

Computational Abstraction Steps

Lone Leth Thomsen, Bent Thomsen, and Kurt Nørmark,
Department of Computer Science, Aalborg University, Denmark

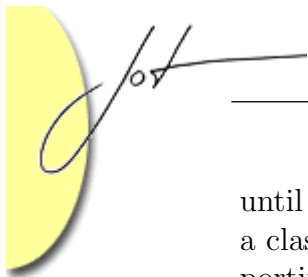
In this paper we discuss computational *abstraction steps* as a way to create class abstractions from concrete objects, and from examples. Computational abstraction steps are regarded as symmetric counterparts to computational *concretisation steps*, which are well-known in terms of function calls and class instantiations. Our teaching experience shows that many novice programmers find it difficult to write programs that use abstractions which materialise to concrete objects much later in the development process. As the contribution of this paper we propose to initiate a programming process by creating or capturing concrete values, objects, or actions. As the next step, some of these are lifted to a higher level by computational means. In the object-oriented paradigm the target of such steps is classes. We hypothesise that the proposed approach primarily will be beneficial to novice programmers or during the exploratory phases of a program development process. In some specific niches it is also expected that our approach will be a help to professional programmers.

1 INTRODUCTION

Object-oriented programming is currently the de-facto industrial programming methodology. Its widespread use can be accredited to the main structuring mechanisms of the paradigm. Objects and classes allow programmers to deal with the ever increasing complexity of software systems by exploiting the human mind's natural capability for thinking in those terms.

Today many educators use an *objects first approach* when teaching object-oriented programming. According to Gary Pollice: "Computer science students who learn object-oriented concepts right from the start have an easier time applying them to software development projects than programmers who are steeped in structural programming techniques" [20]. The objects first approach starts with object-oriented analysis and design which emphasises partitioning system behaviour into small, cohesive parts, and composing the final solution by making the parts cooperate. Modern object-oriented programming languages support this notion through classes, instantiated to create objects, that interact by calling methods on each other. A class is therefore an encapsulation entity which defines a set of objects together with operations on these. A class can be instantiated, hereby creating an object. Class instantiation represents a step from an abstract notion to a concrete example. In this paper a class instantiation will be seen as a *concretisation step*.

Many proponents of the objects first approach encourage students to create objects and interact with them in an experimental and iterative development approach



until the students reach an understanding of the implementation that is suitable for a class definition. The BlueJ development environment¹ makes an attempt at supporting such a development process [9]. In contrast, mainstream object-oriented languages such as Java only support interacting with objects that have been created by instantiating classes. Thus in reality, the objects first approach should be called the *class first approach* as the programmer has to identify the needed set of classes before instantiating such classes to objects.

As reported by Hadar and Leron [13], even experienced object-oriented programmers can get the direction of abstraction and concretisation wrong. Our experience [23, 18] from teaching introductory programming courses shows that many novice programmers find the characteristics of going from classes at a high abstraction level to concrete objects via inheritance and object creation extremely counter intuitive. We believe this to be especially so because going in the opposite direction, from objects to classes, is either not supported or only supported in limited and cumbersome ways by mainstream object-oriented programming languages.

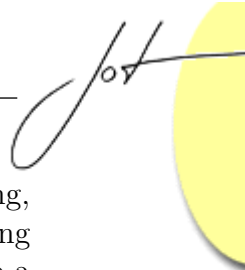
In this paper we explore *abstraction steps*. Abstraction steps can be seen as the symmetric counterpart to existing and well-known concretisation steps, such as class instantiation. In particular we deal with *computational abstraction steps* - carried out during program execution - as opposed to static abstraction steps, which are handled entirely in the source program. We briefly discuss static abstraction steps and syntactic support for handling classes as first class objects in section 5.

Thus, an abstraction step brings us from a concrete example to an abstraction which is more general than the example. The concretisation steps represented by class instantiations are unidirectional operations. The inverse operations - from concrete examples to abstract notions - are not always easy to deal with, in part because of ambiguities.

Segapeli and Cavarero [21] describe very succinctly three levels of generalisation from examples in object-oriented design. The first level is simple generalisation, which they describe as the classical way of turning an example object into a class. The second level is abstraction, which they describe as an upper level of generalisation, where we keep common features, but do away with features not common to a set of examples. This can support reuse as this process may be repeated to create increasingly abstract classes, eventually leading to classes that cannot be instantiated, but only reused via inheritance. The third level is a meta level, which introduces meta classes or classes of classes. A meta class may contain behaviour or features common to several classes.

To some extent the notion of computational abstraction steps can bridge the rift between prototype based and class based object-oriented programming. Proponents of prototype based object-oriented programming have forcefully argued, that incremental learning of concepts is facilitated by prototypes and that prototypes are

¹The BlueJ development environment is now integrated with the NetBeans programming environment [2].



better suited for expressing defaults than class based object-oriented programming, see e.g. [15]. With computational abstraction steps we can initiate a programming process akin to prototype based object-oriented programming and then switch to a class based object-oriented programming style when sufficient instances of a concept have been created by the programmer to justify a generalisation characterising the set of all instances of a given concept.

In the remaining parts of this paper we will introduce abstraction steps in the object-oriented paradigm. In section 2 we introduce abstraction steps by means of an example in an experimental programming language, called ASL (Abstraction Step Language). Section 3 contains an overview of the primitives in ASL. In the setup of this paper ASL is based on dynamic typing as known from e.g. Smalltalk, Python, and Ruby. We will introduce a number of *dynamic programming operations* that allow us to elaborate and refine classes. In Section 4 we elaborate the example in a number of existing programming languages. Or more precisely, we study how far the example can be described in some existing languages. We will briefly discuss static abstraction steps and syntactic support for handling classes as first class objects in section 5. In section 6 we will look at using computational abstraction steps beyond the learning process, i.e. in situations where professional programmers may use the approach to solve real problems. Finally, in section 7 we draw some conclusions and point to future directions of work.

2 A CLASS ABSTRACTION STEP SCENARIO

We start with a sample scenario which creates classes via computational abstraction steps. We develop the scenario in an experimental programming language, called ASL (Abstraction Step Language). There exists an implementation of ASL with a slightly different syntax than shown below. In appendix A we reproduce the example on the basis of the implemented variant of ASL.

