An anomaly of subtype relations at component refinement, and a generative solution in C++

Zoltán Porkoláb and István Zólyomi

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{gsd, scamel}@elte.hu

Abstract. Separation of concerns and collaboration based design is a good design choice: it results an easily maintainable and readable code. After separating orthogonal functionalities we assemble the required concerns as needed. However in real life, components could be used only after appropriate refinement steps, thus orthogonal concerns form independent specialization hierarchies. Such hierarchies provide individual subtype relations. The specific solution for a particular task can be finally produced by composing a set of classes from these refinements. However, a subtype anomaly occurs between collaborating groups having different number of participating classes from different refinement stages. In this article we walk around this anomaly we called chevron-shape inheritance and present a framework to handle collaborating groups of classes using template metaprogramming based on standard C++ features.

1 Introduction

The creation of large scale software systems is still a critical challange of software engineering. Several design principles exist to keep the complexity of large systems manageable. Different methodologies are used to divide the problem into smaller orthogonal parts that can be planned, implemented and tested separately with moderate complexity. In a fortunate case such parts are already exist in some foundation library, otherwise they can be produced by reasonable efforts. This *separation of concerns* is widely discussed in [17] and [20]. In object-oriented systems these concerns are mostly implemented in separate classes.

Having premanufactured components we have several methodologies to assemble a full system from the required code parts. This so-called *collaboration based design* is supported by aspect oriented programming [15], composition filters [11], subject oriented programming [19] [21] and HyperJ [18]. Besides, the assembly can be naturally expressed using multiple inheritance by deriving from all required components in languages supporting this feature such as C++. This *mixin-based* technique is highly attractive for implementing collaboration-based design [6]. Whichever approach we choose, the basic idea is to create a union of the interfaces of the collaborating classes. These classes represent orthogonal concepts thus the result is a disjunction of their functionalities.

However, in real life it is hard to find a component that represents the required concept *exactly*. In most cases we have to customize the components to fit the needs of the current task. Specializations for every separate concern are made independently which leads to separated specialization hierarchies, each representing refinement steps of an individual concept. The specific solution for a particular task can be finally produced by assembling specialized concepts from appropriate levels of different hierarchies. In object-oriented languages we mostly represent our concepts as classes. Specializations are regurarly expressed using inheritance, hence we gain a subtype relationship between the refined and the original component.

The problem appears when we try to refer to a subset of supertypes of the collaborating classes. It is a desired feature because a client code should be separated from the knowledge of the exact type of the (refined) components. But the collaboration of original components are not a base class of the collaboration of refined components. Because automatic conversion is out of order, objects of the collaboration of original components cannot be used in place of objects of collaboration of refined components which is against the Liskov Substitution Principle [22]. Similarly, clients are unable to utilize dynamic binding calling functions of derived objects in a type safe way.

2 Importance of the problem

Programmers may argue that such side effects may ever appear in practice. In this section we intend to convince the reader showing real-life examples.

We start with one from the C++ Standard Library of C++: in figure 1 you can see the stream class hierarchy of the standard library[1].

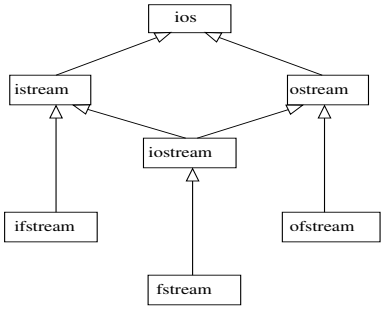

Fig. 1. I/O library according to the C++ standard

Classes `istream` and `ostream` are representing input and output streams as orthogonal concerns. (There is a common base class `ios` for both classes holding

[1] We omit the fact that all the followng classes are *templates* by the standard, because this does not affect our problem.

some general stream functionality.) Class `iostream` is created using multiple inheritance unifies input and output functionalities representing streams that can be both read and written[2]. The library contains two refinements of both input and output stream concepts. Streams opened over certain physical devices belong to classes `ifstream` or `ofstream` as refinements of `istream` and `ostream` respectively[3]. These specializations are implemented using inheritance. Class `fstream` (and `stringstream` also) inherits from `iostream` and represents file streams for both input and output operations.

Surprisingly, this construction causes some unexpected results. `fstream` is clearly a subtype of both `ifstream` and `ofstream`. The inheritance hiearchy described above does not express this, hence there is no conversion from `fstream` to neither `istream` nor `ostream`. Clients handling input files are not able to use objects from `fstream` as an instance of `ifstream`, they are enforced to use `istream` as a more general interface losing file specific information. After taking a look at classes `iostream` and `istream` this fact may be an astonishing fact.

