ilips

hierarchy
hijerarhija

ELLIPSE-CIRCLE DILEMMA AND INVERSE INHERITANCE

Kazimir Majorinc
Faculty of Natural Sciences and Mathematics, University of Zagreb
Marulićev trg 19/III., 10 000 Zagreb, Croatia
kmajor@public.srce.hr

Abstract: *We consider problems related to inheritance in object-oriented languages, on the example called ellipse-circle dilemma. We criticise the existing solutions and propose a new kind of inheritance, called here inverse inheritance.*

Keywords: inverse inheritance, Liskov substitution principle, ellipse-circle dilemma

1. Introduction

The generalisation, also known as *is-A* relation, is usually considered as one of the three basic properties of the so called object-oriented (OO) systems (languages, design, etc.) An OO system supports *generalisation* if it provides language constructs for the definition of the subclass of the existing class. Each object, instance of a subclass is automatically an instance of the superclass (Rumbaugh *et al.*, 1991). Each subclass of the given class automatically contains all data- and function- members declared in a superclass. Such a mechanism is called *inheritance*. One may think that if we use class *C* for a description of a set of the entities, that the subset of that set may be correctly described as a subclass of class *C* without further considerations. A well known counterexample is the *ellipse-circle (EC-) dilemma* (Cline and Lomow, 1995). Suppose that we need to define the class of central ellipses and the class of central circles, and a few simple operations on them. The central ellipse is completely determined by a pair of halfaxes *a*, *b* and a circle is a special case of the ellipse such that *r=a=b*. The problem is how to declare this two classes: should one of them be derived from the other or should they be independent? According to the nature of the generalisation, the circle *is-An* ellipse, perhaps the most natural code that implements these two classes will be something like this C++ code:

```
class Ellipse
{ public:
  float a, b;
  float Area(){ return 3.14*a*b; }
  void  SetXYAxes( float x0, float y0 ){ a=x0; b=y0; return; }
  void  Double() { a*=2; b*=2; return; }
};
class Circle: public Ellipse {};
```

However, it is an unexpected fact that code

```
Circle c;
c.SetXYAxes( 1, 2 );
```

is legal. It is obvious that *SetXYAxes* will assign illegal state to the object of class *Circle*. We say that the state of the object is *illegal* if it does not describe any possible states of the intended model. One of the goals of OO programming - greater data safety - is strongly corrupted if such statements are legal. As we will see, this problem is harder to solve than it looks (Shang, 1996; Martin and Ottinger, 1997).

2. Solutions with run-time checking

The first idea may be that it is enough to redefine function *SetXYAxes* in class *Circle*. For example,

```
class Circle: public Ellipse
{  void SetXYAxes( float x0, float y0 ){ exit(1); return; }
};
```

The disadvantages of such a disabling of the inherited function are (1) *c.SetXYAxes*(1,2) is still a legal code, and it will be successfully compiled although it will produce a run-time error if executed. Such a level of safety is less than that typical to the procedural statically typed languages. (2) If we want to disable all functions that may assign an illegal state to the object, we must know all functions that are inherited from superclasses, maybe hundreds of them, although we may need only a few. It may require a lot of work and changes in the future maintenance. The procedure must be repeated every time when the base class is exchanged with, for example, a new and refined version from some library, etc.

Two OO languages, Eiffel and Sather support *design by contract,* a very powerful feature that allows us to declare an expression which objects of the given class must satisfy in all *stable* times. This expression will be checked during the run-time and will produce a run-time error when the objects come into an illegal state. An example is in Eiffel.

```
class CIRCLE
inherit ELLIPSE
invariant
   equalaxes: a=b
end
```

All actions that assign an illegal state to the object of class *CIRCLE* will produce a run-time error, including those made by any function inherited from superclasses. However, there are two obstacles to this method: (1) statement *c.SetXYAxes*(1,2) is still legal and will be compiled and (2) a characteristic function of a set of legal states may be too complicated, not known or even undecidable, although the implementation of the subclass may be natural and useful even if we do not know how to recognise all elements. Actually, it is possible to define contracts only in special cases.