The motivation behind this approach is - in part - pedagogical. It is our experience that many novice programmers have a hard time understanding the abstractions behind concrete objects. In earlier papers [23, 18] we have documented our experiences from teaching introductory programming and listed some of the challenges of this teaching task, e.g. getting the students to “think” in a way suited for a specific programming paradigm. Based on this experience we hypothesise that it is beneficial to start the development process with the creation of a number of concrete objects. Afterwards, classes can be introduced via abstraction steps. Methods can be added to both objects and classes.

We start by defining a concrete object which represents a student in terms of first name, last name, age, and the area of study:

```
james =  
  [[firstName => "James", lastName =>"Nielsen", age => 25, area = Math]]
```

The notation `[[...]]` provides for *manifest input notation* of objects, similar to the notation provided by object initialisers in some object-oriented programming languages such as C# [8]. The object is formed by simple aggregation of named fields. In some programming languages such an aggregation can be dealt with as an associative array. The new student object is bound to the name `james` by use of a conventional name binding form.

We can extract information from the sample object. As an example we concatenate the first name and the last name, and bind the result to the variable `fullName`:

```
fullName = james.firstName + " " + james.lastName
```

At this stage of the development it is natural to enrich the object with a method that calculates the full name:²

```
james.AddMember(FullName,
                lambda()(this.firstName + " " + this.lastName))
```

The added method is only applicable for a single object - the object referred to by the variable `james`.

The first actual parameter of the dynamic programming operation `AddMember` is the name of the method. The second actual parameter is a function, which is *refurnished* as a method in the object to which it is added. Methods are obtained from ordinary functions, which refer to the current object via the reserved name `this`. The refurbishing of a function as a method consists - in general - of a binding of `this` to the hosting object. The static environment of the function is not affected by this refurbishing. If the function is applied outside the context of an object, `this` is bound to the current global environment.

Using the `FullName` method, the full name of `james` can be extracted and calculated more conveniently:

```
fullName = james.FullName();
```

It is natural to provide *manifest object output* notation, similar to the proposed input notation. Thus, when we query the name `james` we get the following response:

```
[[firstName = "James", lastName = "Nielsen", age = 25, area = Math,
  FullName = lambda()(this.firstName + " " + this.LastName) ]]
```

In addition to `james`, we can now create a number of other “person objects”. Some of these objects may be very similar to `james` (i.e. they would be instances of a shared class, if classes were present at this stage of the development). Other objects will be less similar, such as `mike` which is an employee rather than a student.

²We apply a coding style where the first letter of a field name is in lower case, and the first letter of a method name is in upper case.



```
mike = [[firstName => "Mike", lastName =>"Madsen", salary => 12500]]
```

This object shares the fields `firstName` and `lastName` with `james` and it introduces a new `salary` field. The `FullName` method of `james` makes sense to `mike` as well, and therefore `mike` *borrow*s it from `james` in the following way:

```
mike.AddMember(FullName, james.FullName)
```

With this member addition, both `mike` and `james` have `FullName` methods. If a number of methods need to be borrowed, they can be *bulk transferred* with the use of simple iteration (in a programmed loop).

After some initial experience with our concrete student object we are ready for an abstraction step towards students in general.

```
Student = james.Kappa()
```

The `Kappa` operation, applied on an object, infers the class of the given object. The `FullName` method of the object bound to `james` is lifted to a method in the class. The fields of the object bound to `james` become default values of forthcoming instances of class `Student`. The name `Student` is bound to an object which represents the class - a *first class class*.

The use of `Kappa` represents a *computational abstraction* step, because the class is formed dynamically in the executing program. `Kappa` is inspired by `lambda` from functional programming. We discuss `Kappa` and its relationship to `lambda` in section 5. If the language relies on static typing, the types of the fields are assumed to be inferred from the individual values of fields. The static environment of the inferred class is assumed to be the global environment.

It is possible to register existing objects as instances of an existing class. This requires, however, that the object possesses all the data fields of the class. First we register `james` as an instance of class `Student`:

```
james.AsInstanceOfClass(Student)
```

This preserves the data fields of `james`, but the `FullName` object method is taken out of `james`, because it has been lifted to class `Student`. Forthcoming instances of class `Student`, as well as the object `james`, share the `FullName` method.

Given another student object

```
oliver = [[firstName => "Oliver", lastName =>"Persson", age => 20,  
          area = Bio]]
```

we can similarly state that `oliver` is an instance of class `Student` by

```
oliver.AsInstanceOfClass(Student)
```

As an immediate benefit to `oliver`, the methods of class `Student` hereby become available to the object.

It is possible to apply concretisation steps on class `Student` in order to make additional students. In this way, the following creates yet another instance of class `Student`:

```
anne = Student.New(firstName => "Anne", age => 20)
```

The dynamic programming operation called `New` corresponds to the conventional `new` operator in object-oriented languages such as Java or C#. The `firstName` and the `age` fields of the new object are initialised from the supplied keyword parameters. The remaining fields are initialised by the default values in class `Student`, which originate from the definition of `james`. Thus, if we query the name `anne` we get

```
[[firstName = "Anne", lastName = "Nielsen", age = 20, area = Math]]
```

because the default last name of students is `"Nielsen"` and the default area of students is `Math`.

In the following steps it is natural to define additional methods in class `Student`. For the sake of the example we add a method `BornYear` which calculates the difference between this year (passed as a parameter) and the age of the current student.

```
Student.AddMember(BornYear, lambda(thisYear)(thisYear - this.age))
```

We try out the new method by evaluating the expression

```
Student.New().BornYear(2009)
```

the value of which is 1984. The expression `Student.New()` is a student object with default fields that can be tracked back to the first object, namely `james`. The age of `james` is 25, and therefore the new student is born in 1984.

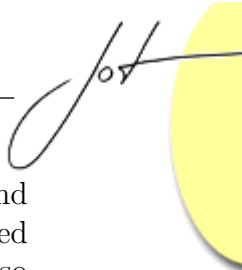
It is important to notice that the existing objects `james`, `oliver`, and `anne` also get the method `BornYear`. `james` and `oliver` were both transformed to `Student` objects, and `anne` was created as an instance of class `Student`. It is, for instance, possible to send the message `BornYear` to `oliver`:

```
oliver.BornYear(2009)
```

The result is 1989, because `oliver` is of age 20.

