Can Programming be Liberated from the Two-Level Style?

Multi-Level Programming with DeepJava

Thomas Kühne
Darmstadt University of Technology
kuehne@informatik.tu-darmstadt.de

Daniel Schreiber
Darmstadt University of Technology
schreiber@tk.informatik.tu-darmstadt.de

Abstract

Since the introduction of object-oriented programming few programming languages have attempted to provide programmers with more than objects and classes, i.e., more than two levels. Those that did, almost exclusively aimed at describing language properties—i.e., their metaclasses exert linguistic control on language concepts and mechanisms—often in order to make the language extensible. In terms of supporting logical domain classification levels, however, they are still limited to two levels.

In this paper we conservatively extend the object-oriented programming paradigm to feature an unbounded number of domain classification levels. We can therefore avoid the introduction of accidental complexity into programs caused by accommodating multiple domain levels within only two programming levels. We present a corresponding language design featuring "deep instantiation" and demonstrate its features with a running example. Finally, we outline the implementation of our compiler prototype and discuss the potentials of further developing our language design.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.2 [*Programming Languages*]: Language Classifications—Extensible languages

General Terms Languages, Design

Keywords domain metatypes, ontological metamodeling, deep characterization

1. Introduction

Among the factors responsible for the success of object-oriented programming languages is certainly their ability to extend the number of types available to programmers. In the days of FORTRAN, types such as 'Complex Number,' 'Fraction' or 'Book' were either directly supported or had to be emulated with the help of supported types. There was no direct support for handling the creation and management of multiple instances of a type and in combination with the lacking support for data abstraction this caused program structures that hardly reflected the corresponding domain types. In contrast, object-oriented languages allow the use of such types, as if they had been built-in into the language.

One way to analyze the reason for this advantage of object-oriented languages is to interpret user definable types as a feature of an extensible language. The natural consequence then is to look into other ways of making a language extensible. However, even though many attempts have been made to create languages and systems with programmer control over their concepts and semantics [29], none of them managed to attract the majority of programmers.

We argue that object-oriented languages are not successful because they feature a form of extensibility, but because they provide means for accurately reflecting the problem domain. They can thus be understood as allowing programmers to achieve a "direct mapping" [30] from problem domain to program structure. However, given that object-oriented programming languages support only two levels (classes and objects) for reflecting the logical domain classification levels, obviously the "direct mapping" quality cannot be maintained if the problem domain involves more than two classification levels.

In this paper, we use a running example that features three domain classification levels in order to demonstrate how workaround techniques—that accommodate multiple domain levels within two programming levels—introduce accidental complexity into programs (section 2). After recognizing the need for *deep characterization* and explaining our corresponding *deep instantiation* mechanism (section 3), we then show how multi-level support and deep instantiation can be integrated into a programming language and apply its features to our example (section 4). Subsequently, we briefly describe our prototype compiler (section 5) and conclude with a discussion of related work (section 6) and the potentials of further developing our language design (section 7).

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

2. Multi-Level Programming

In this section we introduce the concept of domain classification levels and demonstrate the accidental complexity introduced by mapping multiple domain levels into two programming levels.

2.1 Ontological Metamodeling

In order to understand the difference between metaclasses as used in reflective languages (targeting flexibility and extensibility [13]) and metaclasses used for mirroring a multi-level domain model in a program structure, it is important to make an explicit distinction between *linguistic* classification versus *ontological* classification [23].

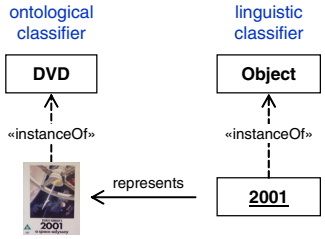

Figure 1. Ontological vs Linguistic Classification

Fig. 1 uses the UML [34] notation for objects and classes to show the relationships between a real product (the DVD "2001: A Space Odyssey"), the object representing it (object '2001') and their respective classifiers. Looking at '2001' from a language perspective results in its classification as an 'Object', since '2001' constitutes the usage of the UML concept 'Object'[1].

If one wants to understand what the type of '2001' is in terms of the domain where it occurs, one needs to examine the domain element it represents (here the DVD "2001: A Space Odyssey") and then determine the latter's domain type. In our example, the domain type turns out to be 'DVD'. The fact that in an object-oriented program object '2001' would have class 'DVD' reflects that the "instance of" relationship between objects and their classes is of ontological nature. In other words, the programming levels mirror the domain classification levels.

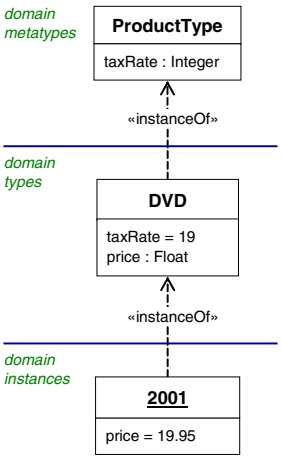

Figure 2. Three Domain Classification Levels

In a reflective language, 'DVD' would be further classified as 'Class', i.e., the classification would change from ontological (between objects and classes) to linguistic (between classes and metaclasses) (see also section 6). In our approach, we continue to use ontological classification.

Fig. 2 shows 'ProductType' as the domain type of 'DVD' assuming that an online store offers products of various types, such as 'Book', 'CD', 'DVD', etc. Note that 'ProductType' is not a generalization of 'DVD', i.e., not the latter's supertype, but its type. Fig. 3 illustrates the difference between a supertype (here 'Product') and a type (here 'ProductType') of 'DVD' using a 3D Venn diagram notation in which the third dimension is used to denote instantiation. Figures 3(a) & 3(b) graphically depict that the set theoretic interpretations of instantiation and specialization are the elementhood ($\in$) and the subset ($\subset$) relations respectively, showing types and their subtypes as sets and their subsets.

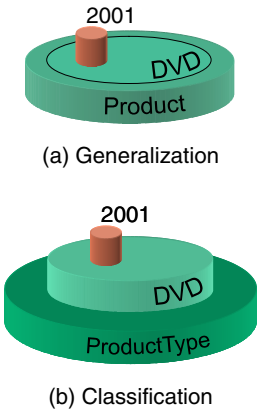

Figure 3. Super vs Meta

It is easy to distinguish between the two cases, where a concept X is either the type or supertype of a given concept T, by using the following litmus test: Take an instance I (here '2001') of T (here 'DVD') and check whether it can be regarded as an (indirect) instance of X. If yes (as is the case with 'Product'), then X is T's supertype. If not (as is the case with 'ProductType'), then X is T's type. In this case, using absolute terminology, X may then be regarded as a *metatype* (relative to I).

While one can easily come up with examples involving further levels[2] giving rise to meta-metatypes and so forth, three levels are already sufficient to demonstrate the difficulties entailed by the necessity to accommodate multiple domain levels within only two programming levels.

2.2 Workarounds

A number of design patterns such as the "Item Description" pattern [12] or the "Type Object" pattern [20] testify to the recurring need for representing structures like that of Fig. 2, involving multiple domain levels, with only two programming levels. As in our online store example, there is often the need to have a dynamic type level, i.e., be able to introduce new types such as 'HD DVD', 'Blu-ray', etc. at runtime. As a consequence, these types cannot be mapped to classes, but need to be represented at the object level. Fig. 4 shows the generic structure of the "Item Description" pattern in which 'a description' plays the role of a type for 'an item' at runtime.

[1] In recent versions "Object" has been replaced by "Instance Specification". Here, we are sticking to "Object" for brevity and clarity.

[2] For instance, element '2001' could be regarded as the type for all copies of this movie that may differ in packaging, condition when sold used, etc.

