Enriching the Object-Oriented Paradigm via Shadows in the Context of Mathematics

Marc Conrad, University of Luton, UK
Tim French, University of Luton, UK
Marianne Huchard, Université Montpellier II, France
Carsten Maple, University of Luton, UK
Sandra Pott, University of Glasgow, UK

It is well-known that few object-oriented programming languages allow objects to change their nature at run-time. In this paper we discuss the need for object-oriented programming languages to reflect the dynamic nature of problems, particularly those arising in a mathematical context. It is from this context that we present a framework, together with a Java-like implementation of that framework, that realistically represents the dynamic and evolving characteristic of problems and algorithms.

1 Introduction

It is acknowledged that there is a strong relationship between Mathematics and object oriented techniques as indeed this issue has recently been discussed and reported in some detail in [9] and [7]. The strong links that appear to exist will come as no surprise given the close conceptual cross-fertilization as between object-orientation and certain familiar generic mathematical concepts as abstraction, generalization, and overloading. Mathematicians are able to better contextualize and also instantiate these and other relevant mathematical constructs within software artefacts using programming languages such as Java or C++. Equally, Computer Scientists are able to explore fundamental mathematical constructs through an easily accessible, natural medium of expression (i.e. a simple Java program) that enables them to concrete-operationalize mathematical abstractions.

On the face of it therefore we have a "marriage made in heaven" or at the very least a marriage of convenience. However, in practice this ideal and synergistic relationship can become seriously undermined. For example, whilst the pure object-oriented paradigm is closely linked to mathematical structures, the actual implementation of mathematics within software artefacts often does not follow a strict object-oriented design process. Rather, the methodology of choice or convenience is quite often based on what could be called ad hoc methods that combine rapid prototyping with eXtreme Programming (XP) approaches [2], [14]. We do not advocate the mass re-education of mathematicians to adopt more rigorous and formal object oriented design processes such as exemplified in the UML (Unified Modelling Process). We prefer to simply accept the de facto situation as a given,

and go on later to suggest ways in which an extension to object orientation, namely shadows can be used to actively support the development of mathematical software artefacts, within informal and rapid software development contexts of use.

Aside from the issue of how to extend object orientation in ways that support these ad hoc design methods, we go on to probe more fundamentally the role of standard Java in such contexts. It may be the case for example that Java itself is acting as a kind of conceptual constraint for both mathematicians and Computer Scientists alike, preventing and restraining the instantiation of mathematical abstractions in ways that might seem perfectly "natural" to both of these client groups. For example, it is well known that Java only supports a highly restrictive form of inheritance whereas mathematically, other forms are commonly encountered as part of the normal diet (such as dynamic inheritance and method renaming). One of the aims of this paper is therefore to critically examine some wider types of inheritance in some detail so as to encourage our target client groups to move their thinking beyond the (artificial) limitations of the standard Java environment (or in fact any specific programming environment imposes). In order to illustrate the value of "thinking outside the box" we go on to present an implementation of shadows in the context of Java. Shadows appear to be well known and used in Computer Games development, but have failed thus far to become integrated within mainstream languages.

It is of course quite possible to incorporate additional features into Java (or C++, ...) through the use of various extensions such as the Darwin project [13]. Equally it is possible to develop customized software implementations that seek to explore mathematical structures in an overtly axiomatic manner such as com.perisic.ring [5] or Axiom [12]. In this paper we present a conceptually simpler solution based on a modified Java compiler.

Whilst Java is undeniably industrially credible and popular, it is nevertheless useful to examine its limitations too. We go on to explore the limitations in respect of inheritance and proceed to suggest ways in which these limitations may be overcome through widening thinking processes, and/or by a process of extension through the creation of additions or tailored development environments. Our solution is therefore more consistent and less intrusive as other solutions (so they exist). We demonstrate in this paper our implementation of a modified Java compiler that nonetheless produces legal bytecode that can be interpreted by the standard Java virtual machine. This compiler together with the accompanying Java package is available at our website [6].

The remaining of the paper is organized as follows. First we have a closer look at the link between mathematical structures and object oriented concepts. This is followed in section 3 by a description on how this link creates demands on an object oriented language. In section 4 we propose shadows as a solution to these demands. Finally (before the conclusion), in section 5, we demonstrate how we implement shadows as an extension of Java based on a modified Java compiler.

2 The link between Mathematics and Object Orientation

In the context of Computer Algebra there are some packages that support the object oriented paradigm. For example Axiom [12] has type hierarchies ordered in an inheritance like structure and similarly Mupad [26] that explicitly enables the defining of child classes of existing classes as groups, fields, etc. Even mainstream mathematical software packages, such as Maple [31] and Mathematica [33], now reflect the significance of Java as a programming language by offering an interface between the package and Java applets and applications.

A radical approach to describing mathematical relationships within object oriented software can be identified in the experimental Java package com.perisic.ring [5]. It is focussed on an object oriented implementation of mathematical structures in an axiomatic manner. In the following we will take a closer look at aspects of the design ideas of this package as they serve well as a "reference model" for the discussion on non-standard inheritance relationships presented in section 3. For details we refer the reader to the web site and documentation of [5].

The main philosophy of the com.perisic.ring package is that postulated properties of a domain are reflected as abstract methods of Java classes. For example an algebraic ring *has* by definition addition and multiplication. Of course, addition and multiplication are not known *algorithmically* for an arbitrary (unspecified) ring: they cannot be implemented.

