UDC 004.434

Using Customizable Properties to make Object Representation a First-class Citizen

Koen Vanderkimpen, Marko van Dooren, and
Eric Steegmans

Department of Computer Science,
Katholieke Universiteit Leuven,
Celestijnenlaan 300A
3000 Leuven, Belgium
{koen.vanderkimpen, marko.vandooren, eric.steegmans}@cs.kuleuven.be

Abstract. Many object-oriented programming languages use fields to represent object state. This representation however, cannot be altered sufficiently when subclassing. Moreover, in languages such as Java and C#, proper encapsulation of fields necessitates a lot of boilerplate code. In this paper, we introduce our concept of properties, which are far more flexible for use with inheritance and greatly reduce boilerplate code compared to C# properties. Using our properties makes it easier for programmers to model programs in a more consistent manner. Furthermore, our Properties allow redefining an object's attributes in ways that equal the possibilities for redefinition of virtual methods in many programming languages, which makes them better suited to deal with unanticipated reuse. Specifically, using our construct, it becomes possible to join several superclass attributes into only one at the subclass level, conjointly decreasing memory consumption.

1. Introduction

In object-oriented programming languages, the representation or state of an object is stored in fields. This representation however, cannot be altered sufficiently when subclassing: classes simply inherit all features, including the fields, from their superclasses. This has an adverse effect on a number of subclassing problems that are already difficult or impossible to deal with if the superclasses were not designed to anticipate them. One problem is consistency: when two superclass fields represent the same concept in a subclass, it requires effort to ensure they remain equal at all times. Another problem is redundancy: when a superclass field is no longer used by a subclass, it requires effort to ensure it is not used by accident.

Specialization inheritance, for instance, usually involves a strengthening of constraints on the representation, typically caused by stronger class invariants that come with this kind of subtyping. This could cause certain fields to become obsolete and thus redundant, because they can be derived

from others. For example, suppose we have a class of rectangles that we specialize into a subclass of squares. A rectangle has a width and a height, but for a square, the class invariant says these two are equal. It would therefore be enough to store only one field, making the other redundant.

Such problems are even worse when using multiple inheritance. When a class inherits from two similar superclasses, for instance, we can select and refine features of both, but when each class introduces a field for one and the same feature, we get an overlap in state, and thus, again, a redundant field. For example, suppose we are trying to make two codebases compatible, and we use multiple inheritance to inherit from classes introduced by both codebases. Suppose the two codebases both introduce a class of cars and that both those classes introduce a weight for a car. When we inherit from the two car classes, we will get two fields for the weight, of which one will be redundant.

Beyond those problems, there's also the fact that proper encapsulation of the representation is usually not imposed. Not encapsulating fields makes it all but impossible to guarantee class contracts while still being able to revise an object's representation, as most changes would break already existent clients that depend on the non-encapsulated fields. For example, a field that is not encapsulated in the superclass makes it impossible to impose any constraints on it [12], or an encapsulated field, accessed by other methods than by its getter and setter, increases the chance for unexpected side effects when these methods, or the getters and setters, are overridden. Unfortunately, in languages such as Java and C#, encapsulation necessitates a lot of boilerplate code, which complicates class contracts.

Effectively dealing with these kinds of problems in current programming languages is difficult, especially if the classes were not designed with the needed kind of subclassing in mind. For example, redundant state and most other problems can be anticipated and/or solved by clever programming methods and design patterns [4,8], or by completely reengineering a (self-contained) program [14,15]. From the subclass point of view, this could go as far as to wrap the superclasses entirely. In the superclasses, it often involves encapsulating a field and allowing to override the methods that access it. Like that, two fields that represent the same feature can be kept the same at all times, or an unnecessary field can be kept out of the code so it does not get used. These solutions, however, are not elegant because they blur the connection between the modelled state and the code used to model it. They also require a lot of effort: both from the creators of the superclasses, who have to anticipate reuse, as from the designers of the subclasses, who need to carefully override many methods. In addition, this effort usually involves introducing boilerplate code. In Java, for example, when we want proper encapsulation of a property, but still want to make it available to clients, we first need to write a getter and a setter for it. Thus the current solutions are not satisfying: redundant fields should be non-existent fields, boilerplate code should be non-existent code, and useless effort should be non-existent effort.