We may also add fields to individual objects, or to a class, using the `AddMember` operation.

```
Student.AddMember(startOfStudy, 2008)
```



Only future instances of class `Student` get the `startOfStudy` field. The second parameter of `AddMember` is the default initial value of `startOfStudy` when added to a class, or the initial value of the field when added to an object. There is also a dynamic programming operation, `DeleteMember`, which deletes a field or method from an object or class.

With the above we are able to deal with students. A few particular student exemplars are around in addition to the concept of students, represented by the `Student` class. We are now ready for yet another abstraction step, namely the generalisation of students to persons.

```
Person = Student.Generalise(firstName, lastName, FullName)
```

The parameters passed to `Generalise` are the members of `Student` that will be “elevated” to members of class `Person`. The elevated members are deleted from class `Student` when they are added to class `Person`. The objects `james`, `oliver` and `anne` are now instances of class `Student`, which is a subclass of class `Person`.

The object called `mike`, which is not “Student-like”, can be generalised to an `Employee` class.

```
Employee = mike.Kappa()
```

In turn, the class `Employee` should be generalised to class `Person` - in the same way as we generalised class `Student` to `Person`. We do not want to make a new `Person` class, however. Instead we want that `Student` and `Employee` *share a common superclass*. It is natural to obtain such a shared common superclass via *multiple generalisation* of `Student` and `Employee`. Thus, a generalisation can be formed simultaneously, relative to a number of existing specialisations. The generalisation of `Student`, as shown above, is replaced by the following:

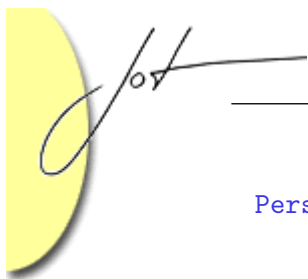
```
Person = {Student, Employee}.Generalise(firstName, lastName, FullName)
```

With this, both `Student` and `Employee` lose the given members (`firstName`, `lastName`, and `FullName`), which in turn are added to `Person`. Moreover, both `Student` and `Employee` refer to the same superclass (in the sense of reference equality). In that way, future new `Person` methods will be applicable to a broad range of existing objects (specifically, in our example, to `james`, `oliver`, `anne`, and `mike`).

We add a new data field to class `Person`.

```
Person.AddMember(sex, Female)
```

Hereby the default sex of *future instances* of class `Person` will be female. Existing instances of class `Person` will not get this new field. We can also add a method which toggles the sex of a person:



```
Person.AddMember(SexChange, lambda() (if (this.sex == Male)
                                     this.sex = Female
                                     this.sex = Male))
```

With these `Person` members in place we can play with `Person` objects in the following way.

```
axel = Person.new(firstName => "Axel")
```

Presumably, `axel` is a male person, but he inherits the default sex of a `Person` (which happens to be `Female`). We fix this problem by

```
axel.SexChange()
```

which toggles the `sex` field of `axel`.

In mainstream object-oriented programming we start by programming the classes, starting with the most general classes. Thus, the class `Person` is programmed first, and classes `Student` and `Employee` are programmed next as subclasses of `Person`. When the classes have been established it is possible to make instances of the classes. The abstraction steps, as explained above, are used in a development process which reverses the order of conventional concretisation steps.

Relative to the scenario describe above, the objects created early in the development process will be outdated after classes have been derived by using `Kappa`, and after new fields have been added to the derived classes. Along the line of Ruby we could keep track of all objects registered as instances of a given class, and via this registration facilitate a systematic update of all objects when new fields are added to the class.

So far we have mainly elaborated on computational abstraction steps as a useful pedagogical programming technique. In section 6 we briefly look at an example to demonstrate that professional programmers may also benefit from the use of abstraction steps.

3 THE ASL PRIMITIVES AND THEIR ORGANISATION

In this section we provide a systematic and categorised overview of the dynamic programming operations for working with abstraction steps. Before that, however, we will describe the overall organisation of ASL objects.

An ASL object can either be self-contained, or it can be an instance of a class. All self-contained objects share a number of methods, which are organised in a root class-object called `Object`. An object which is registered as an instance of a class may itself have methods, but it also has access to the methods obtained from the superclass chain of the class of the object.

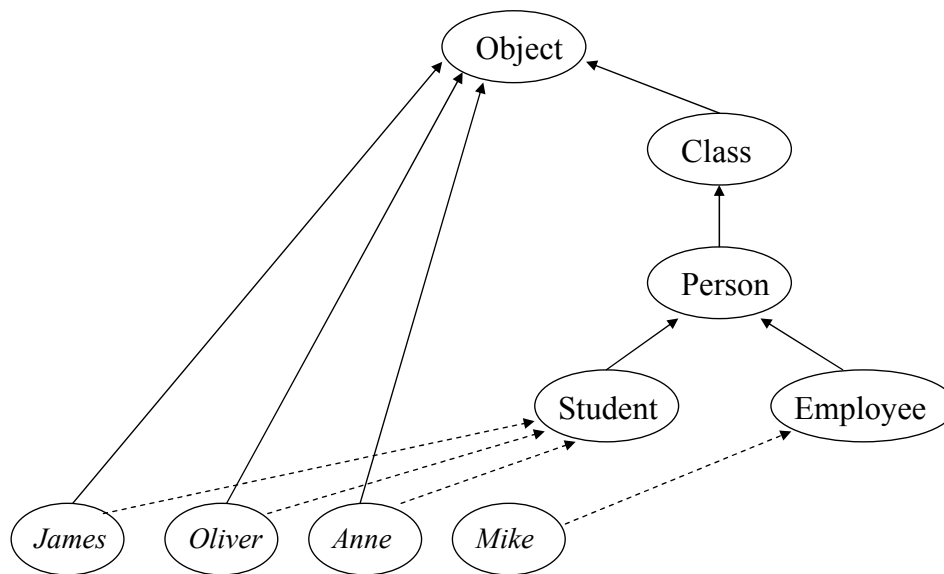


Figure 1: *The organisation of objects and class-objects in an ASL system.*

Some ASL objects are class-objects. Such objects may also be self-contained. Self-contained class-objects can contain their own methods. In addition, some class-specific methods are shared in a class-object called `Class`. Like non-class objects, self-contained class objects share the methods in `Object`. Class-objects can also be instances of classes (meta classes). Then the methods from the superclass chain of the meta class are applicable to the class-object.

