract class Transportation { ... }
  class Bus inherits Transportation { ... }
  ...
}
```

Figure 4. *The route planning DSL implemented using nested classes*

Inner classes are considered to be non-static. Thus, accessing the inner classes is done via an object of the enclosing class. This object represents a specific language. Figure 5 illustrates this. We will not consider static nested classes in this paper.

```
var rp : RoutePlanning init RoutePlanning.new
var d : Destination init rp.Destination.new
```

Figure 5. *Instantiation of the route planning DSL and creation of a* Destination *object*

A model comprises both an instance of the enclosing class and instances of the inner classes. Several models can be created using the same instance of the enclosing class. This is contrary to packages which can not be instantiated. We have identified three usages of nested classes in metamodelling. These are:

- Customisation and extension using subtyping
- Generic metamodels by using type parameters on the enclosing class
- Language profiling based on properties of the enclosing class

Customisation and extension using subtyping

Subtyping is used to specialise a class. When using nesting of classes it is possible to subtype the enclosing class, and subsequently specialise one or more of the inner classes. Let us return to the route planning DSL and see how this language can be customised to create a language for travel planning, named *Travelling*. The language should capture information about whether transportations are available, the length of each route and information about accomodation and facilities. The latter two concepts should be represented by new constructs.

As illustrated, destinations can be reached using different means of transportation, like busses and flights. It is, however, not certain that a destination is reachable at a specific time due to closed roads, cancelled flights, and similar. A DSL for calculating reachable destinations would simplify route planning and make it easier to select the shortest route. The travel planning language should facilitate dynamic semantics that calculates whether a given route is reachable, the length of each route, and estimated travelling time. This information should be shown on the screen. The metamodel of the DSL is given in Figure 6.

```
class Travelling inherits RoutePlanning {
  class TravellingRoute inherits Route {
    operation reachable() is do ... end
    operation length() is do ... end
    operation travellingTime() is do ... end
  }
  class Transportation inherits RoutePlanning.Transportation {
    attribute length : Real
    attribute available : Boolean
  }
  class Destination inherits RoutePlanning.Destination {
    attribute hotels : Hotel[0..*]
    attribute facilities : Facility[0..*]
  }
  class Hotel {
    attribute name : String
    attribute address : String
    ...
  }
  class Facility {
    attribute type : String
    attribute description : String
    ...
  }
}
```

Figure 6. *Specialisation of RoutePlanning*

Figure 6 illustrates specialisation of RoutePlanning. The Route class from RoutePlanning is specialised by TravellingRoute. This class contains additional operations that capture the semantics for calculating whether the final destination of a route is reachable, the total length of a route and estimated travelling time. Transportation and Destination have been specialised with additional attributes. (The original class names are kept.) In addition, the classes Hotel and Facility have been added to the language definition.

There are two interesting properties of subtyping the enclosing class. Firstly, new language variations can easily be created, where only the relevant constructs are specialised. Secondly, the name of a specialised construct does not need to be changed as required when subtyping non-nested classes. This means that users will immediately be familiar with the new language. Specialising languages that are defined using class nesting yields language hierarchies. A language hierarchy provides a user with the ability to precisely select what language to use for a given context and situation. Still there is very little cost in maintaining the hierarchy since specialisation is used to relate the languages.

Generic metamodels by using type parameters on the enclosing class

We will here briefly illustrate a current research topic of generic metamodels. This notion builds on a combination of nested classes and generic types. Before going into details, we will discuss why generic metamodels are interesting.

The principal idea of generic metamodels is the ability to apply customisation by providing one or more "customisation" types. These types are metaclasses that contain the necessary details for adaption to a specific situation. There are a couple of different approaches to the definition of generic metamodels. With nested classes, a type parameter for the enclosing class will be a type parameter for the entire enclosed metamodel.