This paper tackles the inflexibility with which object representation is inherited by introducing our properties as a new, more versatile concept for

object representation. The use of our properties allows a more consistent relation between the code and the state it models, which allows programmers to better reflect the modelled state in the code. Furthermore, our properties allow redefining an object's attributes in ways that equal the possibilities for redefinition of virtual methods in many programming languages, which makes them better suited to deal with unanticipated reuse. Specifically, using our construct, it becomes possible to join several superclass attributes into only one at the subclass level, effectively decreasing memory consumption.

In the next section, properties are introduced for single classes, already showing the benefit of a reduction in boilerplate code. In section 3, we explain the use of properties in hierarchies of classes, in which their true power becomes visible. Section 4 deals with transformation to Java and direct compilation of properties. Section 5 evaluates the property concept and section 6 relates it to existing work. In section 7, we present future work, and we conclude in section 8.

2. Properties in a Single Class

In this section, we introduce our property concept, with syntax similar to that of a property in C#. In our examples, we use a language based on Java[1], enriched with multiple class inheritance, as in [2].

2.1. Basic Semantics

This section discusses the semantics of basic, customizable, stored properties. A basic stored property consists of a private value, a getter, and a setter, all of which can be customized.

Default Properties. Consider the class of persons in Figure 1, which plays a central role in a framework for computer games, specifically, a framework for role playing games or RPGs, in which the human player is the hero of a story taking place in a fantasy world. The hero interacts with other people who reside in the game world and who are controlled by the computer. Both the Hero and the non-player characters can be represented by the `Person` class. A Person has two properties: a name and a number of hitpoints.

[1] A major advantage being that we can omit `virtual` and `override` keywords, since all our examples use virtual methods.

```
public class Person {
   public property String name;
   public property int hitPoints = 10;
}
```

Fig. 1. Properties in the Class of Persons.

In the example, the second line declares a private `String` field `name` and its associated public getter and setter. Just as in C#, it is possible to allow getting and setting the property as if it were a field, but using our properties, it is not necessary to declare the name and type of a field twice (as is sometimes necessary in C#: once as a field and once as a property). The main differences between a property and a normal field are automatic encapsulation and the possibility to override the getter and/or the setter to accommodate custom behaviour. In the third line, `hitpoints`, which represent how much damage the character can take in a fight, are initialized to ten, which implicitly invokes the default setter.

Customizing Properties. Our methods of customizing a property's behaviour are at least equally powerful and more self-contained as the possibilities offered by C#: everything non-standard about a property can be declared inside a property block. There are however some syntactic differences. Since there no longer exists an explicitly declared private field to store the property, an implicit variable `value`, which defaults to the property's type, is used to denote it when changes, such as a different type, are needed. Another implicit variable called `argument`, which for isolated properties always has the property's own type, is used to denote the argument for the setter[2].

```
public class Person {
   ...
   public property Date birthDate {
      long value = 0;
      get { return new Date(value); }
      set { value = argument.getTime(); }
   }
}
```

Fig. 2. A Customized Property

To illustrate this, a `birthDate`, represented internally by a `long` and externally by a `Date` object, has been added to the class `Person` in Figure 2. This is a robust way to circumvent the encapsulation problems associated with the `Date` class in Java [3], which is mutable and should therefore not be disclosed to clients.

[2] Note that the name value is used differently in C#, where it denotes the setter's argument.

The private value of a property is only visible inside the declaring property block and in overriding property blocks. The getter's visibility is linked to that of the property and the setter's default visibility also equals that of the property, but it can be made more restricted. This is for example needed when we want a property that can be assigned to in the declaring class, but not by clients. When a property is not `private`, however, the setter cannot be made less visible than `protected`, since it has to remain overridable.

2.2. Derived Properties

Often, many of an object's properties are derived from others: they are not stored separately. A square's surface and perimeter, for example, can be derived from its side. In C# [7], it is possible to use derived properties, simply by neglecting to introduce a new instance variable and programming the getter and setter to use other variables and/or methods. Using that feature, all getters without arguments in a class can be made properties.

Our properties can also be made derived. When the getter and setter are overridden in the property-block without use of the `value` keyword, the compiler will infer that the property is derived and no private value need be stored. Often, the setter is omitted and the property is made `readonly`, since derived properties cannot usually be altered directly.

```
public class Person {
   public property int vitality;

   public property int hitPoints {
      get { return vitality*10; }
      set { vitality = argument/10; }
   }
}
```

Fig. 3. A Derived Property

Consider our `Person` class in Figure 3. We introduce an integer property `vitality` and we make a person's hitpoints equal to this vitality multiplied[3] by ten.

