

# Inheritance is Specialisation

## Position paper for the Inheritance workshop of ECOOP'02

Mads Torgersen

Computer Science Department  
University of Aarhus  
Aabogade 34, Aarhus, Denmark  
madst@daimi.au.dk

**Abstract.** How can we get a simpler but much more general subclass construct? This position paper takes a “specialisationist” approach to inheritance. Old SIMULA virtues are restored to prominence, but boiled with new unificational ingredients to obtain a substrate of specialisation.

Ever since the advent of Smalltalk there has been a strong tension in the object-oriented community between two opposing views of the role of inheritance: as an incremental modification mechanism or as a vehicle for conceptual modelling. Madsen [5] and many others characterise the two approaches as the “American” and “Scandinavian” schools, respectively. Nowadays, such a geographical terminology hardly remains valid (if ever it was), but the tension remains: should we strive for maximal flexibility of inheritance, to improve the possibilities for later unanticipated reuse, or should we attend above all to the conceptual integrity of the subclassing mechanism to aid the use of the programmers’ intuition in the development process.

Here we shall adopt a different terminology, focusing instead of the primary role of subclassing in the two views, and thus refer to *modificationism* and *specialisationism*: the former view would in principle endorse a description of oranges by inheritance from apples (sometimes called “sideways inheritance” because it does not obey the structure of a classification hierarchy), the latter would allow these to be related only through a common superclass.

The line of argument in this position paper will be as follows:

- Modificationism is loosing out. Its basic assumptions do not hold, and the main benefits of object-orientation in all levels of software development derive from the core principles of specialisationism
- Therefore language designers do not have to pay special heed to modificationist needs, but can concentrate on improving inheritance as a specialisation construct
- This idea is acted out in a very simple and general programming language, RUNE, which derives its simplicity from abandoning modificationist ambitions
- It is concluded that you can get much nicer languages by sticking to a specialisation view of inheritance.

To avoid confusion, we use the term “modify” to signify incremental modification of classes through inheritance, while “change” denotes actual changes to the source code of a class.

## 1 The Demise of Modificationism

The golden modificationist vision of object-orientation maintained that programmers about to embark on writing a new class would save a lot of work by inheriting from existing classes sufficiently similar to what was desired. In this way, you don’t need to rewrite the base class, and you get changes and bug fixes for free. This has not happened much in practice, and for many good reasons.

If the classes to reuse belong to some other project or organisation they may:

- not be possible to find
- not be modifiable in exactly the desired way
- change unexpectedly once we start depending on them (the fragile base class problem).

If the class to reuse is developed in the same project there is usually no good reason not to change it: better for the project as a whole to factor out a common superclass: you get a better overall class structure, and avoid unnecessary interdependencies of the code.

Thus, in practice there is little use for modificationist inheritance in modern object-oriented software development.

## 2 The Benefits of Specialisationism

The specialisationism view has proven its beneficial effect on software development in a number of ways.

### 2.1 Conceptual clarity

The classical argument for specialisationism (see e.g., [8]) is conceptual: Classes represent concepts, and subclassing represents conceptual specialisation. Thus, “Apple” specialises “Fruit”, because any apple is a fruit, and has all the properties required to be worthy of the term “Fruit”. Insistence on this intimate relationship of programming and classification opens up the door to the programmers’ intuition and aids them greatly in the design and structuring of programs. In the words of Kristen Nygaard, “to program is to understand”.

In typed object-oriented programming languages, this realisation is reflected in the type system as *subtyping*: classes represent types, and subclasses are subtypes. Thus, inheritance is constrained by the commitment of subclasses not to contradict their superclasses.

Even in untyped systems, however, a conceptual approach to subclassing has the desirable effect that you get to depend only on changes that really should affect you. The concept of “apple” should be affected by changes to “fruit”, but not to “orange”.

### 2.2 Designing for change

Closely related to the conceptual view, although using quite a different terminology, is the design philosophy embodied in the notion of *design patterns*, and very well described in the “GoF” book [3]. Here the focus is on black-box over white-box reuse: composition is a much better vehicle for reuse than inheritance, because it minimises the dependencies between classes, making them more robust towards changes in other code.

In the GoF book, subclassing is used for polymorphism: Abstract superclasses describe the commonalities among different related concrete classes, freeing clients from dependencies on particular implementation details that are none of their business.

Frameworks build on a similar philosophy, providing large amounts of concrete functionality operating on abstract classes. Concrete subclasses of these are provided by the individual framework user, so independence of their implementation details is essential.

