INHERITANCE AND CLASS STRUCTURE

Ruslan Batdalov, Russia, Moscow, linnando@acm.org

Introduction

Being a set of objects [1], a class has at the same time its internal structure allowing it to be constructively defined and implemented in real computational systems. Our distant goal is to formally describe how such a structure can be constructed, which means are necessary for construction of the structure of all classes including the basic ones and what their possibilities and limitations are. This paper addresses the issues related to inheritance.

Inheritance had been generally considered a way of conceptual specialization, but later this point of view was criticized as too simple and insufficient. Taivalsaari mentions a number of situations where using inheritance has other purposes [5]. Cook et al. propose completely different typed model of inheritance not requiring an inherited type to be a subtype [2]. So the notion of inheritance was reduced to a mechanism of incremental definition of new classes on the base of the existing ones [1, 5]. Nevertheless strongly-typed object-oriented languages preserve the subset notion of inheritance and allow subclass instantiation (i.e. assignment an object value of a child class to an object variable of the parent class) [2, 3]. So at least in this sense any object of a child class is an object of the parent class too, and the child class is a subset of the parent class. Our first task is to reconcile this statement with the critics mentioned above. Probably the structure of the parent class should be more complex than it seems to be and include all its subclasses too.

1. Aggregate structure

The most common or even the only way to derive a child class is the supplementing it with new members, which it would have in addition to the ones of the parent class [5]. While a class itself can be seen as a Cartesian product of its members, the subset interpretation requires more sophisticated view. Indeed, a set of triples <integer, integer, integer> is not a subset of a set of pairs <integer, integer>. In order to support subclass instantiation we should assume that a class as a set is a union of Cartesian products that include all enumerated members and possibly something else. Following traditional structural view we can define aggregate component of a class structure as a relation <name1->class1, name2->class2, …, nameN->classN>[1]. But unlike in the usual structural interpretation this does not mean that the class having this aggregate structure is a Cartesian product <class1, class2, …, classN>. Now it is a union of all Cartesian products compatible with the given aggregate structure. The simplest concept of compatibility is that a Cartesian product is compatible with an aggregate structure if its first N members are <class1, class2, …, classN>. We will extend this concept later.

2. Restrictions on class and its members

If there is a predicate on a class[2], the corresponding subset of objects can be found. Using the analogy between child classes and subsets we can consider any such subset a child class[3]. Obviously, having imposed a restriction on a class member in the same manner we will get another subset of the class. Considering any subset of a class its subclass we can reword this statement: subclassing of any member of a class results in a subclass of this class.

Object-oriented programming languages often lack the possibility to derive a child class using an explicit restriction [3]. Nevertheless this feature seems to be useful in building taxonomies based on classes. For example, knowledge that the integer numbers are real numbers with zero fractional part can help a compiler to use particular optimization techniques. The possibility to declare set-

---

[1] *nameX* can be either identifier or index (as in arrays).

[2] A predicate can be implemented by either a class member or a function defined elsewhere. Concrete means by which a predicate can be defined lie beyond the boundaries of the paper.

[3] Strictly speaking the deduction above does not require this interpretation, but nothing prevents this view either. Further the terms ‘subclass’ and ‘child class’ will be used as synonyms.

subset relations between user-defined classes may allow the compiler to use similar techniques for them too.

An important special case of restricting a set of objects is its reduction to a trivial set consisting of an only element. Unlike the general case this one is common for programming languages: the restriction is imposed implicitly on declaring a free object or a class member as a constant[4]. It is important in implementation but from the theoretical point of view it is just a restriction of a class to a subset.

3. Polymorphism

Polymorphism is the ability of different classes to respond to the same message and each implement the method appropriately [1]. Its common implementation in object-oriented programming languages is polymorphic redefinition of class members which is sometimes named a reason why child classes are not subsets of the parent one, because object-oriented systems do not typically provide any guarantees in that this technique is used for conceptual specialization [5].

With polymorphism a constant class member (most often a method) can be redefined in a child class to have another value. A variant of this technique is using abstract members, i.e. declaring a method or another constant member without giving it a value. Less common form of polymorphism is changing the class of a class member (constant or variable). Some programming languages (C++, Java) prohibit redefinition of all members except the ones explicitly declared as redefinable ('virtual'). Other ones allow polymorphic redefinition of any members.