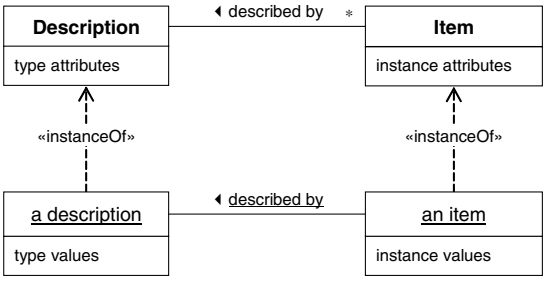

Figure 4. Item Description Pattern

Note that the 'type values' in Fig. 4 can be thought of as being shared by all item instances, thus avoiding their replication (see Fig. 5 for an example). Even if replication would be tolerable in terms of memory efficiency, the information kept in 'type values' (e.g., on flight routes) should be available even if no instances (e.g., actual flights) exist at a particular point in time [24].

Since many object-oriented languages feature class attributes (e.g., so-called "static variables" (sic) in JAVA), one may adequately represent the structure of Fig. 2 without resorting to the "Item Description" pattern, provided that there is no requirement to introduce new types at runtime. However, if the number of domain classification levels exceeds three, one has to use the "Item Description" pattern a number of times [20] to obtain the required classification depth, even if all type structures are completely static.

Together with the "Property Pattern" the "Type Object Pattern" [20] supports the creation of "Adaptive Object-Model" / "User Defined Product" frameworks , i.e., the runtime creation of types with a dynamic specification of their attributes. In such scenarios, the "Type Object" pattern is applied twice in a "type square" [37]. The "Dynamic Template Pattern" [27] addresses the potential need for inheritance between represented types.

2.3 Accidental Complexity

Given a two-level limitation, the above described design patterns provide welcome guidance as to how one may work around the limitation. However, ideally one should not be forced to know the patterns and live with their implementation overheads. Fig. 5 depicts the application of the "Item Description" pattern to our example[3]. More explicitly than Fig. 4, it shows that two domain levels are squeezed into one programming level, causing considerably overhead:

- the programmer has to deal with two forms of instantiation, i.e., the built-in language instantiation and the "instance of" 'isOfType'/'isOfType' relationships between language types and language instances respectively.
- the 'isOfType' between domain instances and domain types must be emulated by the programmer. There is no language support for making sure that domain instances typecheck against their domain types.
- inheritance between types has to be emulated.
- late binding of methods calls has to be emulated.
- the programmer has to deal with pattern artifacts which do not represent any element in the domain. While 'Product', in principle, *could* be a useful generalization in the domain, it may not be required at all. In Fig. 5, it is mandatory and its primary purpose is to be a class for objects that represent domain instances, since the latter's domain types are just objects themselves which cannot instantiate or control other objects.

[3] German taxation features two tax rates, depending on the product type.

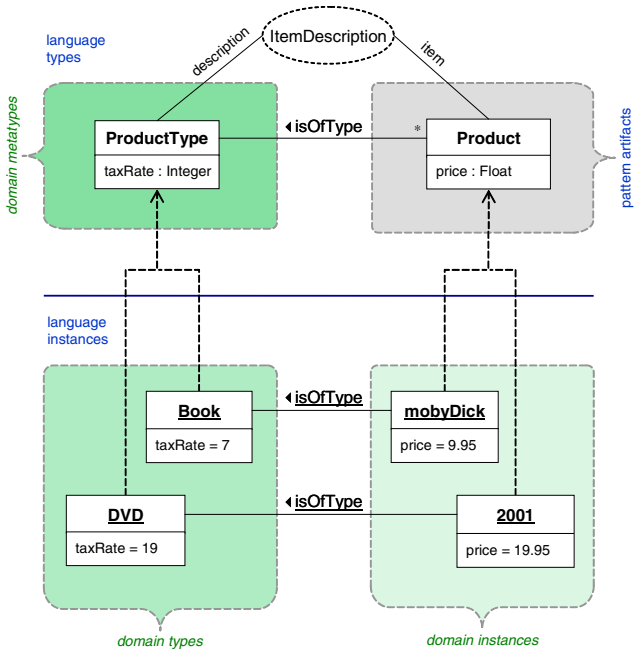

Figure 5. Squeezing Three Levels into Two

The above list clearly enumerates what has been coined *accidental complexity* by Brooks [8]. It is complexity that is not induced by the complexity of the problem domain, but artificially introduced by limitations of the solution paradigm. The solution shown in Fig. 5 is a good example for a design that does not maintain a *direct mapping* (one of Meyer's modularity criteria [30]) to the problem domain that induced it (see Fig. 2). Many of the built-in concepts of object-oriented programming languages have to be emulated to create a solution that is less efficient, more error-prone, and most importantly, less easy to understand and maintain [4].

The preceding discussion should have made it clear that language support for an unbounded number of programming levels—in order to be able to mirror any number of domain classification levels—would be very desirable. Fortunately, one can expect programmers with proficiency in the use of the object-oriented paradigm to easily adapt to a corresponding language. After all, one can always look at an n-level

classification hierarchy with a (sliding) two-level window, treating the upper level as classes and the lower level as objects. With such a relative perspective, the middle level of Fig. 2 contains objects, created and controlled by the classes above it. The fact that said elements are classes and metaclasses respectively in an absolute sense, can be safely ignored in understanding their relative relationships.

However, there is one issue that distinguishes an n-level classification hierarchy from a simple stacking of two-level building blocks, which concerns the notion of instantiation.

3. Deep Instantiation

In a two-level classification hierarchy the type level only describes the instance level directly below it. As soon as a third level is added, the question arises whether elements in the top level may influence elements in the bottom level, requiring control extending over two level boundaries, as opposed to just one.

Indeed, our online store example already motivates the desire of being able to control elements across more than one level boundary. Consider Fig. 2 and imagine the introduction of a new type, such as 'Book'. The new type is guaranteed to have a 'taxRate' property but whether or not it declares a 'price' attribute is left to the discretion of the programmer or code introducing the type. Yet, online store code dealing with objects that represent products in a generic manner should be guaranteed to always find a 'price' property.

We refer to any mechanism allowing the presence of the 'price' property to be specified from two (or more) levels above as achieving *deep characterization*. In contrast, traditional object-oriented classification/instantation semantics only achieves *shallow characterization*. Deep characterization is required whenever one level should not only govern the well-formedness (i.e., allowed relationships between and required presence of properties within elements) of the immediate level below, but also make sure that the latter prescribes some well-formedness rules for the level below it (and possibly further on) as well. One well-known application for deep characterization are so-called process metamodels, used in the definition of software development methodologies [10]. The designer of a process metamodel (at level 2) wishes to constrain the creation of process models (level 1), but in addition also needs to make sure that process enactments (level 0) obey certain rules. For instance, the process metamodel may want to enforce that task enactments are guaranteed to have a 'duration' property. Gonzalez-Perez and Henderson-Sellers achieve deep characterization by employing *powertypes* [10]. In this section, we are discussing an alternative mechanism, called *deep instantiation*[4], which was originally developed for addressing issues in the UML infrastructure [3].

[4] We compare deep instantiation to powertypes in section 6.

3.1 Clabjects

Understanding how, using deep instantiation, one may influence elements beyond one level boundary is easiest if one first introduces the notion of a *clabject*, an element that is both *cla*ss and ob*ject* at the same time [2]. Consider the elements in the middle level of Figures 2 & 6. They are objects (with corresponding 'taxRate' properties) with respect to the level above them and classes (with corresponding 'price' attributes) with respect to the level below them. In fact, every element in a classification hierarchy has both an instance facet (w.r.t. its object role) and a type facet (w.r.t. its class role), with the exception of the elements at the bottommost and topmost levels. Hence, it makes sense to abandon the distinction between objects, classes, metaclasses, etc. and simply regard them as clabjects which only differ in the level they occupy. The same unification can be applied to (object) properties and (class) attributes to yield the notion of a *field*.