Figure 1 exemplifies and summarises the object and the object-class organisation relative to the scenario described in Section 2. Solid lines represent *inheritance* and dotted lines represent the *instance of* relation. The inheritance relation is used in two different ways: (1) between class-objects in a class hierarchy, and (2) between an object and the shared behaviour in a class object.

The following object-creation operations are supported in ASL:

- `MakeObject(fieldName, fieldValue...)`
- `object.Clone()`
- `class.New(...)`

Here, and in the following, the use of the name *object* covers all objects, including class-objects. The name *class* only covers class-objects. The name *x* covers arbitrary objects and values. The object initialiser notation `[[fieldName => fieldValue, ...]]` is syntactic sugar for a call to `MakeObject(fieldName, fieldValue...)`.

Class creation is supported by these operations:

- `object.Kappa()`
- `{class, ...}.Generalise(memberName, ...)`
- `class.Specialise(memberName, memberValue, ...)`
- `class.Flatten()`

As discussed in Section 2, `Kappa` is the operation behind the abstraction step that derives a class from an existing object. `Generalise` is the abstraction step that constructs a superclass from a number of (sub)classes. `Specialise` is, in some sense, the inverse operation of `Generalise`, and as such it represents a concretisation step. The expression `C.Flatten()` returns a class which includes all the fields of `C` and its superclasses.

Member addition, deletion, and classification is facilitated by the following operations:

- `object.AddMember(memberName, memberValue)`
- `object.DeleteMember(memberName)`
- `object.AsInstanceOfClass(class)`

As discussed in Section 2 the operation `AsInstanceOfClass` registers an object as an instance of a given class.

A number of class and object introspection operations are supported:

- `object.ClassOf()`
- `class.SuperClassOf()`
- `object.FieldsArray()`

The expression `object.classOf()` navigates the *instance-of* relation, and `class.SuperClassOf()` navigates the *inheritance* relation. The operation `FieldsArray` returns an associative array of the fields in the receiving object.

Finally, the following predicates are supported:

- `object.FieldExistsInObject?(field)`
- `object.InstanceOf?(class)`
- `x.Object?()`
- `x.Kappa?()`

In addition, there is a primitive `MakeFunction` for creation of a function, which can be refurbished to a method in an object or a class if the function is added as a member by use of `AddMember`.

In the current implementation of ASL all instances of a class will have access to a new method whenever a method is added to the class with `class.AddMember`. However, new fields added to a class will only have effect on future instances, as mentioned in section 2. The reason for this is simplicity considerations and this is clearly a debatable implementation choice. An alternative implementation could add a new field to every object of the class by keeping a list of all objects of the particular class, and when a new field is added to the class, this list is iterated through and the new field added (with the use of `object.AddMember`). Obviously this is not very efficient. Another alternative could be to keep the fields, with default values, with the class. When a field is accessed and not found with the object, the field is searched for in the class hierarchy and added to the object at this point for future, more efficient access.

4 CLASS ABSTRACTION STEPS IN EXISTING PROGRAMMING LANGUAGES

So far we have described the steps and operations supporting computational abstraction in a purpose built language (ASL). We will now investigate to which degree existing languages such as Self, JavaScript, Python and C# 3.0, can be used for the purposes outlined in Section 2.

Self

Object-oriented programming without classes was pioneered in the Self project [24]. Data objects in Self consist of a sequence of slot descriptors (enclosed in bar syntax - `|...|`) followed by some optional code. The object that describes data about a person, such as `james` from Section 2, can in Self be made by

```
( | firstName = 'James'. lastName = 'Nielsen'. age = 18.  
  sex = 'Male' | )
```

Self is very flexible with respect to organising shared behaviour among objects in one or more parent objects (akin to *traits*). Instead of attempting to simulate classes as a place of shared behaviour of all instances of the class, a Self programmer will factor out the shared behaviour to another object which becomes a common parent of the objects. The abstraction step from a given person object such as `james` to “class Person” is in Self dealt with by elevating shared Person properties to a shared parent object. Thus, Self deals with class abstractions in terms of differential programming [11] - involving reorganisation of objects in multiple object inheritance hierarchies - rather than introducing a new concept (the class) at a higher level of abstraction.

JavaScript

JavaScript [3] is an object-oriented language without classes. As such it resembles the Self programming language. But as we shall see, JavaScript is less radical than Self.

In JavaScript it is possible to make an object that describes the properties of `james` introduced in the previous sections:

```
var james = {firstName: "James", lastName: "Nielsen", age: 25,
            sex: "Male"};
```

The property names may be identifiers (as shown above) or strings. It is, alternatively, possible to make an empty object and add properties programmatically:

```
var james = {};
james.firstName = "James"; james.lastName = "Nielsen";
james.age = 25; james.sex = "Male";
```

Properties, like `firstName`, `lastName`, `age`, and `sex` can be enumerated in an object, checked for existence, and individually removed.

JavaScript approaches classes with use of constructor functions. The function

```
function Person(fn, ln, a, s){
  this.firstName = fn; this.lastName = ln;
  this.age = a; this.sex = s;
}
```

is implicitly a constructor. The expression

```
james = new Person("James", "Nielsen", 25, "Male")
```

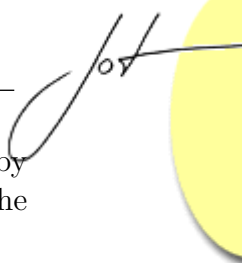
makes an empty object and runs the function `Person` on it, hereby adding the four fields. By using a special property, called `constructor`, the object remembers that the function `Person` was used to populate the object. Thus, the object “is of class `Person`”.

It is possible to add functions to classes, in the same way as data properties:

```
james.bornYear = function(){return thisYear - this.age;}
```

With this, the anonymous function becomes a method in the object. We anticipate that all `Person` objects share this method. This is easy to accommodate in JavaScript; Just substitute the lines above with

```
james.prototype.bornYear = function(){return thisYear - this.age;}
```



In a similar way to Self the prototype property points to the properties shared by all persons. It is possible to simulate class inheritance by low-level tweaking of the prototype property.

As it appears from the discussion above, JavaScript is flexible with respect to creation and manipulation of objects. However, classes are only simulated via constructor functions.