2. The cause of the problem

Let us make a distinction between states and legal states of a class. We will say that *S'* is a *subclass* of a *S* if *S'* is derived by language rules from *S*. Let us suppose that there are no additional data members in *S'*; it is enough for our purpose. Every state of *S'* is automatically a state of *S*. Let us consider the case in which a class describes the *intended model:* a set of some real-world or mathematical entities. We will say that a state is *legal* if it describes a possible entity in the intended model. Class *S'* is a *subset* of *S* if the set of all legal states of *S'* is a subset of the set of all legal states of *S*. Encapsulation, assertions and design by contract have the role to prevent an accidental assign of an illegal state to the object. Let us distinguish a few kinds of function-members. A *selector* is a function-member that returns value of a data-member of the object that called it or some function of those values, and does not change the state of the object, i.e. value of its data-members. Example of the selector is a function-member *Area.* A

modifier on the class is every function-member that changes the state of the caller object. An example is a function-member *SetXYAxes*. It is possible to distinguish two kinds of modifiers: some modifiers change the state of the object without using the previous state of the object. Such functions may be considered as a generalisation of the *assignment* operator and we will
1) call them *assignors*. The function *SetXYAxes* is an assignor. The second kind of modifiers are
2) *mutators* or *real modifiers*, i.e. those that use the previous state of the object to calculate a new state. The function *Double* is a mutator. Now, let S' be both a subclass and a subset of S. This type of subclassing is called *specialisation via constraints*. In further text, for simplicity, we will use the same labels for classes and sets of legal values. Let selector s, assignor a and mutator m be defined on class S. Their declarations are s: $S \times P_1 \rightarrow Q_1$, a: $P_2 \rightarrow S \times Q_2$, and m: $S \times P_3 \rightarrow S \times Q_3$. With P_1, P_2, P_3, Q_1, Q_2, Q_3 we will denote the classes whose values are used in calculating functions s, a, m. If these functions are inherited from S to S' then they should satisfy the declarations (D) s: $S' \times P_1 \rightarrow Q_1$, a: $P_2 \rightarrow S' \times Q_2$, and m: $S' \times P_3 \rightarrow S' \times Q_3$. It is easy to see that if S' is a real subset of S, then s will satisfy the declaration on S', $s(S' \times P_1) \subseteq Q_1$ but it is not necessary that modifiers will, i.e. that $a(P_2) \subseteq S' \times Q_2$, $m(S' \times P_3) \subseteq S' \times Q_3$. In a general case, this will not be true (as for *SetXYAxes*) but in some special cases, like for *Double*, it will be true. Nevertheless, a and m are defined in S' and they can assign an illegal state to the object. The rule that all kinds of function-members should be inherited in the same way is too simple.

3. Liskov substitution principle

A widely accepted principle for the solution of the EC-dilemma and similar problems is a *Liskov substitution principle (LSP)* that claims that S' should be declared as a subclass of S only if S' is a *subtype* of S. According to the usually accepted definition (Liskov, 1988), class S' is a subtype of class S if for each object o_1 of class S there is an object o_2 of class S' such that for all programs P defined in terms of S the behaviour of P is unchanged when o_1 is substituted for o_2. In our terms, S' is a subtype of S if all modifiers satisfy conditions (D) from the previous chapter. In our example, *Circle* should not be derived from *Ellipse*. Indeed, that rule excludes the possibility that some inherited functions may assign an illegal state to the object. However, there are a few obstacles to LSP. (1) The selector is well defined on every subset of a class, like function *Area*. LSP does not support the re-use of such functions if a subset is not a subtype. (2) The subtype relation is neither a frequent nor a natural relation between the classes, and it is doubtful what the benefits of implementation are of the whole mechanism of inheritance only for this relation. (3) OO languages do not support a subtype relation, its implementation depends on the discipline of the programmers. (4) The subtype relation is very *fragile*, i.e. every addition of the function-member to the superclass may destroy it. For example, addition of a very natural assignor *RestoreObjectFromFile*(char * *filename*) to any superclass will break every subtype relation with any of the derived subclasses. Note that subset relation is not so fragile: it does not depend on the functions defined in a superclass, but only on the set of legal values. Once established, addition of a function can not change it.

void Ellipse :: SetXYAxes (PairOfFloats)
void Circle :: SetXYAxes (PairOfEqualFloats)

4. Restriction of arguments

Some languages, like Eiffel, Sather or Transframe implement various methods of a restriction of the type of the arguments in inherited function-members. Idea is that a: $P_2 \rightarrow S \times Q_2$, and m: $S \times P_3 \rightarrow S \times Q_3$ can be inherited in S' if P_2 and P_3 are restricted to small enough classes P_2' and P_3'. It is true, for $P_2' \subseteq a^{-1}(S' \times Q_2)$ and for $P_3' \subseteq \{ p \in P_3 \mid S' \times \{ p \} \in m^{-1}(S' \times Q_3)\}$.