### 2.3 Refactoring

Specialisationism is often criticised for requiring too much foresight: how should you know in advance whether you will later be needing an abstract superclass of “apple”? You can’t, but due to the reasons above, sideways inheritance will at best help you temporarily, until the penalty of undesired dependency starts to hit you. Thus, whichever way you turn it, code will have to be changed frequently, and modern software development practices address this not as a failure of foresight but a natural evolutionary component of the development process. As an example, Extreme Programming [1] encourages constant change at all levels of the development process, including refactoring of code to produce common superclasses whenever necessary.

This does not exclude the development of durable, reusable abstractions, but suggests that they are best designed as an emergent result of concrete use.

## 3 Specialisationist Language Design

To obtain the full benefits of specialisationism the following points appear crucial, at least in typed languages.

### Classes should be types

In a specialisationist world classes represent concepts. So do types. A separation of the two appears not only confusing, but pointless: Why would we need two independent descriptions of apples?

This does not imply that there can be no other types than classes - whether or not “everything is an object” as in Smalltalk is a separate discussion outside the scope of this paper.

### Subclasses should be subtypes

Although this is rarely observed, subtyping is a natural counterpart to conceptual specialisation: Both hinge on the recognition of common properties among a heterogeneous set of entities. Modificationist subclassing might therefore not give rise to subtyping; this is the case e.g., in Sather [10] and O’Caml [4].

Specialisationism insists that subclasses should be subtypes, hence inheritance mechanisms must be limited to ensure that this will be the case.

This does not imply that subtyping should *only* apply to classes, not other kinds of types, or that subclasses should be the *only* subtypes of classes.

### Typing should be static and modular

If a subtype relationship is too complex to be automatically verified on the basis of local information, it is probably also too weird for a programmer to make good use of. Thus, implicit runtime type checks as in BETA[6], or non-modular type checking as in (idealised) Eiffel [9] are not just technically undesirable: they are also a signal of half solutions, offering a certain flexibility (in both cases related to covariance), but not the means to control it.

### Overriding is not essential

Overriding of concrete methods is a cornerstone of the modificationist desire to modify the behaviour of an already-concrete class. To the specialisationist, the important thing is to implement *abstract* methods defined in common superclasses. In [3] only a single design pattern (Decorator) uses overriding of a concrete method, and not in a manner essential to the function of the pattern.

This raises the question whether overriding (and the associated issues of method combination, multiple inheritance conflicts etc.) is a worthwhile construct at all. BETA disallows conventional overriding for conceptual reasons, but does allow incremental augmentation of methods with the `inner` construct.

I would argue that there is still a more modest role for overriding to play even in a specialisationist setting; to express default implementations. In some cases a very large number of subclasses would want the same implementation of a given method, and some sort of overriding would therefore be convenient. But restricted versions or completely different solutions might serve this purpose - conventional overriding is not essential.

## 4 A Unified Approach to Specialisation

The RUNE model [12] is an attempt at an extreme unification of specialisation. It tries to make classes play as fundamental a role as functions in functional languages, and to be as lightweight and versatile, only of course, in a specialisationist object-oriented fashion.

The design of RUNE rests on the following principles of unification:

**Only one abstraction mechanism:** Classes are the *only* abstraction mechanism - there are no separate notions of procedures or methods, type parameterisation or modules.

**Only one kind of entity:** All entities, including objects, classes, procedures, types and primitive values, are first class citizens of the language. They have types, which are all subtypes of a built-in “type of all things”, `Value`.

**Only one kind of names:** Attributes are the only way to declare names of values, and thus act not only as fields and methods of classes, but also as the parameters, locals and results of methods, as type parameters and as class and type declarations.

**Only one kind of specialisation:** Parameters are expressed as abstract attributes which get “passed” by subclassing. Thus, functional specialisation (parameter passing) and object-oriented specialisation (subclassing) are the same thing in RUNE.

**Only one kind of generation:** Class instantiation generates both substance (new objects) and activity (constructing attributes) and is therefore used both for object creation and procedure invocation.

To show some of the ideas of RUNE here is a crash view of a simple class declaration highlighting some of the basic notions of the language:

```

Counter: = {
  contents: IntBox = !IntBox;
  add: = {
    arg: Int;
    result: Int = contents.get()+arg;
    doIt: Value = contents.put(result);
  };
  inc: = add { arg = 1; };
  dec: = add { arg = -1; };
};
myCounter: = !Counter;
myResult: = (! (myCounter.add { arg = 5; })).result;

```

Classes are expressions containing an optional superclass and a body in curly braces. In the above example, `Counter`, `add`, `inc` and `dec` are all declared to be classes, `add` being the superclass of `inc` and `dec`. The last line contains an anonymous class having `myCounter.add` as a superclass.