In the following we will continue to use terms like "type", "class", "subclassing", "attribute", etc., but note that in a multi-level environment populated with clabjects the above terms refer to roles and relationships that may occur at any level. The term "class", for instance, thus should be read as "the type facet of a clabject" without any determination of the absolute level it may reside on.

3.2 Potency

Whether a field corresponds to a traditional property or attribute is governed by its associated *potency* value. A field with a potency value of 0 indicates an object property which has no influence on potential instances of its owning clabject (see the 'taxRate' property of the L_1 elements in Fig. 6). A field with a potency value of 1 indicates an attribute, which specifies the presence of a corresponding object property for all instances of its owning clabject (see the 'taxRate' attribute of 'ProductType' in Fig. 6).

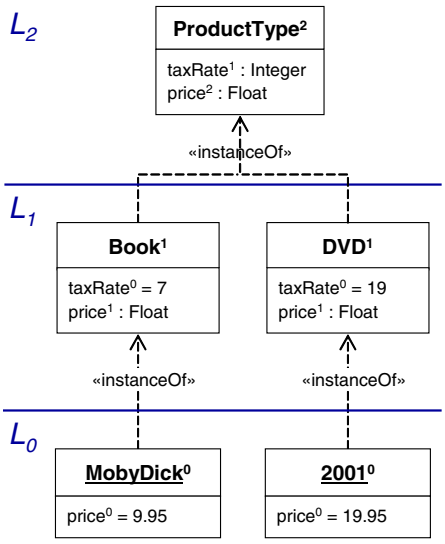

Figure 6. Deep Instantiation

Deep instantiation conservatively extends traditional shallow instantiation by letting potency values range over integer values $\{0, \ldots, n\}$ where n corresponds to the maximum characterization depth required. Fig. 6 shows the same scenario as Fig. 5, however, this time making use of three programming levels and a potency value of 2 for field 'price' at L_2, in order to guarantee the presence of a 'price' property for objects representing products at L_0.

Note that not only fields have potency values but clabjects as well. As a result, a programmer can control the instantiation depth for a given clabject and, for instance, choose a potency value of 0 in order to specify an abstract class (i.e., a clabject at L_1), whose fields may have potency values > 0. At what level a clabject resides can be specified explicitly but this is optional since we are assuming a discipline according to which only "instance of" relationships can cross level boundaries and must not cross more than one level boundary. With this assumption, "classification" is a *level-respecting* relationship which implies that no ambiguity w.r.t. level membership may arise [23].

3.3 Instantiation Semantics

Given the two unified concepts clabject and field it becomes very easy to informally define the semantics of instantiation which intuitively simply amounts to creating a copy of the element (that is required to have level and potency values > 0), considering only fields with potency values > 0, and decreasing all level and potency values by one. We do not present a formal semantics here since its verbosity does not match the simple intuition behind deep instantiation.

4. Programming Language Integration

In this section we show how an unbounded number of programming levels and deep instantiation can be supported by extending JAVA to DEEPJAVA. DEEPJAVA is a conservative superset of JAVA, i.e., every correct JAVA program is also a correct DEEPJAVA program.

4.1 Clabjects and Potency

Since DEEPJAVA naturally supports class properties by defining them through metaclass attributes, there is no need for a special concept like "static variables" (JAVA) or "class variables" (SMALLTALK (see also section 6)).

Listing 1 shows how metaclass 'ProductType' from Fig. 6 is specified in DEEPJAVA. Note that we use pretty printing for potency values. Using a standard text editor, potency values are entered and appear as in "`ProductType^2`".

Compared to Fig. 6, we have slightly changed metaclass 'ProductType' to use a 'netPrice' and let a 'price' method compute the retail price by referring to the corresponding tax rate for a particular product. Method 'price^2' hence demonstrates how product instances may access the tax rate of their types. When reading the code for method 'price^2' one must realize that its potency value is 2, i.e., it becomes an ordinary

```
public class ProductType^2
       extends ProductCategory^2 {

  public ProductType(String categoryName,
                     int categoryCount,
                     int taxRate) {
    super(categoryName, categoryCount);
    taxRate(taxRate);
  }

         int    taxRate;
  public void   taxRate(int t)    { taxRate = t; }
  public int    taxRate ()        { return taxRate; }

  private float netPrice^2;
  public void   price(float p)^2  { netPrice = p; }
  public float  price()^2 { return netPrice *
                            (1 + type.taxRate / 100f);
  }
}
```

Listing 1. Metaclass Definition

method in instances of 'ProductType', such as 'DVD'[5]. One must therefore understand its execution from the perspective of products, e.g., instances of 'DVD'. It is then obvious that, in the body of 'price^2', we need to navigate to the 'type' of a product in order to obtain its 'taxRate'.

Note that in our code example 'ProductType' has a super clabject 'ProductCategory', which we are going to use in the following (see Fig. 8 for a complete overview over all static and some dynamic online store elements).

4.2 Dynamic Class Creation

Our online store distinguishes between various product categories, one of them being software items. Listing 2 shows in line 1 how to create the new class 'SoftwareItem' with initial values for its category name and the number of sold items. The same 'ProductCategory' constructor was already used in line 7 of listing 1.

DEEPJAVA class properties, such as 'taxRate', correspond to SMALLTALK class instance variables [17], i.e., each instance of a class has its own set of class properties. In the context of our online store example a useful application for them is to store the number of sold items in product types (such as 'DVD', 'Book', etc.) respectively and aggregate these individual subcategory counts in respective supertypes (e.g., 'SoftwareItem'), and so forth. Listing 3 shows how method 'soldOne' (from 'ProductType') not only updates the count for subcategories (product types, such as 'DVD') but also for categories (such as 'SoftwareItem').

[5] Guaranteeing that 'DVD' instances understand the message 'price'.

```
1  ProductCategory{"Software Items", 333} SoftwareItem extends Product⁰;          // create SoftwareItem
2
3  DigitalMedium{DVD_Player, "DVDs", 222, 19} DVD extends SoftwareItem {          // create DVD
4    "public String toString () { return name() + \" (\" + type.categoryName() + \")\" +
5                                    (promoProduct() == null ? \"\" : \" -> \"+promoProduct());}"
6  };
```

Listing 2. Dynamic Type Creation

```
public void soldOne() {
  categorySoldCount++;
  superType().categorySoldCount++;
}
```

Listing 3. Making Use of Unshared Class Properties

Note that attribute 'categorySoldCount' is defined in 'ProductCategory' so that all categories (such as 'SoftwareItem') have a counter of items sold. However, only product types (such as 'DVD') have instances that can be sold, hence, method 'soldOne' is defined in 'ProductType'.

By using the option to specify a superclass for the new class by stating "**extends** Product⁰", in line 1 of listing 2, we make the freshly created class a subclass of the preexisting class 'Product'. The latter simply defines a name attribute and corresponding access methods. Intriguingly, one can already tell by "Product⁰" in line 1 that 'Product' is abstract (has potency value 0), since DEEPJAVA optionally allows the declaration of potency values in use positions (as opposed to defining positions) of clabject identifers.

Instead of using 'Product' as a superclass, we could have also defined 'name' with potency value 2 at 'ProductCategory'—achieving the same effect for 'SoftwareItem' instances—but we specifically introduced 'Product' to demonstrate DEEPJAVA's ability to combine dynamic clabjects with statically existing superclabjects.

If in a type creation (such as the one in line 1) the intersection between the superclabject features and the features defined through the clabject type by virtue of deep instantiation is not empty, the superclass definitions take precedence, thus allowing programmers to override generic feature definitions for specific cases. In analogy to subclassing, it is left to the discretion of the programmer to provide only compatible—in the sense of subtyping [26]—redefinitions.