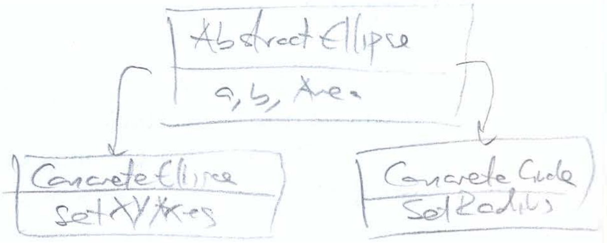
too complicated, not known or undecidable.

However, there is a same obstacle (2) as for the solution with run-time checking. We do not deny that such restrictions may be useful for other purposes.

5. Derivation from abstract classes

Let us define an *abstract* class as a class that has no instances. Any non-abstract class is *a concrete* class. (Meyer, 1996) suggested a rule that soon became popular (Martin and Ottinger, 1997; Grosberg, 1997) inside OO community: a concrete class must be derived from an abstract class and never from a concrete one. If we follow this rule, we must split every non-leaf class into two parts: a concrete and an abstract one. An abstract part should be derived from the abstract part of a superclass. A concrete part should be derived from the abstract part. All subclasses should be derived from the abstract part. Function-members that are harmful for subclasses should be placed in an abstract part, and those that are dangerous in a concrete part. In our example, it is:

```
class AbstractEllipse
{ public:
  float a, b;
  float Area(){ return 3.14*a*b; }
  void  Double() { a*=2; b*=2; return; }
};
class ConcreteEllipse : public AbstractEllipse
{ void  SetXYAxes( float x0, float y0 ){ a=x0; b=y0; return; }
};
class ConcreteCircle: public AbstractEllipse {};
```



Really, *c.SetXYAxes*(1,2) is now an illegal code. Again, there are three obstacles: (1) the rule is not radical enough. The leaf classes must be split into two parts as well, simple because we must expect that there will be a need for even smaller subclasses. (2) The rule is too radical: it is in direct contradiction with one of the basic ideas of OO: encapsulation of data and functions. An abstract class is the opposite of that goal. (3) It is complicated in practice. It may be expected that it will not be accepted in practical programming. On the other side, this method reveals and makes available just what we need: different rules of inheritance for different function-members.

6. Inverse inheritance

Our idea is to change the rules of inheritance to obtain this feature: if the function-member of class *S* called with legal states of the arguments cannot assign an illegal state to the caller-object of class *S*, it cannot assign an illegal value to the object of class *S'* in which function is implicitly and automatically inherited. On the other side, it should be allowed for a programmer to inhere any function he wants explicitly. If we take a look at various function-members, we will see that (1) selectors like *Area* could always be inherited in the subclass. (2) Some modifiers, like *Double,* may be inherited into subclass, but it will depend on the definition of the modifier. It is unexpected that (3) every assignor a': $P_2 \rightarrow S' \times Q$ may be inherited inversely, from a subclass to a superclass. Obviously a: $P_2 \rightarrow S \times Q$ is well defined. For example, a function-member *SetRadius*(float *r*) can be defined in *Circle* and inherited to *Ellipse*. Also, (4) in some cases, a modifiers can be inherited from a subclass to superclass. Again, it depends on the definition of the modifier. An example is the same function-member *Double*. We propose the next three rules for inheritance: (R1) selectors should be inherited from a superclass to a subclass automatically; (R2) assignors should be inherited from a subclass to a superclass automatically; (R3) other function-members, i.e. real modifiers can be inherited from any class that contains the same data members, in any position in the class hierarchy, but such an inheritance must be explicitly declared. For example, the code in proposed pseudo C++ may look as

```
class Ellipse
{ float a,b;
  void SetXYAxes(float x0, float y0);
  float Area();
  void Double(){ a*=2; b*=2; return; }
  Circle::Triple();
};
class Circle: public Ellipse
{ void SetRadius(float r){ a=b=r; }
  void Triple(){ a*=3; b*=3; }
  Ellipse::Double();
}
```

After this declaration assignor *SetRadius*, selector *Area*, and modifiers *Double* and *Triple* should be available in both classes *Ellipse* and *Circle*. Advantages over traditional inheritance rules implemented according to LSP are: (1) it is possible to describe every subset as a subclass, including all subtypes. (2) Subclass relation is not fragile any more. The addition of any function-member to any class is no danger: if the function is an assignor or selector its definition has a sense in every class where such function is automatically inherited. If the function is a mutator, then it is not inherited in any class, except explicitly. (3) Rule (R2) and especially (R3) are powerful and wider re-use of code is possible. In comparison with the method of derivation from an abstract class, the method described here has the following advantages: (1) it is simpler for practical programming (2) it should be built in into the language, i.e. it should not be possible to ignore the security mechanism, and (3) same like advantage over LSP. We will name the whole system described here *inverse inheritance*.