```
package genericRoutePlanning;

class Tbound { ... }
class Ebound { ... }

class RoutePlanning<T : Tbound, E : Ebound> {
  class RoutePlan { attribute t : T[1..*] }

  class Destination {
    attribute t : T[0..1]
    attribute e : E[1..1]
  ...
  }

  class Transportation {
    attribute e : E[1..*]
    ...
  }
  ...
}
```

Figure 7. *Metamodel with type parameters for enclosing class*

In Figure 7, the RoutePlanning class has been revised with two type parameters. The T parameter is used in both the RoutePlan and Destination classes, while the E parameter is used in Destination and Transportation. This allows creating route plan models where additional details are captured by the actual type arguments. In particular, DSLs that compute values are subject to change by choosing a different semantics. This can be as simple as increasing the precision of a formulaic constant, or substituting several statements of code. Common to these kinds of customisations is that the language's abstract syntax does not need to be changed. Only the dynamic semantics of one or more metaclasses has to be refined. Generic metamodels yields a convenient mechanism for specifying these kinds of customisations. Type parameters can also be used to add new constructs to a language. We will not go into further details in this paper.

Language profiling based on properties of the enclosing class

Until this point, the enclosing class has functioned merely as a grouping of the inner classes. An enclosing class may additionally contain attributes, references and operations. These properties can be accessed by the inner classes and used to specify profiling information. This makes it possible e.g. to choose among different semantics.

```
class Travelling inherits RoutePlanning {
  attribute maxSteps : Integer
  operation tooManySteps() : Boolean is do ... end

  class RouteSteps inherits Route {
    operation reachable() is do
      if tooManySteps() then
        // ignore the route if it has more transportations than maxSteps
        ...
      else
        ...
      end
    end
  }
  ...
}
```

Figure 8. *Language profiling using attribute of enclosing class*

Figure 8 illustrates how the value of the maxSteps attribute can be used as a criterion in reachable(). The value of this attribute is set when instantiating the travel planning language. A model contains the instance of the enclosing class, and thus, the value of maxSteps. Common operations can also be provided to the inner classes by defining them in the enclosing scope. This is illustrated by the tooManySteps() operation.

Preservation of conformance

An important aspect of metamodelling is *model conformance*[2]. This term dictates that a given model is a legal instance of a metamodel (conforms to). A model consists of instances of metaclasses in the metamodel. Syntactic changes to these classes compromise conformance.

There are situations where it is desirable to preserve conformance. Specialising a language's semantics does not necessarily impose structural changes, thus, there is in principle no reason why existing models, according to the more general language, should not still be compatible with the specialised language. Preserving conformance also makes it possible to interpret the same models differently according to context.

```
class Travelling inherits RoutePlanning {
  class TravellingRoute inherits Route {
    operation available() is do ... end
    operation length() is do ... end
    operation travellingTime() is do ... end
    ...
  }
  ...
}

class TravellingVariant inherits Travelling {
  class TravellingRoute inherits Travelling.TravellingRoute {
    operation available() is do /* new semantics */ end
    operation length() is do /* new semantics */ end
    operation travellingTime() is do /* new semantics */ end
    ...
  }
  ...
}
```

Figure 9. *Specialisation of Travelling*

We have seen how class nesting can be used to support creation of different, specialised languages. Specifically, a special class that only redefines operations will be syntactically identical to its superclass. Figure 9 illustrates this. The semantics of Route is specialised in a new language variation - TravellingVariant. The only changes to the language constructs are related to semantics. Consequently, a model conforming to the general language will also conform to the variation. This would allow a user to selectively decide how to interpret existing models by choosing the appropriate semantics.

Clearly, the ability to efficiently specify new semantics without breaking conformance is advantageous for highly algorithmic languages. DSLs for audio processing and graphics are examples of such languages, where mathematical approximations may improve with time.

An enclosing class can potentially address the question on how to realise model types in OMG's *Meta Object Facility (MOF)*, as discussed in [7].

[2]We use the term *model conformance* as a syntactic quality based on name and type equivalence. A model conforming to a metamodel X will also conform to a metamodel Y if Y has the same syntactic properties as X.

4 Virtual classes

Virtual classes are nested classes that can be redefined in a subclass of the enclosing class. A virtual class is bound at runtime[3]. We will see how this property can be utilised for defining new semantics for a language. Let us assume that there is a print() operation defined in each class of the route planning language. The purpose of these operations is to print a runtime log during model execution.