The other example is from the programming language Eiffel [10]. The kernel library of Eiffel contains several abstract classes like `NUMERIC` for arithmetics, `COMPARABLE` for sorting, `HASHABLE` for associative containers, etc. These classes are practical to have because in Eiffel we can require a template parameter to be a subclass of such an "interface". These classes can be combined as needed using multiple inheritance, hence we can derive a `NUMERIC_COMPARABLE_HASHABLE` or a `NUMERIC_COMPARABLE` interface directly from the bases. Again, the problem appears when we try to use an object of the first class with a generic algorithm requiring the latter type: no subtype relation is realized, we have to resolve it by hand creating funtions for conversion.

3 The chevron-shape anomaly

In this section we formulate the problem showing a general description and suggest a name for the anomaly. It appears in strongly typed object-oriented languages which base their subtype relation on inheritance; consequently it appears in all widely used object-oriented languages, such as Java, C++, C#, Eiffel, Object Pascal, etc. The problem is closely related to class refinement using multiple inheritance[4].

Assume we have a set of independent base classes implementing orthogonal concerns. These classes are to be refined stepwise, thus each concept forms a separate inheritance hierarchy. The solution for a specific user requirement can be constructed as a group of refined concerns. In the same case, we should be able

[2] This results in a known anomaly called *diamond-shape inheritance*. In this case it is resolved using *virtual inheritance* in classes `istream` and `ostream`.

[3] Similar specializations exist for streams stored in a memory buffer (e.g. `istringstream` and `ostringstream`).

[4] Note that some languages (e.g. Java) do not support multiple inheritance directly, but are able to simulate it (e.g. using interfaces). The problem exists in these cases, too.

to use any subset of classes from these hierarchies as interfaces to the previously constructed group. Therefore subtype relation should stand between any of these groups irrespectively of the number and refinement level of participant concern classes. The subtype relation should be closed under union (disjunction), but this is not fulfilled in object-oriented languages. Thus we have to decide: if we derive the refined collaboration from the original collaboration class we lose the subtype relationship with the refined bases; otherwise (deriving from the refined bases) we lose the subtype relationship with the original collaboration. In most design cases the latter situation is preferred. In figure 2 the general structure of the anomaly can be seen according to the two mentioned cases respectively. In the picture missing subtype relations are marked with dashed lines.

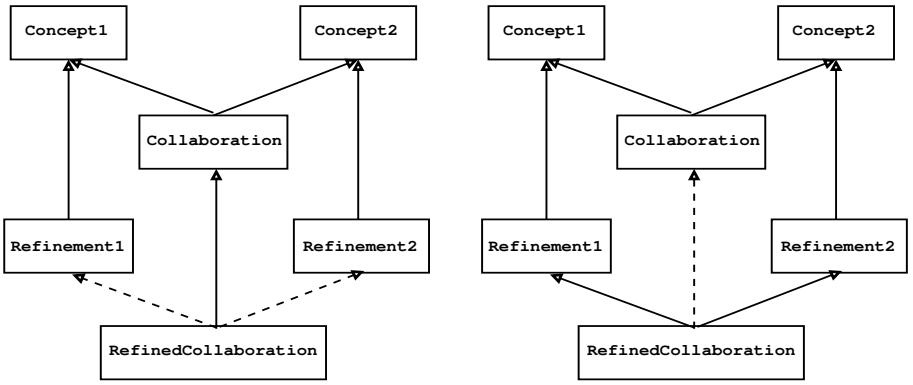

Fig. 2. Chevron-shape inheritance

We gave the name *chevron-shape inheritance* to this anomaly[5]. It is easy to understand our choice taking a look at figure 2.

In addition, having several concept hierarchies we should be able to express subtype relationship between collaborating groups having different number of classes of different refinement stages.

4 CSet

Besides its object-oriented tools the C++ language also has a rich feature set for supporting generative programming with templates. Theoretically template metaprogramming in C++ is a Turing complete language itself, therefore any algorithm can be expressed as a metaprogram[6] (see [7]). This "language" is "executed" in compilation time: the result is a C++ program which is still about to be checked by the language strong type system.

[5] With this name we also intended to refer to *diamond-shape inheritance.*

[6] Practically compilers have limitations in resources (e.g. a maximal depth of recursion during template instantiation).