Therefore the mathematical entity "algebraic ring" is implemented as an abstract class `Ring` with abstract methods as in

```
abstract public class Ring {
  abstract public RingElement
add(RingElement,RingElement);
  abstract public RingElement mult(RingElement,
RingElement);
...
}
```

Both methods have "ring elements" as arguments and return values. A `RingElement` is a class (it could also be an interface) that can be (polymorphically) associated to any object (e.g. a multi-precision integer, or a vector of `RingElement` objects).

Not all methods are abstract, hence `Ring` could not be declared as an interface. For instance a method `square` is implemented via $a^2 = a \cdot a$ using the abstract multiplication method. More sophisticated non-abstract methods are exponentiation or evaluation of a polynomial.

Thus, an abstract Java class is able to mimic the axiomatic definition of a mathematical entity, here a "ring". The axioms *addition* and *multiplication* appear as abstract methods in `Ring` while *squaring* is not axiomatic and therefore may be im-

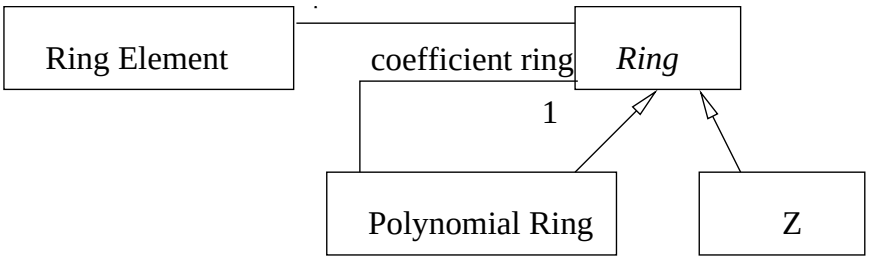

Figure 1: A UML diagram of the implementation of polynomials over $\mathbb{Z}$ following the com.perisic.ring approach.

plemented. Note that structural axioms such as *associativity* or *distributivity* cannot be declared as methods but rather have to be formulated as constraints. Note that in the context of the UML we could have used the OCL (Object Constraint Language) to express these constraints, but in doing so they would become more descriptive than operational in their characterization.

In Figure 1 we give an example illustrating the power of this concept by showing how multivariate polynomials arise naturally by a straightforward implementation of univariate polynomials.

The abstract class `Ring` bidirectionally links with many `RingElement` objects as described above. The class `Ring` has two child classes, namely the ring of integers $\mathbb{Z}$ and a univariate polynomial ring. Both of these rings implement addition and multiplication. For $\mathbb{Z}$ this is possible via built-in integer operations, whilst the polynomial ring implements addition and multiplication via the addition and multiplication methods of the coefficient ring. That is, the polynomial ring is not only derived from the class `Ring` but also *uses* a `Ring` object in its role as a coefficient ring. Starting with an instantiation of $\mathbb{Z}$ we can then recursively instantiate polynomial ring objects as $\mathbb{Z}[a]$, $\mathbb{Z}[a][b]$, $\mathbb{Z}[a][b][x]$, and so forth.

In a similar way other structures can be implemented using the abstract class `Ring`. For instance the com.perisic.ring package supports quotient rings, modular rings, and universal rings. Obviously the design pattern can also be applied to groups, vectors spaces, and metric spaces, for example.

3 Non-standard features

We go on with the discussion by explaining which features canonically arise as a demand from the link between algebraic structures and object oriented principles as described in the previous section. We discuss in particular Dynamic Inheritance, Dynamic Generalization, and Method Renaming. In section 4 and 5 we see how these features are implemented in software using shadows.

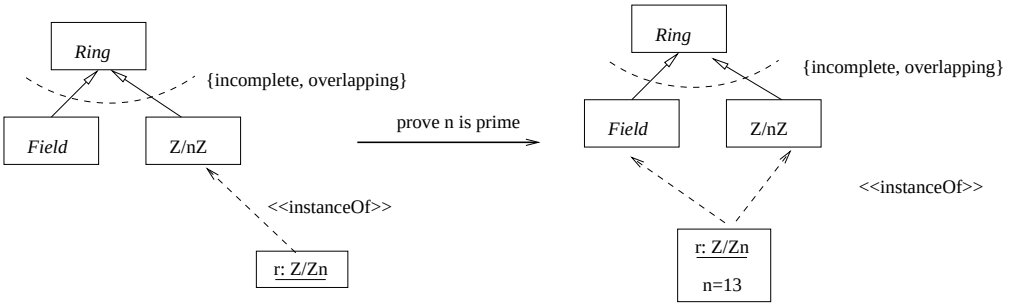

Figure 2: When n is proven to be prime the instantiation relationship changes.

Dynamic Inheritance

When implementing the mathematical entities "field" and "ring" as classes, it is straightforward to have them arranged in an inheritance relationship as shown in Figure 2. A field *is a* ring with additional properties (such as division). Some algorithms, for instance vector space computations, work only for fields and not rings thereby justifying an extra class "field", rather than introducing "fieldness" as a property of ring.

For any given ring object however we do not always know *a priori* – that is at compile time – if it is a field. For instance the ring r declared of type $\mathbb{Z}/n\mathbb{Z}$ in Figure 2 is a field if and only n is a prime number. Obviously the developer of the class hierarchy or a compiler usually cannot predict whether or not a variable holds a prime number at a certain point of program execution (actually this is a special variant of the halting problem). The most appropriate solution is to assume at the start of the program that r is a ring. When the primality of n has been proven the instantiation relationship should change, so that r, now also an instance of a field, can for example be used in the context of vector space computations.

