THE ESSENCE OF SUBCLASSING

Ole Lehrmann Madsen[1], Birger Møller-Pedersen[2]
[1]*The Alexandra Institute & Aarhus University*
[2]*Department of Informatics, University of Oslo, Oslo, Norway*
olm@alexandra.dk, birger@ifi.uio.no

The essence of subclassing is the ability to represent classification hierarchies reflecting domain concepts. With the right class mechanism there is no need for a separate type/subtype mechanism, and general classes may have behavior specifications so that subclassing also implies code reuse. Sometimes the application is so well defined (known) that developers may readily start out with classes that represent domain concepts. Singular objects have been around for some time, so a new kind of language with equal support for both objects and classes would simply ask for tool support for objects in the spirit of e.g. Self and support for classes as described above.

1. Introduction

The class/subclass mechanism as introduced in SIMULA [1] was originally intended for representing *classification hierarchies* and thus also as a mechanism for defining the type/subtype of variables, parameters and return types. An inherent property of this mechanism was that subclasses *'inherited' (reused)* the code of the superclass, i.e. the code of the methods of the superclass.

Modeling languages [2] maintain class/subclass as a mechanism for representing classification hierarchies. Most mainstream programming languages support class/subclass in the style of SIMULA, but in practice they are often used as an implementation mechanism for reuse resulting in messy and unstructured class libraries.

Starting with [3], discussing the pros and cons of prototypical languages, followed up by Self (http://www.selflanguage.org), there are now a number of prototypical languages where the class/subclass mechanism is abandoned, and there is now even a debate on what the concepts of class/subclass are good for. The most recent [4] argues that class/subclass is only for incremental program structuring, and even that classes are considered harmful.

2. What classes were meant for

SIMULA introduced class as a type, influenced by record class/subclass proposed by Hoare, where the intention was to classify data plus operations. The examples speak for themselves: `vehicle`, `car`, `bus`, `truck` where common properties are described in `vehicle` and program fragments may manipulate `vehicles` – being `cars`, `busses` and/or `trucks`.

In the same way as predefined types like integer and Boolean ensure that only valid operators are applied to values of these types, the class/subclass mechanism was introduced in order to type reference variables so that only valid operations are performed on objects.

Since Morrison [5] it has been generally accepted that a type is not just a **set** of values. A class represents a concept, it describes the ***intension*** of the concept in the form of attributes. The objects of the class represent the ***extension*** of the concept. A subclass extends the description of the intension and the extension of the subclass is a subset of the extension of the superclass.

3. What classes are often used for

Reuse of code has been put forward as the main advantage of object-orientation in contrast to the modeling capabilities of object-orientation. However, unrestricted use of class/subclass for reuse often leads to messy and badly structured class hierarchies where an instance of a subclass may not be substituted for an instance of its superclass. Subclass substitution was an essential property of SIMULA – and it is surprising that this had to be reinvented as the Liskov substitution principle. Using class/subclass for pure reuse should be avoided/prohibited.

Unfortunately, it is not possible for the compiler to check that a subclass is behavioral compatible with its superclass. Subtyping in terms of method signatures is easy but not sufficient, as real superclasses may have behaviour. Class invariants and pre- and post-conditions on methods [6] may help on this. However, at the end of the day, it depends on the discipline of the programmers – *to program is to* understand – it is not *to get away with it*.

Language designers considering class to be a reuse mechanism often add a separate type-system in order to make (static) type checking. Types/subtypes in this context is then often based upon sets of method signatures, although it does have to be like that.

4. A class is more than method signatures

Type systems that only rely on signatures will lack the opportunity to define general types that have behavior specified. A simple example is a general class `Shape` that defines a `center` point and a `move` method that simply moves the graphical figure by moving the center.

```
Shape: {
 centre: obj Point;
 final move(p: obj Point):{
   centre = p }
}
```

Code that relies on the property that graphical figures are moved this way will work for any special `Shape` object, i.e. objects of any subclass of `Shape` may take the place of a `Shape` object.

A related issue is how to support frameworks/libraries/APIs and generics without classes?

5. Reuse of code without subclassing

The above is an example of code reuse that comes with subclassing. It ensures that all objects of subclasses of `Figure` will behave the same way when moved. Not all languages support this kind of subclassing, some language allow any method to be redefined completely (apart from parameters) in subclasses, and some languages allow that some of the properties of the superclass do not apply to objects of subclasses (e.g. Eiffel and Grace).