Python

To get closer to our desired treatment of classes, objects and functions, we now turn to the dynamic language Python [6].

Python does not have object syntax for defining and instantiating an object directly, but it has syntax for dictionaries. Thus we can start by defining `james` as a dictionary:

```
james = {'firstName' : 'James', 'lastName' : 'Nielsen',
         'age' : 25, 'sex' : 'M'}
```

It is relatively easy to program a function that converts a dictionary into an object (belonging to an anonymous class):

```
def nameless_class_from_dict(d):
    return type("", (object,), d)
```

The above function returns an anonymous Python type object, or rather a type object with the empty string as name, which inherits from the class `object` and from no other classes. The class will have members defined by the dictionary `d` with member names based on the dictionary keys. Furthermore, objects created from this class will have default values for members based on the associated values. We can use this function (and the fact that Python is a dynamic language) to turn `james` into an object:

```
james = nameless_class_from_dict(james)()
```

Alternatively, we could create a `james` object, step by step, and then capture its (anonymous) class, in the following way:

```
class Jamestype(object): pass

james = Jamestype()
james.firstName = 'James'
james.lastName = 'Nielsen'
james.age = 25
james.sex = 'M'
```

Note that `pass` is the Python keyword for a statement that does nothing.

We can then create a `FullName` function as follows:

```
FullName =
    lambda obj:obj.firstName + ' ' + obj.lastName
```

and add this function to the `james` object. Python just adds the new member to the object:

```
james.FullName = lambda:FullName(james)
```

The `kappa` primitive, which turns an object into a class, is easily implemented in Python as follows:

```
def kappa(d):
    return type("",(object,),d.__dict__)
```

Thus we can use `kappa` on `james` to create a `Student` class:

```
Student = kappa(james)
```

Similarly we can create a new object `oliver` in two steps; first creating an object from a dictionary, then creating an anonymous class and instantiating it to an object:

```
oliver = {'firstName' : 'Oliver', 'lastName' : 'Persson',
         'age' : 20, 'sex' : 'M'}

oliver = nameless_class_from_dict(oliver)()
```

Now `oliver` is an object (as an instance of an anonymous class), but we can assign a new class to the `oliver` object as follows:

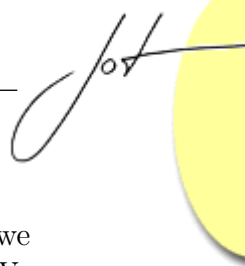
```
oliver.__class__ = Student
```

With a bit of dictionary programming, a function that generalises a class can be programmed as follows:

```
def generalise_class(c, *members):
    return
    type("",(object,),
         dict([(x,c.__dict__[x]) for x in members]))

Person =
    generalise_class(Student,'firstName','lastName','FullName')
```

We may even create a new meta class with the `generalise_class` function as one of its methods and use the traditional dot notation. Thus Python is close to being able to support directly many of the operations we seek for programmed abstraction steps, but Python is still far away from supporting all of ASL's features.



C#

Self, JavaScript and Python are dynamically typed languages. In this section we investigate how abstraction steps are supported in a statically typed language. We use C# 3.0 [8] which introduces anonymous types and local type inference.

In C# 3.0 it is possible to create an object of an anonymous reference type:

```
var james = new{
    firstName = "James", lastName = "Nielsen",
    age = 25, sex = Sex.Male
};
```

This corresponds to the creation of `james` in Section 2. The anonymous type behind this object is only compatible with the type `Object`, thus it is not possible to convert or treat it as an instance of the following named type:

```
class Person {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public Sex Sex { get; set; }
}
```

Furthermore, as the anonymous type behind the object referenced by `james` is only compatible with the type `Object`, it can only be passed to methods that take arguments of type `Object`, because the dynamic (anonymous) type of the object cannot be named in method signatures. Thus at first glance such objects with anonymous types seem rather useless.

However, using meta programming features of the namespace `System.Reflection`, it is possible to capture the type of the object referred to by `james` and instantiate other objects of the same type:

```
var jamestype = james.GetType();
var thomas = Activator.CreateInstance(jamestype, "Thomas", "Hansen",
    22, Sex.Male);
```

C# 3.0 has a notion of *extension methods* which could give the illusion of adding new members to existing classes. However, only static methods in static classes are allowed as extension methods. If we could get at the type name for our anonymous class (named `jamestype` below) we could write an extension method `bornyear` as:

```
public static class JamesTypeHelper{
    public static int bornyear(this jamestype s){
        return (DateTime.Today.Year - s.age);
    }
}
```

However, although .Net assigns the anonymous type an auto-generated name

```
<>f\_\_AnonymousType0'4[System.String, System.String, System.Int32,
    ConsoleApplication1.Sex]
```

there is no way of getting at this type at compile time, and we can only give the illusion of extending the `jamestype` class with a `bornyear` method by extending an existing type, e.g. `int`:

```
public static class JamesTypeHelper{
    public static int bornyear(this int s){
        return (DateTime.Today.Year - s);
    }
}
```

This extension method may be invoked by `james.age.bornyear()`.

As we can only get at the anonymous type behind the object referenced by `james` at run-time, we may explore creating classes and methods dynamically. This is also possible in C# 3.0, where dynamic creation of types is supported by the classes in the namespace `System.Reflection.Emit`. Using these features we may e.g. attempt to inherit from the class of `james` and add methods.

As seen from the code in Figure 2, adding constructors and methods requires a substantial Intermediate Language (IL) programming effort. In the example in Figure 2 it is necessary to create a constructor, which explicitly calls the base class constructor, i.e. the constructor of the anonymous class pointed to by `jamestype`, as this class does not have a default constructor - which is a quirk of, but a reasonable design choice for, the C# 3.0 implementation of anonymous classes. Furthermore, the code in Figure 2 will fail at runtime, as it turns out that the anonymous class pointed to by `james` is sealed and can thus not be subclassed further - again a reasonable design choice in C# 3.0 for the purpose of anonymous classes.

In conclusion, dynamic creation of types in C# takes place at a relatively low level compared to ordinary “static” program construction. Creation (emission) of constructors and methods in types is particularly cumbersome. Relative to our practical interests, as reflected in Section 2, it is worth noticing that it is not possible to remove fields or methods from existing types or classes derived from existing classes by use of `System.Reflection.Emit`. Thus the idea of class abstraction steps is not realistic in the context of C#.