Of course we freely acknowledge that Dynamic Inheritance is hardly a "new" feature. A C++ implementation (or rather workaround) is already discussed in [10]. Also strongly related is the concept of *predicate classes* [3] that is rooted in Cecil [4] but in essence is language independent. In [22] a Java extension featuring Dynamic Inheritance is proposed. Dynamic Inheritance is supported in Lava as part of the Darwin project [13].

Although our focus is on mainstream languages such as Java, we briefly discuss other concepts dealing with dynamic inheritance in order to get a better idea how such features could be incorporated into Java, namely Self [29] and *Fickle* [17].

Self [29] emphasizes an object driven approach rather than a class/instance driven approach. New objects are generated by copying existing objects and modifying them. This increases the flexibility of objects since they can act more independently than in a more rigid class driven paradigm. One consequence is that an object can flexibly add and remove parent objects any time during program execution.

From a mathematical point of view it makes sense, in certain cases, for an object

to be able to change its ancestors. For instance it is well known that the ring $\mathbb{Z}[i]$ is Euclidian, while most other quadratic orders $\mathbb{Z}[\sqrt{d}]$ are not. Therefore, from an object-oriented point of view, the information *Euclidian* is located in the ring object itself and not in the class. The Expert design pattern [24] suggests that the object should be able to change its parent class (from Ring to Euclidian Ring).

From the point of flexibility, languages such as Self are optimal and below we further discuss how such flexibility can be transferred into a mainstream language such as Java using shadows. Mathematical intuition, i.e. examining this issue from a formal conceptual perspective, suggests that we should not *give up* on classes too easily. In fact the ideal situation would be that $\mathbb{Z}[i]$ is a child object of a Euclidian ring whilst still being of class *quadratic order*. For example, with reference to the earlier example of modular integer rings, $\mathbb{Z}/13\mathbb{Z}$ is and should remain of type modular integer rings, i.e. an instance of a class $\mathbb{Z}/n\mathbb{Z}$, instantiated with $n = 13$. After proof of primality, its parent class will be changed to *Field*, but it will not cease to be a Modular Ring. We will see that this concept could easily be modelled with shadows.

A possible compromise solution is to introduce a class *Finite Prime Field* that is a child class of *Modular Integer Ring* which implements a *Field* interface (we have to abandon the idea of an abstract *class* Field, as Java does not support multiple inheritance). Then the task is "simply" to *reclassify* at runtime the $\mathbb{Z}/13\mathbb{Z}$ object from *Modular Integer Ring* to *Finite Prime Field*. Reclassification is a concept both discussed and directly implemented in *Fickle*.

The general idea is this: An object is related to a *Root Class* (in this case the class *Ring*). Then it can be reclassified to each child class (called *State Classes*) of this Root class. A special operator *!!* in *Fickle* reclassifies an object from one State Class to another when both belong to the same Root Class. Please see [17] for further details.

The translation of *Fickle* into Java as described in [1] is conceptually interesting. Here, each *Fickle* object is translated in Java to a pair $< id, imp >$ where *id* is the *identity* of an object and *imp* is the *implementation* of the object. Whilst the *identity* remains unchanged for a particular object, the *implementation* has the class information and can thus change. From this concept we can deduce the feasibility of dynamic inheritance in Java. From a pragmatic viewpoint the resulting Java code is complex, difficult to follow, and counterintuitive, to say the least. For instance a method implying a possible change of class has to be translated into a pair (a static and a member method). Calls of member methods in *Fickle* are translated into calls of static methods with an object as an argument (this is quite similar to C workarounds for object oriented programming).

Nevertheless the *Fickle* idea can serve as a roadmap for an implementation of reclassification in Java directly. An object that is to be reclassified is represented as a pair of an identity and an object of the current class. That means, from a user perspective that the only additions to the Java language would be:

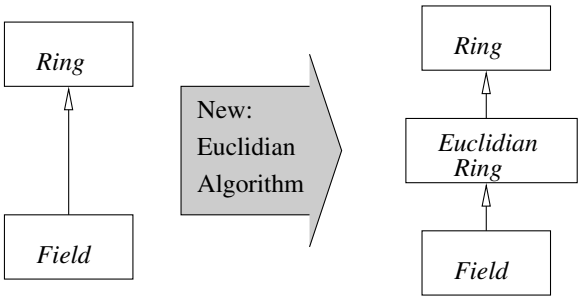

Figure 3: The "new" Euclidian ring has its place in the middle of the hierarchy

- a keyword, say `dynamic`, in connection with the `new` keyword, indicating that a new object can be reclassified;
- a method `reclassify` indicating a reclassification.

In this paper we present a more radical approach to reclassification that however avoids an extension of the Java language on the syntax level: No new keywords are introduced. In section 4 we will describe how Shadows can be used to address and solve the problem of reclassification.

Dynamic Generalization

Assume for the moment that Euclid were a contemporary mathematician who had just discovered the Euclidian division (also known as division with remainder), and that he wants to add Euclidian division into existing mathematical software that features a ring/field implementation as on the left hand side in Figure 3. The proper place for a Euclidian Ring – a ring with Euclidian division – is between the ring and field. Not every Ring is Euclidian and every field is trivially a Euclidian ring [23].

As this fictional example shows mathematical progress and invention does not only confine to deriving child classes from parent classes but also encompasses the invention or discovering of properties that do not apply to all entities (here: Rings) and may be trivial for (but apply to) a subset of these entities (here: Fields). In object oriented terms that means that in mathematics it is natural for new classes to appear as a *generalization* of an existing class, or even as both a specialization and a generalization of existing classes. This process is known as *interclassing*. For a motivation of interclassing from a software engineering point of view see [11].