Note that a derived property's type is independent from the type of the properties it is derived from; the programmer is responsible for the code of the getter and setter being able to handle the derived property's declared type and perform any calculations accordingly.

[3] We ignore overflow in the setter.

2.3. Additional Qualifiers

Several additional qualifiers for properties exist. These allow modifications to the default in a quick and easy way, and they are also a clear sign to clients of the special qualifications a property might posses.

The Like Keyword. The `like` keyword can be used to declare properties in a shorter way. When declaring a property to be like another property, we say it has the same type as the base property, while still remaining a separate property. In Figure 4, we have used this in the class `Person` to introduce the property `intelligence`. By declaring it to be like the vitality, it's type also becomes integer.

```
public class Person {
   ...
   public property int vitality = 10;

   public property intelligence like vitality = 20;
}
```

Fig. 4. The Like Keyword

The benefit of using the like keyword for properties in a single class is that we are able to declare a large number of related properties and quickly change the type for all of them, if a change in design would require so. For example, a set of lengths of sides in a class of polygons.

The use of the `like` keyword for completely unrelated properties that happen to be of the same type, such as the house number and age of a `Person`, should of course be avoided.

The Readonly Keyword. It often occurs that properties in a class may only be read. To accommodate this, we allow the keyword `readonly` to be added to a property's declaration. As a result, it does not have a setter, and clients know that this property cannot be assigned to. When using the `readonly` keyword, the compiler forces us to omit the setter when customizing the property, and if we omit the setter, we have to include the `readonly` keyword or this will also result in a compile time error. When the keyword is used with a default stored property, the setter simply does not exist. An example of a readonly property is given in Figure 5.

```
public class Person {
   ...
   public readonly property int hitPoints {
      get {
         return vitality*10;
      }
   }
}
```

Fig. 5. A Readonly Property

Our `readonly` keyword should not be confused with the same keyword in [16], in which it means that the object the reference points to cannot be mutated through this reference. For default stored properties, the definition of our `readonly` keyword is equal to the definition of the `final` keyword in Java. For derived properties, like the hitpoints in Figure 5, however, it does not mean that the value of the property will never change, only that the property cannot be assigned to, not even in an initializer or in the constructor. For default stored readonly properties, a single such initialization is still possible and would apply to the private value directly.

The Constant Keyword. Another useful construct, especially for the clients of a class, is the `constant` keyword. Its definition is simple: the property remains constant for the duration of the object's life. Obviously, this implies it being `readonly`, but it is a stronger constraint. A constant property can still be derived, but it can only be derived from other constant properties.

3. Properties in Class Hierarchies

The true power of properties is revealed when using inheritance: about everything about a property can be overridden when redeclaring it in a subclass.

Suppose the class `Person` from Figure 1 had already introduced a `birthDate` for persons, using a default stored property, and we were not able to alter that class to use the `birthDate` property from Figure 2. If we were working with C#, we could override the getter and setter of the property in a subclass `MyPerson`, but not the privately stored `Date` field from the superclass that would be inherent to such an implementation. As such, the field would become hidden and a waste of memory space for every `MyPerson` object.

Using our properties, we can override the property as a whole, simply by providing the implementation from Figure 2 in a subclass. The getter and setter, as well as the private value, are overridden by the new ones provided in the customized implementation, and in that manner, the redundant field is avoided. Replacing the stored value itself is possible because the mandatory

encapsulation of the value inside the property block ensures unexpected side effects cannot occur, as it cannot be used outside the block.

Overriding simple properties to avoid redundant fields is one thing, but if the superclass(es) provide us with redundant fields to begin with, we can still remove them, by joining multiple properties together, which we will discuss in more detail, first for multiple inheritance, and then for single inheritance.

3.1. Joining Properties in Multiple Inheritance

When subclassing, two or more properties coming from the superclasses can be merged together, freeing up memory slots in the subclass objects. This is used when a relationship between two previously independent variables from the superclasses occurs in the subclass.

A common situation that benefits from multiple inheritance, enhanced with our properties, is the following: consider two frameworks. The first is a framework for computer games, without anything built on top of it, with our `Person` class as its central component. The second framework includes graphical user interfaces (GUIs), and the use of the Observer pattern [8] for communication between the GUI and the program logic.

```
public class Character {
   public property String name;
   public property Date dateOfBirth;

   public void addObserver(...) {...};
}
```

Fig. 6. The Class of Characters.