7. Emulation of inverse inheritance with multiple inheritance

In the language that supports multiple inheritance it is possible to implement inverse inheritance with using abstract classes. For simplicity, let us suppose that we want to declare classes C_1, ..., C_n where C_{i+1} is the subclass defined by constraint on C_i, for i=1, ..., n-1. All such classes contain the same data members, and these data members should be encapsulated in the abstract class *Ad*. For every i=1,...,n we should declare two abstract classes As_i and Aa_i. These classes should contain declarations of the selectors and assignors that we want to be defined in C_i. Classes As_1 and Aa_n must be derived from *Ad*. Class As_{i+1} must be derived from As_i, i=1, ..., n. Contrary, Aa_i must be derived from Aa_{i+1}. Every modifier m_j must be encapsulated in its own abstract class, for example Am_j, derived from *Ad*. Every concrete class C_i should be derived from abstract classes As_i, Aa_i and Am_j for every modifier m_j we want be available for class C_i. For example, this code is compiled and checked with Borland C++ compiler.

|Adata

```
class AData{ public: float a,b; };
class AGetEllipse: virtual public AData
{ public: float GetArea(){ return a*b*3.14; } };
class AGetCircle: public AGetEllipse
{ public: float GetRadius(){ return a; } };
```

```
class ASetCircle: virtual public AData
{ public: void SetRadius( float r ){ a=b=r; return; } };
class ASetEllipse: public ASetCircle
{ public: void SetXYAxes( float a0, float b0 )
                              { a=a0; b=b0;  return; };
};
class Ellipse: public ASetEllipse, public AGetEllipse {};
class Circle:  public ASetCircle, public AGetCircle {};
```

After that, *GetArea* is available in *Ellipse* and *Circle*, *GetRadius* only in *Circle*, *SetRadius* in *Ellipse* and *Circle* and *SetXYAxes* only in *Ellipse*. Every function and data member is declared and defined exactly once.

8. Conclusion

EC-dilemma shows some unexpected side effects of the conception of inheritance, one of the main ideas of the OO paradigm. At least, there is no consensus in the OO community how to implement is-A relation. Until the solution to this problem is found, in our opinion, there is no possibility to write a quality non-trivial software using inheritance. On our opinion, the rules of inheritance are far too simple. They may be significantly improved with the proposal made here. However, the whole concept of the inheritance, like two other basic OO concepts: encapsulation and polymorphism, is still questionable.

9. Acknowledgements

Thanks to Marshall Cline, Patrick Doyle (author of the term *assignor*), Joachim Durcholtz, Sanja Kušec, Nick Leaton, Michael Lee Finney, Robert Martin, Scott McKay, James Rogers (author of the term *inverse inheritance*), Juergen Schlegenmilch, and Tim Ottinger, in alphabetical order.

10. References

1. Cline, M. and Lomow, G. (1995), *C++ FAQs: Frequently Asked Questions,* Addison-Wesley, Reading.
2. Grosberg, J. (1997), "Design guideliness for Is-A hierarchies", Dr Dobbs J., 266, pp. 36-42.
3. Liskov, B. (1988), "Data abstraction and hierarchy", SIGPLAN Notices 23.
4. Martin, R. (1996), "The Liskov substitution principle", C++ Report, Vol 9(2).
5. Martin, R. (1997), "The principles of OOD", http://www.oma.com.
6. Martin, R. and Ottinger, T. (1997), "Inheritance 'Is-A' problem", Object Magazine Online, May 1997, http://www.objectmagazine.com.
7. Mattos, N. and DeMichiel. L. (1994), "Recent design trade-offs in SQL3", ACM SIGMOD Record, Vol 23, No 4.
8. Meyers, S. (1996), *More Effective C++,* Addison-Wesley, Reading.
9. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W (1991), *Object-Oriented Modeling and Design,* Prentice-Hall, New Yersey.
10. Shang, D. (1996), "Is a cow an animal?", Object Currents, Vol. 1, SIGS Publications, http://www.objectmagazine.com.
11. Zdonik, S. B. and Maier, D. (1989), *Readings in Object-Oriented Databases,* Morgan Kaufmann