```
class RoutePlanning {
  virtual class RoutePlan {
    attribute destinations : Destination[0..*]

    operation print() is
    do
      ...
      destinations.each{ d | d.print() }
    end
    ...
  }
  virtual class Destination {
    operation print is do ... end
  }
  ...
}

class Travelling inherits RoutePlanning {
  class Destination {
    operation print() is do /* new semantics */ end
    ...
  }
}
```

Figure 10. *Definition of new semantics for the travel planning DSL*

A subset of RoutePlanning is given in Figure 10. It consists of the two classes RoutePlan and Destination. The print() operation of RoutePlan invokes print() of Destination. Both RoutePlan and Destination are virtual. Let us assume that we would like to redefine Destination including redefinition of print(). This can be achieved by subtyping RoutePlanning and redefine the Destination class.

The redefined semantics of Destination.print() is accessible by instantiating Travelling and the inner class Destination. Virtual classes are highly suitable when metaclasses are instantiated as part of the dynamic semantics. E.g., instantiations of Destination within the route planning language would now refer to the Destination class of Travelling.

5 Package templates

A complex domain concept is most likely to be represented by several strongly related classes. Package templates is a mechanism that supports reuse of a collection of classes by grouping these in a template. A template can be instantiated and tailored for a specific situation. Package templates also support merging of two or more classes from different templates. There are many examples of how package templates may improve programming [8]. Here, we will focus on how package templates can be used in metamodelling.

We have identified two usages of package templates in metamodelling:

- Customisation and extension of a language
- Merging of languages

[3]Runtime here refers to the execution of dynamic semantics.

Customisation and extension of a language

Here we will illustrate how package templates can be used to customise the route planning language. In this context, a template is used to encapsulate the metaclasses of the metamodel. This template can be instantiated to construct the actual metamodel in the scope where the instantiation is performed.

```
template RoutePlanning {
  class RoutePlan {
    attribute routes : Route[0..*]
    attribute destinations : Destination[0..*]

    operation estimateTraffic() is do  ... end
    operation calculateStatistics() is do ... end
  }
  class Route {
    attribute description : String
    attribute transportation : Transportation[0..*]
    reference startDestination : Destination[1..1]
  }
  class Destination { ... }
  abstract class Transportation { ... }
  class Bus inherits Transportation { ... }
  ...
}
```

Figure 11. *The route planning DSL as defined using a package template*

Figure 11 defines a template for the route planning language. A template can be instantiated in either a new template or a package. We will see how the above language can be adapted to create the language for travel planning. A template containing this language is given in Figure 12. It builds on the RoutePlanning template.

```
template Travelling {
  inst RoutePlanning with
    Route => TravellingRoute
      (description -> information)

  class TravellingRoute adds {
    operation available() is do ... end
    operation length() is do ... end
    operation travellingTime() is do ... end
  }
  class Transportation adds {
    attribute length : Real
    attribute available : Boolean
  }
  class Destination adds{ ... }
  class Hotel { ... }
  class Facility { ... }
}
```

Figure 12. *Definition of the Travelling template*

Travelling adapts the more general RoutePlanning template. The Travelling template has to be instantiated in order to use the travel planning language. This is shown in Figure 13. Here, the template Travelling is instantiated (using the keyword inst) within the package travelPlanning making the resulting metamodel available in this scope.