A class body contains declarations and modifications of attributes. The declaration of `contents` has a type *constraint* `IntBox` and a *constructor* `!IntBox`. The constraint describes what kind of value the attribute holds, and the constructor (similar to a Java initialiser) how the attribute is to get its value when the containing class is instantiated. The declaration is *concrete*, because it contains a constructor, otherwise it would be *abstract*, causing the enclosing class to be also abstract.

The declarations of `add`, `inc` and `dec` are also concrete. The omission of an explicit constraint means that it is implicitly derived as the most specific statically known type of the constructor. The type system includes singleton types, so in this case, the constraint becomes the singleton type containing the exact value provided by the constructor.

All attributes of `Counter` are thus concrete. This makes the class itself concrete, so that it can be instantiated. The instantiation operator ‘!’ creates a new instance of a class. Thus when `!Counter` is executed in the next-to-last line, the `contents` attribute of the resulting object will be initialised with a fresh instance of `IntBox`.

Attributes are immutable. In order to provide state, special built-in classes are used to create objects that are mutable boxes with `get` and `put` operations. `IntBox` is such a built-in class, which is not implementable in the language itself. Other examples of built-in classes would include synchronisation entities and interfaces to the operating system.

The `add` declaration is an example of a class used as a procedure. It has an abstract attribute `arg`, which may later be modified in a subclass to become concrete. When instantiated, the `result` attribute becomes the sum of `arg` and the current value of `contents`, whereas `doIt`, which is there only for the side effect, updates `contents` with the new value.

`inc` specialises `add`, modifying `arg` so that it becomes concrete, with a constructor yielding the value 1. In the absence of overriding it may be no further modified. Now `inc` is a concrete class, and may be instantiated for the effect of increasing the `contents` with one. the generated instance of `inc` may be kept, and the value of its `result` attribute extracted as a return value.

Thus, all in all, a function call is described using subclassing for parameter passing, instantiation for execution and attribute access for passing of return values. The last line of the example combines these steps in one statement. The syntax is clumsy, because all steps of a method invocation are explicit. It is an obvious move to introduce syntactic sugar for declaration and invocation of methods, the latter being exemplified by the calls to the `get` and `put` methods of `contents`. But this should not prevent them from being used in the general sense of classes, e.g., by providing multiple parameters gradually, through incremental specialisation, or by holding on to the “invocation objects” to extract multiple results or simply to keep it as a log record of the operation.

#### 4.1 Types for classes

Using these mechanisms we can express abstractness of attributes, and we can describe functions as classes, but to get virtual methods we need to be able to give proper type constraints to our abstract attributes describing the invocation “signature” of the classes they are to be bound to, so that they can be meaningfully called by clients.

The signature of an abstract method, or the type of a function in a functional languages for that matter, describes two things: what kind of thing does it return, and what must we provide when we call it. When using classes as functions in our manner this boils down to: what is it a subclass of, and which abstract attributes remain to be implemented to make it concrete. In RUNE, signatures have the form  $[ x_1, \dots, x_n ] C$ , where  $C$  is a class and the

$x_i$  are attributes of  $C$ . Intuitively, the signature ranges over all subclasses of  $C$  (which we call the *template* of the signature) where the  $x_i$  (which we call the *parameters*) are abstract while all other attributes (which we call the *results*) are concrete. When specialised with constructors for all the abstract  $x_i$  attributes (the parameters), any class belonging to the signature will therefore become concrete, so that it can be instantiated.

As an example we could define a common superclass `IntToInt` for functions from `Int` to `Int`:

```
IntToInt: = {
  arg: Int;
  result: Int;
};
```

This can be used to define a more abstract counter class where the representation of its state is not decided, and the `add` operation therefore abstract:

```
AbsCounter: = {
  add: [ arg ]IntToInt;
  inc: = add { arg = 1; };
  dec: = add { arg = -1; };
};
```

Even though `add` is abstract, `inc` and `dec` can still specialise it in the way they do, because they know by the signature that the `arg` attribute is abstract and constrained by `Int`. Also, `add` may be called on any object of the type `AbsCounter`, because it is known that any concrete subclass will provide a suitable implementation in the constructor for the `add` attribute. Thus, `add` is really a virtual method, which can be called by clients and implemented by subclasses.

We can now implement a concrete `Counter` class with the same meaning as that of the previous section by providing an implementation for `add` that matches the signature given for it in `AbsCounter`:

```
Counter: = AbsCounter {
  contents: IntBox = !IntBox;
  add = {
    result = contents.get() + arg;
    doIt: Value = contents.put(result);
  };
};
```

The implementation of `add` fulfils the inherited constraint for it, because it is a subclass of `IntToInt` which is abstract in `arg` and concrete in all other attributes (`result` and `doIt` in this case).