Although the imaginary scenario that Euclid invented Euclidian division after the invention of the concept of a ring and field may be somewhat artificial, defining new structures in the context of an existing hierarchy is a standard technique in contemporary mathematics. A typical example taken from Functional Analysis are the Triebel-Lizorkin spaces, which were introduced in the 1970's as simultaneous generalizations of a number of well-known classes of function spaces, e.g. L^p spaces, Hardy spaces, the space of functions of bounded mean oscillation (BMO), Lipschitz

spaces and Sobolev spaces (see e.g. [30]). A Triebel-Lizorkin space is a specialization of a Banach function space. A more recent example are the so-called real Q-spaces, which are a simultaneous generalization of the space BMO and certain other Banach function spaces [18].

This "interclassing" in Mathematics is often motivated by the desire to create a unifying framework for several known classes of mathematical objects in a certain context (as in the first of the two examples mentioned above), or to bring existing mathematical techniques to new applications (in the second example). We see here that dynamic generalization is crucial for any software library that maps abstract mathematics into software.

Note that the problem of interclassing is substantially different from the problem of run-time reclassification described earlier in this section. Here, we start with a class hierarchy that may be arranged in a package and that may not even contain any source code. We want to extend this class hierarchy by adding a class on a well defined position in an inheritance tree. Even if the source code is available it may be not desirable to change this code, especially if the class library is well established and the addition of the new class has *experimental* character, or is only relevant for a specialized application area. We will come back to that point later in the discussion of shadows for prototyping and deprecating.

Outside of a mathematical context the idea of *interclassing* is already discussed in [28]. In [11] an implementation is described using the OFL model. However, in terms of pragmatic usage OFL is inadequate as it requires *de facto* to learn OFL as an additional language, namely the understanding of the correct use of hyper-generic parameters. Also, in using hyper-generic parameters the developer of a library already unnecessarily restricts possible extensions.

For these and other reasons we demonstrate in this paper a mechanism for interclassing at *run-time* an existing library that is conceptually simpler and is also proven to be workable in practice in the application area of computer games [27].

We will illustrate this by extending the earlier Euclidian Ring example: Imagine an inheritance hierarchy with a Ring as a parent class and two rings, say field and polynomial ring over real numbers as child classes, and we further assume that these classes are part of a library and cannot be changed simply by adding source code. However we *can* assume that the developer of the class hierarchy wants to allow interclassing, assuming that a mathematical developer knows that new structures are likely to be added.

Adding a new class *Euclidian Ring* in this hierarchy will be between the Ring class (as parent) and both the Field and Polynomial Ring (as child classes). Thus, at least in principle, both Field and Polynomial Ring, obtain additional behaviors. The obvious problem is, *how do the Field and Polynomial Ring "know" about this additional features?* The solution is, as we will see later in more detail, *shadowing* of objects and/or classes. Shadowing of an object means that each message sent to this object is "filtered" through a set of shadows. If the message is already understood

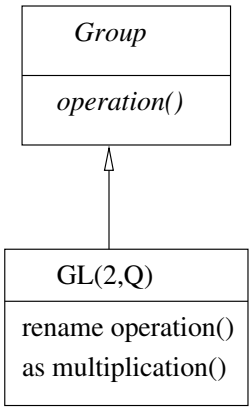

Figure 4: $GL(2, \mathbb{Q})$ overrides operation() and renames it as multiplication().

in the shadow, it is executed in the shadow. If the message is not understood, then it is passed to the shadowed object. In addition, the shadow itself is able to send a message directly to the shadowed object. Shadowing is a language feature of LPC [27], a language designed for implementing MUDs (namely LPMuds). The shadow mechanism for objects is documented in [15]. As LPC is classless (using rather a prototype approach with cloneable objects as in Self), there is obviously no mechanism for class shadowing. Therefore we assume that shadowing a class means that a shadow is thrown (automatically) onto each object instantiated from this class. Also, LPC does not allow to shadow an inheritance relationship. Again, there is conceptionally no problem to allow this in our solution (compare e.g. with Self [29] where each parent slot is in fact a special kind of a data slot), as demonstrated in section 4.

Our shadow implementation uses a modified compiler for compiling the Java source code into bytecode. There seems to be no straightforward workaround for dynamic generalization at run-time (other than changing the source code and re-compiling) that is based on the use of Java libraries only. The main problem that any workaround faces is the inherent difficulty of informing an object (in this case a Field) to accept a message (in this case Euclidian Division) that hasn't been originally defined in a member method. We will see in section 5 that we tailored our Java compiler to exactly address this point.

Overriding with Renaming

A group is a set with an operation and certain properties (the existence and uniqueness of a neutral element, associativity, etc). In general the operation is denoted by the symbol $\circ$ as in $c = a \circ b$. In a concrete situation there is often a standard notation for the group operation. The most familiar are $+$ for addition in an additive group and $*$ or $\times$ for multiplication in a multiplicative group.

For instance Figure 4 shows the multiplicative group $GL(2, \mathbb{Q})$ of invertible 2×2

matrices as a child class of an abstract group with an operation. The natural way to implement the group operation includes renaming the operation multiplication.