5 STATIC ABSTRACTION STEPS AND FIRST CLASS CLASSES

So far we have discussed how to create and manipulate objects and classes at run-time. When creating larger programs it may be useful to have syntactic support for class objects or static abstraction steps, as a counterpart to the dynamic abstraction



```

AppDomain appDomain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName("MyDynamicAssembly");

AssemblyBuilder assemblyBuilder =
    appDomain.DefineDynamicAssembly(aname, AssemblyBuilderAccess.Run);

ModuleBuilder modBuilder = assemblyBuilder.DefineDynamicModule("DynModule");

TypeBuilder tb = modBuilder.DefineType ("kurt", TypeAttributes.Public, jamestype);

Type[] ctorParams = new[]{typeof(string),typeof(string),typeof(int),typeof(Sex)};

ConstructorBuilder c = tb.DefineConstructor(MethodAttributes.Public,
    CallingConventions.Standard,
    ctorParams);

ILGenerator gen = c.GetILGenerator();

gen.Emit(OpCodes.Ldarg_0);
ConstructorInfo baseConstr = jamestype.GetConstructor(ctorParams);
gen.Emit(OpCodes.Call, baseConstr);
gen.Emit(OpCodes.Ret);

Type t = tb.CreateType();
var kurt = Activator.CreateInstance(t, "Kurt", "Hansen", 49, Sex.Male);

```

Figure 2: Dynamically building a class which inherits from `jamestype`

steps discussed so far. We first bring classes on par with functions with respect to expressions that create classes. The counterpart to lambda expressions is called *kappa expressions*. The kappa expression

```

kappa
  classMembers
end

```

returns a value - a *class object* - which represents an anonymous class. `classMembers` is a list of named variables and method definitions which are statically encapsulated in the class object. In some sense, the idea of kappa expressions follows directly from Tennent's principle of abstraction [22], in particular section 9.6, and from the lambda expression counterpart.

Like function objects, class objects are first class objects. As such, class objects can be passed as parameters, returned as results from functions, and organised in data structures. Hence we talk about *first class classes*.

The class instantiation operator, typically called `new`, can now be applied to class objects:

```

new kappa
  classMembers
end (...)

```

The value of this expression is a (singleton) instance of the class, which is denoted by the kappa expression. In this paper we have consistently used dot-notation (as known from object-oriented programming) for application of dynamic programming operations. Thus, the conventional `new` operator becomes a dynamic programming operation called `New`, which can be applied to a class value:

```

kappa
  classMembers
end.New(...)

```

The application of the `new` operator and the `New` operation represent a *concretisation step*, from a class object to an instance of the class.

With the syntax for static abstraction steps in place, our notation for dynamic abstraction steps naturally becomes:

```

object.Kappa()

```

As discussed in section 2, the value of this expression is a class object, inferred from `object`. We assume that `object` refers to some concrete object, which has been created dynamically by a simple aggregation of a number of values/objects or has been created statically as an instance of a class. In both cases the abstraction step via the `Kappa` operation becomes the symmetric counter part to the concretisation operation via the `new` operator.

6 ABSTRACTION STEPS FOR PROFESSIONAL PROGRAMMERS

Until now we have studied abstraction steps from a pedagogical point of view. In this section we will briefly look at a couple of examples to demonstrate that professional programmers may also benefit from the use of abstraction steps.

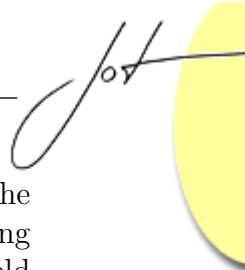
Data in text files

In a blog article about Ruby meta programming [4] it is suggested to deal with a situation where data is encountered in a line-oriented and comma-separated text file:

```

firstName, lastName, age, area
"James", "Nielsen", 25, Math
"Oliver", "Persson", 20, Bio
"Anne", "Nielsen", 20, Math
"Line", "Hansen", 17, English

```



The first line contains field names, and the following lines contain data that follow the classification given in the first line. After reading the first line, meta programming techniques can be used to establish a class (possibly anonymous) with four field names: `firstName`, `lastName`, `age`, and `area`. (If the static types of these are crucial, it is easy to extend the example with a second line that enumerates these). The remaining lines contain data which can be used in instantiations of the class from the first step. As it appears, the line organisation and comma separation represents a table with four columns and five rows (of which the initial few rows contain static information).

Alternatively, we assume that the data appears in the following form in the file:

```
firstName = "James", lastName = "Nielsen", age = 25, area= Math
firstName = "Mike", lastName = "Madsen", salary = 12500
lastName = "Persson", firstName = "Oliver", age = 20, area= Bio
salary = 20000, firstName = "Ole", lastName = "Jensen"
```

Apparently, two different kinds of person data are now mixed together, still in a line-oriented and comma-separated format. There is no information “up front” that describes the format of the rest of the file. In this situation it will be attractive to proceed as follows:

1. Instantiate four objects by use of the ASL `MakeObject` primitive.
2. Cluster the objects according to a similarity analysis, two “students” and two “employees”, and construct two classes according to this analysis.
3. Identify the fields possessed by both classes: `firstName` and `lastName`.
4. Generalise the two classes to a `Person` class.

With this approach, the *abstraction step language primitives* proposed in this paper support a natural bottom-up solution.

In case we insist on a top-down solution, we need to pass over the data twice. In the first pass, static information about field names is collected. Following that, appropriate classes can be made by using meta programming techniques (as in the Ruby-inspired example discussed above). In the second pass the classes can be instantiated according to the actual form of each of the lines.