Obviously, we want to be able to reuse the second framework because of its GUI, but we still want to retain the functionality of our first framework. The second one, however, introduces its own model classes. For example, consider Figure 6. The Observer-based[4] framework class `Character` also introduces a name and birth date for a person. The birth date in `Character` does not behave like it does in the class `Person` of our first framework, and the class does not include any of the other properties introduced in `Person`. So we want to use our model from the first framework with the GUI of the second one.

[4] For simplicity, the actual code to observe the state has been omitted.

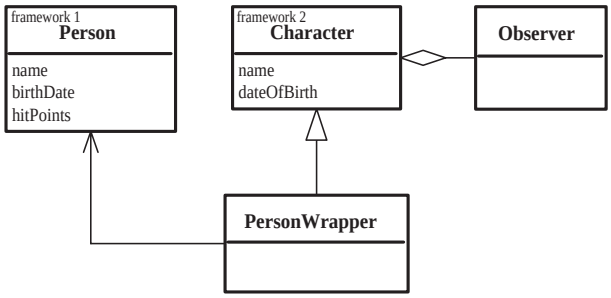

Fig. 7. Using the Adaptor Pattern as a Work-Around.

A traditional method to deal with this problem is to use the adaptor pattern: we can wrap an instance of `Person` in an instance of a subclass of `Character`. Like that, our object has the correct interface for use with framework 2, and the behaviour is that of the `Person` class in framework 1, as all calls are forwarded. This design is demonstrated in Figure 7.

But such an approach has several downsides. For one, we get a longer call chain, because all method calls need to be forwarded. Second and more important is the fact that we inherit two redundant variables from the `Character` class.

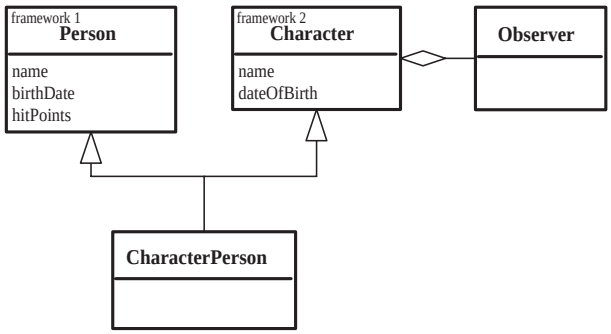

Fig. 8. Using Multiple Inheritance and Properties.

Using properties, we can simply subclass both the model classes to get the behaviour and interface we need, as seen in Figure 8. Obviously, we do not want to store the `name` and `birthDate` attributes twice, so we join those properties together. Multiple inheritance takes care of our objects having the correct interface.

The Join Keyword. Joining properties can be done by placing a `join` keyword after the property declaration in the subclass, followed by a list of the superclass properties to be joined. The effect being that the subclass property replaces both superclass properties, and at most one private field exists. The getter and setter in the subclass property override all the getters and setters in the superclass properties. If the join would cause inheritance of two or more conflicting versions of the getter, setter or private values, they have to be overridden or one of the versions must be selected, as is the case for the `birthDate` property in Figure 9, which demonstrates the `Character-Person` class as a subclass of `Person` and `Character`. One exception to this rule exists: when all superclass properties are default properties (no block) and have the same type, no further specification is needed. This is the case for the `name` property in the example in Figure 9.

When the superclass properties have different names, as is often the case, polymorphism allows those properties to be accessed using their old name through their superclass interface, and by both the old and the new names through the subclass interface. Since properties are virtual, a virtual function table makes sure that the correct implementation is used under all these circumstances. The difference in names can be safely ignored for the getters and setters, as these methods are never literally used by clients, but only function as a means to implement the retrieval and assignment of the property as if it were a field.

```
public class CharacterPerson extends Person, Character {
   public property String name join Person.name, Character.name;
   public property Date birthDate join Person.birthDate,
       Character.dateOfBirth {
      value,get,set : Person.birthDate;
   }
}
```

Fig. 9. The Class CharacterPerson.

Joining properties coexists with dynamic binding: the single property in the subclass is used when objects of that class get polymorphic calls to their superclass interface(s). For this to work correctly, the Liskov substitution principle [10] has to be respected.

More precisely, this means that the argument type for the setter in the new subclass has to be a supertype for all the joined setter argument types, since for method arguments, we have contravariance [5]. The obvious choice is the least upper bound, which always exists, but might degenerate to `Object`. We can, however, statically check the type to be any of the types allowed by the superclasses, so we do not have to accept assignments where the right-hand side is of type `Object`.