Renaming is hardly a new feature in object oriented contexts. In Eiffel renaming is the preferred method of choice to avoid ambiguity in multiple inheritance relationships [25]. Renaming exists also as standard feature in Python [20]. In contrast to the problems discussed earlier (Dynamic Inheritance and Dynamic Generalization) this is not a sophisticated problem from a software engineering point of view. However adding this concept to Java would be an easy step to improve usability of Java within a mathematical context. Having method renaming as a first-class feature would be desirable. However shadows at least provide a gentle workaround for that feature.

4 The shadow concept

Shadows have been first introduced as a feature of the programming language LPC for computer games. LPC is an interpreted language created in 1988 by Lars Pensjö (and later further developed by other contributors) for his invention LP-MUD, an interactive multi-user environment mainly used for text based adventure games ("Muds") [32]. The basic syntax owes – similar as in Java – much to C with the addition of an object oriented structure. There are no classes in LPC. Objects are either instantiated by loading them from a file or by cloning them from other objects.

The evolution of LPC has been highly pragmatic driven by the demand of the active programmers in various Muds rather then by a systematic, academically based, concept for designing a programming language. The shadow concept in LPC must be seen in this context: It has been proven to be useful "as is" but it has so far not been evaluated academically.

For enhancing clarity we leave the application area of Mathematics and focus to a more general setting. In the following we give first a description of the basic principle. After that we show how shadows could be useful in a number of application areas. We hope that this paper may also encourage others to adopt shadows as a first class feature of a programming language.

As outlined above we freely acknowledge that there exist also other solutions for each of these areas, however a shadow could provide a *unified* approach for all these otherwise unrelated application areas. This is even more desirable as the basic principle of a shadow is comparatively easy to understand.

How the shadow concept works

The idea with the shadow functionality is to mask one or more methods in a target object (the "shadowed" object). A shadow is usually applied at run-time. So a

shadow gives an object for a certain amount of time a behavior that is different from the default behavior of the object. In the following we list some properties of shadows in LPC to give a better idea of the concept. After that we compare this with the corresponding features of our implementation. These are the features of shadows in LPC:

- A shadow cannot be shadowed.
- When an object returns a value "true" on a function call `query_prevent_shadow()` it cannot be shadowed.
- A method cannot be shadowed if it is declared `nomask`
- Attributes cannot be shadowed.
- Only calls made from "external" objects can be shadowed. Translated to a Java situation imagine that the method `bla()` of an object `foo` is shadowed. Then a a method call `foo.bla()` will be received by the shadow while a call `bla()` in the object `foo` itself will not. We achieve the shadowing of an "internal" call by artificially making it external using the syntax `this.bla()`.

The concept of our Java implementation has been first presented in [8] and the feedback received there has had a major influence on the current design of our solution.

The Java implementation differs slightly from that in LPC. Instead of a function call `query_prevent_shadow()` all shadowable objects must inherit from a base class `com.perisic.shadow.jshadow.Shadowable`. Also in the current version there is no equivalent to the `nomask` keyword of LPC, as we want to avoid the addition of new syntax elements to Java. In addition to the other features of LPC we also provide a mechanism for shadowing a class and an exception class.

Application areas for Shadows

The usefulness of shadows is not only restricted to interclassing and reclassification in the context of mathematics. Therefore this section closely follows [9] in discussing the usefulness of shadows outside of mathematical context and in addition to reclassification and interclassing we include also prototyping and deprecating as possible application areas.

Deprecated methods

Software libraries are under constant evolution and it is a fact that sometimes methods are replaced by other methods that may be more consistent with naming conventions etc. The problem is that legacy code usually uses these deprecated methods. So it is impossible for the provider to remove them completely from the library.

A shadow system could help the provider of the software to separate an object in two parts. The actual, official version of the object that is not messed up with any deprecated methods and a shadow for this object that contains deprecated methods. In cases that a deprecated method is necessary for an application the object has to be shadowed. This creates the overhead of having the additional method only locally where the deprecated method is needed.

Syntactically this could be solved similar as with the Java properties mechanism that it is possible in the environment to specify Default shadows for each class. The class loader then automatically throws these shadows when the class is loaded.

Prototyping

Similar as shadows could be used for "fading out" deprecated methods, shadows could also be used for prototyping in software development. Especially in the case that a development process starts from an existing library and it is vital that the library is not to be changed (or that it is not *possible* to change the library for instance because of, say, that it is bought from an external supplier or because of *copyright* issues).

A shadow then changes the behavior of a class or object temporarily or for a well defined situation during development. An evaluation with extended experiments using the shadow may then be used to collect arguments for or against the adaption of the proposed change of the class.

We could even imagine that shadows may be used (automatically) in a concurrent version system (eg. CVS) to implement branching in software development.

Reclassification and Dynamic Inheritance

Reclassification and the special case of it, dynamic inheritance, means to change the class of an object at run-time. A work-around for dynamic inheritance in C++ is already discussed in [10]. Automatic reclassification based on the value of predicates is implemented as *predicate classes* [3] in Cecil [4]. In [22] a Java extension featuring Dynamic Inheritance is proposed. It is supported in Lava as part of the Darwin project [13]. The most consequent approach for reclassification can be found in *Fickle* (e.g. [17], [1], [16]).