In 1992, it was demonstrated that for code reuse, there would be no reason to (mis)use subclassing. The mechanism of part-objects [7] readily provides code reuse. Given a class `Addressable` with properties `street` and `number` and a method `print`, then any class of objects that should be addressable simply gets a part-object `addr` of type `Addressable`, and the `print` method is accessible via `addr.print()`. Note that this is not full-fledged delegation, but rather method forwarding. By providing renaming constructs it would even be possible to avoid the '`addr.`'.

```
Addressable: {
  street: obj String;
  no: obj Int;
  print: {
    inner;
    -- print street and no
    }
}
Person: {
  name: obj String;
  addr: obj Addressable{
    void print(){ print name }
  }
}
Company: {
  logo: obj String
  addr: obj Addressable{
    void print(){print logo }
  }
}
```

As demonstrated above, the method `print` may even be tailored to wherever there is an `Addressable` part-object, simply by defining the part object as a specialization of `Addressable`, and as this specialization is done in the scope of the class with the part-object, the attributes of this class are visible in the specializations.

The GO language (https://golang.org), is a widely used language without classes. It has struct types (with methods as functions with a struct reference as the first parameter), and it uses anonymous part-objects instead of subclassing. However, it is not possible to specialize part-objects to the structs they are in (as above), and they of course have the same problems with name conflicts as in multiple inheritance.

6. Typing/subtyping revisited

In [4] the benefits of purely prototype-based languages is partly taken from [3], however, they have not mentioned that [3] also mentions a major drawback: Even if you remove classes as types, you will have to have types for predefined things like integer, Booleans, etc.

Types/subtypes, parameterized types, co-/ contravariant types and static versus dynamic type checking has been an issue for many years. In [8] the relationship between subclassing and subtyping is discussed. It is argued that an approach with a combination of static and dynamic type checking gives a reasonable balance between static and dynamic type checking. It is also concluded that this approach makes it possible to base the type system on the class/subclass mechanism including covariance, which from a modeling point-of-view is preferable.

7. Classification of actions

It is common practice in object-oriented modelling and programming to represent (physical) objects such as vehicles, persons, and flight reservations as objects. The same is the case for values such as numbers, points, and vectors and although values are timeless concepts, they must be represented as objects or other data structures. In Beta, we distinguish between objects and values as described in [9] .

In addition to objects and values, a program describes a (possible parallel) sequence of actions being executed. Actions are thus also phenomena as objects and values that must be represented in the program execution. Procedures and process types are abstraction mechanisms that may describe a set of instances of action-sequences.

As first pointed of by Jean Vaucher ([10]), procedures may also be organized in a procedure/subprocedure hierarchy similar to a class/subclass hierarchy. In Beta, this is covered by the general notion of pattern/subpattern that may define both class pattern/subpatterns of objects (for parallel execution) and procedure pattern/subpatterns (for sequential execution).

Such patterns are useful for defining control abstractions, including iterators.

For concurrency abstractions, the use of specialized actions is especially useful. In Beta, it is possible to define a Monitor abstraction (originally due to Jean Vaucher) in the following way.

```
Monitor:
  { entry:
      { do M.wait; inner; M.signal }
    M: obj Semaphore
  }
Buffer: Monitor
  { put: entry{ … add to B … }
    get: entry{ … get from B … }
    B: obj Queue
  }
```

The methods `put` and `get` are submethods of `entry` and the actions of `entry` works as a wrapper for the actions in `put` and `get` (these actions are executed when executing `inner`). This implies that at most one of `put` and `get` are executed at the same time.

The use of method pattern/subpattern is further described in ([11] , [12], [13]).

8. The programming process

Black et al [4] describe "*the essence of inheritance is its ability to override a concrete entity, and thus effectively turn a constant into a parameter*". They also state:" *Introducing a new concept by means of a series of examples is a technique as old as pedagogy itself*", and they advocate inheritance as a technology to support writing a series of example programs.

It should be clear from the above, that we do not agree that the essence of inheritance is overriding - the essence of subclassing is the ability to represent classification hierarchies.

Regarding the programming style as a series of examples, we agree, but this does not imply that these examples should be related through inheritance. The programming process is exploratory: (1) In the initial phases you have to understand the individual objects and their behavior without necessarily understanding the relations between objects. (2) In subsequent phases where you have more knowledge, you may classify your objects into classes and subclasses – to form a hierarchy of concepts. (3) You then apply this knowledge in the further process to understand the relations between objects.