Singleton

The Singleton design pattern [12] can be used to guarantee that multiple instances of some class `C` cannot be created. In ASL it is possible to program an abstraction that generates a singleton class from an arbitrary class. Below, we show such a function which we add as a method to the class `Class`.

```

Class.AddMember(Singleton,
  lambda()
    if (this.Kappa?())
      let clonedClass = this.Clone()
      clonedClass.AddMember(uniqueInstance, nil)
      clonedClass.AddMember(New,
        lambda ()
          if (this.uniqueInstance == nil)
            let newInstance = super.New()
            this.uniqueInstance = newInstance
            newInstance
            this.uniqueInstance)
        clonedClass
      error("Only classes respond to Singleton"))

```

If we, for instance, want a singleton `Student` class, it can be made in the following way:

```
SingletonStudent = Student.Singleton().
```

When the `Singleton` message is received by the `Student` class, it is first checked that the receiver is a class. If so, the receiver is cloned, and the field `uniqueInstance` is added to the cloned class. In addition, the `New` method is redefined in the cloned class, hereby shadowing the `New` method in class `Class`. When `SingletonStudent` receives the `New` message, it returns the singleton instance, if this instance is already created. If the singleton instance has not yet been created, it is made by applying the `New` method above the redefined version of `New`, assigning it to `uniqueInstance`, and finally returning it.

The `Singleton` method above follows the standard approach [12] to making a singleton variant of a class. The (static) `Instance` method is, however, replaced by a redefinition of the `New` method. With this twist it becomes transparent from client classes that they interact with a singleton class.

Other program patterns can be developed as program abstractions as well. In [17] we have explored how the visitor pattern can be implemented as program abstractions. We used the F# programming language exploiting its combination of functional and object-oriented programming. A similar approach to implementing the visitor pattern using the Scala programming language [7] is described by Oliveira et al. [19]. In his recent thesis [16] Skeel Løkke explores how many of the original GOF patterns [12] can be implemented as program abstractions in the Scala programming language. Bosch [10] explores a layered object model which explicitly supports syntactic constructs and abstraction mechanisms for implementing program patterns in a purpose built language.



7 DISCUSSIONS AND CONCLUSIONS

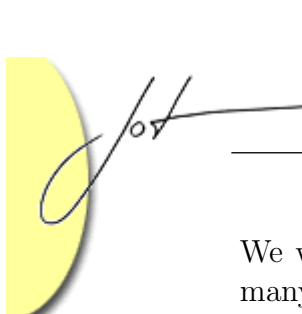
In this paper we have discussed computational *abstraction steps* as a way to create class abstractions from concrete objects, and from examples. We have described how computational abstraction steps can be seen as symmetric counterparts to concretisation steps such as object creation via class instantiation. To experiment with computational abstraction steps we have developed Abstraction Step Language (ASL), a small experimental language, and developed a prototype implementation.

We envision that computational abstraction steps may introduce a truly objects first programming approach, where we envision initiating a programming process by creating or capturing concrete values, objects, and actions as a first step. As the next step, some of these are lifted to a higher level by computational means. In the object-oriented paradigm the target of such steps is classes. We hypothesise that the proposed approach primarily will be beneficial to novice programmers, but in some specific application areas it is also expected that our approach will be a help to professional programmers. We have presented two such cases, one where a set of related classes and superclasses are created from data in comma-separated text files, and one where the singleton pattern is implemented as a programmed abstraction.

Furthermore, we envision that abstraction steps may also be useful if a large number of existing classes need to be refactored systematically. For instance, assume that we need to introduce a new abstract class which generalises a number of existing classes. Instead of refactoring the source program by using an integrated development environment (IDE) it may be a viable and attractive alternative to carry out the refactorings via application of dynamic programming operations on class objects. It may be possible to automate such refactorings by means of an iteration on a suitable list of class objects.

We have explored to which extent abstraction steps are supported by existing languages. Dynamic languages such as Self and JavaScript can to some extent simulate our notion of abstraction steps, and Python could support nearly all abstraction step mechanisms. However, with strongly typed languages the story is different. By using local type inference, anonymous classes and quite a bit of reflection and runtime code generation, a limited form of abstraction steps could be supported by C# 3.0. It will be interesting to explore C# 4.0 [1] and its extended support for dynamic programming in the near future.

It is relevant to clarify the relations between the work described in this paper and the field of meta programming and reflection. Meta programming is the kind of programming which works on (introspects and modifies) programs - the program itself or other programs. Meta programming calls for a particular data representation of programs. In the object-oriented paradigm, the data representation of programs is quite naturally made up by objects. A particular type of meta objects represents the classes in an object-oriented program. In our approach this kind of objects is elevated to the same status as the value of lambda expressions in a functional programming language. Informally, we use the term *first class classes* to emphasise this viewpoint.



We want the *first class class* values/objects to be readily available, in contrast to many mainstream systems of today, where the meta objects are very expensive to deal with. Thus, *first classes classes* and *first class functions* are special and central data types in our work, in the same way as numbers and strings usually are considered as central and special in many programming languages. Furthermore, a rich set of dynamic programming operations on objects and class objects is provided by ASL.

In the terminology of Segapeli and Cavarero [21], we are mainly concerned with level one and level two of generalisation. Clearly the introduction of first class classes and operations on these, combined with features of higher order functional programming, might facilitate new ways of approaching Segapeli and Cavarero's third level, which introduces meta classes or classes of classes. We expect to pursue this line of research in the near future.

This also leads us to the possibility of exploring programmed abstraction steps in other programming paradigms. A first and natural candidate is the functional paradigm, where function application represents a natural concretisation step, and where lambda expressions represent a static (syntactic) abstraction mechanism, but a dynamic counter part is missing. However, generalising an expression to a function at run-time is not directly as appealing as generalising an object to a class. Generalising a list of pairs of values to a function is, if not impossible, computationally intractable (see [5] for recent progress in this field).

Acknowledgement

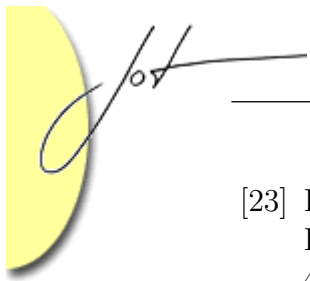
The authors would like to thank Simon Kongshøj for comments on earlier drafts of this paper.

REFERENCES

- [1] <http://code.msdn.microsoft.com/csharpfuture>.
- [2] <http://edu.netbeans.org/bluej/>.
- [3] https://developer.mozilla.org/en/about_javascript.
- [4] <http://www.devsource.com/c/a/languages/an-exercise-in-metaprogramming-with-ruby/1/>.
- [5] <http://www.news.cornell.edu/stories/april09/naturallaws.ws.html>.
- [6] <http://www.python.org/>.
- [7] <http://www.scala-lang.org/>.
- [8] The C# language specification version 3.0. Microsoft Corporation, 2007.