It is not a coincidence that the *Fickle* example in [16] is located in the context of a computer game: A *Player* that is (an instance of) a *Frog* is reclassified to a *Prince* after being kissed. It is exactly this kind of problems that are in practice (namely in LPC muds) pragmatically solved with shadows. The main goal in reclassification is the change of the behavior of an object: The behavior of a *Frog* is different from the behavior of a *Prince*. This is achieved by the application of a shadow to a player object. Different shadows may change the default behavior od the player. For instance a magic spell might add a "Frog shadow" to the prince, thus enabling

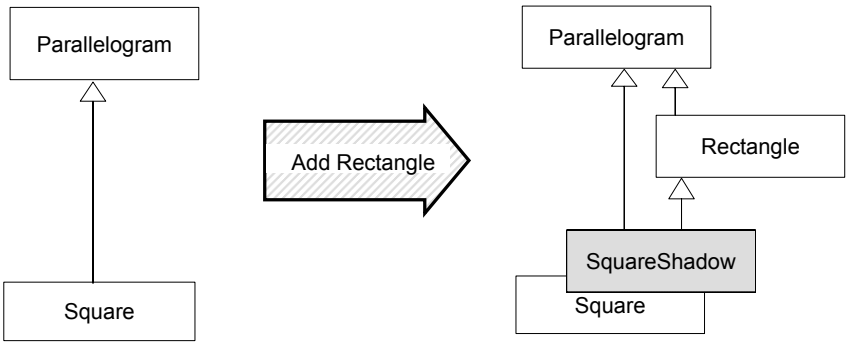

Figure 5: Interclassing via Shadows

"Frog"-like behavior. A kiss of a princess may then replace the "Frog shadow" by a "Prince shadow".

Interclassing

For a motivation of Interclassing we refer to [28], [11] (in a general context), or the previous section (in a mathematical context). Basically the idea is the insertion of a new class in an existing inheritance hierarchy. Here the assumption is that the inheritance hierarchy to be modified is in the context of an existing library that cannot be changed (for instance because of a copyright or that it has to be left unchanged for existing applications etc).

A shadow could be useful even in this situation. We illustrate this from the example in [11]: An existing hierarchy with *Parallelogram* as a parent of a class *Square* should be extended by a class *Rectangle*. In [11] the proposed solution is the introduction of a "reverse inheritance" (specialization) relationship established from the *Rectangle* to the *Square*.

A shadow – in contrast to that – could change the methods in the *Square* directly. A shadow class inherits from the Rectangle and shadows the *Square*.

Roles and Specialization

The usual way in a class based language to implement specialization is to derive a child class and make appropriate changes. However, alternatively we could also think of instantiating an object and then shadowing the object. When using the well known Vehicle/Car inheritance example (The vehicle is abstract with an abstract method move() that is implemented in Car), we could as well think of an instantiation of a vehicle object that is shadowed with a shadow that implements the move() method. That means that in fact we define during run-time the role of the vehicle as being a car.

5 Implementation

The implementation of the shadows into Java uses an approach based on both changing the compiler and adding a Java package. We follow the philosophy to keep the additions to the compiler at a minimum, and to have as much as possible of the additional functionality in Java libraries. In the following subsection we discuss the modifications to the Java compiler while in section 5 we give an overview of the Java package that works together with the compiler. Please note that the classes produced with the modified Java compiler can be run by the standard Java virtual machine.

The key strategy for the design is to implement the possibility to interfere at run-time (i.e. during program execution) at the very moment when a method is received at an object but before the method is executed.

The compiler

Our compiler is an extension of the Java compiler Jikes [21]. Jikes is a project of open source type and in that context we have added minor modifications. The modifications are "minimal" in the sense that more then 99% of the original Jikes code remained unchanged. We want to emphasize that we have no affiliation to the Jikes developer group and that our goals are different from the goals of the Jikes developer community that aims for an efficient (i.e. more efficient than Sun's javac) compiler that fully complies to the Java specification. The Jikes compiler is implemented in C++.

In terms of usage the modified Jikes compiler has an additional flag -dy that enables an alternative compilation process: If a class is compiled using the -dy flag then the method lookup mechanism works differently. Assume that an object a calls the method `foo(String str)` of an object b of type B. The default (and expected) behavior of a standard Java compiler is that the compiler seeks for a method `foo(String str)` with an appropriate signature in B. The modified Jikes compiler instead checks first if a method of the name `method_received` and appropriate signature is present. In that case the bytecode that is emitted has a method call `method_received(a, "foo", str)` that means the first argument is the calling object, the second is the name of the method, and the remaining arguments are the original arguments of the method to be called. The process is illustrated in Figure 6.

In addition each method call from the object a that does not correspond to a matching method in the target class B is also emitted as a `method_received` call in the bytecode (a Java compliant compiler would report a method not found error in that case). This behavior is necessary as a shadow may later define that method at runtime for an object of type B. Conceptually that behavior of the compiler is equivalent to the consequent replacement of any method calls by appropriate `method_received` calls whenever a method `method_received` is implemented in the

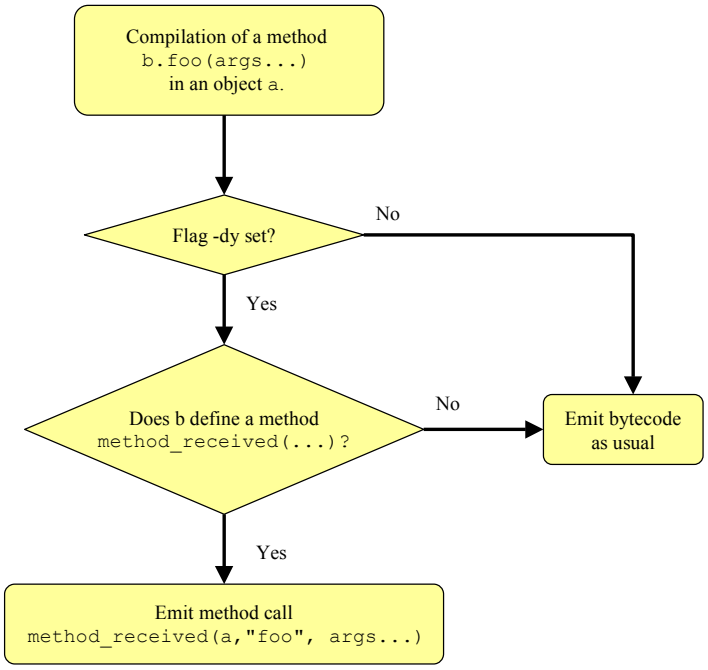

Figure 6: A simple flowchart diagram showing how the compiler redirects method calls.

target class.