On the first glance polymorphic redefinition looks really inconsistent with the subset interpretation of inheritance. Nevertheless the subclass instantiation is still allowed without any explicit or implicit conversion.

Apparently this means that the constant restriction on the value of a class member allowing polymorphic redefinition is not inherited. This 'polymorphic constantness' means only that the value of the member cannot change during its lifetime[5], but does not prohibit this member from having different values for different instances of the class. In implementation it requires to store the value of the member either together with other members or in a 'virtual method table' unlike of other constant members.

Changing the class of a member follows the same scheme. A member allowed to change its class in child classes has in fact another class instead of what is said in its definition[6]. It is the union of all classes that can be used in the polymorphic redefinition. If the programming language does not have any means to limit possible classes, such a member will be of the topmost class in the class hierarchy.

So polymorphism does not violate the subset interpretation of inheritance independently of which particular values and classes are substituted instead of other ones. Nevertheless it is not to diminish the value of compatibility rules, necessity of which was noted by Wegner [6]. Such rules are of great importance for checking semantic consistency between classes and subclasses.

4. Cancellation

Some programming languages allow cancellation or deletion of class members (i.e. absence of a class member from a subclass). This deletion is inconsistent with the definition of compatibility between Cartesian products and aggregate structures above, so we need to reconcile them.

Often the deletion is appearing only. For example, if the class of real numbers were a subclass of the class of complex numbers, omitting an imaginary part would seem to be its deletion. But in fact we have only restricted it to be always zero. We have to provide such default values in every

---

[4] Methods declared inside a class and free functions are usually always constant in the sense that they always point to the same code while some languages allow them to be variable.

[5] Restrictions of this kind require further investigation and are not discussed in this paper.

[6] Of course here we speak about changing a class to another one when the latter is not a subset of the former.

case of deletion in order to support subclass instantiation. So in this case cancellation is nothing but restriction (or maybe polymorphic redefinition) described above.

But Taivalsaari also mentions another use of cancellation – hiding class members that are not needed [5]. An example is deriving class stack from deque if the latter is already implemented and the former is not [4]. In this case we delete some class members without providing default values. If the parent class does not have default values for the members either, subclass instantiation can raise an error later. As it was mentioned in the cited works, inheritance is used for generalization instead of specialization here.

Of course deriving a more general class from a more special is very convenient when needed. Nevertheless we cannot agree that it is impossible to rely on any set-subset relation between a class and its child class. Probably the object-oriented programming languages lack the possibility to incrementally derive a superclass of the given one instead of a subclass. If we could declare stack as a superclass of deque and then delete members that are not needed, there would be no danger of run-time error after subclass instantiation. Also stack variables would allow instantiation with deque objects as it would be if deque were derived from stack as a subclass in the general manner.

It is worth noting that declaring a class as a superclass of another one should allow not only cancellation but also adding new class members or lessening restrictions. For example complex numbers can be introduced as a superclass of previously implemented real numbers what correlate with the real cognitive process. As explained above, this only requires us to define a default value for the newly added member – imaginary part. But conceptually this means that the aggregate structure of a class can be supplemented with new members by declarations of other classes. We hope that it is just a thing to bear in mind, not to raise problems.

5. Multiple inheritance

Multiple inheritance is an arguable issue in object-oriented programming. Some languages do allow it (for instance C++, Eiffel), some prohibit (Smalltalk) and some replace with other techniques (such as interfaces or mixins).

As a mere set of objects a class with two or more parent classes must be a subset of their intersection. The worst thing that can happen here is that the intersection is empty. For example it is possible if the same constant member of two classes has different values in them (unless it allows polymorphic redefinition).

The impact of multiple inheritance on the concept of aggregate structure is more significant. Members with the same names can have different orders in parent classes and do not have to be the first members of a Cartesian product. So we need to add one more component to the class structure: a correspondence between class members' names and Cartesian product members' indexes. Of course it can vary for different Cartesian products. As a result a Cartesian product is compatible with an aggregate structure <name1->class1, name2->class2, …, nameN->classN> if there exists a one-to-one correspondence <name1<->index1, name2<->index2, …, nameN<->indexN> and for each X the indexX^{th} member of the product is classX. In the case of multiple inheritance the aggregate structure of the child is the union of the parents' aggregate structures with addition of new members defined in the child class itself. If we encounter common names in parents' aggregate structures while uniting them, the corresponding class in the union will be the intersection of corresponding classes in parents' structures.