Template metaprogramming features discussed above make us able to solve the chevron-shape anomaly. To achieve this goal we have to simulate a subtype relationship between adequate sets of collaborating classes: based on the possibilities of template metaprogramming we implement conversions between the sets[7]. Presenting the technical implementation details is out of the scope of this paper. The main issue in `CSet` is to build the needed class hierarchy, templated conversion operators and smart pointers automatically in compilation time.

5 Summary and related works

The subtyping mechanism of current object oriented languages is not flexible enough to express required subtype relationships arising at implementation of collaboration based designs. We described an anomaly called chevron-shape inheritance which arises assembling sets of collaborating concerns created in stepwise refinement of concepts. We introduced a framework called CSet based on C++ template metaprogramming to extend the possibilities of subtyping mechanism between sets of collaborations. CSets make the subtype relation disjunctive with respect to multiple inheritance. It supports coercion polymorphism between appropriate collaborating groups or inclusion polymorphism allowing dynamic binding of methods with smart pointers. The framework is strictly based on standard C++ features, therefore neither language extensions nor additional tools are required.

Another candidate for solution can be the signature facility of C++ from Gerald Baumgartner [4]. Signatures provide a facility similar to interfaces in Java, but in a non-intrusive way: if a class intends to implement a signature, it does not have to define it explicitly to do so. Signatures are non-standard language extensions and are implemented only in g++ compilers, thus their usability is strictly limited.

As an alternative solution Structural subtyping [13] provides an excellent possibility for solution: languages supporting this feature do not suffer from our anomaly. Unfortunately no widely used object-oriented language provides structural subtyping.

References

1. István Zólyomi, Zoltán Porkoláb, Tamás Kozsik: An extension to the subtype relationship in C++. GPCE 2003, LNCS 2830, pp. 209 - 227, 2003 (Springer-Verlag Berlin Heidelberg)
2. Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
3. David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2003)

[7] we call these sets `CSets` where `C` can be pronounced as any of class, concern, collaboration, chevron, etc as conceptually needed.

4. Gerald Baumgartner, Vincent F. Russo: Implementing Signatures for C++ ACM Transactions on Programming Languages and Systems (TOPLAS) Vol. 19 Issue 1. 1997. pp. 153-187.
5. Todd Veldhuizen: Using C++ Template Metaprograms. C++ Report vol. 7, no. 4, May 1995, pp. 36-43.
6. Yannis Smaragdakis, Don Batory: Mixin-Based Programming in C++. In proceedings of Net.Object Days 2000 pp. 464-478
7. Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
8. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
9. Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)
10. Bertrand Meyer: Eiffel: The Language. Prentice Hall (1991)
11. Lodewijk Bergmans, Mehmet Aksit: Composing Crosscutting Concerns Using Composition Filters. Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.
12. Kim B. Bruce: Foundations of Object-Oriented Languages. The MIT Press, Cambridge, Massachusetts (2002)
13. Luca Cardelli: Structural Subtyping and the Notion of Power Type. Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, January 1988. pp. 70-79.
14. Erik Ernst: Family Polymorphism. in Proceedings ECOOP 2001, Budapest, Hungary, Springer-Verlag LNCS 2072, 2001 pp. 303–326,
15. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin: Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241, June 1997.
16. Ulrich W. Eisenecker, Frank Blinn and Krzysztof Czarnecki: A Solution to the Constructor-Problem of Mixin-Based Programming in C++. Presented at the GCSE2000 Workshop on C++ Template Programming.
17. Harold Ossher, Peri Tarr: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. IBM Research Report 21452, April, 1999. IBM T.J. Watson Research Center. http://www.research.ibm.com/hyperspace/Papers/tr21452.ps
18. Harold Ossher, Peri Tarr: Hiper/J. Multidemensional Separation of Concerns for Java. International Conference on Software Engineering. ACM pp. 734-737. 2001
19. William Harrison, Harold Ossher: Subject-oriented programming: a critique of pure objects Proceedings of 8th OOPSLA 1993, Washington D.C., USA. pp. 411-428. 1993
20. Don Bathory, Jia Liu, Jacob Neal Sarvela: Refinements and multi-dimensional separation of concerns Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering. helsinki, Finland, 2003.
21. Subject Oriented Programming. http://www.research.ibm.com/sop
22. Barbara Liskov: Data Abstraction and Hierarchy SIGPLAN Notices. 23(5), May 1988
23. Jonathan E. Shopiro: An Example of Multiple Inheritance in C++: a Model of the Iostream Library. ACM SIGPLAN Notices, December, 1989