Lines 3–6 of listing 2 create a new class 'DVD' as a subclass of 'SoftwareItem' and Listing 4 shows how this new class can be used to create 'DVD' instances.

```
7  DVD aso = new DVD();
8  aso.price(19.95f / (1 + aso.type().taxRate() / 100f));
9  aso.name("2001: A Space Odyssey");
10 aso.promoProduct(haChi_779);
```

Listing 4. Using a Dynamic Type

4.3 Dynamic Feature Creation

Lines 4–5 of listing 2 demonstrate DEEPJAVA's ability to equip dynamically created types with any number of new attributes or methods defined in strings (which need not be based on string constants). This language feature allows the dynamic introduction of new clabjects with features that were not anticipated at the time their types and supertypes were defined. Obviously, this level of flexibility does not come with typesafety. In our DEEPJAVA implementation (see section 5) we check the validity (syntax and well-formed access to features) at runtime and this may result in an exception (e.g., 'NoSuchMethodException') being thrown. Furthermore, if the new features do not override existing features then access to them cannot be typesafe, since there is no static definition that may guarantee their presence. The 'DVD' example of listing 2 redefines a statically known method which can be used in a typesafe manner, but assuming we had additionally provided features regarding the director of DVD movies in lines 4–5 of listing 2 then listing 5 illustrates how one can make use of one of them. Note the use of a '#' instead of a dot before the method name (supplied as a string value) in order to allow access to a statically unknown feature and to signal the potential failure of such an attempt.

```
11 try {
12   System.out.println("Director of " + aso.name() +
13                      " is " + aso#("director")());
14 } catch (Exception e) { e.printStackTrace(); }
```

Listing 5. Invocation of a Statically Unknown Method

4.4 Typeparameters

In line 3 of listing 2 we use the type 'DigitalMedium' (see listing 6) which subclasses 'Medium', which in turn subclasses 'ProductType' (see Fig. 8). Note that the constructor for digital media features an additional argument compared to that of product types/categories. By passing the clabject 'DVD_Player' as a parameter to the constructor that creates 'DVD' in line 3, we demonstrate DEEPJAVA's ability to not only use types as values but also use these values in type positions.

In listing 6 we can see that said first parameter of 'DigitalMedium''s constructor has type 'HarwdwareType' (another subclass of 'ProductType'). The idea is to be able

```
public class DigitalMedium[2] extends Medium[0] {

  public DigitalMedium(HardwareType HT,
                       String categoryName,
                       int categoryCount,
                       int taxRate) {
    super(categoryName, categoryCount, taxRate);
    CrossPromoType = HT;
  }

  final HardwareType CrossPromoType;

  CrossPromoType recProd[2];

  public void promoProduct(CrossPromoType rp)[2] {
    recProd = rp;
  }

  public CrossPromoType promoProduct()[2] {
    return recProd;
  }
}
```

Listing 6. Genericity through Typeparameters

to link digital media *types* (such as 'DVD') to hardware item *types* (such as 'DVD_Player') for cross promotion purposes. This is why 'DVD_Player' is passed to the constructor that creates 'DVD'. The type 'CD' would link to type 'CD_Player', etc. Note, however, that the ultimate motivation of the online store for doing this is to link up digital media instances to hardware instances. In line 10 of listing 4 the 'DVD' instance '2001' is linked to a corresponding 'DVD_Player' instance (referenced by 'haChi_779'). This call to 'promoProduct' only accepts arguments of type 'DVD_Player' since the argument type 'CrossPromoType' has been instantiated to 'DVD_Player' when class 'DVD' was created. This is the reason for declaring 'CrossPromoType' to be final in listing 6, i.e., make it immutable after it has received its initial value by the constructor. Otherwise, instances of 'DigitalMedium' (such as 'DVD') would have volatile type facets, in other words, clients could not assume stable feature types.

In essence, the parameterization of 'DVD' to only accept cross promotion items of type 'HardwareItem' corresponds to supplying a type argument to a generic class. In other words, DEEPJAVA's ability to use types as values and use these values in type positions, creates an alternativ approach to genericity. With types (at all levels and all potencies) as first-class entities in place, there is no need to add another genericity concept to JAVA. In fact, the approach followed by DEEPJAVA leads to more powerful ways of using type parameterization than afforded by JAVA-genericity including wildcards.

4.5 Static Typing

A comprehensive discussion of the type system required for the opportunities opened up by DEEPJAVA is out of scope for this paper. However, Sections 6 & 7 elaborate on the nature of the required type system to some extent. Our compiler prototype (see section 5) currently implements a pragmatic mixture between rejecting a large class of type incorrect programs and admitting some which may or may not produce runtime errors. Any type-related runtime error is dealt with by throwing a corresponding exception, though. As a result, the currently supported version of DEEPJAVA is not "unsafe" in the sense that a type-related error may go unnoticed or corrupt program execution.

We also chose to allow some features (such as the ability to call statically unknown methods) in order to provide the associated flexibility in favor of an approach that attempts to discover all type-related errors at compile time. Note that JAVA itself follows a similar philosophy, e.g., by allowing downcasts which may fail.

In spite of all the dynamics introduced to types by DEEPJAVA, thanks to deep instantiation and the possibility for dynamic types to refer to statically known superclasses, it is possibly to statically typecheck code as generic and flexible as the one in listing 7. The following sections explain listing 7 and discuss a compatibility issue arising in the presence of multi-level class hierarchies.

4.5.1 Abstract Type Declarations

The heading of this section is to be read as "abstract type-declarations", since we are introducing a way to declare a variable type in DEEPJAVA that enables very generic code. Line 15 of listing 7 declares an array whose element's types are only known to be an instance of 'ProductType'. Line 21 of listing 7 uses the same declaration—in which "@" is to be read as "value of"—for a single variable. These declarations are abstract in the sense that the concrete type of the referenced elements need not be known.

A similar argument could be made for using a supertype (such as 'Product') of all the types one wishes to capture. However, note that instances of 'ProductType' may be part of a number of unrelated inheritance hierarchies. In contrast to the supertype declaration using 'Product', the abstract type declarations in listing 7 do not require these inheritance hierarchies to have a common root. An abstract DEEPJAVA type declaration is hence more generic, i.e., admits more types, than traditional declarations using supertypes.

Listing 7 iterates[6] through a number of product type instances and prints them (implicitly using their 'toString' method) and their prices to the standard output. Note that this will cause the dynamically redefined 'toString' method (see lines 4–5 of listing 2) to be invoked in the case of 'aso'

[6] We cannot use JAVA's "for each" syntax since our compiler currently only accepts Java 1.4 syntax.

```
@ProductType products[] = {mobyDick, aso,
                           haChi_779};

System.out.println("In stock:");

for (int i=0; i<products.length; i++) {
   @ProductType p = products[i];
   System.out.println(p + " for " + p.price());
}
```

Listing 7. Using Abstract Type Declarations

which will then include a reference to the cross promoted 'haChi_779' player.

By replacing '@ProductType' with 'ProductCategory' and populating the array with 'SoftwareItem', 'HardwareItem', 'DVD', 'DVD_Player', etc., very similar code could be used to flexibly provide information about the number of sold items in various (sub-)categories.

4.5.2 Metaclass Compatibility

In a multi-level class hierarchy where intra-level subclassing may occur at more than one level and such generalization hierarchies may be connected by inter-level "instance of" relationships, two *Upward Compatibility* and *Downward Compatibility* rules must be obeyed in order to avoid runtime errors [6]. In the following we rephrase these rules and explain how they are implemented in DEEPJAVA.