Conversely, the greatest lower bound of all joined property types is the new type for the joined property and its getter, because we have covariance [5] of

the return types of the getters. The latter may not always exist, and such a join would thus result in a compile-time error.

```
public class CharacterPerson extends Person, Character {
   ...
   public property CharacterPerson spouse
          join Person.spouse, Character.spouse {
      //argument's type is Object, value's is CharacterPerson
      set {
         value = (CharacterPerson)argument;
      }
   }
}
```

Fig. 10. Joining Properties with Different Types.

This selection mechanism may result in different types for the getter and setter of a property. In that case, the programmer has two options. The first is overriding the setter to accept arguments of the more general type, transforming and storing them in a private value of the more specific type. The second is to override the getter to transform a private value of the more general type into the most specific type before returning it. Beyond overriding one of the methods, it will sometimes be necessary to select an implementation for the other one, when the mechanism does not choose one with a desirable type.

An example of joining when properties with different types are inherited, is shown in Figure 10. Suppose both the `Person` and `Character` class introduce a property for the spouse of a person, which of course has the same type as the class it is declared in. In the depicted class `CharacterPerson`, the type of `spouse` therefore needs to be specialized. The implementation in Figure 10 overrides the setter and uses a default getter that simply returns the private value, which defaults to the type of the property. If it is guaranteed that when `CharacterPerson` is used in the program, no instances of `Person` or `Character` are used, then the code in the setter will never throw a `ClassCastException`, even if the argument type is in fact `Object`. The latter can be explained by the fact that statically, we can still reject code in which the argument type is `Object`, or anything other than a subclass of `Person` or `Character`.

We can go even further when type-checking statically. Contravariance of the setter arguments does not necessarily mean that, at compile time, we have to accept a contravariant right-hand side of an assignment. For example: a class A introduces a property p of type C, and subclass B of A redefines the property's type to type D, subtype of C. In some client code, we have a variable b of type B and a variable c of type C. While the getter in class B still accepts arguments of type C, we can choose to forbid the assignment 'b.p = c', because, statically, c's type does not match the redefined property p's type in class B. This check is not obligatory, but can be easily implemented, as it can be done statically.

Joining and Deriving. Derived properties can be joined just like stored ones, and both types can be mixed together. Depending on which versions of getter and setter are selected, the resulting property uses zero to one memory slots. When the selection mechanism would choose a getter or setter from a stored property being put together with a setter or getter from a derived one, we have to override or manually select both.

3.2. Joining Properties in Single Inheritance

It is also possible to join multiple properties from a single superclass; the same mechanics (such as the selection mechanism) apply. The merger of properties coming from one superclass is useful when a relationship between previously independent fields from that class occurs in the subclass, usually because of stronger class invariants. Consider a typical two-class example: a superclass `Rectangle` and a subclass `Square`. A rectangle has a width and a height, which for a square are equal and are named the side. This can be accomplished easily by joining the `width` and `height` of `Rectangle` with a property `side` in the subclass `Square`, as can be seen in Fig. 11.

```
public class Rectangle {
   public property int width;
   public property int height;
}

public class Square extends Rectangle {
   public property int side join width, height;
}
```

Fig. 11. The Classes of Rectangle and Square.

This has several consequences. First, it means that there is now an implicit constraint on the representation, saying that for squares, the width and height are equal (this is like making the class invariant stronger, as allowed by the principles of substitutability [10]). Second, `width` and `height` are both renamed to `side`. If any of the properties `width`, `height` or `side` are assigned to on a `Square`, both the `width` and `height` properties of a variable of type `Rectangle` that points to that `Square` will be changed.

The ability to join properties in both single and multiple inheritance can avoid a catastrophic blow-up of unused variables in certain class hierarchies, without the need for specific hacks, such as storing all variables in a hashmap or arraylist. Consider a more extended hierarchy of quadrangles, as given in Table 1. Each successive subclass of `Quadrangle` requires less variables to completely define its form. Using normal inheritance without joining, the class `Square` would have four extra unused variables, wasting memory space for each square in the system. Using a combination of joining and deriving

properties, we can reduce the number of used variables to what is strictly needed.

Table 1. The Need for Fields

Class	Required Fields	How to reduce Field
Quadrangle	4 lengths, 1 angle	
Trapeze	3 lengths, 1 angle	Derive one length
Parallelogram	2 lengths, 1 angle	Join two lengths twice
Rhombus	1 length, 1 angle	Join two lengths
Rectangle	2 lengths	Make angle constant
Square	1 length	Join two lengths / Make angle constant

3.3. Advanced Redefinitions

As was mentioned before, anything about a property can be redefined in a subclass. In this section, we discuss the behaviour of the various modifiers for properties under inheritance.