```
package travelPlanning;

inst Travelling
```

Figure 13. *Instantiation of the Travelling template*

Let us return to the definition of the Travelling template in Figure 12. Required language additions are specified using adds clauses. Package templates also allow for redefining operations in such clauses. A property of package templates, that is unique with regard to the other mechanisms, is the ability to rename both classes and class properties. In this case, the Route class of RoutePlanning has been renamed to TravellingRoute. The description attribute of Route has also been renamed to information.

Merging of languages

One of the basic features of package templates is class merge. Class merge has been indentified as a method for metamodel composition in [9].

```
template CityAndRoad{
  class City {
    attribute name : String
  }
  class Road { ... }
  ...
}

template Travelling {
  inst RoutePlanning with
    Route => TravellingRoute
      (description -> information)
  inst CityAndRoad with
    Road => TravellingRoute,
    City => Destination
      (name -> cityName)
  ...
}
```

Figure 14. *Merging of the two languages RoutePlanning and CityAndRoad*

In Figure 14, two languages are merged together. CityAndRoad contains several constructs, including City and Road. We assume that these classes represent natural unification points with the Destination and Route classes of RoutePlanning. Route and Road are merged into the class TravellingRoute, by giving both classes this name. Destination and City are combined to the class Destination as well. Destination and City have a conflicting property since both classes contain an attribute name. This is easily resolved by renaming the name attribute of City to cityName. This is a deep renaming, in the sense that entities referring to name in the CityAndRoad template will at compile-time refer to this attribute as cityName.

6 Related work

An approach for defining reusable metamodel components is described in [10]. This work illustrates how export and import interfaces can be used to compose metamodels. Composition has also been formalised using graph morphisms. The underlying motivation is the lack of a convenient manner in MOF for defining and maintaining Visual Domain-Specific Modelling Languages.

Using aspect-orientation in metamodelling is elaborated in [11]. Obliviousness is a term used to describe whether a metamodel is unaware of being extended by another metamodel.

This property is desirable. It has been shown that obliviousness can be achieved by considering relationships as first-class citizens using relationship aspects. Results include improved localisation and reuse of relationships and reduced coupling between metamodel implementations.

7 Conclusion

In this paper we have addressed four programming language mechanisms and how these may be applied in metamodelling. We have illustrated that these mechanisms can both increase usability and reuse of metamodels and thereby improve the design of Domain-Specific Languages.

References

[1] Tony Clark, Paul Sammut and James Willans. *Applied metamodelling (second edition).* Ceteva, 2008.

[2] Pierre-Alain Muller, Franck Fleurey and Jean-Marc Jézéquel. *Weaving Executability into Object-Oriented Meta-Languages.* Proceedings of MODELS 2005, 2005.

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-Oriented Programming.* Xerox Corporation, 1997.

[4] *Eclipse Modeling Framework.* http://www.eclipse.org/modeling/emf

[5] Ole Lehrmann Madsen and Birger Møller-Pedersen. *Virtual Classes - A powerful mechanism in object-oriented programming.* Proceedings of OOPSLA '89, ACM Press, 1989.

[6] Erik Ernst, Klaus Ostermann and William R. Cook. *A Virtual Class Calculus.* Proceedings of POPL'06, 2006.

[7] Artur Boronat and José Meseguer. *An Algebraic Semantics for MOF.* Journal of Formal Aspects of Computing, vol. 22, issue 3-4, 2010.

[8] Stein Krogdahl, Birger Møller-Pedersen and Fredrik Sørensen. *Exploring the use of Package Templates for flexible re-use of Collections of related Classes.* Journal of Object Technology, vol. 8, no. 7, 2005.

[9] Matthew Emerson and Janos Sztipanovits. *Techniques for Metamodel composition.* The 6th OOPSLA Workshop on Domain-Specific Modeling, 2006.

[10] Ingo Weisemöller and Andy Schürr. *Formal Definition of MOF 2.0 Metamodel Components and Composition.* Proceedings of MODELS 2008, 2008.

[11] A. M. Reina Quintero and J. Torres Valderrama. *Using Aspect-orientation Techniques to Improve Reuse of Metamodels.* Electronic Notes in Theoretical Computer Science 163, 2007.