> *Upward Compatibility:* If a clabject C of type T is subclassed by another clabject C' (with type T'), i.e, $C' < C$ then T' must offer at least the features that are offered by T. This rule is necessary to ensure that all methods of C inherited by C' (whose method bodies may navigate to T and use its features) are guaranteed to also work in C' (in which the same type-navigation leads to T').

Note that the "upward compatibility" rule is trivially fulfilled if a class and its subclass have the same type. In particular, this is the case when they do not use any specific domain type, but just a generic DEEPJAVA type such as 'class_1', ..., 'class_n', which is available for every level n.

> *Downward Compatibility:* If a clabject T has an instance I (i.e., is not abstract) and is subclassed by another clabject T', i.e., $T' < T$, then an instance I' of T' must offer at least the features that T expects from I. This rule is necessary to ensure that all methods of T (which may create T-instances in a generic fashion and then call their methods) are guaranteed to also work in T' (whose instances must then be able to perform the same method calls).

With the usual definition $C < C$ (i.e., a class qualifies as its own subclass), one can address the above compatibility issues, simply by demanding $C' < C \implies \text{type}(C') < \text{type}(C)$, i.e., subclassing between two classes implies subclassing between their types ($\rightarrow$ upward compatibility) and $\text{type}(C') < \text{type}(C) \implies C' < C$ subclassing between two classes implies subclassing between their respective instances ($\rightarrow$ downward compatibility). Of course, the latter rule only applies for C/C' with level values > 0. In combination, these two rules enforce parallel subclassing hierarchies, i.e., the approach followed in Smalltalk [17] (see also section 6).

5. Prototype Implementation

The following sections briefly describe our approach to creating a prototype compiler for DEEPJAVA.

5.1 Compiler

DEEPJAVA syntax, typechecking rules, and semantics are defined by using the *Polyglot* compiler front end which makes it very easy to define language extensions for JAVA [32]. It provides a JAVA 1.4 grammar and an extensible LALR parser generator. In contrast to other parser generators (e.g., *JavaCC*), *Polyglot* also supports the adaptation of typechecking—the DEEPJAVA type system is a true extension of the JAVA type system—and code generation.

Compiling a DEEPJAVA program is a two-phase process. First, DEEPJAVA-sources are typechecked and transformed into JAVA-sources which implement DEEPJAVA semantics. Second, the generated JAVA-code is compiled to JAVA bytecode by the regular `javac` compiler.

Our decision to design DEEPJAVA as a conservative superset of JAVA implies that we could not always realize our first choice regarding additional syntax, because we needed to avoid shift/reduce conflicts for the overall grammar that the *Polyglot* parser could not handle. For instance, potency values for methods have to be specified after the parameter list, instead of after the method name, and we would have preferred to use round or square parentheses for constructor parameter lists. Yet, maintaining the compatibility with the standard JAVA grammar and being able to draw on the *Polyglot* framework made it easy to tolerate these minor syntactical deficiencies.

Our *Polyglot* front end generates JAVA code that implements DEEPJAVA semantics, but it does not support the dynamic creation of classes and their features. Even though JAVA provides a class loader which can dynamically load classes for execution, it is per se not possible to define classes at runtime. However, the class loader may be used to incorporate bytecode that has been dynamically generated at runtime. This is the main idea behind the *Javassist* class library [11] which enables the definition and manipulation of JAVA classes at runtime without requiring the programmer to deal with bytecode directly. We therefore use *Javassist* as a runtime support for the output of the *Polyglot* compiler in order to enable the creation of classes and their features.

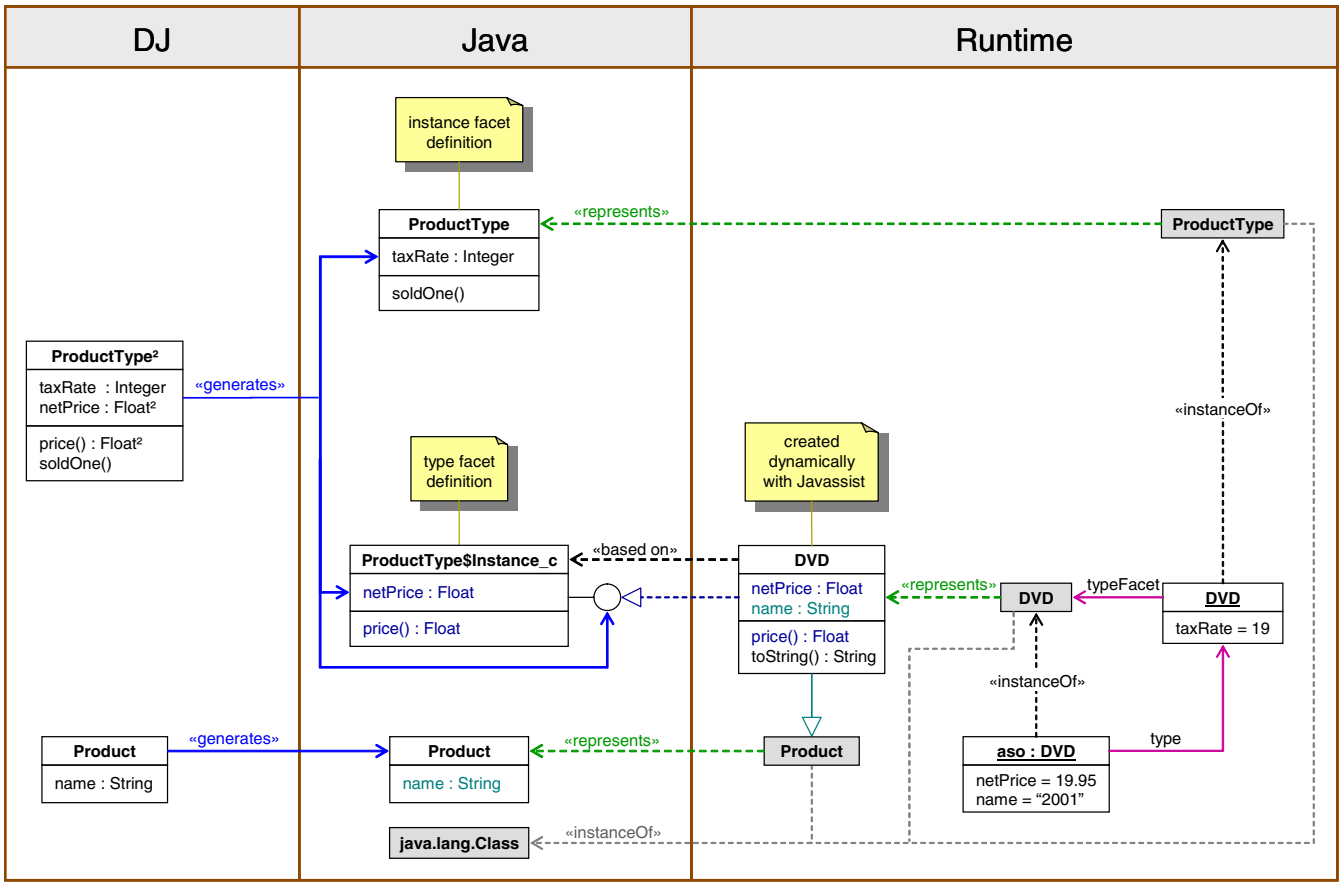

Figure 7. Compilation Approach

5.2 Compilation Approach

Since the focus of this paper is on demonstrating the utility of DEEPJAVA, as opposed to detailing the inner mechanics of its prototype compiler, we only briefly allude to the general compilation approach in the following.

For every type facet induced by a DEEPJAVA class definition, we generate a corresponding JAVA class. For example, 'ProductType' from listing 1 induces two facet definitions, one instance facet definition (featuring 'taxRate') and one type facet definition (featuring 'price'). We consequently generate one JAVA class definition 'ProductType' for the instance facet and another, 'ProductType$Instance_c' for the type facet (see Fig. 7). Had 'ProductType' declared features with potency values of 3, we would have obtained a third definition 'ProductType$Instance$Instance_c', defining the potency 3 features, and so forth.