This is a simplified description of *the process of knowledge* [12]. In [14] it is further pointed out that languages with singular objects and/or prototype-based languages like Self (with cloning and delegation instead of subclassing) are useful for supporting (1), but it is a must to be able to re-organize objects into classes in order to obtain structure on your program.

Teaching students to use a programming style where you write some code, extend it by redefining methods, etc. may adversely affect their ability to write well-structured software systems.

There is of course a difference when using class-libraries / frameworks where you may not modify the source. But here you should use other techniques such as inheritance from part objects and/or wrappers.

9. Challenges

We do not see redefining the essence of subclassing as one of the major challenges in making new object-oriented languages. However, there are some challenges related to multiple classification, and the support for objects *and* classes.

Subclassing as described above can be used to represent tree-structured classification hierarchies. Many languages have support for so-called multiple inheritance, however, mainly only as a method for code reuse. In [12], we have discussed various challenges for further supporting the representation of classification hierarchies. This includes multiple classification – this is similar to multiple inheritance, but with an emphasis on modeling. An example is that people may be independently classified according to their nationality, profession, religion, etc. Multiple inheritance for this will lead to an exponential number of combinations. We are not aware of (mainstream) languages supporting multiple classification. In addition, objects may evolve over time and this may even imply change of class membership. Some languages – at a primitive level – support dynamic class membership, but we still have a challenge with representing dynamic classification in object-oriented languages.

A programming language should support singular objects as well as classes, and the type system should be based on classes. The programming process should support reorganization of code and objects into class hierarchies as you obtain knowledge about your domain. Mechanisms like cloning and delegation may still have a role, but delegation then understood literally as delegating responsibility. In [15] it is illustrated that in order to have a design pattern that both support state machines with composite states, and subclassing of state machines, contained states should be able to delegate event method calls to their enclosing state. Usual design patterns represent composite states by state subclasses of the enclosing state class. A delegation approach to composite states implies that subclassing may be used for specialization of state machines.

REFERENCES

1. Dahl, O.-J., B. Myhrhaug, and K. Nygaard, *SIMULA 67 Common Base Language (Editions 1968, 1970, 1972, 1984)*. 1968, Norwegian Computing Center: Oslo.
2. Madsen, O.L. and B. Møller-Pedersen. *A Unified Approach to Modeling and Programming*. in *MoDELS 2010*. 2010. Oslo: Springer.
3. Borning, A., *Classes versus Prorotypes in Object-Oriented Languages*, in *ACM/IEEE Fall Joint Computer Conference*. 1986: Dallas, Texas. p. 36-40.
4. Black, A.P., K.B. Bruce, and J. Noble, *The Essence of Inheritance*, in *A List of Successes That Can Change the World*. 2016, Springer International Publishing.
5. Morris, J.H., *Types are not Sets*, in *ACM Symposium on the Principles of Programming Languages (POPL)*. 1973, ACM. p. 120-124.
6. Hoare, C.A.R., *Proof of Correctness of Data Representations.* Acta Informatica, 1972. **1**.
7. Madsen, O.L. and B. Møller-Pedersen. *Part Objects and Their Location*. in *TOOLS'92: Technology of Object-Oriented Languages and Systems*. 1992. Dortmund: Prentice Hall.
8. Madsen, O.L., B. Magnusson, and B. Møller-Pedersen. *Strong Typing of Object-Oriented Languages Revisited*. in *Joint OOPSLA/ECOOP'90*. 1990. Ottawa, Canada: ACM Press
9. MacLennan, B.J., *Values and objects in programming languages.* ACM SIGPLAN Notices, 1982. **17**(12): p. 70-79.
10. Vaucher, J., *Prefixed Procedures: A Structuring Concept for Operations.* IN-FOR, 1975. **13**(3).
11. Kristensen, B.B., et al. *Classification of Actions or Inheritance Also for Methods*. in *ECOOP'87 – European Conference on Object-Oriented Programming*. 1987. Paris: Springer Verlag.
12. Madsen, O.L., B. Møller-Pedersen, and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*. 1993: Addison Wesley.
13. Madsen, O.L., *Building Safe Concurrency Abstractions* in *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, G. Agha, et al., Editors. 2014, Springer Berlin Heidelberg.
14. Madsen, O.L., *Open Issues in Object-Oriented Programming.* Software Practice & Experience, 1995. **25**(S4).
15. Møller-Pedersen, B. and R.K. Runde, *State Pattern supporting both composite States and extension/specialization of State Machines*, in *PLoP 2016: PATTERN LANGUAGES OF PROGRAMS CONFERENCE*. 2016: University of Illinois at Urbana-Champaign.