In the same manner deriving a superclass from several children should be possible. Resulting class should be a superset of the union of the given ones, its aggregate structure should be the intersection of the ones of the given classes and the classes for common names should be the unions of the ones in children's structures. The same rules are applicable to deriving a class from both a superclass and a subclass (i.e., 'inserting' it into the class hierarchy).

It seems that these rules solve the problem of multiple inheritance in relation to class structure. However we have not addressed the question of whether additional restrictions on a multiple inheritance system are necessary for its real application.

6. Conclusion

Summarizing all said above we can conclude that two components of class structure are necessary for defining inheritance relations between classes (for strongly-typed object-oriented languages):

- An aggregate structure that is a relation <name1->class1, name2->class2, …, nameN->classN>.
- A predicate defined in terms of the aggregate structure and restricting the set of objects representing the class.

The set of objects representing the class is the subset of the union of Cartesian products compatible with the aggregate structure satisfying the predicate. A Cartesian product is compatible with an aggregate structure <name1->class1, name2->class2, …, nameN->classN> if there exists a one-to-one correspondence <name1<->index1, name2<->index2, …, nameN<->indexN> and for each X the indexX^{th} member of the product is classX.

Deriving a child class from a parent one can be done by means of supplementing the aggregate structure of the latter with new members, tightening its predicate and/or subclassing any of its members. Multiple inheritance implies uniting aggregate structures of the parents and &-ing their predicates. Deriving a parent class from one or more children or inserting a new class between a parent and a child can be done in a similar manner.

While declaring a class member as constant generally imposes implicit restriction on its class, it is not true when the member allows polymorphic redefinition. When the member allows polymorphic changing of its class not requiring it to be a subclassing, the corresponding class in the aggregate structure is wider than the one stated in the class declaration.

With all these assumptions inheritance does not violate a set-subset relation between a parent and a child even in non-trivial cases. Of course this cannot prohibit using inheritance for other purposes (not only for conceptual specialization), but the relation will remain nonetheless. As a result other uses of inheritance can potentially lead to unexpected results when either the parent class is too wide or the child class is too narrow (if it matters at all).

The following ways to derive a new class from an existing one mentioned above are not common for the existing object-oriented programming languages:

- Imposing an explicit restriction on the class by requiring it to satisfy a predicate.
- Declaring the derived class to be a superclass of the existing one instead of a subclass.

7. Further work

The following steps to the goal to describe the structure of classes seem to be (they have mostly been mentioned in the course of the paper):

- To determine how we can define predicates and algebraic structures on classes.
- To find additional components of the structure necessary for interpretation of a class as an automaton (i.e. to take arguments and to return output). It is particularly needed for interpretation of functions as objects.
- To formally describe the components of the class structure using lambda-calculus.
- To formulate possible restrictions on objects run-time behavior.
- To find the conditions of semantic consistency between classes and subclasses with polymorphic members' redefinition.
- To understand whether the multiple inheritance system requires introducing any restrictions to be implementable.

References

1. D. Armstrong. The Quarks of Object-Oriented Development. In Communications of the ACM, Vol. 49, Issue 2, February 2006.
2. W.R. Cook, W.L. Hill, P.S. Canning. Inheritance is not subtyping. In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1989.
3. B. Stroustrup. C++. Addison-Wesley, 1987.

4. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In Conference proceedings on Object-oriented programming systems, languages and applications, 1986.
5. Taivalsaari. On the Notion of Inheritance. In ACM Computing Surveys, Vol. 28, No. 3, September 1996.
6. P. Wegner. Concepts and Paradigms of Object-Oriented Programming. In ACM SIGPLAN OOPS Messenger, Vol. 1, Issue 1, August 1990.

Batdalov, Ruslan. Inheritance and Class Structure. P.92-96.
In: Объектные системы - 2010 : материалы I Международной научно-практической конференции : Россия, Ростов-на-Дону, 10-12 мая 2010 г. = Object Systems - 2010 : First International Scientific-Practical Conference Proceedings : Russia, Rostov-on-Don, 10-12 May 2010 / edited by Pavel P. Oleynik. - Ростов-на-Дону, 2010. ISSN 2309-8856. ISBN 978-5-9902226-2-5.