Each generated facet definition exclusively defines the *instance facet* of its instances. For example, 'ProductType' defines that its instances have 'taxRate' features. Each clabject that may instantiate further clabjects (e.g., 'DVD' in Fig. 7) is provided with a reference to the *type facet* of its instances. This type facet, again describes the instance facet of its instances, here the fact that instances of 'ProductType' instances have 'price' features. The DEEPJAVA compiler automatically generates respective 'typeFacet' attributes, 'setTypeFacet' methods, and 'newInstance' methods, in order to support the creation of clabjects behind the scenes.

Note that dynamically created classes, such as 'DVD', are also associated with corresponding JAVA runtime objects (here the element 'DVD' with the grey background) that represent classes. Such class objects (instances of 'java.lang.Class') are used for instance creation and can be accessed via the JAVA reflection API. Hence, 'aso' is an instance of 'DVD' but note that its DEEPJAVA type is 'DVD' so that type features such as 'taxRate' can be accessed.

The leftmost 'DVD' element is dynamically composed from a number of sources, overriding of identical features taking place in the following order:

- it implements its type facet definition interface. Fig. 7 shows the latter using the UML lollipop notation. As we can only implement an interface (single inheritance may be required for another class, see below), the features to be implemented must be copied from the corresponding source. Hence the 'based on' relationship from 'DVD' to 'ProductType$Instance_c'.

- it inherits features from a static JAVA class 'Product', due to the "**extends** SoftwareItem" line 3 of listing 2. We have excluded 'SoftwareItem' from Fig. 7 for brevity since it does not contribute any type facet features.
- the 'toString()' method is added last, due to lines 4–5 of listing 2.

Interfaces for type facet definitions are required for yet another reason: Dynamically created DEEPJAVA classes (such as 'DVD') cannot be associated with any known specific type. Variables referring to instances of such classes are therefore typed with the interface of the type facet definition. In particular for abstract type declarations, such as '@ProductType', said interface is required since it may be implemented by a multitude of classes which need not have a common concrete root superclass.

The above described use of interfaces implies that DEEPJAVA programs may not directly access attributes with potency values ≥ 2, since JAVA interfaces do not permit the declaration of attributes. However, we perceive this limitation as a built-in feature to remind programmers of the loss in encapsulation implied by using public attributes instead of access methods to private attributes.

6. Related Work

Work on programming languages abounds with attempts to equip programmers with meta-level descriptions (and possibly further classification levels).

The meta-object protocol [21] for CLOS is probably the prototypical example for the approach to use meta-level descriptions in order to adjust the language to better suit the needs of its users. In contrast to DEEPJAVA's classification levels, however, the CLOS meta level interface is not intended to provide a direct mapping from domain classification levels to programming levels, but to give programmers control over basic programming mechanisms, such as slot access, method dispatch, and multiple inheritance. This approach can therefore be used to support programmer-defined additions such as asynchronous communication, exceptions, pre/post conditions, etc. This intention is not specific to CLOS but characteristic of a whole range of meta-level description approaches:

> One of the key ideas is to provide at the metalevel generic metacomponents describing standard OO language features and their decomposition into basic facets [13].

Some of these approaches also aim at describing class properties, but from a *linguistic* perspective, addressing properties such as being abstract, being final, supporting multiple inheritance, etc. DEEPJAVA class properties, in contrast, are purely *ontologically* motivated, i.e. derive themselves from the domain (see, e.g., our 'taxRate' feature).

The most recent addition to the linguistic-control camp is Aspect-Oriented-Programming [22] whose aspect definitions can be understood as second order predicates ranging over programs. AOP applications typically do not address domain-related concepts but system-oriented ones, such as the famous logging aspect. There is also work on so-called "early aspects", addressing cross-cutting concerns at the requirements level, but in any event, aspects cannot help to adequately accomodate an n-level domain classification.

SMALLTALK [17] features metaclasses which are ontologically motivated, i.e., represent class level properties induced by the domain, but supports them in a very restricted way only. Metaclasses are singletons (have only one instance), and classes have exactly one anonymous metaclass which can only be reached by sending a 'class' message to a class. As a result, SMALLTALK metaclasses enable programmers to deal with classes in the same way as with objects and introduce the ability to define class level properties (analog to the static features of JAVA classes), but do not offer any further advantages. It is, for instance, not possible for two classes to have the same metaclass or build hierarchies with a classification depth > 2, since SMALLTALK metaclasses just add an instance facet to the class level and there is no way to define meta-metaclasses.

The OBJVLISP reflective architecture [14] introduces the analog of the concept of a clabject, but again without an ontological motivation, but to create a more uniform SMALLTALK classification hierarchy with the intention to control inheritance, internal representation of objects, caching techniques, etc. The same linguistic control motivation is characteristic of other attempts to extend SMALLTALK, such as CLASSTALK [25].

NEOCLASSTALK [6] introduces an approach that ensures upward and downward compatibility (see section 4.5.1) without causing all the properties of a metaclass to be inherited by all its (transitive) subclasses. This is important in the context of NEOCLASSTALK since metaclasses are intended to introduce class properties, such as being abstract or disallowing subclassing. This is why NEOCLASSTALK puts an emphasis on "per class properties" and supports independent "property composition". In our DEEPJAVA design we do not require such an approach since it is—for our ontologically induced clabjects—desirable that (ontological) clabject properties are transitively inherited.

BETA [28] also supports relative types, i.e, accessing type values through attributes, and also uses a genericity mechanism that differs from parametric polymorphism as used in JAVA 1.5. However, unlike DEEPJAVA, BETA's formal generic type parameters (*virtual classes*) are not constrained by a type (e.g., 'HardwareType') but by an upper bound (e.g., 'HardwareItem') that may be further bound in redefinitions until constrained to a single type value in a final binding. In analogy to the discussion on DEEPJAVA abstract typ declarations (see section 4.5.1), type variables constrained by a type are more generic then those controlled by a supertype (upper bound) because they may admit types from different inheri-

tance hierarchies. We have not yet fully explored all potential further differences between the different approaches, but believe that the simplicity of DEEPJAVA's approach—a uniform classification hierarchy were everything is a value and genericity naturally drops out as a byproduct—is convincing even without being more powerful.

In section 3, we have already mentioned *powertypes* as an alternative to achieving deep characterization. Apparently invented by Cardelli [9] and introduced to the modeling community by Odell [33], the *powertype* concept works by establishing an "instance of" relationship between a metatype M and all the subtypes $C_1, \ldots, C_n$, of a supertype S. While Odell only used *powertypes* to motivate the need for (and explain a way to) describe class properties, a powertype M can be used to prescribe the properties of instances of the $C_1, \ldots, C_n$, by forcing their classes to inherit the to be guaranteed properties from the supertype S.

Deep instantiation achieves the same effect more concisely, since *powertypes* distribute the description of the instance facet and the common type facet of the $C_1, \ldots, C_n$ to M and S respectively. Using deep instantiation, one only needs to declare the S features at M and increase their potency values by one. No matter how deep the characterization depth is, deep instantiation allows the concise description at a single concept. *Powertypes*, in contrast, require a staged application of multiple powertypes and supertypes, adding up to complex whole.

Furthermore, *powertypes* require a superclass S (e.g., 'Product') independently of its utility in reflecting the domain structure. Using deep instantiation, one has a choice of either introducing S or not. Due to DEEPJAVA's abstract type declarations, S is not even necessary as a variable type for generic code, since one may use (more flexibly) '@M'.