## 4.2 Covariance and intensional subtyping

Because attributes are immutable, it is well known that covariance is safe. We therefore extend the modification model so that an abstract attribute may be modified with a stronger type constraint. Thus, instead of the direct declaration of the `IntToInt` class above we can declare a general `Function` class, and describe `IntToInt` as a subclass with tightened constraints:

```
Expression: = { result: Value; };
Function: = Expression { arg: Value; };
IntToInt: = Function { arg:: Int; result:: Int; };
```

With this extension it would be nice if we did not have to explicitly declare every specialisation of `Function` for specific pairs of parameter and result types. `RUNE` has a more structural subtyping rule (which we call *intensional subtyping*), allowing classes to be subtypes even when they are not declared subclasses, if their type constraints fit. We can therefore declare the `add` method of `AbsCounter` above with an anonymous subclass of `Function`:

```
add: [ arg ]Function{arg:: Int; result:: Int};
```

Because a later concrete modification of `add` can still fit this type structurally. Thus, there is no need for an explicit declaration of an `IntToInt` class.

### 4.3 Higher order types

Types are also values in RUNE, so that type parameters and attributes may be expressed in just the same way as other parameters and attributes. Again we need to define what are the types of types. RUNE has two type constructors for this purpose: *subtype* and *supertype bounds*. For any type  $T$ , the subtype bound  $<T$  ranges over all subtypes of  $T$ , and the supertype bound  $>T$  ranges over all supertypes of  $T$ .

Syntactically this is all there is to it, but combined with the covariance and intensional subtyping of the previous section, these metatypes provide for a polymorphic expressiveness that normally requires a significantly larger syntactic and semantic framework. In particular, both universal and existential bounded quantification may be expressed. Here we shall constrain ourselves to an example of simple genericity:

```
Container: = {  
  T: <Value;  
  put: [ arg ]Function{arg:: T};  
  get: [ ]Expression{result:: T};  
};  
Stack: = Container{...};  
myContainer: Container{T:: = Int} = !Stack{T:: = Int};
```

This scheme not only allows `Stack{T:: = Int}` to be a structural subtype of `Container{T:: = Int}`, as in parameterised class approaches such as GJ [2], but also allows `Container` to act as a common supertype for all its type instantiations, like BETA's virtual types [7]. Genericity in RUNE therefore corresponds closely to the model of [11], which unifies the two kinds of genericity.

## 5 Conclusion

By renouncing any ambition of modificationist expressiveness, we have been able to create in RUNE a highly unified and expressive inheritance mechanism dealing with all the specialisation needs of a modern programming language. While the RUNE model is by no means a polished and final language, it shows that there is much still to discover in object-oriented language design: but to gain something new, sometimes one has to throw out something old. I believe it is time to turn from the dead end of sideways inheritance and go for more lightweight, general-purpose specialisation-oriented class mechanisms.

## References

1. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
2. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Object Oriented Programming: Systems, Languages and Applications*, Vancouver, BC, Oct. 1998. OOPSLA98, ACM Press. Craig Chambers, editor.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Designs*. Addison-Wesley, 1994.
4. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system. Documentation and user's manual*. INRIA, 2000.
5. O. L. Madsen. Open issues in object-oriented programming – a Scandinavian perspective. *Software-Practice and Experience*, 25(S4), Dec. 1995.
6. O. L. Madsen, B. Magnusson, and B. Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Object Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming*, Ottawa, Canada, Oct. 1990. OOPSLA/ECOOP90, ACM Press. Norman K. Meyrowitz, editor.
7. O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Object Oriented Programming: Systems, Languages and Applications*, New Orleans, Louisiana, Oct. 1989. OOPSLA89, ACM Press. Norman K. Meyrowitz, editor.

8. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.
9. B. Meyer. *Eiffel: The Language*. Interactive Software Engineering, 2.2 edition, Dec. 1989.
10. D. Stoutamire and S. Omohundro. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, Aug. 1996.
11. K. K. Thorup and M. Torgersen. Unifying genericity. In *European Conference on Object-Oriented Programming*, pages 186–204, Lisbon, Portugal, June 1999. ECOOP99, LNCS 1628, Springer Verlag. Rachid Guerraoui, editor.
12. M. Torgersen. *Unifying Abstractions*. PhD thesis, Computer Science Department, University of Aarhus, bogade 34, DK-8200 rhus N, Sept. 2001.