This partly conflicts with the concept of proper type checking. However we emphasize that in order to identify any method calls that do not relate to an implemented method the developer has the possibility to compile the same code without the -dy flag switched on.

This is useful for example for the following application scenario: Assume that a Java package is to be implemented such that the package itself is consistent Java code, but the package should be extensible later via the shadow mechanism. Then the whole development process can be done with the original Java compiler and only at the last stage before the software is shipped the package is compiled with the -dy flag to enable the later addition of shadows to objects by the user.

The Java package

The Java package that has to be used with our modified Jikes compiler contains four classes. We emphasize here only on the features that are important for illustrating the concept. For the full application interface of these classes see [6]. In the following we have a closer look on the conceptional ideas behind these classes.

- *The class com.perisic.jshadow.Shadow:* This class defines only one method namely `query_shadow_owner()`. A shadow is the only object that can send a message directly to the shadowed object. Thus `query_shadow_owner()` returns

the shadowed object. Note, that the shadows of an object are arranged in an hierarchy, that is, if an object a is shadowed by three shadows s_1, s_2, and s_3 (and if this is the order in which the shadows are added), then a method call of shadow s_2 via `query_shadow_owner()` is received by s_1. This is consistent with the conceptual idea behind shadows. When s_2 is added to the shadowed object, it "sees" this object only via the shadow s_1.

- *The class com.perisic.jshadow.Shadowable:* Objects of this class maintain a list of shadows, so the obvious methods

  `addShadow(Shadow s)` and `removeShadow(Shadow s)`

  are implemented. Also defined is the method

  ```
  reveive_call(Object whodunnit, String methodName, Object..
  args)
  ```

  (the current version differs slightly as variable argument lists and autoboxing are not yet available in this version of the Jikes compiler). The first argument `whodunnit` is the (reference to the) object that did the method call. The second argument is the name of the method that should have been called in the first class. The remaining arguments are the arguments of the original method call.

  The behavior of this method then is that in the case that no shadow is added to `this` (i.e. to the object that received the call), the appropriate method is looked up following the Java specification (cf. [19]). Note that knowledge of the name of the method and the type of the arguments is sufficient to determine which method is to be called in this object. If the object has been shadowed by one or more shadows, the appropriate method is first looked up in the shadows and will be executed there.

- *The class com.perisic.jshadow.Util:* The (static) methods of this utility class (that cannot be instantiated), are the methods `addShadow(java.lang.Class owner, java.lang.Class shadow)` and `removeShadow(java.lang.Class owner, java.lang.Class shadow)`. Shadowing a class `owner` by a class `shadow` means that an instance of `shadow` is added whenever an `owner` object is instantiated.

- *The class com.perisic.jshadow.ShadowException:* This is the generic exception that is thrown whenever an error is happening, e.g. when the method is not defined in the object. A full conceptually consistent development of a hierarchy of exceptions is currently work in progress.

6 Conclusion

From the preceding discussion it is clear that a consideration of inheritance purely from a Java implementation perspective is a potentially conceptually self-limiting form of intellectual enquiry, though unquestionably highly pragmatic and vocationally useful. It is of course possible that mainstream languages will eventually evolve so as to directly support dynamic inheritance, generalization, and renaming or otherwise provide more explicit support for more abstract mathematical structures such as groups, rings, or vector spaces. However in the short-term it is perhaps more realistic to envisage that those seeking to explore the close conceptual cross-fertilization between pure mathematics and the object oriented paradigm will seek to supplement the standard Java "diet" through the creation of tailor made packages that adopt an overtly axiomatic vision. In any event it is clear that by only considering the types of inheritance that a particular programming language actually happens to support, mathematical developers will fail to appreciate the wider (conceptual) picture. Equally, those who consider their main interest to be in software design and development may have much to gain intellectually by expanding their knowledge of topics such as inheritance by exploring beyond the features that happen to be supported by any particular programming language.

We have seen that shadows provide a useful tool also in an number of application areas outside of mathematics. Moreover it would provide a *unified* approach for such a diversity of applications as deprecating methods, prototyping, reclassification, interclassing, and specialization. Especially the first two application areas as deprecating and prototyping may provide a useful tool to link XP (extreme programming) to Object Oriented concepts.

The proposed concept of shadows, together with the presented solution in Java, is non-intrusive in so far that the user, when compiling the code has a choice between the compilation following the Java specification and compilation that enables shadows. We have shown that using double compilation, first without the flag to allow proper type checking, then with flag to enable shadows results in a Java library that is fully Java compliant but allows the extension of this class library by the user via reclassification and interclassing. As this is a new view towards extensibility of software packages we think that shadows should be considered as a central, first class feature in the development of future programming languages.