However, if the introduction of S appears to be beneficial for other reasons then we can achieve the semantics of *powertypes* with DEEPJAVA by using 'extends' relationships with a potency value of 1. A standard 'extends' relationship between two classes has a potency value of 0 because it links two classes with each other without being transferred to the class instances (which in JAVA would amount to inheritance between objects).

Line 2 of Listing 8 shows how we have previously created a dynamic class with a superclass reference to a statically known class ('Product').

```
1 // instead of
2 ProductCategory SoftwareItem extends Product^0;
3
4 // with
5 class ProductCategory^2 extends^1 Product^0;
6
7 // we only need
8 ProductCategory SoftwareItem = new ProductCategory();
```

Listing 8. Powertypes with Deep Instantiation

If we extend the definition of 'ProductCategory' as shown in line 5 of listing 8, we are guaranteed that every 'ProductCategory' instance (such as 'SoftwareItem' or (indirectly) 'DVD') will subclass 'Product'. Hence, we may then create 'SoftwareItem' without manually linking it to the 'Product' superclass as shown in line 8 of listing 8. Unfortunately, due to the desire to stay compatible with the plain JAVA syntax, we cannot abbreviate this line to 'ProductType{} SoftwareItem;'.

'SoftwareItem' may still be assigned a superclass upon creation as in line 2 of listing 8 but DEEPJAVA's type system then demands the specified type to be a subtype of 'Product', otherwise we would introduce a form of multiple inheritance. In summary, we can naturally achieve *powertype* semantics with deep instantiation but *powertypes* cannot attain the conciseness of deep instantiation.

Materialization is a relationship between two concepts which can also achieve deep characterization and has been introduced for modeling databases [18]. Fig. 1 from [18] and in particular the account on *materialization* given by Pirotte et al. [35], reveals that *materialization* can be explained as using the same principles as *powertypes*. In terms of our example, 'Product' $*-$ (materializes) 'ProductType' holds, and both concepts are used to control the type and instance facets of instances respectively (see Fig. 3 in [35]).

ConceptBase [19] is a knowledge representation system based on Telos [31] which supports an arbitrary number of ontological classification levels. Due to its focus on knowledge representation, as opposed to programming, and the fact that one models knowledge in terms of propositions, it apparently, as of today, did not influence the design of programming languages.

There are approaches, e.g., METABORG [7] and METAASPECTJ [38], that support the static checking of strings containing program code, for instance the one shown at lines 4–5 of listing 2. In general, this feature helps to reject improper string contents at compile time and thus avoid runtime errors. However, note that in order to support a fully dynamic creation of types, DEEPJAVA allows any dynamic string content in the definition of a dynamic type. The string value shown at lines 4–5 of listing 2 could have been typed in by a user at runtime. Static checking could still be applied to string constants, such as the one in listing 2, though.

7. Future Work

The main purpose of this paper is to motivate multi-level programming and to present the basis of a supporting programming language. To some extent deliberately but to some extent also out of necessity we did not attempt to present a complete language design. While our working prototype compiler for DEEPJAVA allows the parsing, typechecking, and execution of exciting multi-level programs, such as our running example, it does not yet address a number of issues.

The most interesting issues concern DEEPJAVA's type system and extensions thereof.

In section 4.4 we have demonstrated how DEEPJAVA can support generic classes without requiring additional concepts such as JAVA generics or BETA's virtual classes. By only allowing 'final' type parameters, we give the programmer and type system a chance to exploit static knowledge about type parameters, since—although they are assigned dynamically once—they are immutable. Although we have not finalized our work on an appropriate type system that is permissive enough to allow interesting programs and strong enough to reject as many type-related errors at runtime as possible, we are strongly influenced by the approach of the GBETA language [15]. The latter demonstrates how static typechecking is possible in the presence of types as values, which can be accessed as pattern members, and virtual types whose type is only known by an upper bound. GBETA hence allows a number of intriguing applications, such as *family polymorphism* [16]. Just to hint at how types as dynamic as those of GBETA or DEEPJAVA are still amenable to a static type discipline, consider the following example.

The type of 'aso.promoProduct()' is unknown at compile time, since its owning class 'DVD' and the associated 'CrossPromotionType' (in this case 'DVD_Player') are not known before their creation at runtime. However, the result of 'aso.promoProduct()' can still be used in a type-safe manner: Assuming a further method of class 'DVD', 'playOn(CrossPromotionType)', the following code is statically safe: 'aso.playOn(aso.promoProduct())' or alternatively, 'aso.playOn(new aso.CrossPromotionType())'.

We are currently planning to add *dependent types* to DEEPJAVA which would not only provide a more than satisfactory solution for fully typing generic classes, but would also solve the important problem of covariant redefinitions. When a class redefines a superclass method, it must not redefine the method arguments to be more specific, otherwise either runtime errors may occur or polymorphism has to be forbidden [1]. This is a problem since polymorphism is the key to generic program structures, while programmers often find a need to strengthen the argument types of methods upon redefinition (e.g., ideally the 'equals' method would constrain its argument type to be the receiver type, the latter obviously becoming more specific in a subclass). Many solutions have been proposed to this problem but the only one that really adequately pins down the source of the problem and addresses it accordingly, uses a form of dependent types [36]. We therefore expect to kill two birds with one stone by fitting a type system based on dependent types onto DEEPJAVA.

8. Conclusion

Most of today's programs have a high level of inherent complexity. Any accidental complexity that is additionally introduced because the solution technology cannot accommodate an adequate representation of the problem is therefore doubly unwelcome. In this paper, we argued that a number of systems need to adequately reflect multiple levels of domain classification and that this is in conflict with the current two-level limitation of the object-oriented paradigm. While workaround techniques exist, documenting the need to address this issue, they add a lot of accidental complexity to programs by requiring the programmer to reinvent the wheel in terms of emulating built-in mechanisms of the programming language.

This is why the title of this paper paraphrases the title of John W. Backus's 1977 ACM Turing Award Lecture "Can Programming Be Liberated from the von Neumann Style?" [5] in wich Backus sought for ways of escaping an imperative programming style based on single assignment semantics, because he judged this solution technology to only inadequately address the challenges of programming.

Although a large body of intriguing work successfully extended the original two-level design of object-oriented programming languages to three or more levels, the focus hitherto has been on exerting linguistic control on classes. This approach is useful for creating extensible languages and systems, but does not address the mismatch between the number of domain classification levels and the number of ontological programming levels. Our DEEPJAVA language design, in contrast, does not constitute an extensible language. The capability to let the number of supported ontological levels grow as desired is a fixed, built-in feature of the language. Through conservatively extending the two levels of programming and the associated shallow instantiation mechanism of the object-oriented paradigm to multiple levels with an associated deep instantiation mechanism, we have achieved several advantages. The DEEPJAVA language design

- enables a direct mapping of domain classification levels to programming levels.
- allows the dynamic creation of classes and their features.
- integrates its dynamic features with static typing.
- supports *deep instantiation* as a concise mechanism for deep characterization.
- offers abstract type declarations, thus achieving a level of abstraction beyond powertypes and virtual classes.
- features a natural approach to genericity as a byproduct of its uniform ontological classification hierarchy.
- is a perfect target for incorporating dependent types and their ability to solve the covariance problem.

We are well aware of the fact that the current state of DEEPJAVA's language design cannot be considered final. However, we hope that we have managed to impart our enthusiasm for its already existing advantages and its potential for further development on to the reader.