The Like Keyword. The `like` construct is a kind of type anchoring, which can also be found in Eiffel [11]. In a subclass, it is possible to define a property to be `like` a property from the superclass. This makes sure it always has the same type, even if we change this type for the superclass property. This has little effect on stable libraries, but it can be quite useful for a codebase under development, and can make it easier to reason about fragile base class or fragile subclass problems.

Another one of the keyword's effects is that, when redefining a property's type, property's that are declared to be `like` that property also have their type redefined automatically. An example where this could be useful is when we have a class dealing with financial transactions, in which a lot of properties of a type '`MoneyAmount`' exist. If we want to specialize this class for transactions of a single kind of currency, we could redefine the property on which the others are based to be of the type of that currency (e.g. '`EuroAmount`'), and all the other properties would be redeclared without further effort.

Derived to Stored and Vice Versa. Stored properties can easily be made derived in a subclass. This is done by overriding them with a new property block that makes no use of the `value` keyword, essentially the same as when declaring a new derived property.

For the reverse, going from derived to stored, it is possible to write a new property-block that uses the `value` field, but we can also simply redeclare the property without block, in which case the default stored implementation will override the previous one.

Readonly Properties and Subclassing. The `readonly` keyword is an indication that a property cannot be assigned to for a single class. It implies not providing a setter and guarantees clients that objects of that class's type will not allow that property to be assigned to.

This guarantee, however, only stretches as far as the class itself. When the property is overridden in a subclass, the `readonly` constraint is not necessarily repeated, and the property can receive a mutator.

We introduce a new modifier, called `immutable`, to denote properties that are readonly in the class they first receive this modifier, and in all subclasses.

When overriding, a `readonly` property can be made mutable as well as `immutable`. A mutable property can never be made `readonly` or `immutable`, because that would be against the principles of substitutability. An `immutable` property must always remain `immutable`, so that clients using it benefit from a very strong contract.

4. Transformation and Compilation

This section explains how properties can be transformed to Java, providing a way in which they can already be compiled. Direct compilation is also discussed, but is considered future work.

It is our intention to provide this compilation in the future, to be able to do benchmarking on a medium- to large-scale case study.

4.1. Transformation to Java

Our properties can be easily transformed into regular Java code. For each property, the necessary get- and set- methods are simply created in the translated Java class. An example of a transformed model can be seen in Figure 12.

The main difficulty is to get rid of unused fields. These appear when stored properties become derived or when two or more of them are joined. We can solve this using the State pattern [8]. For each class in our enhanced version of the language, we create two classes in plain Java: one representing the class itself, and another representing the state or representation. This separation between behaviour and state makes it possible to simulate the removal of stored variables and thus the joining and overriding of properties.

The primary class follows the correct inheritance hierarchy (e.g. `Square` inherits from `Rectangle`). It contains all the necessary methods, including those generated because of properties, but does not include any instance variables, except one: a reference to an object of the corresponding `Representation` class. Those latter classes do not follow the normal inheritance hierarchy (so `SquareRepresentation` does not inherit from `RectangleRepresentation`). Instead, they all directly inherit from the

class `Representation`, a class introduced so we cannot have arbitrary objects as representations. This method allows classes to have the correct number of instance variables at all times.

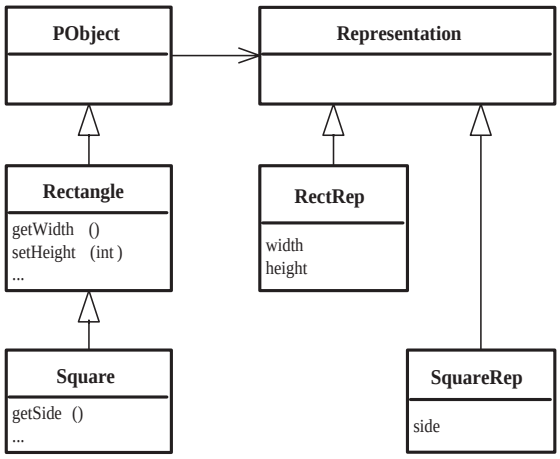

Fig. 12. The Classes of Rectangles and Squares after Transformation.