- [9] David J. Barnes and Michael Kölling. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, October 2002.
- [10] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.
- [11] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in self. *Lisp and Symbolic Computation: An International Journal*, 4(3):207–222, 1991.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996.
- [13] Irit Hadar and Uri Leron. How intuitive is object-oriented design? *Commun. ACM*, 51(5):41–46, 2008.
- [14] Richard Kelsey, William Clinger, and Jonathan Rees. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.
- [15] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.
- [16] Fredrik Skeel Løkke. *Scala and Design Patterns - exploring Language Expressivity*. PhD thesis, Aarhus University, Aarhus, Denmark, August 2009.
- [17] Kurt Nørmark, Bent Thomsen, and Lone Leth Thomsen. Mapping and visiting in functional and object-oriented programming. *Journal of Object Technology*, 7(7):75–107, 2008.
- [18] Kurt Nørmark, Lone Leth Thomsen, and Kristian Torp. *Reflections on the teaching of programming*, chapter Mini Project Programming Exams, pages 229–243. Springer Verlag, LNCS 4821, 2008.
- [19] Bruno C.d.S. Oliveira, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 439–456, New York, NY, USA, 2008. ACM.
- [20] Gary Pollice. Beyond an objects-first approach. IBM, 2005. <http://www.ibm.com/developerworks/rational/library/oct05/pollice/-index.html>.
- [21] S. Segapeli and A. Cavarero. Meta-induction: 3-level generalization from examples in object-oriented design. *Electrical and Computer Engineering, 1996. Canadian Conference on*, 1:298–301 vol.1, May 1996.
- [22] R. D. Tennent. *Principles of Programming Languages*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.



- [23] Bent Thomsen. *Reflections on the teaching of programming*, chapter Using On-Line Tutorials in Introductory IT Courses, pages 68–74. Springer Verlag, LNCS 4821, 2008.
- [24] David Ungar and Randall B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation: An International Journal*, 4(3):187 – 205, 1991.



A THE SCHEME-BASED ASL LANGUAGE

The abstraction step language ASL has been implemented in Scheme [14], on top of a simple meta circular interpreter. In this appendix we reproduce the example from Section 2 using the actual implementation of ASL.

```
; We start by making an object:
(define james (make-object 'firstName "James" 'lastName "Nielsen" 'age 25
                          'area "Math"))

; A FullName method is added to the object:
(send james 'AddMember 'FullName (function ()
  (string-append
    (get-field this 'firstName) " " (get-field this 'lastName))))

; We try out the FullName method on james:
(send james 'FullName)      ; => "James Nielsen"

; We derive a Student class from the object james.
(define Student (send james 'Kappa))

; We refurbish james as an instance of class Student:
(send james 'AsInstanceOfClass Student)

; We instantiate class Student, hereby defining anne:
(define anne (send Student 'New))
(send anne 'SetField 'firstName "Anne")
(send anne 'SetField 'age 20)

; We make oliver:
(define oliver (make-object 'firstName "Oliver" 'lastName "Persson" 'age 20
                            'area "Biology"))

; ... and refurbish it as a Student:
(send oliver 'AsInstanceOfClass Student)

; We test that oliver has the FullName methods (from class Student):
(send oliver 'FullName)    ; => "Oliver Persson"

; We add another method BornYear to class Student:
(send Student 'AddMember 'BornYear
  (function (thisYear)
    (- thisYear (get-field this 'age))))

; We send BornYear messages to all our sample objects:
(send (new Student) 'BornYear 2009)  ; => 1984
(send james 'BornYear 2009)          ; => 1984
(send oliver 'BornYear 2009)         ; => 1989
(send anne 'BornYear 2009)           ; => 1989

; A data field is added to class Student:
(send Student 'AddMember 'startOfStudy 2008)
```

```

; We make a new 'employee-like' object:
(define mike (make-object 'firstName "Mike" 'lastName "Madsen" 'salary
12500))

; We borrow the FullName method from class Student, and add it to mike:
(send mike 'AddMember 'FullName (get-field Student 'FullName))

(send mike 'FullName) ; => "Mike Madsen"

; The class Employe is derived from mike:
(define Employe (send mike 'Kappa))

; Via multiple generalization we generalize Student and Employe to class Person.
(define Person (generalize-multiple-classes! (list Student Employe)
'firstName 'lastName))

; A new data field is added to class Person:
(send Person 'AddMember 'sex 'Female)

; A sex change method is added to class Person:
(send Person 'AddMember
'SexChange!
(function ()
(if (eq? (get-field this 'sex) 'Male)
(set-field! this 'sex 'Female)
(set-field! this 'sex 'Male))))

; A Person instance is made:
(define axel (new Person 'firstName "Axel")) ; Female per default

; ... who is now sex-changed:
(send axel 'SexChange!) ; now Male
(send axel 'GetField 'sex) ; => Male

; A Singleton method is added to class Class:
(send Class 'AddMember 'Singleton
(function ()
(if (send this 'Kappa?)
(let ((cloned-class (send this 'Clone)))
(send cloned-class 'AddMember 'uniqueInstance 'nil)
(send cloned-class 'AddMember 'New
(make-function '()
'(if (eq? (send this 'GetField 'uniqueInstance) 'nil)
(let ((new-instance (send-super this 'New)))
(send this 'SetField 'uniqueInstance new-instance)
new-instance)
(send this 'GetField 'uniqueInstance)) '()))
cloned-class)
(error "Only classes respond to Singleton"))))

; A new class SingletonStudent is generated:
(define SingletonStudent (send Student 'Singleton))

```



ABOUT THE AUTHORS



Kurt Nørmark is an associate professor at the department of computer science at Aalborg University, Denmark. His research interests include programming languages and tools, including program documentation tools and the relationships between XML technology and programming technology. Contact him at normark@cs.aau.dk.



Bent Thomsen is an associate professor at the department of computer science at Aalborg University, Denmark. His research interests include object-oriented, functional and concurrent programming, real-time programming, future programming languages and programming technology. Contact him at bt@cs.aau.dk.



Lone Leth Thomsen is an associate professor at the department of computer science at Aalborg University, Denmark. Her research interests include object-oriented, functional and concurrent programming, web and BPM programming, future programming languages and programming technology. Contact her at lone@cs.aau.dk.