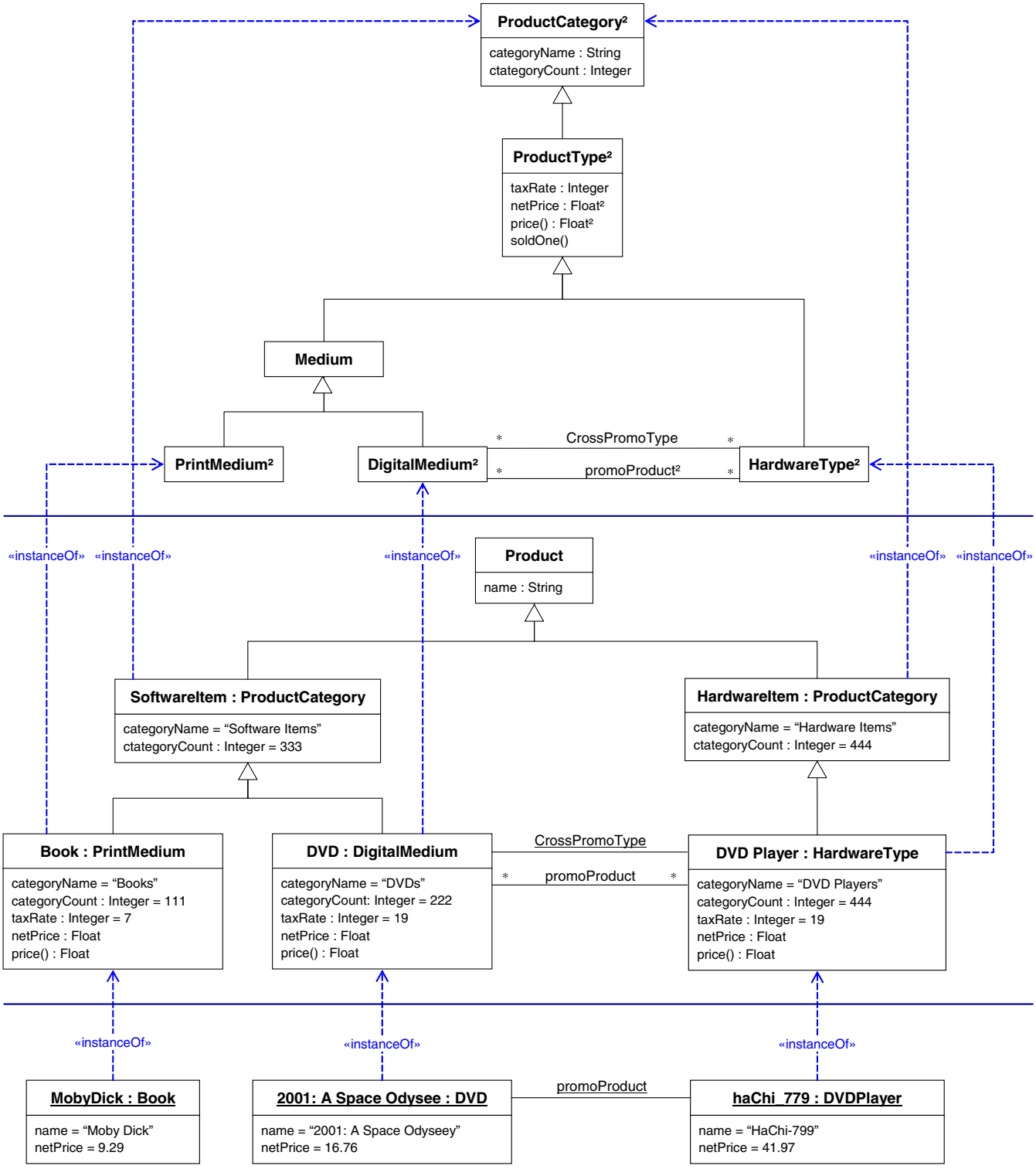

Figure 8. DEEPJAVA Online Store Snapshot

References

[1] Martín Abadi and Luca Cardelli. On subtyping and matching. In W. Olthoff, editor, *Proceedings ECOOP '95*, LNCS 952, pages 145–167, Aarhus, Denmark, August 1995. Springer.

[2] Colin Atkinson and Thomas Kühne. Meta-level independent modeling. In *International Workshop Model Engineering (in Conjunction with ECOOP'2000)*. Springer Verlag, Cannes, France, June 2000.

[3] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of the 4^{th} International Conference on the UML 2000, Toronto, Canada*, LNCS 2185, pages 19–33. Springer Verlag, October 2001.

[4] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Journal on Software and Systems Modeling*, to appear 2008, DOI: 10.1007/s10270-007-0061-0.

[5] John W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[6] Noury M. N. Bouraqadi-Saâdani, Thomas Ledoux, and Fred Rivard. Safe metaclass programming. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 84–96, New York, NY, USA, 1998. ACM Press.

[7] Martin Bravenboer, René de Groot, and Eelco Visser. MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In R. Lämmel and J. Saraiva, editors, *Proceedings of GTTSE'05*, LNCS 4143, pages 297–311. Springer, 2006.

[8] Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[9] Luca Cardelli. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, California, 1988.

[10] CesarGonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software and Systems Modeling*, V5(1):72–90, April 2006.

[11] Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, LNCS 1850, pages 313 – 336, 2000.

[12] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, September 1992.

[13] P. Cointe. Reflective languages and metalevel architectures. *ACM Comput. Surv.*, 28(4es):151, 1996.

[14] Pierre Cointe. Metaclasses are first class: The objvlisp model. *SIGPLAN Notices*, 22(12):156–162, 1987.

[15] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.

[16] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings of ECOOP 2001*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[17] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[18] Robert C. Goldstein and Veda C. Storey. Materialization. *IEEE Transactions on Knowledge & Data Engineering*, 6(5):835–842, 1994.

[19] Matthias Jarke, Rainer Gallersdörfer, Manfred A. Jeusfeld, Martin Staudt, and Stefan Eherer. ConceptBase—a deductive object base for meta data management. *Journal of Intelligent Information Systems*, 4(2):167–192, 1995.

[20] Ralph Johnson and Bobby Woolf. Type object. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 47–65. Addison-Wesley, 1997.

[21] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[22] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97, Finland*, pages 222–242, Berlin, June 1997. Springer Verlag, LNCS 1241.

[23] Thomas Kühne. Matters of (meta-) modeling. *Journal on Software and Systems Modeling*, 5(4):369–385, 2006.

[24] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, second edition edition, 2002.

[25] Thomas Ledoux and Pierre Cointe. Explicit metaclasses as a tool for improving the design of class libraries. In *Proceedings of ISOTAS*, LNCS 1049, pages 38–55. Springer Verlag, March 1996.

[26] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[27] Fernando Daniel Lyardet. The dynamic template pattern. In *Proceedings of the Conference on Pattern Languages of Design*, 1997.

[28] Ole L. Madsen, Birger Möller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[29] Pattie Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*. Elsevier Science Inc., New York, USA, 1988.

[30] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, ISBN 0-13-629155-4, 2nd edition, 1997.

[31] John Mylopoulos, Alexander Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *Information Systems*, 8(4):325–362, 1990.

[32] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference*

on Compiler Construction, LNCS 2622, pages 138–152. Springer, April 2003.

[33] Jim Odell. Power types. *Journal of Object-Oriented Programming*, 7(2):8–12, May 1994.

[34] OMG. *Unified Modeling Language Superstructure Specification, Version 2.1.1, OMG document formal/07-02-05*, February 2007.

[35] Alain Pirotte, Esteban Zimányi, David Massart, and Tatiana Yakusheva. Materialization: A powerful and ubiquitous abstraction pattern. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 630–641. Morgan Kaufman, 1994.

[36] David L. Shang. Covariant deep subtyping reconsidered. *ACM SIGPLAN Notices*, 30(5):21–28, May 1995.

[37] Joseph W. Yoder and Ralph E. Johnson. The adaptive object-model architectural style. In *Proceedings of the 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, pages 3–27, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[38] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In Gabor Karsai and Eelco Visser, editors, *Proceedings of GPCE'04*, LNCS 3286, pages 1–18. Springer, 2004.