The representation classes support the simple get- and set- methods for their fields, which can then be called by the primary classes. All superclass methods related to an overridden property, which call the representation in the superclass, are overridden to call the one from the subclass. When joining properties, two methods previously calling different representation methods now call the same one, which can be derived from the join-clause for the joining property. For example, `setWidth` and `setHeight` in `Rectangle` need to be overridden in `Square` to call the representation's `setSide` method.

This is obviously tedious and repetitive work, but it can easily be done by a transformation tool, which we have written. It makes use of the Chameleon [17] framework for metamodels of programming languages. It parses the code and transforms the AST to a model, then transforms that model to a regular Java-model, and finally outputs the code for the transformed model.

4.2. Direct Compilation

Such a simple transformation to Java obviously fails to support the memory-efficiency benefits of properties, since, for each object in the source model, we now have two objects in the transformed model. Even worse, the obligation to work with the existing Java API prevents us from using properties beneficially when inheriting from already existing classes.

Therefore, to realize the full potential of properties, we need some form of direct compilation to byte-code.

A deeper investigation of how properties could be compiled would draw us too far in this paper, but we can at least conclude it would necessitate fields being treated as redefinable (virtual) methods and included in a virtual function table, instead of the fixed position in an object's memory block they have now. Since properties would be treated the same as virtual methods and all changes about a superclass, when it is redefined in a subclass, can be done when compiling the subclass, classes can still be compiled separately.

The virtualization of properties would cause a space/time trade-off, since stored state takes longer to access in this way, but there are no longer any unused variables included in that state. Nevertheless, when programming in object-oriented languages in a decent way (certainly in Java), fields are encapsulated in getters and setters, which also requires a method lookup. Bearing this in mind, the difference in lookup-time is negligible. Furthermore, much research has been done in the area of eliminating virtual function calls in optimizing compilers [1,9,18].

5. Evaluation

The use of properties and the ability to join them have several advantages. Above all, we can represent objects more consistently than in other object-oriented languages. If a square has one defining feature while a rectangle has two, we are able to drop one and make `Square` a subtype of `Rectangle`, as it is in the real world. If we inherit from multiple classes and two or more of them have implemented a similar property, we can join those properties and all of their corresponding methods easily. This increases the consistency between code and the state it models.

Another advantage is simplicity: programmers don't have to worry about rewriting methods to make sure a redundant variable is no longer used in a subclass. The alternative, to keep several existing variables consistent behind the scenes, is at least equally cumbersome to accomplish. All getters and setters are merged by joining properties, and the other, more complex methods, are written in function of those, so they don't have to change.

The compilation of properties necessitates a virtualization of stored features, potentially decreasing runtime efficiency. This, however, is balanced by better memory efficiency and countered by constantly improving compiler optimization techniques. Furthermore, the same reduction in runtime efficiency occurred before, when we manually encapsulated those features, whereas now, they are automatically encapsulated. Thus, practically, there is no real decrease. In any case, when it comes to being able to write code more correctly, faster and more easily, we feel that runtime efficiency is of secondary importance. With today's increasingly performant machinery, well-designed and reusable code is a lot more important than being able to execute it faster.

6. Related Work

C#. In C# [7], it is possible to define a property block, but the getter and setter still need to be explicitly implemented, as well as the private field that stores the property. Moreover, in C#, properties cannot be joined and stored properties cannot be eliminated.

Eiffel. In Eiffel [11], features (methods and attributes) inherited from multiple superclasses can be joined by renaming and undefining them in all except one of the superclasses. For this to work, they need to be renamed to the same name as the feature in the class in which they are not undefined. But it is not possible to rename two features of a single superclass to the same name; that would introduce an ambiguity. Deferring one of them is not an option, since the rename clause has to precede the undefine clause, and thus, when they are renamed to the same name, they would both be deferred.

Furthermore, it is possible in Eiffel to merge two or more stored properties, also called attributes, when they are from superclasses with a common base class, but not in other cases. In the language, we can only turn derived properties into stored ones, not vice versa.

Cecil. In Cecil [6], fields are similar to our properties. An instance variable, or field, can be declared, and will automatically be hidden. Getter and setter methods are also automatically created for such a field. Because the representation is always wrapped by methods, it becomes possible in Cecil to override methods by instance variables and vice versa. Joining fields explicitly without having to override many methods, however, is not possible.

Self. In the dynamically-typed language Self [13], variables and procedures are united into a single construct. Because it is prototype-based, redefinitions not resulting in redundant state, similar to those for properties, are possible. As in Cecil, explicit joining is not supported.