Our contention is that through using shadows and by supporting new dynamic forms of inheritance within mainstream language implementations, it is possible to more easily and naturally model and design software that embodies mathematical abstractions (hence natural ways of conceptualizing and reasoning) with constructs such as rings, groups etc. More subtly, we are of course also suggesting that with the use of these extensions to the standard object orientation model the synergies between the subjects can be better exploited. Indeed, it may be the case that through the use of these techniques mathematicians may indeed become "better" more natural Java programmers (since artificial language constraints will have been removed).

Equally Computer Scientists will have become more familiar with and hence more confident handling pure mathematics constructs, and hence have acquired greater mathematical maturity. Then indeed we may well have a marriage made in heaven in which the true synergies between these subject areas are fully exploited.

REFERENCES

[1] D. Acnona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca. *A type preserving translation of Fickle into Java*, Electronic Notes in Theoretical Computer Science 62 (2001). Available at: http://www.elsevier.nl/locate/entcs/volume62.html

[2] K. Beck, *Extreme Programming Explained*, Addison-Wesley 2000.

[3] C. Chambers. *Predicate classes*, in: *Proceedings of the ECOOP'93*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, Kaiserslautern, Germany, July 1993.

[4] C. Chambers. *The Cecil Language: Specification & Rationale*, avialable at: http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html.

[5] M. Conrad. *com.perisic.ring – A Java package for multivariate polynomials*, http://ring.perisic.com.

[6] M. Conrad. *Implementing a Java shadow using a jikes extension*. Available at: http://www.perisic.com/shadow/jshadow.

[7] M. Conrad, T. French. *Exploring the synergies between the object oriented paradigm and mathematics: a Java led approach*, International Journal of Mathematical Education in Science and Technology, Volume 35, Number 5 / September-October 2004, pp. 733–742.

[8] M. Conrad, T. French, C. Maple. *Object shadowing - a Key Concept for a Modern Programming Language*, Proc of 2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity, Workshop at ECOOP 2004. Available at: http://prog.vub.ac.be/∼wdmeuter/PostJava04.

[9] M. Conrad, T. French, C. Maple, S. Pott. *Mathematical Use Cases Lead Naturally to Non-Standard Inheritance Relationships* Proc. of MASPEGHI, Workshop at ECOOP 2004. Available at: http://www.i3s.unice.fr/maspeghi2004

[10] J. Coplien. *Advanced C++ programming styles and idioms*, Addison-Wesley 1992.

[11] P. Crescenzo, P. Lahire. *Using Both Specialisation and Generalisation in a Programming Language: Why and How?* Advances in Object-Oriented Information Sytems, OOIS 2002 Workshops, Montpellier, pages 64–73, September 2002.

[12] T. Daly et. al. *Axiom Computer Algebra System*, http://savannah.nongnu.org/projects/axiom.

[13] *The Darwin Project*, http://javalab.iai.uni-bonn.de/research/darwin.

[14] D. Dench (2003) *eXtreme Programming (XP) with Java and Jython in the Classroom*, Procs JICC-7 Conference (Java in the Computing Curriculum Conference-7, London Metropolitan University. Available at: http://www.ics.ltsn.ac.uk/pub/jicc7

[15] *Documentation of the Shadow function*, in: http://www.lysator.liu.se/mud/MudOS-doc/efuns/system/shadow.html

[16] F. Damiani, M. Dezani-Ciancaglini, P. Giavinni *Re-classification and Multi-thrading: Fickle$_{MT}$*, *SAC 2004*, pp. 1297–1304.

[17] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini and P. Giannini. *Fickle: Dynamic object re-classification*, in: *ECOOP'01*, LNCS **2072** (2001), pp. 130–149.

[18] M. Essén, S Janson, L. Peng and J. Xiao, *Q-spaces of several real variables*, Indiana University Mathematics Journal, vol 49, no 2(2000), 575 – 615

[19] J. Gosling, B. Joy, G. Steele, G. Bracha. *Section 15.12.2.2: Choose the Most Specific Method*, in: *The Java Language Specification, Second Edition*, 2000.

[20] J. Hylton. *Introduction to Object-Oriented Programming in Python (Outline),* http://www.python.org/j̃eremy/tutorial/outline.html, Januar 2000.

[21] *IBM Jikes Compiler for the Java Language.* Available at: http://sourceforge.net/projects/jikes/

[22] G. Kniesel. *Darwin & Lava - Object-based Dynamic Inheritance ... in Java*, Poster presentation at ECOOP 2002.

[23] S. Lang. *Algebra*, third ed., Addison-Wesley, 1993.

[24] C. Larman. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design*, Prentice Hall, 2001.

[25] B. Meyer. *Overloading vs. Object Methodology*, Journal of Object-Oriented Programming, October/November 2001.

[26] The MuPAD Research Group. *MuPAD – The Open Computer Algebra System,* http://www.mupad.de.

[27] L. Pensjö. *LPC*. Documentation available at: http://www.lysator.liu.se/mud/lpc.html

[28] P. Rapicault, A. Napoli. *Evolution d'une hirarchie de classes par interclassement.* In: LMO'2001, Hermes Sc. Pub. *"L'objet"*, vol. 7 - no. 1–2/2001.

[29] The Self Group. *Self*, http://research.sun.com/research/self

[30] H. Triebel, *Theory of Function Spaces*, Monographs in Mathematics, vol 78, Birkhäuser Verlag Basel, 1983

[31] Waterloo Maple Inc. *Maple* http://www.maplesoft.com

[32] R. Wikh, *LPC*, available at: http://genesis.cs.chalmers.se/coding/lpcdoc/lpc.html (last update 2003)

[33] Wolfram Research. *Mathematica*, http://www.wolfram.com.