7. Future Work

We would like to introduce constraints on properties as a language concept, as is possible for methods in Eiffel. This will allow us to better explore the consequences of using and joining properties on class contracts. If we were able to associate invariants with properties, it would be possible to see a join between properties as simply an extra class invariant, declaring equality between existing properties.

The current manner in which properties deal with redefinitions of their type, is to allow the setter to get a more general type and for the getter to receive a more specific type, allowing the programmer to work around the difference, if any. A better solution would be to have covariance for the getter as well as

the setter, which would more logically comply with the subclass receiving a stronger class invariant: A property of type P in the superclass will be of subtype P' in the subclass. This would however break certain calls on variables of the superclass type pointing to objects of the subclass type that have an object of type P as their argument. In that case, we propose a new kind of exception, a so-called 'MaltypedAssignmentException', which would be thrown automatically, just like a NullpointerException. As future work, the implications of such a construct will be studied.

8. Conclusion

We have shown how the current storage of fields in statically typed object-oriented languages is inadequately able to cope with certain changes when subclassing, which results in redundant fields or an obscured connection between the modelled state and the code. We have also explained how this may unnecessarily complicate a programmer's work.

To solve these problems, we have introduced the concept of properties in those languages. Properties are a kind of advanced attributes, which encapsulate fields and automatically introduce getters and setters in a class. They can be read and assigned to as if they were fields, but the getters and setters will be called in the background.

Properties can be made stored or derived, and this can be changed in subclasses. Derived properties do not take up space in memory. Properties can also be joined in subclasses, stating that in the subclass, several superclass attributes always have the same value. Joined properties use only one slot of memory. Some properties are readonly, meaning they do not introduce a public setter.

The use of properties helps to program an object's representation more logically and consistently, and increases the memory efficiency of programs. Furthermore, it reduces boilerplate code.

9. References

1. G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In ECOOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming, pages 142–166, London, UK, 1996. Springer-Verlag.
2. L. Bettini, M. Loreti, and B. Venneri. On multiple inheritance in Java. In Technology of Object-Oriented Languages, Systems and Architectures, Proc. of TOOLS Eastern Europe 2002, pages 1–15. Kluwer Academic Publishers, 2003.
3. J. Bloch. Effective Java: Programming Language Guide. Java series. Addison-Wesley, Boston, 2001.
4. G. Booch. Object-oriented analysis and design with applications (2nd ed.). Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.
5. G. Castagna. Covariance and contravariance: conflict without a cause. ACM Trans. Program. Lang. Syst., 17(3):431–447, 1995.

6. C. Chambers. The Cecil language: Specification and rationale. Technical Report TR-93-03-05, 1993.
7. M. Corporation. Microsoft C# Language Specifications. Microsoft Press, Redmond, WA, USA, 2001.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts, 1994.
9. K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 294–310, New York, NY, USA, 2000. ACM Press.
10. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst., 16(6):1811–1841, 1994.
11. B. Meyer. Object-oriented software construction (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
12. D. L. Parnas, P. C. Clements, and D. M. Weiss. Enhancing reusability with information hiding. In Software reusability: vol. 1, concepts and models, pages 141–157, New York, NY, USA, 1989. ACM Press.
13. R. B. Smith and D. Ungar. Self: The power of simplicity. In LISP and Symbolic Computation, pages 187–205, Netherlands, 2005. Springer.
14. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, pages 99–110, New York, NY, USA, 1998. ACM Press.
15. F. Tip and P. F. Sweeney. Class hierarchy specialization. OOPSLA SIGPLAN Not., 32(10):271–285, 1997.
16. M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005), pages 211–230, San Diego, CA, USA, October 18–20, 2005.
17. M. van Dooren. Abstractions for improving, creating, and reusing object-oriented programming languages. PhD, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2007.
18. Y. Zibin and J. Gil. Two-dimensional bi-directional object layout. In ECOOP '03 - Proceedings of the 17th European Conference on Object-Oriented Programming, pages 329–350, Berlin, Germany, 2003. Springer-Verlag.

Koen Vanderkimpen is a researcher at the software development methodology research group of the department of computer science at the K.U.Leuven, Belgium, since 2005. His research focuses on improvements to object oriented programming languages.

Marko van Dooren is a postdoctoral researcher at the software development methodology research group of the department of computer science at the K.U.Leuven, Belgium, since 2001. He received his PhD degree in 2007. His research focuses on improvements to object oriented programming languages.

Eric Steegmans is a professor at the department of computer science at the K.U.Leuven, Belgium, since 1990. He is the leader of the software

development methodology research group and teaches several courses in object oriented software development.